JavaScript has evolved over the last five years with the introduction of Node.js (https://nodejs.org/en/). Basically, It serves as a language that can be used for Front end application, server side application, desktop application, mobile application, etc.

The ecosystem is growing really fast with different innovation and technologies springing out from it. StackOverflow's recent survey shows that Javascript is by

far the most commonly used programming language on earth.

That said, over the past 5 years, Node.js has allowed developers to write command-line tools that are interactive and very easy to use.

Table of Contents

If you have used git (https://git-scm.com/),

heroku (https://devcenter.heroku.com/articl cli) , gulp (http://gulpjs.com/), grunt (https://gruntjs.com/) or any other package that allows you to bootstrap your project like create-react-app (https://github.com/facebookincubat react-app) , angular-cli (https://cli.angular.io/) , yeoman (http://yeoman.io/) , etc, you've used the power of Node.js in writing command-line tools. I think it's easy and simple to start writing your own command-line tools today with Node.js and its libraries for increased productivity.

Project description

We will be building an interactive command-line contact management system that allows users to create, read, and manage their contacts from the shell(command prompt). This will allow users to perform basic CRUD (http://searchdatamanagement.techtcycle) operation from the shell with ease.

Here is a demo of the application we're going to build

We will use Node.js (https://nodejs.org/en/) as the core framework, commander.js (https://github.com/tj/commander.j for command-line interfaces, inquirer.js (https://github.com/SBoudrias/Inqu for gathering user input from the command-line, and MongoDB for data persistence.

Technologies

1) Node.js (https://nodejs.org/en/) - A server-side platform wrapped around the JavaScript language with the largest package(npm)

ecosystem in the world.

Essential Reading: Learn React from Scratch! (2020 Edition) (https://bit.ly/2TtV1sA)

2) Commander.js (https://github.com/tj/commander.j - An elegant and light-weight command-line library for Node.js.

3) Inquirer.js (https://github.com/SBoudrias/Inqu - A collection of powerful interactive command-line interfaces.

4) MongoDB (https://www.mongodb.com/) - A free and opensource NoSQL document-oriented database.

Here is a picture of what the application workflow looks like.

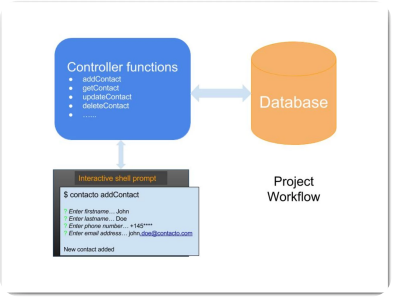

Fig: Project workflow

Steps to building an interactive command-

line application with Node.js

1) Project setup

2) Define application logic

3) Handle command-line arguments

4) Interactive run time user inputs

5) Convert application to a shell command

6) More application logic

Step 1 of 5: Project setup

Ensure the version of your installed Node.js is >=6. Run `node --version` on your terminal to verify. We will use yarn (https://code.facebook.com/posts/18 , a package manager for Javascript developed by Facebook to manage our project dependencies. If you don't have it installed, run

```
$ npm install -g yarn
```

You can also use npm (https://www.npmjs.com/) to run your application as `yarn` is just a personal preference though

with some advantages.

Let's create a project directory and initialize it as a Node.js application.

```bash
$ mkdir contac
$ cd contact-m
$ yarn add mon
```

Now that we have our Node.js application initialized, you will see `package.json` in your project's root directory with updated `dependencies` to get started.

```
BASH

{
  "name": "con
  "version": "
  "description
  ............
 "dependencies
    "commander
    "inquirer"
    "mongoose"
  }
}
```

Step 2 of 5: Define application logic

In this section, we will define our schema, model, and the controller functions that handles user input and persist them to the database.

This step requires that your MongoDB server is running.

Let's create **logic.js** in the project directory

```
JAVASCRIPT

const mongoose
const assert =
mongoose.Promi

// Connect to
// We assign t
const db = mon

// Converts va
function toLow
  return v.toL
}

// Define a co
const contactS
  firstname: {
  lastname: {
  phone: { typ
  email: { typ
```

```
});

// Define mode
const Contact

/**
 * @function
 * @returns {S
 */
const addConta
  Contact.crea
    assert.equ
    console.in
    db.disconn
  });
};

/**
 * @function
 * @returns {J
 */
const getConta
  // Define se
  const search
  Contact.find
  .exec((err,
    assert.equ
    console.in
    console.in
    db.disconn
  });
};
```

```
// Export all
module.exports
```

Basically, we define our schema and interact with the database via our model. We then define controller functions(**addContact**, **getContact**, etc) that would be called depending on the user's interaction with the application.

You will notice we're using `toLower` function to convert the values to lowercase before saving to the database. Also, like in any other contact management system

we want to allow users to make *case-insensitive inexact matches*. However, if you want to make *case-insensitive exact matches*, see below for options.

```
JAVASCRIPT

// case-insens
const search =
// case-insens
const search =
```

For the purpose of this demo, we're only searching by `firstname` and `lastname` fields.

Step 3 of 5: Handle

command-line arguments

To test our progress so far, we need a mechanism for accepting users' input and passing it to our controller functions defined in the step above.

Commander.js is our friend for reading command-line inputs and saves us the stress of managing different options and parsing values. It comes with Git-like sub-commands which makes it interactive and easy to use. It allows us to define `option`, `alias`, `command`, `version`, `action`, `description`, etc. We will see them in action in a bit.

Let's create **contact.js** in the project directory

```
JAVASCRIPT

const program =
// Require log
const { addCon

program
  .version('0.
  .description

program
  .command('ad
  .alias('a')
  .description
  .action((fir
    addContact
  });

program
  .command('ge
  .alias('r')
  .description
  .action(name

program.parse(
```

Whoop! We're all set up. Before we test the application let's understand the key idea here. commander.js' API exposes some functions which are chainable.

`.command()` allows you to specify the command name with optional parameters. In our own case, they're not optional because we specified them using `<>`. To make any parameter optional, use `[the parameter goes in here]` instead.

`.action()` takes a callback and runs it

each time the command name is specified.

Let's test it out in the terminal.

```
BASH

$ node contact
$ node contact
```

Now that we've gotten a hang of the program, let's try adding a contact to our database from the terminal

```
Usage: contact
[options] [command]
```

```
BASH

$ node contact
```

The parameters to `addContact` are space separated and since each of the parameters is marked as required using `<>`, you *must* provide all the parameters.

We can also use `alias` instead of the full command name. Now, let's retrieve the contact we just added

```bash
$ node contact
```

####** Step 4 of 5: Interactive runtime

user inputs** We can accept and parse command-line arguments, thanks to commander.js. However, we can enhance the user experience by allowing a user respond to questions in a more interactive way. Let's use inquirer.js (https://github.com/SBoudrias/Inqu for user interface and inquiry session flow.

First we define the questions to be presented to the user and based on the answer, we save the contact.

Let's update **contact.js**

JAVASCRIPT

```javascript
.........
const { prompt

// Craft quest
const question
  {
    type : 'inp
    name : 'fi
    message :
  },
  {
    type : 'inp
    name : 'la
    message :
  },
  {
    type : 'inp
    name : 'ph
    message :
  },
  {
    type : 'inp
    name : 'em
    message :
  }
];
.........

program
  .command('ad
```

```
        .alias('a')
        .description
        .action(() =
          prompt(que
            addConta
        });
      ..........
```

This looks a lot cleaner with some interactive flow. We can now do away with the parameters. `prompt` launches an inquiry session by presenting the `questions` to the user. It returns a promise, `answers` which is passed to our controller function, `addContact`.

Let's test our program again and respond to the prompt

```bash
$ node contact
# The Above co
? Enter firstn
? Enter lastna
? Enter phone
? Enter email
New contact ad
```

Step 5 of 6: Convert application to a shell command

We are able to accept command-line arguments in an interactive way but what about we make our tool a regular shell command? The good news is, it's easy to achieve.

First, we add `#!/usr/bin/env node` at the top of **contact.js**, which tells the shell how to execute our script, otherwise the script is started without the node executable.

```
JAVASCRIPT

#!/usr/bin/env

const program =
...............
```

Next, we configure the file to be executable. Let's update our package.json file with the necessary properties

```javascript
"name": "conta
...........
"preferGlobal"
  "bin": "./co
..........
```

First, we set `preferGlobal` to `true` because our application is primarily a command-line application that should be installed globally. And then we add `bin` property which is a map of command name to local file name. Since, we have just one executable the name should be the name of the package.

Finally, we create a symlink from **contact.js** script to `/usr/local/bin/contact` by running

```
BASH

$ yarn link # 
```

Let's test that our tool now works globally. Open a new terminal and ensure you're not in your project directory.

```
BASH

$ contacto --h
$ contacto r j
```

Yes, our tool is ready to be shipped to the world. What a user needs to do is simply install it on their system and everything would be fine.

Step 6 of 6: More application logic

Now, we are able to add and get contact(s). Let's add the functionality to *list*, *update*, and *delete* contacts in the application. We have a problem though. How do we get a specific contact to update/delete since we don't have unique contraints? The only unique key is the `_id`. For this demo, let's use the `_id` from the database. This

requires we first search for a contact and then select the `_id` of the contact we want to delete/update.

Let's update **logic.js.**

```
JAVASCRIPT

const mongoose
const assert =
mongoose.Promi

// Connect to
// We assign t
const db = mon

// Convert val
function toLow
  return v.toL
}

// Define a co
const contactS
  firstname: {
  lastname: {
  phone: { typ
```

```
  email: { typ
});

// Define mode
const Contact

/**
 * @function
 * @returns {S
 */
const addConta
  Contact.crea
    assert.equ
    console.in
    db.disconn
  });
};

/**
 * @function
 * @returns {J
 */
const getConta
  // Define se
  const search

  Contact.find
  .exec((err,
    assert.equ
    console.in
    console.in
    db.disconn
  });
```

```
};

/**
 * @function
 * @returns {S
 */
const updateCo
  Contact.upda
  .exec((err,
    assert.equ
    console.in
    db.disconn
  });
};

/**
 * @function
 * @returns {S
 */
const deleteCo
  Contact.remo
  .exec((err,
    assert.equ
    console.in
    db.disconn
  })
}

/**
 * @function
 * @returns [c
 */
const getConta
```

```
    Contact.find(
    .exec((err, c
      assert.equa
      console.in
      console.in
      db.disconn
    })
  }

  // Export all f
  module.exports
    addContact,
    getContact,
    getContactLi
    updateContac
    deleteContac
  };
```

Also, update **contact.js.**

```
JAVASCRIPT

#!/usr/bin/env

const program =
const { prompt
```

```
const {
  addContact,
  getContact,
  getContactLi
  updateContac
  deleteContac
} = require('.,

const question
  {
    type : 'in
    name : 'fi
    message :
  },
  {
    type : 'in
    name : 'la
    message :
  },
  {
    type : 'in
    name : 'ph
    message :
  },
  {
    type : 'in
    name : 'em
    message :
  }
];

program
```

```
  .version('0.
  .description

program
  .command('ad
  .alias('a')
  .description
  .action(() =
    prompt(que
      addConta
  });

program
  .command('ge
  .alias('r')
  .description
  .action(name

program
  .command('up
  .alias('u')
  .description
  .action(_id
    prompt(que
      updateCo
  });

program
  .command('de
  .alias('d')
  .description
  .action(_id
```

```
program
  .command('ge
  .alias('l')
  .description
  .action(() =

// Assert that
if (!process.a
  program.outp
  process.exit
}
program.parse(
```

It's time to test things out in the terminal

```
BASH

$ contacto l #
$ contacto u <
$ contacto d <
```

If you fail to provide any of the values, it's

set to `'""`.

Good news! You have built a command-line tool with Node.js.

Wrap up

Now, we know it's easy to build command-line tools with Node.js. The libraries used in this tutorial are not compulsory for building command-line application. We only used them to avoid being distracted with some tiny details. You can use Child process (https://nodejs.org/api/child_proces module from Node.js to achieve the same result natively.

Yes, think of the tool we built and its use; think about its flaws and improve it.

Like this article? Follow @rowlandekemezie on Twitter (https://twitter.com/rowlandekemez

Read next...

Conversation (19)

Sort by **Best**

Have a Disqus Account?

Log In

ochsec · 22 Jan, 2018

Great tutorial! I just ran into one issue where I had to call mongoose.

disconnect () in the callbacks for the CRUD functions instead of db.disconnect(), which threw an undefined error.

Reply · Share · *7 Likes* ·

testingxp s
· 16 Aug, 2017

Very nice article.

Thanx for sharing with us.

Reply · Share · *4 Likes* ·

RedFlask
· 28 May, 2018

Nice article! For those who's getting "command not found" error when running "contacto --help", I got the same error. Run "chmod u+x

contact.js" to make contact.js executable. Then, "contacto --help" should work.

Reply · Share · *3 Likes* ·

BlueSush · 24 Feb, 2018

Thanks, Rowland! It was interesting.

Reply · Share · *2 Likes* ·

GoldPaw · 16 Nov, 2017

Exactly the sort of quick tutorial I was looking for, thanks!

Reply · Share · *2 Likes* ·

BlueTeep · 12 Apr, 2018

yarn link doesn't work. keep getting the error "command not found" after entering "contacto --help"

the following link is for those that get a mongoose disconnect is not a function: https://stackoverflow.com/questions/48481040/mongoose-disconnect-is-not-a-function (https://stackoverflow.com/questions/48481040/mongoose-disconnect-is-not-a-function)

Mongoose "disconnect is not a function" (https://stackoverflow.com/questions/48481040/mongoose-disconnect-is-not-a-function)

I'm using this tutorial to make a node/mongo application. When I run addContact, it seems like the contact...

STACKOVERFLOW.COM (HTTPS://STACKOVERFLOW.COM/QUESTIONS/48481040/MONGOOSE-DISCONNECT-IS-NOT-A-FUNCTION)

Reply · Share
· *1 Like*
·

nomanm ood
· 3 Oct, 2017

How Can I keep the program running ? this program execute one command and exit. How can We keep this running ?
Like after entertaining 1 command it

should accept other commands

Reply · Share · 1 Like ·

Er Newton · 28 Feb, 2018

nomanmaqsood

that's not a recommended approach for keeping cli type programs simple.

if you wanna make it do more things, use bash or cmd, and kee looping on user input of what command to rur while running the cli using the user's desired command.

Reply · Share ·

Vika Cat · 15 Jul

Nice tut, thanks.)

Reply · Share ·

GoldPuzz · 27 Feb

Thanks, it helps me a

lot

Reply · Share

BlueTree

· 14 Feb

Your tutorial was super informative and helpful. Thank you!

Reply · Share

SHOW MORE COMMENTS...

www.akajlm.net