Table of Contents

Node.js Cheatsheet

It seems like the future of the web highly depends on Node.js. Node.js is an open-source, cross-platform runtime environment for developing server-side Web applications.

Node.js is great but if you are a developer comming from platforms like ruby, you will spend days or even several months just searching for the right way on how to write Node.js applications, before you will be able to start your first serious and long-term sustainable Node.js project.

This book intends to help developers write Node.js applications easy and fast. The book explains best practices and general development flows.

Installing Node.js

Before we can start building our first Node.js application, we need to install Node.js on our system.

The most common and elegant way is to use the Node Version Manager which is a simple bash script to manage multiple active Node.js versions.

Go to Node Version Manager repository and follow the installation steps there to complete this first step.

Project Initialization

Let's start by creating a new project directory.

```
$ mkdir myapp
$ cd myapp
```

Run the command below to create the main configuration file called `package.json`.

```
$ npm init --yes
```

Install the `nodemon` package globally. We will use it in development.

```
$ npm install -g nodemon
```

Create also the configuration file `nodemon.json`.

```
{
  "verbose": true
}
```

Create a simple application within the `src` directory.

```
$ mkdir src
$ echo 'console.log(`Hello World!`);' >> ./src/index.js
```

Open the `package.json` file and set the variables as shown below.

```
{
  "main": "./src/index.js",
  "scripts": {
    "start": "node ./src/index.js"
  }
}
```

A common convention is also to create a `README.md` file with the description of your application.

```
$ touch README.md
```

Every project should use some kind of code revision control. Git is the most popular, free and open source distributed version control system so we will use that. Let's initialize the repository.

```
$ git init .
```

Create a new file called `.gitignore` with a list of files and folders which should not be commited into Git repository.

```
$ echo .DS_Store >> .gitignore; echo node_modules >> .gitignore
```

We can now do the initial commit.

```
$ git add .
$ git commit -m 'project initialized'
```

Run `npm start` to run the application. Run `nodemon --exec npm run start` to start the application in development.

Future Javascript Today

The web is in a major transition phase. It's not clear any more how we should write web applications. We can read about new HTML and Javascript/ES6+ features every day, but you get frustrated as soon as you realize that you can not use these features today.

New scripting languages, transpilers and compilers like TypeScript, Traceur and Babel.js emerged in the last couple of years. By using these tools we can write the latest Javascript syntax today.

Which of these new technologies should we use? The answer is still blurry but we need to pick the lesser evil. Maybe after a year or two we will be using TypeScript but right now only Bable.js is something that we can safely integrate into our Node.js applications.

Installing Babel.js

Install the required packages.

```
npm i --save-dev babel-cli babel-preset-node5
```

Create a new file called `.babelrc` and add the configuration below.

```
{
  "presets": ["node5"]
}
```

Set variables inside the `packages.json` file as follows.

```
{
  "main": "./dist/index.js",
  "scripts": {
    "clean": "rm -Rf ./dist",
    "prebuild": "npm run clean; mkdir -p ./dist",
    "build": "babel ./src --out-dir ./dist --copy-files",
    "prestart": "npm run build",
    "start": "node ./dist"
  },
}
```

Now when we run the `npm start` command, the `app` directory will first be transpiled into `ES5` Javascript and then saved into `dist` directory. Note that the `dist` directory should be committed with the rest of the code.

Since we use `nodemon --exec` command, we need to exclude the dist directory from watch list in the `nodemon.json` file.

```
{
  ...
  "ignore": ["dist/*"]
}
```

We can now use all the latest Javascript features in our code.

Automated Testing

A test driven development cycle simplifies the thought process of writing code, makes it easier, and quicker in the long run.

Ava

Before we start, let's open the `./src/index.js` file and write some code that can be tested.

```
exports.echo = function () {
  return 'Hello World';
}
```

Install the required packages.

```
$ npm i --save-dev ava
```

Open the `./package.json` and add some configuration as shown below.

```
{
  "ava": {
    "files": [
      "./tests/*.js",
      "./tests/**/*.js"
    ],
    "concurrency": 5,
    "failFast": true
  },
  "scripts": {
    "test": "ava"
  }
}
```

Create a new file `./tests/index.js` and write a simple test.

```
const test = require('ava');
const {echo} = require('../src');

test('returns Hello World', async (t) => {
  t.is(echo(), 'Hello World');
});
```

Execute the `npm test` command to run the test.

Simple API Server

This chapter explains how to create a simple Node.js API server using the Express.js framework.

First, install the dependencies.

```
$ npm install --save express@5.0.0-alpha.2
```

Router

Create a new file `./src/router/index.js` and define a router.

```
const express = require('express');
const usersRoutes = require('./users');

exports.createRouter = function () {
  const router = express.Router({
    mergeParams: true
  });

  router.get('/users', usersRoutes.index);
  router.get('/users/:id', usersRoutes.show);

  return router;
};
```

Routes

Create a new routes file `./src/router/users.js` which will expose user-related actions.

```
exports.index = function (req, res) {
  res.send('list of users');
};

exports.show = function (req, res) {
  let id = req.params.id;
  res.send(`user with ID${id}`);
};
```

Server

Open the `./src/index.js` file and paste the content below.

```
const express = require('express');
const {createRouter} = require('./router');

exports.Server = class {

  constructor(config) {
    this._config = config;
    this._app = null;
    this._server = null;
  }

  listen() {
    return new Promise((resolve) => {
      if (this._server) return this;

      this._app = express();
      this._app.use('/', createRouter());

      let {port, host} = this._config;
      this._server = this._app.listen(port, host, resolve);
    });
  }

  close() {
    return new Promise((resolve) => {
      if (!this._server) return this;

      this._server.close(resolve);

      this._server = null;
      this._app = null;
    });
  }

}
```

Configuration

Open the `./package.json` file and set configuration variables and a start script.

```
{
  "config": {
    "server": {
      "port": 4444,
      "host": "127.0.0.1"
    }
  },
  "scripts": {
    "start": "node ./src/scripts/start.js"
  }
}
```

Create a new configuration file `./src/config.js` and set application variables.

```
exports.serverConfig = {
  port: process.env.npm_package_config_server_port,
  host: process.env.npm_package_config_server_host
};
```

Create a startup script `./src/scripts/start.js` which start the application.

```
const {Server} = require('..');
const {serverConfig} = require('../config');

const app = new Server(serverConfig);
app.listen().then(() => {
  console.log('Server started');
}).catch((error) => {
  app.close();
  console.log('Error', error);
});
```

Run the `npm start` command to start the server then navigate to `http://localhost:4444/users` to see the result. Use npm config command to configure the server.

Advanced Application Structure

A real world Node.js application will most likely consist of multiple pieces. An API application for example will include at least a HTTP server and a database layer. These two components are equally important - two features - and thus should be treated as two unrelated standalone sub applications.

This chapter explains how to properly structure more complex Node.js application which consists of multiple components.

In the previous chapter `Simple API Server`, we built a simple HTTP server. Let's continue where we left off.

The Plan

We need to split the application features into standalone components and then expose them through application's public interface.

Components are unaware of a global application context, application configuration or application state. A component must be reusable and can be used multiple times inside or outside the project. This means that we need to instantiate and configure it before any interaction take place.

The application will have a single entry point (as before) which will automatically instantiate and configure application components when the application instance is created.

Refactoring

Moving the HTTP server files into a new sub folders called `server`. The application structure should look similar the the one below.

```
src
└─ server
   └─ routes
      └─ users.js
   └─ index.js
   └─ router.js
└─ scripts
   └─ start.js
└─ config.js
└─ index.js
.gitignore
package.json
README.md
```

Create a new file `./src/index.js` and expose the server feature.

```
exports.Server = require('./server').Server;
```

MongoDB With Mongoose

NEEDS UPDATE!

Mongoose is an elegant MongoDB object modeling library for Node.js. It provides a straight-forward, schema-based solution for modeling Node.js application data.

Let's start by installing the required packages.

```
$ npm install --save mongoose
```

Open the `app/config.js` file and add configuration variables as shown below.

```
export default {
  mongoose: {
    uri: process.env.MONGOOSE_URI || 'mongodb://localhost:27017/
apidb',
    options: {
      autoIndex: false,
      server: {poolSize: 10, socketOptions: {keepAlive: 1, conne
ctTimeoutMS: 30000}},
      replset: {poolSize: 10, socketOptions: {keepAlive: 1, conn
ectTimeoutMS: 30000}}
    }
  },
};
```

Create a new file `app/mongoose/schemas/user.js` and define schema for a `User` model.

```
import {Schema} from 'mongoose';

export const schema = new Schema({
  name: {
    type: String
  },
  email: {
    type: String
  }
}, {
  timestamps: true
});
```

Create a new file `app/mongoose/index.js` with the code below.

```
import mongoose from 'mongoose';
import {schema as userSchema} from './schemas/user';

export function createConnection({uri, options}) {
  let conn = mongoose.createConnection(uri, options);
  conn.model('User', userSchema);
  conn.on('error', console.log);
  return conn;
};
```

We can use our new component like this:

```
import config from '../config';
import {createConnection} from '../mongoose';

(async function() {
  let db = createConnection(config.mongoose);
  let User = db.model('User');
  let users = await User.find();
  await db.close();
})().catch(console.error);
```

Using Elasticsearch

NEEDS UPDATE!

Elasticsearch is a distributed, open source search and analytics engine, designed for horizontal scalability, reliability, and easy management.

Project Setup

Let's start the implementation by installing the required modules.

```
$ npm install --save elasticsearch
```

Open the app/config.js file and add configuration variables as shown below.

```
export default {
  elasticsearch: {
    host: process.env.ELASTICSEARCH_HOST || 'localhost:9200',
    log: process.env.ELASTICSEARCH_LOG || 'trace'
  }
};
```

Create a new file `app/elasticsearch/index.js` and add the code below.

```
import elasticsearch from 'elasticsearch';

export function createClient({host, log}) {
  return new elasticsearch.Client({host, log});
}
```

We can use our new component like this:

```
import config from '../config';
import {createClient} from '../elasticsearch';

(async function() {
  let es = createClient(config.elasticsearch);
  let res = await es.search({index: 'users', q: 'title:test'});
  await es.close();
})();
```

OS X Setup

Use Homebrew with `brew gapple/services` for installing and managing services. Then you can execute the commands below.

```
$ brew install java7
$ brew install elasticsearch
$ brew install kibana
$ kibana plugin --install elastic/sense
$ brew services restart kibana
```

Ubuntu 15.10 Setup

Java JDK

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
$ sudo apt-get install oracle-java8-set-default
```

Elasticsearch.

```
$ cd /tmp
$ wget https://download.elasticsearch.org/elasticsearch/release/
org/elasticsearch/distribution/deb/elasticsearch/2.2.0/elasticse
arch-2.2.0.deb
$ tar -xvf elasticsearch-2.2.0.tar.gz
$ sudo dpkg -i elasticsearch-2.2.0.deb
$ sudo /bin/systemctl daemon-reload
$ sudo /bin/systemctl enable elasticsearch.service
$ sudo /bin/systemctl start elasticsearch.service
```

Kibana

```
$ cd /tmp
$ wget https://download.elastic.co/kibana/kibana/kibana-4.4.2-li
nux-x64.tar.gz
$ tar -zxvf kibana-4.4.2-linux-x64.tar.gz
$ mv kibana-4.4.2-linux-x64 /etc/kibana4
$ sed -i 's/#pid_file/pid_file/g' /etc/kibana4/config/kibana.yml
```

Create a new file `/etc/systemd/system/kibana4.service` and copy the code below.

```
[Unit]
Description=Kibana 4 Web Interface
After=elasticsearch.service

[Service]
ExecStartPre=/bin/rm -rf /var/run/kibana.pid
ExecStart=/etc/kibana4/bin/kibana
ExecReload=/bin/kill -9 $(cat /var/run/kibana.pid) && /bin/rm -r
f /var/run/kibana.pid && /etc/kibana4/bin/kibana
ExecStop=/bin/kill -9 $(cat /var/run/kibana.pid)

[Install]
WantedBy=multi-user.target
```

Install Kibana plugins.

```
$ /etc/kibana4/bin/kibana plugin --install elastic/sense
```

Start and enable kibana to start automatically at system startup.

```
$ systemctl start kibana4.service
$ systemctl enable kibana4.service
```

Kibana will be available on port `5601` .

Deployment

NEEDS UPDATE!

Sooner or later we will need to move our Node.js application on a production server.

PM2 is a production process manager for Node.js applications with a built-in load balancer. It allows you to keep applications alive forever, to reload them without downtime and to facilitate common system admin tasks.

Remote Server

Let's start by describing how to use PM2 to setup a Node.js production environment on an Ubuntu 15.10 VPS.

Installing Git

Install the Git package which is required later when deploying our app to the server.

```
$ sudo aptitude install git
```

Installing Node.js

Create a new user called `worker` which will be used for running and deploying Node.js applications.

```
$ sudo useradd -s /bin/bash -m -d /home/worker -c "web worker" w
orker
```

Set a password for it or install a SSH key so you will be able to log in over SSH.

Log to the remote server **using the new user you just created**. Go to Node Version Manager repository and follow the installation steps there.

Log out and log in again **with the new user**, then install the latest node.

```
$ nvm install v5.8.0
$ nvm alias default v5.8.0
```

Installing PM2

SSH to the remote server **with the new user** and install the PM2 package.

```
$ npm install -g pm2
$ pm2 install pm2-logrotate
```

Now run the `pm2 startup`. This will print out the command which we have to run as one of the `sudo` users.

PM2 is not using an `interactive` mode. We must enable that to be the default behavior. Open `~/.bashrc` and comment out the lines starting with a comment `If not running interactively, don't do anything`.

Reboot the server.

Installing Nginx

Node.js applications do not have permissions to run on ports like `80`. It's a common practice to use Nginx as a public gateway.

```
$ sudo aptitude update
$ sudo aptitude install nginx
```

Open the default server configuration file `/etc/nginx/sites-available/default` and configure the server to forward all traffic on port `80` to the private address `http://127.0.0.1:4444` at which our Node.js application will be available.

```
server {
  listen 80;

  location / {
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_set_header X-NginX-Proxy true;
    proxy_pass http://127.0.0.1:4444;
    proxy_redirect off;
    proxy_buffering off;
  }
}
```

Run `sudo service nginx restart` to apply the changes.

Setting Environment Variables

Application secrets and configuration data should be served to the application through environment variables.

Our Node.js application, which we will build later, will be using only `SERVER_PORT` and `SERVER_HOST` environment variables.

SSH to the server as user `worker`, open `~/.bashrc` file and define the variables that we need at the top of the file.

```
export SERVER_HOST="127.0.0.1"
export SERVER_PORT=4444
```

Git Repository Access

At this point we have to make sure, that the new user which we created earlier, has access to the the remote Git repository where our Node.js application is saved. This means that the new user can `clone` the repository.

We should try to clone the repository to the `/tmp` directory before we deploy our app.

Development Machine

Let's assume that we've already created a working HTTP server application. Configure it for deployment.

Install the dependencies.

```
$ npm install --save pm2@latest
```

PM2 uses a special file called `ecosystem.json` as its configuration file. Create that file and add configuration based on the example below (`node_args` is optional).

```
{
  "apps": [{
    "name": "API",
    "script": "build/scripts/start.js",
    "node_args": "--optimize_for_size --max-old-space-size=460 -
-gc_interval=100"
  }],
  "deploy": {
    "production" : {
      "user": "worker",
      "host": "testing2",
      "ref": "origin/master",
      "repo": "git@bitbucket.org:me/api.git",
      "path": "/home/worker/api",
      "post-deploy": "npm install; npm run build; pm2 startOrRes
tart ecosystem.json"
    }
  }
}
```

Open the `package.json` file and set the deploy script.

```
{
  "scripts": {
    "deploy": "pm2 deploy",
  },
}
```

Setup the application directory on the server.

```
$ npm run deploy production setup
```

Deploy, start the application and save state.

```
$ npm run deploy production
$ npm run deploy production exec "pm2 save"
```

GraphQL HTTP Server

NEEDS UPDATE!

GraphQL is a language used to query application servers for data. In this chapter we will create a simple API server using GraphQL.

Before we dig into it, let's create a new directory for our app, initialize it and make sure that Bable.js is configured.

Queries

Install the required packages.

```
$ npm install --save graphql
```

Create a new file `./src/schema/index.js` and define graph entry point.

```
import {GraphQLSchema} from 'graphql';
import QueryType from './types/QueryType';

export const schema = new GraphQLSchema({
  query: QueryType
});
```

Create a new file `./src/schema/types/UserType.js` and describe a user structure.

```
import {GraphQLObjectType, GraphQLID, GraphQLString} from 'graph
ql';

export default new GraphQLObjectType({
  name: 'User',
  fields: {
    id: {
      type: GraphQLID,
      resolve: (doc) => doc.id
    },
    name: {
      type: GraphQLString,
      resolve: (doc) => doc.name
    },
    email: {
      type: GraphQLString,
      resolve: (doc) => doc.email
    },
  }
});
```

Create another file `./src/schema/types/QueryType.js` and define the root query type.

```
import {GraphQLObjectType} from 'graphql';
import users from '../queries/users';

export default new GraphQLObjectType({
  name: 'Query',
  fields: {users}
});
```

Finally create a file `app/schema/queries/users.js` and define a resolve method for the root query field `users` which returns a list of users.

```
import {GraphQLInt, GraphQLList} from 'graphql';
import UserType from '../types/UserType';

global.users = [
  {id: 1, name: 'John', email: 'john@domain.com'},
  {id: 2, name: 'Nina', email: 'nina@domain.com'}
]

export default {
  type: new GraphQLList(UserType),
  args: {},
  resolve: async (root, args, info) => {
    return global.users;
  }
};
```

Create a script file `./src/scripts/schema.js` which will serve for executing a GraphQL query.

```
import {graphql} from 'graphql';
import {schema} from '../schema';

(async function() {
  let query = process.argv[2];
  let res = await graphql(schema, query);
  let json = JSON.stringify(res, null, 2);
  console.log(json);
})().catch(console.error);
```

Open the `package.json` and set a script.

```
{
  "scripts": {
    "schema:query": "nodemon --exec babel-node app/scripts/schem
a.js",
  },
}
```

We can now execute a query from the command line.

```
$ npm run schema:query '{users {id name email}}'
```

HTTP Server

Let's add an HTTP server as our public access point to our graph. The easiest way to create a GraphQL HTTP server is to use the express-graphql middleware.

```
$ npm install --save express express-graphql
```

Create a new file `./src/http/index.js` and write a simple HTTP server.

```
import express from 'express';
import graphqlHTTP from 'express-graphql';
import {schema} from '../schema';

export class HttpServer {

  constructor(config) {
    this.config = config;

    this.app = express();
    this.app.use('/graphql', graphqlHTTP({schema, graphiql: true
}));
  }

  get server() {
    return this.app.server;
  }

  start() {
    return new Promise(resolve => {
      let {host, port} = this.config.http;
      this.app.server = this.app.listen(port, host, () => resolv
e(this));
    });
  }

  stop() {
    return new Promise(resolve => {
      this.app.server.close(() => resolve(this));
      delete this.app.server;
    });
  }
}
```

Create a script file `./src/scripts/http.js` with the code which starts the server.

```
import config from '../config';
import {HttpServer} from '../http';

(async function() {
  const http = new HttpServer(config);
  await http.start();
  console.log(`HTTP server started ...`);
})().catch(console.error);
```

Create also the configuration file `app/config.js` and set HTTP server properties.

```
export default {
  http: {
    host: process.env.HTTP_HOST || '0.0.0.0',
    port: process.env.HTTP_PORT || 4444
  }
};
```

Open the `package.json` and update scripts.

```
{
  "scripts": {
    "http:start": "nodemon --exec babel-node app/scripts/http.js"
,
  },
}
```

Start the server by executing the command `npm run http:start`. Navigate to `http://localhost:4444/graphql?query={users{name}}` to see the result.

Mutations

Mutations are used when editing or deleting data on the server. Let's create a simple mutation which adds a new user to the `global.users` variable.

First, open the `app/schema/index.js` and define the `mutation` type.

```
import MutationType from './types/MutationType';

export const schema = new GraphQLSchema({
  ...
  mutation: MutationType
});
```

Create a new file `app/schema/types/MutationType.js` and add the code below.

```
import {GraphQLObjectType} from 'graphql';
import createUser from '../mutations/createUser';

export default new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    createUser
  }
});
```

User data will be sent to the GraphQL through an input argument. We need to define a `GraphQLInputObjectType` as we previously did for the `GraphQLObjectType` output object. Create a new file `app/schema/UserInputType.js` and define the user input type and set the `name` field as a required field.

```
import {GraphQLInputObjectType, GraphQLID, GraphQLString, GraphQ
LNonNull} from 'graphql';

export default new GraphQLInputObjectType({
  name: 'UserInput',
  fields: {
    name: {
      type: new GraphQLNonNull(GraphQLString)
    },
    email: {
      type: GraphQLString
    }
  }
});
```

Create a new file `app/schema/mutations/createUser.js` and define fields for the mutation which creates a new user.

```
import {GraphQLNonNull} from 'graphql';
import UserType from '../types/UserType';
import UserInputType from '../types/UserInputType';

export default {
  type: UserType,
  args: {
    input: {
      type: new GraphQLNonNull(UserInputType)
    }
  },
  resolve: async (root, args, info) => {
    let {input} = args;
    global.users.push(input);
    return input;
  }
};
```

Navigate to `http://localhost:4444/graphql` and execute the mutation.

```
mutation {
  createUser(input: {
    name: "Bob"
    email: "bob@domain.com"
  }) {name email}
}
```

Custom Scalar Types

In the example above we don't check if the email field is formatted correctly. We need to set a custom type to do that.

Install the dependencies first.

```
$ npm install --save validator
```

Create a new file `app/schema/scalars/EmailAddressType.js` and describe what a valid content should look like.

```
import validator from 'validator';
import {GraphQLScalarType} from 'graphql';
import {Kind} from 'graphql/language';
import {GraphQLError} from 'graphql/error';

export default new GraphQLScalarType({
  name: 'EmailAddress',
  serialize: (value) => value,
  parseValue: (value) => value,
  parseLiteral: (ast) => {
    if (ast.kind !== Kind.STRING) {
      throw new GraphQLError(`Invalid kind: Expecting value to b
e a string.`, [ast]);
    }
    if (!validator.isEmail(ast.value, {allow_display_name: false
})) {
      throw new GraphQLError(`Validation failed: Expecting a val
id email address.`, [ast]);
    }
    return ast.value;
  }
});
```

Open the `app/schema/types/UserInputType.js` and update the code.

```
import EmailAddressType from '../scalars/EmailAddressType';

export default new GraphQLInputObjectType({
  ...
  fields: {
    ...
    email: {
      type: EmailAddressType
    }
  }
});
```

We can not run the mutation again and it will fail when the email field is invalid.