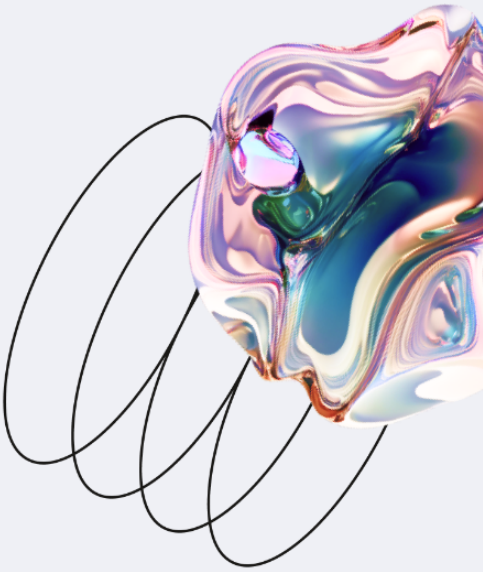
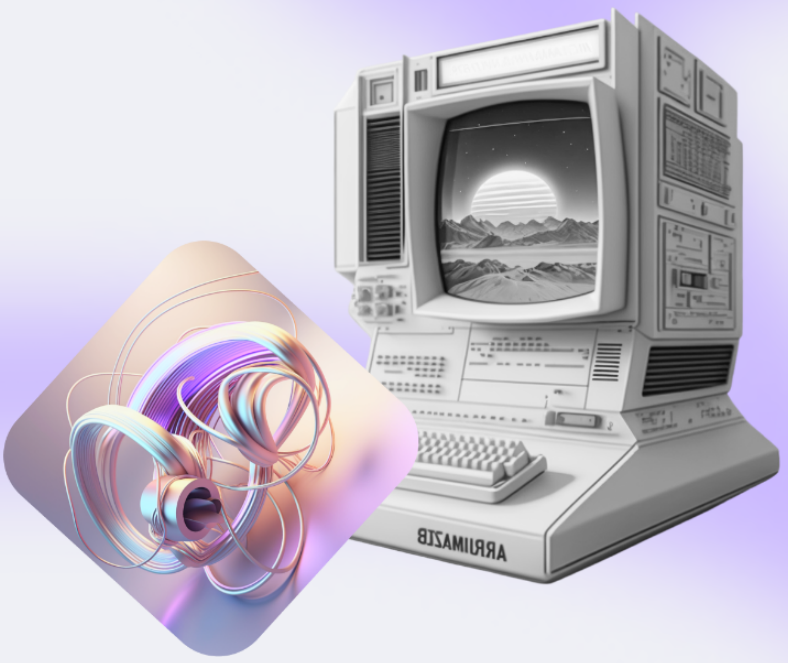
GeekBrains



Введение в NPM

Основы Node.js



Оглавление

Введение

В этой лекции вы познакомитесь с NPM – одним из самых популярных и полезных инструментов для разработчиков на JavaScript. NPM – это менеджер пакетов, который позволяет устанавливать, обновлять и управлять JavaScript библиотеками и фреймворками, которые вы можете использовать в своих проектах. Вы узнаете, как работать с NPM, какие термины и концепции связаны с NPM, какие команды и файлы нужны для работы с NPM, и как опубликовать собственную библиотеку. А также вы познакомитесь с семантическим версионированием.

Что такое NPM?

NPM (Node Package Manager) – это менеджер пакетов для Node.js. Однако NPM не ограничивается только Node.js – вы можете использовать NPM для управления пакетами для любого проекта на JavaScript, например, для фронтенда.

Что такое пакетный менеджер?

Чтобы понять, что такое пакетный менеджер, приведём аналогию.

Для того чтобы приготовить дома какое-нибудь блюдо, вам необходим определённый набор продуктов. В современном мире существует как минимум 2 варианта, как можно получить эти продукты.

Первый вариант – это самостоятельная покупка продуктов в магазине. Вы составляете список продуктов и идёте в магазин. Там вы выбираете нужные вам продукты, оплачиваете и идёте домой готовить блюдо. Но может возникнуть ситуация, когда в ближайшем магазине нет нужного ингредиента. Из-за этого придётся идти в другой магазин и искать нужный ингредиент там.

Второй вариант – это доставка продуктов курьером. Для этого необходимо открыть приложение на смартфоне, набрать в корзину нужные продукты, оплатить и просто ждать, когда курьер принесёт продукты. При этом вы не знаете из какого именно магазина будут привезены продукты, но точно уверены, что через некоторое время все продукты будут у вас дома. Всё это немного упрощает нашу жизнь и экономит время.

Теперь давайте вернёмся в мир Node.js.

Допустим, вы решили написать какое-то приложение (наше блюдо) и знаете, что для его реализации можно использовать несколько готовых библиотек (ингредиенты или продукты), которые написали другие программисты. Тут также существует два варианта, как получить эти библиотеки:

Первый вариант – это найти исходный код нужных библиотек. Для этого вам понадобится поисковик. По названию библиотеки найти сайт (магазин продуктов), где этот код хранится. Например, это может быть Github или Gitlab. Далее вам нужно скачать код и интегрировать его в ваш проект согласно инструкции.

Второй вариант – это использовать пакетный менеджер NPM (приложение для заказа товаров и курьер). Для того чтобы получить код нужной библиотеки, достаточно знать её название и через консоль установить библиотеку в ваш проект. Вам не нужно самостоятельно искать исходный код, так как NPM хранит в себе огромное количество исходных кодов библиотек или, другими словами, пакетов. Как видите, второй вариант проще, чем первый, так как снимает с вас необходимость в самостоятельном поиске.

NPM имеет множество преимуществ для разработчиков на JavaScript:

- Вы можете легко находить и использовать тысячи готовых решений для разных задач, таких как работа с датами, строками, массивами, объектами, HTTP запросами, базами данных и т. д.
- Вы можете контролировать версии пакетов, которые вы используете в своих проектах, и избегать конфликтов и ошибок.
- Вы можете повышать качество и безопасность своего кода, используя пакеты для тестирования, линтинга, форматирования, анализа и т. д.
- Вы можете делиться своими решениями с другими разработчиками и получать обратную связь и поддержку.

На самом деле вы наверняка уже работали с пакетными менеджерами вроде NPM. Microsoft Apps на Windows, App Store на Mac, Google Play на Android, Магазин расширений Google Chrome и многие другие. Всё это пакетные менеджеры, как и NPM. Но NPM, в свою очередь, позволяет распространять исходный код библиотек и фреймворков, а вышеперечисленные пакетные менеджеры призваны распространять программное обеспечение более прикладного характера.

Как установить NPM?

Если вы уже установили Node.js, это значит, что и NPM также установлен на вашем компьютере, так как NPM является неотъемлемой частью Node.js. Поэтому, для того чтобы убедиться, что npm установлен, достаточно ввести в терминал следующую команду `npm -v`. Эта команда позволяет узнать установленную версию npm. Если вы увидели версию, то значит в порядке и NPM у вас есть!

```
student@geekbrains % npm -v
8.10.0
student@geekbrains %
```

Из чего состоит NPM?

NPM состоит из двух основных компонентов: **NPM registry** и **NPM CLI**.

NPM registry. Вы можете искать, просматривать и скачивать пакеты из npm registry с помощью командной строки или браузера. Вы также можете опубликовать собственные пакеты в npm registry и делиться ими с другими разработчиками. Вы всегда можете посетить официальный сайт NPM https://www.npmjs.com/ и поискать необходимые пакеты для вашего проекта, воспользовавшись строкой поиска.

NPM CLI – это интерфейс командной строки для работы с NPM. С помощью NPM CLI вы можете выполнять различные операции с пакетами, такие как установка, удаление, обновление, просмотр информации и т. д.

Как инициализировать проект npm?

Прежде чем мы начнём работу с npm, необходимо познакомиться с таким понятием, как npm проект.

Проект npm – это любая директория на вашем компьютере, которая содержит файл **`package.json`**. Файл **`package.json`** – это специальный файл в формате JSON, который содержит метаданные о вашем проекте, такие как имя, версия, описание, автор, лицензия и т. д. Также в файле **`package.json`** указываются зависимости вашего проекта, то есть пакеты, которые нужны для работы вашего кода.

Для того чтобы инициализировать NPM проект, вам нужно создать файл **`package.json`**. Этот файл можно создать руками, но есть способ более простой, можно запустить команду **`npm init`** в любом каталоге на вашем компьютере, который вы хотите сделать проектом npm. Вам будет задано несколько вопросов о вашем проекте, таких как имя, версия, описание, автор и т. д. Вы можете ответить на них или пропустить нажатием Enter. По умолчанию будут использованы значения из глобальных настроек npm.

После того как вы ответите на все вопросы или пропустите их, вам будет показано содержимое файла **`package.json`**, который будет создан в вашем каталоге.

Что хранится в файле package.json?

Такой JSON объект вы увидите в содержимом файла:

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

Давайте разберёмся, что значат все эти поля:

- **name** – имя вашего пакета, которое должно быть уникальным в реестре npm. Оно должно быть коротким, описательным и должно содержать в себе только прописные буквы, дефисы и цифры;
- **version** – версия вашего пакета, которая должна следовать семантическому версионированию. Она состоит из трёх чисел, разделённых точками: основной версии, подверсии и номера исправления. О семантическом версионировании мы поговорим немного позже;
- **description** – краткое описание вашего пакета, которое помогает пользователям понять, что он делает и зачем он нужен. Оно должно быть одним или двумя предложениями и не содержать HTML-тегов или URL-адресов;
- **main** – эта строка указывает имя файла, который экспортирует основной модуль вашего пакета. Это можно понимать, как точку входа в ваш проект. Об этом поле мы более подробно поговорим немного позже;
- **scripts** – объект, содержащий скрипты проекта, которые могут быть запущены с помощью команды npm run. Ключи являются названиями скриптов, а значения – командами для запуска;
- **author** – объект или строка, которая указывает автора или создателя вашего пакета. Если это объект, он может иметь три поля: name, email и url;
- **license** – эта строка указывает лицензию, под которой распространяется ваш пакет. Можете пока не обращать внимания на значение лицензии, будем использовать значение по умолчанию ISC.

Пример установки пакета

Давайте теперь попробуем установить какой-нибудь пакет в наш npm проект.

Для примера установки возьмём пакет uuid.

Пакет uuid позволяет генерировать уникальные идентификаторы. UUID – это строки из 36 символов, которые состоят из цифр и букв и имеют определённый формат. UUID могут быть полезны для разных целей, например, для создания уникальных имён файлов, ключей в базах данных, токенов аутентификации и т. д.

Для того чтобы установить пакет uuid в ваш проект, вам нужно выполнить команду `npm install uuid`. Эта команда скачает пакет uuid из npm registry и добавит его в ваш проект. Также эта команда обновит файл `package.json` и добавит пакет uuid как ключ в раздел `dependencies`, а также запишет текущую версию пакета в

качестве значения. Это означает, что пакет uuid является зависимостью вашего проекта – то есть ваш код не может работать без него.

Вот пример того, как выглядит файл **`package.json`** после установки пакета uuid:

```
{
  "name": "my_awesome_package",
  "version": "1.0.0",
  "description": "A project to learn npm",
  "main": "index.js",
  "scripts": {
    "test": "echo \\"Error: no test specified\\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "uuid": "^9.0.0"
  }
}
```

Обратите внимание, что версия пакета uuid в начале строки имеет специальный символ `^`. Этот символ означает, что версию пакета необходимо обновлять. Об установке и обновлении зависимостей мы поговорим в главе Установка зависимостей проекта.

Для того чтобы использовать пакет uuid в вашем коде, вам нужно создать новый файл рядом с `package.json` и расширением `.js`. Например, `index.js` как указано на пятой строке в файле `package.json`.

После этого необходимо открыть файл в редакторе кода и вставить туда следующий код:

```
// Импортируем пакет uuid в наш файл
const uuid = require('uuid');

// Генерируем UUID
const id = uuid.v4();
// Выводим сгенерированный идентификатор в консоль
console.log(id);
```

Теперь давайте немного разберём код. Для того чтобы использовать установленные пакеты, необходимо в нашем скрипте получить доступ к этому коду. В Node.js можно использовать функцию `require()`. В функцию нужно передать название пакета, как в файле package.json в поле dependencies. Функция `require()` возвращает экспортируемые методы и поля пакета uuid. Всё это можно записать в переменную и использовать в коде.

В нашем примере мы записали в константу uuid объект с экспортируемыми модулями и полями пакета uuid. Далее, для генерации уникального идентификатора, мы вызвали функцию `.v4()` и получили результат в виде строки.

Для того чтобы запустить код, можно использовать Node.js. Для этого в консоли выполните команду `node index.js`.

node – позволяет запускать JavaScript файлы. Для запуска достаточно написать команду node и через пробел указать имя файла для запуска.

После запуска вы увидите в консоли примерно такой текст:

```
1 1b9d6bcd-bbfd-4b2d-9b5d-ab8dfbbd4bed
```

Как видите, всё довольно просто. У любого пакета, который вы импортируете в ваш проект, есть функции и поля, которые можно использовать.

Если вы все сделали правильно, то могли заметить, что после установки пакета uuid в директории вашего проекта автоматически создалась директория node_modules и файл package-lock.json. Давайте для начала разберёмся, для чего нужен файл package-lock.json.

Что такое node_modules и как он устроен?

node_modules – это каталог в вашем проекте npm, в котором хранятся все установленные пакеты и их зависимости. Когда вы устанавливаете пакет с помощью NPM, он скачивается из npm registry и размещается в каталоге node_modules. Также в каталоге node_modules размещаются все пакеты, от которых зависит установленный пакет.

Например, если вы устанавливаете пакет express, то в каталоге node_modules будут также пакеты body-parser, cookie-parser, debug и т. д., которые нужны для работы express.

Структура папки node_modules зависит от того, какая версия NPM используется для установки пакетов. До третьей версии NPM использовал древовидную структуру, в которой каждый пакет имел свою подпапку node_modules со своими зависимостями. Это приводило к дублированию и избыточности файлов, а также к проблемам с длиной пути файлов на некоторых операционных системах.

Ниже приведён пример зависимостей:

- package3 не имеет зависимостей;
- package2 имеет зависимость от package3;
- package1 имеет зависимость от package 2.

```
.
└── node_modules/
    ├── package1/
    │   ├── node_modules/
    │   │   └── package2/
    │   │       ├── node_modules/
    │   │       │   └── package3/
    │   │       │       ├── node_modules
    │   │       │       ├── package.json
    │   │       │       └── index.js
    │   │       ├── package.json
    │   │       └── index.js
    │   ├── package.json
    │   └── index.js
    ├── package2/
    │   ├── node_modules/
    │   │   └── package3/
    │   │       ├── node_modules
    │   │       ├── package.json
    │   │       └── index.js
    │   ├── package.json
    │   └── index.js
    └── package3/
        ├── package.json
        └── index.js
```

Как видите, пакет package3 содержится в зависимостях package1 и package 2, что является избыточным.

Начиная с третьей версии NPM стал использовать плоскую структуру, в которой все зависимости устанавливаются на одном уровне в папке node_modules. Это уменьшило количество файлов и длину пути файлов, а также упростило разрешение конфликтов версий.

Ниже приведён пример зависимостей:

- package3 не имеет зависимостей
- package2 имеет зависимость от package3
- package1 имеет зависимость от package2

```
.
└── node_modules/
    ├── package1/
    │   ├── package.json
    │   └── index.js
    ├── package2/
    │   ├── package.json
    │   └── index.js
    └── package3/
        ├── package.json
        └── index.js
```

В этом случае невооружённым взглядом видно, что структура стала намного проще. package2 просто переиспользует уже установленный в корне node_modules пакет package3 и package1 также переиспользует package2 из корня node_modules. Обратите внимание, что у пакетов нет собственных директорий node_modules, так как нет необходимости хранить зависимости внутри директории пакета.

> 💡 Начиная с третьей версии NPM, если два пакета требуют разные версии одной и той же зависимости, то NPM создаст подпапку node_modules для одного из пакетов и установит туда нужную версию зависимости.

Каталог node_modules может быть очень большим и содержать много пакетов и файлов, поэтому его не рекомендуется добавлять в систему контроля версий, такую как Git. Вместо этого вы можете добавить файл **`.gitignore`** в ваш проект и указать в нём, что каталог node_modules должен быть игнорирован. Если другой разработчик скачает с Git ваш проект, то он всегда может установить все зависимости с помощью команды npm install, вследствие чего у него появится директория node_modules со всеми зависимостями. Таким образом, вы сэкономите место и время при работе с вашим проектом.

Установка зависимостей проекта

Когда проект инициализирован и опубликован в системе контроля версий, его может скачать любой разработчик команды разработки для работы с кодом. Как вы помните, node_modules не стоит добавлять в систему контроля версий, по этому разработчику, чтобы запустить проект, необходимо установить все зависимости. Для этого он может выполнить две команды: npm install или npm ci

Отличие npm install от npm ci

npm install (без указания конкретного пакета) создаёт директорию node_modules и скачивает туда все зависимости, указанные в package.json, а также создаст файл package-lock.json. Если node_modules и package-lock.json уже существуют, npm install проверит, все ли пакеты, которые есть в package.json, есть в node_modules и установит недостающие пакеты. Также важно понимать, что npm install обновит пакеты, у которых в версии есть специальный символ ^ до более актуальных и обновит информацию о версиях в package-lock.json. Если специальный символ ^ не указан в версии пакета, то пакет обновлён не будет.

npm ci (clean install) так же как и npm install, создаст директорию node_modules, если она отсутствует, но при этом версии пакетов будет сверять с файлом package-lock.json, при этом версии пакетов npm ci не обновляет. Если директория node_modules уже существует, то npm ci удалит её и создаст, заново скачав туда все необходимые зависимости. Это более безопасный способ установки зависимостей, так как иногда автоматическое обновление пакетов с помощью npm install может привести к поломке вашего кода, если разработчики пакета допустили ошибку в новой версии пакета, хотя такое происходит крайне редко.

Удаление пакетов

NPM позволяет удалить установленный пакет, если он вам больше не нужен. Для этого существует команда npm uninstall <имя_пакета>.

Эта команда удалит все файлы, связанные с пакетом, из node_modules, а также удалит строку с названием и версией пакета из package.json.

Также вы можете удалить пакет, удалив в package.json строку с его упоминанием и набрав в консоли команду npm ci или npm install. Эти команды удаляют из node_modules любые пакеты, которые не упоминаются в node_modules.

Что такое package-lock.json?

package-lock.json – это специальный файл, который создаётся и обновляется автоматически при установке или обновлении пакетов npm. Этот файл содержит точную информацию обо всех установленных пакетах в вашем проекте и их зависимостях, включая номера версий, хеши и пути к файлам.

package-lock.json нужен, для того чтобы гарантировать, что вы и другие разработчики, работающие с вашим проектом, используете одинаковые версии пакетов и зависимостей. Это помогает избежать проблем с несовместимостью или ошибками при запуске или развёртывании вашего проекта.

package-lock.json гарантирует, что если другой разработчик скачает ваш проект с Git и установит зависимости с помощью команды `npm ci`, то все версии пакетов будут такими же, какие и у вас.

Вы не должны редактировать файл `package-lock.json` вручную или удалять его из вашего проекта. Вы должны добавить его в систему контроля версий, чтобы сохранять его историю и синхронизировать его с другими разработчиками.

Вот пример того, как выглядит файл package-lock.json после установки пакета uuid:

```json
{
  "name": "my_awesome_package",
  "version": "1.0.0",
  "lockfileVersion": 2,
  "requires": true,
  "packages": {
    "": {
      "name": "my_awesome_package",
      "version": "1.0.0",
      "license": "ISC",
      "dependencies": {
        "uuid": "^9.0.0"
      }
    },
    "node_modules/uuid": {
      "version": "9.0.0",
      "resolved": "<https://registry.npmjs.org/uuid/-/uuid-9.0.0.tgz>",
      "integrity": "sha512-MXcSTerfPa4uqyzStbRoTgt5XIe3×5+42+q1sDuy3R5MDk66URdLMOZe5aPX/SQd+kuYAh0FdP/pO28IkQyTeg==",
      "bin": {
        "uuid": "dist/bin/uuid"
      }
    }
  },
  "dependencies": {
    "uuid": {
      "version": "9.0.0",
      "resolved": "<https://registry.npmjs.org/uuid/-/uuid-9.0.0.tgz>",
      "integrity": "sha512-MXcSTerfPa4uqyzStbRoTgt5XIe3×5+42+q1sDuy3R5MDk66URdLMOZe5aPX/SQd+kuYAh0FdP/pO28IkQyTeg=="
    }
  }
}
```

💡 Мы не будем детально разбирать файл `package-lock.json` в рамках этой лекции. Разработчики редко правят этот файл для решения каких-либо проблем. Но если интересно, можете почитать об этом файле в официальной документации.

Что такое dependencies и devDependencies и в чём их различие?

Ещё стоит поговорить про разные типы зависимостей. Ранее мы установили пакет uuid, и в package.json он добавился в объект dependencies. Но существует другой вариант установки пакета.

Когда вы устанавливаете пакеты в ваш проект с помощью NPM, вы можете указать тип зависимости для каждого пакета: dependencies или devDependencies.

dependencies – это зависимости, которые необходимы для работы вашего кода в продакшене – то есть когда вы запускаете или разворачиваете свой проект на реальном сервере или клиенте. Например, если вы создаёте веб-приложение на React, то React является зависимостью типа dependencies.

devDependencies – это зависимости, которые необходимы только для разработки вашего кода – то есть когда вы пишете, тестируете или отлаживаете свой код на локальном компьютере. Например, если вы используете Jest для тестирования своего кода, то Jest является зависимостью типа devDependencies.

Разделение зависимостей на два типа помогает оптимизировать размер и скорость вашего проекта в продакшене, так как вам не нужно устанавливать и загружать лишние пакеты, которые не используются в продакшн среде.

Для того чтобы установить пакет как зависимость типа dependencies, вы должны использовать команду **`npm install <package-name>`** без дополнительных флагов. Для того чтобы установить пакет как зависимость типа devDependencies, вы можете использовать команду **`npm install <package-name> --save-dev`** или **`npm install <package-name> -D`**.

Вот пример того, как выглядит файл **`package.json`** после установки пакетов React и Jest как зависимостей разных типов:

```json
{
  "name": "my-npm-project",
  "version": "1.0.0",
  "description": "A project to learn npm",
  "main": "index.js",
  "scripts": {
    "test": "echo \\"Error: no test specified\\" && exit 1"
  },
  "dependencies": {
    "react": "^17.0.2"
  },
  "devDependencies": {
    "jest": "^27.2.5"
  }
}
```

💡 Если вам сейчас не до конца понятно, для чего такое разделение – это нормально! Со временем вы начнёте понимать, какой пакет является зависимостью с типом devDependencies, а какой с типом dependencies. Пока что можете устанавливать все пакеты обычным способом с помощью команды **`npm install <package-name>`**.

Что такое скрипты NPM?

Теперь давайте разберёмся с ещё одной возможностью npm. Вы можете заметить, что в package.json есть поле `scripts`. Сюда вы можете помещать различные скрипты, которые вам помогут в разработке.

Скрипты NPM – это способ автоматизировать различные задачи, связанные с вашим проектом. Вы можете запускать скрипты npm с помощью команды **`npm run <script-name>`** в командной строке, где `<script-name>` – это название вашего скрипта. Названием скрипта является само поле в объекте `scripts`.

Скрипты npm могут выполнять различные действия, такие как компиляция, форматирование, проверка, тестирование, сборка или публикация вашего кода.

Например, вы можете создать скрипт для запуска вашего кода, который генерирует уникальный идентификатор и выводит его в консоль.

Для этого необходимо добавить в объект `scripts` новое поле `start` со значением `node index.js`:

```
{
  "name": "my_awesome_package",
  "version": "1.0.0",
  "description": "A project to learn npm",
  "main": "index.js",
  "scripts": {
    "test": "echo \\"Error: no test specified\\" && exit 1",
        "start": "node index.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "uuid": "^9.0.0"
  }
}
```

Для использования нашего нового скрипта необходимо в консоли написать следующую команду: `npm run start` и нажать Enter. В консоли будет примерно следующий текст:

```
npm run start

> my_awesome_package@1.0.0 start
> node index.js

3a7b4dbf-8bc7-49fa-a69c-f5e611f40bb5
```

Фактически мы выполнили команду `node index.js`, то есть запустили свой код, но через npm, и получили в выводе уникальный идентификатор.

Конечно, такой простой скрипт избыточен и наверно не сильно упрощает жизнь. Но когда вы начнёте писать серьёзные проекты, вы сможете использовать эти скрипты, для упрощения запуска нескольких команд.

Также вы могли заметить, что в `scripts` есть поле `test` – это тестовый скрипт, который создаётся по умолчанию в проекте npm. Если запустите этот скрипт `npm run test` , то увидите сообщение в консоли `Error: no test specified`. Базово этот скрипт нужен для указания команды на запуск тестов. Но с тестами вы познакомитесь позже, а пока можете удалить этот скрипт.

В итоге наш файл package.json должен выглядеть следующим образом:

```json
{
  "name": "my_awesome_package",
  "version": "1.0.0",
  "description": "A project to learn npm",
  "main": "index.js",
  "scripts": {
        "start": "node index.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "uuid": "^9.0.0"
  }
}
```

Как создать собственный пакет?

Теперь самое интересное! Мы попробуем создать свою первую библиотеку и опубликуем её. Но прежде чем это делать, нам необходимо разобраться с экспортом функций и переменных из модуля.

Экспорт из модулей

Экспорт функций и переменных в node.js позволяет использовать их в других файлах или модулях, позволяя повторно использовать код и упрощая отладку и поддержку приложения.

Для экспорта функций и переменных в node.js нужно использовать объект **`module.exports`**, который является свойством глобального объекта **`module`**. Этот объект содержит всё то, что будет доступно для импорта в других файлах или модулях.

Давайте создадим файл в нашем проекте и назовём его `math.js`

Откроем файл и добавим туда две функции: `add()` – функция сложения чисел и `subtract()` – функция вычитания чисел:

```js
function add (x, y) {
  return x + y;
}

function subtract (x, y) {
  return x - y;
}

// Здесь мы нашему модулю указываем какие функции можно импортировать в других файлах
module.exports = { add, subtract };
```

Для импорта функций и переменных в node.js нужно использовать уже знакомую вам функцию **`require()`**. Она принимает путь к файлу или модулю, из которого нужно импортировать. Эта функция возвращает объект с импортированными данными, который можно сохранить в переменной и использовать для вызова функций или доступа к переменным.

Например, если мы хотим импортировать функции `add` и `subtract` из файла `math.js` в файл `index.js`, мы можем сделать следующее:

```js
// Импортируем пакет uuid в наш файл
const uuid = require('uuid');
// Импортиируем наш новый модуль math.js
const math = require('./math');

// Генерируем UUID
const id = uuid.v4(); // Выводим сгенерированный идентификатор в консоль
console.log(id);

// Складываем 2 числа и сохраняем результат в константе
const result = math.add(5, 5);
console.log(result); // Выводим в консоль результат сложения двух чисел
```

💡 Обратите внимание на два вызова функции `require()`.

В первом случае `require('uuid')` нет необходимости указывать относительность пути, то есть добавлять `./` или `../` , так как это установленный npm пакет.

> Во втором случае также можно не указывать относительность пути `./` , если файл находится в той же директории, в которой мы импортируем модуль, но следует помнить, что если файл будет находиться в другой директории, например, `libs` , то нам нужно будет вызвать `require` следующим образом: `require('./libs/math.js')`. Для того чтобы не путать локальные файлы с установленными пакетами, рекомендуется всегда указывать относительность пути, как в примере с кодом: `require('./math');`
>
> Также стоит обратить внимание на то, что мы не указали расширение файла `math`. Это допустимо в node.js. Node.js позволяет указывать файлы как с расширением, так и без.

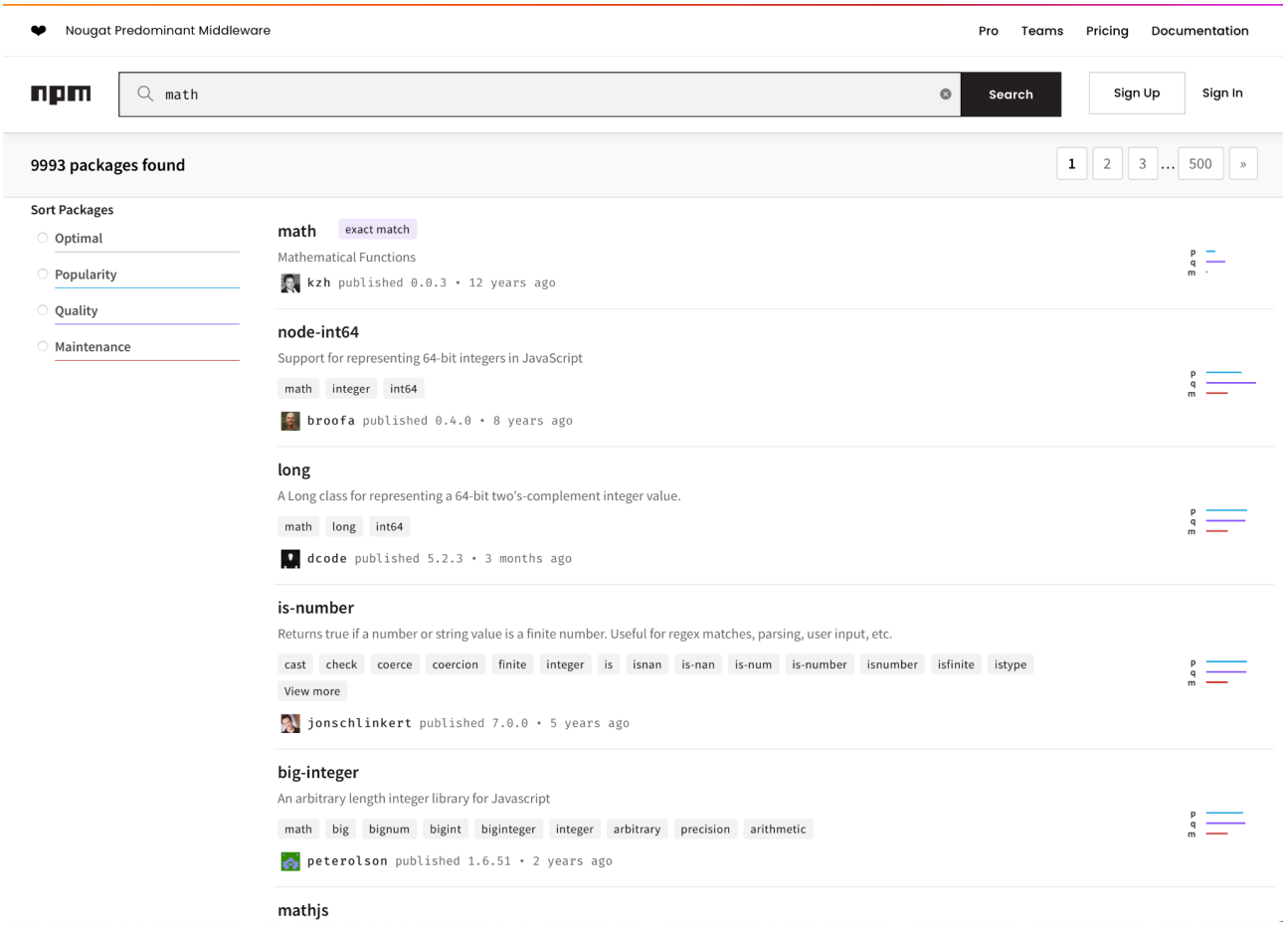
Теперь вы понимаете, что код в проекте можно писать в разных файлах и импортировать с помощью `require()` в любом месте.

В целом, если заглянуть в исходный код любого npm пакета, то можно увидеть, что npm пакеты устроены точно так же, как и наш собственный модуль `math.js` . Например, функция v4 пакета uuid – это функция, которую экспортирует главный модуль пакета.

Теперь, зная то, как работает импорт и экспорт в модулях, давайте приступим к публикации нашего собственного пакета!

Публикация собственного пакета

Прежде чем публиковать свой пакет, давайте разберёмся с тем, как искать и просматривать пакеты других пользователей. Для этого нужно зайти на официальный сайт NPM и в строке поиска ввести любое название пакета, которое вам необходимо. Например, если ввести слово math, то мы увидим ряд пакетов, которые содержат в названии, в описании, либо в тегах слово math.



Когда мы создадим свой пакет, то мы также сможем увидеть его в результатах поиска.

Теперь давайте создадим новый npm проект. Выполним те шаги, которые мы выполняли ранее:

Инициализируем проект

```
npm init -y
```

Создадим файл `math.js` и заполним его следующим кодом:

```
function add (x, y) {
  return x + y;
}

function subtract (x, y) {
  return x - y;
}

module.exports = { add, subtract };
```

Да, это такой же код, как и в модуле `math.js`. Наш пакет будет помогать нам складывать и вычитать числа.

Также необходимо рядом с package.json создать файл README.md с описанием проекта. Это описание будет использоваться на сайте npm.

Добавим туда следующее содержимое:

```
# My awesome math package
This package allows you to add and subtract numbers!
```

Теперь необходимо подготовить файл `package.json`

- Нужно придумать название нашего пакета и написать в поле `name`. Оно должно быть уникально в рамках npm registry. Проверить уникальность достаточно просто: нужно зайти на официальный сайт NPM и в поисковой строке поискать по названию, которое вы хотите дать своему пакету, другие пакеты с таким же названием. Если поиск не дал результатов – это означает, что имя не занято и его можно использовать.
- Также нужно добавить описание нашего пакета в поле `description`
- И ещё не забываем указать себя в качестве автора пакета в поле `author`
- Теперь обратите внимание на поле `main` . В этом поле хранится путь до файла, который будет использован, как файл для импорта. То есть пользователи, скачавшие ваш пакет, когда в `require()` укажут название вашего пакета, получат результатом выполнения функции `require()` значение, которое экспортируется из модуля. В нашем случае необходимо заменить `index.js` на `math.js`

Давайте посмотрим, что получилось в итоге:

```
{
  "name": "alex_math_package",
  "version": "1.0.0",
  "description": "A package that allows you to add and subtract numbers",
  "main": "math.js",
  "scripts": {
    "test": "echo \\"Error: no test specified\\" && exit 1"
  },
  "author": "Aleksandr Mirovov",
  "license": "ISC"
}
```

Теперь публикуем! Для этого необходимо проделать некоторые действия:

- Создайте аккаунт на https://www.npmjs.com/ или войдите в свой существующий аккаунт.
- Подтвердите свой адрес электронной почты.
- В терминале авторизуйтесь в npm registry с помощью команды **`npm login`**. Эта команда отправит вас в браузер и попросит ввести код из письма на почте.
- Опубликуйте ваш пакет в npm registry с помощью команды **`npm publish`**
- Проверьте, что ваш пакет появился на сайте https://www.npmjs.com/ и доступен для установки и использования.

Теперь можно устанавливать этот пакет и переиспользовать его код в любом проекте! Достаточно установить его `npm install alex_math_package`

Что такое семантическое версионирование?

Давайте теперь разберёмся, как работают версии пакетов в проекте. Для версионирования пакетов используется семантическое версионирование.

Семантическое версионирование – это система правил и соглашений для присвоения номеров версий пакетам. Семантическое версионирование помогает разработчикам понимать, какие изменения произошли в пакете при обновлении его версии, и как эти изменения могут повлиять на совместимость с другими пакетами.

Семантическое версионирование использует следующий формат для номеров версий:

`major.minor.patch`

Где:

- **major** – это мажорная версия, которая увеличивается, когда в пакете происходят серьёзные изменения, которые ломают обратную совместимость с предыдущими версиями. Например, если в пакете изменяется API, удаляются функции или меняются зависимости.
- **minor** – это минорная версия, которая увеличивается, когда в пакете добавляются новые функции или улучшения, которые не ломают обратную совместимость с предыдущими версиями. Например, если в пакете добавляются новые опции, параметры или методы.
- **patch** – это патч-версия, которая увеличивается, когда в пакете исправляются ошибки или баги, которые не влияют на функциональность или совместимость с предыдущими версиями. Например, если в пакете исправляются опечатки, улучшается производительность или безопасность.

При изменении более старшей версии, более младшие обнуляются. Например, если у нас есть версия `2.5.2` и увеличивается минорная версия, то получится `2.6.0`, то есть патч версия обнулилась.

Сейчас я приведу примеры увеличения версии пакета в зависимости от изменений в коде.

Допустим, вы создали простую функцию для сложения чисел и решили опубликовать ее в качестве пакета с названием `math`:

```
function add(a, b) {
  return a + b;
}

module.exports = { add };
```

Установим в package.json версию `1.0.0` и опубликуем `npm publish` .

Теперь можно установить наш пакет:

```
npm install math
```

Импортируем пакет и используем:

```
const math = require('math');

console.log(math.add(5, 5)); // 10
```

Работает, как и ожидается.

Теперь давайте внесём некоторые изменения в код пакета. Например, мы хотим добавить функцию вычитания:

```
function add(a, b) {
  return a + b;
}

function subtract (x, y) {
  return x - y;
}

module.exports = { add, subtract };

```

Подумайте, какую версию теперь нужно установить нашему пакету перед публикацией?

Так как мы не изменили старый функционал и просто добавили новый, то необходимо поднимать минорную версию, то есть цифру посередине.

Получается, что новая версия нашего пакета теперь `1.1.0`

Если вернуться к использованию нашего пакета, то мы увидим, что для нас ничего не поменялось, кроме того, что мы можем использовать функцию `subtract`, помимо `add`

```
const math = require('math');

console.log(math.add(5, 5)); // 10
console.log(math.subtract(10, 5)); // 5
```

Теперь давайте снова попробуем внести изменения в код нашего пакета. Поменяем названия аргументов функций, чтобы они были похожи друг на друга:

```
function add(a, b) {
  return a + b;
}

// Раньше было (x, y), теперь (a, b)
function subtract (a, b) {
  return a - b;
}

module.exports = { add, subtract };
```

Как видите, функционал у нас совсем не изменился, мы просто изменили названия аргументов. Такие изменения должны поднимать `patch` версию пакета. Теперь версия нашего пакета должна быть `1.1.1`

Если вернуться к использованию нашего пакета, то для нас снова ничего не изменилось

```
const math = require('math');

console.log(math.add(5, 5)); // 10
console.log(math.subtract(10, 5)); // 5
```

А теперь давайте представим, что мы хотим изменить название функции нашего пакета, например, вот так:

```
function add(a, b) {
  return a + b;
}

// Раньше было subtract, теперь sub
function sub (a, b) {
  return a - b;
}

module.exports = { add, sub }
```

В таком случае нужно поднимать мажорную версию на `2.0.0`, так как мы сломали обратную совместимость. Нашей библиотекой уже пользуются, изменив название

функции и не изменив мажорную версию, у пользователей при обновлении библиотеки сломается код, так как функции `subtract` большее не существует!

```
const math = require('math');

console.log(math.add(5, 5)); // 10
console.log(math.subtract(10, 5)); // TypeError: math.subtract is
not a function
```

Теперь вы понимаете, что значат цифры в версиях пакетов npm.

Какие ещё есть команды для npm cli?

Давайте теперь в целом посмотрим на список основных команд npm cli:

- **`npm help`** или **`npm h`** – выводит справочную информацию о командах npm cli;
- **`npm uninstall <package-name>`** или **`npm un <package-name>`** – удаляет пакет из вашего проекта и из файла **`package.json`**;
- **`npm list`** или **`npm ls`** – выводит список всех установленных пакетов в вашем проекте и их версии;
- **`npm view <package-name>`** или **`npm v <package-name>`** – выводит информацию о пакете из npm registry, такую как имя, описание, версия, лицензия, зависимости и т. д.;
- **`npm search <keyword>`** или **`npm s <keyword>`** – ищет пакеты в npm registry по ключевому слову и выводит их названия и описания.

💡 Это не полный список команд для npm cli, а только самые часто используемые. Вы можете найти больше команд и подробности о них на официальном сайте NPM.

Подведение итогов

В этой лекции мы с вами познакомились с NPM: узнали, для чего он нужен, как его использовать и как устанавливать пакеты. Также поняли как можно опубликовать собственный пакет и разобрались с семантическим версионированием.

NPM очень мощный инструмент, который помогает разработчикам реализовывать проекты любого уровня за счёт огромного количества пакетов на любые случаи жизни.

В следующей лекции мы применим все полученные знания для реализации http сервера на основе легковесного фреймворка express.

Домашнее задание

- Инициализируйте NPM проект и попробуйте установить пакет uuid
- Создайте файл index.js и импортируйте пакет uuid
- Попробуйте воспользоваться возможностями пакета uuid

Что можно почитать ещё?

Если вы хотите узнать больше о NPM и его возможностях, вы можете почитать следующие ресурсы:

- Официальная документация NPM на https://docs.npmjs.com/, где вы найдете подробные инструкции и примеры по использованию NPM и его команд.
- Полное описание правил семантического версионирования https://semver.org/lang/ru/

Словарь терминов

- **Пакет** – это набор файлов кода, который можно установить и использовать в своём проекте. Пакет может содержать один или несколько модулей JavaScript, а также другие ресурсы, такие как документация, тесты, конфигурации и т. д. Пакеты могут быть созданы вами или другими разработчиками и опубликованы в npm registry.
- **Модуль** – это отдельный файл кода JavaScript, который экспортирует одну или несколько функций, объектов или переменных. Модуль может быть импортирован в другой модуль с помощью оператора `require` или `import`. Модули позволяют разбивать код на логические части и повторно использовать его в разных местах.
- **NPM registry** – это онлайн-репозиторий, в котором хранятся все пакеты, опубликованные с помощью NPM. Вы можете искать, просматривать и скачивать пакеты из npm registry с помощью командной строки или браузера. Вы также можете опубликовать собственные пакеты в npm registry и делиться ими с другими разработчиками.
- **NPM CLI** – это интерфейс командной строки для работы с NPM. С помощью NPM CLI вы можете выполнять различные операции с пакетами, такие как установка, удаление, обновление, просмотр информации и т. д.
- **Проект npm** – это любой каталог на вашем компьютере, который содержит файл `package.json`. Файл `package.json` – это специальный файл в формате JSON, который содержит метаданные о вашем проекте, такие как имя, версия, описание, автор, лицензия и т. д. Также в файле `package.json` указываются зависимости вашего проекта – то есть пакеты, которые нужны для работы вашего кода.
- **Семантическое версионирование** – это система правил и соглашений для присвоения номеров версий пакетам. Семантическое версионирование помогает разработчикам понимать, какие изменения произошли в пакете при обновлении его версии, и как эти изменения могут повлиять на совместимость с другими пакетами.