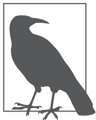
ГЛАВА 1

Что такое Node?

Node — это открытая кросс-платформенная среда выполнения, в которой разработчики могут создавать бэкенд-сервисы с помощью языка программирования JavaScript. Node построена на базе JavaScript-движка браузера Chrome, V8, и задействует множество встроенных модулей для асинхронности благодаря событийно-ориентированному подходу, известному также как неблокирующая модель. Разработчики Node могут использовать события и обработчики запросов, чтобы несколько операций выполнялись параллельно. Этот процесс эффективен и позволяет избегать сложностей, которые возникают при работе со множеством процессов и потоков.

Нам предстоит многое узнать. В главе 1 мы сперва познакомимся с Node, выделим принципы ее работы и поймем, почему она так популярна. Затем изучим основы интерфейса командной строки Node, работу с модулями и пакетами, а также синхронные и асинхронные операции. Далее мы обсудим основы событийно-ориентированной неблокирующей модели Node и узнаем, как использовать коллбэки (callbacks), промисы (promises) и события (events) для обработки результатов асинхронных операций.



Официальное название этой среды — Node.js, но для краткости я буду называть ее просто Node, как это делают другие специалисты.

Введение в Node

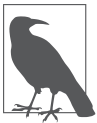
Работу над Node Райан Даль (Ryan Dahl) начал в 2009 году под впечатлением от производительности V8, JavaScript-движка Google Chrome. V8 основан на *событийно-ориентированной модели* и поэтому эффективно обрабатывает параллельные подключения и запросы. Райан захотел перенести эту событийно-ориентированную архитектуру на серверные приложения и добиться от них такой же производительности. Чтобы понимать, как работает Node (а заодно и V8), вам нужно запомнить, что оба они основаны на событийно-ориентированной модели. В этой главе я кратко введу вас в курс дела, а более подробно объясню все в главе 3.

Я начал работать с Node после того, как посмотрел презентацию Райана Даля. Мне кажется, вам тоже стоит начать изучение Node с нее. Найдите на YouTube ролик «Ryan Dahl introduction to Node» (https://oreil.ly/TACAt), но помните, что с тех пор Node сильно изменилась, так что советую обращать внимание не на примеры, а на идеи и объяснения.

По сути, Node позволяет использовать JavaScript на любой машине, не запуская браузер. Ее часто называют «JavaScript на бэкенде». До появления Node это было нетривиальной задачей, поэтому JavaScript в основном применяли во фронтенде.

Однако «JavaScript на бэкенде» — не совсем точное определение, ведь Node нужна не только для того, чтобы запускать JavaScript на сервере. На самом деле JavaScript выполняется на движке V8, а Node лишь предоставляет ему интерфейс для запуска кода.

V8 — это JavaScript-движок от Google с открытым исходным кодом, который компилирует и выполняет код, написанный на JavaScript. Он используется в Node, а также в Chrome и многих других браузерах. Кроме того, он применяется в Deno, новой среде выполнения для JavaScript, которую Райан Даль создал в 2018 году.



V8 — не единственный движок для JavaScript. Также существуют, например, SpiderMonkey, используемый в Firefox, и JavaScriptCore, который применяется в браузере Safari и в Bun, универсальной среде выполнения, менеджере пакетов и сборщике JavaScript.

Более точное определение Node звучит так: это серверная среда выполнения, построенная на базе V8 и предоставляющая модули, которые помогают разработчикам создавать и запускать эффективные приложения на JavaScript.

Ключевое слово в этом определении — *эффективные*. Node переняла и развила событийно-ориентированную модель, на которой работает V8. На этой модели построены большинство встроенных модулей Node. Они могут использоваться асинхронно, не блокируя основной поток, в котором выполняется код.

По сути, поток — это небольшой процесс, который происходит внутри более крупного. Каждый процесс может порождать несколько потоков. Каждый из этих потоков привязан к ядру процессора. Потоки могут одновременно использовать память и ресурсы в рамках общего процесса.

В многопоточном программировании тяжелые операции обычно запускаются в отдельных потоках. В Node же код выполняется в одном, главном, а тяжелые операции — асинхронно, вне этого потока.

Вам нужно прочитать содержимое внешнего файла? Это можно сделать асинхронно, не блокируя главный поток. Хотите запустить веб-сервер? Работать с сетевыми сокетами? Парсить, сжимать или шифровать данные? В Node у всех низкоуровневых операций асинхронный API, так что их можно совершать, не мешая остальным.

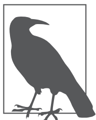
В Node для параллельного выполнения операций не нужно работать со множеством потоков вручную и тратить ресурсы на те, которые простаивают. Вы можете спокойно писать код в одном потоке, потом использовать асинхронные API, а Node возьмет на себя эффективное выполнение асинхронных операций вне главного потока.

Запуск любого кода после тяжелой операции можно организовать с помощью *событий* (event) и *обработчиков* (handler). Событие — это сигнал о том, что что-то произошло и нужно выполнить какое-то действие. Это действие задается функцией-обработчиком, связанной с этим событием. Каждый раз, когда поступает сигнал о событии, запускается нужный обработчик. Это и называется *событийно-ориентированным* подходом.

Подробнее мы разберем эти важные идеи, когда научимся запускать код Node и работать с ее модулями и пакетами.

Язык JavaScript

Изучив такие языки программирования, как Python, Lua и Haskell, Райан Даль остановился на JavaScript. Он решил, что этот язык программирования лучше всего подходит для Node, потому что он простой, гибкий и популярный. Но самое главное: в JavaScript можно обращаться с функциями высшего порядка так же, как с любыми другими объектами. Их можно сохранять в переменные, передавать в другие функции в виде аргументов и даже возвращать как результат, при этом сохраняя их состояние. Это свойство JavaScript активно используется в Node для обработки асинхронных операций.



Несмотря на ряд проблем, с которыми JavaScript сталкивался с самого начала, это вполне достойный язык программирования, и его можно сделать более эффективным с помощью TypeScript (это мы обсудим в главе 10).

JavaScript как браузерный язык программирования не только упрощает реализацию асинхронных операций, но и позволяет Node использовать один язык для всего стека. Также это дает массу других неочевидных преимуществ.

- Работа с одним языком позволяет реализовывать и держать в голове меньше синтаксиса, меньше API и меньше инструментов, с которыми нужно разбираться, а значит, это помогает делать меньше ошибок.
- Один язык обеспечивает более эффективную интеграцию между фронтендом и бэкендом и позволяет применять один код и там и там. Например, можно создать фронтенд-приложение с использованием React, а затем отрендерить те же компоненты на сервере с помощью Node и сгенерировать HTML-страницу для фронтенд-приложения. Это называется серверным рендерингом страницы (SSR). Многие фронтенд-фреймворки для Node поддерживают его по умолчанию.

- Единый язык программирования дает командам возможность распределять проект между разработчиками, не создавая отдельные подразделения для фронт- и бэкенда. Над всем стеком: над API, веб- и сетевыми серверами, интерактивными сайтами, мобильными и даже десктопными приложениями — может работать одна команда. Работодателю выгодно нанимать JavaScript-разработчиков, способных работать как с фронтом, так и с бэком.

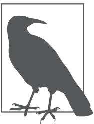
Запуск кода на Node

Если вы уже установили Node, команды `node` и `npm` должны быть доступны в терминале. Убедитесь, что у вас версия Node 20.x или выше. Чтобы это проверить, откройте терминал и выполните команду `node -v`.

Если команда `node` не распознается, вам нужно скачать и установить Node с официального сайта https://nodejs.org. Это просто и занимает всего несколько минут.

Пользователи macOS также могут установить Node с помощью системы управления пакетами Homebrew, выполнив команду:

```
$ brew install node
```



В этой книге символ `$` обозначает команды, которые вводятся в терминале. `$` не является частью команды. Как правило, он используется как символ приглашения для ввода команды в терминале.

Также установить Node можно с помощью Node Version Manager (NVM). Он позволяет запускать несколько версий Node и легко переключаться между ними. Это удобно, если для одного проекта требуется старая версия Node, а для другого — самая новая. NVM работает на Mac и Linux, а для Windows есть аналог — *nvm-windows*.

Node для Windows

Все примеры, которые я привожу в этой книге, написаны для macOS, но должны работать и на системах на базе Linux. Для Windows необходимы их эквиваленты.

Я не рекомендую использовать Node на Windows в «родной» среде при наличии альтернатив. Если у вас современный компьютер с Windows, советую установить подсистему Windows для Linux (WSL). Это отличный компромисс: вы сможете запускать Linux, не перезагружая компьютер, и даже редактировать код в Windows и выполнять его в Linux!

Если вы используете NVM, установите последнюю версию Node. Для этого введите команду:

```
$ nvm install node
```

У Node довольно часто выходят мажорные обновления. Новая версия на полгода получает статус *Current*, чтобы разработчики библиотек успели адаптировать свои проекты. Через полгода разработчики перестают поддерживать нечетные версии (19, 21 и т. д.), а четные (18, 20 и т. д.) переходят в статус *Active LTS*» (долгосрочная поддержка), обычно на 30 месяцев. В продуктовом окружении стоит использовать только версии с LTS-статусом.

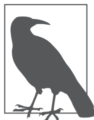
После того как команда `node` заработала, откройте терминал и введите `node` без аргументов. Запустится интерактивная сессия Node *REPL* (Read-Eval-Print-Loop[1]). Это удобный способ быстро протестировать простой JavaScript- или Node-код. Вы можете ввести любую строку на JavaScript. Например, попробуйте `Math.random()`, как показано на рис. 1.1.

```
TERMINAL                                   node + ⌄ ◫ 🗑 ···

○ $ node
  >
  > Math.random()
  0.08949350637305153
  >
  > |
```

Рис. 1.1. Режим Node REPL

Node прочитает строку, проведет вычисления, выведет результат и будет повторять этот цикл со всем кодом, пока вы не завершите сессию сочетанием клавиш Ctrl+D. Обратите внимание, что *вывод* результата происходит автоматически, без ввода дополнительных команд. Node сам покажет результат каждой строки. А вот при работе со скриптами Node вводить дополнительные команды все же придется. Давайте рассмотрим и этот вариант.



Подробнее особенности работы режима REPL в Node мы разберем в главе 2.

Создайте новую папку для отработки упражнений из этой книги и перейдите в нее с помощью команд:

---

1 Цикл «чтение — вычисление — вывод». — *Примеч. пер.*

```
$ mkdir efficient-node
$ cd efficient-node
```

Откройте редактор кода и создайте новый файл. Назовите его `test.js` и введите строку `Math.random()`:

```
Math.random();
```

Чтобы выполнить этот скрипт, введите следующую команду:

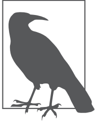
```
$ node test.js
```

Обратите внимание: после ввода команды ничего не произошло. Так получилось потому, что мы не вывели результат скрипта. Чтобы это сделать, нужно использовать глобальный объект `console`, аналогичный тому, что доступен в веб-браузерах. Например:

```
console.log(
  Math.random()
);
```

Теперь при запуске `test.js` в терминале выведется случайное число (как показано на рис. 1.2). В этом простом примере используется как стандартный объект JavaScript, `Math`, так и объект `console`, который есть в Node API. Метод `console.log` записывает значения своих аргументов в стандартный поток вывода текущего процесса (*stdout*).

Рис. 1.2. Выполнение скрипта Node



`console` — один из множества *глобальных объектов* высшего уровня, доступных в Node без необходимости подключать зависимости. По аналогии с тем, как в браузерах доступ к глобальному объекту `window` можно получить через свойство `globalThis`, в Node `globalThis` является глобальным объектом, а `console` — его частью. Node дает прямой доступ ко всем свойствам `globalThis`: например, `console.log` вместо `globalThis.console.log` (хотя второй вариант тоже работает).

Использование встроенных модулей

Чтобы настроить простой веб-сервер в Node, можно использовать встроенный модуль `node:http`. Создайте файл `server.js` и вставьте в него следующий код:

```
// Пример простейшего веб-сервера

const { createServer } = require('node:http');

const server = createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World');
});

server.listen(3000, '127.0.0.1', () => {
  console.log('Server is running...');
});
```

Так в Node выглядит хрестоматийное «Hello World». И этот скрипт запустится на обычной Node, без установки чего-либо еще! Отличный пример того, насколько Node функциональна сама по себе.

При запуске этого скрипта Node создаст веб-сервер и запустит его по адресу http://127.0.0.1:3000 (рис. 1.3).

Учитывайте, что у Node есть одна особенность: процесс не завершится, пока не произойдет непредвиденная ошибка. Это связано с тем, что Node работает в фоновом режиме в ожидании запросов пользователей и выполняет их по мере поступления.

Несмотря на то что это простейший веб-сервер, в нем есть несколько важных концептов. Давайте разберем их подробнее.

Функция `require` в Node — часть оригинальной системы управления зависимостями. Она позволяет одному модулю (например, `server.js`) использовать функциональность другого (например, `node:http`). Когда мы подключаем модуль `node:http` с помощью функции `require`, модуль `server.js` становится *зависимым* от модуля `node:http` и отдельно работать не может.

Также функциональность другого модуля позволяет задействовать инструкция `import`, которая относится к системе ES-модулей. Сейчас при работе на Node лучше

применять именно эту систему, так как она является частью современного стандарта ECMAScript для модулей на JavaScript. Большинство примеров, которые мы будем рассматривать в этой книге, основаны именно на ней. Однако стоит изучить и оригинальную систему модулей, CommonJS, поскольку множество проектов и библиотек построено на ней и, скорее всего, рано или поздно вам придется с ней столкнуться, даже если вы будете создавать проект с нуля.

server.js

```
3   const { createServer } = require('node:http');
4
5   const server = createServer((req, res) => {
6     res.writeHead(200, { 'Content-Type': 'text/plain' });
7     res.end('Hello World');
8   });
9
10  server.listen(3000, '127.0.0.1', () => {
11    console.log('Server is running...');
12  });
13
```

TERMINAL

```
$ node server.js
Server is running...
```

127.0.0.1:3000

Hello World

Рис. 1.3. Простейший веб-сервер

Существует множество библиотек для создания веб-сервера, но `node:http` встроена в Node по умолчанию, и чтобы ее использовать, не нужно устанавливать дополнительные компоненты. Однако ее нужно явно импортировать в проект с помощью функции `require` или инструкции `import`.

В Node REPL встроенные модули (например, `node:http`) доступны глобально — их не нужно подключать отдельно. Но чтобы использовать в исполняемых скриптах даже встроенные модули, следует сначала объявить зависимость от них.

При подключении модуля не обязательно загружать все его содержимое. Можно выбрать только те компоненты, что необходимы. В этом примере подключается только функция `createServer`, хотя в `node:http` доступно множество других функций и объектов.

Чтобы создать объект сервера, мы вызываем `createServer`. В качестве аргумента указываем функцию `RequestListener`. В синтаксис пока можете не вникать: сейчас важно понять суть.

В Node функции-слушатели связаны с конкретными событиями и запускаются при их срабатывании. В нашем примере Node будет выполнять функцию `RequestListener` при каждом входящем подключении к веб-серверу — это и есть событие этой функции.

Функция-слушатель принимает два аргумента.

Объект запроса (в примере назван `req`*)*

Предоставляет информацию о входящих запросах, например URL или IP-адрес клиента.

Объект ответа (в примере назван `res`*)*

Отправляет ответ клиенту. Именно это делает наш простой веб-сервер. Он устанавливает успешный код ответа на 200, а заголовок `Content-Type` меняет на `text/plain`. Затем он отправляет текст `Hello World` методом `end` объекта `res`.

Функция `createServer` лишь создает объект сервера, но не запускает его. Чтобы запустить веб-сервер, нужно вызвать на созданном сервере метод `listen`.

Метод `listen` принимает множество аргументов — например, хост и порт, на которых будет работать сервер. Последним аргументом передается функция, которая вызовется один раз при успешном запуске сервера. В нашем примере эта функция выводит в консоль сообщение о том, что сервер работает.

Обе функции, которые используются в методах `createServer` и `listen`, это примеры функций-обработчиков: они привязаны к событиям и связаны с асинхронными операциями. Как управляются такие события и функции-обработчики, мы подробнее изучим в главе 3.

Обратите внимание: при работе со встроенными модулями в Node я пишу префикс `node:`. Он помогает сразу отличить встроенный модуль от внешнего. Для некоторых модулей (например, `node:test`) префикс обязателен, так что лучше использовать его везде — для единообразия.

Сочетание клавиш Ctrl+C в терминале останавливает работу веб-сервера.

Использование пакетов

Менеджер пакетов для Node npm — простой инструмент командной строки, который позволяет устанавливать в проект внешние *пакеты* и управлять ими. Пакет может включать как один, так и несколько модулей, которые сгруппированы

вместе и предоставляют единый API. Подробнее об npm, его командах и работе с пакетами мы поговорим в главе 5. Сейчас же давайте рассмотрим простой пример установки и использования npm-пакета.

В качестве примера возьмем популярный пакет `lodash` — это вспомогательная библиотека JavaScript, содержащая множество полезных методов, которые можно применять к числам, строкам, массивам, объектам и другим элементам.

Сначала нужно скачать пакет. Для этого используется команда `npm install`:

```
$ npm install lodash
```

Она скачает пакет `lodash` из репозитория[1] npm и поместит его в папку `node_modules` (а также создаст эту папку, если ее еще нет). Проверить это можно, запустив команду `ls`:

```
$ ls node_modules
```

После выполнения команды в папке `node_modules` должна появиться папка `lodash`.

Теперь вы можете подключить и использовать в своем коде модуль `lodash` с помощью `require`. Например, у `lodash` есть метод `random`, который генерирует случайное число в заданном диапазоне значений. Вот пример его использования:

```
const _ = require('lodash');

console.log(
  _.random(1, 99)
);
```

После выполнения этого скрипта на экране появится случайное число от 1 до 99 (рис. 1.4).

Подчеркивание (_) — распространенное имя переменной для `lodash`, но вы можете использовать любое другое.

Так как мы импортировали внешний (не встроенный) модуль `lodash` с использованием метода `require`, Node будет искать его в папке `node_modules` — и найдет благодаря npm.

---

[1] В оригинале используется словосочетание npm registry, которое в современном техническом языке полностью калькируется с английского. Под русским словом «реестр» чаще понимается windows registry — централизованное хранилище параметров операционной системы. При работе с пакетами, как правило, употребляется слово «репозиторий». — *Примеч. науч. ред.*

Рис. 1.4. Использование npm-пакета

При работе в команде важно уведомить других разработчиков о добавлении внешней зависимости. В Node это делается через файл `package.json`, который располагается в корне проекта.

При установке модуля с помощью `npm install` команда `npm` автоматически добавляет его и его текущую версию в файл `package.json` — в раздел `dependencies`[1]. Посмотрите, как это выглядит в файле `package.json`, который был автоматически создан после установки пакета `lodash` (рис. 1.5).

Файл `package.json` может содержать такую информацию о проекте, как имя, версия, описание и так далее. Также в него можно включить сведения о сценариях командной строки, которые должны быть запущены для выполнения разнообразных задач, например сборки и тестирования проекта.

Вот так обычно выглядит файл `package.json`:

```
{
  "name": "efficient-node",
  "version": "1.0.0",
  "description": "A guide to learning Node.js",
  "license": "MIT",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "lodash": "^4.17.21"
  }
}
```

[1] Зависимости. — *Примеч. пер.*

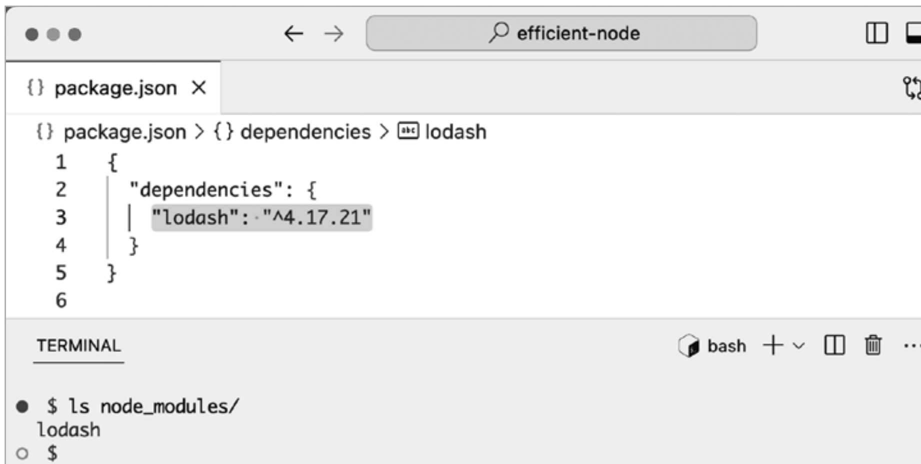


Рис. 1.5. Файл `package.json`

Файл `package.json` для нового проекта также можно создать в интерактивном режиме. Для этого следует ввести команду:

```
$ npm init
```

После этого нужно будет ответить на несколько вопросов (или каждый раз нажимать Enter, чтобы оставить значения по умолчанию, — чаще всего они подойдут, так как команда старается автоматически определить информацию о проекте). Затем введите команду `npm install` и установите новый пакет (например, `chalk`). Он появится в файле `package.json` в разделе `dependencies`. Удалите пакет с помощью `npm uninstall` и убедитесь, что он исчез из `package.json`.

Со временем в файле `package.json` появится множество зависимостей. Другим разработчикам, если им придется работать с вашим кодом, будет достаточно ввести команду `npm install`, и она установит все зависимости, перечисленные в `package.json`, в папку `node_modules`.

Некоторые пакеты, например ESLint, нужны только в разработке, а в продуктовом окружении в них необходимости нет. При запуске команды `npm install` можно указать, что эта зависимость используется только для разработки. Это можно сделать с помощью аргумента `--save-dev` (или просто `-D`):

```
$ npm install -D eslint
```

Эта команда установит пакет `eslint` в папку `node_modules` и добавит его в раздел `devDependencies` файла `package.json`. В этот раздел помещаются тестовые фреймворки, средства форматирования кода и другие утилиты, которые используются только при разработке.

Помимо `dependencies` и `devDependencies` в `package.json` можно указывать `optionalDependencies` — необязательные зависимости — и `peerDependencies` — зависимости, которые должны работать вместе с другими пакетами, но не зависят от них напрямую. `PeerDependencies` чаще всего используются авторами библиотек.

Давайте заглянем в папку `node_modules` после установки `eslint`. Обратите внимание: там появилось множество новых пакетов (рис. 1.6).

```
efficient-node
{} package.json ×
{} package.json > ...
 8    },
 9    "dependencies": {
10      "lodash": "^4.17.21"
11    },
12    "devDependencies": {
13      "eslint": "^9.11.1"
14    }
15  }

TERMINAL
$ ls node_modules/
@eslint                  esutils                        minimat
@eslint-community        fast-deep-equal                ms
@humanwhocodes           fast-json-stable-stringify     natural-
@nodelib                 fast-levenshtein               optiona
@types                   fastq                          p-limit
acorn                    file-entry-cache               p-locat
acorn-jsx                find-up                        parent-
ajv                      flat-cache                     path-ex
ansi-regex               flatted                        path-ke
ansi-styles              glob-parent                    prelude-
argparse                 globals                        punycod
balanced-match           has-flag                       queue-m
brace-expansion          ignore                         resolve-
callsites                import-fresh                   reusify
color-convert            imurmurhash                    run-par
color-name               is-extglob                     shebang-
concat-map               is-glob                        shebang-
cross-spawn              is-path-inside                 strip-a
debug                    isexe                          strip-j
```

Рис. 1.6. Пакеты npm и их транзитивные зависимости

Пакет `eslint` зависит от всех этих пакетов. Эти транзитивные связи нужно учитывать. При подключении одного пакета проект автоматически становится зависимым от всех его зависимостей, зависимостей подзависимостей и так далее. С установкой каждого пакета вы добавляете в код дерево зависимостей.