The Ultimate

Cheat Sheet

Node JS Cheat Sheet

TABLE OF CONTENTS

Running Node.js

For running Node.js:

Command	Comments
node	Run the Node REPL in your terminal
node —version	Print your current Node version
node filename.js	Execute the Node code in filename.js

REPL stands for **R**ead **E**val **P**rint **L**oop. This is the list of steps that happen when you run the `node` command and then type some code.

Node.js Global Object

In Node, we have a `global` object that we can always access. Features that we expect to be available everywhere live in this `global` object.

For example, to have some code execute after 5 seconds we can use either `global.setTimeout` or just `setTimeout`. The `global` keyword is optional.

```
setTimeout(() => {
  console.log('hello');
}, 5000);
```

Probably the most famous global is `global.console.log` which we write as just `console.log` .

Node.js Module System

In Node.js each file is treated as a separate module. Modules provide us a way of re-using existing code.

The require Function

We can re-use existing code by using the Node built-in `require()` function. This function imports code from another module.

```
const fs = require('fs');
fs.readFileSync('hello.txt');

// OR...

const { readFileSync } = require('fs');
readFileSync('hello.txt');
```

Built-in Modules

Some modules like fs are built in to Node. These modules contain Node-specific features.

Key built-in modules include:

- **fs** - read and write files on your file system
- **path** - combine paths regardless of which OS you're using
- **process** - information about the currently running process, e.g. process.argv for arguments passed in or process.env for environment variables
- **http** - make requests and create HTTP servers
- **https** - work with secure HTTP servers using SSL/TLS
- **events** - work with the EventEmitter
- **crypto** - cryptography tools like encryption and hashing

Creating Modules

We can create our own modules by exporting a function from a file and importing it in another module.

```
// In src/fileModule.js
function read(filename) { }
function write(filename, data) { }

module.exports = {
  read,
  write,
};

// In src/sayHello.js
```

```
const { write } = require('./fileModule.js')
write('hello.txt', 'Hello world!');
```

Some Node modules may instead use the shorthand syntax to export functions.

```
// In src/fileModule.js
exports.read = function read(filename) { }
exports.write = function write(filename, data) { }
```

ECMAScript Modules

The imports above use a syntax known as CommonJS (CJS) modules. Node treats JavaScript code as CommonJS modules by default. More recently, you may have seen the ECMAScript module (ESM) syntax. This is the syntax that is used by TypeScript.

```
// In src/fileModule.mjs
function read(filename) { }
function write(filename, data) { }

export {
  read,
  write,
};

// In src/sayHello.mjs
import { write } from './response.mjs';
write('hello.txt', 'Hello world!');
```

We tell Node to treat JavaScript code as an ECMAScript module by using the .mjs file extension. Pick one approach and use it consistently throughout your Node project.

Node.js Packages

Node developers often publicly share *packages*, that other developers can use to help solve common problems. A package is a collection of Node modules along with a *package.json* file describing the package.

To work with Node packages we use NPM. NPM includes two things:

1. The NPM registry with a massive collection of Node packages for us to use.

2. The NPM tool that you installed when you installed Node.

> 🔎 We can search the NPM registry for packages at www.npmjs.com. The NPM tool will by default install packages from this NPM registry.

NPM Commands

Command	Comments
npm start	Execute the current Node package defined by package.json. Defaults to executing `node server.js`.
npm init	Initialize a fresh package.json file
npm init -y	Initialize a fresh package.json file, accepting all default options. Equivalent to npm init —yes
npm install	Equivalent to npm i
npm install <package>	Install a package from the NPM registry at www.npmjs.com Equivalent to npm i <package>
npm install -D <package>	Install a package as a development dependency Equivalent to npm install —save-dev <package>
npm install -g <package>	Install a package globally.
npm update <package>	Update an already installed package Equivalent to npm up <package>
npm uninstall <package>	Uninstall a package from your node_modules/ folder Equivalent to npm un <package>
npm outdated	Check for outdated package dependencies
npm audit	Check for security vulnerabilities in package dependencies
npm audit fix	Try to fix any security vulnerabilities by automatically updating vulnerable packages.

package.json

Most Node applications we create include a *package.json* file, which means our Node applications are also Node packages.

The *package.json* file contains:

1. Name, version, description, license of the current package.
2. Scripts to automate tasks like starting, testing, and installing the current package.
3. Lists of dependencies that are required to be installed by the current package.

node_modules

This folder lives next to your package.json file.

When you run `npm install` the packages listed as dependencies in your package.json are downloaded from the NPM registry and put in the node_modules folder.

It contains not just your *direct dependencies*, but also the dependencies of those dependencies. The entire *dependency tree* lives in node_modules.

package-lock.json

The package-lock.json file is automatically created by NPM to track the exact versions of packages that are installed in your node_modules folder. Share your package-lock.json with other developers on your team to ensure that everyone is running the exact same versions of every package in the dependency tree.

Node.js Event Emitter

Node.js provides a built-in module to work with events.

```
const EventEmitter = require('events');
const celebrity = new EventEmitter();

celebrity.on('success', () => {
  console.log('Congratulations! You are the best!');
});

celebrity.emit('success'); // logs success message
celebrity.emit('success'); // logs success message again
celebrity.emit('failure'); // logs nothing
```

Many features of Node are modelled with the EventEmitter class. Some examples include the currently running Node `process`, a running HTTP server, and web sockets. They all *emit* events that can then be listened for using a listener function like `on()`.

For example, we can listen for the exit event on the current running process. In this case, the event has a code associated with it to be more specific about how the process is exiting.

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

process.on('exit', (code) => {
  console.log(`About to exit with code: ${code}`);
});
```

Backend Concepts

▼ Client-server architecture

Your frontend is usually the client. Your backend is usually the server.

In a client-server architecture, clients get access to data (or "resources") from the server. The client can then display and interact with this data.

The client and server communicate with each other using the HTTP protocol.

▼ API

Short for Application Programming Interface.

This is the set of functions or operations that your backend server supports. The frontend interacts with the backend by using only these operations.

On the web, backend APIs are commonly defined by a list of URLs, corresponding HTTP methods, and any queries and parameters.

▼ CRUD

Short for Create Read Update and Delete.

These are the basic operations that every API supports on collections of data. Your API will usually save (or "persist") these collections of data in a database.

▼ RESTful

RESTful APIs are those that follow certain constraints. These include:

- Client-server architecture. Clients get access to resources from the server using the HTTP protocol.

- Stateless communication. Each request contains all the information required by the server to handle that request. Every request is separate from every other request.
- Cacheable. The stateless communication makes caching easier.

In RESTful APIs each of our CRUD operations corresponds to an HTTP method.

CRUD Operation	HTTP method	Example
Create	POST	POST /cards Save a new card to the cards collection
Read	GET	GET /cards Get the whole cards collection or... GET /cards/:cardId Get an individual card
Update	PUT (or more rarely PATCH)	PUT /cards/:cardId Update an individual card
Delete	DELETE	DELETE /cards/:cardId Delete an individual card or more rarely... DELETE /cards Delete the entire collection of cards

Express.js

GET Routes

```
// Get a whole collection of JSON objects
app.get("/cards", (req, res) => {
  return res.json(cards);
});

// Get a specific item in a collection by ID
app.get("/cards/:cardId", (req, res) => {
  const cardId = req.params.cardId;
  return res.json(cards[cardId]);
});
```

POST Routes

```
app.post("/cards", (req, res) => {
  // Get body from the request
  const card = req.body;

  // Validate the body
```

```
  if (!card.value || !card.suit) {
    return res.status(400).json({
      error: 'Missing required card property',
    });
  }

  // Update your collection
  cards.push(card);

  // Send saved object in the response to verify
  return res.json(card);
});
```

Routers

```
// In src/cards.router.js
const cardsRouter = express.Router();

cardsRouter.get("/", (req, res) => {
  return res.json(cards);
});

cardsRouter.get("/:cardId", (req, res) => {
  const cardId = req.params.cardId;
  return res.json(cards[cardId]);
});

// In src/api.js
const cardsRouter = require('./cards.router');

const api = express.Router();

api.use('/cards', cardsRouter);
```

Node.js Folder Structure

One typical folder structure for an API following RESTful architecture and using the Express framework can be found below. Node servers typically follow the Model View Controller pattern. Models live together in one folder. Controllers are grouped together based on which feature or collection they are related to. Views are typically managed by the front end, although some Node servers may serve static HTML or use templating engines like Handlebars.

```
node-project/                    # top level project
  node_modules/                  # all installed node packages
  data/                          # static data files, if needed
    database.json
  src/                           # the source code for your server
    models/                      # models following the model-view-controller pattern
      comment.model.js
      post.model.js
    routes/                      # one folder for each collection in your API
      feeds/                     # folder for the user feeds collection
        feed.router.js           # router listing all possible routes for user feeds
        feed.controller.js       # controller with the implementation for each route
      posts/
        post.router.js
        post.controller.js
      api.js                     # top level router connecting all the above routes
    services/                    # any long running services or utilities
      mongo.js                   # e.g. connecting to a MongoDB database
    app.js                       # all Express middleware and routers
    server.js                    # the top level Node HTTP server
  .gitignore
  package-lock.json
  package.json
```

This is just a reference. In the real world, every project will have differences in the requirements and the ideal project structure.

Cross Origin Resource Sharing

Something *all* web developers soon come across is **Cross Origin Resource Sharing (CORS)**.

Browsers follow the **Same Origin Policy (SOP)**, which prevents requests being made across different origins. This is designed to stop malicious servers from stealing information that doesn't belong to them.

Cross Origin Resource Sharing (CORS) allows us to allow or whitelist other origins that we trust, so that we can make requests to servers that don't belong to us. For example, with CORS set up properly, https://www.mydomain.com could make a POST request to https://www.yourdomain.com

In Express we commonly set up CORS using the following middleware package: https://www.npmjs.com/package/cors

PM2 Commands

PM2 is a tool we use to create and manage Node.js clusters. It allows us to create clusters of processes, to manage those processes in production, and to keep our applications running forever. We can install the PM2 tool globally using `npm install -g pm2`.

Command	Comments
pm2 list	List the status of all processes managed by PM2.
pm2 start server.js -i 4	Start server.js in cluster mode with 4 processes.
pm2 start server.js -i 0	Start server.js in cluster mode with the maximum number of processes to take full advantage of your CPU.
pm2 logs	Show logs from all processes.
pm2 logs —lines 200	Show older logs up to 200 lines long.
pm2 monit	Display a real-time dashboard in your terminal with statistics for all processes.
pm2 stop 0	Stop running process with ID 0.
pm2 restart 0	Restart process with ID 0.
pm2 delete 0	Remove process with ID 0 from PM2's list of managed processes.
pm2 delete all	Remove all processes from PM2's list.
pm2 reload all	Zero downtime reload of all processes managed by PM2. For updating and reloading server code already running in production.

Useful Links

- Node.js Official Documentation
- Node.js Best Practices Compilation
- Phases of the Event Loop
- Semantic Versioning Reference
- ZTM Node.js Bootcamp