

- [Введение](#)
- [Основные функции, предоставляемые Next.js](#)
- [Next.js vs Gatsby vs create-react-app](#)
- [Как установить Next.js](#)
- [Просмотр источника, чтобы подтвердить работу SSR](#)
- [Пакеты приложений](#)
- [Что это за иконка справа внизу?](#)
- [Установка React Developer Tools](#)
- [Другие методы отладки, которые вы можете использовать](#)
- [Добавление второй страницы на сайт](#)
- [Связывание двух страниц](#)
- [Динамический контент с роутером](#)
- [Предзагрузка](#)
- [Использование роутера для обнаружения активной ссылки](#)
- [Использование next/router](#)
- [Подача данных в компоненты с помощью getInitialProps](#)
- [CSS](#)
- [Заполнение тега head пользовательскими тегами](#)
- [Добавление компонента-оболочки](#)
- [API-маршруты](#)
- [Выполнять код только на стороне сервера или на стороне клиента](#)
- [Развертывание-Deploying рабочей версии](#)
- [Развертывание на Now](#)
- [Анализируем комплекты приложений](#)
- [Ленивая загрузка модулей](#)
- [Куда идти дальше?](#)

Это руководство идеально подойдет для вас, если у вас практически нет знаний о Next.js, или если вы уже использовали React в прошлом и хотите больше углубиться в экосистему React, в частности, рендеринг на стороне сервера.

Я считаю Next.js отличным инструментом для создания веб-приложений, и в конце этого поста я надеюсь, что вы будете хорошо понимать его принципы работы и сможете создавать собственные приложения. И я надеюсь, что это поможет вам изучить Next.js!

## Введение

Работать над современным JavaScript-приложением на основе React - это круто, пока вы не поймете, что есть пара проблем, связанных с отображением всего контента на стороне клиента.

*Bo-первых*, для того, чтобы страница стала видимой пользователю, требуется больше времени, потому что перед загрузкой контента должен загрузиться весь JavaScript, и ваше приложение должно быть запущено, чтобы определить, что показывать на странице.

*Bo-вторых*, если вы создаете общедоступный веб-сайт, у вас есть проблема с SEO контента. Поисковые системы хорошоправляются с запуском и индексацией приложений JavaScript, но гораздо лучше, если мы сможем отправить им контент, а не дать им самим понимать его (роботы паучки и пр :-)).

Решением обеих этих проблем является рендеринг на стороне сервера, также называемый статическим предварительным рендерингом.

[Next.js](#) - это одна из сред React, которая делает все это очень простым способом, но не ограничивается этим. Создатели рекламируют его как набор инструментов с **одной командой для приложений React с нулевой конфигурацией**.

Он обеспечивает общую структуру, которая позволяет легко создавать приложение React с внешним интерфейсом, и прозрачно обрабатывает рендеринг на стороне сервера.

## Основные функции, предоставляемые Next.js

Вот неполный список основных функций Next.js:

### 1. Hot Code Reloading

Т.е. Горячая перезагрузка кода.  
Next.js перезагружает страницу, когда обнаруживает любые изменения, сохраненные на диске.

### 2. Automatic Routing

Т.е. Автоматическая маршрутизация  
Любой URL-адрес сопоставляется с файловой системой, с файлами, помещаемыми в папку страниц - [pages](#), и вам не нужно ничего настраивать (конечно, у вас есть опции настройки).

### 3. Single File Components

Т.е. Компоненты одного файла

Используя `styled-jsx`, полностью интегрированный и созданный той же командой, trivialно добавить стили, ограниченные компонентом.

### 4. Server Rendering

Рендеринг на стороне сервера

Вы можете визуализировать компоненты React на стороне сервера перед отправкой HTML клиенту.

### 5. Ecosystem Compatibility

Совместимость экосистемы

Next.js хорошо сочетается с остальной частью экосистемы JavaScript, Node и React.

### 6. Automatic Code Splitting

Автоматическое разделение кода

Страницы отображаются только с теми библиотеками и JavaScript, которые им нужны, не более. Вместо создания одного единственного файла JavaScript, содержащего весь код приложения, приложение автоматически разбивается на Next.js в нескольких различных ресурсах.

При открытии страницы, Next.js загружает только тот JavaScript, который необходим для этой конкретной страницы.

Next.js делает это путем анализа импортируемых ресурсов.

Например, если только одна из ваших страниц импортирует библиотеку `Axios`, эта конкретная страница будет включать библиотеку в свой пакет.

Это гарантирует, что ваша загрузка первой страницы будет настолько быстрой, насколько это возможно, и только будущие загрузки страниц (если они когда-либо будут запущены) отправят JavaScript, необходимый клиенту.

Есть одно заметное исключение. *Часто используемые операции импорта перемещаются в основной пакет JavaScript, если они используются по крайней мере на половине страниц сайта.*

### 7. Prefetching

Предзагрузка

Компонент `Link`, используемый для связывания вместе разных страниц, поддерживает функцию предварительной выборки, которая автоматически выполняет предварительную выборку ресурсов страницы (включая код, отсутствующий из-за разделения кода) в фоновом режиме.

### 8. Dynamic Components

Динамические Компоненты

Вы можете импортировать модули JavaScript и React Components динамически.

### 9. Static Exports

Статический экспорт

Используя следующую команду экспорта - `next export`, Next.js позволяет вам экспортировать полностью статический сайт из вашего приложения.

### 10. TypeScript Support

Поддержка TypeScript

Next.js написан на TypeScript и как таковой имеет отличную поддержку TypeScript.

## Next.js vs Gatsby vs `create-react-app`

Next.js, [Gatsby](#) и [create-react-app](#) - это удивительные инструменты, которые мы можем использовать для разработки наших приложений.

Давайте сначала скажем, что у них общего. У всех есть React под капотом, который обеспечивает весь опыт разработки. Они также абстрагируют веб-пакет и все те вещи низкого уровня, которые мы использовали для ручной настройки в старые добрые времена.

`create-react-app` не поможет вам легко создать приложение на стороне сервера. Все, что идет с ним (SEO, скорость ...), предоставляется только такими инструментами, как Next.js и Gatsby.

### Когда Next.js лучше, чем Гэтсби?

Они оба могут помочь с рендерингом на стороне сервера, но двумя разными способами.

Конечным результатом использования Gatsby является генератор статических сайтов без сервера. Вы создаете сайт, а затем статически развертываете результат процесса сборки на Netlify или другом статическом хостинге.

Next.js предоставляет серверную часть, которая может на стороне сервера отображать ответ на запрос, позволяя вам создать динамический веб-сайт, что означает, что вы развернете его на платформе, которая может запускать Node.js.

Next.js также может генерировать статический сайт, но я бы не сказал, что это его основной вариант использования.

Если бы моей целью было создание статического сайта, мне было бы трудно выбрать, и, возможно, у Гэтсби есть лучшая экосистема плагинов, в том числе и для блогов, в частности.

Gatsby также в значительной степени основан на [GraphQL](#), что вам может действительно нравиться или не нравиться в зависимости от ваших мнений и потребностей.

## Как установить Next.js

Чтобы установить Next.js, у вас должен быть установлен Node.js.

Убедитесь, что у вас установлена последняя версия Node. Проверьте, запустив в своем терминале `node -v`, и сравните его с последней версией LTS, указанной на <https://nodejs.org/>.

После установки Node.js в командной строке будет доступна команда npm.

Если у вас возникли проблемы на этом этапе, я рекомендую следующие уроки, которые я написал для вас [JS уроки на моем сайте](#)

Теперь, когда у вас есть Node, обновленный до последней версии, и npm, мы настроены!

Сейчас мы можем выбрать 2 варианта установки Next.js:

1. Используя `create-next-app`
2. Классический подход, который предполагает установку и настройку приложения Next вручную.

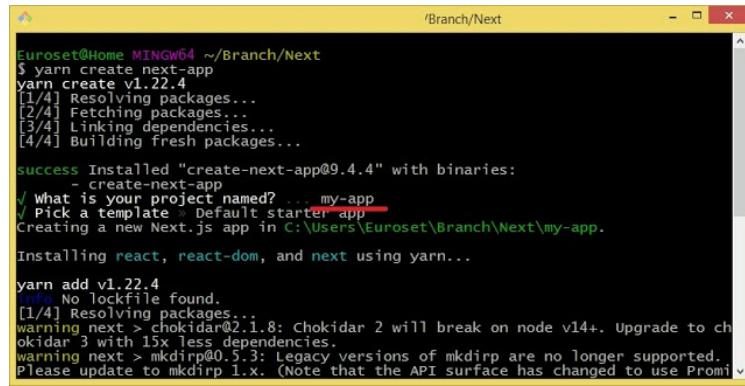
### Установка с `create-next-app`

Если вы знакомы с `create-react-app`, то `create-next-app` - это то же самое, за исключением того, что оно создает приложение «Next» вместо приложения «Реакт», как следует из названия.

Я предполагаю, что вы уже установили Node.js, который поставлялся в комплекте с командой npm. Этот удобный инструмент позволяет нам загружать и выполнять команду JavaScript, и мы будем использовать ее следующим образом:

```
npm create-next-app  
// или  
yarn create next-app
```

Команда запрашивает имя приложения (и создает для вас новую папку с таким именем - введем имя, в моем случае это `my-app`, и нажмем Enter), согласимся с установкой стартового пакета по умолчанию (Default starter app и нажмем Enter), и загрузим все необходимые пакеты (`react`, `react-dom`, `next`).



```
Euroset@Home MINGW64 ~/Branch/Next
$ yarn create next-app v1.22.4
yarn create v1.22.4
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

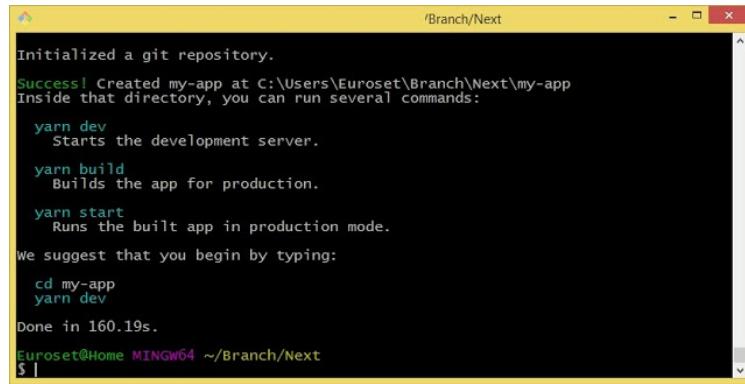
success Installed "create-next-app@9.4.4" with binaries:
  - create-next-app
  / What is your project named? ... my-app
  / Pick a template - Default starter app
Creating a new Next.js app in C:\Users\Euroset\Branch\Next\my-app.

Installing react, react-dom, and next using yarn...
yarn add v1.22.4
  No lockfile found.
[1/4] Resolving packages...
warning next > chokidar@2.1.8: Chokidar 2 will break on node v14+. Upgrade to chokidar 3 with 15x less dependencies.
warning next > mkdirp@0.5.3: Legacy versions of mkdirp are no longer supported.
Please update to mkdirp 1.x. (Note that the API surface has changed to use Promises)

```

Я использую yarn, но это не принципиально. Вы можете тоже самое делать с npm, с несколько другими командами. В руководстве я их приведу, наряду с командами для yarn.

Если все прошло нормально, то в конце установки вы увидите такие команды:



```
Initialized a git repository.

Success! Created my-app at C:\Users\Euroset\Branch\Next\my-app
Inside that directory, you can run several commands:

  yarn dev
    Starts the development server.
  yarn build
    Builds the app for production.
  yarn start
    Runs the built app in production mode.

We suggest that you begin by typing:

  cd my-app
  yarn dev

Done in 160.19s.
Euroset@Home MINGW64 ~/Branch/Next
$ |
```

Если сейчас открыть файл `package.json` нашего проекта, то мы увидим там все установленные зависимости и скрипты, с помощью которых мы будем управлять приложением.

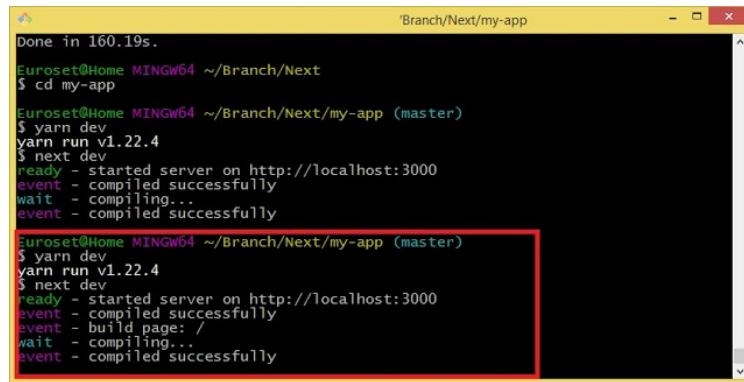
### package.json

```
{
  "name": "my-app",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start"
  },
  "dependencies": {
    "next": "9.4.4",
    "react": "16.13.1",
    "react-dom": "16.13.1"
  }
}
```

Давайте перейдем в папку проекта и запустим наше приложение:

```
cd my-app  
yarn dev  
// или  
npm dev
```

В консоли увидим:



```
Done in 160.19s.  
Euroset@Home MINGW64 ~/Branch/Next  
$ cd my-app  
Euroset@Home MINGW64 ~/Branch/Next/my-app (master)  
$ yarn dev  
yarn run v1.22.4  
$ next dev  
ready - started server on http://localhost:3000  
event - compiled successfully  
wait - compiling...  
event - compiled successfully  
Euroset@Home MINGW64 ~/Branch/Next/my-app (master)  
$ yarn dev  
yarn run v1.22.4  
$ next dev  
ready - started server on http://localhost:3000  
event - compiled successfully  
event - build page: /  
wait - compiling...  
event - compiled successfully
```

Если перейдем на <http://localhost:3000/>, то увидим, что стартовое приложение запустилось!

## Welcome to Next.js!

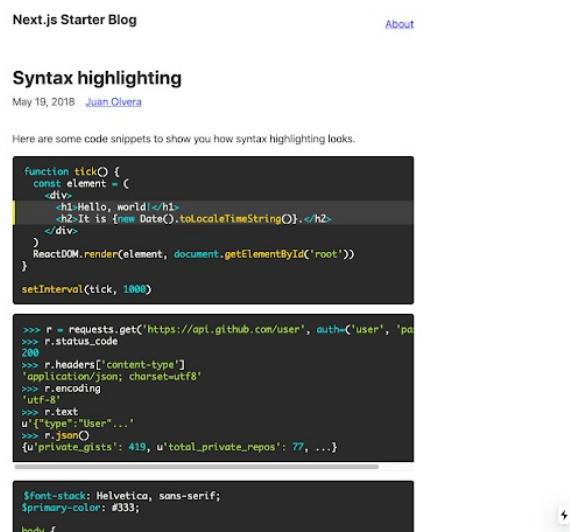
Get started by editing [pages/index.js](#)



Это рекомендуемый способ запуска приложения Next.js, поскольку он дает вам структуру и пример кода для игры. Это больше, чем просто пример приложения по умолчанию; Вы можете использовать любой из примеров, хранящихся по адресу <https://github.com/zeit/next.js/tree/canary/examples>, используя параметр --example. Например попробуйте [blog-starter](#):

```
npx create-next-app --example blog-starter blog-starter-app  
# or  
yarn create next-app --example blog-starter blog-starter-app
```

Это даст вам сразу используемый экземпляр блога с подсветкой синтаксиса:



The screenshot shows a blog post titled 'Syntax highlighting' by Juan Olvera, published on May 19, 2018. The post content includes several code snippets with syntax highlighting. One snippet shows a JavaScript function 'tick()' that creates a div with h1 and h2 headings. Another snippet shows a Python requests.get call to the GitHub API, returning a JSON response with user information like 'private\_gists' and 'total\_private\_repos'.

## Создание приложение Next.js вручную

Вы можете избежать создания [create-next-app](#) приложения, если вы хотите создать следующее приложение с нуля. Вот как: создайте пустую папку где угодно, например, в своей

домашней папке, и перейдите в нее:

```
mkdir nextjs  
cd nextjs
```

и создайте свою первую структуру Next:

```
mkdir firstproject  
cd firstproject
```

Теперь используйте команду `npm` или `yarn`, чтобы инициализировать его как проект Node:

```
npm init -y  
#or  
yarn init -y
```

```
Euroset@Home MINGW64 ~/Branch/Next  
$ mkdir firstproject  
Euroset@Home MINGW64 ~/Branch/Next  
$ cd firstproject/  
Euroset@Home MINGW64 ~/Branch/Next/firstproject  
$ yarn init -y  
yarn init v1.22.4  
warning The yes flag has been set. This will automatically answer yes to all questions, which may have security implications.  
success Saved package.json  
Done in 0.11s.  
Euroset@Home MINGW64 ~/Branch/Next/firstproject  
$ |
```

Опция `-y` указывает `npm` (и `yarn` тоже) использовать настройки проекта по умолчанию, заполняя образец файла `package.json`.

`package.json`

```
{  
  "name": "firstproject",  
  "version": "1.0.0",  
  "main": "index.js",  
  "license": "MIT"  
}
```

Установим зависимости:

```
npm install next react react-dom  
#or  
yarn add next react react-dom
```

После этого в папке вашего проекта появится папка `node_modules` и файл `package-lock.json` или, если вы работали с `yarn`, как я, то `yarn.lock`. В файле `package.json` появятся установленные зависимости:

`package.json`

```
{  
  "name": "firstproject",  
  "version": "1.0.0",  
  "main": "index.js",  
  "license": "MIT",  
  "dependencies": {  
    "next": "^9.4.4",  
    "react": "^16.13.1",  
    "react-dom": "^16.13.1"  
  }  
}
```

Нам остается просто добавить в него скрипты, для управления приложением:

`package.json`

```
{  
  "name": "firstproject",  
  "version": "1.0.0",  
  "main": "index.js",  
  "license": "MIT",  
  "scripts": {  
    "dev": "next",  
    "build": "next build",  
    "start": "next start"  
  },  
  "dependencies": {  
    "next": "^9.4.4",  
    "react": "^16.13.1",  
    "react-dom": "^16.13.1"  
  }  
}
```

Если вы хотите изменить порт запуска вашего приложения, то можно немного изменить скрипты:

Например так:

```
"dev": "next -p 3001"
```

Теперь создайте папку **pages** и добавьте файл **index.js**.

В этом файле давайте создадим наш первый компонент React.

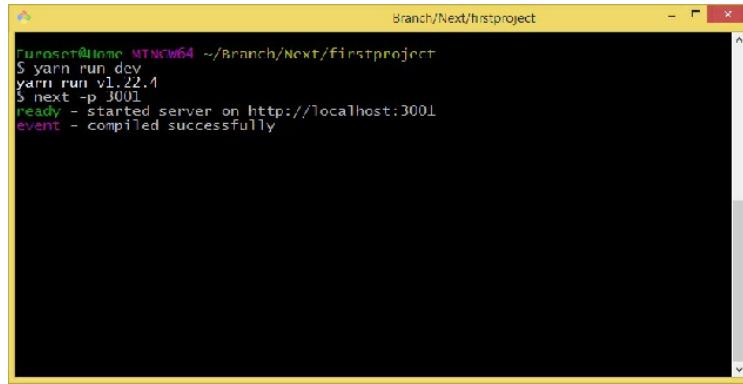
Мы собираемся использовать его как экспорт по умолчанию:

```
const Index = () => (
  <div>
    <h1>Home page</h1>
  </div>
);

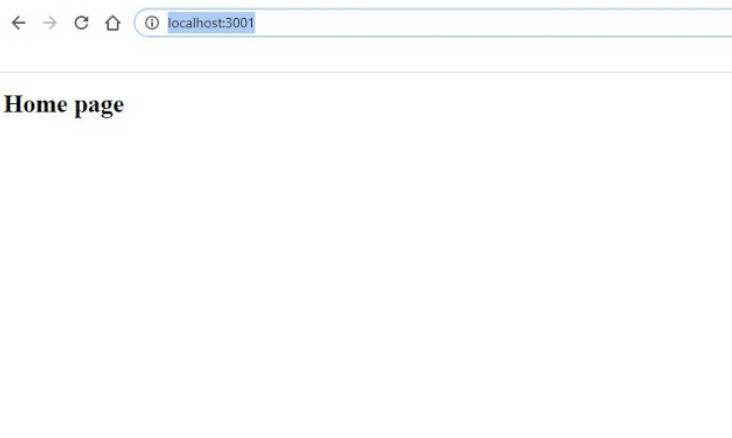
export default Index;
```

Сохраняем и запускаем наше приложение в консоли:

```
npm run dev
#or
yarn run dev
```



По адресу <http://localhost:3001/> мы увидим наше приложение:



## Просмотр источника, чтобы подтвердить работу SSR

Теперь давайте проверим, работает ли приложение, как мы ожидаем, и что оно будет рендерить на стороне сервера (**SSR - server side rendering**). Это приложение Next.js, поэтому оно должно отображаться на стороне сервера.

Это одно из главных преимуществ Next.js: если мы создаем сайт с помощью Next.js, страницы сайта отображаются на сервере, который доставляет HTML в браузер.

Это имеет 3 основных преимущества:

- Клиенту не нужно создавать экземпляр React для рендеринга, что делает сайт быстрее для ваших пользователей.
- Поисковые системы будут индексировать страницы без необходимости запуска клиентского JavaScript. Что-то, что Google начал делать, но открыто признал, что это более медленный процесс (вы должны как можно больше помочь Google, если хотите получить хороший рейтинг).
- Вы можете использовать мета-теги в социальных сетях, полезные для добавления изображений для предварительного просмотра, настройки заголовка и описания для любой из ваших страниц, размещенных в Facebook, Twitter и т. д.

Давайте посмотрим на исходник приложения. Используя Chrome, вы можете щелкнуть правой кнопкой мыши в любом месте страницы и нажать **View page source**.

```

<!DOCTYPE html><html><head><style data-next-hide-fouc="true">body{display:none}</style></head><script data-next-hide-fouc="true"><style>body{display:block}</style></script><meta name="viewport" content="width=device-width"/><meta name="next-head-count" content="2"/><link rel="preload" href="/_next/static/development/pages/_app.js?ts=1592569265801" as="script"/><link rel="preload" href="/_next/static/development/pages/index.js?ts=1592569265801" as="script"/><link rel="preload" href="/_next/static/runtime/webpack.js?ts=1592569265801" as="script"/><link rel="preload" href="/_next/static/runtime/main.js?ts=1592569265801" as="script"/><noscript id="__next_css__DO_NOT_USE__"></noscript></head><body><div id="__next"><div><h1>Home page</h1></div></div><script src="/_next/static/runtime/react-refresh.js?ts=1592569265801"></script><script src="/_next/static/development/dll.dll_f9de5cbc314ale4lf9le.js?ts=1592569265801"></script><script src="/_next/static/runtime/polyfills.js?ts=1592569265801"></script><script type="application/json" data-next-page="/_app"></script><script src="/_next/static/development/pages/_app.js?ts=1592569265801"></script><script src="/_next/static/development/pages/index.js?ts=1592569265801"></script><script src="/_next/static/runtime/webpack.js?ts=1592569265801" as="script"/><script src="/_next/static/runtime/main.js?ts=1592569265801" as="script"/><script src="/_next/static/development/_buildManifest.js?ts=1592569265801" as="script"/><script src="/_next/static/development/_ssgManifest.js?ts=1592569265801" as="script"/></script></body></html>

```

Если вы просмотрите исходный код страницы, вы увидите фрагмент

```

<div>
  <h1>Home page</h1>
</div>;

```

в теле HTML вместе с набором файлов JavaScript - пакетами приложений.

Нам не нужно ничего настраивать, SSR (рендеринг на стороне сервера) уже работает для нас.

Приложение React будет запущено на клиенте, и оно будет активным взаимодействием, таким как нажатие на ссылку с использованием рендеринга на стороне клиента. Но перезагрузка страницы перезагрузит ее с сервера. И при использовании Next.js не должно быть никакой разницы в результатах внутри браузера - страница, отображаемая на сервере, должна выглядеть точно так же, как страница, отображаемая клиентом.

## Пакеты приложений

Когда мы просмотрели исходный код страницы, мы увидели несколько загружаемых файлов JavaScript:

```

<!DOCTYPE html><html><head><style data-next-hide-fouc="true">body{display:none}</style></head><script data-next-hide-fouc="true"><style>body{display:block}</style></script><meta name="viewport" content="width=device-width"/><meta name="next-head-count" content="2"/><link rel="preload" href="/_next/static/development/pages/_app.js?ts=1592569265801" as="script"/><link rel="preload" href="/_next/static/development/pages/index.js?ts=1592569265801" as="script"/><link rel="preload" href="/_next/static/runtime/webpack.js?ts=1592569265801" as="script"/><link rel="preload" href="/_next/static/runtime/main.js?ts=1592569265801" as="script"/><noscript id="__next_css__DO_NOT_USE__"></noscript></head><body><div id="__next"><div><h1>Home page</h1></div></div><script src="/_next/static/runtime/react-refresh.js?ts=1592569265801"></script><script src="/_next/static/development/dll.dll_f9de5cbc314ale4lf9le.js?ts=1592569265801"></script><script id="__NEXT_DATA__" type="application/json">{"props": {"page": "/"}, "page": "/", "query": {}, "buildId": "development", "nextExport": true, "autoExport": true, "isFallback": false}</script><script nomodule="" src="/_next/static/runtime/polyfills.js?ts=1592569265801"></script><script type="application/json" data-next-page="/_app"></script><script src="/_next/static/development/pages/index.js?ts=1592569265801"></script><script src="/_next/static/development/dll.dll_f9de5cbc314ale4lf9le.js?ts=1592569265801"></script><script src="/_next/static/runtime/webpack.js?ts=1592569265801" as="script"/><script src="/_next/static/runtime/main.js?ts=1592569265801" as="script"/><script src="/_next/static/development/_buildManifest.js?ts=1592569265801" as="script"/><script src="/_next/static/development/_ssgManifest.js?ts=1592569265801" as="script"/></script></body></html>

```

Давайте начнем с того, что поместим код в средство форматирования HTML [HTML formatter](#), чтобы лучше отформатировать его и лучше понять:

```

<!DOCTYPE html>
<html>
  <head>
    <style data-next-hide-fouc="true">
      body {
        display: none;
      }
    </style>
    <noscript data-next-hide-fouc="true">
      <style>
        body {
          display: block;
        }
      </style>
    </noscript>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <meta name="next-head-count" content="2" />
    <link rel="preload" href="/_next/static/development/pages/_app.js?ts=1592569265801" as="script" />
    <link rel="preload" href="/_next/static/development/pages/index.js?ts=1592569265801" as="script" />
    <link rel="preload" href="/_next/static/runtime/webpack.js?ts=1592569265801" as="script" />
    <link rel="preload" href="/_next/static/runtime/main.js?ts=1592569265801" as="script" />
    <noscript id="__next_css__DO_NOT_USE__"></noscript>
  </head>
  <body>
    <div id="__next">
      <div><h1>Home page</h1></div>
    </div>
    <script src="/_next/static/runtime/react-refresh.js?ts=1592569265801"></script>
    <script src="/_next/static/development/dll.dll_f9de5cbc314ale4lf9le.js?ts=1592569265801"></script>
    <script id="__NEXT_DATA__" type="application/json">
      {"props": {"page": "/"}, "page": "/", "query": {}, "buildId": "development", "nextExport": true, "autoExport": true, "isFallback": false}
    </script>
    <script nomodule="" src="/_next/static/runtime/polyfills.js?ts=1592569265801"></script>
    <script type="application/json" data-next-page="/_app"></script>
    <script src="/_next/static/development/pages/index.js?ts=1592569265801"></script>
    <script src="/_next/static/runtime/webpack.js?ts=1592569265801" as="script"/>
    <script src="/_next/static/runtime/main.js?ts=1592569265801" as="script"/>
    <script src="/_next/static/development/_buildManifest.js?ts=1592569265801" as="script"/>
    <script src="/_next/static/development/_ssgManifest.js?ts=1592569265801" as="script"/>
  </body>
</html>

```

У нас есть 4 файла JavaScript, которые объявлены предварительно загруженными в head, используя `rel = "preload" as = "script"`:

- `/_next/static/development/pages/index.js` (96 LOC)
- `/_next/static/development/pages/_app.js` (5900 LOC)

- /\_next/static/runtime/webpack.js (939 LOC)
- /\_next/static/runtime/main.js (12k LOC)

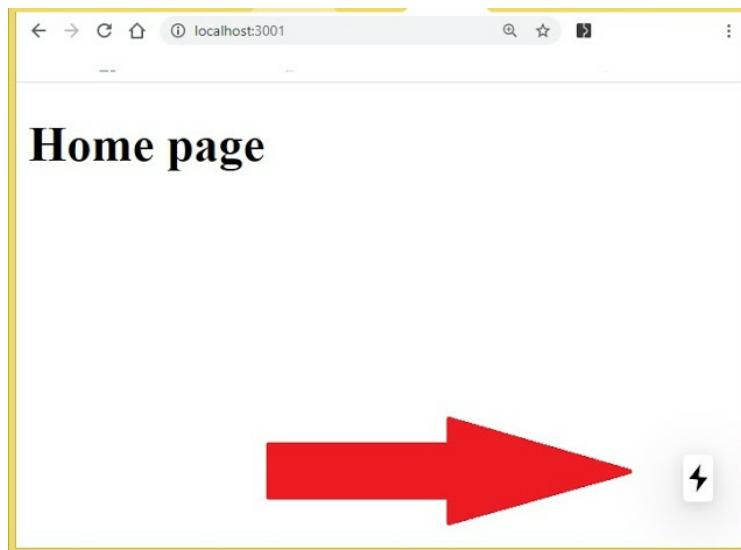
Это говорит браузеру начать загрузку этих файлов как можно скорее, прежде чем начнется обычный процесс рендеринга. Без них скрипты будут загружаться с дополнительной задержкой, что повышает производительность загрузки страницы. Затем эти 4 файла загружаются в конце тела вместе с /\_next/static/development/dll/dll\_01ec57fc9b90d43b98a8.js (31k LOC) и фрагментом JSON, который устанавливает некоторые значения по умолчанию для данных страницы:

```
<script id="__NEXT_DATA__" type="application/json">
{
  "dataManager": "[ ]",
  "props": {
    "pageProps": {}
  },
  "page": "/",
  "query": {},
  "buildId": "development",
  "nextExport": true,
  "autoExport": true
}
</script>
```

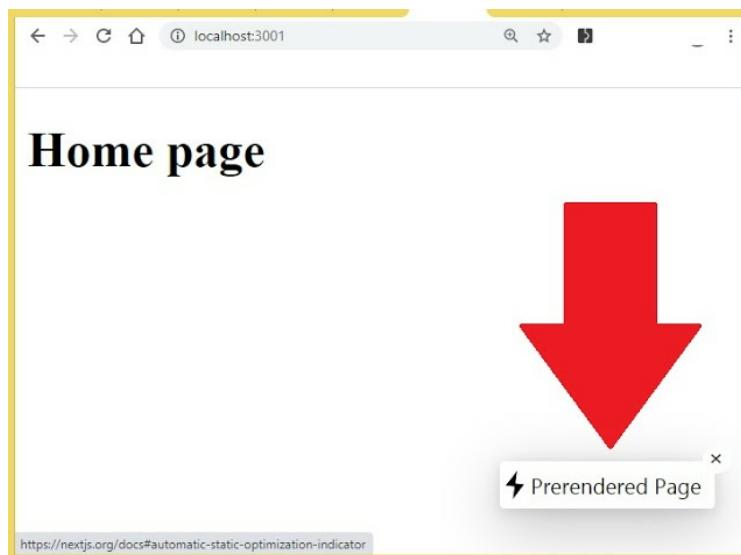
4 загруженных файла пакета уже реализуют одну функцию, называемую *разделением кода*. Файл index.js предоставляет код, необходимый для компонента index, который обслуживает / route, и если бы у нас было больше страниц, у нас было бы больше пакетов для каждой страницы, которые затем будут загружаться только при необходимости - для обеспечения большей производительности и уменьшения времени загрузки страницы.

## Что это за иконка справа внизу?

Вы видели эту маленькую иконку в правом нижнем углу страницы, которая выглядит как молния?



Если вы наведите указатель мыши, на нем будет написано «Prerendered Page»:



Этот значок, который, конечно, виден только в режиме разработки, говорит о том, что страница подходит для автоматической статической оптимизации, что в основном означает, что она не зависит от данных, которые необходимо получить во время вызова, и может быть предварительно обработана и построена как статический HTML-файл во время сборки (когда мы запускаем npm, запускаем сборку - npm run build).

Далее это можно определить по отсутствию метода getInitialProps (), прикрепленного к компоненту страницы.

В этом случае наша страница может быть еще быстрее, потому что она будет статически обслуживаться как файл HTML, а не через сервер Node.js, который генерирует вывод

HTML.

Другой полезный значок, который может появиться рядом с ним или вместо него на страницах без предварительной обработки, представляет собой небольшой анимированный треугольник:



Это индикатор компиляции, который появляется, когда вы сохраняете страницу, а Next.js компилирует приложение до того, как начнется горячая перезагрузка кода, чтобы автоматически перезагрузить код в приложении.

Это действительно хороший способ сразу определить, было ли приложение уже скомпилировано, и вы можете протестировать часть, над которой вы работаете.

## Установка React Developer Tools

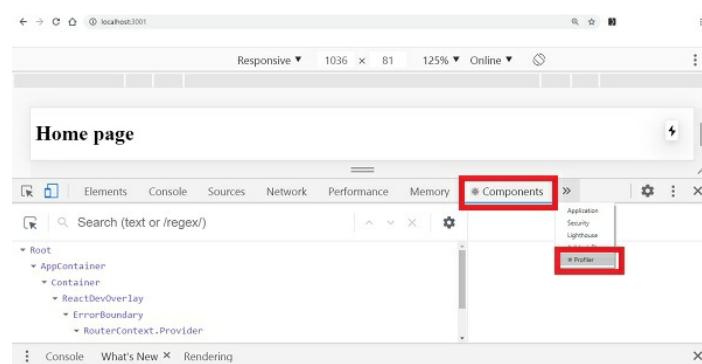
Next.js основан на React, поэтому есть один очень полезный инструмент, который нам абсолютно необходимо установить (если вы этого еще не сделали), это **React Developer Tools**.

Доступные как для [Chrome](#), так и для [Firefox](#), инструменты разработчика React являются важным инструментом, который вы можете использовать для проверки приложения React.

Теперь инструменты разработчика React не являются специфическими для Next.js, но я хочу представить их, потому что вы, возможно, не на 100% знакомы со всеми инструментами, которые предоставляет React. Лучше немного заняться инструментами отладки, чем предполагать, что вы их уже знаете.

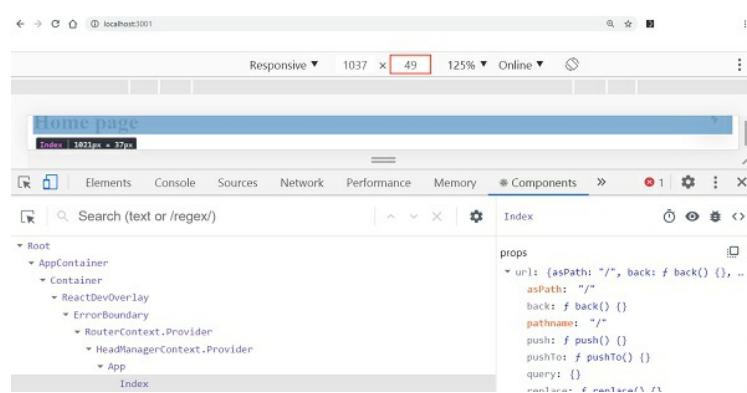
Они предоставляют инспектора, который показывает дерево компонентов React, которое создает вашу страницу, и для каждого компонента вы можете посмотреть и проверить реквизиты - `props`, состояние - `state`, хуки и многое другое.

После того, как вы установили React Developer Tools, вы можете открыть обычный браузер devtools (в Chrome щелкнуть правой кнопкой мыши на странице, затем нажать «Проверить»), и вы увидите 2 новые панели: Компоненты - **Components** и Профилировщик - **Profiler**.



Если вы наведете указатель мыши на компоненты, вы увидите, что на странице браузер будет выбирать части, отображаемые этим компонентом.

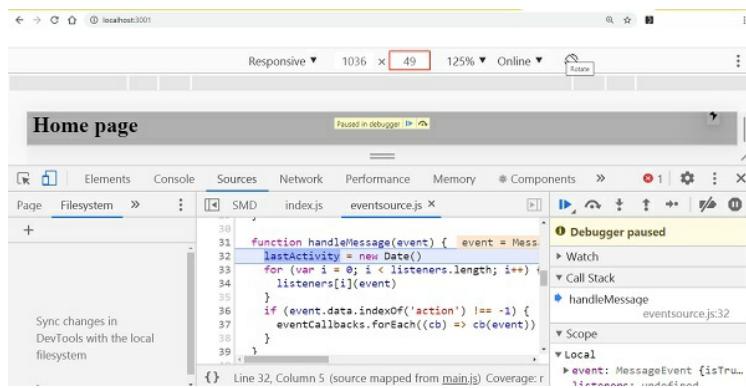
Если вы выберете какой-либо компонент в дереве, на правой панели отобразится ссылка на родительский компонент, а реквизиты - `props` будут переданы ему:



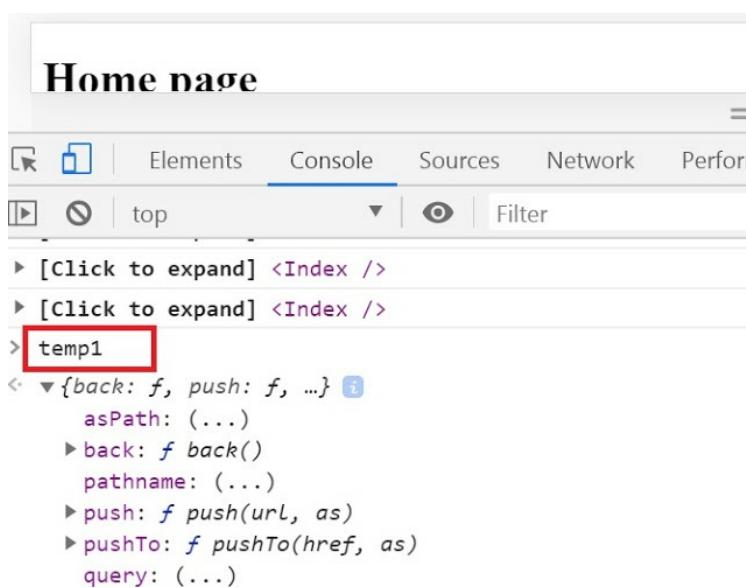
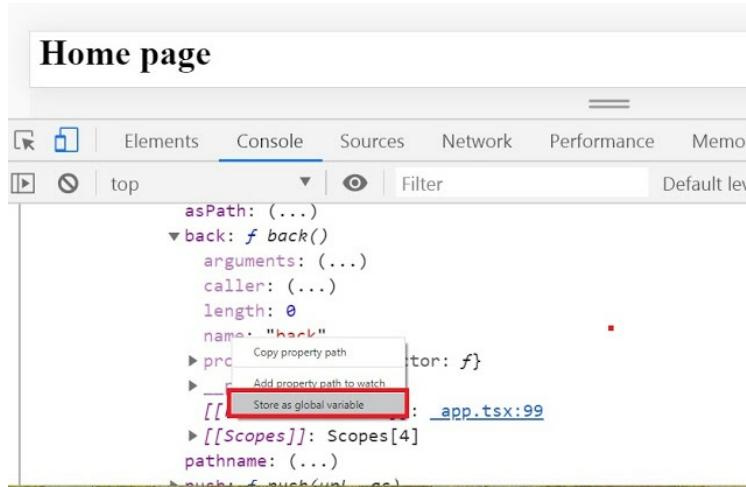
Вы можете легко перемещаться, нажимая на названия компонентов.

Вы можете щелкнуть значок глаза на панели инструментов «Инструменты разработчика», чтобы просмотреть элемент DOM, а также, если вы используете первый значок со значком мыши (который удобно расположен под аналогичным обычным значком DevTools), вы можете навести элемент на пользовательский интерфейс браузера для непосредственного выбора компонента React, который его отображает.

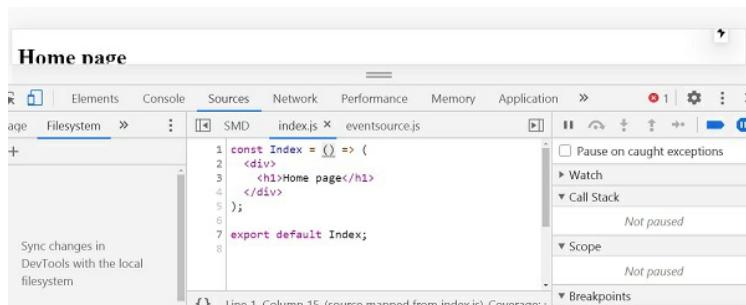
Вы можете использовать значок **bug**, чтобы записать данные компонента на консоль.



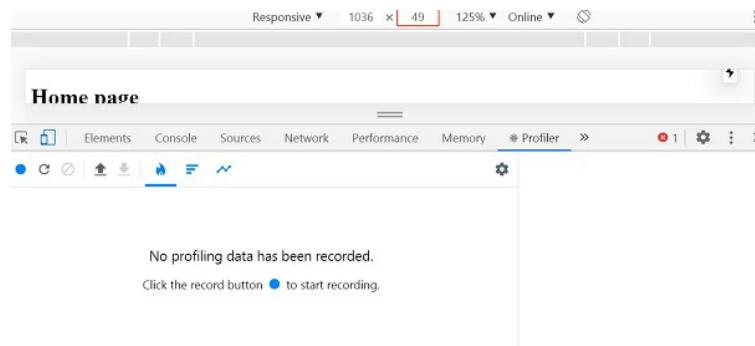
Это довольно круто, потому что, как только вы напечатаете там данные, вы можете щелкнуть правой кнопкой мыши по любому элементу и нажать «Сохранить как глобальную переменную - Store as global variable». Например, здесь я сделал это с помощью url prop, и я смог проверить его в консоли, используя временную переменную, назначенную ему, temp1:



Используя Карты исходного кода - **Source Maps**, которые автоматически загружаются Next.js в режиме разработки, на панели «Компоненты» мы можем щелкнуть код <>, и DevTools переключится на панель «Источник», показывая нам исходный код компонента:



Вкладка «Профилизовщик», по возможности, еще более крутая. Это позволяет нам записывать взаимодействие в приложении и видеть, что происходит. Я пока не могу показать пример, потому что для создания взаимодействия нужны как минимум 2 компонента, а у нас есть только один. Я поговорю об этом позже.



Я показал все скриншоты, используя Chrome, но React Developer Tools работает точно так же в Firefox:

## Другие методы отладки, которые вы можете использовать

В дополнение к инструментам разработчика React, которые необходимы для создания приложения Next.js, я хочу подчеркнуть два способа отладки приложений Next.js:

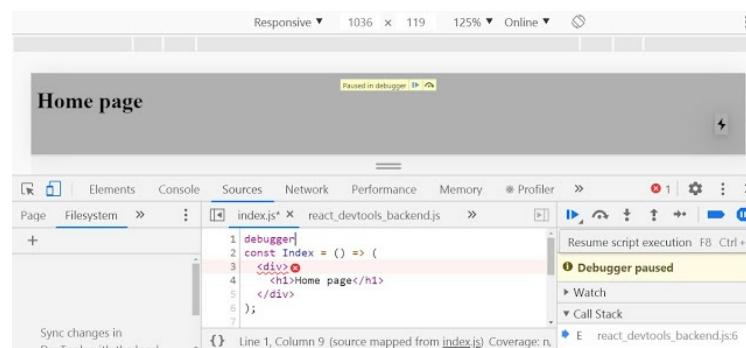
1. Первый, очевидно, `console.log()` и все остальные [инструменты API консоли](#). Работа приложений Next заставит оператор `log` работать в консоли браузера ИЛИ в терминале, с которого вы начали Next, используя `npm run dev`.

В частности, если страница загружается с сервера, когда вы указываете на нее URL-адрес или нажимаете кнопку обновления / cmd / `ctrl + R`, любое ведение журнала консоли происходит в терминале.

Последующие переходы страниц, которые происходят при щелчке мыши, приведут к тому, что вся запись в консоли будет происходить внутри браузера.

Просто помните, если вы удивлены отсутствием реакции.

2. Другим важным инструментом является оператор отладчика - `debugger`. Добавление этого оператора к компоненту приостановит отображение страницы браузером:



Действительно круто, потому что теперь вы можете использовать отладчик браузера для проверки значений и запуска своего приложения по одной строке за раз.

Вы также можете использовать отладчик VS Code для отладки кода на стороне сервера. Я упоминаю эту технику и [этот урок](#), чтобы настроить его.

## Добавление второй страницы на сайт

Теперь, когда мы хорошо разбираемся в инструментах, которые мы можем использовать для разработки приложений Next.js, давайте продолжим с того места, где мы оставили наше первое приложение:



Я хочу добавить вторую страницу на этот сайт, блог. Она будет называться `blog`, и на данный момент она будет содержать простую статическую страницу, как наш первый компонент `index.js`:

The screenshot shows the Visual Studio Code interface. On the left is the Explorer sidebar with a tree view of files and folders. The current file, `blog.js`, is selected and highlighted in blue. The main area is a code editor with the following content:

```
pages > blog.js > ...
1 const Blog = () => (
2   <div>
3     |   <h1>Blog page</h1>
4   </div>
5 );
6
7 export default Blog;
8 |
```

После сохранения нового файла уже запущенный процесс `npm run dev` уже способен отображать страницу без необходимости ее перезапуска.

Когда мы нажимаем на URL <http://localhost:3001/blog>, у нас появляется новая страница



и вот что сказал нам терминал:

```
event - compiled successfully
event - build page: /next/dist/pages/ error
wait  - compiling...
event - compiled successfully
event - build page: /
wait  - compiling...
event - compiled successfully
event - build page: /
wait  - compiling...
event - compiled successfully

Euroset@Home MINGW64 ~/Branch/Next/firstproject
$ yarn run dev
yarn run v1.22.4
$ next -p 3001
[ready] - started server on http://localhost:3001
event - compiled successfully
event - build page: /
wait  - compiling...
event - compiled successfully
event - build page: /blog
wait  - compiling...
event - compiled successfully
```

Теперь тот факт, что URL является `/blog`, зависит только от имени файла и его положения в папке страницы.

Вы можете создать страницу `pages /hey/ho`, и эта страница будет отображаться по URL-адресу <http://localhost:3001/hey/ho>

Для URL не имеет значения имя компонента внутри файла.

Попробуйте просмотреть исходную страницу, когда она будет загружена с сервера, в качестве одного из загруженных пакетов будет указан `/next/static/development/pages/blog.js`, а не `/next/static/development/pages/index.js`, как на домашней странице. Это потому, что благодаря автоматическому разделению кода нам не нужен пакет, обслуживающий ломаную страницу. Просто пакет, который обслуживает страницу блога.

```
<!DOCTYPE html><html><head><style data-next-hide-fou="true">body{display:none}</style><noscript data-next-hide-fou="true"><body><script>document.write('');</script></body></noscript><meta charset="utf-8" data-name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=0" data-bbox="450 100 886 115" /><link rel="preload" href="/_next/static/development/pages/_app.js?ts=15925790280580" as="script"/><link rel="preload" href="/_next/static/development/pages/blog.js?ts=15925790280580" as="script"/><link rel="preload" href="/_next/static/runtime/webpack.js?ts=15925790280580" as="script"/><script id="__next_cc_DO_NOT_USE__"></script></head><body data-bbox="109 115 886 250" id="__next"><div><h1>Blog Page</h1></div></body></html>
```

Мы также можем просто экспортить анонимную функцию из blog.js

```
export default () => (
  <div>
    <h1>Blog</h1>
  </div>
);
```

или если вы предпочитаете синтаксис функции без стрелки:

```
export default function () {
  return (
    <div>
      <h1>Blog</h1>
    </div>
  );
}
```

## Связывание двух страниц

Теперь, когда у нас есть 2 страницы, определенные `index.js` и `blog.js`, мы можем ввести ссылки.

Обычные HTML-ссылки на страницах выполняются с помощью тега `a`:

```
<a href="/blog">Blog</a>;
```

Мы не можем сделать это в Next.js.

Почему? Мы технически можем, конечно, потому что это Интернет, и в Интернете вещи никогда не ломаются (хотя почему мы все еще можем использовать тег `<marquee>`). Но одно из основных преимуществ использования Next заключается в том, что после загрузки страницы происходит переход на другую страницу очень быстро благодаря рендерингу на стороне клиента.

Если вы используете простую ссылку `a` тега:

```
const Index = () => (
  <div>
    <h1>Home page</h1>
    <a href="/blog">Blog</a>
  </div>
);

export default Index;
```

Теперь откройте DevTools и, в частности, панель «Network». При первой загрузке `http://localhost:3001/` загружаются все пакеты страниц:

Name	Status	Type	Initiator	Size	Time
._buildManifest.js?ts=1...	200	script	blog	393 B	25 ms
dll_f9de5cbc314a1e41f...	200	script	blog	223 kB	378 ms
react-refresh.js?ts=159...	200	script	blog	23.0 ...	92 ms
main.js?ts=159258216...	200	script	blog	393 kB	440 ms
webpack.js?ts=159258...	200	script	blog	21.2 ...	94 ms
blog.js?ts=1592582162...	200	script	blog	4.1 kB	150 ms
._app.js?ts=1592582162...	200	script	blog	89.0 ...	162 ms

Теперь, если вы нажмете кнопку «Сохранить журнал» (чтобы избежать очистки панели «Сеть») и нажмете ссылку «Блог», количество загружаемых объектов удвоится и они будут повторять друг друга.

Name	Status	Type	Initiator	Size	Time
react-devtools-backend.js	200	script	injectGlobalH...	154 kB	190 ms
0.js	200	script	bootstrap:936	1.9 kB	7 ms
._ssgManifest.js?ts=15925824...	200	script	blog	388 B	38 ms
._buildManifest.js?ts=1592582...	200	script	blog	393 B	36 ms
dll_f9de5cbc314a1e41f91e.js?...	200	script	blog	223 kB	370 ms
react-refresh.js?ts=15925824...	200	script	blog	23.0 kB	92 ms
main.js?ts=1592582471420	200	script	blog	393 kB	417 ms
webpack.js?ts=1592582471420	200	script	blog	21.2 kB	83 ms
blog.js?ts=1592582471420	200	script	blog	4.1 kB	131 ms
._app.js?ts=1592582471420	200	script	blog	89.0 kB	167 ms
react-devtools-backend.js	200	script	injectGlobalH...	154 kB	31 ms
0.js	200	script	bootstrap:936	1.9 kB	7 ms

Мы снова получили весь этот JavaScript с сервера! Но .. нам не нужен весь этот JavaScript, если мы его уже получили. Нам просто нужен пакет страниц `blog.js`, единственный новый для этой страницы.

Чтобы решить эту проблему, мы используем компонент Next, который называется `Link`.

Мы импортируем это:

```
import Link from 'next/link'
```

и затем мы используем его, чтобы обернуть нашу ссылку, вот так:

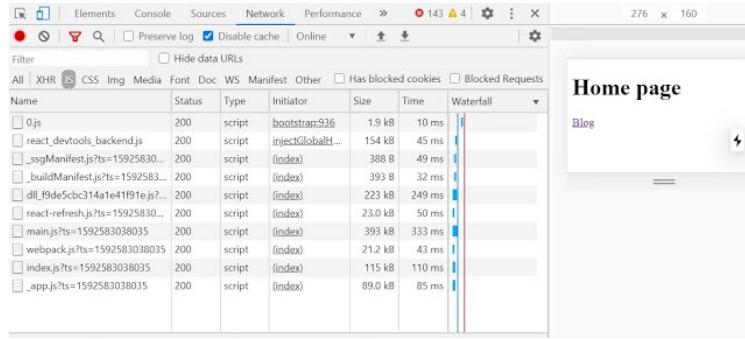
```
import Link from "next/link";

const Index = () => (
  <div>
    <h1>Home page</h1>
    <Link href="/blog">
      <a>Blog</a>
    </Link>
  </div>
);

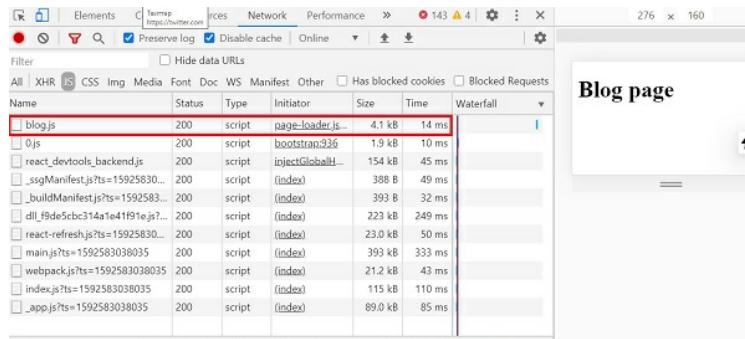
export default Index;
```

Теперь, если вы повторите попытку, которую мы делали ранее, вы увидите, что при переходе на страницу блога загружается только пакет `blog.js`:

Первая загрузка:



Переход на блок - добавился один файл - `blog.js`



и страница загружалась быстрее, чем раньше, обычная прокрутка браузера на вкладке даже не появлялась. Тем не менее, как вы видите, URL изменился. Это работает без проблем с API истории браузера.

Это рендеринг на стороне клиента в действии.

Что если вы сейчас нажмете кнопку «Назад»? Ничего не загружается, потому что в браузере все еще есть старый пакет `index.js`, готовый загрузить маршрут `/index`. Это все автоматически!

## Динамический контент с роутером

В предыдущей главе мы увидели, как связать страницу `home` со страницей `blog`.

Блог - отличный пример использования Next.js, который мы продолжим исследовать в этой главе, добавляя посты в блоге.

Сообщения блога имеют динамический URL. Например, пост под названием «Hello World» может иметь URL `/blog/hello-world`. Сообщение под названием «Мой второй пост» - (My second post) может иметь URL `/blog/my-second-post`.

Это содержимое является динамическим и может быть взято из базы данных, файлов разметки или другого.

Next.js может обслуживать динамический контент на основе **динамического URL**.

Мы создаем динамический URL, создавая динамическую страницу с синтаксисом `[ ]`.

Как? Мы добавляем файл `pages /blog/[id].js`. Этот файл будет обрабатывать все динамические URL-адреса в `/blog/route`, например, те, которые мы упомянули выше: `/blog/hello-world`, `/blog/my-second-post` и другие.

В имени файла `[id]` в квадратных скобках означает, что все это является динамическим, будет помещено в параметр `id` свойства запроса маршрутизатора.

Маршрутизатор -роутер является библиотекой, предоставленной Next.js.

Импортируем его из next/router:

```
import { useRouter } from 'next/router'
```

и как только мы используем useRouter, мы создаем объект маршрутизатора, используя:

```
const router = useRouter()
```

Как только мы получим этот объект маршрутизатора, мы сможем извлечь из него информацию.

В частности, мы можем получить динамическую часть URL в файле [id].js, обратившись к router.query.id.

Динамическая часть также может быть просто частью URL, например post-[id].js.

Итак, давайте продолжим и применим все эти вещи на практике.

Создайте файл pages/blog/[id].js:

```
import { useRouter } from "next/router";
export default () => {
  const router = useRouter();
  return (
    <>
      <h1>Blog post</h1>
      <p>Post id: {router.query.id}</p>
    </>
  );
};
```

Теперь, если вы перейдете к маршрутизатору <http://localhost:3001/blog/test>, вы должны увидеть это:



Мы можем использовать этот параметр id, чтобы собрать сообщение из списка сообщений. Из базы данных, например. Для простоты мы добавим файл posts.json в корневую папку проекта:

posts.json

```
{
  "test": {
    "title": "test post",
    "content": "Hey some post content"
  },
  "second": {
    "title": "second post",
    "content": "Hey this is the second post content"
  }
}
```

Теперь мы можем импортировать его и найти сообщение из ключа id:

posts/[id].js

```
import { useRouter } from "next/router";
import posts from "../../posts.json";
export default () => {
  const router = useRouter();
  const post = posts[router.query.id];
  return (
    <>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </>
  );
};
```

Перезагружаем страницу и ожидаем увидеть этот результат:

The screenshot shows a browser window with the URL 'localhost:3001/blog/test'. The page displays the title 'Hello' and the post content 'test post' followed by 'Hey some post content'.

Но это не так! Вместо этого мы получаем ошибку в консоли, а также ошибку в браузере:

The screenshot shows the browser's developer tools console with an error message: '1 of 1 unhandled error'. The error is a 'Server Error' with the message 'TypeError: Cannot read property "title" of undefined'. It also states that the error happened while generating the page. Below the error message is a 'Call Stack' section. The bottom of the screenshot shows the browser's address bar with the URL 'http://localhost:3001/blog/test' and a status of '500 (Internal Server Error)'.

Почему? Потому что .. во время рендеринга, когда компонент инициализирован, данных еще нет. Мы увидим, как предоставить данные компоненту с помощью `getInitialProps` на следующем уроке.

А пока добавьте небольшую проверку `if (! Post) return <p> </p>` перед возвратом JSX:

#### pages/blog/[id].js

```
import { useRouter } from "next/router";
import posts from "../../posts.json";

export default () => {
  const router = useRouter();

  const post = posts[router.query.id];

  if (!post) return <p> </p>
  return (
    <>
      <h2>Hello</h2>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </>
  );
}
```

Теперь все должно работать. Первоначально компонент отображается без динамической информации `router.query.id`. После рендеринга Next.js запускает обновление со значением запроса, и на странице отображается правильная информация.

И если вы просматриваете источник, в HTML есть пустой тег `<p>`:

The screenshot shows the browser's developer tools source tab with the generated HTML code. It includes a blank `<p>` tag where the dynamic content would normally be placed.

Вскоре мы исправим эту проблему, которая не позволяет реализовать SSR, и это вредит как времени загрузки для наших пользователей, так и SEO и социальному обмену, как мы уже обсуждали.

Мы можем завершить пример блога, перечислив эти посты в `pages/blog.js`:

#### pages/blog.js

```

import posts from "../posts.json";

const Blog = () => (
  <div>
    <h1>Blog</h1>

    <ul>
      {Object.entries(posts).map((value, index) => {
        return <li key={index}>{value[1].title}</li>;
      })}
    </ul>
  </div>
);

export default Blog;

```

И мы можем связать их с отдельными страницами сообщений, импортировав `Link` из `next/link` и используя ее в цикле сообщений:

#### pages/blog.js

```

import Link from "next/link";
import posts from "../posts.json";

const Blog = () => (
  <div>
    <h1>Blog</h1>

    <ul>
      {Object.entries(posts).map((value, index) => {
        return (
          <li key={index}>
            <Link href={`/blog/${id}`} as={`/blog/${value[0]}`}>
              <a>{value[1].title}</a>
            </Link>
          </li>
        );
      });
    </ul>
  </div>
);

export default Blog;

```

## Предзагрузка

Ранее я упоминал, как компонент `Link` Next.js можно использовать для создания ссылок между двумя страницами, и когда вы его используете, Next.js прозрачно обрабатывает маршрутизацию веб-интерфейса для нас, поэтому, когда пользователь нажимает на ссылку, веб-интерфейс выполняет показ новой страницы без запуска нового запроса / ответа клиента / сервера, как это обычно происходит с веб-страницами.

Еще одна вещь, которую Next.js делает для вас, когда вы используете `Link`.

Как только элемент, заключенный в `<Link>`, появляется в окне просмотра (что означает, что он виден пользователю веб-сайта), Next.js предварительно выбирает URL-адрес, на который он указывает, при условии, что это локальная ссылка (на вашем веб-сайте), делая приложение супер быстрым для пользователя.

Такое поведение запускается только в производственном режиме - `production mode` (об этом мы поговорим позже), что означает, что вам нужно остановить приложение, если вы запускаете его с помощью `npm run dev`, скомпилировать свой рабочий пакет с помощью `npm run build` и запустить его командой - `npm run start`.

Используя инспектор сети - Network inspector в DevTools, вы заметите, что любые ссылки выше, при загрузке страницы, запускают предварительную выборку, как только событие загрузки - `load` запускается на вашей странице (срабатывает, когда страница полностью загружена, и происходит после события `DOMContentLoaded`).

Любой другой тег `Link`, отсутствующий в области просмотра, будет предварительно выбран при прокрутке пользователем.

Предварительная выборка выполняется автоматически на высокоскоростных соединениях (соединения Wi-Fi и 3G +, если браузер не отправляет HTTP-заголовок `Save-Data`).

Вы можете отказаться от предварительной выборки отдельных экземпляров `Link`, установив для параметра `prefetch` значение `false`:

```

<Link href="/a-link" prefetch={false}>
  <a>A link</a>
</Link>;

```

## Использование роутера для обнаружения активной ссылки

Одна очень важная функция при работе со ссылками - это определение текущего URL-адреса и, в частности, назначение класса активной ссылке, чтобы мы могли стилизовать его не так, как другие.

Это особенно полезно, например, в заголовке вашего сайта.

Компонент `Link` от Next.js по умолчанию, предлагаемый в `next/link`, не делает это автоматически для нас.

Мы можем создать компонент `Link` самостоятельно, и мы храним его в файле `Link.js` в папке `Components`, и импортируем его вместо `next/link` по умолчанию.

В этом компоненте мы сначала импортируем `React` из `'react'`, `Link` из `'next/link'` и `useRouter` из `next/router`.

Внутри компонента мы определяем, соответствует ли текущее имя пути `href` prop компонента, и если так, мы добавляем выбранный класс к дочерним элементам.

Наконец, мы возвращаем эти `children` с обновленным классом, используя `React.createElement` ():

#### components/Link.js

```

import React from "react";
import Link from "next/link";
import { useRouter } from "next/router";

export default ({ href, children }) => {

```

```
const router = useRouter();

let className = children.props.className || "";
if (router.pathname === href) {
  className = `${className} selected`;
}

return <Link href={href}>{React.cloneElement(children, { className })}</Link>;
};
```

## Использование `next/router`

Мы уже видели, как использовать компонент `Link` для декларативной обработки маршрутизации в приложениях Next.js.

Управлять маршрутизацией в JSX очень удобно, но иногда вам нужно запускать изменение маршрутизации программно.

В этом случае вы можете получить прямой доступ к маршрутизатору (роутеру) Next.js, указанному в пакете `next/router`, и вызвать его метод `push ()`.

Вот пример доступа к маршрутизатору:

```
import { useRouter } from 'next/router'

export default () => {
  const router = useRouter()
  //...
}
```

Как только мы получим объект `router`, вызвав `useRouter ()`, мы сможем использовать его методы.

Это маршрутизатор на стороне клиента, поэтому методы должны использоваться только во внешнем коде. Самый простой способ убедиться в этом - заключить вызовы в ловушку `React useEffect ()` или внутри `componentDidMount ()` в компонентах с состоянием React.

Наиболее вероятно, что вы будете использовать чаще всего `push ()` и `prefetch ()`.

`push ()` позволяет нам программно инициировать изменение URL во внешнем интерфейсе:

```
router.push('/login')
```

`prefetch ()` позволяет нам программно выполнять предварительную выборку URL, что полезно, когда у нас нет тега `Link`, который автоматически обрабатывает предварительную выборку для нас:

```
router.prefetch('/login')
```

Полный пример:

```
import { useRouter } from 'next/router'

export default () => {
  const router = useRouter()

  useEffect(() => {
    router.prefetch('/login')
  })
}
```

Вы также можете использовать маршрутизатор для прослушивания событий изменения маршрута - [route change events](#).

## Подача данных в компоненты с помощью `getInitialProps`

В предыдущей главе у нас была проблема с динамическим генерированием страницы публикации, потому что компоненту требовалось некоторые данные заранее, когда мы пытались получить данные из файла JSON:

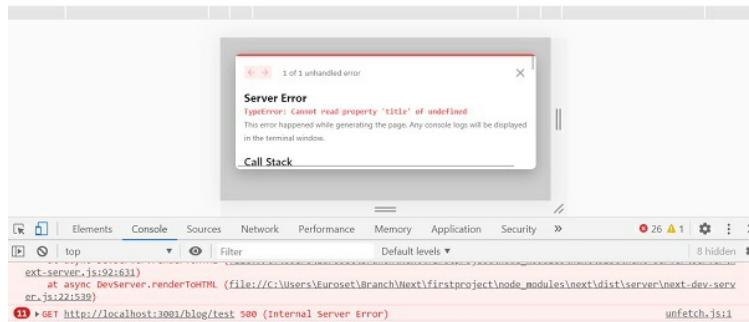
```
import { useRouter } from 'next/router'
import posts from '../posts.json'

export default () => {
  const router = useRouter()

  const post = posts[router.query.id]

  return (
    <>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </>
  )
}
```

мы получили эту ошибку:



Как мы решаем это? И как мы заставляем SSR работать для динамических маршрутов?

Мы должны предоставить компоненту реквизиты-props, используя специальную функцию `getInitialProps ()`, которая прикреплена к компоненту.

Для этого сначала назовем компонент:

```
const Post = () => {
  //...
}

export default Post
```

Затем мы добавляем функцию к нему:

```
const Post = () => {
  //...
}

Post.getInitialProps = () => {
  //...
}

export default Post
```

Эта функция получает объект в качестве аргумента, который содержит несколько свойств. В частности, сейчас нас интересует то, что мы получаем объект запроса (`query object`), который мы использовали ранее для получения идентификатора записи.

Таким образом, мы можем получить его, используя синтаксис деструктуризации объекта:

```
Post.getInitialProps = ({ query }) => {
  //...
}
```

Теперь мы можем вернуть `post` из этой функции:

```
Post.getInitialProps = ({ query }) => {
  return {
    post: posts[query.id]
  }
}
```

И мы также можем удалить импорт `useRouter`, и мы получаем сообщение из свойства `props`, переданного компоненту `Post`:

```
import posts from "../../posts.json";

const Post = (props) => {
  return (
    <div>
      <h1>{props.post.title}</h1>
      <p>{props.post.content}</p>
    </div>
  );
};

Post.getInitialProps = ({ query }) => {
  return {
    post: posts[query.id],
  };
};

export default Post;
```

Теперь ошибки не будет, и SSR будет работать как положено, как вы можете видеть, если посмотреть View Page Source в консоли:

```

<!DOCTYPE html><html><head><style data-next-hide=fouc="true">body{display:none}</style></head><body><noscript data-next-hide=fouc="true"><style>body{display:block}</style></noscript><meta charset="utf-8"/><meta name="viewport" content="width=device-width,initial-scale=1,minimum-scale=1,maximum-scale=1,viewport-fit=cover"><script id="next-data" type="application/json">{"props": {"pageProps": {"post": {"id": "test", "title": "test post", "content": "Hey some post content"}, "query": {"id": "test", "isFallback": false, "gip": true}}}, "page": "/blog/[id]", "query": {"id": "test", "isFallback": false, "gip": true}}</script><script id="__NEXT_DATA__" type="application/json">{"props": {"pageProps": {"post": {"id": "test", "title": "test post", "content": "Hey some post content"}, "query": {"id": "test", "isFallback": false, "gip": true}}}, "page": "/blog/[id]", "query": {"id": "test", "isFallback": false, "gip": true}}</script><script nomodule="" src="/_next/static/runtime/polyfills.js?ts=1592745418820"></script><script async="" data-next-page="/_app" src="/_next/static/runtime/pages/_app.js?ts=1592745418820"></script><script async="" data-next-page="/blog/[id]" src="/_next/static/development/pages/_app.js?ts=1592745418820"></script><script src="/_next/static/runtime/webpack.js?ts=1592745418820" async=""></script><script src="/_next/static/runtime/main.js?ts=1592745418820" as="script"/><script id="__next_css_DONT_USE__" data-next-page="/_next/static/development/_buildManifest.js?ts=1592745418820" as="script"/></script></div><script src="/_next/static/runtime/react-refresh.js?ts=1592745418820"></script><script src="/_next/static/development/dll/dll_f9de5cb31a4a1e1f91e.js?ts=1592745418820"></script><script id="__NEXT_DATA__" type="application/json">{"props": {"pageProps": {"post": {"id": "test", "title": "test post", "content": "Hey some post content"}, "query": {"id": "test", "isFallback": false, "gip": true}}}, "page": "/blog/[id]", "query": {"id": "test", "isFallback": false, "gip": true}}</script><script nomodule="" src="/_next/static/runtime/polyfills.js?ts=1592745418820"></script><script async="" data-next-page="/_app" src="/_next/static/development/pages/_app.js?ts=1592745418820"></script><script async="" data-next-page="/blog/[id]" src="/_next/static/development/pages/_app.js?ts=1592745418820"></script><script src="/_next/static/runtime/webpack.js?ts=1592745418820" as="script"/><script src="/_next/static/runtime/main.js?ts=1592745418820" as="script"/><script src="/_next/static/development/_buildManifest.js?ts=1592745418820" as="script"/></script></body></html>

```

Функция `getInitialProps` будет выполняться на стороне сервера, а также на стороне клиента, когда мы перейдем на новую страницу, используя компонент `Link`, как мы это делали. Важно отметить, что `getInitialProps` получает в объекте контекста, который он получает, в дополнение к объекту запроса - `query` эти другие свойства:

- `pathname: path` раздел пути URL
- `asPath` - Стока фактического пути (включая запрос) отображается в браузере

что в случае вызова <http://localhost:3001/blog/test> приведет соответственно к:

- `/blog/[id]`
- `/blog/test`

И в случае рендеринга на стороне сервера он также получит:

- `req` - HTTP request object (HTTP - объект запроса)
- `res` - HTTP response object (HTTP - объект ответа)
- `err` - error object (объект ошибки)

`req` и `res` будут вам знакомы, если вы выполняли какую-либо кодировку Node.js. Если забыли, то стоит посмотреть [мои статьи по работе с Node.js](#)

## CSS

Как нам оформить компоненты React в Next.js?

У нас для этого много свободы и вариантов, поэтому мы можем использовать любую библиотеку, какую пожелаем.

Но Next.js поставляется со встроенным `styled-jsx`, потому что это библиотека, созданная теми же людьми, которые работают над Next.js

И это довольно крутая библиотека, которая предоставляет нам ограниченный CSS, который отлично подходит для сопровождения, потому что CSS влияет только на компонент, к которому он применяется.

Я думаю, что это отличный подход при написании CSS, без необходимости применять дополнительные библиотеки или препроцессоры, которые увеличивают сложность.

Чтобы добавить CSS в компонент React в Next.js, мы вставляем его во фрагмент кода в JSX, который начинается с

```
<style jsx>`
```

и заканчивается

```
`</style>
```

Внутри этих странных блоков мы пишем простой CSS, как мы это делаем в файле `file.css`:

```
<style jsx>`  
h1 {  
  font-size: 3rem;  
}  
</style>;
```

Вы пишете это внутри JSX, вот так:

```
const Index = () => (  
  <div>  
    <h1>Home page</h1>  
  
    <style jsx>`  
      h1 {  
        font-size: 3rem;  
      }  
    `</style>  
  </div>  
)  
  
export default Index;
```

Внутри блока мы можем использовать интерполяцию для динамического изменения значений. Например, здесь мы предполагаем, что родительский компонент передает размер `-size` в `props`, и мы используем его в блоке `styled-jsx`:

```
const Index = (props) => (  
  <div>  
    <h1>Home page</h1>  
  
    <style jsx>`
```

```
h1 {  
  font-size: ${props.size}rem;  
}  
`}</style>  
</div>  
);
```

Если вы хотите применить некоторые CSS глобально, не ограничивая область действия компонента, вы добавляете ключевое слово `global` к тегу `style`:

```
<style jsx global>`  
body {  
  margin: 0;  
}`</style>
```

Если вы хотите импортировать внешний CSS-файл в компонент Next.js, вы должны сначала установить `@zeit/next-css`:

```
npm install @zeit/next-css  
#or  
yarn add @zeit/next-css
```

а затем создайте файл конфигурации в корневом каталоге проекта с именем `next.config.js` с таким содержимым:

```
const withCSS = require('@zeit/next-css')  
module.exports = withCSS()
```

После перезапуска приложения Next.js вы можете импортировать CSS, как это обычно делается с библиотеками или компонентами JavaScript:

```
import '../style.css'
```

Вы также можете импортировать файл SASS напрямую, используя вместо этого библиотеку `@zeit/next-sass`.

## Заполнение тега `head` пользовательскими тегами

С любого компонента страницы Next.js вы можете добавить информацию в заголовок страницы.

Это удобно, когда:

- Вы хотите настроить заголовок страницы
- Вы хотите изменить метатег

Как ты можешь это сделать?

Внутри каждого компонента вы можете импортировать компонент `Head` из `next/head` и включить его в вывод JSX вашего компонента:

```
import Head from "next/head";  
  
const House = (props) => (  
  <div>  
    <Head>  
      <title>The page title</title>  
    </Head>  
    {/* the rest of the JSX */}  
  </div>  
)  
  
export default House;
```

Вы можете добавить любой HTML-тег, который хотите отображать в разделе `<head>` страницы.

При монтировании компонента Next.js будет следить за тем, чтобы теги внутри `Head` добавлялись в заголовок страницы. То же самое при размонтировании компонента, Next.js позаботится об удалении этих тегов.

## Добавление компонента-оболочки

Все страницы на вашем сайте выглядят более или менее одинаково. Там есть окно браузера, общий базовый слой, и вы просто хотите изменить то, что внутри.

Там есть навигационная панель, боковая панель, а затем фактический контент.

Как вы строите такую систему в Next.js?

Есть 2 способа. Один из них использует компонент высшего порядка (HOC - [Higher Order Component](#), создав компонент `component/Layout.js`:

`component/Layout.js`

```
export default (Page) => {  
  return () => (  
    <div>  
      <nav>  
        <ul>....</ul>
```

```
</hav>
<main>
  <Page />
</main>
</div>
);
};
```

Там мы можем импортировать отдельные компоненты для заголовка `heading` и / или боковой панели - `sidebar`, а также можем добавить весь необходимый нам CSS.

И вы используете его на каждой странице, как это:

```
import withLayout from "../components/Layout.js";
const Page = () => <p>Here's a page!</p>;
export default withLayout(Page);
```

Но я обнаружил, что это работает только для простых случаев, когда вам не нужно вызывать `getInitialProps ()` на странице.

Почему?

Потому что `getInitialProps ()` вызывается только на компоненте страницы. Но если мы экспортим Компонент Высшего Порядка с помощью `Layout ()` со страницы, `Page.getInitialProps ()` не вызывается, `withLayout.getInitialProps ()` будет.

Чтобы избежать ненужного усложнения нашей кодовой базы, альтернативный подход заключается в использовании `props`:

```
export default (props) => (
  <div>
    <nav>
      <ul>....</ul>
    </nav>
    <main>{props.content}</main>
  </div>
);
```

и теперь на наших страницах мы используем это так:

```
import Layout from "../components/Layout.js";
const Page = () => <Layout content={<p>Here's a page!</p>} />;
```

Этот подход позволяет нам использовать `getInitialProps ()` из нашего компонента страницы, с единственным недостатком - писать JSX компонента внутри `content props`:

```
import Layout from "../components/Layout.js";
const Page = () => <Layout content={>p>Here's a page!</p>} />;
Page.getInitialProps = ({ query }) => {
  //...
};
```

## API-маршруты

Помимо создания маршрутов страниц, что означает, что страницы передаются в браузер как веб-страницы, Next.js может создавать маршруты API.

Это очень интересная функция, поскольку она означает, что Next.js можно использовать для создания внешнего интерфейса для данных, которые хранятся и извлекаются самим Next.js, передавая JSON через запросы выборки.

Маршруты API (API routes) находятся в папке `/pages/api/` и сопоставляются с конечной точкой `/api`.

Эта функция очень полезна при создании приложений.

На этих маршрутах мы пишем код Node.js (а не код React). Это смена парадигмы, вы переходите с внешнего интерфейса на внутренний, но очень плавно.

Предположим, у вас есть файл `/pages/api/comments.js`, целью которого является возвращение комментариев к записи блога в формате JSON.

Допустим, у вас есть список комментариев, хранящихся в файле `comments.json`:

```
/pages/api/comments.js
```

```
[{
  "comment": "First"
},
{
  "comment": "Nice post"
}]
```

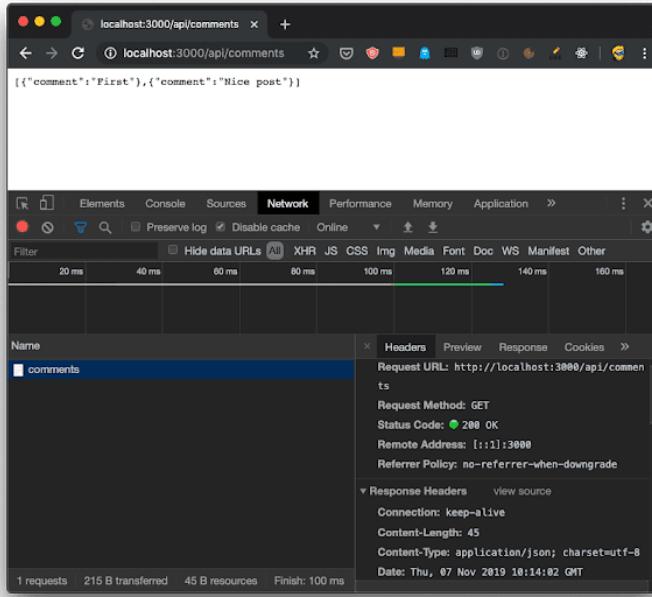
Вот пример кода, который возвращает клиенту список комментариев:

```
/pages/api/comments.js
```

```
import comments from './comments.json'
```

```
export default (req, res) => {
  res.status(200).json(comments)
}
```

Он будет прослушивать URL-адрес `/api/comments` для запросов GET, и вы можете попробовать вызвать его с помощью браузера:



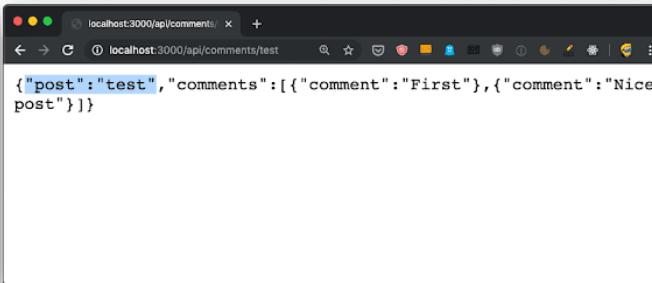
Маршруты API также могут использовать динамическую маршрутизацию, например страницы, использовать синтаксис `[]` для создания динамического маршрута API, например `/pages/api/comments/[id].js`, который будет извлекать комментарии, относящиеся к идентификатору записи.

Внутри `[id].js` вы можете получить значение `id`, посмотрев его внутри объекта `req.query`:

```
import comments from '../comments.json'

export default (req, res) => {
  res.status(200).json({ post: req.query.id, comments })
}
```

Вот вы можете увидеть приведенный выше код в действии:



На динамических страницах вам нужно будет импортировать `useRouter` из `next/router`, затем получить объект `router` с помощью `const router = useRouter()`, и тогда мы сможем получить значение `id` с помощью `router.query.id`.

На стороне сервера все проще, так как запрос привязан к объекту запроса.

Если вы делаете запрос POST, все работает одинаково - все проходит через экспорт по умолчанию.

Чтобы отделить POST от GET и других методов HTTP (PUT, DELETE), найдите значение `req.method`:

```
export default (req, res) => {
  switch (req.method) {
    case 'GET':
      //...
      break
    case 'POST':
      //...
      break
    default:
      res.status(405).end() //Method Not Allowed
      break
  }
}
```

В дополнение к `req.query` и `req.method`, которые мы уже видели, мы имеем доступ к cookie-файлам, ссылаясь на `req.cookies`, тело запроса в `req.body`.

Под капотом все это работает на Micro, библиотеке, которая поддерживает асинхронные HTTP-микросервисы, созданной той же командой, которая создала Next.js.

Вы можете использовать любое промежуточное ПО [Micro](#) в наших маршрутах API, чтобы добавить больше функциональности.

## Выполнять код только на стороне сервера или на стороне клиента

В ваших компонентах страницы вы можете выполнить код только на стороне сервера или на стороне клиента, проверив свойство окна.

Это свойство существует только внутри браузера, поэтому вы можете проверить

```
if (typeof window === 'undefined') {  
}
```

и добавьте серверный код в этот блок.

Точно так же вы можете выполнить код на стороне клиента, только проверив

```
if (typeof window !== 'undefined') {  
}
```

JS Совет: здесь мы используем оператор `typeof`, потому что мы не можем обнаружить значение, которое не определено другими способами. Мы не можем сделать `if (window === undefined)`, потому что мы получили бы ошибку "окно не определено" - ("window is not defined" runtime error) во время выполнения.

Next.js, как оптимизация во время сборки, также удаляет код, который использует эти проверки из пакетов. Пакет на стороне клиента не будет включать содержимое, заключенное в блок `if (typeof window === 'undefined') {}`.

## Развертывание-Deploying рабочей версии

Развертывание приложения всегда остается последним в руководствах.

Здесь я хочу представить его на ранней стадии, просто потому, что развернуть приложение Next.js настолько просто, что мы можем погрузиться в него сейчас, а затем перейти к другим более сложным темам.

Помните, в главе [«Как установить Next.js»](#) я говорил вам добавить эти 3 строки в раздел скрипта `package.json`:

```
"scripts": {  
  "dev": "next",  
  "build": "next build",  
  "start": "next start"  
}
```

Мы использовали `npm run dev` до сих пор, чтобы вызвать следующую локально установленную команду в `node_modules/next/dist/bin/next`. Это запустило сервер разработки, который предоставил нам исходные карты и горячую перезагрузку кода, две очень полезные функции при отладке.

Эту же команду можно вызвать для создания веб-сайта с флагом сборки, запустив команду `npm run build`. Затем эту же команду можно использовать для запуска производственного приложения, передающего флаг запуска, путем запуска `npm run start`.

Эти 2 команды - те, которые мы должны вызывать для успешного развертывания рабочей версии нашего сайта локально. Рабочая версия высоко оптимизирована и не содержит исходных карт и других вещей, таких как горячая перезагрузка кода, которая не будет полезна для наших конечных пользователей.

Итак, давайте создадим производственное развертывание нашего приложения. Постройте это, используя:

```
npm run build
```

```
Euroset@Home MINGW64 ~/Branch/Next/firstproject (master)  
$ yarn run build  
yarn run v1.22.4  
$ next build  
Creating an optimized production build  
Compiled successfully.  
Automatically optimizing pages  
Page Size First Load  
ad.js 1.76 kB 60  
404 3.25 kB 61  
blog 1.89 kB 60  
/blog 420 B 59  
/blog/[id] 1 kB  
First Load JS shared by all 58.7 kB  
static/js/2d11a2638277c3844a8d011a7f28db706714.36a881.js 38.7 kB  
chunks/framework.c6faae.js 40 kB  
runtime/main.99661a.js 6.28 kB  
runtime/webpack.c21266.js 746 B  
A (Server) server-side renders at runtime (uses getInitialProps or getServerSideProps)  
P (Static) automatically rendered as static HTML (uses no initial props)  
P (SSG) automatically generated as static HTML + JSON (uses getStaticProps)  
Done in 39.39s.  
Euroset@Home MINGW64 ~/Branch/Next/firstproject (master)  
$
```

Выходные данные команды говорят нам, что некоторые маршруты (/ и /blog теперь представлены в виде статического HTML, а /blog/[id] будет обслуживаться серверной частью Node.js).

Затем вы можете запустить `npm run start` для локального запуска рабочего сервера:

```
Done in 39.39s.  
Euroset@Home MINGW64 ~/Branch/Next/firstproject (master)  
$ yarn run start  
yarn run v1.22.4  
$ next start  
ready - started server on http://localhost:3000
```

Идем на <http://localhost:3000> (кстати, обратите внимание, что именно 3000, а не 3001) и посмотрим нам локальную версию приложения.

## Развертывание на Now

В предыдущей главе мы развернули приложение Next.js локально.

Как мы развернем его на реальном веб-сервере, чтобы другие люди могли получить к нему доступ?

Один из самых простых способов развертывания приложения Next - через платформу Now, созданную Zeit, той же компанией, которая создала проект Open Source Next.js. Вы можете использовать сейчас для развертывания приложений Node.js, статических веб-сайтов и многоного другого.

Теперь процесс развертывания и распространения приложения становится очень, очень простым и быстрым, и в дополнение к приложениям Node.js они также поддерживают развертывание Go, PHP, Python и других языков.

Вы можете думать об этом как о «облаке», поскольку вы на самом деле не знаете, где будет развернуто ваше приложение, но вы знаете, что у вас будет URL-адрес, по которому вы сможете добраться до него.

Теперь вы можете начать пользоваться бесплатным щедрым планом, который в настоящее время включает 100 ГБ хостинга, 1000 вызовов функций без сервера в день, 1000 сборок в месяц, 100 ГБ пропускной способности в месяц и одно местоположение CDN. Страница с ценами помогает получить представление [о расходах](#), если вам нужно больше.

Лучший способ начать использовать Now - использовать официальный CLI Now:

```
npm install -g now
```

Как только команда станет доступна, запустите

```
now login
```

и приложение спросит вас о вашей электронной почте.

Если вы еще не зарегистрировались, создайте учетную запись на <https://zeit.co/signup>, прежде чем продолжить, а затем добавьте свою электронную почту в клиент CLI.

После этого из корневой папки проекта Next.js запустите

```
now
```

и приложение будет немедленно развернуто в облаке Now, и вам будет предоставлен уникальный URL-адрес приложения:

```
firstproject — now /Users/flaviocopes/dev/nextjs/firstproject — node /usr/local/bin/now — 75x12  
→ firstproject now  
> Deploying ~/dev/nextjs/firstproject under flaviocopes  
> Using project firstproject  
> Synced 7 files [962ms]  
> NOTE: This is the first deployment in the firstproject project. It will be promoted to production.  
> NOTE: To deploy to production in the future, run `now --prod`.  
> https://firstproject-2pv7khwrr.now.sh [5s]  
> Ready! Deployment complete [32s]  
- https://firstproject-sepia-ten.now.sh  
- https://firstproject.flaviocopes.now.sh [in clipboard]
```

Почему так много?

Первый - это URL, идентифицирующий развертывание. Каждый раз, когда мы разворачиваем приложение, этот URL будет меняться.

Вы можете протестировать сразу, изменив что-то в коде проекта и снова запустив сейчас:

```

firstproject now
> Deploying ~/dev/nextjs/firstproject under flaviocopes
> Using project firstproject
> Synced 1 file [982ms]
> https://firstproject-rdcwfkwt6.now.sh [4s]
> Ready! Deployed to https://firstproject.flaviocopes.now.sh [in clipboard]
[38s]

```

Другие 2 URL не изменятся. Первый случайный, второй - имя вашего проекта (по умолчанию это папка текущего проекта, имя вашей учетной записи, а затем now.sh).

Если вы посетите URL, вы увидите приложение, развернутое в производство.

## second post

Hey this is the second post content

Вы можете настроить сейчас, чтобы обслуживать сайт в своем собственном домене или поддомене, но я не буду сейчас углубляться в это.

Субдомена `now.sh` достаточно для тестирования.

### Анализируем комплекты приложений

Next.js предоставляет нам способ проанализировать сгенерированные пакеты кода.

Откройте файл `package.json` приложения и в разделе скриптов добавьте эти 3 новые команды:

```
"analyze": "cross-env ANALYZE=true next build",
"analyze:server": "cross-env BUNDLE_ANALYZE=server next build",
"analyze:browser": "cross-env BUNDLE_ANALYZE=browser next build"
```

Вот так:

```
{
  "name": "firstproject",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "scripts": {
    "dev": "next -p 3001",
    "build": "next build",
    "start": "next start",
    "analyze": "cross-env ANALYZE=true next build",
    "analyze:server": "cross-env BUNDLE_ANALYZE=server next build",
    "analyze:browser": "cross-env BUNDLE_ANALYZE=browser next build"
  },
  "dependencies": {
    "next": "^9.4.4",
    "react": "^16.13.1",
    "react-dom": "^16.13.1"
  }
}
```

затем установите эти 2 пакета:

```
npm install --dev cross-env @next/bundle-analyzer
#or
yarn add cross-env @next/bundle-analyzer --dev
```

Создайте файл `next.config.js` в корне проекта со следующим содержимым:

```
next.config.js
const withBundleAnalyzer = require("@next/bundle-analyzer")({
  enabled: process.env.ANALYZE === "true",
});

module.exports = withBundleAnalyzer({});
```

Теперь запустим:

```
npm run analyze  
# or  
yarn run analyze
```

```
uroset@Home MINGW64 ~/Branch/Next/firstproject (master)
$ npx nextjs --analyze
yarn run v1.22.4
$ cross-env ANALYZE=true next build
webpack Bundle Analyzer saved report to C:\Users\Uroset\Branch\Next\firstproject\next\analyze\server.html
webpack Bundle Analyzer saved report to C:\Users\Uroset\Branch\Next\firstproject\next\analyze\client.html
Creating an optimized production build
Compiled successfully.

Automatically optimizing pages

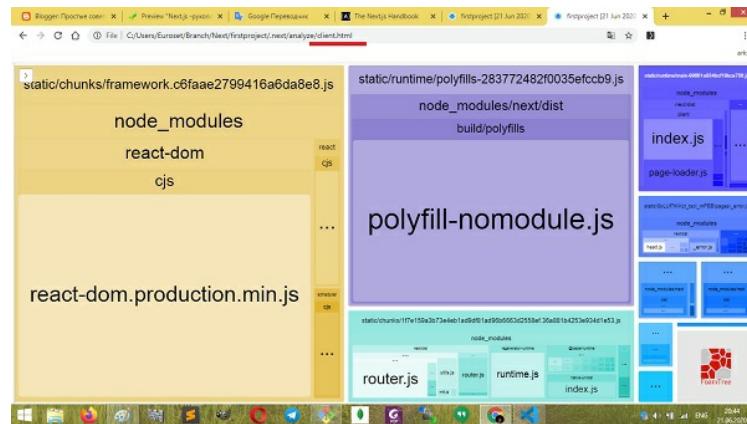
Page                                         Size    First Load JS
[ o /                                          1.76 kB   60.4 kB
[ o /404                                       3.25 kB   61.9 kB
[ o /page                                      1.93 kB   60.6 kB
[ x /blog/[id]                                 420 B    59.1 kB
+ First Load JS shared by all                58.7 kB
  static/pages/_app.js                         60 kB
  static/chunks/main.159a30d3c4eb1ad9f81ad96b6663d2558ef.36a881.js 10.7 kB
  static/chunks/framework.cbf9ae.js            40 kB
  runtime/main.99661a.js                        6.28 kB
  runtime/webpack.c21266.js                     746 B

A (Server) server-side renders at runtime (uses getInitialProps or getServerSideProps)
P (Static) automatically rendered as static HTML (uses no initial props)
● (SSG) automatically generated as static HTML + JSON (uses getStaticProps)

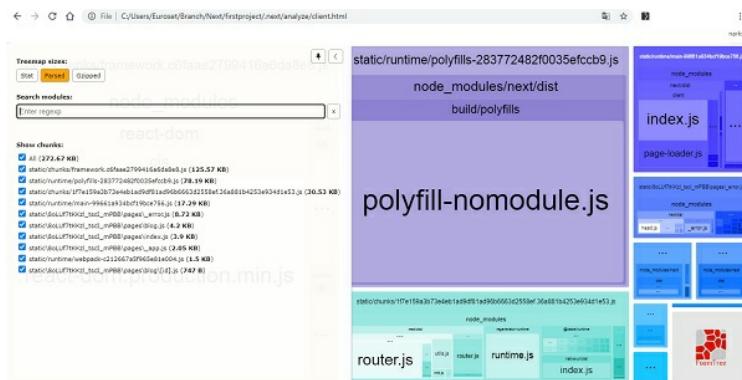
Done in 16.42s.

uroset@Home MINGW64 ~/Branch/Next/firstproject (master)
$ |
```

Это должно открыть 2 страницы в браузере. Одну для клиентских пакетов и другую для серверных.



Это невероятно полезно. Вы можете проверить, что занимает больше всего места в пакетах, а также использовать боковую панель для исключения пакетов, чтобы упростить визуализацию меньших:



## Ленивая загрузка модулей.

Возможность визуально анализировать пакет великолепна, потому что мы можем очень легко оптимизировать наше приложение.

Скажем, нам нужно загрузить библиотеку Moment в наши записи в блоге. Погнали:

```
npm install moment  
# or  
yarn add moment
```

включить его в проект.

Теперь давайте смоделируем тот факт, что он нам нужен на двух разных маршрутах: `/blog` и `/blog/[id]`.

Мы импортируем его в `pages/blog/[id].js`:

```
import moment from 'moment'  
  
...  
  
const Post = props => {  
  return (  
    <div>  
      <h1>{props.post.title}</h1>  
      <p>Published on {moment().format('ddd D MMMM YYYY')}</p>  
      <p>{props.post.content}</p>  
    </div>  
  )  
}
```

Я просто добавляю сегодняшнюю дату, как пример.

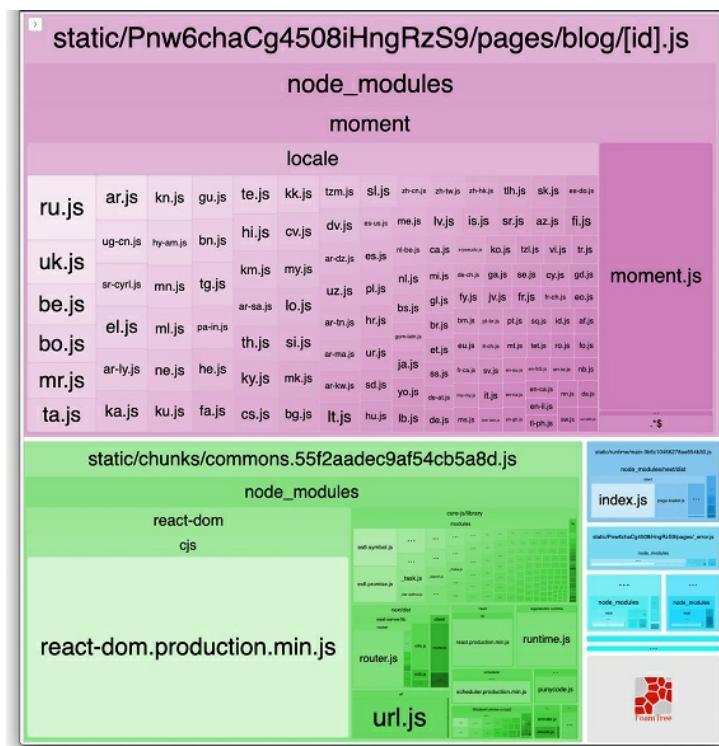
Это будет включать `Moment.js` в комплект страницы поста блога, как вы можете увидеть, запустив `npm run analyze`:

```
Euroset@Home MINGW64 ~/Branch/Next/firstproject (master)  
$ yarn run analyze  
yarn run v1.22.10  
$ cross-env ANALYZE=true next build  
webpack Bundle Analyzer saved report to C:\Users\Euroset\Branch\Next\firstproject\.next\analyze\server.html  
webpack Bundle Analyzer saved report to C:\Users\Euroset\Branch\Next\firstproject\.next\analyze\client.html  
Creating an optimized production build  
Compiled successfully.  
Automatically optimizing pages  
Page Size First Load JS  
└─ 404 1.76 kB 60.4 kB  
└─ /blog 3.25 kB 61.9 kB  
└─ /blog/[id] 1.89 kB 60.6 kB  
  + First Load JS shared by all 80.4 kB 139 kB  
    + static/pages/_app.js 18.7 kB  
    | chunks/_app/ab369637b061de8b198d08c9cbc.36a881.js 984 B  
    | chunks/_framework/c6fiae.js 10 kB  
    | runtime/main.72caef.js 40 kB  
    | runtime/webpack.c21266.js 6.28 kB  
    + runtime/webpack.c21266.js 746 B  
  ↳ (Server) server-side renders at runtime (uses getInitialProps or getServerSideProps)  
  ↳ (Static) automatically rendered as static HTML (uses no initial props)  
  • (SSG) automatically generated as static HTML + JSON (uses getStaticProps)  
Done in 31.80s.  
Euroset@Home MINGW64 ~/Branch/Next/firstproject (master)
```

Обратите внимание, что теперь у нас есть желтая запись в `/blog/[id]`, маршрут, к которому мы добавили `Moment.js`!

Он довольно сильно увеличился. И это потому, что сама библиотека `Moment.js` имеет размер 349 КБ.

Визуализация клиентских пакетов теперь показывает нам, что больший пакет - это страница, которой раньше было очень маленькой. И 99% его кода - это `Moment.js`.



Каждый раз, когда мы загружаем сообщение в блоге, мы собираемся передать весь этот код клиенту. Что не идеально.

Одним из исправлений было бы поискать библиотеку меньшего размера, поскольку **Moment.js** не известен своей легковесностью (особенно из коробки со всеми включенными локалями), но давайте предположим ради примера, что мы должны использовать именно её.

Вместо этого мы можем отдельить весь код **Moment.js** в отдельный пакет.

Как? Вместо того, чтобы импортировать Moment на уровне компонента, мы выполняем асинхронный импорт внутри `getInitialProps` и вычисляем значение для отправки компоненту.

Помните, что мы не можем вернуть сложные объекты внутри возвращенного объекта `getInitialProps ()`, поэтому мы вычисляем дату внутри него:

```
import posts from "../../posts.json";

const Post = (props) => {
  return (
    <div>
      <h1>{props.post.title}</h1>
      <p>Published on {props.date}</p>
      <p>{props.post.content}</p>
    </div>
  );
};

Post.getInitialProps = async ({ query }) => {
  const moment = (await import("moment")).default();
  return {
    date: moment.format("ddd D MMMM YYYY"),
    post: posts[query.id],
  };
};

export default Post;
```

Видите этот специальный вызов `.default ()` после импорта? Это необходимо для экспорт по умолчанию в динамическом импорте (см. [Https://v8.dev/features/dynamic-import](https://v8.dev/features/dynamic-import))

Теперь, если мы снова запустим `npm run analyze`, мы увидим это:

```
laptop:home MINGW64 ~/Branch/Next/firstproject (master)
$ yarn run analyze
yarn run v1.22.4
$ cross-env ANALYZE=true next build
$ npx next-analyzer saved report to C:\Users\Euroset\Branch\Next\firstproject\.next\analyze\server.html
Webpack Bundle Analyzer generated report to C:\Users\Euroset\Branch\Next\firstproject\.next\analyze\client.html
Creating an optimized production build
Compiled successfully.

automatically optimizing pages

Page          Size     First Load JS
- /           1.77 kB   61.5 kB
- 404        2.57 kB   62.3 kB
- /blog/100  39.9 kB   61.6 kB
- /blog/[id] 734 kB   60.4 kB

First Load JS shared by all
static/pages/_app.js          985 B
static/pages/_error.js        1.22 kB
static/chunks/main.2f9b2f9113190da20dd0f78c283.3cf1d0.js 59.7 kB
static/chunks/commons.a0e556.js 9.8 kB
static/chunks/framework.e84fab.js 40 kB
static/runtime/main.cfaaac.js 6.28 kB
static/runtime/webpack.0a3420.js 1.21 kB

(Server) server-side renders at runtime (uses getInitialProps or getServerSideProps)
(Static) automatically rendered as static HTML (uses no initial props)
(SSG) automatically generated as static HTML + JSON (uses getStaticProps)
done in 37.62s.
```

Наш пакет `/blog/[id]` снова очень маленький, так как Moment был перемещен в собственный файл пакета, загружаемый браузером отдельно.

## Куда идти дальше?

Есть еще много информации о Next.js. Я не говорил об управлении сессиями пользователей с помощью входа в систему, без сервера, управлении базами данных и так далее.

Цель этого Руководства не состоит в том, чтобы научить вас всему, но вместо этого оно призвано постепенно познакомить вас со всей мощью Next.js.

Следующий шаг, который я рекомендую, - внимательно прочитать официальную документацию Next.js, чтобы узнать больше обо всех функциях и возможностях, о которых я не говорил, и взглянуть на все дополнительные функции, представленные плагинами Next.js, некоторые из которых довольно удивительны.

Хотите освоить самые современные методы написания React приложений? Надоели простые проекты? Нужны курсы, книги, руководства, индивидуальные занятия по React и не только? Хотите стать разработчиком полного цикла, освоить стек MERN, или вы только начинаете свой путь в программировании, и не знаете с чего начать, то пишите через форму связи, подписывайтесь на мой канал в Телеге, вступайте в группу на Facebook.

Удачного кодирования!

Выражая особую благодарность [Flavio Copes](#), который вдохновил меня на этот труд.



Телеграм канал - Full Stack JavaScript Developer

Помочь проекту (любая валюта).

DONATE

- [Введение.](#)
- [Основные функции, предоставляемые Next.js](#)
- [Next.js vs Gatsby vs create-react-app](#)

- [Как установить Next.js](#)
- [Просмотр источника, чтобы подтвердить работу SSR](#)
- [Пакеты приложений](#)
- [Что это за иконка справа внизу?](#)
- [Установка React Developer Tools](#)
- [Другие методы отладки, которые вы можете использовать](#)
- [Добавление второй страницы на сайт](#)
- [Связывание двух страниц](#)
- [Динамический контент с роутером](#)
- [Предзагрузка](#)
- [Использование роутера для обнаружения активной ссылки](#)
- [Использование `next/router`](#)
- [Подача данных в компоненты с помощью `getInitialProps`](#)
- [CSS](#)
- [Заполнение тега `head` пользовательскими тегами](#)
- [Добавление компонента-оболочки](#)
- [API-маршруты](#)
- [Выполнять код только на стороне сервера или на стороне клиента](#)
- [Развертывание-Deploying рабочей версии](#)
- [Развертывание на Now](#)
- [Анализируем комплексы приложений](#)
- [Ленивая загрузка модулей](#)
- [Куда идти дальше?](#)

Это руководство идеально подойдет для вас, если у вас практически нет знаний о Next.js, или если вы уже использовали React в прошлом и хотите больше углубиться в экосистему React, в частности, рендеринг на стороне сервера.

Я считаю Next.js отличным инструментом для создания веб-приложений, и в конце этого поста я надеюсь, что вы будете хорошо понимать его принципы работы и сможете создавать собственные приложения. И я надеюсь, что это поможет вам изучить Next.js!

## Введение

Работать над современным JavaScript-приложением на основе React - это круто, пока вы не поймете, что есть пара проблем, связанных с отображением всего контента на стороне клиента.

Во-первых, для того, чтобы страница стала видимой пользователю, требуется больше времени, потому что перед загрузкой контента должен загрузиться весь JavaScript, и ваше приложение должно быть запущено, чтобы определить, что показывать на странице.

Во-вторых, если вы создаете общедоступный веб-сайт, у вас есть проблема с SEO контента. Поисковые системы хорошоправляются с запуском и индексацией приложений JavaScript, но гораздо лучше, если мы сможем отправить им контент, а не дать им самим понимать его (роботы паучки и пр :‐) ).

Решением обеих этих проблем является рендеринг на стороне сервера, также называемый статическим предварительным рендерингом.

[Next.js](#) - это одна из сред React, которая делает все это очень простым способом, но не ограничивается этим. Создатели рекламируют его как набор инструментов с **одной командой для приложений React с нулевой конфигурацией**.

Он обеспечивает общую структуру, которая позволяет легко создавать приложение React с внешним интерфейсом, и прозрачно обрабатывает рендеринг на стороне сервера.

## Основные функции, предоставляемые Next.js

Вот неполный список основных функций Next.js:

### 1. Hot Code Reloading

Т.е. Горячая перезагрузка кода.  
Next.js перезагружает страницу, когда обнаруживает любые изменения, сохраненные на диске.

### 2. Automatic Routing

Т.е. Автоматическая маршрутизация  
Любой URL-адрес сопоставляется с файловой системой, с файлами, помещаемыми в папку страниц - `pages`, и вам не нужно ничего настраивать (конечно, у вас есть опции настройки).

### 3. Single File Components

Т.е. Компоненты одного файла  
Используя `styled-jsx`, полностью интегрированный и созданный той же командой, trivialно добавить стили, ограниченные компонентом.

### 4. Server Rendering

Рендеринг на строне сервера  
Вы можете визуализировать компоненты React на строне сервера перед отправкой HTML клиенту.

### 5. Ecosystem Compatibility

Совместимость экосистемы  
Next.js хорошо сочетается с остальной частью экосистемы JavaScript, Node и React.

### 6. Automatic Code Splitting

Автоматическое разделение кода  
Страницы отображаются только с теми библиотеками и JavaScript, которые им нужны, не более. Вместо создания одного единственного файла JavaScript, содержащего весь код приложения, приложение автоматически разбивается на Next.js в нескольких различных ресурсах.

При открытии страницы, Next.js загружает только тот JavaScript, который необходим для этой конкретной страницы.

Next.js делает это путем анализа импортируемых ресурсов.

Например, если только одна из ваших страниц импортирует библиотеку `Axios`, эта конкретная страница будет включать библиотеку в свой пакет.

Это гарантирует, что ваша загрузка первой страницы будет настолько быстрой, насколько это возможно, и только будущие загрузки страниц (если они когда-либо будут запущены) отправят JavaScript, необходимый клиенту.

Есть одно заметное исключение. *Часто используемые операции импорта перемещаются в основной пакет JavaScript, если они используются по крайней мере на половине страниц сайта.*

## 7. Prefetching

Предзагрузка

Компонент [Link](#), используемый для связывания вместе разных страниц, поддерживает функцию предварительной выборки, которая автоматически выполняет предварительную выборку ресурсов страницы (включая код, отсутствующий из-за разделения кода) в фоновом режиме.

## 8. Dynamic Components

Динамические Компоненты

Вы можете импортировать модули JavaScript и React Components динамически.

## 9. Static Exports

Статический экспорт

Используя следующую команду экспорта - `next export`, Next.js позволяет вам экспортировать полностью статический сайт из вашего приложения.

## 10. TypeScript Support

Поддержка TypeScript

Next.js написан на TypeScript и как таковой имеет отличную поддержку TypeScript.

## Next.js vs Gatsby vs [create-react-app](#)

Next.js, [Gatsby](#) и [create-react-app](#) - это удивительные инструменты, которые мы можем использовать для разработки наших приложений.

Давайте сначала скажем, что у них общего. У всех есть React под капотом, который обеспечивает весь опыт разработки. Они также абстрагируют веб-пакет и все те вещи низкого уровня, которые мы использовали для ручной настройки в старые добрые времена.

[create-react-app](#) не поможет вам легко создать приложение на стороне сервера. Все, что идет с ним (SEO, скорость ...), предоставляется только такими инструментами, как Next.js и Gatsby.

### Когда Next.js лучше, чем Гэтсби?

Они оба могут помочь с рендерингом на стороне сервера, но двумя разными способами.

Конечным результатом использования Gatsby является генератор статических сайтов без сервера. Вы создаете сайт, а затем статически развертываете результат процесса сборки на Netlify или другом статическом хостинге.

Next.js предоставляет серверную часть, которая может на стороне сервера отображать ответ на запрос, позволяя вам создать динамический веб-сайт, что означает, что вы развернете его на платформе, которая может запускать Node.js.

Next.js также может генерировать статический сайт, но я бы не сказал, что это его основной вариант использования.

Если бы моей целью было создание статического сайта, мне было бы трудно выбрать, и, возможно, у Гэтсби есть лучшая экосистема плагинов, в том числе и для блогов, в частности.

Gatsby также в значительной степени основан на [GraphQL](#), что вам может действительно нравиться или не нравиться в зависимости от ваших мнений и потребностей.

## Как установить Next.js

Чтобы установить Next.js, у вас должен быть установлен Node.js.

Убедитесь, что у вас установлена последняя версия Node. Проверьте, запустив в своем терминале `node -v`, и сравните его с последней версией LTS, указанной на <https://nodejs.org/>.

После установки Node.js в командной строке будет доступна команда `npm`.

Если у вас возникли проблемы на этом этапе, я рекомендую следующие уроки, которые я написал для вас [JS уроки на моем сайте](#)

Теперь, когда у вас есть Node, обновленный до последней версии, и npm, мы настроены!

Сейчас мы можем выбрать 2 варианта установки Next.js:

1. Используя [create-next-app](#)
2. Классический подход, который предполагает установку и настройку приложения Next вручную.

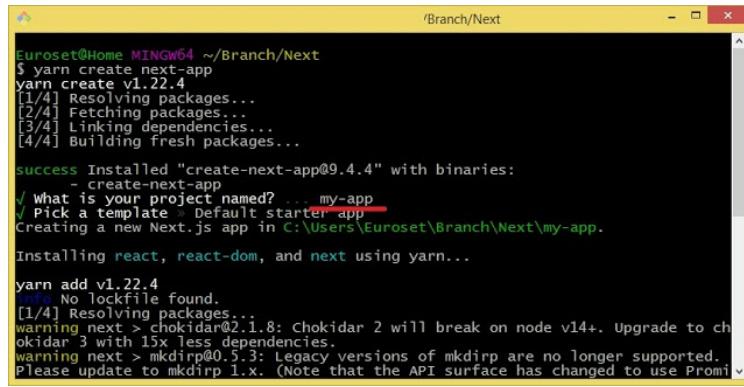
### Установка с [create-next-app](#)

Если вы знакомы с [create-react-app](#), то [create-next-app](#) - это то же самое, за исключением того, что оно создает приложение «Next» вместо приложения «Реакт», как следует из названия.

Я предполагаю, что вы уже установили Node.js, который поставлялся в комплекте с командой prx. Этот удобный инструмент позволяет нам загружать и выполнять команду JavaScript, и мы будем использовать ее следующим образом:

```
prx create-next-app  
// или  
yarn create next-app
```

Команда запрашивает имя приложения (и создает для вас новую папку с таким именем - введем имя, в моем случае это **my-app**, и нажмем **Enter**), согласимся с установкой стартового пакета по умолчанию (**Default starter app** и нажмем **Enter**), и загрузим все необходимые пакеты (**react, react-dom, next**).



```
Euroset@Home MINGW64 ~/Branch/Next
$ yarn create next-app
yarn create v1.22.4
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

success Installed "create-next-app@0.4.4" with binaries:
  - create-next-app
  - What is your project named? ... my-app
  - Pick a template » Default starter app
Creating a new Next.js app in C:\Users\Euroset\Branch\Next\my-app.

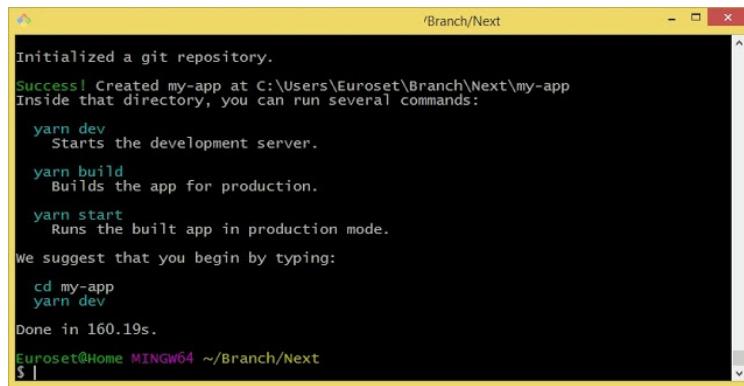
Installing react, react-dom, and next using yarn...

yarn add v1.22.4
  info No lockfile found.
[1/4] Resolving packages...
warning next > chokidar@2.1.8: Chokidar 2 will break on node v14+. Upgrade to chokidar 3 with 15x less dependencies.
warning next > mkdirp@0.5.3: Legacy versions of mkdirp are no longer supported.
Please update to mkdirp 1.x. (Note that the API surface has changed to use Promises)

Done in 160.19s.
```

Я использую yarn, но это не принципиально. Вы можете тоже самое делать с npm, с несколько другими командами. В руководстве я их приведу, наряду с командами для yarn.

Если все прошло нормально, то в конце установки вы увидите такие команды:



```
Initialized a git repository.

Success! Created my-app at C:\Users\Euroset\Branch\Next\my-app
Inside that directory, you can run several commands:

  yarn dev
    Starts the development server.

  yarn build
    Builds the app for production.

  yarn start
    Runs the built app in production mode.

we suggest that you begin by typing:
  cd my-app
  yarn dev

Done in 160.19s.
```

Если сейчас открыть файл **package.json** нашего проекта, то мы увидим там все установленные зависимости и скрипты, с помощью которых мы будем управлять приложением.

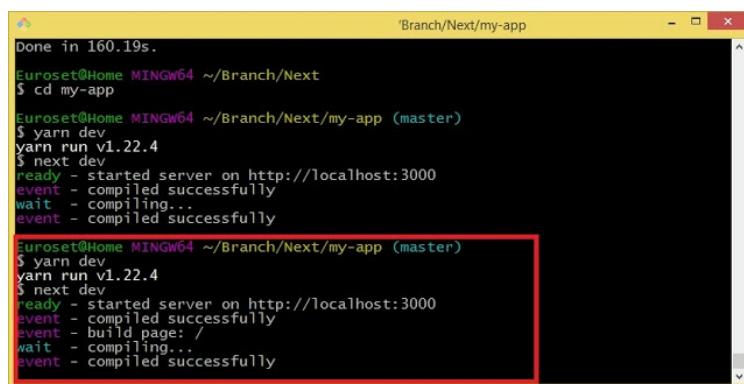
**package.json**

```
{
  "name": "my-app",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start"
  },
  "dependencies": {
    "next": "9.4.4",
    "react": "16.13.1",
    "react-dom": "16.13.1"
  }
}
```

Давайте перейдем в папку проекта и запустим наше приложение:

```
cd my-app
yarn dev
// или
npm dev
```

В консоли увидим:

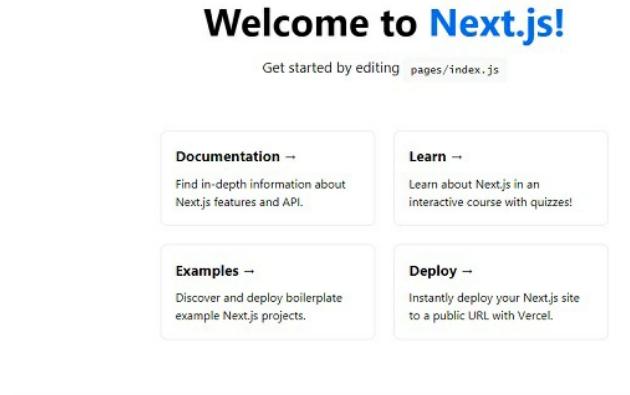


```
Done in 160.19s.

Euroset@Home MINGW64 ~/Branch/Next
$ cd my-app
Euroset@Home MINGW64 ~/Branch/Next/my-app (master)
$ yarn dev
yarn run v1.22.4
$ next dev
ready - started server on http://localhost:3000
event - compiled successfully
wait - compiling...
event - compiled successfully

Euroset@Home MINGW64 ~/Branch/Next/my-app (master)
$ yarn dev
yarn run v1.22.4
$ next dev
ready - started server on http://localhost:3000
event - compiled successfully
event - build page: /
wait - compiling...
event - compiled successfully
```

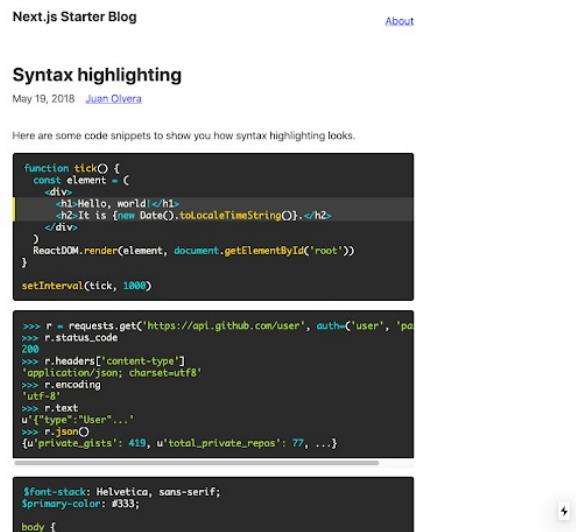
Если перейдем на <http://localhost:3000/>, то увидим, что стартовое приложение запустилось!



Это рекомендуемый способ запуска приложения Next.js, поскольку он дает вам структуру и пример кода для игры. Это больше, чем просто пример приложения по умолчанию; Вы можете использовать любой из примеров, хранящихся по адресу <https://github.com/zeit/next.js/tree/canary/examples>, используя параметр --example. Например попробуйте **blog-starter**:

```
npx create-next-app --example blog-starter blog-starter-app  
# or  
yarn create next-app --example blog-starter blog-starter-app
```

Это даст вам сразу используемый экземпляр блога с подсветкой синтаксиса:



## Создание приложения Next.js вручную

Вы можете избежать создания `create-next-app` приложения, если вы хотите создать следующее приложение с нуля. Вот как: создайте пустую папку где угодно, например, в своей домашней папке, и перейдите в нее:

```
mkdir nextjs  
cd nextjs
```

и создайте свою первую структуру Next:

```
mkdir firstproject  
cd firstproject
```

Теперь используйте команду `npm` или `yarn`, чтобы инициализировать его как проект Node:

```
npm init -y  
#or  
yarn init -y
```

```
Euroset@Home MINGW64 ~/Branch/Next
$ mkdir firstproject
Euroset@Home MINGW64 ~/Branch/Next
$ cd firstproject/
Euroset@Home MINGW64 ~/Branch/Next/firstproject
$ yarn init -y
warning The yes flag has been set. This will automatically answer yes to all questions, which may have security implications.
success Saved package.json
Done in 0.11s.

Euroset@Home MINGW64 ~/Branch/Next/firstproject
$ |
```

Опция `-y` указывает npm (и yarn тоже) использовать настройки проекта по умолчанию, заполняя образец файла `package.json`.

#### package.json

```
{  
  "name": "firstproject",  
  "version": "1.0.0",  
  "main": "index.js",  
  "license": "MIT"  
}
```

Установим зависимости:

```
npm install next react react-dom  
#or  
yarn add next react react-dom
```

После этого в папке вашего проекта появится папка `node_modules` и файл `package-lock.json` или, если вы работали с yarn, как я, то `yarn.lock`. В файле `package.json` появятся установленные зависимости:

#### package.json

```
{  
  "name": "firstproject",  
  "version": "1.0.0",  
  "main": "index.js",  
  "license": "MIT",  
  "dependencies": {  
    "next": "^9.4.4",  
    "react": "^16.13.1",  
    "react-dom": "^16.13.1"  
  }  
}
```

Нам остается просто добавить в него скрипты, для управления приложением:

#### package.json

```
{  
  "name": "firstproject",  
  "version": "1.0.0",  
  "main": "index.js",  
  "license": "MIT",  
  "scripts": {  
    "dev": "next",  
    "build": "next build",  
    "start": "next start"  
  },  
  "dependencies": {  
    "next": "9.4.4",  
    "react": "16.13.1",  
    "react-dom": "16.13.1"  
  }  
}
```

Если вы хотите изменить порт запуска вашего приложения, то можно немного изменить скрипты:

Например так:

```
"dev": "next -p 3001"
```

Теперь создайте папку `pages` и добавьте файл `index.js`.

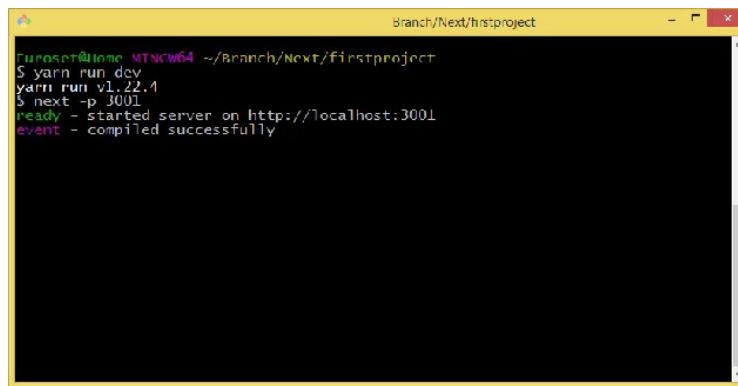
В этом файле давайте создадим наш первый компонент React.

Мы собираемся использовать его как экспорт по умолчанию:

```
const Index = () => (  
  <div>  
    <h1>Home page</h1>  
  </div>  
>);  
  
export default Index;
```

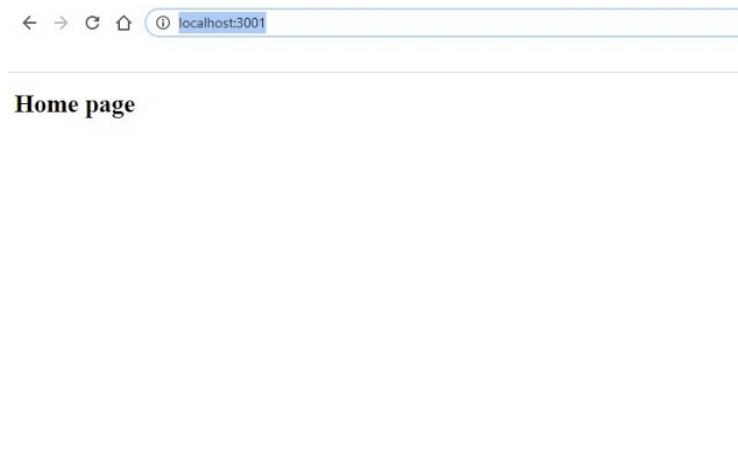
Сохраняем и запускаем наше приложение в консоли:

```
npm run dev  
#or  
yarn run dev
```



```
YARN v1.22.4  
$ yarn run dev  
yarn run v1.22.4  
$ next -p 3001  
ready - started server on http://localhost:3001  
event - compiled successfully
```

По адресу <http://localhost:3001/> мы увидим наше приложение:



## Просмотр источника, чтобы подтвердить работу SSR

Теперь давайте проверим, работает ли приложение, как мы ожидаем, и что оно будет рендерить на стороне сервера (**SSR - server side rendering**). Это приложение Next.js, поэтому оно должно отображаться на стороне сервера.

Это одно из главных преимуществ Next.js: если мы создаем сайт с помощью Next.js, страницы сайта отображаются на сервере, который доставляет HTML в браузер.

Это имеет 3 основных преимущества:

- Клиенту не нужно создавать экземпляр React для рендеринга, что делает сайт быстрее для ваших пользователей.
- Поисковые системы будут индексировать страницы без необходимости запуска клиентского JavaScript. Что-то, что Google начал делать, но открыто признал, что это более медленный процесс (и вы должны как можно больше помогать Google, если хотите получить хороший рейтинг).
- Вы можете использовать мета-теги в социальных сетях, полезные для добавления изображений для предварительного просмотра, настройки заголовка и описания для любой из ваших страниц, размещенных в Facebook, Twitter и т. д.

Давайте посмотрим на исходник приложения. Используя Chrome, вы можете щелкнуть правой кнопкой мыши в любом месте страницы и нажать `View page source`.



```
<!DOCTYPE html><html><head><style data-next-hide-for="true">body{display:none}</style><noscript data-next-hide-for="true"><style>body{display:block}</style></noscript><meta charset="utf-8"/><meta name="viewport" content="width=device-width"/><meta name="next-head-count" content="2"/><link rel="preload" href="/_next/static/development/pages/_app.js?ts=1592569265801" as="script"/><link rel="preload" href="/_next/static/development/pages/_index.js?ts=1592569265801" as="script"/><link rel="preload" href="/_next/static/runtime/webpack/_jstts=1592569265801" as="script"/><link rel="preload" href="/_next/static/runtime/main.js?ts=1592569265801" as="script"/><script id="__NEXT_CSS__DO_NOT_USE__"></script></head><body><div id="__next"><div><h1>Home page</h1></div><script src="/_next/static/runtime/react-refresh.js?ts=1592569265801"></script><script src="/_next/static/development/dll/01_f9de5cbc314a1e1f91a.js?ts=1592569265801"></script><script id="__NEXT_DATA__" type="application/json">{"props":{},"page":"/","query":{},"buildId": "development","nextExport": true,"autoExport": false}</script><script nomodule="" src="/_next/static/runtime/polyfills.js?ts=1592569265801"></script><script async="" data-next-page="/_app" src="/_next/static/development/pages/_app.js?ts=1592569265801"></script><script async="" data-next-page="/" src="/_next/static/development/pages/_index.js?ts=1592569265801"></script><script src="/_next/static/runtime/main.js?ts=1592569265801" async=""></script><script src="/_next/static/development/_buildManifest.js?ts=1592569265801" async=""></script><script src="/_next/static/development/_ssgManifest.js?ts=1592569265801" async=""></script></body></html>
```

Если вы просмотрите исходный код страницы, вы увидите фрагмент

```
<div>  
  <h1>Home page</h1>  
</div>;
```

в теле HTML вместе с набором файлов JavaScript - пакетами приложений.

Нам не нужно ничего настраивать, SSR (рендеринг на стороне сервера) уже работает для нас.

Приложение React будет запущено на клиенте, и оно будет активным взаимодействием, таким как нажатие на ссылку с использованием рендеринга на стороне клиента. Но перезагрузка страницы перезагрузит ее с сервера. И при использовании Next.js не должно быть никакой разницы в результатах внутри браузера - страница, отображаемая на сервере, должна выглядеть точно так же, как страница, отображаемая клиентом.

## Пакеты приложений

Когда мы просмотрели исходный код страницы, мы увидели несколько загружаемых файлов JavaScript:



The screenshot shows a browser's developer tools Network tab with several JavaScript files listed under the 'Development' section. The files include '\_index.js?ts=1592569265801', '\_app.js?ts=1592569265801', '\_static/runtime/main.js?ts=1592569265801', '\_static/runtime/\_index.js?ts=1592569265801', '\_static/runtime/\_app.js?ts=1592569265801', '\_static/runtime/\_polyfills.js?ts=1592569265801', and '\_static/development/\_buildManifest.js?ts=1592569265801'. These files are all loaded with the 'preload' attribute.

Давайте начнем с того, что поместим код в средство форматирования HTML [HTML formatter](#), чтобы лучше отформатировать его и лучше понять:

```
<!DOCTYPE html>
<html>
  <head>
    <style data-next-hide-fouc="true">
      body {
        display: none;
      }
    </style>
    <noscript data-next-hide-fouc="true">
      <style>
        body {
          display: block;
        }
      </style>
    </noscript>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <meta name="next-head-count" content="2" />
    <link rel="preload" href="/_next/static/development/pages/_app.js?ts=1592569265801" as="script" />
    <link rel="preload" href="/_next/static/development/pages/_index.js?ts=1592569265801" as="script" />
    <link rel="preload" href="/_next/static/runtime/webpack.js?ts=1592569265801" as="script" />
    <link rel="preload" href="/_next/static/runtime/main.js?ts=1592569265801" as="script" />
    <noscript id="__next_css__DO_NOT_USE__"></noscript>
  </head>
  <body>
    <div id="__next">
      <div><h1>Home page</h1></div>
    </div>
    <script src="/_next/static/runtime/react-refresh.js?ts=1592569265801"></script>
    <script src="/_next/static/development/dll/dll_f9de5cbc314ale4lf9le.js?ts=1592569265801"></script>
    <script id="__NEXT_DATA__" type="application/json">
      {"props": {"pageProps": {}}, "page": "/", "query": {}, "buildId": "development", "nextExport": true, "autoExport": true, "isFallback": false}
    </script>
    <script nomodule="" src="/_next/static/runtime/polyfills.js?ts=1592569265801"></script>
    <script async="" data-next-page="/_app" src="/_next/static/development/pages/_app.js?ts=1592569265801"></script>
    <script async="" data-next-page="/" src="/_next/static/development/pages/_index.js?ts=1592569265801"></script>
    <script src="/_next/static/runtime/webpack.js?ts=1592569265801" as="script" />
    <script src="/_next/static/runtime/main.js?ts=1592569265801" as="script" />
    <script src="/_next/static/development/_buildManifest.js?ts=1592569265801" as="script" />
    <script src="/_next/static/development/_ssgManifest.js?ts=1592569265801" as="script" />
  </body>
</html>
```

У нас есть 4 файла JavaScript, которые объявлены предварительно загруженными в head, используя rel = "preload" as = "script":

- /\_next/static/development/pages/\_index.js (96 LOC)
- /\_next/static/development/pages/\_app.js (5900 LOC)
- /\_next/static/runtime/webpack.js (939 LOC)
- /\_next/static/runtime/main.js (12k LOC)

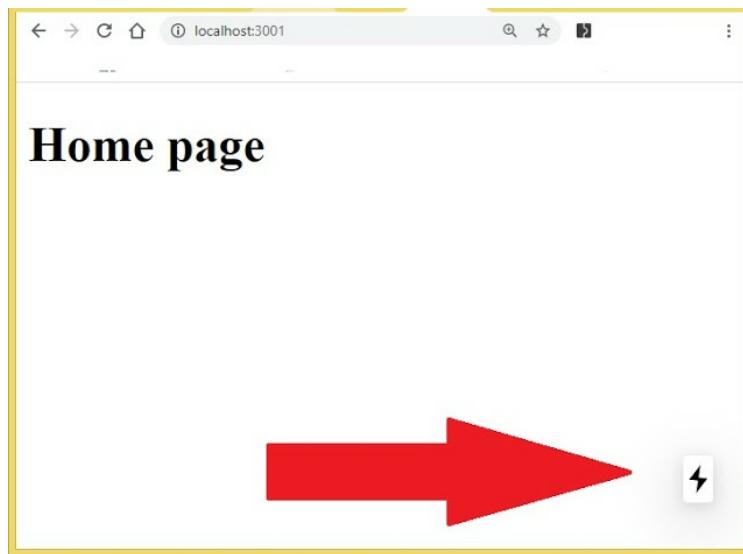
Это говорит браузеру начать загрузку этих файлов как можно скорее, прежде чем начнется обычный процесс рендеринга. Без них скрипты будут загружаться с дополнительной задержкой, что повышает производительность загрузки страницы. Затем эти 4 файла загружаются в конце тела вместе с /\_next/static/development/dll/dll\_01ec57fc9b90d43b98a8.js (31k LOC) и фрагментом JSON, который устанавливает некоторые значения по умолчанию для данных страницы:

```
<script id="__NEXT_DATA__" type="application/json">
{
  "dataManager": [],
  "props": {
    "pageProps": {}
  },
  "page": "/",
  "query": {},
  "buildId": "development",
  "nextExport": true,
  "autoExport": true
}</script>
```

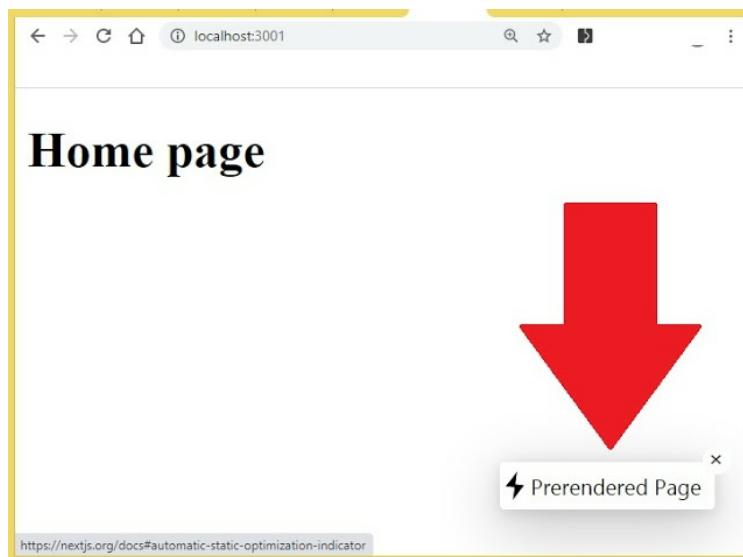
4 загруженных файла пакета уже реализуют одну функцию, называемую *разделением кода*. Файл index.js предоставляет код, необходимый для компонента index, который обслуживает / route, и если бы у нас было больше страниц, у нас было бы больше пакетов для каждой страницы, которые затем будут загружаться только при необходимости - для обеспечения большей производительности и уменьшения времени загрузки страницы.

## Что это за иконка справа внизу?

Вы видели эту маленькую иконку в правом нижнем углу страницы, которая выглядит как молния?



Если вы наведите указатель мыши, на нем будет написано «Prerendered Page»:



Этот значок, который, конечно, виден только в режиме разработки, говорит о том, что страница подходит для автоматической статической оптимизации, что в основном означает, что она не зависит от данных, которые необходимо получить во время вызова, и может быть предварительно обработана и построена как статический HTML-файл во время сборки (когда мы запускаем прм, запускаем сборку - `npm run build`).

Далее это можно определить по отсутствию метода `getInitialProps ()`, прикрепленного к компоненту страницы.

В этом случае наша страница может быть еще быстрее, потому что она будет статически обслуживаться как файл HTML, а не через сервер Node.js, который генерирует вывод HTML.

Другой полезный значок, который может появиться рядом с ним или вместо него на страницах без предварительной обработки, представляет собой небольшой анимированный треугольник:



Это индикатор компиляции, который появляется, когда вы сохраняете страницу, а Next.js компилирует приложение до того, как начнется горячая перезагрузка кода, чтобы автоматически перезагрузить код в приложении.

Это действительно хороший способ сразу определить, было ли приложение уже скомпилировано, и вы можете протестировать часть, над которой вы работаете.

## Установка React Developer Tools

Next.js основан на React, поэтому есть один очень полезный инструмент, который нам абсолютно необходимо установить (если вы этого еще не сделали), это **React Developer Tools**.

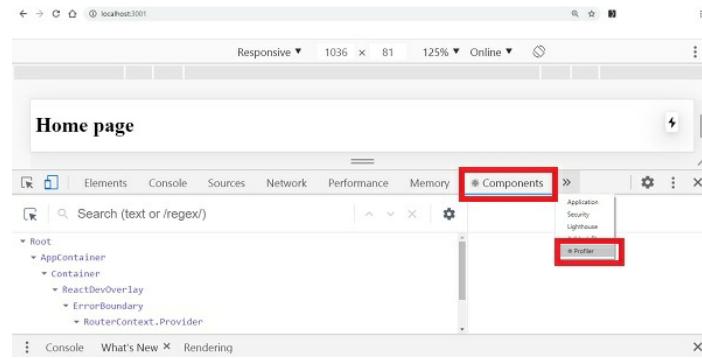
Доступные как для [Chrome](#), так и для [Firefox](#), инструменты разработчика React являются важным инструментом, который вы можете использовать для проверки приложения React.

Теперь инструменты разработчика React не являются специфическими для Next.js, но я хочу представить их, потому что вы, возможно, не на 100% знакомы со всеми

инструментами, которые предоставляет React. Лучше немного заняться инструментами отладки, чем предполагать, что вы их уже знаете.

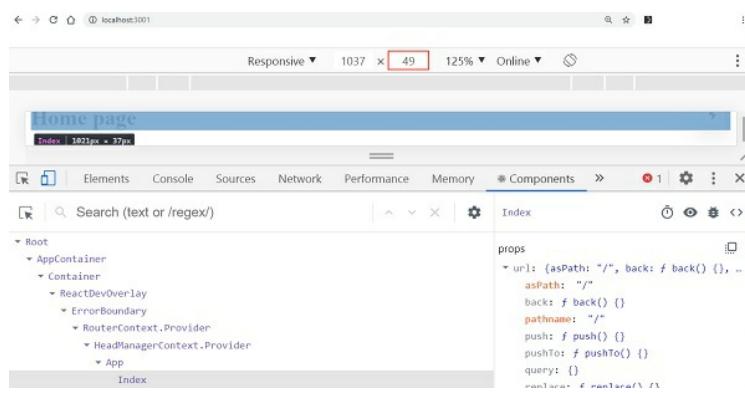
Они предоставляют инспектора, который показывает дерево компонентов React, которое создает вашу страницу, и для каждого компонента вы можете посмотреть и проверить реквизиты - `props`, состояние - `state`, хуки и многое другое.

После того, как вы установили React Developer Tools, вы можете открыть обычный браузер devtools (в Chrome щелкнуть правой кнопкой мыши на странице, затем нажать «Проверить», и вы увидите 2 новые панели: Компоненты - **Components** и Профилировщик - **Profiler**).



Если вы наведете указатель мыши на компоненты, вы увидите, что на странице браузер будет выбирать части, отображаемые этим компонентом.

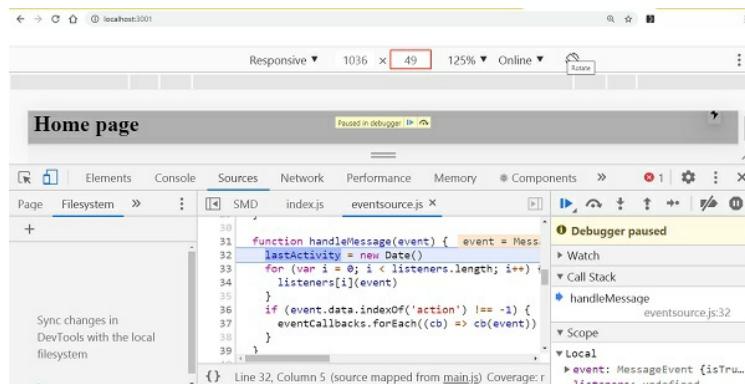
Если вы выберете какой-либо компонент в дереве, на правой панели отобразится ссылка на родительский компонент, а реквизиты - `props` будут переданы ему:



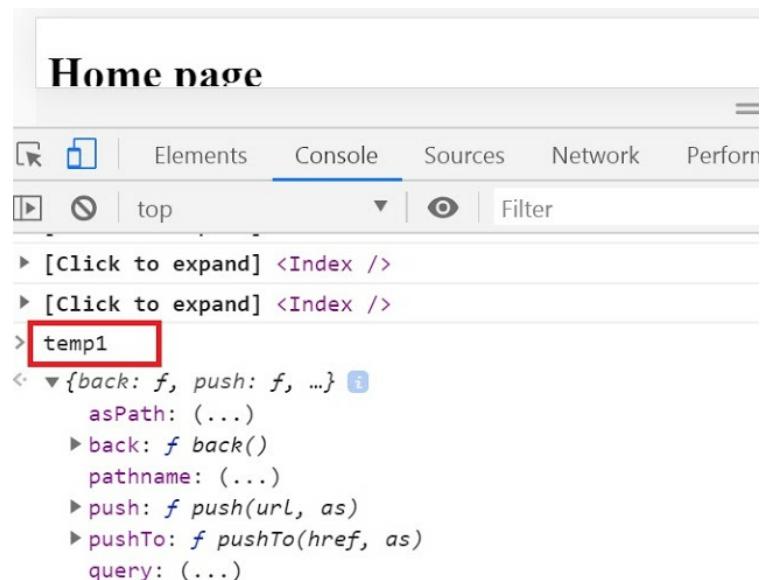
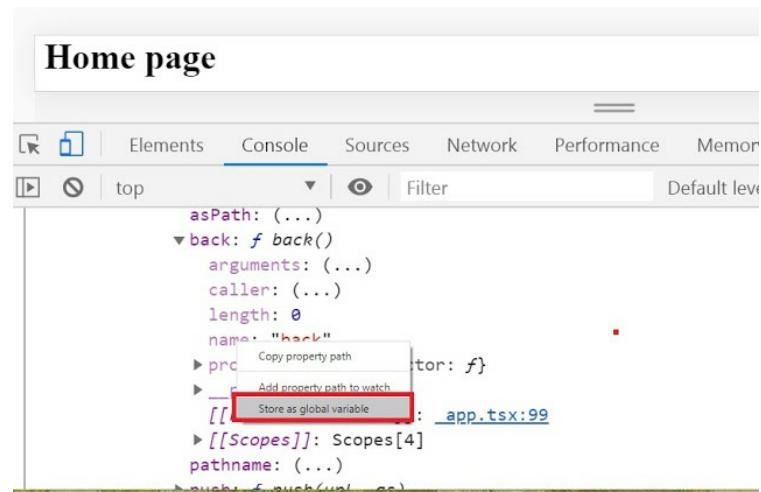
Вы можете легко перемещаться, нажимая на названия компонентов.

Вы можете щелкнуть значок глаза на панели инструментов «Инструменты разработчика», чтобы просмотреть элемент DOM, а также, если вы используете первый значок со значком мыши (который удобно расположен под аналогичным обычным значком DevTools), вы можете навести элемент на пользовательский интерфейс браузера для непосредственного выбора компонента React, который его отображает.

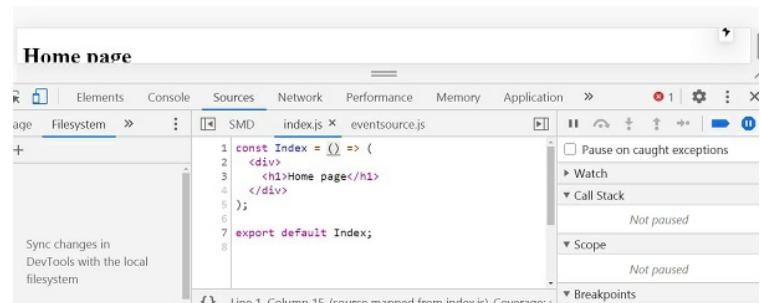
Вы можете использовать значок **bug**, чтобы записать данные компонента на консоль.



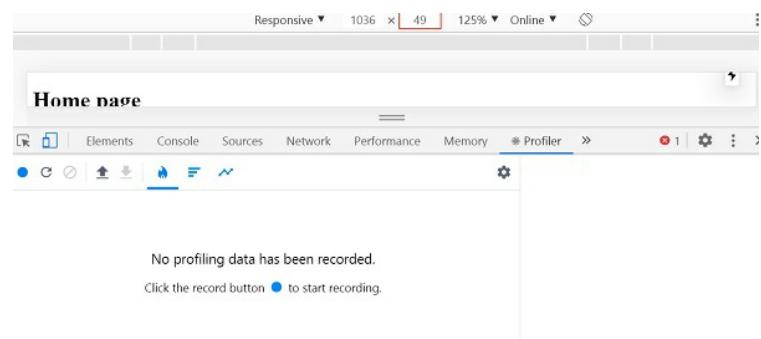
Это довольно круто, потому что, как только вы напечатаете там данные, вы можете щелкнуть правой кнопкой мыши по любому элементу и нажать «Сохранить как глобальную переменную - Store as global variable». Например, здесь я сделал это с помощью `url` проп, и я смог проверить его в консоли, используя временную переменную, назначенную ему, `temp1`:



Используя Карты исходного кода - [Source Maps](#), которые автоматически загружаются Next.js в режиме разработки, на панели «Компоненты» мы можем щелкнуть код <>, и DevTools переключится на панель «Источник», показывая нам исходный код компонента:



Вкладка «Профильтровщик», по возможности, еще более крутая. Это позволяет нам записывать взаимодействие в приложении и видеть, что происходит. Я пока не могу показать пример, потому что для создания взаимодействия нужны как минимум 2 компонента, а у нас есть только один. Я поговорю об этом позже.



Я показал все скриншоты, используя Chrome, но React Developer Tools работает точно так же в Firefox:

## Другие методы отладки, которые вы можете использовать

В дополнение к инструментам разработчика React, которые необходимы для создания приложения Next.js, я хочу подчеркнуть два способа отладки приложений Next.js:

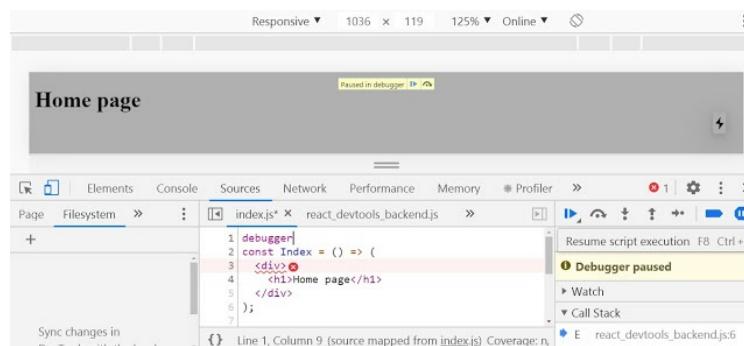
1. Первый, очевидно, `console.log()` и все остальные [инструменты API консоли](#). Работа приложений Next заставит оператор `log` работать в консоли браузера ИЛИ в терминале, с которого вы начали Next, используя `npm run dev`.

В частности, если страница загружается с сервера, когда вы указываете на нее URL-адрес или нажимаете кнопку обновления / cmd / ctrl + R, любое ведение журнала консоли происходит в терминале.

Последующие переходы страниц, которые происходят при щелчке мыши, приведут к тому, что вся запись в консоли будет происходить внутри браузера.

Просто помните, если вы удивлены отсутствием реакции.

2. Другим важным инструментом является оператор отладчика - `debugger`. Добавление этого оператора к компоненту приостановит отображение страницы браузером:

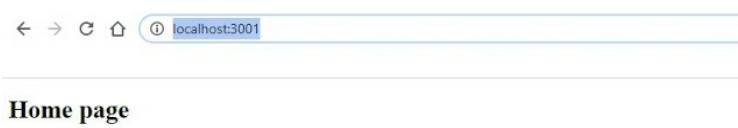


Действительно круто, потому что теперь вы можете использовать отладчик браузера для проверки значений и запуска своего приложения по одной строке за раз.

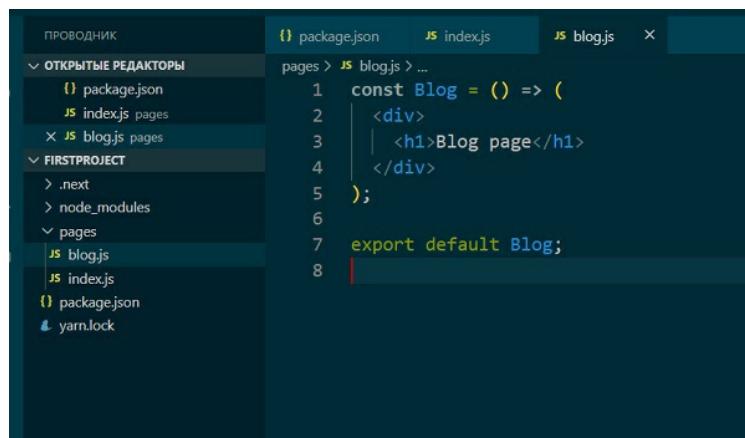
Вы также можете использовать отладчик VS Code для отладки кода на стороне сервера. Я упоминаю эту технику и [этот урок](#), чтобы настроить его.

## Добавление второй страницы на сайт

Теперь, когда мы хорошо разбираемся в инструментах, которые мы можем использовать для разработки приложений Next.js, давайте продолжим с того места, где мы оставили наше первое приложение:

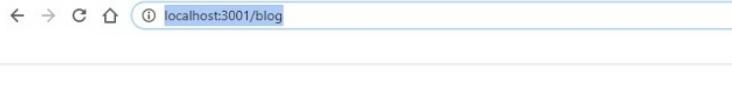


Я хочу добавить вторую страницу на этот сайт, блог. Она будет называться `blog`, и на данный момент она будет содержать простую статическую страницу, как наш первый компонент `index.js`:



После сохранения нового файла уже запущенный процесс `npm run dev` уже способен отображать страницу без необходимости ее перезапуска.

Когда мы нажимаем на URL <http://localhost:3001/blog>, у нас появляется новая страница:



## Blog page

и вот что сказал нам терминал:

A screenshot of a terminal window titled 'Branch/Next/firstproject'. It shows the output of a 'yarn run dev' command. The logs indicate the compilation of pages, starting with '/next/dist/pages/' and then moving through various stages of compilation for pages like '/index', '/blog', and '/error'. The terminal window has a red box highlighting the log output.

Теперь тот факт, что URL является `/blog`, зависит только от имени файла и его положения в папке страниц.

Вы можете создать страницу `/hey/ho`, и эта страница будет отображаться по URL-адресу <http://localhost:3001/hey/ho>.

Для URL не имеет значения имя компонента внутри файла.

Попробуйте просмотреть исходную страницу, когда она будет загружена с сервера, в качестве одного из загруженных пакетов будет указан `/next/static/development/pages/blog.js`, а не `/next/static/development/pages/index.js`, как на домашней странице. Это потому, что благодаря автоматическому разделению кода нам не нужен пакет, обслуживающий домашнюю страницу. Просто пакет, который обслуживает страницу блога.

A screenshot of browser developer tools showing the DOM structure of the 'Blog page'. The code is heavily minified but shows the structure of the page, including the header, body, and various script tags for Next.js runtime and polyfills.

Мы также можем просто экспортовать анонимную функцию из `blog.js`:

```
export default () => (
  <div>
    <h1>Blog</h1>
  </div>
)
```

или если вы предпочитаете синтаксис функции без стрелки:

```
export default function () {
  return (
    <div>
      <h1>Blog</h1>
    </div>
  );
}
```

## Связывание двух страниц

Теперь, когда у нас есть 2 страницы, определенные `index.js` и `blog.js`, мы можем ввести ссылки.

Обычные HTML-ссылки на страницах выполняются с помощью тега `a`:

```
<a href="/blog">Blog</a>;
```

Мы не можем сделать это в Next.js.

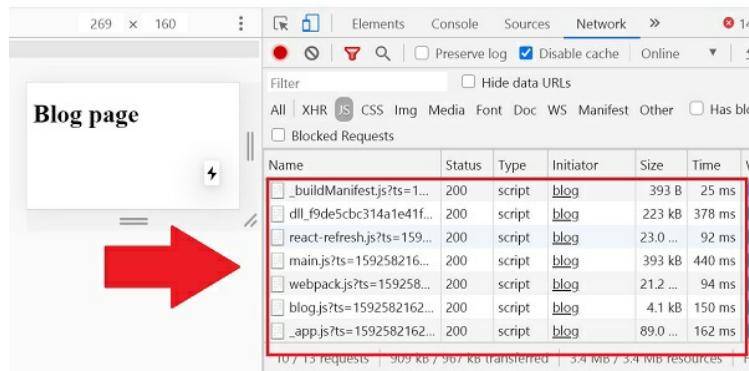
Почему? Мы технически можем, конечно, потому что это Интернет, и в Интернете вещи никогда не ломаются (хотя почему мы все еще можем использовать тег `<marquee>`). Но одно из основных преимуществ использования Next заключается в том, что после загрузки страницы происходит переход на другую страницу очень быстро благодаря рендерингу на стороне клиента.

Если вы используете простую ссылку `a` тега:

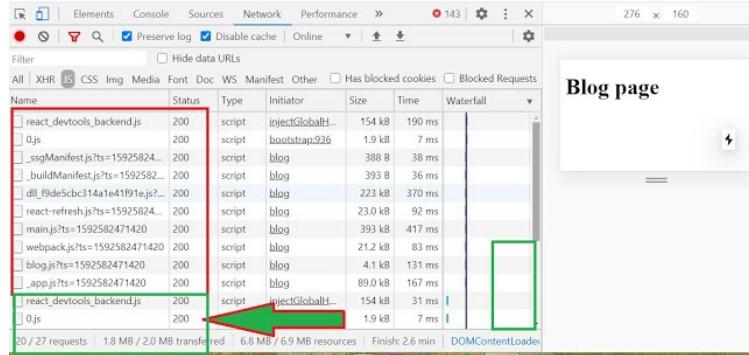
```
const Index = () => (
  <div>
    <h1>Home page</h1>
    <a href="/blog">Blog</a>
  </div>
);

export default Index;
```

Теперь откройте DevTools и, в частности, панель «Network». При первой загрузке `http://localhost:3001/` загружаются все пакеты страниц:



Теперь, если вы нажмете кнопку «Сохранить журнал» (чтобы избежать очистки панели «Сеть») и нажмете ссылку «Блог», количество загружаемых объектов удвоится и они будут повторять друг друга.



Мы снова получили весь этот JavaScript с сервера! Но .. нам не нужен весь этот JavaScript, если мы его уже получили. Нам просто нужен пакет страниц `blog.js`, единственный новый для этой страницы.

Чтобы решить эту проблему, мы используем компонент Next, который называется `Link`.

Мы импортируем это:

```
import Link from 'next/link'
```

и затем мы используем его, чтобы обернуть нашу ссылку, вот так:

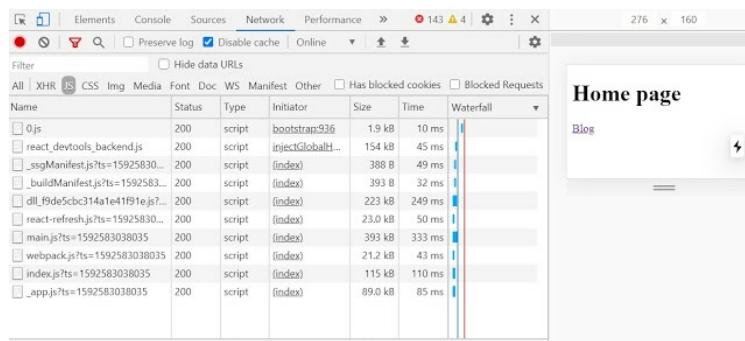
```
import Link from "next/link";

const Index = () => (
  <div>
    <h1>Home page</h1>
    <Link href="/blog">
      <a>Blog</a>
    </Link>
  </div>
);

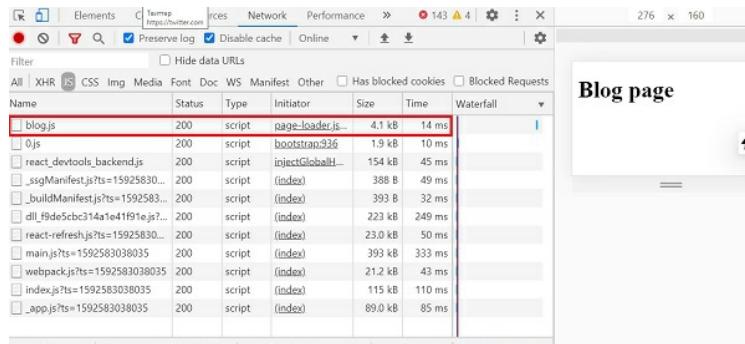
export default Index;
```

Теперь, если вы повторите попытку, которую мы делали ранее, вы увидите, что при переходе на страницу блога загружается только пакет blog.js:

Первая загрузка:



Переход на блок - добавился один файл - blog.js



и страница загружалась быстрее, чем раньше, обычная прокрутка браузера на вкладке даже не появлялась. Тем не менее, как вы видите, URL изменился. Это работает без проблем с API истории браузера.

Это рендеринг на стороне клиента в действии.

Что если вы сейчас нажмете кнопку «Назад»? Ничего не загружается, потому что в браузере все еще есть старый пакет index.js, готовый загрузить маршрут /index. Это все автоматически!

## Динамический контент с роутером

В предыдущей главе мы увидели, как связать страницу home со страницей blog.

Блог - отличный пример использования Next.js, который мы продолжим исследовать в этой главе, добавляя посты в блоге.

Сообщения блога имеют динамический URL. Например, пост под названием «Hello World» может иметь URL /blog/hello-world. Сообщение под названием «Мой второй пост» - (My second post) может иметь URL /blog/my-second-post.

Это содержимое является динамическим и может быть взято из базы данных, файлов разметки или другого.

Next.js может обслуживать динамический контент на основе динамического URL.

Мы создаем динамический URL, создавая динамическую страницу с синтаксисом [ ].

Как? Мы добавляем файл pages /blog/[id].js. Этот файл будет обрабатывать все динамические URL-адреса в /blog/route, например, те, которые мы упомянули выше: /blog/hello-world, /blog/my-second-post и другие.

В имени файла [id] в квадратных скобках означает, что все это является динамическим, будет помещено в параметр id свойства запроса маршрутизатора.

Маршрутизатор -роутер является библиотекой, предоставленной Next.js.

Импортируем его из next/router:

```
import { useRouter } from 'next/router'
```

и как только мы используем useRouter, мы создаем объект маршрутизатора, используя:

```
const router = useRouter()
```

Как только мы получим этот объект маршрутизатора, мы сможем извлечь из него информацию.

В частности, мы можем получить динамическую часть URL в файле [id].js, обратившись к router.query.id.

Динамическая часть также может быть просто частью URL, например post-[id].js.

Итак, давайте продолжим и применим все эти вещи на практике.

Создайте файл pages/blog/[id].js:

```
import { useRouter } from "next/router";

export default () => {
  const router = useRouter();

  return (
    <>
      <h1>Blog post</h1>
      <p>Post id: {router.query.id}</p>
    </>
  );
};
```

Теперь, если вы перейдете к маршрутизатору <http://localhost:3001/blog/test>, вы должны увидеть это:



## Blog post

Post id: test

Мы можем использовать этот параметр `id`, чтобы собрать сообщение из списка сообщений. Из базы данных, например. Для простоты мы добавим файл `posts.json` в корневую папку проекта:

```
posts.json
{
  "test": {
    "title": "test post",
    "content": "Hey some post content"
  },
  "second": {
    "title": "second post",
    "content": "Hey this is the second post content"
  }
}
```

Теперь мы можем импортировать его и найти сообщение из ключа `id`:

```
posts/[id].js
import { useRouter } from "next/router";
import posts from "../../posts.json";

export default () => {
  const router = useRouter();

  const post = posts[router.query.id];

  return (
    <>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </>
  );
};
```

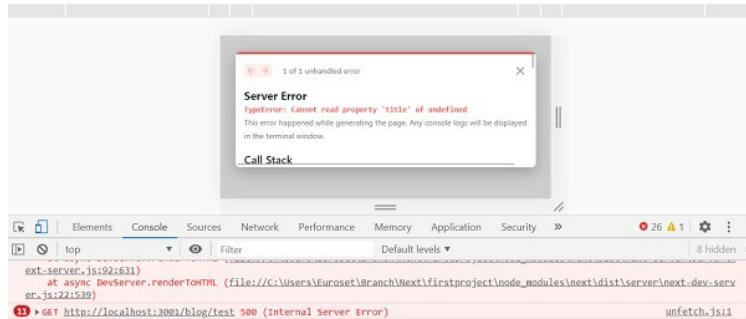
Перезагружаем страницу и ожидаем увидеть этот результат:

# Hello

## test post

Hey some post content

Но это не так! Вместо этого мы получаем ошибку в консоли, а также ошибку в браузере:



Почему? Потому что .. во время рендеринга, когда компонент инициализирован, данных еще нет. Мы увидим, как предоставить данные компоненту с помощью `getInitialProps` на следующем уроке.

А пока добавьте небольшую проверку `if (! Post) return <p> </p>` перед возвратом JSX:

```
pages/blog/[id].js

import { useRouter } from "next/router";
import posts from "../../posts.json";

export default () => {
  const router = useRouter();

  const post = posts[router.query.id];

  if (!post) return <p></p>
  return (
    <>
      <h2>Hello</h2>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </>
  );
}
```

Теперь все должно работать. Первоначально компонент отображается без динамической информации `router.query.id`. После рендеринга Next.js запускает обновление со значением запроса, и на странице отображается правильная информация.

И если вы просматриваете источник, в HTML есть пустой тег `<p>`:

```
<!DOCTYPE html><html><head><style data-next-hide-fouc="true">body{display:none}</style><noscript data-next-hide-fouc="true"><style>body{display:block}</style></noscript><meta charset="utf-8"/><meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no" /><link rel="preload" href="/_next/static/development/pages/_app.js?ts=1592651214126" as="script"/><link rel="preload" href="/_next/static/development/pages/blog/556d5D.js?ts=1592651214126" as="script"/><link rel="preload" href="/_next/static/runtime/webpack.js?ts=1592651214126" as="script"/><link rel="preload" href="/_next/static/runtime/main.js?ts=1592651214126" as="script"/><script id="__NEXT_CSS__" data-next-hide-fouc="true"></script><script src="/_next/static/development/d1/d1_f9de5cb314a1e4f91e.js?ts=1592651214126"></script><script id="__NEXT_DATA__" type="application/json" data-next-hide-fouc="true">{"props":{"pageProps":{}}, "page":"/blog/[id]", "query":{}, "buildId": "development", "nextExport": true, "autoExport": true, "isFallback": false}</script><script nomodule="" src="/_next/static/runtime/polyfills.js?ts=1592651214126"></script><script async="" data-next-page="/_app" src="/_next/static/development/pages/_app.js?ts=1592651214126"></script><script async="" data-next-page="/blog/[id]" src="/_next/static/development/pages/blog/556d5D.js?ts=1592651214126"></script><script src="/_next/static/runtime/webpack.js?ts=1592651214126" as="script"/><script src="/_next/static/runtime/main.js?ts=1592651214126" as="script"/><script src="/_next/static/development/_buildManifest.js?ts=1592651214126" as="script"/><script src="/_next/static/development/_ssgManifest.js?ts=1592651214126" as="script"/></body></html>
```

Вскоре мы исправим эту проблему, которая не позволяет реализовать SSR, и это вредит как времени загрузки для наших пользователей, так и SEO и социальному обмену, как мы уже обсуждали.

Мы можем завершить пример блога, перечислив эти посты в `pages/blog.js`:

```
pages/blog.js

import posts from "../../posts.json";

const Blog = () => (
  <div>
```

```

<h1>Blog</h1>
<ul>
  {Object.entries(posts).map((value, index) => {
    return <li key={index}>{value[1].title}</li>;
  })
</ul>
</div>
);
export default Blog;

```

И мы можем связать их с отдельными страницами сообщений, импортируя `Link` из `next/link` и используя ее в цикле сообщений:

### pages/blog.js

```

import Link from "next/link";
import posts from "../posts.json";

const Blog = () => (
  <div>
    <h1>Blog</h1>

    <ul>
      {Object.entries(posts).map((value, index) => {
        return (
          <li key={index}>
            <Link href={`/blog/[id]` as={`/blog/${value[0]}}>
              <a>{value[1].title}</a>
            </Link>
          </li>
        );
      })
    </ul>
  </div>
);

export default Blog;

```

## Предзагрузка

Ранее я упоминал, как компонент `Link` Next.js можно использовать для создания ссылок между двумя страницами, и когда вы его используете, Next.js прозрачно обрабатывает маршрутизацию веб-интерфейса для нас, поэтому, когда пользователь нажимает на ссылку, веб-интерфейс выполняет показ новой страницы без запуска нового запроса / ответа клиента / сервера, как это обычно происходит с веб-страницами.

Еще одна вещь, которую Next.js делает для вас, когда вы используете `Link`.

Как только элемент, заключенный в `<Link>`, появляется в окне просмотра (что означает, что он виден пользователю веб-сайта), Next.js предварительно выбирает URL-адрес, на который он указывает, при условии, что это локальная ссылка (на вашем веб-сайте), делая приложение супер быстрым для пользователя.

Такое поведение запускается только в производственном режиме - `production mode` (об этом мы поговорим позже), что означает, что вам нужно остановить приложение, если вы запускаете его с помощью `npm run dev`, скомпилировать свой рабочий пакет с помощью `npm run build` и запустить его командой - `npm run start`.

Используя инспектор сети - Network inspector в DevTools, вы заметите, что любые ссылки выше, при загрузке страницы, запускают предварительную выборку, как только событие загрузки - `load` запускается на вашей странице (срабатывает, когда страница полностью загружена, и происходит после события `DOMContentLoaded`).

Любой другой тег `Link`, отсутствующий в области просмотра, будет предварительно выбран при прокрутке пользователем.

Предварительная выборка выполняется автоматически на высокоскоростных соединениях (соединения Wi-Fi и 3G +, если браузер не отправляет HTTP-заголовок `Save-Data`).

Вы можете отказаться от предварительной выборки отдельных экземпляров `Link`, установив для параметра `prefetch` значение `false`:

```

<Link href="/a-link" prefetch={false}>
  <a>A link</a>
</Link>;

```

## Использование роутера для обнаружения активной ссылки

Одна очень важная функция при работе со ссылками - это определение текущего URL-адреса и, в частности, назначение класса активной ссылке, чтобы мы могли стилизовать его не так, как другие.

Это особенно полезно, например, в заголовке вашего сайта.

Компонент `Link` от Next.js по умолчанию, предлагаемый в `next/link`, не делает это автоматически для нас.

Мы можем создать компонент `Link` самостоятельно, и мы храним его в файле `Link.js` в папке `Components`, и импортируем его вместо `next/link` по умолчанию.

В этом компоненте мы сначала импортируем `React` from `'react'`, `Link` from `'next/link'` и `useRouter` из `next/router`.

Внутри компонента мы определяем, соответствует ли текущее имя пути `href` prop компонента, и если так, мы добавляем выбранный класс к дочерним элементам.

Наконец, мы возвращаем эти `children` с обновленным классом, используя `React.createElement ()`:

### components/Link.js

```

import React from "react";
import Link from "next/link";
import { useRouter } from "next/router";

export default ({ href, children }) => {
  const router = useRouter();

  let className = children.props.className || "";
  if (router.pathname === href) {

```

```
    className = `${className} selected`;
}

return <Link href={href}>{React.cloneElement(children, { className })}</Link>;
};
```

## Использование `next/router`

Мы уже видели, как использовать компонент `Link` для декларативной обработки маршрутизации в приложениях Next.js.

Управлять маршрутизацией в JSX очень удобно, но иногда вам нужно запускать изменение маршрутизации программно.

В этом случае вы можете получить прямой доступ к маршрутизатору (роутеру) Next.js, указанному в пакете `next/router`, и вызвать его метод `push ()`.

Вот пример доступа к маршрутизатору:

```
import { useRouter } from 'next/router'

export default () => {
  const router = useRouter()
  //...
}
```

Как только мы получим объект `router`, вызвав `useRouter ()`, мы сможем использовать его методы.

Это маршрутизатор на стороне клиента, поэтому методы должны использоваться только во внешнем коде. Самый простой способ убедиться в этом - заключить вызовы в ловушку `React useEffect ()` или внутри `componentDidMount ()` в компонентах с состоянием React.

Наиболее вероятно, что вы будете использовать чаще всего `push ()` и `prefetch ()`.

`push ()` позволяет нам программно инициировать изменение URL во внешнем интерфейсе:

```
router.push('/login')
```

`prefetch ()` позволяет нам программно выполнять предварительную выборку URL, что полезно, когда у нас нет тега `Link`, который автоматически обрабатывает предварительную выборку для нас:

```
router.prefetch('/login')
```

Полный пример:

```
import { useRouter } from 'next/router'

export default () => {
  const router = useRouter()

  useEffect(() => {
    router.prefetch('/login')
  })
}
```

Вы также можете использовать маршрутизатор для прослушивания событий изменения маршрута - [route change events](#).

## Подача данных в компоненты с помощью `getInitialProps`

В предыдущей главе у нас была проблема с динамическим генерируением страницы публикации, потому что компоненту требовалось некоторые данные заранее, когда мы пытались получить данные из файла JSON:

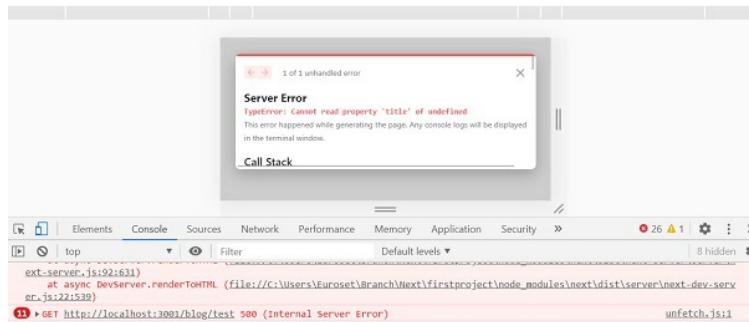
```
import { useRouter } from 'next/router'
import posts from '../../posts.json'

export default () => {
  const router = useRouter()

  const post = posts[router.query.id]

  return (
    <>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </>
  )
}
```

мы получили эту ошибку:



Как мы решаем это? И как мы заставляем SSR работать для динамических маршрутов?

Мы должны предоставить компоненту реквизиты-props, используя специальную функцию `getInitialProps()`, которая прикреплена к компоненту.

Для этого сначала назовем компонент:

```
const Post = () => {
  //...
}

export default Post
```

Затем мы добавляем функцию к нему:

```
const Post = () => {
  //...
}

Post.getInitialProps = () => {
  //...
}

export default Post
```

Эта функция получает объект в качестве аргумента, который содержит несколько свойств. В частности, сейчас нас интересует то, что мы получаем объект запроса (`query object`), который мы использовали ранее для получения идентификатора записи.

Таким образом, мы можем получить его, используя синтаксис деструктуризации объекта:

```
Post.getInitialProps = ({ query }) => {
  //...
}
```

Теперь мы можем вернуть `post` из этой функции:

```
Post.getInitialProps = ({ query }) => {
  return {
    post: posts[query.id]
  }
}
```

И мы также можем удалить импорт `useRouter`, и мы получаем сообщение из свойства `props`, переданного компоненту `Post`:

```
import posts from "../../posts.json";

const Post = (props) => {
  return (
    <div>
      <h1>{props.post.title}</h1>
      <p>{props.post.content}</p>
    </div>
  );
};

Post.getInitialProps = ({ query }) => {
  return {
    post: posts[query.id],
  };
};

export default Post;
```

Теперь ошибки не будет, и SSR будет работать как положено, как вы можете видеть, если посмотреть View Page Source в консоли:

```

<!DOCTYPE html><html><head><style data-next-hide=fouc="true">body{display:none}</style></head><body><script data-next-hide=fouc="true"><style>body{display:block}</style></script><meta charset="utf-8"/><meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=0" /><link rel="preload" href="/_next/static/_development/pages/_app.js?ts=1592745418820" as="script"/><link rel="preload" href="/_next/static/_development/pages/_blog/[id].js?ts=1592745418820" as="script"/><link rel="preload" href="/_next/static/runtime/webpack.js?ts=1592745418820" as="script"/><link rel="preload" href="/_next/static/runtime/main.js?ts=1592745418820" as="script"/><script id="__NEXT_DATA__" type="application/json">{"props":{"pageProps":{"post":{"id":"test","title":"test post","content":"Hey some post content"}}},"page":"/blog/[id]","query":{"id":"test"},"buildId":"development","isFallback":false,"gip":true}</script><script id="__NEXT_DATA__" type="application/json">{"props":{"pageProps":{"post":{"id":"test","title":"test post","content":"Hey some post content"}}},"page":"/blog/[id]","query":{"id":"test"},"buildId":"development","isFallback":false,"gip":true}</script><script nomodule="" src="/_next/static/runtime/polyfills.js?ts=1592745418820"></script><script async="" data-next-page="/_app" src="/_next/static/_development/pages/_app.js?ts=1592745418820"></script><script async="" data-next-page="/blog/[id]" src="/_next/static/_development/pages/_blog/[id].js?ts=1592745418820" as="script"/><script src="/_next/static/runtime/webpack.js?ts=1592745418820" as="script"/><script src="/_next/static/runtime/main.js?ts=1592745418820" as="script"/><script src="/_next/static/_development/_buildManifest.js?ts=1592745418820" as="script"/><script src="/_next/static/_development/_ssgManifest.js?ts=1592745418820" as="script"/></script></body></html>

```

Функция `getInitialProps` будет выполняться на стороне сервера, а также на стороне клиента, когда мы перейдем на новую страницу, используя компонент `Link`, как мы это делали. Важно отметить, что `getInitialProps` получает в объекте контекста, который он получает, в дополнение к объекту запроса - `query` эти другие свойства:

- `pathname: path` раздел пути URL
- `asPath` - Стока фактического пути (включая запрос) отображается в браузере

что в случае вызова <http://localhost:3001/blog/test> приведет соответственно к:

- `/blog/[id]`
- `/blog/test`

И в случае рендеринга на стороне сервера он также получит:

- `req` - HTTP request object (HTTP - объект запроса)
- `res` - HTTP response object (HTTP - объект ответа)
- `err` - error object (объект ошибки)

`req` и `res` будут вам знакомы, если вы выполняли какую-либо кодировку Node.js. Если забыли, то стоит посмотреть [мои статьи по работе с Node.js](#)

## CSS

Как нам оформить компоненты React в Next.js?

У нас для этого много свободы и вариантов, поэтому мы можем использовать любую библиотеку, какую пожелаем.

Но Next.js поставляется со встроенным `styled-jsx`, потому что это библиотека, созданная теми же людьми, которые работают над Next.js

И это довольно крутая библиотека, которая предоставляет нам ограниченный CSS, который отлично подходит для сопровождения, потому что CSS влияет только на компонент, к которому он применяется.

Я думаю, что это отличный подход при написании CSS, без необходимости применять дополнительные библиотеки или препроцессоры, которые увеличивают сложность.

Чтобы добавить CSS в компонент React в Next.js, мы вставляем его во фрагмент кода в JSX, который начинается с

```
<style jsx>`
```

и заканчивается

```
`</style>
```

Внутри этих странных блоков мы пишем простой CSS, как мы это делаем в файле `file.css`:

```
<style jsx>`  
h1 {  
  font-size: 3rem;  
}  
</style>;
```

Вы пишете это внутри JSX, вот так:

```
const Index = () => (  
  <div>  
    <h1>Home page</h1>  
  
    <style jsx>`  
      h1 {  
        font-size: 3rem;  
      }  
    `</style>  
  </div>  
)  
  
export default Index;
```

Внутри блока мы можем использовать интерполяцию для динамического изменения значений. Например, здесь мы предполагаем, что родительский компонент передает размер `-size` в `props`, и мы используем его в блоке `styled-jsx`:

```
const Index = (props) => (  
  <div>  
    <h1>Home page</h1>  
  
    <style jsx>`
```

```
h1 {
  font-size: ${props.size}rem;
}
`}</style>
</div>
);
```

Если вы хотите применить некоторые CSS глобально, не ограничивая область действия компонента, вы добавляете ключевое слово `global` к тегу `style`:

```
<style jsx global>`  
body {  
  margin: 0;  
`}</style>
```

Если вы хотите импортировать внешний CSS-файл в компонент Next.js, вы должны сначала установить `@zeit/next-css`:

```
npm install @zeit/next-css  
#or  
yarn add @zeit/next-css
```

а затем создайте файл конфигурации в корневом каталоге проекта с именем `next.config.js` с таким содержимым:

```
const withCSS = require('@zeit/next-css')
module.exports = withCSS()
```

После перезапуска приложения Next.js вы можете импортировать CSS, как это обычно делается с библиотеками или компонентами JavaScript:

```
import '../style.css'
```

Вы также можете импортировать файл SASS напрямую, используя вместо этого библиотеку `@zeit/next-sass`.

## Заполнение тега `head` пользовательскими тегами

С любого компонента страницы Next.js вы можете добавить информацию в заголовок страницы.

Это удобно, когда:

- Вы хотите настроить заголовок страницы
- Вы хотите изменить метатег

Как ты можешь это сделать?

Внутри каждого компонента вы можете импортировать компонент `Head` из `next/head` и включить его в вывод JSX вашего компонента:

```
import Head from "next/head";

const House = (props) => (
  <div>
    <Head>
      <title>The page title</title>
    </Head>
    {/* the rest of the JSX */}
  </div>
);

export default House;
```

Вы можете добавить любой HTML-тег, который хотите отображать в разделе `<head>` страницы.

При монтировании компонента Next.js будет следить за тем, чтобы теги внутри `Head` добавлялись в заголовок страницы. То же самое при размонтировании компонента, Next.js позаботится об удалении этих тегов.

## Добавление компонента-оболочки

Все страницы на вашем сайте выглядят более или менее одинаково. Там есть окно браузера, общий базовый слой, и вы просто хотите изменить то, что внутри.

Там есть навигационная панель, боковая панель, а затем фактический контент.

Как вы строите такую систему в Next.js?

Есть 2 способа. Один из них использует компонент высшего порядка (HOC - [Higher Order Component](#), создав компонент `component/Layout.js`:

`component/Layout.js`

```
export default (Page) => {
  return () => (
    <div>
      <nav>
        <ul>....</ul>
```

```
</hav>
<main>
  <Page />
</main>
</div>
);
};
```

Там мы можем импортировать отдельные компоненты для заголовка `heading` и / или боковой панели - `sidebar`, а также можем добавить весь необходимый нам CSS.

И вы используете его на каждой странице, как это:

```
import withLayout from "../components/Layout.js";
const Page = () => <p>Here's a page!</p>;
export default withLayout(Page);
```

Но я обнаружил, что это работает только для простых случаев, когда вам не нужно вызывать `getInitialProps ()` на странице.

Почему?

Потому что `getInitialProps ()` вызывается только на компоненте страницы. Но если мы экспортим Компонент Высшего Порядка с помощью `Layout ()` со страницы, `Page.getInitialProps ()` не вызывается, `withLayout.getInitialProps ()` будет.

Чтобы избежать ненужного усложнения нашей кодовой базы, альтернативный подход заключается в использовании `props`:

```
export default (props) => (
  <div>
    <nav>
      <ul>....</ul>
    </nav>
    <main>{props.content}</main>
  </div>
);
```

и теперь на наших страницах мы используем это так:

```
import Layout from "../components/Layout.js";
const Page = () => <Layout content={<p>Here's a page!</p>} />;
```

Этот подход позволяет нам использовать `getInitialProps ()` из нашего компонента страницы, с единственным недостатком - писать JSX компонента внутри `content props`:

```
import Layout from "../components/Layout.js";
const Page = () => <Layout content={>p>Here's a page!</p>} />;
Page.getInitialProps = ({ query }) => {
  //...
};
```

## API-маршруты

Помимо создания маршрутов страниц, что означает, что страницы передаются в браузер как веб-страницы, Next.js может создавать маршруты API.

Это очень интересная функция, поскольку она означает, что Next.js можно использовать для создания внешнего интерфейса для данных, которые хранятся и извлекаются самим Next.js, передавая JSON через запросы выборки.

Маршруты API (API routes) находятся в папке `/pages/api/` и сопоставляются с конечной точкой `/api`.

Эта функция очень полезна при создании приложений.

На этих маршрутах мы пишем код Node.js (а не код React). Это смена парадигмы, вы переходите с внешнего интерфейса на внутренний, но очень плавно.

Предположим, у вас есть файл `/pages/api/comments.js`, целью которого является возвращение комментариев к записи блога в формате JSON.

Допустим, у вас есть список комментариев, хранящихся в файле `comments.json`:

```
/pages/api/comments.js
```

```
[{
  "comment": "First"
},
{
  "comment": "Nice post"
}]
```

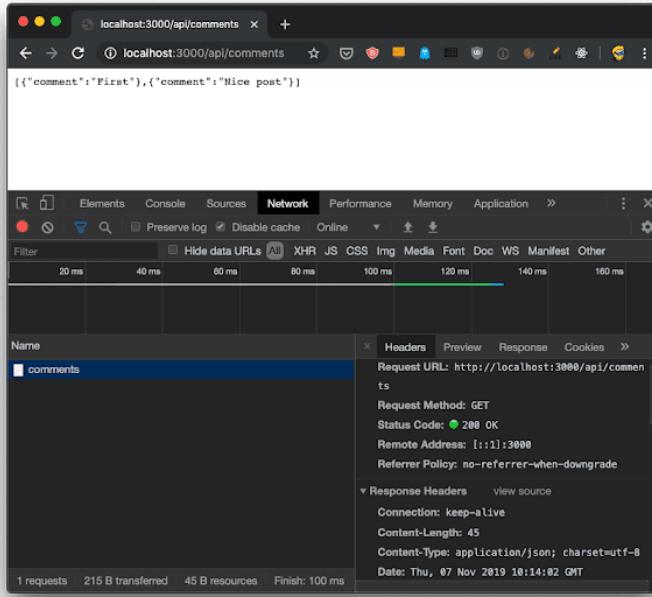
Вот пример кода, который возвращает клиенту список комментариев:

```
/pages/api/comments.js
```

```
import comments from './comments.json'
```

```
export default (req, res) => {
  res.status(200).json(comments)
}
```

Он будет прослушивать URL-адрес `/api/comments` для запросов GET, и вы можете попробовать вызвать его с помощью браузера:



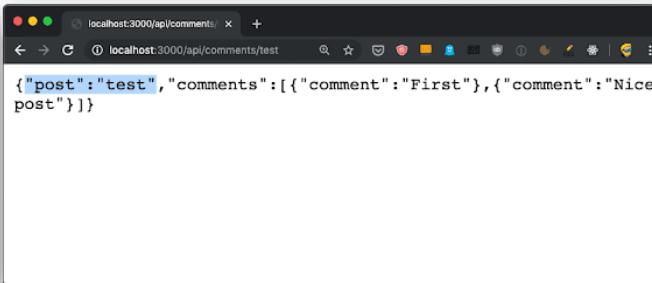
Маршруты API также могут использовать динамическую маршрутизацию, например страницы, использовать синтаксис `[]` для создания динамического маршрута API, например `/pages/api/comments/[id].js`, который будет извлекать комментарии, относящиеся к идентификатору записи.

Внутри `[id].js` вы можете получить значение `id`, посмотрев его внутри объекта `req.query`:

```
import comments from '../comments.json'

export default (req, res) => {
  res.status(200).json({ post: req.query.id, comments })
}
```

Вот вы можете увидеть приведенный выше код в действии:



На динамических страницах вам нужно будет импортировать `useRouter` из `next/router`, затем получить объект `router` с помощью `const router = useRouter()`, и тогда мы сможем получить значение `id` с помощью `router.query.id`.

На стороне сервера все проще, так как запрос привязан к объекту запроса.

Если вы делаете запрос POST, все работает одинаково - все проходит через экспорт по умолчанию.

Чтобы отделить POST от GET и других методов HTTP (PUT, DELETE), найдите значение `req.method`:

```
export default (req, res) => {
  switch (req.method) {
    case 'GET':
      //...
      break
    case 'POST':
      //...
      break
    default:
      res.status(405).end() //Method Not Allowed
      break
  }
}
```

В дополнение к `req.query` и `req.method`, которые мы уже видели, мы имеем доступ к cookie-файлам, ссылаясь на `req.cookies`, тело запроса в `req.body`.

Под капотом все это работает на Micro, библиотеке, которая поддерживает асинхронные HTTP-микросервисы, созданной той же командой, которая создала Next.js.

Вы можете использовать любое промежуточное ПО [Micro](#) в наших маршрутах API, чтобы добавить больше функциональности.

## Выполнять код только на стороне сервера или на стороне клиента

В ваших компонентах страницы вы можете выполнить код только на стороне сервера или на стороне клиента, проверив свойство окна.

Это свойство существует только внутри браузера, поэтому вы можете проверить

```
if (typeof window === 'undefined') {  
}
```

и добавьте серверный код в этот блок.

Точно так же вы можете выполнить код на стороне клиента, только проверив

```
if (typeof window !== 'undefined') {  
}
```

JS Совет: здесь мы используем оператор `typeof`, потому что мы не можем обнаружить значение, которое не определено другими способами. Мы не можем сделать `if (window === undefined)`, потому что мы получили бы ошибку "окно не определено" - ("window is not defined" runtime error) во время выполнения.

Next.js, как оптимизация во время сборки, также удаляет код, который использует эти проверки из пакетов. Пакет на стороне клиента не будет включать содержимое, заключенное в блок `if (typeof window === 'undefined') {}`.

## Развертывание-Deploying рабочей версии

Развертывание приложения всегда остается последним в руководствах.

Здесь я хочу представить его на ранней стадии, просто потому, что развернуть приложение Next.js настолько просто, что мы можем погрузиться в него сейчас, а затем перейти к другим более сложным темам.

Помните, в главе [«Как установить Next.js»](#) я говорил вам добавить эти 3 строки в раздел скрипта `package.json`:

```
"scripts": {  
  "dev": "next",  
  "build": "next build",  
  "start": "next start"  
}
```

Мы использовали `npm run dev` до сих пор, чтобы вызвать следующую локально установленную команду в `node_modules/next/dist/bin/next`. Это запустило сервер разработки, который предоставил нам исходные карты и горячую перезагрузку кода, две очень полезные функции при отладке.

Эту же команду можно вызвать для создания веб-сайта с флагом сборки, запустив команду `npm run build`. Затем эту же команду можно использовать для запуска производственного приложения, передающего флаг запуска, путем запуска `npm run start`.

Эти 2 команды - те, которые мы должны вызывать для успешного развертывания рабочей версии нашего сайта локально. Рабочая версия высоко оптимизирована и не содержит исходных карт и других вещей, таких как горячая перезагрузка кода, которая не будет полезна для наших конечных пользователей.

Итак, давайте создадим производственное развертывание нашего приложения. Постройте это, используя:

```
npm run build
```

```
Euroset@Home MINGW64 ~/Branch/Next/firstproject (master)  
$ yarn run build  
yarn run v1.22.4  
$ next build  
Creating an optimized production build  
Compiled successfully.  
Automatically optimizing pages  
Page          Size    First Load  
  /           1.76 kB   60  
  /404        3.25 kB   61  
  /blog       1.89 kB   60  
  /blog/[id]  420 B    59  
  /First Load JS shared by all  
  | static-pages.js          387 B  
  | chunks/main/c638277c3844a8d011a7f28db706714.36a881.js 387 kB  
  | chunks/framework.c6faae.js 40 kB  
  | runtime/main.99661a.js   6.28 kB  
  | runtime/webpack.c21266.js 746 B  
  (Server) server-side renders at runtime (uses getInitialProps or getServerSideProps)  
  (Static)  automatically rendered as static HTML (uses no initial props)  
  (SSG)     automatically generated as static HTML + JSON (uses getStaticProps)  
Done in 39.39s.  
Euroset@Home MINGW64 ~/Branch/Next/firstproject (master)  
$
```

Выходные данные команды говорят нам, что некоторые маршруты (/ и /blog теперь представлены в виде статического HTML, а /blog/[id] будет обслуживаться серверной частью Node.js).

Затем вы можете запустить `npm run start` для локального запуска рабочего сервера:

```
Done in 39.39s.  
Euroset@Home MINGW64 ~/Branch/Next/firstproject (master)  
$ yarn run start  
yarn run v1.22.4  
$ next start  
ready - started server on http://localhost:3000
```

Идем на <http://localhost:3000> (кстати, обратите внимание, что именно 3000, а не 3001) и посмотрим нам локальную версию приложения.

## Развертывание на Now

В предыдущей главе мы развернули приложение Next.js локально.

Как мы развернем его на реальном веб-сервере, чтобы другие люди могли получить к нему доступ?

Один из самых простых способов развертывания приложения Next - через платформу Now, созданную Zeit, той же компанией, которая создала проект Open Source Next.js. Вы можете использовать сейчас для развертывания приложений Node.js, статических веб-сайтов и многоного другого.

Теперь процесс развертывания и распространения приложения становится очень, очень простым и быстрым, и в дополнение к приложениям Node.js они также поддерживают развертывание Go, PHP, Python и других языков.

Вы можете думать об этом как о «облаке», поскольку вы на самом деле не знаете, где будет развернуто ваше приложение, но вы знаете, что у вас будет URL-адрес, по которому вы сможете добраться до него.

Теперь вы можете начать пользоваться бесплатным щедрым планом, который в настоящее время включает 100 ГБ хостинга, 1000 вызовов функций без сервера в день, 1000 сборок в месяц, 100 ГБ пропускной способности в месяц и одно местоположение CDN. Страница с ценами помогает получить представление [о расходах](#), если вам нужно больше.

Лучший способ начать использовать Now - использовать официальный CLI Now:

```
npm install -g now
```

Как только команда станет доступна, запустите

```
now login
```

и приложение спросит вас о вашей электронной почте.

Если вы еще не зарегистрировались, создайте учетную запись на <https://zeit.co/signup>, прежде чем продолжить, а затем добавьте свою электронную почту в клиент CLI.

После этого из корневой папки проекта Next.js запустите

```
now
```

и приложение будет немедленно развернуто в облаке Now, и вам будет предоставлен уникальный URL-адрес приложения:

```
firstproject — now /Users/flaviocopes/dev/nextjs/firstproject — node /usr/local/bin/now — 75x12  
→ firstproject now  
> Deploying ~/dev/nextjs/firstproject under flaviocopes  
> Using project firstproject  
> Synced 7 files [962ms]  
> NOTE: This is the first deployment in the firstproject project. It will be promoted to production.  
> NOTE: To deploy to production in the future, run `now --prod`.  
> https://firstproject-2pv7khwrr.now.sh [5s]  
> Ready! Deployment complete [32s]  
- https://firstproject-sepia-ten.now.sh  
- https://firstproject.flaviocopes.now.sh [in clipboard]
```

Почему так много?

Первый - это URL, идентифицирующий развертывание. Каждый раз, когда мы разворачиваем приложение, этот URL будет меняться.

Вы можете протестировать сразу, изменив что-то в коде проекта и снова запустив сейчас:

```

firstproject now
> Deploying ~/dev/nextjs/firstproject under flaviocopes
> Using project firstproject
> Synced 1 file [982ms]
> https://firstproject-rdcwfkwt6.now.sh [4s]
> Ready! Deployed to https://firstproject.flaviocopes.now.sh [in clipboard]
[38s]

```

Другие 2 URL не изменятся. Первый случайный, второй - имя вашего проекта (по умолчанию это папка текущего проекта, имя вашей учетной записи, а затем now.sh).

Если вы посетите URL, вы увидите приложение, развернутое в производство.

## second post

Hey this is the second post content

Вы можете настроить сейчас, чтобы обслуживать сайт в своем собственном домене или поддомене, но я не буду сейчас углубляться в это.

Субдомена `now.sh` достаточно для тестирования.

### Анализируем комплекти приложений

Next.js предоставляет нам способ проанализировать сгенерированные пакеты кода.

Откройте файл `package.json` приложения и в разделе скриптов добавьте эти 3 новые команды:

```
"analyze": "cross-env ANALYZE=true next build",
"analyze:server": "cross-env BUNDLE_ANALYZE=server next build",
"analyze:browser": "cross-env BUNDLE_ANALYZE=browser next build"
```

Вот так:

```
{
  "name": "firstproject",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "scripts": {
    "dev": "next -p 3001",
    "build": "next build",
    "start": "next start",
    "analyze": "cross-env ANALYZE=true next build",
    "analyze:server": "cross-env BUNDLE_ANALYZE=server next build",
    "analyze:browser": "cross-env BUNDLE_ANALYZE=browser next build"
  },
  "dependencies": {
    "next": "^9.4.4",
    "react": "^16.13.1",
    "react-dom": "^16.13.1"
  }
}
```

затем установите эти 2 пакета:

```
npm install --dev cross-env @next/bundle-analyzer
#or
yarn add cross-env @next/bundle-analyzer --dev
```

Создайте файл `next.config.js` в корне проекта со следующим содержимым:

```
next.config.js
const withBundleAnalyzer = require("@next/bundle-analyzer")({
  enabled: process.env.ANALYZE === "true",
});

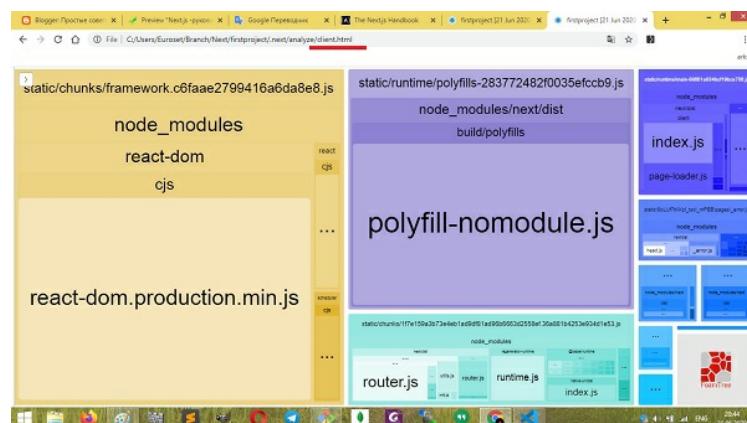
module.exports = withBundleAnalyzer({});
```

Теперь запустим:

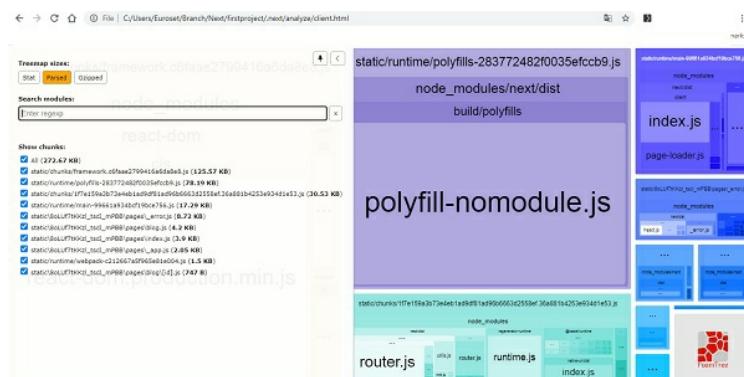
```
npm run analyze  
# or  
yarn run analyze
```

```
Euroset@Home MINGW64 ~/Branch/Next/Firstproject (master)  
$ yarn run analyze  
yarn analyze v1.0.0  
  cross-env ANALYZE=true next build  
webpack Bundle Analyzer saved report to C:\Users\Euroset\Branch\Next\firstproject\.next\analyze\server.html  
webpack Bundle Analyzer saved report to C:\Users\Euroset\Branch\Next\firstproject\.next\analyze\client.html  
Creating an optimized production build  
Compiled successfully.  
Automatically optimizing pages  
Page Size First Load JS  
[o] / 404 1.1 kB 61 kB  
[o] /blog 1.89 kB 60.6 kB  
[x] /blog/_document 42.8 kB 59.1 kB  
  ↳ 404 3.25 kB 61.9 kB  
  ↳ static/pages/_app.js 983 B 10.7 kB  
  ↳ chunks/1f7e159a3b3e4eb1ad9df81ad96b0663d2558ef.36a881.js 40 B 0.25 kB  
  ↳ runtime/webpack.c21266.js 746 B  
  ↳ runtime/_document.ejs 746 B  
  ↳ (Server) server-side renders at runtime (uses getInitialProps or getServerSideProps)  
  ↳ (Static) automatically rendered as static HTML (uses no initial props)  
  ↳ (SSG) automatically generated as static HTML + JSON (uses getStaticProps)  
Done in 16.42s.  
Euroset@Home MINGW64 ~/Branch/Next/Firstproject (master)  
$ |
```

Это должно открыть 2 страницы в браузере. Одну для клиентских пакетов и другую для серверных:



Это невероятно полезно. Вы можете проверить, что занимает больше всего места в пакетах, а также использовать боковую панель для исключения пакетов, чтобы упростить визуализацию меньших:



**Ленивая загрузка модулей.**

Возможность визуально анализировать пакет великолепна, потому что мы можем очень легко оптимизировать наше приложение.

Скажем, нам нужно загрузить библиотеку Moment в наши записи в блоге. Погнали:

```
npm install moment  
# or  
yarn add moment
```

включить его в проект.

Теперь давайте смоделируем тот факт, что он нам нужен на двух разных маршрутах: `/blog` и `/blog/[id]`.

Мы импортируем его в `pages/blog/[id].js`:

```
import moment from 'moment'  
  
...  
  
const Post = props => {  
  return (  
    <div>  
      <h1>{props.post.title}</h1>  
      <p>Published on {moment().format('ddd D MMMM YYYY')}</p>  
      <p>{props.post.content}</p>  
    </div>  
  )  
}
```

Я просто добавляю сегодняшнюю дату, как пример.

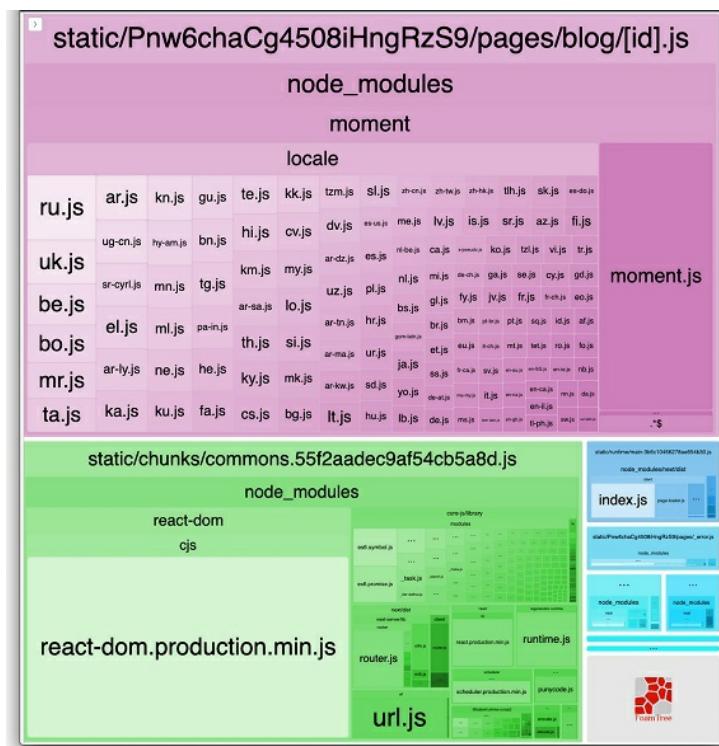
Это будет включать `Moment.js` в комплект страницы поста блога, как вы можете увидеть, запустив `npm run analyze`:

```
Euroset@Home MINGW64 ~/Branch/Next/firstproject (master)  
$ yarn run analyze  
yarn run v1.22.10  
$ cross-env ANALYZE=true next build  
webpack Bundle Analyzer saved report to C:\Users\Euroset\Branch\Next\firstproject\.next\analyze\server.html  
webpack Bundle Analyzer saved report to C:\Users\Euroset\Branch\Next\firstproject\.next\analyze\client.html  
Creating an optimized production build  
Compiled successfully.  
Automatically optimizing pages  
Page Size First Load JS  
└─ 404 1.76 kB 60.4 kB  
└─ /blog 3.25 kB 61.9 kB  
└─ /blog/[id] 1.89 kB 60.6 kB  
  + First Load JS shared by all 80.4 kB 139 kB  
    + static/pages/_app.js 18.7 kB  
    | chunks/_app/ab369637b061de8b198d08c9cbc.36a881.js 984 B  
    | chunks/_framework/c6fiae.js 10 kB  
    | runtime/main.72caef.js 40 kB  
    | runtime/webpack.c21266.js 6.28 kB  
    + runtime/webpack.c21266.js 746 B  
  ⚡ (Server) server-side renders at runtime (uses getInitialProps or getServerSideProps)  
  ⚡ (Static) automatically rendered as static HTML (uses no initial props)  
  • (SSG) automatically generated as static HTML + JSON (uses getStaticProps)  
Done in 31.80s.  
Euroset@Home MINGW64 ~/Branch/Next/firstproject (master)
```

Обратите внимание, что теперь у нас есть желтая запись в `/blog/[id]`, маршрут, к которому мы добавили `Moment.js`!

Он довольно сильно увеличился. И это потому, что сама библиотека `Moment.js` имеет размер 349 КБ.

Визуализация клиентских пакетов теперь показывает нам, что больший пакет - это страница, которой раньше было очень маленькой. И 99% его кода - это `Moment.js`.



Каждый раз, когда мы загружаем сообщение в блоге, мы собираемся передать весь этот код клиенту. Что не идеально.

Одним из исправлений было бы поискать библиотеку меньшего размера, поскольку **Moment.js** не известен своей легковесностью (особенно из коробки со всеми включенными локалями), но давайте предположим ради примера, что мы должны использовать именно её.

Вместо этого мы можем отдельить весь код **Moment.js** в отдельный пакет.

Как? Вместо того, чтобы импортировать Moment на уровне компонента, мы выполняем асинхронный импорт внутри `getInitialProps` и вычисляем значение для отправки компоненту.

Помните, что мы не можем вернуть сложные объекты внутри возвращенного объекта `getInitialProps ()`, поэтому мы вычисляем дату внутри него:

```
import posts from "../../posts.json";

const Post = (props) => {
  return (
    <div>
      <h1>{props.post.title}</h1>
      <p>Published on {props.date}</p>
      <p>{props.post.content}</p>
    </div>
  );
};

Post.getInitialProps = async ({ query }) => {
  const moment = (await import("moment")).default();
  return {
    date: moment.format("ddd D MMMM YYYY"),
    post: posts[query.id],
  };
};

export default Post;
```

Видите этот специальный вызов `.default ()` после импорта? Это необходимо для экспорт по умолчанию в динамическом импорте (см. [Https://v8.dev/features/dynamic-import](https://v8.dev/features/dynamic-import))

Теперь, если мы снова запустим `npm run analyze`, мы увидим это:

```
npm@6.14.0 /home/oleksik/next-project (master)
$ yarn run analyze
yarn run v1.22.4
$ cross-env ANALYZE=true next build
$ npx next-analyzer saved-report to C:\Users\Euroset\Branch\Next\firstproject\.next\analyze\server.html
$ webpack Bundle Analyzer saved-report to C:\Users\Euroset\Branch\Next\firstproject\.next\analyze\client.html
Creating an optimized production build
Compiled successfully.

Automatic optimization pages
Page          Size   First Load JS
- /           1.77 kB  61.5 kB
- 404         2.57 kB  62.3 kB
- /index       39.4 kB  61.6 kB
- /blog/[id]   734 kB  60.4 kB
First load JS shared by all
  static/pages/_app.js          985 B
  static/pages/_error.js        2.22 kB
  static/chunks/main.2f9113190da20dd0f78c283.3cf1d0.js  59.7 kB
  static/chunks/commons.a0e556.js  8.84 kB
  static/chunks/framework.e84fab.js  40 kB
  static/runtime/main.cfaa4c.js  6.28 kB
  static/runtime/webpack.0a3420.js  1.21 kB

  (Server) server-side renders at runtime (uses getInitialProps or getServerSideProps)
  (Static) automatically rendered as static HTML (uses no initial props)
  (SSG) automatically generated as static HTML + JSON (uses getStaticProps)

Done in 37.62s.
```

Наш пакет `/blog/[id]` снова очень маленький, так как Moment был перемещен в собственный файл пакета, загружаемый браузером отдельно.

## Куда идти дальше?

Есть еще много информации о Next.js. Я не говорил об управлении сессиями пользователей с помощью входа в систему, без сервера, управлении базами данных и так далее.

Цель этого Руководства не состоит в том, чтобы научить вас всему, но вместо этого оно призвано постепенно познакомить вас со всей мощью Next.js.

Следующий шаг, который я рекомендую, - внимательно прочитать официальную документацию Next.js, чтобы узнать больше обо всех функциях и возможностях, о которых я не говорил, и взглянуть на все дополнительные функции, представленные плагинами Next.js, некоторые из которых довольно удивительны.

Хотите освоить самые современные методы написания React приложений? Надоели простые проекты? Нужны курсы, книги, руководства, индивидуальные занятия по React и не только? Хотите стать разработчиком полного цикла, освоить стек MERN, или вы только начинаете свой путь в программировании, и не знаете с чего начать, то пишите через форму связи, подписывайтесь на мой канал в Телеге, вступайте в группу на Facebook.

Удачного кодирования!

---

Выражая особую благодарность [Flavio Copes](#), который вдохновил меня на этот труд.



Телеграм канал - Full Stack JavaScript Developer

Помочь проекту (любая валюта).

DONATE