GeekBrains

Курс по Node.js

Введение в Node.js. Управление зависимостями

[Node.js v14.x]

На этом уроке

1. Узнаем, что такое Node.js и как его установить на различных платформах.
2. Выведем в консоль свой первый Hello World! тремя разными способами.
3. Узнаём, что такое npm и зачем он нужен в проектах Node.js.
4. Познакомимся с различными видами зависимостей.

Оглавление

Теория урока

Что такое Node.js

Как известно, язык программирования JavaScript появился в качестве инструмента, благодаря которому статические html-страницы в интернете «ожили». Другими словами, он был узкоспециализированным и применялся в одном конкретном месте — при разработке сайтов для осуществления сложной логики на веб-страницах. Но с появлением Node.js в 2009 году всё изменилось.

Node.js — это программная среда, предназначенная для запуска и выполнения JavaScript-программ вне браузера. Благодаря этой среде JavaScript превратился из узкоспециализированного языка в язык общего назначения. И сейчас JavaScript используется не только для создания frontend-части веб-приложений, но и для разработки серверной части, мобильных и десктопных программ, и даже программ для микроконтроллеров!

Node.js в своей основе использует тот же движок — V8, который выполняет JS-код в браузере Chrome. Этот движок разрабатывается и поддерживается компанией Google. Основная задача движка — компиляция JavaScript-кода в машинный код.

Помимо движка V8, в состав Node.js входит библиотека, написанная на C, которая называется libuv. Она позволяет работать с операционной системой — читать и записывать файлы, отправлять и принимать данные по сети и т. д. Эта библиотека предоставляет возможность использовать асинхронные операции ввода или вывода. Более подробно поговорим о ней в следующем уроке.

В состав Node.js также входят стандартные модули:

1. **os** — позволяет получать информацию об операционной системе, в которой работает Node.js.
2. **events** — модуль для работы с событиями. Большая часть Node.js основывается на асинхронной событийно-ориентированной архитектуре. Этот стандартный модуль позволяет создавать инструментарий для работы со своими «кастомными» событиями, а также считается базой для большинства модулей в Node.js.
3. **http** — предоставляет различные функции для создания http-серверов.
4. **streams** — инструментарий для работы с потоками. О потоках поговорим в одном из следующих уроков курса.
5. **path** — модуль для работы с путями к файлам и папкам.
6. **Process** — модуль, доступный в Node.js в качестве глобальной переменной, содержит в себе информацию о текущем процессе Node.js.

7. **url** — модуль для работы с url-адресами. Он используется, чтобы, например, распарсить строку вида https://geekbrains.ru и вычленить из неё протокол, доменное имя и т. д.

В этом списке есть модули, которые чаще всего встречаются в работе. Однако, это далеко не все имеющиеся в Node.js стандартные модули. Полный список — в официальной документации Node.js.

Стандартные модули не нужно устанавливать дополнительно, их можно сразу подключать в файлы, например:

```
const os = require('os')
```

Некоторые же из них доступны как глобальные переменные — например, вышеупомянутый модуль process или класс Buffer, о котором мы поговорим в одном из следующих уроков.

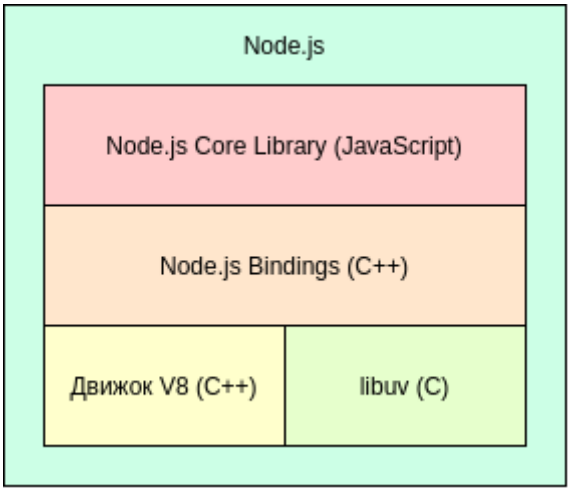


Рис. 1. Структура Node.js

Установка Node.js

Действия, требуемые для установки Node.js на свой компьютер, зависят от операционной системы. В разделе Downloads на официальном сайте есть установщики разных версий Node.js для основных операционных систем — Windows, macOS, Linux. Для установки можно выбрать одну из последних версий — LTS или Current.

LTS расшифровывается как Long Term Support или «долгосрочная поддержка». Это стабильные версии, которые будут поддерживаться в течение 30 месяцев. В LTS-релизы попадают чётные мажорные версии Node.js — первая цифра версии чётная — спустя 6 месяцев после выхода.

Current-версии — это нечётные мажорные релизы, которые после 6 месяцев перестают поддерживаться.

Важно! *В продакшене рекомендуется использовать только LTS-версии. На момент написания последние версии Node.js — 14.15.5 (LTS) и 15.5.1 (Current). Этот курс также предназначается для Node.js не ниже 14 версии.*

В UNIX также используются встроенные консольные инструменты для установки Node.js.

Например, в Ubuntu, чтобы установить Node.js, применяются утилиты apt:

```
apt install nodejs
```

А в masOS — утилиты Homebrew:

```
brew install node
```

Чтобы убедиться в успешной установке, запускается команда, показывающая текущую версию Node.js:

```
node -v
```

Важно! *Обязательно убедитесь, что ваша версия не ниже 14-й, так как на неё рассчитан этот курс. Если установленная версия ниже — установите требуемую вручную. Все версии находятся на официальном сайте Node.js.*

Hello World #1

Чтобы вывести первый Hello World!, используя Node.js, достаточно в консоли операционной системы набрать команду:

```
node
```

Эта команда запускает Node.js в режиме REPL:

- Read;
- Eval;
- Print;
- Loop.

В этом режиме Node.js считывает JS-код, который программист пишет в консоль, выполняет его, показывает результат в консоли и зацикливает эту последовательность.

Чтобы вывести свой первый Hello World!, достаточно просто присвоить это значение в виде строки любой переменной. Пример:

```
Welcome to Node.js v14.15.1.
Type ".help" for more information.
> let foo = "Hello World!"
undefined
```

Значение **undefined** показывает результат, возвращаемый выражением, которое передаётся для выполнения. Здесь операция объявления и присваивания значения переменной ничего не возвращает. Результат выполнения такой команды — **undefined**. Если же в консоли набрать, например, **Date.now()**, то вместо undefined в консоли появится текущее значение timestamp.

Теперь, чтобы увидеть значение переменной foo, достаточно ввести в консоль её название:

```
> foo
'Hello World!'
```

Итак, мы вывели на экран первый Hello World! в Node.js! Режим REPL имеет много разных функциональностей и фишек, более подробно о них можно почитать на официальном сайте. На практике его очень удобно использовать для проверки каких-то новых фишек языка, синтаксиса, чтобы решать какие-то несложные задачи, ради которых нет смысла писать полноценную программу.

Hello World #2

Теперь создадим файл helloworld.js и запишем в него всего одну строчку:

```
console.log("Hello World!");
```

Чтобы запустить этот скрипт в консоли, нам нужно набрать команду:

```
node helloworld.js
```

Команда console.log отработала, и в терминале появилось сообщение Hello World!

Теперь мы поздоровались с миром уже двумя способами!

Резюмируем:

1. Node.js выполняет JS-код в режиме REPL. Для этого требуется вызвать команду node. Дальше в консоли есть возможность вызывать команды JS и сразу же получать результат выполнения.

2. Node.js может запускать и выполнять JavaScript-файлы. Для этого нужно набрать команду node и указать имя файла, который требуется запустить. Тогда js-код внутри файла также выполнится, и если там есть какой-то вывод в консоль — эти данные покажутся на экране.

Параметры командной строки

В начале этого урока мы упоминали модуль process, который доступен в Node.js в качестве глобальной переменной и содержит в себе информацию о текущем процессе Node.js. Полный перечень данных, содержащий этот модуль, есть в официальной документации. Сейчас же мы познакомимся только с одной его составляющей — параметрами командной строки или массивом входящих аргументов **process.argv**. Это массив, в котором доступны все параметры, переданные в node.js-процесс в момент его запуска из командной строки.

Допустим, что мы хотим динамически менять сообщение в предыдущем примере. Причём само сообщение нам нужно передавать в скрипт извне в момент запуска.

Для этого пригодится массив входящих аргументов:

```
console.log(process.argv[2]);
```

Теперь приветственное сообщение указывается напрямую в команде запуска программы:

```
node helloworld.js "Hello World!"
```

В терминале есть тот же результат, что и в предыдущем примере.

Почему элемент массива мы берём с индексом 2, если передаём всего один аргумент? Потому что кроме явно переданного аргумента, при запуске программы в этот массив передаются и неявные аргументы.

1. argv[0] — это путь до программы запуска скрипта. То есть путь до исполняемого файла node.
2. argv[1] — это путь до исполняемого js-файла.
3. argv[2] — это первый передаваемый при запуске аргумент.
4. argv[3] — второй аргумент и т. д.

NPM

Вместе с Node.js на компьютер также устанавливается npm — Node Package Manager. Это очень мощный и неотъемлемый инструмент для разработки на Node.js. У него есть две важные роли:

1. Глобальный репозиторий для публикаций пакетов Node.js. Любой разработчик может написать своё open-source-решение для какой-нибудь задачи, опубликовать его в NPM — и другие разработчики смогут им пользоваться.

2. Управление зависимостями Node.js-проекта. Разработчики используют npm для установки зависимостей проекта, например, библиотек, требуемых для работы проекта. При установке зависимостей через npm они сохраняются в файл package.json.

В этом курсе рассмотрим только второй аспект использования npm.

Файл package.json

Файл package.json лежит в основе любого проекта. Можно сказать, что с создания этого файла и начинается любой проект на Node.js. Такой файл хранит в себе информацию о проекте — название, описание, версию, автора и т. д. Здесь также есть информация для работы с проектом — команды запуска, список различных зависимостей проекта и другое.

Чтобы создать файл package.json, требуется выбрать или создать в консоли директорию, в которой будет находиться проект.

Шпаргалка по взаимодействию с терминалом Linux — в статье на Хабре.

Затем вызывается команда:

```
npm init
```

Это стандартная команда npm. Она предназначена для создания package.json-файла. После вызова этой команды требуется ответить в терминале на несколько вопросов:

1. Выбрать имя проекта (package name). По умолчанию это будет название директории, в которой вызвалась команда.

2. Указать версию проекта (version). По умолчанию — 1.0.0.

3. Заполнить описание (description). По умолчанию — пустая строка.

4. Указать точку входа (entry point). По умолчанию — это файл index.js. Точка входа — файл, который запускается первым при выполнении программы, а из него запускаются уже все остальные файлы проекта.

5. Указать команду для запуска теста проекта (test command). Здесь по умолчанию — пустая строка.

6. Указать репозиторий проекта (git repository). По умолчанию — это пустая строка. Однако здесь можно разместить ссылку на GitHub проекта или на другую систему контроля версий, в

которой хранится проект. Это особенно актуально для модулей open-source. Последними пользуется широкое сообщество разработчиков. Тогда любой разработчик:

- получает возможность открыть package.json-файл модуля;
- найти там ссылку на репозиторий, в котором он хранится;
- сообщить о найденной в модуле ошибке или предложить свой инструментарий к внедрению.

По умолчанию — пустая строка.

7. Указать ключевые слова, относящиеся к этому проекту (keywords). По ним можно искать модули в npm на официальном сайте. По умолчанию — пустая строка.
8. Указать автора проекта (author). Здесь указывается своё имя или email для связи. По умолчанию — пустая строка.
9. Указать тип лицензии для проекта (license). Это поле также в основном актуально для пакетов, которые потом публикуются в npm. Тогда другие разработчики узнают, как использовать наш пакет. По умолчанию используется лицензия ISC, что дословно расшифровывается как «свободная лицензия для программного обеспечения».
10. Подтвердить, что всю информацию мы ввели правильно (Is this OK?). По умолчанию — значение yes. Если ввести любой другой знак, создание package.json отменится.

Так как у каждого вопроса есть свой ответ по умолчанию, и пройти весь процесс создания можно, просто нажимая Enter в ответ на каждый вопрос, у npm есть флаг, позволяющий заранее указать все значения по умолчанию:

```
npm init --yes
```

Затем в директории сформируется файл с примерно таким содержимым:

```
{
  "name": "nodejs_project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Но ведь json-файл — это не код. Как же начать писать код на Node.js?

Hello World #3

Создадим файл index.js и вставим в него строку, выводящую Hello World! как в одном из предыдущих примеров:

```
console.log("Hello World!");
```

Теперь нужно отредактировать package.json и указать, каким образом будет запускаться проект. Для этого добавим команду запуска в поле scripts. Node.js-код запускается командой вида node + <имя_файла>:

```
"scripts": {
    "dev": "node index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

Поле scripts в package.json используется для добавления различных команд, которые требуются для запуска программы. Это команды для запуска проекта в режиме разработки, production или в любых других режимах работы проекта.

Чтобы запустить команды из поля scripts, в терминале требуется вызвать **npm run + <название_команды>**:

```
npm run dev
```

Результат выполнения также отобразится в терминале:

```
> nodejs_project@1.0.0 dev /Users/user/Documents/nodejs_project
> node index.js

Hello World!
```

Управление зависимостями

В результате получился тот же Hello World! Так зачем же использовать файл package.json и создавать там какие-то команды для запуска, ведь достаточно писать файлами js-код и запускать через вызов node?

Кроме всех перечисленных ранее данных, которые хранит в себе package.json-файл, последний имеет ещё одно важное назначение. В этом файле хранится информация о зависимостях проекта.

Зависимости — это дополнительные модули и библиотеки, они используются в программе, при разработке программы и т. д.

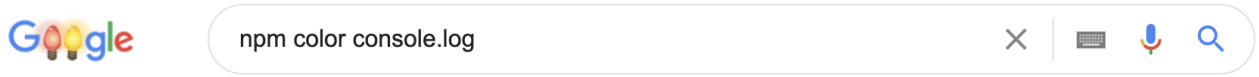
Зависимости бывают разных видов. В Node.js они делятся на четыре типа. Каждый тип зависимости описывается в соответствующем поле в package.json-файле.

Общие зависимости

Это зависимости, требуемые для корректной работы приложения. Они описываются в поле dependencies файла package.json.

Рассмотрим на конкретном примере. В нашем проекте уже есть файл index.js, который выводит строку Hello World! в консоль. Допустим, что требования задачи усложнились, и теперь нужно непросто выводить строку в консоль, а сделать так, чтобы эта строка отображалась в консоли красным цветом.

Можно предположить, что мы далеко не первые, кто сталкивался с подобной задачей. И наверняка кто-то её уже решал. Поищем решение в Node Package Manager, используя Google:



Первая же ссылка в результатах поиска ведёт на страницу модуля colors на сайте npmjs.com. Инструкция по установке модуля на этой странице сообщает, что он устанавливается с применением команды:

```
npm install colors
```

Теперь требуется подключить этот модуль в файле index.js в соответствии с инструкцией на странице модуля:

```
const colors = require("colors/safe");
```

На той же странице в NPM есть информация, как правильно его использовать. Подключим модуль в файл и воспользуемся функциями:

```
const colors = require("colors/safe");

console.log(colors.red("Hello World!"));
```

Теперь запустим проект снова:

```
npm run dev
```

Результат, как и ранее, появится в консоли, но теперь он красный!

Далее надо взглянуть на файл package.json. В нём после установки модуля появилось следующее:

```
"dependencies": {
    "colors": "^1.4.0"
  }
```

Поле dependencies содержит в себе список всех зависимостей, требуемых для этого проекта. Сейчас только один модуль — colors. Его название — это ключ. В качестве значения указывается версия ^1.4.0. Это значит, что установится новейшая второстепенная версия (например, 1.5.0).

Семантическое версионирование

Здесь мы немного отойдём от темы и расскажем о так называемом «семантическом версионировании».

Допустим, модуль colors имеет версию 1.4.0. Вот что означают эти цифры в концепции семантического версионирования:

1. Первая цифра — это мажорная версия. Она повышается, когда в модуль или приложение вносятся несовместимые с предыдущими версиями изменения. У пакета colors мажорная версия — 1.
2. Вторая цифра — это минорная версия. Она повышается, когда модуль или приложение добавляют инструментарий, совместимый с предыдущими версиями. У colors минорная версия — 4.
3. Третья цифра — это патч-версия. Когда изменения совместимы с предыдущими версиями — баг-фиксы, добавления типов и т. д. У пакета colors патч-версия — 0.

Более подробно о семантическом версионировании — на официальном сайте.

Версии пакетов в package.json

Важно упомянуть о системе обозначений версий пакетов при установке и в файле package.json. В примере с модулем colors перед версией модуля стоит символ каретки (^). Этот символ означает, что при установке модуля colors пакетному менеджеру разрешается обновлять минорную версию и

накатывать патчи. То есть устанавливать версии модуля colors от 1.4.0 до 2.0.0, не включая последнюю.

Используются и другие символы:

- «>=» — при установке осуществится поиск версии больше указанной.
- «~» — при установке могут применяться патчи, но без обновлений минорной и мажорной версий.

Теперь вернёмся к зависимостям. Список зависимостей, необходимых для конкретного проекта означает, что без установки модуля colors программа не запустится и выдаст ошибку. И это действительно так, потому что в index.js вызывается функция, явно импортируемая из модуля colors.

Установка таких зависимостей осуществляется:

- запуском команды **npm install + <название_пакета>**, если нужно установить какой-то конкретный пакет;
- запуском **npm install**, если нужно установить все зависимости из списка dependencies.

```
npm install
```

Важно! *Команды установки зависимостей требуется вызывать из директории проекта. В корне проекта всегда должен лежать файл package.json.*

Зависимости разработки

Зависимости нужны для разработки самого приложения. Чаще всего сюда относятся различные линтеры, типы для библиотек, плагины сборщиков кода и т. д.

Такие зависимости устанавливаются командой **npm install + <название_пакета> + --save-dev**.

Например, в проект требуется установить линтер — специальный инструмент, позволяющий соблюдать код-стайл проекта. Линтер используется только в процессе разработки проекта, на конечную функциональность он никак не влияет. Поэтому его следует устанавливать как зависимость разработки:

```
npm install eslint --save-dev
```

В результате в **package.json** появится поле:

```
"devDependencies": {
    "eslint": "^7.15.0"
```

```
  }
```

Все зависимости из **devDependencies** также устанавливаются при вызове npm **install**. Чтобы этого не происходило, например, если проект разворачивается на сервере для работы, а не для разработки, можно передавать флаг **--production**:

```
npm install --production
```

Прямые зависимости

В файле **package.json** описываются в поле **peerDependencies**. Чаще всего используются при публикации собственных пакетов. Необходимость в таких зависимостях возникает, например, при разработке плагина для линтера. Такие плагины разрабатываются для конкретной версии пакета-хоста, но напрямую его не импортируют внутрь своего кода. Соответственно, не имеет никакого смысла добавлять этот пакет в dependencies. Однако необходимость указать, для какой версии пакета-хоста плагин сформировался, никуда не исчезает. Эту зависимость в package.json-файле плагина и указывают как **peerDependecies**.

Например, плагин линтера под названием eslint-plugin-node разработан для **eslint** определённой версии, на момент написания — это >=5.16.0. И в package.json-файле этого плагина eslint указывается в **peerDependencies**:

```
"peerDependencies": {
    "eslint": ">=5.16.0"
  },
```

Важно! *Такие зависимости не устанавливаются при вызове npm install. Если попытаться установить вышеупомянутый плагин без установки самого линтера, то установщик выдаст такое сообщение:*

```
npm WARN eslint-plugin-node@11.1.0 requires a peer of eslint@>=5.16.0 but none
is installed. You must install peer dependencies yourself.
```

При разработке **peerDependencies** нужно указать вручную в package.json-файле.

Необязательные зависимости

Такие зависимости попадают в раздел **optionalDependencies** и устанавливаются командой **npm install + <название_пакета> + --save-optional**, если требуется установить конкретный пакет. Они также устанавливаются при запуске команды npm install, вместе с dependencies и **devDependencies**.

Связанные зависимости

Такие зависимости будут упаковываться вместе с проектом для его публикации. Они указываются в поле **bundledDependencies**. Этот тип зависимостей полезен при разработке проекта:

- если использовались какие-то модули, которых, например, нет в реестре npm;
- если в проекте используются какие-то собственные модули разработчика, которые он по каким-то причинам не опубликовал в npm.

Практическое задание

Напишите программу для вывода в консоль простых чисел, чтобы они попадали в указанный диапазон включительно. При этом числа должны окрашиваться в цвета по принципу светофора:

- первое число выводится зелёным цветом;
- второе — жёлтым;
- третье — красным.

Диапазон, куда попадут числа, указывается при запуске программы.

1. Если простых чисел в диапазоне нет, нужно, чтобы программа сообщила об этом в терминале красным цветом.
2. Если аргумент, переданный при запуске, не считается числом — сообщите об этом ошибкой и завершите программу.

Глоссарий

Current версии Node.js — это нечётные мажорные релизы, которые после 6 месяцев перестают поддерживаться.

LTS версии Node.js — стабильные версии, которые будут поддерживаться в течение 30 месяцев (чётные мажорные версии).

Npm (Node Package Manager) — глобальный репозиторий для публикаций пакетов Node.js, а также инструмент для управления зависимостями Node.js-проекта.

Node.js — это программная среда, предназначенная для запуска и выполнения JavaScript-программ вне браузера.

Package.json — это json-файл, в котором описывается проект (название, версия, автор), указываются команды для работы с проектом и все требуемые для проекта зависимости.

REPL (Read — Eval — Print — Loop) — режим работы, где Node.js считывает вводимый в консоль JS-код, выполняет его, показывает результат в консоли и зацикливает эту последовательность.

Зависимости — это дополнительные модули и библиотеки, которые используются в программе, при разработке программы и т. д.

Зависимости разработки — модули, которые нужны для процесса разработки приложения. Это различные линтеры, типы для библиотек, плагины сборщиков кода и т. д.

Необязательные зависимости — модули, установка которых необязательна для работы проекта.

Общие зависимости — это модули, требуемые для корректной работы приложения.

Прямые зависимости — модули, которые не импортируются напрямую в коде проекта, но требуются для его работы. Например, версия линтера указывается в прямых зависимостях плагина для этого линтера.

Связанные зависимости — зависимости, которые упаковываются вместе с проектом для его публикации в npm.

Дополнительные материалы

1. Официальная документация.
2. Раздел с последними версиями Node.js на официальном сайте.
3. Установка и обновление зависимостей в JavaScript.
4. Основные linux-команды для новичка.

Используемые источники

1. Официальная документация.
2. Семантическое версионирование.