GeekBrains

Курс по Node.js

CLI-приложения

[Node.js v14.x]

На этом уроке

1. Узнаем, что такое CLI-приложения (консольные приложения).
2. Напишем собственную CLI-утилиту для чтения файлов.
3. Изучим различные способы взаимодействия пользователей с программой, передачи в неё данных и настроек.
4. Познакомимся с понятием «исполняемый файл».
5. Оформим программу как самостоятельное приложение, которое можно запустить в любом месте на компьютере.

Оглавление

Теория урока

Введение

CLI-приложения на Node.js — это такие приложения, взаимодействие с которыми осуществляется через интерфейс командной строки (терминал). Аббревиатура CLI так и расшифровывается — Command Line Interface. Они также называются «консольные приложения».

В виде консольных приложений оформляются:

- многие написанные на Node.js сборщики кода (например, webpack);
- различные модули для создания бойлерплейтов — заготовки проектов, в которых уже подключены все важные составляющие;
- многие другие утилиты.

Разработчики часто пишут CLI-приложения для личного использования, чтобы автоматизировать рутинные задачи. Бывают консольные приложения для администрирования сервера, сетей, работы с файлами, обработки данных и т. д.

Мы уже использовали терминал для запуска наших программ и даже передавали через него параметры для запуска программы. В этом уроке углубимся в эту тему и рассмотрим другие возможности и приёмы создания CLI-приложений на Node.js. Для этого используем не только стандартный инструментарий Node.js, но и сторонние модули. Последние установим, воспользовавшись npm.

Инициализация

В этом уроке мы создадим CLI-приложение, идея которого базируется на материале прошлого урока. Напишем консольное приложение, позволяющее пользователю выбрать любой файл из текущей директории, а также вывести его содержимое в терминал. Для удобства и простоты воспользуемся только асинхронными функциями чтения или записи (без использования потоков). То есть приложение не станет работать с большими файлами. Однако в этом практическом задании мы применим знания о создании консольных приложений к практическому заданию прошлого урока. В результате получится готовый инструмент для работы с текстовыми файлами любого размера.

Вспоминаем материалы первого урока и создаём проект (файл package.json):

```
npm init --yes
```

Теперь создадим файл index.js и добавим туда инструментарий для асинхронного чтения файла:

```
const fs = require("fs");

fs.readFile("./access.log","utf8", (err, data) => {
    console.log(data);
});
```

Занесём в package.json команду запуска приложения:

```
"scripts": {
```

```
  "start": "node index.js"
},
```

Таким образом, мы начинаем совмещать ранее изученный материал. Теперь снова применим знания из первого урока и зададим путь к читаемому файлу через параметры командной строки.

Параметры командной строки

Мы уже сталкивались с этим понятием в первом уроке. Вспомним, что параметры командной строки — это массив, в котором доступны все параметры, переданные в Node.js в момент запуска программы из командной строки.

Внутри приложения этот массив доступен по ключу argv глобального объекта process (process.argv). Структура массива:

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.

В первом уроке для работы с параметрами командной строки мы использовали встроенный в глобальный объект process массив argv. Это стандартный инструментарий Node.js. В этом же уроке воспользуемся сторонними модулями. Они предоставляют более разнообразный инструментарий и дополнительные возможности.

Экосистема Node.js настолько велика, что почти под каждую задачу можно найти модуль. Однако, если главное для приложения — безопасность и стабильность — сторонним модулям требуется уделять повышенное внимание, и даже заглядывать к ним «под капот».

Не стоит работать с модулями, которые уже давно не поддерживаются разработчиками или комьюнити. В них, как правило, есть критичные баги, которые влияют на работу приложения.

Для работы с параметрами командной строки установим пакет yargs:

```
npm install yargs
```

На момент написания материала у этого пакета 54 миллиона установок за неделю (подробнее — здесь), а последний релиз появился в декабре 2020 года. На гитхабе модуля создано 255 Issue, в которых активно идут обсуждения. Подробнее о issues на гитхабе — здесь. Такие показатели дают понять, что модуль развивается и поддерживается. Это несколько снижает риски его использования.

По сравнению с process.argv модуль yargs позволяет гораздо более удобно и гибко настраивать работу с параметрами командной строки.

Уберём путь к читаемому файлу из кода (хардкод) и передадим его в программу при запуске.

Подключаем и настраиваем модуль yargs:

```
const yargs = require("yargs");

const options = yargs
      .usage("Usage: -p <path>")
      .option("p", { alias: "path", describe: "Path to file", type: "string",
demandOption: true })
      .argv;
```

Тезисно рассмотрим методы модуля yargs, которые описывают, как наша программа будет взаимодействовать с переменной командной строки.

1. Метод usage описывает сообщение, которое будет показываться пользователю в случае запуска программы без указания требуемой переменной. Это сообщение позволяет новому пользователю понять, какую переменную ожидает приложение, и как именно её в приложение передать.
2. Метод option описывает переменную, которую будет ожидать приложение. В качестве параметров этот метод принимает строку, например, название переменной, которое потребуется указать при запуске (p) и объект. Объект содержит в себе следующие ключи:
    a. alias — сюда записывается название ключа, по которому внутри программы будет доступна переданная переменная. Эта переменная доступна и по тому названию, что указывается при запуске. Однако alias служит для демонстрации более очевидного использования переменной в коде.

       Можно записать полное название вместо p. Но даже переменная с коротким названием path вполне допустима.

       Однако, если у нас много переменных, или их названия слишком длинные, в команде запуска практически невозможно будет разобрать, где указывается название переменной, а где — значение. Поэтому общепринятой практикой считается применение однобуквенных названий при запуске и алиасов при использовании в коде. Кстати, с английского alias переводится как «псевдоним». Это устойчивый термин, который широко распространён в программировании.

    b. describe — это значение устанавливает описание переменной при вызове справки.
    c. type — тип переменной, которую ожидает приложение.
    d. demandOption — это флаг, который определяет, обязательная ли это переменная или нет.

3. argv — по этому ключу переменные командной строки будут доступны в форме объекта с ключами:
    a. _ — по этому ключу доступны все переменные, которые переданы без соответствующих флагов.
    b. $0 — название запущенного скрипта (файла).
    c. p, path и прочие ключи доступны только в нашем примере, так как мы использовали p в качестве названия переменной при запуске приложения и path, как алиас для этой переменной. Если добавить другие переменные или переименовать эти — они также будут доступны в этом объекте под соответствующими ключами.

Для наглядности ниже есть сообщение, которое появится при запуске программы без передачи в неё пути файла:

```
Usage: -p <path>

Options:
      --help     Show help                                             [boolean]
      --version  Show version number                                   [boolean]
  -p, --path     Path to file                                 [string] [required]

Missing required argument: p
```

Пример объекта options, который получился в итоге:

```
{ _: [], p: 'access.log', path: 'access.log', '$0': 'index.js' }
```

Для формирования пути к файлу воспользуемся стандартным модулем Node.js path. Через представленный модуль и его метод join сформируем путь к файлу. В этот метод мы передаём переменную __dirname и переменную, в которую попадёт указанный пользователем путь. Переменная __dirname содержит в себе абсолютный путь к каталогу, где размещается текущий модуль. Соответственно, зная примерное расположение файла относительно текущего модуля, мы составим корректный путь до него.

```
const path = require("path");

const filePath = path.join(__dirname, options.path);
```

Общий код программы, которая принимает на вход путь к читаемому файлу, выглядит следующим образом:

```
const fs = require("fs");
```

```
const yargs = require("yargs");
const path = require("path");

const options = yargs
      .usage("Usage: -p <path>")
      .option("p", { alias: "path", describe: "Path to file", type: "string",
demandOption: true })
      .argv;

const filePath = path.join(__dirname, options.path);

fs.readFile(filePath,'utf8', (err, data) => {
      console.log(data);
});
```

Настало время запустить программу. Чтобы через команду npm run передать аргументы внутрь команды, описанной в разделе scripts файла package.json, предусматривается следующий синтаксис:

```
npm run <command> [-- <args>]
```

Применимо к нашему приложению команда запуска приобретает следующий вид:

```
npm start -- -p access.log
```

Здесь мы читаем файл с логами запросов к серверу, поэтому в терминал выводится его содержимое:

```
89.123.1.41 - - [30/Jan/2021:11:11:20 -0300] "POST /foo HTTP/1.1" 200 0 "-"
"curl/7.47.0
```

Если переместить файл для чтения из каталога программы на уровень выше, то эти изменения потребуется отразить и в команде запуска, так как путь к файлу изменится:

```
npm start -- -p ../access.log
```

Ввод данных

Теперь наша программа принимает путь к файлу, который будет читать. Сделаем так, чтобы вводить его в программу уже после запуска. Это сделает команду запуска более простой, сохранив её функциональность. Пользователю не надо будет заранее изучать, какие переменные и в каком

формате надо передавать в программу при запуске. Программа сама запросит у него данные — юзеру останется только ввести их в терминал.

Для реализации такой функциональности воспользуемся ещё одним стандартным модулем — readline. Согласно официальной документации, этот модуль предоставляет интерфейс для считывания данных из потока на чтение построчно.

Перед использованием такого модуля требуется создать экземпляр класса readline.Interface, который и будет предоставлять весь инструментарий. Для этого в модуле readline предусмотрен метод createInterface. Он принимает на вход объект в качестве аргумента. Полное описание доступных ключей этого объекта — в официальной документации. Для нашей же программы требуются только два:

- input — поток на чтение, который будет прослушиваться модулем;
- output — поток на вывод, куда отправляются данные readline.

Создадим интерфейс readline в нашем приложении:

```
const readline = require("readline");

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});
```

В качестве ввода input воспользуемся системным потоком process.stdin, куда поступают данные, введённые клавиатурой.

Чтобы модуль показывал в терминале сообщение с просьбой ввести данные, применяется output. Поэтому в качестве output мы используем системный поток вывода process.stdout.

Теперь нам надо создать саму просьбу ввести путь к файлу и обработать этот ввод. Для этого у созданного объекта rl есть метод question.

```
rl.question("Please enter the path to the file: ", function(inputedPath) {
    console.log(inputedPath);

    rl.close();
});
```

Как мы видим, метод question принимает первым аргументом строку с сообщением пользователю, а вторым — коллбэк, который будет обрабатывать введённые пользователем данные.

Если в коллбэк метода question вложить ещё один вызов метода question, то можно получить цепочку вопросов, каждый из которых станет появляться после ответа на предыдущий.

После получения данных поток требуется закрыть — для этого вызывается метод close у объекта интерфейса rl. Чтобы влиять на работу приложения по событию close, создадим обработчик этого события, где будем завершать работу приложения.

```
rl.on("close", function() {
    process.exit(0);
});
```

Вызов метода exit глобального объекта process с 0 в качестве аргумента завершает работу приложения в нормальном режиме. Если передать 1 вместо 0 — работа завершится с ошибкой.

Теперь у нас всё готово, остаётся только собрать код.

```
const fs = require("fs");
const path = require("path");
const readline = require("readline");

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.question("Please enter the path to the file: ", function(inputedPath) {
    const filePath = path.join(__dirname, inputedPath);

    fs.readFile(filePath,'utf8', (err, data) => {
        console.log(data);

        rl.close();
    });
});

rl.on("close", function() {
    process.exit(0);
});
```

При запуске программы больше не требуется указывать путь и использовать просто npm run start. После запуска в терминале появится вопрос:

```
Please enter the path to the file:
```

После ввода мы увидим данные, считанные из файла:

```
Please enter the path to the file: access.log
89.123.1.41 - - [30/Jan/2021:11:11:20 -0300] "POST /foo HTTP/1.1" 200 0 "-"
"curl/7.47.0
```

Элементы графического интерфейса

Теперь добавим в наше приложение элементы графического интерфейса. Для этого воспользуемся сторонним модулем Inquirer. С этим модулем выведем список файлов в директории приложения и выберем, какой из них надо прочитать программе. Собственно модуль Inquirer и предназначен для создания инструментов командной строки, например, вопросы пользователю, проверка вводимых пользователем данных и их обработка.

Первым шагом требуется получить список файлов из текущей категории. Для этого воспользуемся методами readdirSync и lstatSync стандартного модуля, чтобы работать с файловой системой fs.

- readdirSync синхронно читает контент в директории и возвращает массив имён файлов и директорий, которые там содержатся;
- lstatSync возвращает объект класса fs.Stats, который содержит в себе информацию о файле: размер файла, дату создания, изменения и т. д. Этот объект также имеет несколько встроенных методов. Нас в этом контексте интересует метод isFile, который позволяет отличать файлы и директории друг от друга.

```
const isFile = fileName => {
    return fs.lstatSync(fileName).isFile();
}

const list = fs.readdirSync(__dirname).filter(isFile);
```

Перед нами константа isFile — этот метод возвращает true, если переданное во входящем аргументе имя соответствует файлу, и false — если директории. Он работает очень просто:

- в метод fs.lstatSync() передаём имя, которое хотим проверить на соответствие файлу или директории;
- у полученного объекта вызываем метод isFile, который и вернёт true или false.

Затем считываем данные о содержащихся в директории файлах и папках, фильтруем этот массив, используя метод isFile, и получившийся список записываем в константу list.

Далее превратим это в меню, где пользователь сможет выбрать файл. Для этого используем метод promt модуля inquirer, который выводит вопросы пользователю. Этот метод принимает на вход массив объектов. Объекты обычно содержат следующие ключи:

1. name — название переменной, по которой будет доступно введённое пользователем значение.
2. type — тип вопроса. Есть следующие типы вопросов:
    a. input — вопрос, ответ на который вводит пользователь;
    b. number — вопрос, в ответ на который надо ввести число;
    c. confirm — вопрос, который даёт пользователю выбрать «Да» или «Нет» и возвращать булево значение;
    d. list — вопрос, в котором пользователю предоставляется выбор одного ответа из представленного списка;
    e. checkbox — вопрос, где пользователь выбирает несколько вариантов ответов из списка в отличие от list;
    f. password — в таком типе вопроса все данные, введённые пользователем, будут скрыты;
3. message — сообщение, которое будет показано пользователю;
4. choices — массив вариантов ответов, которые будут показаны пользователю, если тип вопроса поддерживает варианты ответов, например, list или checkbox.

Метод promt — асинхронный, возвращает промис. Ответ на вопрос, введённый пользователем, доступен в методе then.

В результате мы получаем следующий код нашей программы:

```
const fs = require("fs");
const path = require("path");
const inquirer = require("inquirer");

const isFile = fileName => {
  return fs.lstatSync(fileName).isFile();
}

const list = fs.readdirSync(__dirname).filter(isFile);

inquirer
    .prompt([{
            name: "fileName",
            type: "list",
            message: "Choose file:",
            choices: list,
        }])
    .then((answer) => {
        console.log(answer.fileName);
        const filePath = path.join(__dirname, answer.fileName);

       fs.readFile(filePath,'utf8', (err, data) => {
            console.log(data);
        });
    });
```

В терминале появится выбор из файлов, находящихся в текущей директории:

```
> node index.js

? Choose file: (Use arrow keys)
❯ access.log
  index.js
  package-lock.json
  package.json
```

Исполняемый файл

Исполняемые файлы содержат в себе готовую к запуску программу. Цель — чтобы программа запускалась из любого места и была «самостоятельной». Сейчас же это просто JS-скрипт, запускаемый через Node.js. В разных операционных системах исполнительные файлы имеют разные расширения — bat, exe, com, bin. Исполняемый файл запускается как любая другая программа, для него необязательно знать о командах npm. Однако для запуска исполняемых файлов Node.js всё ещё требуется, чтобы Node.js был установлен на машине.

Перед тем как мы превратим нашу программу в исполняемый файл, переименуем JS-файл index.js в cli.js. Цель — понимать, что речь идёт об исполняемом файле. Это необязательное действие, но в рамках текущего урока это будет удобно.

Чтобы сделать наш JS-файл исполняемым, надо добавить последовательность Шебанга в начало файла:

```
#!/usr/bin/env node
```

Информацию о том, что такое последовательность Шебанга, можно найти по этой ссылке. Но если кратко, то эта последовательность состоит из двух символов — # и ! в начале файла скрипта. Когда файл с такой последовательностью запускается в UNIX-системах, остаток строки воспринимается как имя программы-интерпретатора. В нашем случае — это Node.js.

В Windows эта строка проигнорируется, то есть символ # будет воспринят как комментарий. Однако она всё равно должна быть при упаковывании программы через npm.

WIndows установит оболочку .cmd через npm рядом со скриптом, после чего его можно будет запускать из командной строки. Без последовательности Шебанга npm не сможет этого сделать.

Далее сообщим системе (*nix-системе), что этот файл запускается как программа. Для этого применяется следующая команда:

```
chmod +x cli.js
```

Теперь запускаем программу командой:

```
./cli.js
```

Однако чтобы запустить программу в Windows всё ещё требуется указывать программу-интерпретатор:

```
node.cmd cli.js
```

Имя программы

Сейчас запуск программы жёстко привязан к имени запускаемого файла. Это чаще всего неудобно и неочевидно, особенно в *nix-системах, где все программы запускаются из терминала, без привязки к местонахождению самой программы на диске.

Чтобы создать связь "имя программы": "имя скрипта", снова обратимся к файлу package.json.

Добавим в package.json секцию bin:

```
  "bin": {
    "reader": "index.js"
  },
```

Ключ reader — это имя программы, а значение index.js — имя исполняемого файла.

Теперь, чтобы запускать программу из любой директории в терминале, применим команду:

```
npm link
```

Это очень мощная команда, она выполняет множество разных функций. В нашем случае такая команда эмулирует установку нашего Node.js-пакета как глобального (npm install -g) и делает доступным её вызов по имени, указанному в качестве ключа в секции bin.

Теперь мы можем запускать нашу программу из любой директории в *nix-системах и в Windows:

```
reader
```

Чтобы убрать связь имени reader с нашей программой, запустим команду:

```
npm unlink
```

Глобальный запуск

Теперь наша программа будет запускаться из любой директории, нам необязательно знать, где она находится или какое имя у файла скрипта. Однако если мы действительно запустим её из директории отличной от той, в которой писали программу и тестировали, то получим ошибку:

```
internal/fs/utils.js:307
    throw err;
    ^

Error: ENOENT: no such file or directory, lstat 'access.log'
```

Так происходит из-за этого участка кода:

```
const isFile = fileName => {
    return fs.lstatSync(fileName).isFile();
}

const list = fs.readdirSync(__dirname).filter(isFile);
```

Здесь __dirname продолжает ссылаться на директорию, где располагается модуль. Следовательно, в метод-фильтр isFile попадают файлы этой директории. Однако при попытке считать свойства файла метод fs.lstatSync ищет их в текущей директории. Естественно, их там нет, и программа «падает» с ошибкой.

Исправить это очень просто. Достаточно вместо __dirname использовать метод process.cwd(), который ссылается на ту директорию, из которой Node.js-программа запустилась.

Итоговый код приобретает следующий вид:

```
#!/usr/bin/env node

const fs = require("fs");
const path = require("path");
const inquirer = require("inquirer");

const currentDirectory = process.cwd();

const isFile = fileName => {
  return fs.lstatSync(fileName).isFile();
```

```
}

const list = fs.readdirSync(currentDirectory).filter(isFile);

inquirer
  .prompt([
    {
      name: "fileName",
      type: "list",
      message: "Choose file:",
      choices: list,
    },
  ])
  .then((answer) => {
    const filePath = path.join(currentDirectory, answer.fileName);

      fs.readFile(filePath,'utf8', (err, data) => {
            console.log(data);
      });
  });
```

Теперь программа не привязана к директории, в которой расположена, и её действительно можно запускать из любого места.

Заключение

Мы написали консольное приложение, которое запускается из любой директории. Это приложение будет показывать список файлов в текущей директории и выводить в терминал содержимое выбранного пользователем файла. Программа имеет минималистичный, но удобный для взаимодействия графический интерфейс, и способна работать в различных операционных системах.

В процессе написания программы мы рассмотрели различные подходы и способы реализации консольных приложений и их взаимодействия с пользователями.

Практическое задание

Примените полученные знания к программе, которую вы написали на прошлом уроке.

Для этого превратите её в консольное приложение, по аналогии с разобранным примером, и добавьте следующие функции:

1. Возможность передавать путь к директории в программу. Это актуально, когда вы не хотите покидать текущую директорию, но надо просмотреть файл, находящийся в другом месте.
2. В директории переходить во вложенные каталоги.
3. Во время чтения файлов искать в них заданную строку или паттерн.

Глоссарий

1. **CLI-приложения** (консольные приложения) — приложения, взаимодействие с которыми осуществляется через интерфейс командной строки (терминал).
2. **Параметры командной строки** — это массив, в котором доступны все параметры, переданные в Node.js в момент запуска программы из командной строки.
3. **Переменные окружения** (environment variables) — это переменные, содержащие текстовую информацию, которую используют запускаемые программы.
4. **Последовательность Шебанга** — это последовательность из двух символов — # и ! в начале файла скрипта. В *nix-системах остаток строки после этой последовательности воспринимается как имя программы-интерпретатора.

Дополнительные материалы

1. Официальная документация.
2. Официальная документация модуля yargs.
3. Официальная документация модуля readline.
4. Официальная документация модуля Inquirer.
5. Статья «Последовательность Шебанга».

Используемые источники

1. Официальная документация.