

Apis And Microservices Notes

Patrick Adams

November 1, 2020

Note: This is a draft copy of notes generated by free code camp.
<https://www.freecodecamp.org/>

Contents

1	Managing Packages With Npm	3
1.1	How to Use package.json, the Core of Any Node.js Project or npm Package	3
1.2	Add a Description to Your package.json	4
1.3	Add Keywords to Your package.json	4
1.4	Add a License to Your package.json	4
1.5	Add a Version to Your package.json	4
1.6	Expand Your Project with External Packages from npm	5
1.7	Manage npm Dependencies By Understanding Semantic Versioning	5
1.8	Use the Tilde-Character to Always Use the Latest Patch Version of a Dependency	5
1.9	Use the Caret-Character to Use the Latest Minor Version of a Dependency	5
1.10	Remove a Package from Your Dependencies	6
2	Basic Node And Express	7
2.1	Meet the Node console	7
2.2	Start a Working Express Server	7
2.3	Serve an HTML File	7
2.4	Serve Static Assets	7
2.5	Serve JSON on a Specific Route	8
2.6	Use the .env File	8
2.7	Implement a Root-Level Request Logger Middleware	8
2.8	Chain Middleware to Create a Time Server	9
2.9	Get Route Parameter Input from the Client	9
2.10	Get Query Parameter Input from the Client	9
2.11	Use body-parser to Parse POST Requests	9
2.12	Get Data from POST Requests	10
3	Mongodb And Mongoose	11
3.1	Install and Set Up Mongoose	11
3.2	Create a Model	11
3.3	Create and Save a Record of a Model	11
3.4	Create Many Records with model.create()	11
3.5	Use model.find() to Search Your Database	11
3.6	Use model.findOne() to Return a Single Matching Document from Your Database	12
3.7	Use model.findById() to Search Your Database By _id	12
3.8	Perform Classic Updates by Running Find, Edit, then Save	12
3.9	Perform New Updates on a Document Using model.findOneAndUpdate()	12
3.10	Delete One Document Using model.findByIdAndRemove	12
3.11	Delete Many Documents with model.remove()	12
3.12	Chain Search Query Helpers to Narrow Search Results	12
4	Apis And Microservices Projects	13
4.1	Timestamp Microservice	13
4.2	Request Header Parser Microservice	13
4.3	URL Shortener Microservice	13
4.4	Exercise Tracker	13
4.5	File Metadata Microservice	13

1 Managing Packages With Npm

1.1 How to Use package.json, the Core of Any Node.js Project or npm Package

The package.json file is the center of any Node.js project or npm package. It stores information about your project, similar to how the <head> section of an HTML document describes the content of a webpage. It

consists of a single JSON object where information is stored in key-value pairs. There are only two required fields; "name" and "version", but it's good practice to provide additional information about your project that could be useful to future users or maintainers.

If you look at the file tree of your project, you will find the package.json file on the top level of the tree. This is the file that you will be improving in the next couple of challenges.

One of the most common pieces of information in this file is the author field. It specifies who created the project, and can consist of a string or an object with contact or other details. An object is recommended for bigger projects, but a simple string like the following example will do for this project.

```
“{
  "author": "Jane Doe",
  “
```

1.2 Add a Description to Your package.json

The next part of a good package.json file is the description field; where a short, but informative description about your project belongs.

If you some day plan to publish a package to npm, this is the string that should sell your idea to the user when they decide whether to install your package or not. However, that's not the only use case for the description, it's a great way to summarize what a project does. It's just as important in any Node.js project to help other developers, future maintainers or even your future self understand the project quickly.

Regardless of what you plan for your project, a description is definitely recommended. Here's an example:

```
“{
  "description": "A project that does something awesome",
  “
```

1.3 Add Keywords to Your package.json

The keywords field is where you can describe your project using related keywords. Here's an example:

```
“{
  "keywords": [ "descriptive", "related", "words" ],
  “
```

As you can see, this field is structured as an array of double-quoted strings.

1.4 Add a License to Your package.json

The license field is where you inform users of what they are allowed to do with your project.

Some common licenses for open source projects include MIT and BSD. License information is not required, and copyright laws in most countries will give you ownership of what you create by default. However, it's always a good practice to explicitly state what users can and can't do. Here's an example of the license field:

```
“{
  "license": "MIT",
  “
```

1.5 Add a Version to Your package.json

A version is one of the required fields of your package.json file. This field describes the current version of your project. Here's an example:

```
“{
  "version": "1.2.0",
  “
```

1.6 Expand Your Project with External Packages from npm

One of the biggest reasons to use a package manager, is their powerful dependency management. Instead of manually having to make sure that you get all dependencies whenever you set up a project on a new computer, npm automatically installs everything for you. But how can npm know exactly what your project needs? Meet the dependencies section of your package.json file.

In this section, packages your project requires are stored using the following format:

```
“{
  "dependencies": {
    "package-name": "version",
    "express": "4.14.0"
  }
}”
```

1.7 Manage npm Dependencies By Understanding Semantic Versioning

Versions of the npm packages in the dependencies section of your package.json file follow what's called Semantic Versioning (SemVer), an industry standard for software versioning aiming to make it easier to manage dependencies. Libraries, frameworks or other tools published on npm should use SemVer in order to clearly communicate what kind of changes projects can expect if they update.

Knowing SemVer can be useful when you develop software that uses external dependencies (which you almost always do). One day, your understanding of these numbers will save you from accidentally introducing breaking changes to your project without understanding why things that worked yesterday suddenly don't work today. This is how Semantic Versioning works according to the official website:

```
“{
  "package": "MAJOR.MINOR.PATCH"
}”
```

The MAJOR version should increment when you make incompatible API changes.

The MINOR version should increment when you add functionality in a backwards-compatible manner.

The PATCH version should increment when you make backwards-compatible bug fixes.

This means that PATCHes are bug fixes and MINORs add new features but neither of them break what worked before. Finally, MAJORs add changes that won't work with earlier versions.

1.8 Use the Tilde-Character to Always Use the Latest Patch Version of a Dependency

In the last challenge, you told npm to only include a specific version of a package. That's a useful way to freeze your dependencies if you need to make sure that different parts of your project stay compatible with each other. But in most use cases, you don't want to miss bug fixes since they often include important security patches and (hopefully) don't break things in doing so.

To allow an npm dependency to update to the latest PATCH version, you can prefix the dependency's version with the tilde (~) character. Here's an example of how to allow updates to any 1.3.x version.

```
“{
  "package": "~1.3.8"
}”
```

1.9 Use the Caret-Character to Use the Latest Minor Version of a Dependency

Similar to how the tilde we learned about in the last challenge allows npm to install the latest PATCH for a dependency, the caret (^) allows npm to install future updates as well. The difference is that the caret will allow both MINOR updates and PATCHes.

Your current version of moment should be "~2.10.2" which allows npm to install to the latest 2.10.x version.

If you were to use the caret (^) as a version prefix instead, npm would be allowed to update to any 2.x.x version.

```
“{  
  "package": "^1.3.8"  
}
```

This would allow updates to any 1.x.x version of the package.

1.10 Remove a Package from Your Dependencies

You have now tested a few ways you can manage dependencies of your project by using the package.json's dependencies section. You have also included external packages by adding them to the file and even told npm what types of versions you want, by using special characters such as the tilde or the caret.

But what if you want to remove an external package that you no longer need? You might already have guessed it, just remove the corresponding key-value pair for that package from your dependencies.

This same method applies to removing other fields in your package.json as well

2 Basic Node And Express

2.1 Meet the Node console

During the development process, it is important to be able to check what's going on in your code.

Node is just a JavaScript environment. Like client side JavaScript, you can use the console to display useful debug information. On your local machine, you would see console output in a terminal. On Repl.it, a terminal is open in the right pane by default.

We recommend to keep the terminal open while working at these challenges. By reading the output in the terminal, you can see any errors that may occur.

2.2 Start a Working Express Server

In the first two lines of the file myApp.js, you can see how easy it is to create an Express app object. This object has several methods, and you will learn many of them in these challenges. One fundamental method is `app.listen(port)`. It tells your server to listen on a given port, putting it in running state. For testing reasons, we need the app to be running in the background so we added this method in the 'server.js' file for you.

Let's serve our first string! In Express, routes takes the following structure: `app.METHOD(PATH, HANDLER)`. `METHOD` is an http method in lowercase. `PATH` is a relative path on the server (it can be a string, or even a regular expression). `HANDLER` is a function that Express calls when the route is matched.

Handlers take the form `function(req, res) {...}`, where `req` is the request object, and `res` is the response object. For example, the handler

```
“js
function(req, res) {
  res.send('Response String');
}
“
```

will serve the string 'Response String'.

2.3 Serve an HTML File

You can respond to requests with a file using the `res.sendFile(path)` method. You can put it inside the `app.get('/', ...)` route handler. Behind the scenes, this method will set the appropriate headers to instruct your browser on how to handle the file you want to send, according to its type. Then it will read and send the file. This method needs an absolute file path. We recommend you to use the Node global variable `__dirname` to calculate the path like this:

```
“js
absolutePath = __dirname + relativePath/file.ext
“
```

2.4 Serve Static Assets

An HTML server usually has one or more directories that are accessible by the user. You can place there the static assets needed by your application (stylesheets, scripts, images). In Express, you can put in place this functionality using the middleware `express.static(path)`, where the `path` parameter is the absolute path of the folder containing the assets. If you don't know what middleware is... don't worry, we will discuss in detail later. Basically, middleware are functions that intercept route handlers, adding some kind of information. A middleware needs to be mounted using the method `app.use(path, middlewareFunction)`. The first `path` argument is optional. If you don't pass it, the middleware will be executed for all requests.

2.5 Serve JSON on a Specific Route

While an HTML server serves (you guessed it!) HTML, an API serves data. A REST (REpresentational State Transfer) API allows data exchange in a simple way, without the need for clients to know any detail about the server. The client only needs to know where the resource is (the URL), and the action it wants to perform on it (the verb). The GET verb is used when you are fetching some information, without modifying anything. These days, the preferred data format for moving information around the web is JSON. Simply put, JSON is a convenient way to represent a JavaScript object as a string, so it can be easily transmitted. Let's create a simple API by creating a route that responds with JSON at the path `/json`. You can do it as usual, with the `app.get()` method. Inside the route handler, use the method `res.json()`, passing in an object as an argument. This method closes the request-response loop, returning the data. Behind the scenes, it converts a valid JavaScript object into a string, then sets the appropriate headers to tell your browser that you are serving JSON, and sends the data back. A valid object has the usual structure `{key: data}`. `data` can be a number, a string, a nested object or an array. `data` can also be a variable or the result of a function call, in which case it will be evaluated before being converted into a string.

2.6 Use the .env File

The `.env` file is a hidden file that is used to pass environment variables to your application. This file is secret, no one but you can access it, and it can be used to store data that you want to keep private or hidden. For example, you can store API keys from external services or your database URI. You can also use it to store configuration options. By setting configuration options, you can change the behavior of your application, without the need to rewrite some code.

The environment variables are accessible from the app as `process.env.VAR_NAME`. The `process.env` object is a global Node object, and variables are passed as strings. By convention, the variable names are all uppercase, with words separated by an underscore. The `.env` is a shell file, so you don't need to wrap names or values in quotes. It is also important to note that there cannot be space around the equals sign when you are assigning values to your variables, e.g. `VAR_NAME=value`. Usually, you will put each variable definition on a separate line.

2.7 Implement a Root-Level Request Logger Middleware

Earlier, you were introduced to the `express.static()` middleware function. Now it's time to see what middleware is, in more detail. Middleware functions are functions that take 3 arguments: the request object, the response object, and the next function in the application's request-response cycle. These functions execute some code that can have side effects on the app, and usually add information to the request or response objects. They can also end the cycle by sending a response when some condition is met. If they don't send the response when they are done, they start the execution of the next function in the stack. This triggers calling the 3rd argument, `next()`.

Look at the following example:

```
“js
function(req, res, next) {
  console.log("I'm a middleware...");
  next();
}
“.
```

Let's suppose you mounted this function on a route. When a request matches the route, it displays the string “I'm a middleware...”, then it executes the next function in the stack.

In this exercise, you are going to build root-level middleware. As you have seen in challenge 4, to mount a middleware function at root level, you can use the `app.use(<mware-function>)` method. In this case, the function will be executed for all the requests, but you can also set more specific conditions. For example, if you want a function to be executed only for POST requests, you could use `app.post(<mware-function>)`.

Analogous methods exist for all the HTTP verbs (GET, DELETE, PUT, ...).

2.8 Chain Middleware to Create a Time Server

Middleware can be mounted at a specific route using `app.METHOD(path, middlewareFunction)`. Middleware can also be chained inside route definition.

Look at the following example:

```
“js
app.get('/user', function(req, res, next) {
  req.user = getUserSync(); // Hypothetical synchronous operation
  next();
}, function(req, res) {
  res.send(req.user);
});
“
```

This approach is useful to split the server operations into smaller units. That leads to a better app structure, and the possibility to reuse code in different places. This approach can also be used to perform some validation on the data. At each point of the middleware stack you can block the execution of the current chain and pass control to functions specifically designed to handle errors. Or you can pass control to the next matching route, to handle special cases. We will see how in the advanced Express section.

2.9 Get Route Parameter Input from the Client

When building an API, we have to allow users to communicate to us what they want to get from our service. For example, if the client is requesting information about a user stored in the database, they need a way to let us know which user they're interested in. One possible way to achieve this result is by using route parameters. Route parameters are named segments of the URL, delimited by slashes (/). Each segment captures the value of the part of the URL which matches its position. The captured values can be found in the `req.params` object.

route_path: '/user/:userId/book/:bookId' actual_request_URL: '/user/546/book/6754' req.params: {userId: '546', bookId: '6754'}

2.10 Get Query Parameter Input from the Client

Another common way to get input from the client is by encoding the data after the route path, using a query string. The query string is delimited by a question mark (?), and includes field=value couples. Each couple is separated by an ampersand (&). Express can parse the data from the query string, and populate the object `req.query`. Some characters, like the percent (%), cannot be in URLs and have to be encoded in a different format before you can send them. If you use the API from JavaScript, you can use specific methods to encode/decode these characters.

route_path: '/library' actual_request_URL: '/library?userId=546&bookId=6754' req.query: {userId: '546', bookId: '6754'}

2.11 Use body-parser to Parse POST Requests

Besides GET, there is another common HTTP verb, it is POST. POST is the default method used to send client data with HTML forms. In REST convention, POST is used to send data to create new items in the database (a new user, or a new blog post). You don't have a database in this project, but you are going to learn how to handle POST requests anyway.

In these kind of requests, the data doesn't appear in the URL, it is hidden in the request body. The body is a part of the HTTP request, also called the payload. Even though the data is not visible in the URL, this does not mean that it is private. To see why, look at the raw content of an HTTP POST request:

```
“http
POST /path/subpath HTTP/1.0
From: john@example.com
User-Agent: someBrowser/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 20
name=John+Doe&age=25
“
```

As you can see, the body is encoded like the query string. This is the default format used by HTML forms. With Ajax, you can also use JSON to handle data having a more complex structure. There is also another type of encoding: multipart/form-data. This one is used to upload binary files.

In this exercise, you will use a urlencoded body. To parse the data coming from POST requests, you have to install the body-parser package. This package allows you to use a series of middleware, which can decode data in different formats.

2.12 Get Data from POST Requests

Mount a POST handler at the path /name. It's the same path as before. We have prepared a form in the html frontpage. It will submit the same data of exercise 10 (Query string). If the body-parser is configured correctly, you should find the parameters in the object req.body. Have a look at the usual library example: route: POST '/library' urlencoded_body: {userId=546&bookId=6754} req.body: {userId: '546', bookId: '6754'}

Respond with the same JSON object as before: {name: 'firstname lastname'}. Test if your endpoint works using the html form we provided in the app frontpage.

Tip: There are several other http methods other than GET and POST. And by convention there is a correspondence between the http verb, and the operation you are going to execute on the server. The conventional mapping is:

POST (sometimes PUT) - Create a new resource using the information sent with the request,

GET - Read an existing resource without modifying it,

PUT or PATCH (sometimes POST) - Update a resource using the data sent,

DELETE => Delete a resource.

There are also a couple of other methods which are used to negotiate a connection with the server. Except from GET, all the other methods listed above can have a payload (i.e. the data into the request body). The body-parser middleware works with these methods as well.

3 MongoDB And Mongoose

3.1 Install and Set Up Mongoose

Add mongodb and mongoose to the project's package.json. Then require mongoose. Store your MongoDB Atlas database URI in the private .env file as MONGO_URI. Surround the the URI with single or double quotes and make sure no space exists between both the variable and the '=' and the value and '='. Connect to the database using the following syntax:

```
“js
mongoose.connect(, { useNewUrlParser: true, useUnifiedTopology: true });
“;
```

3.2 Create a Model

CRUD Part I - CREATE

First of all we need a Schema. Each schema maps to a MongoDB collection. It defines the shape of the documents within that collection.

Schemas are building block for Models. They can be nested to create complex models, but in this case we'll keep things simple.

A model allows you to create instances of your objects, called documents.

Repl.it is a real server, and in real servers the interactions with the db happen in handler functions. These function are executed when some event happens (e.g. someone hits an endpoint on your API). We'll follow the same approach in these exercises. The done() function is a callback that tells us that we can proceed after completing an asynchronous operation such as inserting, searching, updating or deleting. It's following the Node convention and should be called as done(null, data) on success, or done(err) on error.

Warning - When interacting with remote services, errors may occur!

```
“js
/* Example */
var someFunc = function(done) {
//... do something (risky) ...
if(error) return done(error);
done(null, result);
};
“;
```

3.3 Create and Save a Record of a Model

In this challenge you will have to create and save a record of a model.

3.4 Create Many Records with model.create()

Sometimes you need to create many instances of your models, e.g. when seeding a database with initial data. Model.create() takes an array of objects like [{name: 'John', ...}, {...}, ...] as the first argument, and saves them all in the db.

3.5 Use model.find() to Search Your Database

Find all the people having a given name, using Model.find() -> [Person]

In its simplest usage, Model.find() accepts a query document (a JSON object) as the first argument, then a callback. It returns an array of matches. It supports an extremely wide range of search options. Check it in

the docs. Use the function argument `personName` as search key.

3.6 Use `model.findOne()` to Return a Single Matching Document from Your Database

`Model.findOne()` behaves like `.find()`, but it returns only one document (not an array), even if there are multiple items. It is especially useful when searching by properties that you have declared as unique.

3.7 Use `model.findById()` to Search Your Database By `_id`

When saving a document, `mongodb` automatically adds the field `_id`, and set it to a unique alphanumeric key. Searching by `_id` is an extremely frequent operation, so `mongoose` provides a dedicated method for it.

3.8 Perform Classic Updates by Running Find, Edit, then Save

In the good old days this was what you needed to do if you wanted to edit a document and be able to use it somehow e.g. sending it back in a server response. `Mongoose` has a dedicated updating method : `Model.update()`. It is bound to the low-level `mongo` driver. It can bulk edit many documents matching certain criteria, but it doesn't send back the updated document, only a 'status' message. Furthermore it makes model validations difficult, because it just directly calls the `mongo` driver.

3.9 Perform New Updates on a Document Using `model.findOneAndUpdate()`

Recent versions of `mongoose` have methods to simplify documents updating. Some more advanced features (i.e. pre/post hooks, validation) behave differently with this approach, so the Classic method is still useful in many situations. `findByIdAndUpdate()` can be used when searching by Id.

3.10 Delete One Document Using `model.findByIdAndRemove`

Delete one person by the person's `_id`. You should use one of the methods `findByIdAndRemove()` or `findOneAndRemove()`. They are like the previous update methods. They pass the removed document to the db. As usual, use the function argument `personId` as the search key.

3.11 Delete Many Documents with `model.remove()`

`Model.remove()` is useful to delete all the documents matching given criteria.

3.12 Chain Search Query Helpers to Narrow Search Results

If you don't pass the callback as the last argument to `Model.find()` (or to the other search methods), the query is not executed. You can store the query in a variable for later use. This kind of object enables you to build up a query using chaining syntax. The actual db search is executed when you finally chain the method `.exec()`. You always need to pass your callback to this last method. There are many query helpers, here we'll use the most 'famous' ones.

4 Apis And Microservices Projects

4.1 Timestamp Microservice

Build a full stack JavaScript app that is functionally similar to this: <https://timestamp-microservice.freecodecamp.rocks/>. Working on this project will involve you writing your code on Repl.it on our starter project. After completing this project you can copy your public Repl.it URL (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but it must be publicly visible for our testing.

Start this project on Repl.it using this link or clone this repository on GitHub! If you use Repl.it, remember to save the link to your project somewhere safe!

4.2 Request Header Parser Microservice

Build a full stack JavaScript app that is functionally similar to this: <https://request-header-parser-microservice.freecodecamp.rocks/>. Working on this project will involve you writing your code using one of the following methods:

- Clone this GitHub repo and complete your project locally.
- Use our repl.it starter project to complete your project.
- Use a site builder of your choice to complete the project. Be sure to incorporate all the files from our GitHub repo.

When you are done, make sure a working demo of your project is hosted somewhere public. Then submit the URL to it in the 'Solution Link' field. Optionally, also submit a link to your project's source code in the 'GitHub Link' field.

4.3 URL Shortener Microservice

Build a full stack JavaScript app that is functionally similar to this: <https://url-shortener-microservice.freecodecamp.rocks/>. Working on this project will involve you writing your code on Repl.it on our starter project. After completing this project you can copy your public Repl.it URL (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but it must be publicly visible for our testing.

Start this project on Repl.it using this link or clone this repository on GitHub! If you use Repl.it, remember to save the link to your project somewhere safe!

4.4 Exercise Tracker

Build a full stack JavaScript app that is functionally similar to this: <https://exercise-tracker.freecodecamp.rocks/>. Working on this project will involve you writing your code on Repl.it on our starter project. After completing this project you can copy your public Repl.it url (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but it must be publicly visible for our testing.

Start this project on Repl.it using this link or clone this repository on GitHub! If you use Repl.it, remember to save the link to your project somewhere safe!

4.5 File Metadata Microservice

Build a full stack JavaScript app that is functionally similar to this: <https://file-metadata-microservice.freecodecamp.rocks/>. Working on this project will involve you writing your code on Repl.it on our starter project. After completing this project you can copy your public Repl.it URL (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but it must be publicly visible for our testing.

Start this project on Repl.it using this link or clone this repository on GitHub! If you use Repl.it, remember to save the link to your project somewhere safe!