**Структура и организация проекта**

app Маршрутизатор приложений

pages Маршрутизатор страниц

public Обслуживаемые статические активы

src Необязательная исходная папка приложения

layout.js .jsx .tsx Планировка

page.js .jsx .tsx Страница

loading.js .jsx .tsx Загрузка пользовательского интерфейса

not-found.js .jsx .tsx Не найден пользовательский интерфейс

error.js .jsx .tsx Пользовательский интерфейс с ошибкой

global-error .js .jsx .tsx Глобальная ошибка пользовательского интерфейса

route .js .ts Конечная точка API

template .js .jsx .tsx Повторно отрисованный макет

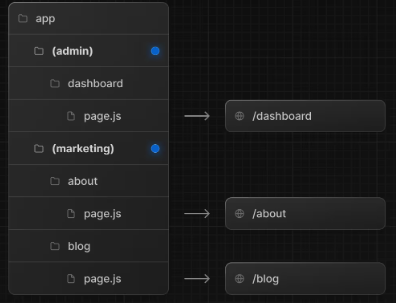
default .js .jsx .tsx Резервная страница параллельного маршрута

[**Личные папки**](https://nextjs.org/docs/app/getting-started/project-structure#private-folders)

Личные папки можно создавать, добавляя к папке символ подчеркивания: **\_folderName**

Это означает, что папка является частной деталью реализации и не должна учитываться системой маршрутизации, тем самым **исключая папку и все её подпапки** из маршрутизации.

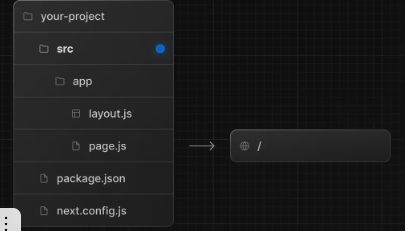
[**Группы маршрутов**](https://nextjs.org/docs/app/getting-started/project-structure#route-groups)

Группы маршрутов можно создать, заключив папку в круглые скобки: (folderName)

Это означает, что папка предназначена для организационных целей и **не должна включаться** в URL-адрес маршрута.

**src каталог**

Next.js поддерживает хранение кода приложения (включая app) в дополнительном src каталоге. Это позволяет отделить код приложения от файлов конфигурации проекта, которые в основном находятся в корневом каталоге проекта.



**Хранить файлы проекта за пределами app**

Эта стратегия позволяет хранить весь код приложения в общих папках в корневом каталоге вашего проекта и использовать каталог app исключительно для маршрутизации.

**Полезно знать**: в приведённых ниже примерах мы используем папки components и lib в качестве общих заполнителей. Их названия не имеют особого значения для фреймворка, и в ваших проектах могут использоваться другие папки, например ui, utils, hooks, styles и т. д.

**Создание страницы**

Страница — это пользовательский интерфейс, который отображается на определённом маршруте. Чтобы создать страницу, добавьте page.tsx файл в каталог app и экспортируйте компонент React по умолчанию. Например, чтобы создать главную страницу (/):

export default function Page() {

return <h1>Hello Next.js!</h1>

}

**Создание макета**

Макет — это пользовательский интерфейс, который используется совместно на нескольких страницах. При навигации макеты сохраняют состояние, остаются интерактивными и не перерисовываются.

Вы можете определить макет, по умолчанию экспортируя компонент React из layout файла. Компонент должен принимать параметр children, который может быть страницей или другим макетом.

Например, чтобы создать макет, который принимает вашу главную страницу в качестве дочерней, добавьте файл layout в каталог app:

export default function DashboardLayout({

children,

}: {

children: React.ReactNode

}) {

return (

<html lang="en">

<body>

{/\* Layout UI \*/}

{/\* Place children where you want to render a page or nested layout \*/}

<main>{children}</main>

</body>

</html>

)

}

**Создание вложенного маршрута**

Вложенный маршрут — это маршрут, состоящий из нескольких сегментов URL. Например, маршрут /blog/[slug] состоит из трёх сегментов:

/ (Корневой сегмент)

blog (Сегмент)

[slug] (Сегмент листа)

В Next.js:

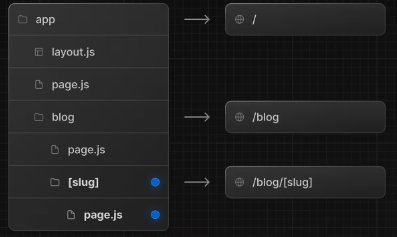
Папки используются для определения сегментов маршрута, которые сопоставляются с сегментами URL.

Файлы (например, page и layout) используются для создания пользовательского интерфейса, который отображается для сегмента.

Чтобы создать вложенные маршруты, вы можете вкладывать папки друг в друга. Например, чтобы добавить маршрут для /blog, создайте папку с названием blog в каталоге app. Затем, чтобы сделать /blog общедоступным, добавьте файл page:



Вы можете продолжать создавать вложенные папки для вложенных маршрутов. Например, чтобы создать маршрут для конкретного поста в блоге, создайте новую папку [slug] или [id] внутри blog и добавьте файл page:



**Вложенные макеты**

По умолчанию макеты в иерархии папок также являются вложенными, то есть они оборачивают дочерние макеты с помощью children prop. Вы можете создавать вложенные макеты, добавляя layout внутри определенных сегментов маршрута (папок).

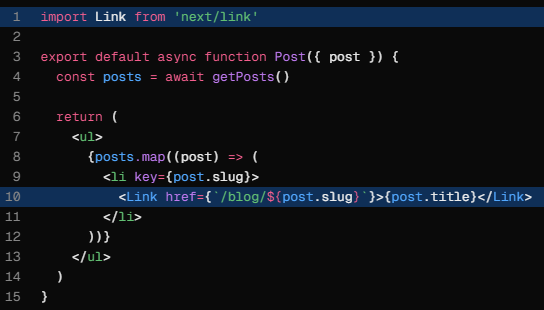
Например, чтобы создать макет для /blog маршрута, добавьте новый layout файл в blog папку.



**Ссылки между страницами**

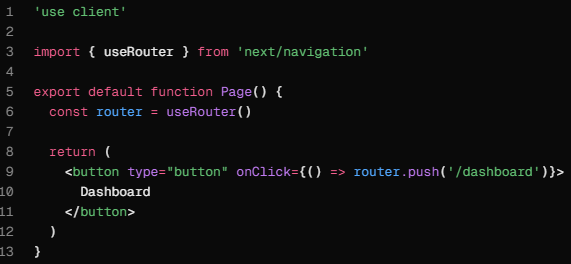
Вы можете использовать <Link>компонент для перехода между маршрутами. <Link> — это встроенный компонент Next.js, который расширяет HTML-тег <a> для обеспечения предварительной загрузки и навигации на стороне клиента.

Например, чтобы создать список постов в блоге, импортируйте <Link> из next/link и передайте href в качестве свойства компоненту:



<Link> Это основной и рекомендуемый способ навигации между маршрутами в приложении Next.js. Однако вы также можете использовать useRouterхук для более продвинутой навигации.

**useRouter Хук** позволяет вам программно изменять маршруты внутри клиентских компонентов.



**useRouter()**

**router.push(href: string, { scroll: boolean }):** Выполните навигацию на стороне клиента по указанному маршруту. Добавляет новую запись в стек истории браузера.

**router.replace(href: string, { scroll: boolean }):** Выполните навигацию на стороне клиента по указанному маршруту, не добавляя новую запись в стек истории браузера.

**router.refresh():** Обновить текущий маршрут. Выполнить новый запрос к серверу, повторно запросить данные и повторно отобразить серверные компоненты. Клиент объединит обновлённую полезную нагрузку серверного компонента React без потери данных, не затронутых клиентским React (например, useState) или состоянием браузера (например, положением прокрутки).

**router.prefetch(href: string):** Предварительная выборка предоставленного маршрута для более быстрых переходов на стороне клиента.

**router.back():** Вернитесь к предыдущему маршруту в стеке истории браузера.

**router.forward():** Перейдите на следующую страницу в стеке истории браузера.

**Как оптимизировать изображения и шрифты**

Вы можете хранить статические файлы, такие как изображения и шрифты, в папке public в корневом каталоге. Файлы внутри public можно использовать в коде, начиная с базового URL (/).

**Оптимизация изображений**

Компонент Next.js <Image> расширяет элемент HTML<img>, предоставляя:

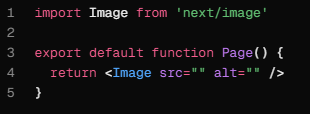
-Оптимизация размера: автоматическая загрузка изображений нужного размера для каждого устройства с использованием современных форматов изображений, таких как WebP.

-Визуальная стабильность: автоматическое предотвращение сдвига макета при загрузке изображений.

-Более быстрая загрузка страниц: загрузка изображений только при их появлении в области просмотра с помощью встроенной в браузер функции отложенной загрузки с дополнительными размытыми заполнителями.

-Гибкость в работе с ресурсами: изменение размера изображений по требованию, даже изображений, хранящихся на удаленных серверах.

Чтобы начать использовать <Image>, импортируйте его из next/image и визуализируйте в своем компоненте.



**Оптимизация шрифтов**

Модуль next/font автоматически оптимизирует ваши шрифты и удаляет запросы к внешним сетям для повышения конфиденциальности и производительности.

Он включает встроенный автоматический самохостинг для любого файла шрифта. Это означает, что вы можете оптимально загружать веб-шрифты без изменения макета.

Чтобы начать использовать next/font, импортируйте его из next/font/local или next/font/google, вызовите его как функцию с соответствующими параметрами и задайте className для элемента, к которому вы хотите применить шрифт. Например:



**Шрифты Google**

Вы можете автоматически размещать любой шрифт Google на своём сервере. Шрифты включаются в развёртывание и обслуживаются с того же домена, что и ваше развёртывание, то есть браузер не отправляет запросы в Google, когда пользователь заходит на ваш сайт.

**Локальные шрифты**

Чтобы использовать локальный шрифт, импортируйте его из next/font/local и укажите src локального файла шрифта в public папке.



**Если вы хотите использовать несколько файлов для одного семейства шрифтов, src может быть массивом:**

const roboto = localFont({

src: [

{

path: './Roboto-Regular.woff2',

weight: '400',

style: 'normal',

},

{

path: './Roboto-Italic.woff2',

weight: '400',

style: 'italic',

},

….

**Как использовать CSS в вашем приложении**

**Модули CSS**

Модули CSS локально ограничивают CSS, генерируя уникальные имена классов. Это позволяет использовать один и тот же класс в разных файлах, не беспокоясь о конфликтах.

Чтобы начать использовать CSS-модули, создайте новый файл с расширением .module.css и импортируйте его в любой компонент в каталоге app:

**.blog** {

padding: 24px;

}

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

export default function Page({ children }: { children: React.ReactNode }) {

return <main className={**styles.blog**}>{children}</main>

}

**Глобальный CSS**

Вы можете использовать глобальный CSS для применения стилей в вашем приложении.

Чтобы использовать глобальные стили, создайте новый CSS-файл, например app/global.css:

Импортируйте файл в корневую разметку (app/layout.js), чтобы применить стили к каждому маршруту в вашем приложении:

import './global.css'

export default function RootLayout({

children,

}: {

children: React.ReactNode

}) {

return (

<html lang="en">

<body>{children}</body>

</html>

)

}

**В Next.js серверные и клиентские компоненты** появились как часть нового подхода к рендерингу в версии 13 и выше (апдейт с App Router). Они позволяют определять, где компоненты приложения будут выполняться — на сервере или на клиенте. Эта модель поддерживает гибкость в управлении производительностью и SEO.

Серверные компоненты обрабатываются и рендерятся на сервере, а затем результат передается уже в виде готового HTML клиенту. Они не содержат JavaScript-кода на стороне клиента.

**Ключевые особенности**:

•Выполняются на сервере. Можно работать с базой данных, API и безопасно управлять серверными процессами.

•Не добавляют никакого JavaScript в браузер, поэтому это подходит для SEO и ускоряет загрузку страницы.

•Идеально подходят для статического или предварительно отрендеренного контента.

**Пример**:

// Серверный компонент (по умолчанию)

// app/dashboard/page.tsx

export default function Dashboard() {

  return (

    <div>

      <h1>Серверный компонент</h1>

      <p>Этот HTML рендерится на сервере.</p>

    </div>

  );

}

Итог: HTML-код генерируется на сервере и отправляется клиенту, без лишнего JS

**Клиентские компоненты выполняются в браузере**. Они полностью поддерживают интерактивное поведение и используют JavaScript на стороне клиента.

**Ключевые особенности**:

•Выполняются на клиенте, что позволяет управлять состояниями (useState, useEffect) и добавлять интерактивность.

•Поддерживают обработку событий (например, клики, формы).

•Требуют большего объема ресурсов, так как JavaScript-код загружается и запускается на клиенте.

Чтобы определить компонент как клиентский, необходимо использовать директиву "use client".

Пример

// Клиентский компонент

"use client";

import { useState } from "react";

export default function Counter() {

const [count, setCount] = useState(0);

return (

<div>

<p>Счетчик: {count}</p>

<button onClick={() => setCount(count + 1)}>Увеличить</button>

</div>

);

}

Итог: Этот компонент требует выполнения JavaScript на клиентской стороне

**Вы можете комбинировать серверные и клиентские компоненты**. Например, серверный компонент может импортировать клиентский компонент для обработки интерактивных элементов.

Пример

// app/dashboard/page.tsx (Серверный компонент)

import Counter from "./Counter";

export default function Dashboard() {

  return (

    <div>

      <h1>Серверный компонент</h1>

**<Counter />**

    </div>

  );

}

// app/dashboard/Counter.tsx (Клиентский компонент)

"use client";

import { useState } from "react";

export default function Counter() {

  const [count, setCount] = useState(0);

  return (

    <div>

      <p>Счетчик: {count}</p>

      <button onClick={() => setCount(count + 1)}>Увеличить</button>

    </div>

  );

}

Dashboard рендерится полностью на сервере.

•Counter изначально грузится как HTML, но затем становится интерактивным на клиенте

**Когда использовать серверные компоненты?**

•Когда нет интерактивности, а нужно сосредоточиться на скорости загрузки и SEO.

•Если требуется рендерить данные с API, базы данных или других серверных источников.

•Если нужно уменьшить JavaScript-бандл в браузере

**Когда использовать клиентские компоненты?**

•Для интерактивного UI (формы, модальные окна, слайдеры, графики, счетчики и т. д.).

•Когда нужно управлять состоянием компонентов через useState и useReducer.

•Если вы используете эффекты через useEffect (например, для работы с локальным хранилищем).

**Серверные Компоненты**

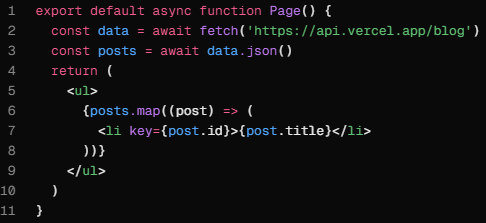
Вы можете извлекать данные из серверных компонентов с помощью:

-fetchAPI

-ORM или база данных

**С помощью fetch API**

Чтобы получить данные с помощью API fetch , превратите свой компонент в асинхронную функцию и дождитесь вызова fetch . Например:



**Клиентские компоненты**

Существует два способа извлечения данных из клиентских компонентов, используя:

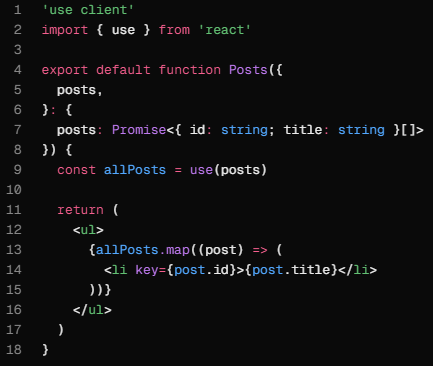
-use Крючок React's

-Общественная библиотека, такая React Query

**С помощью use крючка**

Вы можете использовать useхук React для потоковой передачи данных с сервера на клиент. Начните с получения данных в компоненте сервера и передайте обещание в качестве свойства компоненту клиента:

Затем в клиентском компоненте используйте хук use для чтения промиса:



В приведенном выше примере вам нужно обернуть компонент <Posts /> в <Suspense>границу. Это означает, что резервный вариант будет отображаться во время выполнения обещания. Узнайте больше о потоковой передаче.

**Потоковое вещание**

При использовании async/await в серверных компонентах Next.js будет использовать динамический рендеринг. Это означает, что данные будут извлекаться и отображаться на сервере для каждого запроса пользователя. Если запросы данных будут выполняться медленно, весь маршрут будет заблокирован для отображения.

Чтобы сократить время первоначальной загрузки и улучшить взаимодействие с пользователем, можно использовать потоковую передачу, чтобы разбивать HTML-код страницы на более мелкие фрагменты и постепенно отправлять эти фрагменты с сервера клиенту.

Есть два способа реализовать потоковую передачу в вашем приложении:

1. с loading.js файлом

2. с помощью <Suspense> компонента React

**С loading.js**

Вы можете создать файл loading.js в той же папке, что и ваша страница, чтобы транслировать **всю страницу** во время загрузки данных. Например, чтобы транслировать app/blog/page.js, добавьте файл в папку app/blog.

export default function Loading() {

return <div>Loading...</div>

}

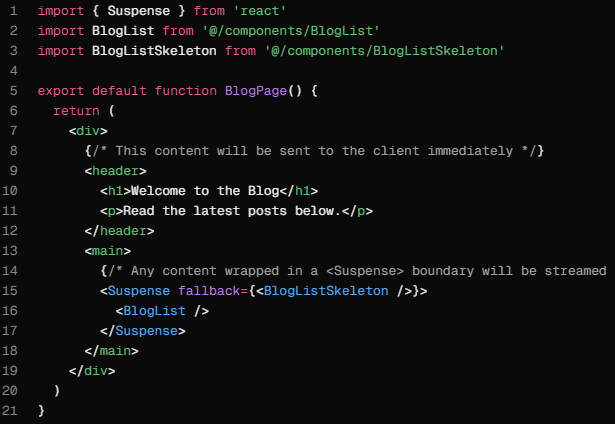
При переходе по ссылке пользователь сразу увидит макет и состояние загрузки во время рендеринга страницы. После завершения рендеринга новый контент будет автоматически заменен.

За кулисами loading.js будет вложен в layout.js и автоматически заключит файл page.js и все дочерние элементы в <Suspense>-оболочку.

Этот подход хорошо работает для сегментов маршрута (макетов и страниц), но для более детальной потоковой передачи можно использовать <Suspense>.

**С <Suspense>**

<Suspense> позволяет более точно определять, какие части страницы нужно транслировать. Например, вы можете сразу отобразить любой контент страницы, выходящий за пределы <Suspense> границы, и транслировать список постов в блоге внутри границы.



**Как обновить данные**

**Создание Серверных функций**

Серверный метод можно определить с помощью директивы use server . Вы можете поместить директиву в начало асинхронной функции, чтобы пометить её как серверный метод, или в начало отдельного файла, чтобы пометить все экспортируемые из этого файла методы. В большинстве случаев мы рекомендуем использовать отдельный файл.

'use server'

export async function createPost(formData: FormData) {}

export async function deletePost(formData: FormData) {}

**Серверные Компоненты**

Функции сервера можно встраивать в компоненты сервера, добавив директиву "use server" в начало тела функции:

export default function Page() {

// Server Action

async function createPost() {

'use server'

// Update data

// ...

}

return <></>

}

**Клиентские компоненты**

Невозможно определить серверные функции в клиентских компонентах. Однако вы можете вызывать их в клиентских компонентах, импортируя из файла, в верхней части которого есть директива "use server":

'use client'

import { createPost } from '@/app/actions'

export function Button() {

return <button formAction={**createPost**}>Create</button>

}

**Вызов серверных функций**

Есть два основных способа, которыми вы можете вызвать серверную функцию:

1.Формы в серверных и Клиентских компонентах

2.Обработчики событий в клиентских компонентах

**Формы**

React расширяет элемент HTML <form> таким образом, чтобы можно было вызывать серверную функцию с помощью свойства HTML action

При вызове в форме функция автоматически получает объект FormData . Вы можете извлечь данные с помощью встроенных FormData методов:

import { createPost } from '@/app/actions'

export function Form() {

return (

<form action={**createPost**}>

<input type="text" name="title" />

<input type="text" name="content" />

<button type="submit">Create</button>

</form>

)

}

'use server'

export async function **createPost(formData: FormData)** {

const title = formData.get('title')

const content = formData.get('content')

// Update data

// Revalidate cache

}

**Обработчики событий**

Вы можете вызвать серверную функцию в клиентском компоненте с помощью обработчиков событий, таких как onClick.

'use client'

import { incrementLike } from './actions'

import { useState } from 'react'

export default function LikeButton({ initialLikes }: { initialLikes: number }) {

const [likes, setLikes] = useState(initialLikes)

return (

<>

<p>Total Likes: {likes}</p>

<button

onClick={async () => {

const updatedLikes = await incrementLike()

setLikes(updatedLikes)

}}

>

Like

</button>

</>

)

}

**Отображение состояния ожидания**

При выполнении серверной функции вы можете отобразить индикатор загрузки с помощью хука useActionState React. Этот хук возвращает pending логическое значение:

'use client'

import { useActionState } from 'react'

import { createPost } from '@/app/actions'

import { LoadingSpinner } from '@/app/ui/loading-spinner'

export function Button() {

**const [state, action, pending] = useActionState(createPost, false)**

return (

<button onClick={async () => action()}>

{pending ? <LoadingSpinner /> : 'Create Post'}

</button>

)

}

**Повторная проверка кэша**

После выполнения обновления вы можете повторно проверить кэш Next.js и отобразить обновлённые данные, вызвав revalidatePath или revalidateTag в серверной функции:

'use server'

import { revalidatePath } from 'next/cache'

export async function createPost(formData: FormData) {

// Update data

// ...

revalidatePath('/posts')

}

**Перенаправление**

Возможно, вы захотите перенаправить пользователя на другую страницу после выполнения обновления. Это можно сделать, вызвав redirect в функции сервера:

'use server'

import { redirect } from 'next/navigation'

export async function createPost(formData: FormData) {

// Update data

// ...

redirect('/posts')

}

**Как обрабатывать ошибки (подробно в документации)**

Ошибки можно разделить на две категории: ожидаемые ошибки и необработанные исключения.

[**Обработка ожидаемых ошибок**](https://nextjs.org/docs/app/getting-started/error-handling#handling-expected-errors)

Ожидаемые ошибки — это ошибки, которые могут возникать при нормальной работе приложения, например, при [проверке формы на стороне сервера](https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions-and-mutations#server-side-form-validation) или неудачных запросах. Такие ошибки следует обрабатывать явно и возвращать клиенту.

[**Действия сервера**](https://nextjs.org/docs/app/getting-started/error-handling#server-actions)

Вы можете использовать хук [useActionState](https://react.dev/reference/react/useActionState" \t "_blank) для управления состоянием [серверных функций](https://react.dev/reference/rsc/server-functions) и обработки ожидаемых ошибок. Не используйте блоки try/catch для ожидаемых ошибок. Вместо этого вы можете моделировать ожидаемые ошибки как возвращаемые значения, а не как генерируемые исключения.

**Not found**

Вы можете вызвать функцию [notFound](https://nextjs.org/docs/app/api-reference/functions/not-found) в сегменте маршрута и использовать файл [not-found.js](https://nextjs.org/docs/app/api-reference/file-conventions/not-found) для отображения интерфейса 404.

export default function NotFound() {

return <div>404 - Page Not Found</div>

}

[**Обработка неперехваченных исключений**](https://nextjs.org/docs/app/getting-started/error-handling#handling-uncaught-exceptions)

Необработанные исключения — это непредвиденные ошибки, которые указывают на сбои или проблемы, которые не должны возникать при нормальной работе приложения. Их следует обрабатывать путём создания ошибок, которые затем будут перехватываться границами ошибок.

[**Границы вложенных ошибок**](https://nextjs.org/docs/app/getting-started/error-handling#nested-error-boundaries)

Next.js использует границы ошибок для обработки необработанных исключений. Границы ошибок перехватывают ошибки в дочерних компонентах и отображают резервный пользовательский интерфейс вместо дерева компонентов, в котором произошла ошибка.

Создайте границу ошибки, добавив файл **[error.js](https://nextjs.org/docs/app/api-reference/file-conventions/error)** в сегмент маршрута и экспортировав компонент React: