# NPM

Курс на хекслете [тут](https://ru.hexlet.io/courses/js-setup-environment).

## Команды

|  |
| --- |
| $ npm install  Установить все зависимые пакеты из json  –g [пакет1] [пакет2] …  Флаг для глобальной установки пакета. Скачаются также все его зависимости.  $ npm install [пакет]  Локально установить отдельный пакет  --save-dev  Установить все зависимости для разработки из json  --save-dev [пакет]  Установить отдельный пакет в папку для разработки  $ npm link  Установить пакет из локального источника  $ npm unlink [пакет?]  Удалить локально установленный пакет  $ npm uninstall -g [пакет]  Удалить пакет стандартным способом  $ npx <пакет> <опции>  Запустить локально установленный пакет через npx  $ npm init  Создать пакет в каталоге  $ npm login  Выполнить вход локально  $ npm publish  Опубликовать пакет в реестре  $ npm publish --dry-run  отладить процесс публикации без добавления пакета в каталог  $ npm info [пакет]  Посмотреть инфо опубликованного пакета  $ npm update  Обновить все зависимости  $ npm update name  Чтобы выполнить обновление одной конкретной зависимости  $ npm view <packagename> versions  Посмотреть доступные версии пакета  $ npm run sloc  Запустить скрипт  $ npm install --save-dev @babel/core @babel/cli @babel/node @babel/preset-env  @babel/core  содержит код, который выполняет всю работу по трансляции, но не содержит внутри себя правил преобразования. Правила описаны в отдельных пакетах, называемых плагинами (например, *babel-plugin-transform-constant-string*).  @babel/preset-env  основной пресет поддерживаемый командой Babel, который содержит внутри себя плагины, реализующие стандартизированные возможности js (пресет - это группа плагинов, которую можно подключить к Babel целиком).  @babel/cli  командная утилита *babel*.  @babel/node  еще одна утилита командной строки.  Самостоятельно создать файл **babel.config.js**:  module.exports = {  presets: [  ['@babel/env', {  targets: {  node: 'current',  firefox: '60',  chrome: '67',  safari: '11.1',  },  }],  ],  };  $ npx babel src --out-dir dist  Команда берет весь код из директории src и создает его транслированную версию в dist  Команда babel предоставляется пакетом @babel/cli  $ npx babel-node файл.js  Трансляция и запуск в node отдельного файла |

В файл с кодом добавляется строчка, указывающая на внешнюю зависимость. С этого момента зависимость начинает «работать»:

|  |
| --- |
| import [имя в проекте] from 'зависимость';  import \_ from 'lodash'; |

## Установка Node

Для запуска кода на js необходимо скачать и **установить Node.js**. Делать это лучше чере [nvm](https://github.com/nvm-sh/nvm) (node version manager):

|  |
| --- |
| $ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.1/install.sh | bash  Через culr  $ wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.1/install.sh | bash  Через wget  $ sudo apt-get remove nodejs  Удалить node |

Установить curl:

$ sudo apt install curl

**Node.js – это рантайм.**

Программы, которые выполняют интерпретацию кода, называют рантаймом или средой исполнения или интерпретатором. Node.js – это один из многих интерпретаторов.

Для его запуска достаточно набрать $ node.

Чтобы запускать код в файлах, надо создать файл с расширением .js, запустить его указав путь:

$ node [file].js

**REPL**

Node.js поставляется со встроенным REPL. Это программа, которая работает как командная оболочка, расшифровывается как Read Eval Print Loop. Она ожидает ввод от пользователя (Read), выполняет введенный код (Eval) и печатает на экран результат (Print), затем снова входит в режим ожидания (Loop).

Чтобы выйти из REPL:

- набрать .exit

- два раза нажать CTRL + C

## NPM

Оф. сайт [здесь](https://www.npmjs.com/)

Руководство по npm [здесь](https://docs.npmjs.com/)

Шпаргалка по npm на хабре [здесь](https://habr.com/ru/post/133363/)

Node.js поставляется вместе с программой NPM (node package manager?). Главная задача NPM – управление зависимостями.

Зависимость – это когда одна программа (пакет) построена на библиотеках, а другие библиотеки – на третьих, и так до бесконечности. И если одну из библиотек изменят, то может сломаться обратная совместимость. NPM призван этого не допустить. Зависимости автоматически обновляются, устанавливаются и удаляются.

В NPM используются следующие понятия:

**Package** (пакет) – базовая единица, которой управляет NPM как единым целым. Может содержать любое количество файлов. NPM позволяет устанавливать пакеты, обновлять или удалять.

**Registry** (реестр) – хранилище пакетов NPM. Каждый желающий может опубликовать пакет в npm registry. Исходный код пакетов, как правило, хранится на гитхабе. Но пакеты в npm никак не связаны с git и github.

## Установка пакетов

Пакеты можно устанавливать глобально или локально. Инструкция по установке пакета обычно находится на странице [репозитория](https://github.com/flosse/sloc) в гитхабе.

Локальная установка пакетов:

|  |
| --- |
| $ npm install  Установить все зависимые пакеты  $ npm install <импортируемый пакет>  Локально установить отдельный пакет  --save-dev  Установить все зависимости для разработки  --save-dev <пакет>  Установить отдельный пакет для разработки  –g [пакет1] [пакет2] …  Флаг для глобальной установки пакета. Скачаются также все его зависимости.  $ npm link  Установить пакет из локального источника |

В файле package.json несколько секций с зависимостями:

dependencies все зависимости для работы программы

devDependencies для разработки

При **локальной** установке пакет сразу сохраняется в проекте в директории «node\_modules». Если набрать install без аргументов, то автоматом установится всё, что указано в секции dependencies. Поэтому локально устанавливаются все внешние пакеты, которые необходимы для работы твоего пакета.

Минус локальной установки – такие программы нельзя запускать через консоль в любом месте. Если пакет не установлен глобально, то bash скажет, что команда не найдена.

Исполняемые файлы программ ищутся по путям, указанным в переменной окружения **$PATH**. А в случае локальной установки, пакет устанавливается в директорию node\_modules, поэтому bash о ней ничего не знает.

Есть два способа запустить локально установленный пакет:

1. прописать прямой путь в папку node\_modules к исполняемому файлу. Такой способ не рекомендуется к использованию: $ ./node\_modules/.bin/eslint

2. использовать прпограмму npx (см. ниже)

При **глобальной** установке пакет можно просто вызывать из консоли в любом месте.

Минусы глобальной установки – когда ты клонируешь репозиторий, команда $ npm install скачивает и ставит только локальные пакеты из проекта (которые в секции dependencies). Если программа их использует, то пользователям придётся искать их и устанавливать самостоятельно. При этом если у вас будут установлены разные версии, то программа может не заработать.

Пример глобальной установки программ [sloc](https://github.com/flosse/sloc) (считает строчки кода) и [Eslint](https://eslint.org/):

|  |
| --- |
| $ sudo npm install -g sloc  $ sudo npm install -g eslint |

## Удаление пакетов

Чтобы удалить любой пакет стандартным способом, надо набрать

|  |
| --- |
| $ npm uninstall -g [packagename] |

Не добавленное в глобальное хранилище локальное приложение можно удалить стандартным способом (без флага –г?) либо командой

|  |
| --- |
| $ npm unlink [пакет?] |

## NPX

NPX автоматически устанавливается вместе с утилитой npm. Она меняет **$PATH** для конкретного вызова, добавляя туда путь ./node\_modules/.bin. Поэтому начинают работать локально установленные программы.

$ npx просто добавляется слева к любому вызову, и всё:

|  |
| --- |
| $ npx <локальная программа> <опции программы> |

## Создание пакета, JSON

NPM работает таким образом, что объединяет понятия «проект» и «библиотека» в одно целое. И то и другое оформляется, как пакет NPM.

Для создания пакета необходимо выполнить команду в той директории, где вы собираетесь писать код. С этого момента эта папка будет корневой для вашего пакета:

|  |
| --- |
| $ npm init  Создать пакет в каталоге |

После инициализации пакета и ответов на вопросы, в каталоге будет создан файл **package.json**.

[JSON](https://ru.wikipedia.org/wiki/JSON) – это формат для представления структурированных данных. Расшифровывается как JavaScript Object Notation. Достаточно выучить несколько правил по формированию данных в виде JSON, для того, чтобы чувствовать себя комфортно.

Кроме информационных полей типа имени автора, есть и функциональные поля, которые влияют на то, как будет работать пакет.

Если в файле package.json содержатся синтаксические ошибки, то при попытке сделать что-то с проектом будут появляться соответствующие сообщения.

Для проверки корректности оформления файла можно воспользоваться валидаторами, например [jsonlint.com](https://jsonlint.com/) или другими по запросу «json validator».

После завершения процесса инициализации, можно приступать к написанию кода. По умолчанию код пакета создается внутри файла *index.js*, лежащего в корне проекта, где создан файл *package.json*.

## Публикация пакета

Документация по созданию и публикации пакета находится [здесь](https://docs.npmjs.com/creating-and-publishing-unscoped-public-packages).

Для публикации пакета необходимо выполнить следующие условия:

1. Назвать пакет уникальным именем

2. Зарегистрироваться на сайте npmjs.com

3. Выполнить вход локально командой $ npm login (синоним [команды](https://docs.npmjs.com/cli/adduser) $ npm adduser). Только зарегистрированный пользователь может публиковать пакет, а обновлять – автор пакета.

4. В случае обновления, версия нового пакета должна быть выше версии текущего опубликованного пакета.

5. Публикация пакета сводится к запуску команды в корне проекта: $ npm publish

Как только публикация пройдет, ваш пакет автоматически станет доступным для установки. Убедиться в этом можно следующим образом:

1. Установить этот пакет себе: $ sudo npm install -g nodejs-package

2. поискать в каталоге на сайте [npmjs.com](https://www.npmjs.com/)

3. Выполнить команду $ npm info nodejs-package. Если вы увидите информацию о пакете, то все хорошо.

|  |
| --- |
| $ npm login  Выполнить вход локально  $ npm publish  Опубликовать пакет в реестре  $ npm publish --dry-run  отладить процесс публикации без добавления пакета в каталог  $ npm info [пакет]  Посмотреть инфо опубликованного пакета |

**Отладка публикации**

Начиная с 6 версии, npm позволяет отладить процесс публикации без добавления пакета в каталог: --dry-run.

При использовании --dry-run пакет не добавляется в основное хранилище и не доступен в поиске на сайте npmjs.com. В таком случае мы должны установить пакет из локального источника. Для этого существует команда $ npm link. Её нужно вызвать без аргуметов из директории, где расположен наш пакет. В этом случае приложение будет установлено в систему глобально.

Для его удаления используйте команду $ npm unlink либо стандартный способ деинсталяции $ npm uninstall -g <packagename>.

## Версионирование

Версия – это номер либо группа номеров, которая обозначает какой-то конкретный этап разработки данного кода. Разные версии нужны для обратной совместимости. Современное версионирование кода основано на универсальном стандарте [SEMVER](http://semver.org/lang/ru) (семантическое версионирование). По стандарту, версия представляет из себя три числа разделенных точками: *1.23.5*.

Первое число – мажорная версия. Меняется только в случае серьезных изменений, как правило, ломающих обратную совместимость.

Второе число – минорная версия. Не должна ломать обратную совместимость (в идеале). Меняется при добавлении новых возможностей.

Третье число – патч. Гарантировано не должна менять обратную совместимость (к сожалению такое бывает). Меняется только при исправлении багов.

## Зависимости

О зависимостях в документации [тут](https://docs.npmjs.com/files/package.json#dependencies).

Зависимости не хранятся в git-репозитории и должны быть добавлены в **.gitignore**: node\_modules

Чтобы использовать пакет-зависимость в своём проекте, надо:

1. установить его локально в папку **node\_modules**,

2. прописать соответствующую зависимость в файле с кодом.

Установка пакета-зависимости в node-modules осуществляется командой $ npm install <пакет>

После завершения установки появится новый файл package-lock.json

Также, после завершения установки, в существующий файл package.json добавится секция **dependencies**, где будут указываться зависимости:

|  |
| --- |
| "dependencies": {  "пакет": "^4.2.1"  } |

В файл с кодом добавляется строчка, указывающая на внешнюю зависимость. С этого момента зависимость начинает «работать»:

|  |
| --- |
| import [имя в проекте] from 'зависимость';  import \_ from 'lodash'; |

Обратите внимание, что при импорте нет указания текущей директории, как обычно «./». Так NPM понимает, что это внешняя зависимость, а не файл, лежащий рядом.

**Пример**

Возьмем для примера библиотеку *lodash*, которая широко используется в JavaScript и добавляет много маленьких полезных функций: [lodash.com/docs/4.17.11](https://lodash.com/docs/4.17.11).

|  |
| --- |
| $ npm install lodash  Установка в node\_modules  Запись в файле с кодом:  import \_ from 'lodash';  По общепринятому соглашению, lodash импортируется под именем \_  \_.capitalize('hello'); // Hello  capitalize делает первую букву заглавной |

## Зависимости --save-dev

Кроме обычных зависимостей, NPM отдельно обрабатывает зависимости необходимые только во время разработки. Их установка осуществляется командой:

|  |
| --- |
| $ npm install --save-dev <пакет> |

В файле package.json появляется раздел devDependencies.

Эти зависимости никак не используются при публикации пакета. Они скачиваются и сохраняются в директорию node\_modules и импортируются внутри кода как и любой другой код.

|  |
| --- |
| "devDependencies": {  "jest": "^23.6.0",  "jest-cli": "^23.6.0"  } |

## .gitignore и $ npm install»

Сами зависимости не хранятся в git-репозитории и должны быть добавлены в **.gitignore**:

|  |
| --- |
| // в файле .gitignore  node\_modules |

Если вы клонировали репозиторий с гитхаба, директория node\_modules будет отсутствовать.

Команда $ npm install (без аргументов) их установит. Эту команду можно выполнять сколько угодно, ничего не сломается.

## Обновления

|  |
| --- |
| $ npm update  Обновить все зависимости  $ npm update name  Чтобы выполнить обновление одной конкретной зависимости  $ npm view <packagename> versions  Посмотреть доступные версии пакета |

Как будет происходить обновление, зависит от того, что написано в **package.json**:

|  |
| --- |
| dependencies {  'package1': "\*",  'package2': "1.3.5",  'package3': "~2.3.4",  'package4': "^2.3.4",  } |

\* означает, что можно ставить любую версию библиотеки

1.3.5 версия библиотеки жестко зафиксирована

~ в автоматическом режиме обновляются только патчи.

^ обновляется только минорная версия и патчи

## package-lock.json

Документация про package-lock [здесь](https://docs.npmjs.com/files/package-locks).

У каждой зависимости могут быть свои собственные зависимости, которые тоже обновляются. Зависимости зависимостей называются транзитивными.

Первый запуск установки зависимостей формирует ***package-lock.json***. Туда записываются все установленные зависимости, в том числе транзитивные, и их версии. При дальнейших запусках npm install всегда ставится то, что указано в lock файле. Теперь всегда можно быть уверенным, что если заработало сейчас, то заработает потом.

Наличие lock файла никак не влияет на поведение команды update для прямых зависимостей. Если пакет, указанный в package.json, обновился и может быть обновлен до указанной версии, то загрузится новая версия, а файл lock обновится автоматически. После этого нужно не забыть залить его в git-репозиторий.

## [Scripts](https://ru.hexlet.io/courses/js-setup-environment/lessons/scripts/theory_unit#coursenav-modal)

Секция Scripts находится в файле ***package.json*** и позволяет описывать произвольные команды, которые затем можно выполнять.

|  |
| --- |
| "scripts": {  "**sloc**": "sloc",  } |

Предположим, что мы установили программу sloc как зависимость и хотим ее запускать с определенными параметрами.

***Имя*** ***свойства*** используется как команда в командной строке, ***значение*** — исполняемая программа.

Теперь программу из скриптов можно запустить в консоле командой:

$ npm run sloc

За некоторыми именами предлагается закрепить особые значения. Это не просто имена, для них предлагается также упрощённый запуск без указания «run»:

|  |
| --- |
| "publish":  "test":  "stop":  "start":  "restart": |

Например, если есть программа для проверки тестов, можно написать так: "jest": "jest", а можно так: "test": "jest".

Достаточно набрать npm test вместо npm run test, хотя второй вариант также сработает.

**Hooks**

Есть целый набор предопределенных свойств в Scripts, которые NPM запускает автоматически в разные моменты жизни. Подробнее об этом можно прочитать в [официальной документации](https://docs.npmjs.com/misc/scripts). Одним из таких свойств является свойство prepublishOnly, которое используется тогда, когда перед публикацией пакета в репозиторий NPM необходимо выполнить некоторые действия, например, сгенерировать документацию. Команда, указанная в нем, запустится во время выполнения $ npm publish.

## [Babel](https://ru.hexlet.io/courses/js-setup-environment/lessons/babel/theory_unit#coursenav-modal)

Благодаря Babel, можно писать код с использованием всех новых фич, но перед публикацией автоматически транслировать его в старую версию. Потому что реализация зачастую отстает от новой спецификации. Приходится писать код, придерживаясь старых стандартов.

У Babel есть собственный [онлайн REPL](https://babeljs.io/repl/), куда можно вставить код и транслировать его. Такая трансляция называется транспайлингом, а сам Babel называют транспайлером, от [transpiler](https://en.wikipedia.org/wiki/Source-to-source_compiler).

Babel состоит из нескольких пакетов-частей, которые надо скачать отдельно. Ставятся так:

|  |
| --- |
| $ npm install --save-dev @babel/core @babel/cli @babel/node @babel/preset-env  @babel/core  содержит код, который выполняет всю работу по трансляции, но не содержит внутри себя правил преобразования. Правила описаны в отдельных пакетах, называемых плагинами (например, *babel-plugin-transform-constant-string*).  @babel/preset-env  основной пресет поддерживаемый командой Babel, который содержит внутри себя плагины, реализующие стандартизированные возможности js (пресет - это группа плагинов, которую можно подключить к Babel целиком).  @babel/cli  командная утилита *babel*.  @babel/node  еще одна утилита командной строки.  Самостоятельно создать файл **babel.config.js**:  module.exports = {  presets: [  ['@babel/env', {  targets: {  node: 'current',  firefox: '60',  chrome: '67',  safari: '11.1',  },  }],  ],  };  $ npx babel src --out-dir dist  Команда берет весь код из директории src и создает его транслированную версию в dist  Команда babel предоставляется пакетом @babel/cli  $ npx babel-node файл.js  Трансляция и запуск в node отдельного файла |

### babel.config.js

Для работы необходимо вручную создать файл babel.config.js. Через него он узнает, как нужно транслировать код. В свойстве targets перечисляются конкретные окружения (и их версии), для которых пишете код:

|  |
| --- |
| module.exports = {  presets: [  ['@babel/env', {  targets: {  node: 'current',  firefox: '60',  chrome: '67',  safari: '11.1',  },  }],  ],  }; |

### src, dist и .gitingone

При появлении в проекте Babel, изменяется файловая структура проекта. Так как код существует в двух состояниях, исходный код принято хранить в директории src в корне проекта, а код полученный в результате трансляции - в директории dist.

src

отправляется в git репозиторий. В git-репозитории хранится исходный код, ещё не обработанный babel.

dist

публикуется в хранилище пакетов npm и обязательно добавляется в *.gitignore*. Она генерируется только в момент публикации пакета и заливается в npm-репозиторий. Он попадает в NPM-репозиторий в момент публикации новой версии пакета за счет выполнения команды prepublishOnly (в которую вы сами должны прописать вызов трансляции).

По умолчанию NPM смотрит в файл *.gitignore* и все, что там перечислено, не попадет в NPM-репозиторий при публикации пакета. Чтобы dist (который игнорируется) был опубликован, есть два решения. Первое связано с файлом *.npmignore* и описан в [документации](https://docs.npmjs.com/misc/developers#keeping-files-out-of-your-package)

**Секция files в package.json** (для обхода .gitignore)

Второй – NPM позволяет указать список файлов и папок, которые нужно опубликовать. Достаточно добавить секцию *files* в *package.json*. Содержимое *files* — массив директорий и файлов:

|  |
| --- |
| "files": [  "dist"  ] |

### Подготовка пакета и публикация

Перед выполнением команды $ npm publish можно вручную сгенерировать каталог dist, используя скрипты: npx babel src --out-dir dist.

NPM содержит множество предопределённых скриптов, которые выполняются автоматически в определённые этапы работы. Например, скрипт prepublishOnly запускается перед непосредственным выполнением публикации.

|  |
| --- |
| "scripts": {  "build": "NODE\_ENV=production babel src --out-dir dist",  "prepublishOnly": "npm run build"  } |

В примере используется небольшой трюк. В prepublishOnly вызывается другой скрипт — build. Два скрипта нужны для того, чтобы оставить возможность запустить сборку руками (скрипт build – это как раз команда для публикации руками).

### Babel-node

Описание способов запуска babel node из командной строки [здесь](https://babeljs.io/docs/en/babel-node/).

Для того, чтобы нод понимал новый синтаксис, код надо транслировать через @babel/node.

|  |
| --- |
| $ npx babel-node файл.js |

Команда babel-node одновременно транслирует код и сразу же запускает его на выполнение. В отличие от команды babel, babel-node не сохраняет результат трансляции, всё происходит во время работы в памяти.

Вам все равно понадобится правильно настроенный файл **babel.config.js** в корне проекта

### Source Map

Трасляция может сильно изменить исходный код. В результате его будет трудно отлаживать.

- Babel может заменять имена переменных.

- Строчки, на которые указывает [stacktrace](https://en.wikipedia.org/wiki/Stack_trace) — отчет о вызванных функциях после возникновения ошибки в программе, в исходном и транслированном файле не совпадают. Более того, stacktrace указывает только на транслированный код.

- В одних и тех же ситуациях возникают разные ошибки. Например, если вы забыли экспортировать функцию по умолчанию, то после трансляции вы увидите такое сообщение: TypeError: (0 , \_path2.default) is not a function вместо Module 'path' has no default export.

Для решения этой проблемы используется специальный механизм под названием "**source map**" или "маппинг". Его принцип действия следующий. При транспайлинге кода создается структура определенного формата, в которой описана связь сгенерированного кода с исходным кодом. Затем эта структура записывается либо в отдельный, либо прямо в сгенерированный файл в виде комментария (inline mode). Затем, во время выполнения программы, она используется интерпретатором для построения правильных стектрейсов и ссылок.

**Babel** поддерживает source map из коробки. Для его генерации достаточно добавить флаг --source-maps в процесс компиляции:

$ npx babel script.js --out-file script-compiled.js --source-maps inline

$ npx babel src --out-dir dist --source-maps inline

В конце транслированных файлов появится source map в виде комментария. На текущий момент Node.js не имеет встроенной поддержки source map. Их поддержку можно добавить с помощью npm пакета [node-source-map-support](https://github.com/evanw/node-source-map-support).

### Babel и VS Code

Чтобы использовать Babel в VS Code, и дописать туда следующее свойство:

Run -> Open Configurations и вам открывается файл launch.json

"runtimeExecutable": "${workspaceRoot}/node\_modules/.bin/babel-node.cmd",

## Точка Входа

Посмотреть список всех модулей стандартной библиотеки можно в официальной [документации](https://nodejs.org/api/modules.html).

Импорт из **локальных файлов** отличает указание относительного пути (./) в секции from:

|  |
| --- |
| import { length } from './strings';  import factorial from './factorial'; |

В **нелокальных импортах** нет начального ./:

|  |
| --- |
| import path from 'path';  import fs from 'fs';  import lodash from 'lodash'; |

Пакет – не то же самое что и модуль. Модуль – это один файл, а пакет – набор файлов, в которые входят модули и package.json. Если внутри одного пакета множество модулей, то что же импортируется?

В npm принято соглашение, что по умолчанию всегда импортируется файл index.js, находящийся в корне проекта. Этот файл является точкой входа в пакет. Все остальные модули, как правило, собираются внутри index.js. Но это поведение можно изменить. За это отвечает свойство **main** (главный) в файле package.json.

В нашем эталонном пакете nodejs-package эта строчка выглядит [так](https://github.com/hexlet-boilerplates/nodejs-package/blob/master/package.json):

|  |
| --- |
| {  "main": "dist/index.js"  } |

Сам файл *dist/index.js* в репозитории отсутствует, как впрочем и папка *dist*. Как вы помните из урока про *babel*, эта директория формируется только в момент публикации пакета и заливается в npm, но в репозиторий ее не добавляют.

## [Исполняемые файлы](https://ru.hexlet.io/courses/js-setup-environment/lessons/bin/theory_unit#coursenav-modal)

В Ubuntu пользовательские программы в большинстве случаев устанавливаются в директорию */usr*, но программа находится не в одной папке, а разделена на части:

*/usr/bin* - исполняемые файлы программ

*/usr/sbin* - исполняемые файлы программ, которые запускаются с правами администратора

*/usr/games* - исполняемые файлы некоторых игр

*/usr/lib* - библиотеки программы

*/usr/share* - остальные файлы программы

Некоторые разработчики программ предпочитают установку в директорию /opt не разделяя файлы программы.

npm unpublish smaltalk\_2km --force

Многие пакеты представляют из себя приложения командной строки, так называемые [cli](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%BD%D0%BE%D0%B9_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B8)-утилиты. Чтобы после установки приложение запускалось из командной строки, в npm существует секция bin в package.json. В этой секции в формате “ключ: значение” указываются символические ссылки ([symlink](https://en.wikipedia.org/wiki/Symbolic_link)), которые будут созданы в ОС при установке пакета:

|  |
| --- |
| "bin": {  "key": "dist/bin/value.js"  } |

- Ключ означает команду, которая будет вводиться в терминал и запускать файл.

- Значение - это где у тебя в пакете лежит исполняемый файл, который будет запускаться по команде.

**Где символические ссылки хранятся после установки?**

При глобальной установке пакета **npm** создаёт символические ссылки в каталоге по умолчанию /usr/bin (или /usr/local/bin, оно может [конфигурироваться](https://docs.npmjs.com/misc/config) с помощью npm). Путь к нему прописан в переменной окружения PATH.

Если используется Node Version Manager, то для каждой версии создаются отдельные каталоги для ссылок на исполняемые файлы. Главное, чтобы путь к каталогу был в переменной окружения PATH (nvm автоматически добавляет путь).

Важно понимать, что ссылки хранятся в /bin, а сами исполняемые файлы - в /lib/node/modules

Посмотреть, в какую директорию npm складывает *ссылки*на исполняемые файлы можно с помощью команд:

|  |
| --- |
| $ npm bin -g  $ npm bin --global  Куда npm складывает ссылки на исполняемые файлы  $ npm bin  Где лежат ссылки на локально установленные пакеты  $ echo $PATH  Посмотреть содержание переменной окружения PATH  $ echo $PATH | grep /usr/bin  Совместно с grep, чтобы отсеять всё лишнее |

**Как сделать исполняемый файл?**

В Linux не существует расширений файлов (например, .exe или .com или .bat), файлу необходим «режим доступа исполняемого файла». Для возможности запуска файла из командной строки у пользователя должно быть право на выполнение (атрибут x) этого файла.

Чтобы сделать файл исполняемым для всех пользователей:

|  |
| --- |
| $ chmod a+x my\_file  Затем проверить это:  $ ls -l  my\_file |

**Шебанг в исполняемых файлах**

Если исполняемый файл содержит код, то надо указать командной оболочке интерпретатор, который будет исполнять этот код. Это делается с помощью [**шебанга**](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D0%B1%D0%B0%D0%BD%D0%B3_(Unix)). Первой строкой в файле написать #! и указать путь до интерпретатора.

У меня это почему-то не работает. Но есть специальная утилита [**env**](https://ru.wikipedia.org/wiki/Env), которая автоматом ищет указанный интерпретатор. Поэтому в шебанге можно указать путь к env. Это предпочтительно, потому что в разных системах путь до интерпретатора будет разный, а env сама всё найдёт:

|  |
| --- |
| #!/usr/bin/env node |

### **Практика**

Предлагается создать, опубликовать и установить глобально программу, которая при наборе «sayHi» и «sayBye» будет выводить приветствие. В каталоге “SmallTalk\_2km” json и 2 файла: sayBye.js и sayHi.js

|  |
| --- |
| Файл sayBye.js  #!/usr/bin/env node  console.log('Bye-bye, Hexlet!');  Файл sayHi.js  #!/usr/bin/env node  console.log('Hello, Hexlet!'); |

Файл package.json

|  |
| --- |
| {  "name": "smalltalk\_2k",  "version": "1.0.0",  "description": "test project",  "main": "",  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1"  },  "author": "",  "license": "ISC",  "bin" : {  "sayHi": "./sayHi.js",  "sayBye": "./sayBye.js"  }  } |

Авторизируемся, публикуем пакет, устанавливаем его глобально

|  |
| --- |
| $ npm login  $ npm publish  $ npm -g install smalltalk\_2km |

Проверяем каталог с интерпретатором

|  |
| --- |
| $ ls -al /home/noname/.nvm/versions/node/v12.13.0/bin | grep 'say'  ../lib/node\_modules/smalltalk\_2f7a/sayBye.js  ../lib/node\_modules/smalltalk\_2f7a/sayHi.js |

Теперь в любом месте можно набирать команды say и получать приветствие.

### Локальная установка

Для целей разработки пакеты устанавливаются локально в проект и «привязываются» к нему. Они находятся внутри проекта в ./node\_modules/.

**Ссылки** на эти исполняемые файлы размещаются в каталоге ./node\_modules/.bin.

$ npm bin

Без аргументов показывает место хранения ссылок

Такие программы ориентированы на использование в скриптах (секция scripts), для чего существует специальный синтаксис (см. Тему выше).

К символическим ссылкам можно также обратиться напрямую, указав нужный путь. Смотришь, что установлено локально, какие есть символические ссылки, набираешь к ним путь и запускаешь, например:

./node\_modules/.bin/babel-node

У**становка пакета из локального источника**

Можно глобально установить пакет, который находится в ОС (не из репозитория).

$ npm link

Она выполняет глобальную установку пакета, который расположен в операционной системе пользователя, и создаёт необходимые символические ссылки для запуска приложения из любой точки файловой системы. Подробнее о команде [здесь](https://docs.npmjs.com/cli/link).

## Extras: каталоги и разделы

Файл **package.json** создаётся при инициализации пакета. Состоит из секций:

- dependencies, где указываются внешние зависимости, необходимые для работы программы;

- devDependencies, где указываются зависимости, используемые только для разработки. Эти зависимости никак не используются при публикации пакета.

- scripts позволяет описывать произвольные команды, которые затем можно выполнять.

- files

- main – точка входа в пакет.

- bin – для запуска исполняемого файла через консоль

Файл ***package-lock.json*** создаётся при локальной установке пакета.

Директория **node\_modules** создаётся при локальной установке пакета. В ней содержатся все внешние зависимости, перечисленные в секциях dependencies и *devDependencies* файла *package.json*.

Сами зависимости не хранятся в git-репозитории и должны быть добавлены в **.gitignore**:

// .gitignore

node\_modules

Команда $ npm install (без аргументов) выполняет установку пакетов, которые указаны в файле package.json. Эту команду можно выполнять сколько угодно, ничего не сломается.

**package-lock.json.** Первый запуск установки зависимостей формирует lock файл. Туда записываются все установленные зависимости, в том числе транзитивные, и их версии. При дальнейших запусках npm install всегда ставится то, что указано в lock файле. Теперь всегда можно быть уверенным, что если заработало сейчас, то заработает потом.

Наличие lock файла никак не влияет на поведение команды update для прямых зависимостей. Если пакет, указанный в package.json, обновился и может быть обновлен до указанной версии, то загрузится новая версия, а файл lock обновится автоматически. После этого нужно не забыть залить его в git-репозиторий.

**babel.config.js**

*npmignore* и описан в [документации](https://docs.npmjs.com/misc/developers#keeping-files-out-of-your-package)

# Чистые таблицы

|  |
| --- |
|  |

|  |
| --- |
|  |