

Managing Packages with Npm

Introduction to the Managing Packages with npm Challenges

The Node Package Manager (npm) is a command-line tool used by developers to share and control modules (or packages) of JavaScript code written for use with Node.js.

When starting a new project, npm generates a `package.json` file. This file lists the package dependencies for your project. Since npm packages are regularly updated, the `package.json` file allows you to set specific version numbers for each dependency. This ensures that updates to a package don't break your project.

npm saves packages in a folder named `node_modules`. These packages can be installed in two ways:

1. globally in a root `node_modules` folder, accessible by all projects.
2. locally within a project's own `node_modules` folder, accessible only to that project.

Most developers prefer to install packages local to each project to create a separation between the dependencies of different projects. Working on these challenges will involve you writing your code on Glitch on our starter project. After completing each challenge you can copy your public Glitch url (to the homepage of your app) into the challenge screen to test it! Optionally you may choose to write your project on another platform but it must be publicly visible for our testing.

Managing Packages with Npm - 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",
```

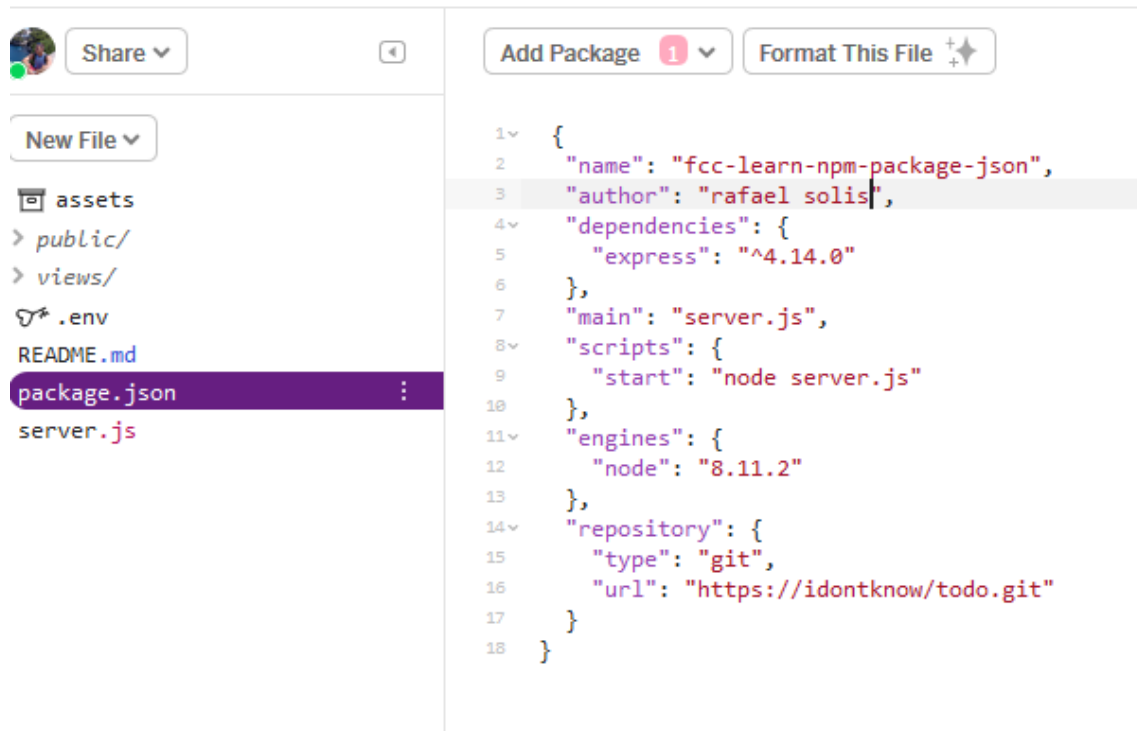
Add your name as the `author` of the project in the `package.json` file.

Note: Remember that you're writing JSON, so all field names must use double-quotes (") and be separated with a comma (,).

//Configuración del Servidor de Glich



```
1-  /*****
2  * PLEASE DO NOT EDIT THIS FILE
3  * the verification process may break
4  * *****/
5
6  'use strict';
7
8  var fs = require('fs');
9  var express = require('express');
10 var app = express();
11
12 if (!process.env.DISABLE_XORIGIN) {
13   app.use(function(req, res, next) {
14     var allowedOrigins = ['https://nanrow-plane.gonix.me', 'https://www.freecodecamp.com'];
15     var origin = req.headers.origin || '';
16     if(!process.env.XORIGIN_RESTRICT || allowedOrigins.indexOf(origin) > -1){
17       console.log(origin);
18       res.setHeader('Access-Control-Allow-Origin', origin);
19       res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
20     }
21     next();
22   });
23 }
24
25 app.use('/public', express.static(process.cwd() + '/public'));
26
27 app.route('/_api/package.json')
28   .get(function(req, res, next) {
29     console.log("requested");
30     fs.readFile(__dirname + '/package.json', function(err, data) {
31       if(err) return next(err);
32       res.type('txt').send(data.toString());
33     });
34   });
35
36 app.route('/')
37   .get(function(req, res) {
38     res.sendFile(process.cwd() + '/views/index.html');
39   })
40
41 // Respond not found to all the wrong routes
42 app.use(function(req, res, next){
43   res.status(404);
44   res.type('txt').send("Not found");
45 });
46
47 // Error Middleware
48 app.use(function(err, req, res, next) {
49   if(err) {
50     res.status(err.status || 500)
51     .type('txt')
52     .send(err.message || 'SERVER ERROR');
53   }
54 })
55
56 app.listen(process.env.PORT, function () {
57   console.log("Node.js listening ...");
58 });
59
```



Managing Packages with Npm - 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",
```

```
{
  "name": "fcc-learn-npm-package-json",
  "author": "rafael solis",
  "description": "this package contains any items to work",
  "dependencies": {
    "express": "^4.14.0"
  },
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "engines": {
    "node": "8.11.2"
  },
  "repository": {
    "type": "git",
    "url": "https://idontknow/todo.git"
  }
}
```

Managing Packages with Npm - 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.

Add an array of suitable strings to the `keywords` field in the `package.json` file of your project.

One of the keywords should be `"freecodecamp"`.

```
{
  "name": "fcc-learn-npm-package-json",
  "author": "rafael solis",
  "description": "this package contains any items to work",
  "keywords": [ "practise", "json", "npm", "freecodecamp" ],
  "dependencies": {
    "express": "^4.14.0"
  },
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "engines": {
    "node": "8.11.2"
  },
  "repository": {
    "type": "git",
    "url": "https://idontknow/todo.git"
  }
}
```

Managing Packages with Npm - 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",
```

Fill the `license` field in the `package.json` file of your project as you find suitable.

```
{
  "name": "fcc-learn-npm-package-json",
  "author": "rafael solis",
  "description": "this package contains any items to work",
  "keywords": [ "practise", "json", "npm", "freecodecamp" ],
  "license": "MIT",
  "dependencies": {
    "express": "^4.14.0"
  },
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "engines": {
    "node": "8.11.2"
  },
  "repository": {
    "type": "git",
    "url": "https://idontknow/todo.git"
  }
}
```

Managing Packages with Npm - 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",
```

Add a version to the package.json file of your project.

Solution

```
/**
 *
 * Test output will go here
 *
 */
package.json should have a valid "version" key
```

```
{
  "name": "fcc-learn-npm-package-json",
  "author": "rafael solis",
  "description": "this package contains any items to work",
  "keywords": [ "practise", "json", "npm", "freecodecamp" ],
  "license": "MIT",
  "version": "1.2.0",
  "dependencies": {
    "express": "^4.14.0"
  },
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "engines": {
    "node": "8.11.2"
  },
  "repository": {
    "type": "git",
    "url": "https://idontknow/todo.git"
  }
}
```

Managing Packages with Npm - 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"
}
```

Add version "2.14.0" of the "moment" package to the `dependencies` field of your `package.json` file.

Note: Moment is a handy library for working with time and dates.

```
{
  "name": "fcc-learn-npm-package-json",
  "author": "rafael solis",
  "description": "this package contains any items to work",
  "keywords": [ "practise", "json", "npm", "freecodecamp" ],
  "license": "MIT",
  "version": "1.2.0",
  "dependencies": {
    "package-name": "version",
    "express": "^1.2.0",
    "moment": "^2.14.0"
  },
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "engines": {
    "node": "8.11.2"
  },
  "repository": {
    "type": "git",
    "url": "https://idontknow/todo.git"
  }
}
```

Managing Packages with Npm - 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.

In the dependencies section of your package.json file, change the `version` of `moment` to match MAJOR version 2, MINOR version 10 and PATCH version 2


```
{
  "name": "fcc-learn-npm-package-json",
  "author": "rafael solis",
  "description": "this package contains any items to work",
  "keywords": [ "practise", "json", "npm", "freecodecamp" ],
  "license": "MIT",
  "version": "1.2.0",
  "dependencies": {
    "package-name": "version",
    "express": "^2.10.2",
    "moment": "2.10.2"
  },
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "engines": {
    "node": "8.11.2"
  },
  "repository": {
    "type": "git",
    "url": "https://idontknow/todo.git"
  }
}
```

Managing Packages with Npm - 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"
```

In the package.json file, your current rule for how npm may upgrade moment is to use a specific version (2.10.2). But now, you want to allow the latest 2.10.x version.

Use the tilde (~) character to prefix the version of moment in your dependencies, and allow npm to update it to any new PATCH release.

Note: The version numbers themselves should not be changed.

```
"dependencies": {  
  "package-name": "version",  
  "express": "2.10.2",  
  "moment": "~2.10.2"
```

```
},
```

Managing Packages with Npm - 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.

Use the caret (^) to prefix the version of moment in your dependencies and allow npm to update it to any new MINOR release.

Note: The version numbers themselves should not be changed.

```
"dependencies": {  
  "package-name": "version",  
  "express": "2.10.2",  
  "moment": "^2.10.2"
```

Managing Packages with Npm - 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

Remove the moment package from your dependencies.

Note: Make sure you have the right amount of commas after removing it.

```
"dependencies": {  
  "package-name": "version",  
  "express": "2.10.2"  
},
```

¿Qué es el package.json?

De cierta forma, podemos considerar este **package.json** como un manifiesto de nuestro proyecto.

Históricamente, Node ha trabajado con una herramienta para administrar paquetes llamada **npm** (*o narwhals play music, de acuerdo a los calcetines de npm que alguna vez [@fforres](#) se ganó en una conferencia*). Esta herramienta, que normalmente se instala junto con Node, tiene dos roles fundamentales:

- Manejar la publicación de un proyecto al registro público de npm (*para que otros puedan descargarlo y utilizarlo como dependencia en sus propios proyectos*).
- Administrar las dependencias de tu proyecto.

Para esto, guarda un registro en un archivo llamado, justamente, **package.json**.

Dentro de este archivo se definen y manejan características como:

- Nombre de tu proyecto.
- Versión.
- Dependencias.
- Repositorio.
- Autores.
- Licencia.

Y más.

*(Adicionalmente, desde hace un tiempo, nuestros amigos personales de Facebook — aunque nosotros somos más amigos de ellos, que ellos de nosotros —, lanzaron una nueva herramienta de administración de paquetes para Node llamada **yarn**. Su funcionamiento, al menos en cuanto al uso del **package.json**, es prácticamente igual, por lo que al menos para efectos de este artículo, podemos considerar lo mismo para ambos, excepto en casos donde indiquemos explícitamente lo contrario.)*

A través de este archivo, finalmente, se puede garantizar la integridad del proyecto. Es decir, podemos asegurar que quienes tengan una copia del mismo, podrán acceder a las mismas propiedades y sincronizar entre múltiples partes cada vez que decidan hacer un cambio. De cierta forma, podemos considerar este **package.json** como un manifiesto de nuestro proyecto.

Por ejemplo, consideremos el siguiente escenario:

Dos personas están trabajando en el mismo proyecto, con copias independientes en cada uno de sus equipos. El primero de ellos determina que para completar la nueva funcionalidad, va a necesitar implementar una nueva librería al proyecto.

Antiguamente, sin manejo de dependencias, era necesario hacer una de dos cosas:

- Incluir la librería (*1 o múltiples archivo(s)*) manualmente en el directorio del proyecto, potencialmente aumentando el peso del proyecto de manera considerable.
- No incluir la librería, pero comunicarle a cada persona que obtuviera una copia del mismo que antes de trabajar en el proyecto necesitaría instalar esta nueva dependencia (*buena forma de hacer nuevos amigos, poco óptimo en términos de tiempo*).

Con el uso de administradores de dependencias, ya estos pasos no son necesarios. Ahora cada persona que decida obtener una copia del proyecto, desde ahora al final de los tiempos, puede instalar todas y cada una de las dependencias que tengamos declaradas en este “manifiesto” sin la necesidad de incluir una copia de éstas en ningún otro lado más que ahí.

Cabe mencionar que si bien muchas características del package.json parecieran ser específicas para proyectos publicados en el registro de npm (*como librerías*), también aplican para proyectos cuya finalidad no es ser publicados en ningún registro (*como por ejemplo aplicaciones Web o móviles, juegos y otros*), pero que si se benefician de las utilidades relacionadas a la administración de dependencias.

¿Cómo crearlo?

One rule, to ring them all (?)

Antes de crear un package.json, hay solo una regla a tener en consideración: El archivo debe ser un JSON de formato válido (*no puede ser un objeto literal de JS exportado desde un archivo*), con todas las especificaciones que esto implica (*por ejemplo, cada key debe tener comillas dobles, solo ciertos valores son válidos, etc.*)

[Más información sobre reglas y formatos permitidos.](#)

Para crearlo, hay 2 formas: hacerlo de forma manual o hacerlo de forma automática:

Creando un package.json manualmente

Si bien es recomendable usar alguno de los asistentes para crear el archivo de forma automática, en caso de que necesitemos hacerlo de forma manual, es solo cosa de crear un archivo llamado **package.json** en la raíz del proyecto e incluir, como mínimo, la siguiente información:

- name.
- version.

Todos los demás campos son opcionales, aunque recomendados.

Creando un package.json automáticamente

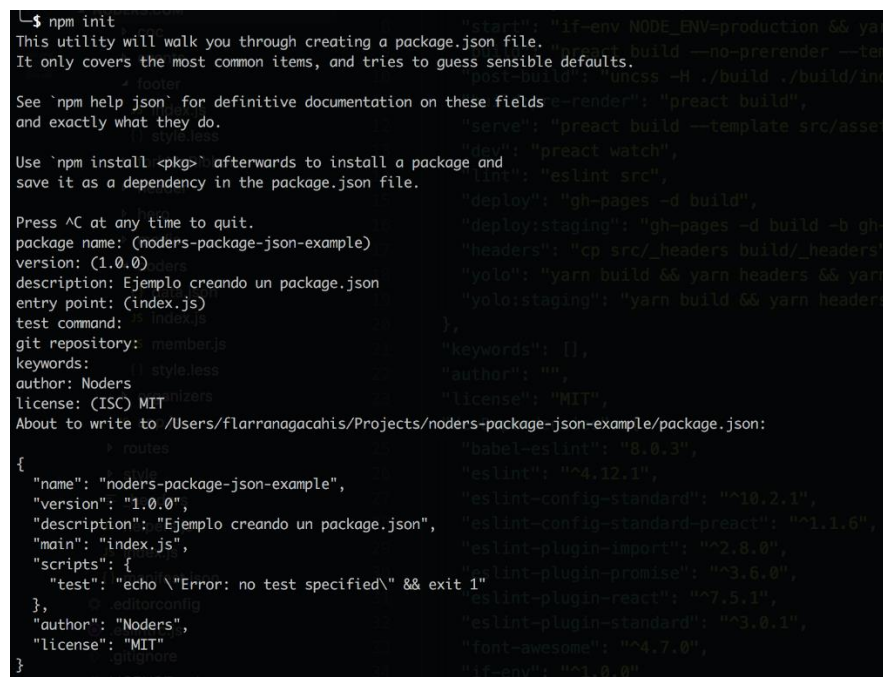
Es la forma más rápida de hacerlo, ya que tanto npm como yarn incluyen un asistente que nos permite crear un package.json con un solo comando:

```
npm init
```

o bien

```
yarn init
```

Dependiendo de cual usemos, el asistente nos hará algunas preguntas para definir la información del proyecto (*nombre, version, archivo de entrada, licencia y repositorio entre otros*)



```

$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (noders-package-json-example)
version: (1.0.0)
description: Ejemplo creando un package.json
entry point: (index.js)
test command:
git repository:
keywords:
author: Noders
license: (ISC) MIT
About to write to /Users/Flarranagacahis/Projects/noders-package-json-example/package.json:
{
  "name": "noders-package-json-example",
  "version": "1.0.0",
  "description": "Ejemplo creando un package.json",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Noders",
  "license": "MIT"
}
```

Creando un package.json con npm

```
$ yarn init
yarn init v1.3.2
question name (noders-package-json-example):
question version (1.0.0):
question description: Ejemplo creando un package.json
question entry point (index.js):
question repository url:
question author: Noders
question license (MIT):
question private:
success Saved package.json
👉 Done in 28.53s.
```

Creando un package.json con yarn

Al terminar, tendremos un nuevo y flamante **package.json** en la raíz del directorio donde hayamos ejecutado el comando.

Las Secciones del package.json

Al crear el package.json, ya bien de forma manual o automáticamente, vamos a encontrar dentro de él un gran objeto con múltiples keys y valores (*como la imagen de cabecera de este artículo*). Adicionalmente es muy probable, que con el tiempo, vayamos agregando nuevas keys y/o veamos en otros proyectos algunas de las cuales ni siquiera teníamos idea que existían. Para no dejar espacio a dudas, este es un listado de las keys que podemos agregar, que significa cada una, y algunas recomendaciones a tener en cuenta:

Nota: Estas son las propiedades oficiales soportadas por npm. Hay múltiples librerías que adicionalmente soportan incluir otros campos en el package.json y leen desde archivo (*ej. Jest y la propiedad "jest"*)

name

Uno de los dos campos obligatorios (*junto a version*). Es un string que representa el nombre del proyecto actual y forma un identificador único entre este campo y version en caso de que sea publicado al registro de npm.

Reglas:

- El nombre no puede contener mayúsculas ni empezar con un punto o guión bajo.
- El largo máximo del nombre es 214 caracteres, cada uno de los cuales debe ser URL safe (*más información sobre caracteres URL safe en [este documento](#), sección 2.3*)

Algunas otras cosas a tener en consideración:

- Si queremos publicar el proyecto al registro de npm, hay que validar que no exista anteriormente un nombre con el mismo proyecto.

- Hay algunas buenas prácticas con respecto al formato de nombres de algunos proyectos que es bueno chequear. Por ejemplo, si bien no es considerado buena práctica incluir “**node**” o “**js**” en los paquetes, si se acostumbra incluir “**react**” en el nombre de paquetes orientados a esa tecnología.

version

Como el nombre lo indica, es un string con la versión actual del proyecto. Los paquetes/proyectos/librerías en Node y JS siguen las convenciones definidas en Semantic Versioning (*o semver para los más amigos*), la cual define la forma de versionamiento:

MAJOR.MINOR.PATCH

[Más información sobre semver.](#)

description

Un string que describa lo que hace este proyecto. En caso de que decidamos publicar en el registro de npm, este texto ayudará a la gente a encontrarlo mediante la búsqueda de npm.

keywords

Igual a description, pero en vez de texto, es un array de strings que incluye términos que puedan ser utilizados para una eventual búsqueda.

homepage

Es un string con la URL del proyecto.

bugs

Es un string con una URL válida donde se puedan reportar problemas con el proyecto. Usualmente el link de los issues del repositorio.

license

Es un string que especifica que tipo de licencia definimos para el uso de este proyecto, ya sea de forma personal, comercial, abierta y/o privada.

[Más información sobre licencias disponibles.](#)

author

Puede ser un string o un objeto con la información del creador del proyecto.

Si es un objeto, incluye las siguientes propiedades:

- name
- email
- url

Si es un string, es en formato:

```
"Nombre <email> (url) "
```

contributors

Igual a author, pero en vez de un objeto o un string, es un array de alguno de estos 2, que incluye la información de colaboradores del proyecto.

files

Es un array de strings o patrones (*ejemplo: *.js*) que serán incluidos en caso de publicar el proyecto en el registro de npm. Si no incluimos esta sección, todos los archivos serán publicados, a excepción de los que tengamos definidos automáticamente para excluir (*por ejemplo los definidos dentro del .gitignore*).

Algunas consideraciones:

- Como alternativa a esta sección, se puede incluir un **.npmignore** que funciona de manera similar a un **.gitignore** dentro de un proyecto.
- En caso de no tener la sección files, ni un .npmignore, se tomará el contenido del .gitignore como referencia para excluir archivos.
- Algunos archivos serán **siempre incluidos**, independiente de lo que definamos: package.json, README, CHANGES / CHANGELOG / HISTORY, LICENSE / LICENCE, NOTICE y el archivo definido en la sección **main** del package.json (*más detalles a continuación*).
- Y tal como eso, hay algunos archivos que **siempre serán ignorados**, independiente de lo que definamos. Pueden encontrar una lista completa en [este enlace](#).

main

Un string que define la ruta del archivo principal o punto de entrada de tu proyecto. Este es el archivo que una persona recibirá si importa tu proyecto al suyo. Por ejemplo:

Si tu proyecto o paquete se llama **super-awesome-library** y alguien lo instala y hace en uno de sus archivos

```
const super-awesome-library = require("super-awesome-library")
```

obtendrá el contenido del archivo que definamos en el main y se asignará a la constante.

bin

Un string (*si es uno solo*) o un objeto (*si son múltiples*) en el que podemos definir scripts que queremos instalar como ejecutables en el PATH. Al instalar el paquete, se creará un enlace simbólico desde /usr/local/bin hacia un archivo dentro de nuestro proyecto, y lo convertirá de esa forma en un ejecutable.

Por ejemplo, si tenemos un script llamado **cli.js** en nuestro proyecto y queremos que se convierta en un ejecutable, podemos agregarlo al package.json de la siguiente forma:

```
{
  "name": "super-awesome-library",
  "bin": "cli.js"
}
```

Si lo agregamos de esa forma (*como un string solamente*), este se va a agregar como un ejecutable utilizando el nombre del proyecto (*en este caso "super-awesome-library"*). Con esto, podríamos ejecutar desde la consola:

```
super-awesome-library
```

Y lo que estaría pasando por debajo es que en realidad se correría algo como:

```
node cli.js
```

Y se ejecutaría el contenido de este archivo.

Por otro lado, si tenemos múltiples archivos que queremos agregar como ejecutables, podemos agregar un objeto como bin, y este creará un enlace por cada propiedad del objeto, apuntando al script que dejemos como valor. Por ejemplo, si definimos:

```
{
  "bin": {
    "script-1": "script1.js",
    "script-2": "script2.js"
  }
}
```

Se agregarán tanto "script-1" como "script-2" al PATH, apuntando a sus respectivos archivos .js (*Los nombres los podemos definir como estimemos conveniente, no es necesario que sean iguales*).

Esto es lo que utilizan muchos paquetes conocidos, como **nodemon** o **react-native**, para que cuando los instalemos como dependencia los podamos ejecutar directamente, sin necesidad de incluir la ruta completa de donde están.

man

Un string, o un array de strings, que especifica uno (*o muchos archivos*) que se relacionarán a este proyecto si se corre el comando **man** en la máquina donde se haya instalado.

directories

Un objeto que especifica las rutas para la estructura de directorios del proyecto.

Ejemplo:

```
{
  "bin": "./bin",
  "doc": "./doc",
  "lib": "./lib"
}
```

repository

Un objeto que especifica el tipo y URL del repositorio donde está el código del proyecto. Se usa el siguiente formato:

```
{
  "type": string,
  "url": string
}
```

Ejemplo:

```
{
  "type": "git",
  "url": "https://github.com/mi-usuario/mi-proyecto"
}
```

scripts

Es un objeto que indica comandos que podemos correr dentro de nuestro proyecto, asociándolos a una palabra clave para que npm (o yarn) los reconozca cuando queramos ejecutarlos.

Hay algunos scripts que vienen predefinidos en todos los proyectos al momento de utilizar npm, como son: **start**, **install**, **preinstall**, **pretest**, **test** y **posttest** entre otros (para una lista completa, pueden revisar [este enlace](#)).

De la misma forma, podemos definir algunos scripts personalizados y asociarlos al tipo de comandos que queramos, ahorrándonos recordar comandos completos que pueden ser simplificados al incluirlos acá.

Por ejemplo: Imaginemos que antes de lanzar una nueva versión de la aplicación en la que estamos trabajando, queremos ejecutar una tarea para minificar los archivos JS del proyecto, y esto lo hacemos mediante un script que tenemos en la ruta **tasks/minify.js**. Normalmente, lo que tendríamos que hacer, sería ejecutar cada vez que lo recordemos **node tasks/minify.js** y esperar que haga su trabajo. Sin embargo, si lo agregamos a los scripts de esta forma:

```
"scripts": {
  "minify": "node tasks/minify.js"
}
```

Podemos ejecutar

npm run minify

y la tarea se va a ejecutar de la misma forma que si hubiésemos corrido el comando directamente. La gracia dentro de esto, es que en un mismo script definido en el package.json, podemos combinar múltiples comandos e incluso múltiples scripts definidos en el mismo package.json, con lo que podemos encadenar tareas y armar nuestros propios flujos de trabajo automatizados.

Como última nota, para correr un script definido en el package.json, debemos hacer **npm run <script>**, a no ser de que sea alguno de los scripts predefinidos (*nombrados más arriba*), los cuales se pueden correr directamente con **npm <script>**. Si están usando yarn, pueden omitir la palabra run completamente y solo ejecutar **yarn <script>**, independiente de si son predefinidos o no.

config

Es un objeto al que podemos pasarle valores y usarlos como variables de ambiente dentro de nuestro proyecto. Cualquier usuario que importe este proyecto al suyo, podrá reescribir esas variables con valores propios y utilizarlos de manera normal.

dependencies

Un objeto que guarda los nombres y versiones de cada dependencia que hemos instalado dentro del proyecto. De esta forma, cada vez que alguien obtenga una copia de este proyecto, y corra el comando **npm install**, se instalarán todas las dependencias que aquí estén definidas y por ende, no habrán problemas de compatibilidad al correr el proyecto. Estas dependencias, así como las de las siguientes categorías, se definen de la siguiente forma:

```
"nombre-de-la-dependencia": "(^|~|version) |url"
```

Algunos ejemplos:

```
"dependencies": {
  "backbone": "1.0.0",
  "lodash": "^4.6.1",
  "mocha": "~3.5.3",
  "super-mega-libreria": "https://noders.com/super-mega-libreria-4.0.0.tar.gz"
}
```

Las dependencias pueden o bien: llevar como valor la versión que están utilizando, o de una URL desde donde obtenerla (*incluso una ruta local en la misma máquina*), la cual por lo general apunta a una versión específica.

¿Qué son los símbolos ^ y ~, que acompañan al número de versión?

Son caracteres opcionales que definen como debe ser tratada esa dependencia la próxima vez que se corra **npm install** en el proyecto:

- **Si la versión tiene un ^:** Se buscará una versión compatible con la que está definida ahí.
- **Si la versión tiene un ~:** Se buscará una versión lo más cercana posible a la definida.
- **Si no tiene ningún símbolo:** Se instalará la misma versión.

Si bien estos 3 son los más comunes, hay algunos otros que también nos podemos encontrar por ahí. Para mayor información sobre todos los existentes, pueden revisar [este enlace](#).

devDependencies

Mismo formato que las dependencias, pero acá podemos incluir todas aquellas librerías que no son necesarias para que este proyecto corra en producción, o cuando sea requerido e instalado dentro de otro. Con esto le ahorramos a quien importe este proyecto de tener dependencias instaladas que pueden no ser necesarias para que esto funcione (*como por ejemplo herramientas de tests*).

peerDependencies

Mismo formato anterior, pero acá definimos dependencias que son necesarias para el uso del proyecto, aún cuando no las tengamos instaladas dentro del proyecto en sí. De esta forma, podemos asegurar que se cumplan ciertas reglas de compatibilidad. Por ejemplo, podemos definir que este proyecto es solo compatible con la versión 16 de react:

```
{
  "peerDependencies": {
    "react": "16.0.0"
  }
}
```

De esta forma se hará un chequeo al momento de importar este proyecto que las condiciones estén cumplidas.

engines

Un objeto en el cual podemos definir la versión mínima de node y npm necesarias para correr este proyecto. La definimos de la forma:

```
"engines": {
  "node": "≥ 6.0.0",
  "npm": "≥ 3.0.0"
}
```

Al momento de instalar el proyecto, se verificará que versiones están instaladas actualmente, y de no cumplir el requisito, no se continuará el proceso de instalación.

Al igual que con las dependencias, podemos usar ~ y ^ junto al número de versión.

Introduction to the Basic Node and Express Challenges

Introduction to the Basic Node and Express Challenges

Node.js is a JavaScript runtime that allows developers to write backend (server-side) programs in JavaScript. Node.js comes with a handful of built-in modules - small, independent programs - that help facilitate this purpose. Some of the core modules include:

- HTTP: a module that acts as a server
- File System: a module that reads and modifies files
- Path: a module for working with directory and file paths
- Assertion Testing: a module that checks code against prescribed constraints

Express, while not included with Node.js, is another module often used with it. Express runs between the server created by Node.js and the frontend pages of a web application. Express also handles an application's routing. Routing directs users to the correct page based on their interaction with the application. While there are alternatives to using Express, its simplicity makes it a good place to begin when learning the interaction between a backend powered by Node.js and the frontend.

Working on these challenges will involve you writing your code on Glitch on our starter project. After completing each challenge you can copy your public Glitch url (to the homepage of your app) into the challenge 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 Glitch using [this link](#) or clone [this repository](#) on GitHub! If you use Glitch, remember to save the link to your project somewhere safe!

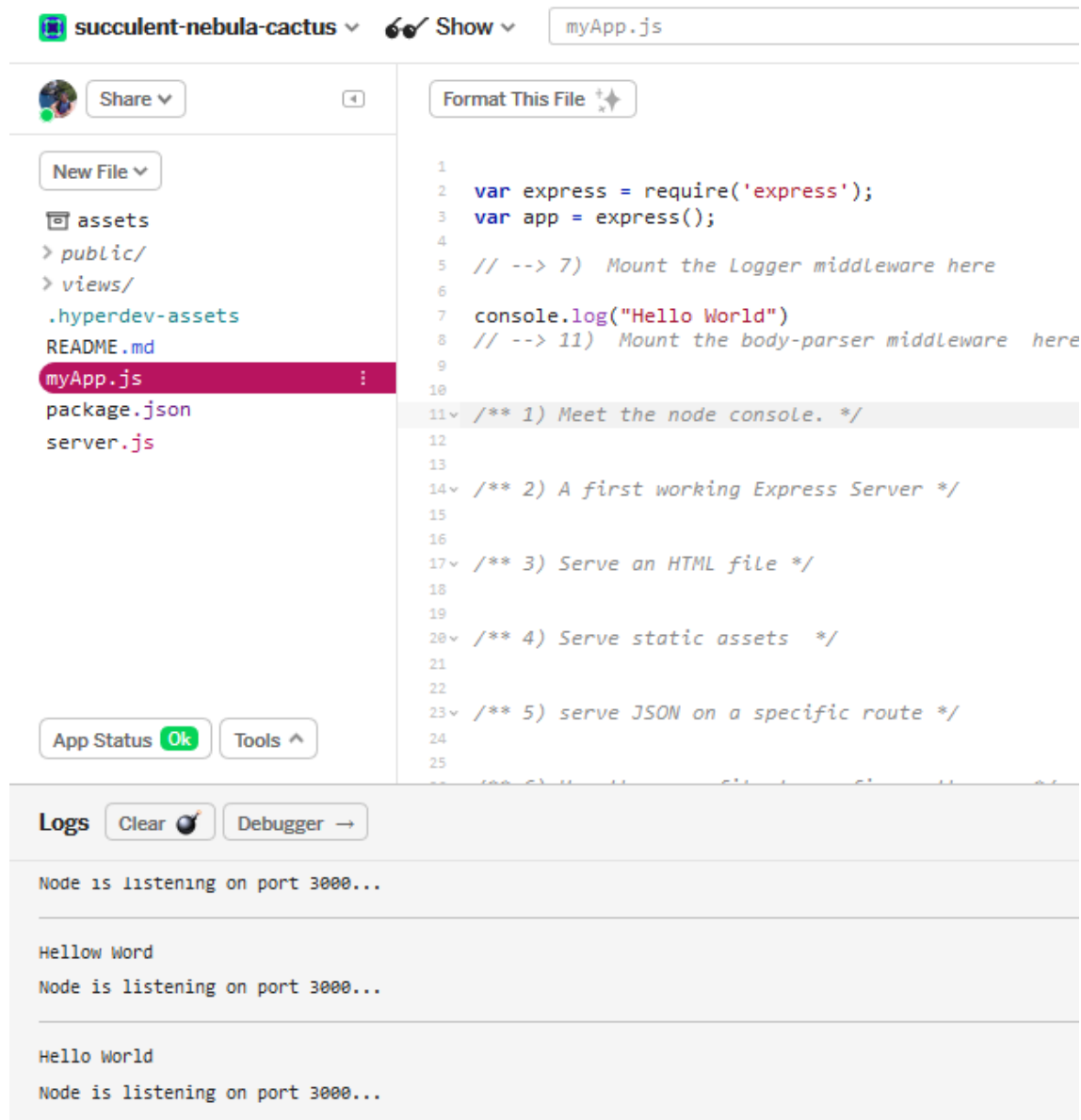
Basic Node and Express - Meet the Node console

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 the console output in a terminal. On Glitch you can open the logs in the lower part of the screen. You can toggle the log panel with the button 'Logs' (lower-left, inside the tools menu).

We recommend to keep the log panel open while working at these challenges. By reading the logs, you can be aware of the nature of errors that may occur.

If you have not already done so, please read the instructions in [the introduction](#) and start a new project on Glitch using [this link](#).

Modify the `myApp.js` file to log "Hello World" to the console.



The screenshot shows a Glitch project editor interface. At the top, the username 'succulent-nebula-cactus' and a 'Show' button are visible. The file 'myApp.js' is selected in the file explorer on the left. The main editor area displays the following code:

```
1
2 var express = require('express');
3 var app = express();
4
5 // --> 7) Mount the Logger middleware here
6
7 console.log("Hello World")
8 // --> 11) Mount the body-parser middleware here
9
10
11 /** 1) Meet the node console. */
12
13
14 /** 2) A first working Express Server */
15
16
17 /** 3) Serve an HTML file */
18
19
20 /** 4) Serve static assets */
21
22
23 /** 5) serve JSON on a specific route */
24
25
```

Below the code editor, the 'App Status' is shown as 'Ok' and the 'Tools' menu is expanded. The 'Logs' section at the bottom shows the following output:

```
Node is listening on port 3000...

Hellow Word
Node is listening on port 3000...

Hello World
Node is listening on port 3000...
```

Basic Node and Express - 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. You can see it at the bottom of the file. It is inside comments because, for testing reasons, we need the app to be running in the background. All the code that you may want to add goes between these two

fundamental parts. Glitch stores the port number in the environment variable `process.env.PORT`. Its value is 3000.

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.

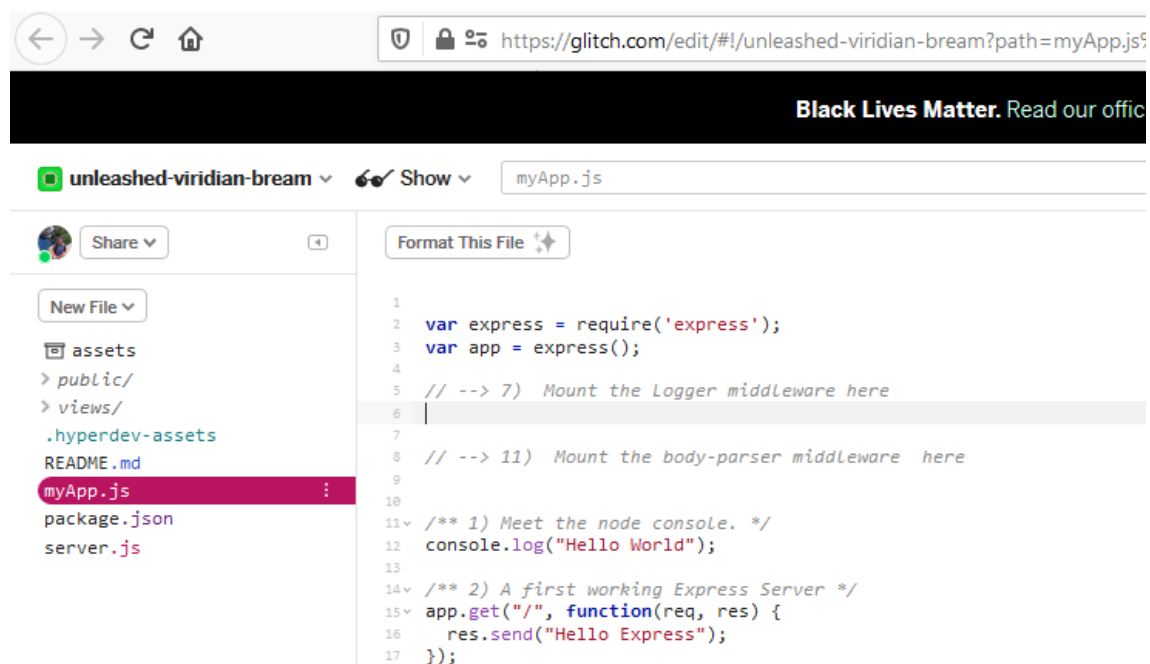
Handlers take the form `function(req, res) {...}`, where `req` is the request object, and `res` is the response object. For example, the handler

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

will serve the string 'Response String'.

Use the `app.get()` method to serve the string "Hello Express" to GET requests matching the `/` (root) path.

Note: Be sure that your code works by looking at the logs, then see the results in your browser by clicking the 'Show Live' button if you are using Glitch.



Basic Node and Express - Serve an HTML File

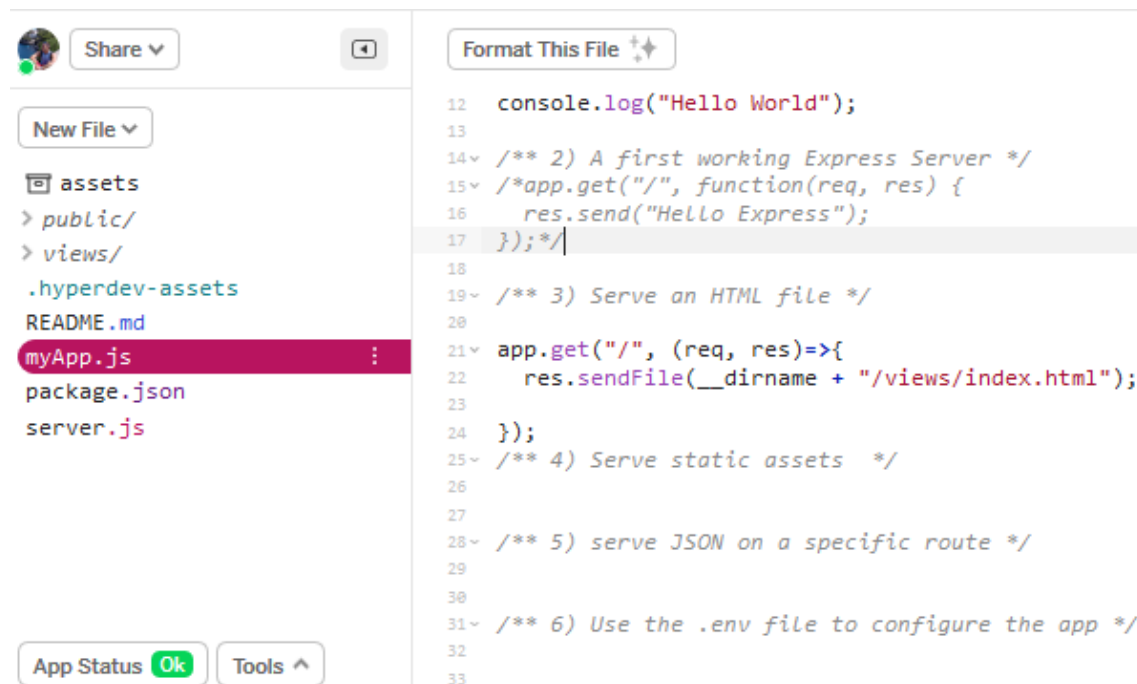
You can respond to requests with a file using the `res.sendFile(path)` method. You can put it inside the `app.get('/', ...)` route handler. Behind the scenes, this method will set the appropriate headers to instruct your browser on how to handle the file you want to send, according to its type. Then 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:

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

Send the `/views/index.html` file as a response to GET requests to the `/` path. If you view your live app, you should see a big HTML heading (and a form that we will use later...), with no style applied.

Note: You can edit the solution of the previous challenge or create a new one. If you create a new solution, keep in mind that Express evaluates routes from top to bottom, and executes the handler for the first match. You have to comment out the preceding solution, or the server will keep responding with a string.

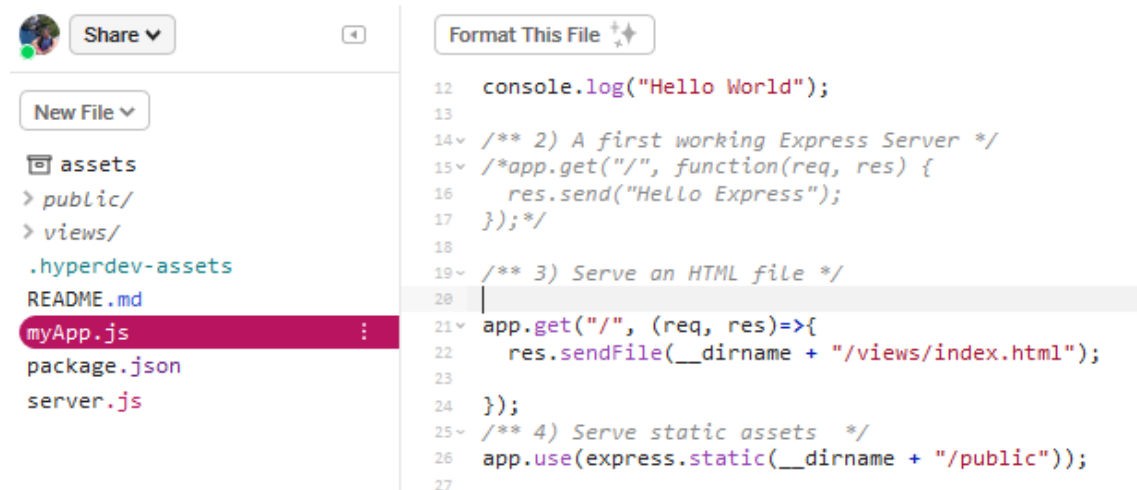


Basic Node and Express - Serve Static Assets

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.

Mount the `express.static()` middleware for all requests with `app.use()`. The absolute path to the assets folder is `__dirname + '/public'`.

Now your app should be able to serve a CSS stylesheet. From outside, the public folder will appear mounted to the root directory. Your front-page should look a little better now!



Basic Node and Express - 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.

Serve the object `{"message": "Hello json"}` as a response, in JSON format, to GET requests to the `/json` route. Then point your browser to `your-app-url/json`, you should see the message on the screen.

```
· /** 5) serve JSON on a specific route */
```

```
· app.get("/json", (req, res)=>{  
  res.json({"message": "Hello json"});});
```

Basic Node and Express - 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.

Let's add an environment variable as a configuration option.

Store the variable `MESSAGE_STYLE=uppercase` in the `.env` file. Then tell the GET `/json` route handler that you created in the last challenge to transform the response object's message to uppercase if `process.env.MESSAGE_STYLE` equals `uppercase`. The response object should become `{"message": "HELLO JSON"}`.

```
· /** 6) Use the .env file to configure the app */  
· process.env.MESSAGE_STYLE="uppercase";  
· app.get("/json", function(req, res) {  
  let message = 'Hello json';  
  if (process.env.MESSAGE_STYLE === 'uppercase') {  
    return res.json({"message": message.toUpperCase()})  
  }  
  return res.json({"message": message})  
})
```

Basic Node and Express - 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:

```
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(<middleware-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(<middleware-function>)`. Analogous methods exist for all the HTTP verbs (GET, DELETE, PUT, ...).

Build a simple logger. For every request, it should log to the console a string taking the following format: `method path - ip`. An example would look like this: `GET /json - ::ffff:127.0.0.1`. Note that there is a space between `method` and `path` and that the dash separating `path` and `ip` is surrounded by a space on both sides. You can get the request method (http verb), the relative route path, and the caller's ip from the request object using `req.method`, `req.path` and `req.ip`. Remember to call `next()` when you are done, or your server will be stuck forever. Be sure to have the 'Logs' opened, and see what happens when some request arrives.

Note: Express evaluates functions in the order they appear in the code. This is true for middleware too. If you want it to work for all the routes, it should be mounted before them.

```
// --> 7) Mount the Logger middleware here  
  
app.use(function(req, res, next){  
  console.log(req.method + " " + req.path + " - " + req.ip);  
  next();  
});
```

Basic Node and Express - 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:

```
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.

In the route `app.get('/now', ...)` chain a middleware function and the final handler. In the middleware function you should add the current time to the request object in the `req.time` key. You can use `new Date().toString()`. In the handler, respond with a JSON object, taking the structure `{time: req.time}`.

Note: The test will not pass if you don't chain the middleware. If you mount the function somewhere else, the test will fail, even if the output result is correct.

Solutions

▼ Solution 1 (Click to Show/Hide)

```
app.get(
  "/now",
  (req, res, next) => {
    req.time = new Date().toString();
    next();
  },
  (req, res) => {
    res.send({
      time: req.time
    });
  }
);
```

▼ Solution 2 (Click to Show/Hide)

You can also declare the middleware beforehand to use in multiple routes as shown below:

```
const middleware = (req, res, next) => {
  req.time = new Date().toString();
  next();
};

app.get("/now", middleware, (req, res) => {
  res.send({
    time: req.time
  });
});
```

Basic Node and Express - 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'}
```

Build an echo server, mounted at the route `GET /:word/echo`. Respond with a JSON object, taking the structure `{echo: word}`. You can find the word to be repeated at `req.params.word`. You can test your route from your browser's address bar, visiting some matching routes, e.g. `your-app-rootpath/freecodecamp/echo`.

```
app.post("/:param1/:param2", (req, res) => {
  // Access the corresponding key in the req.params
  // object as defined in the endpoint
  var param1 = req.params.parameter1;
  // OR use destructuring to get multiple paramters
  var { param1, param2 } = req.params;
  // Send the req.params object as a JSON Response
  res.json(req.params);
});
```

Solutions

▼ Solution 1 (Click to Show/Hide)

```
app.get("/:word/echo", (req, res) => {
  const { word } = req.params;
  res.json({
    echo: word
  });
});
```

Basic Node and Express - 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'}
```

Build an API endpoint, mounted at `GET /name`. Respond with a JSON document, taking the structure `{ name: 'firstname lastname'}`. The first and last name parameters should be encoded in a query string e.g. `?first=firstname&last=lastname`.

Note: In the following exercise you are going to receive data from a POST request, at the same `/name` route path. If you want, you can use the method `app.route(path).get(handler).post(handler)`. This syntax allows you to chain different verb handlers on the same path route. You can save a bit of typing, and have cleaner code.

kj

Basic Node and Express - 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. This is a part of the HTML request, also called payload. Since HTML is text-based, even if you don't see the data, it doesn't mean that it is secret. The raw content of an HTTP POST request is shown below:

```
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.

Install the `body-parser` module in your `package.json`. Then, require it at the top of the file. Store it in a variable named `bodyParser`. The middleware to handle urlencoded data is returned by `bodyParser.urlencoded({extended: false})`. Pass to `app.use()` the function returned by the previous method call. As usual, the middleware must be mounted before all the routes which need it.

Note: `extended=false` is a configuration option that tells the parser to use the classic encoding. When using it, values can be only strings or arrays. The extended version

Problem Explanation

The body-parser should already be added to your project if you used the provided boilerplate, but if not it should be there as:

```
"dependencies": {
  "body-parser": "^1.19.0",
  "express": "^4.17.1"
}
```

You can run `npm install body-parser` to add it as a dependency to your project instead of manually adding it to the `package.json` file.

This guide assumes you have imported the `body-parser` module into your file as `bodyParser`.

In order to import the same, you just need to add the following line at the top of your file:

```
var bodyParser = require("body-parser");
```

All you need to do for this challenge is pass the middleware to `app.use()`. Make sure it comes before the paths it needs to be used on. Remember that `body-parser` returns with `bodyParser.urlencoded({extended: false})`. Use the following as a template:

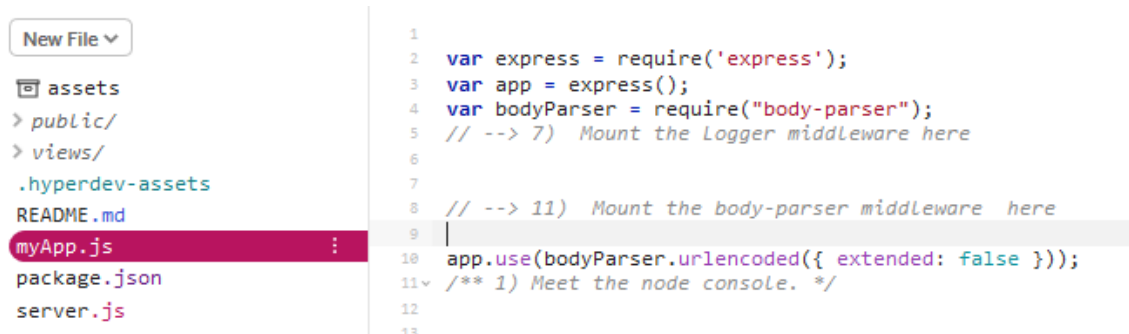
```
app.use(bodyParser.urlencoded({ extended: false }));
```

In order to parse JSON data sent in the POST request, use `bodyParser.json()` as the middleware as shown below:

```
app.use(bodyParser.json());
```

allows more data flexibility, but it is outmatched by JSON.





Basic Node and Express - 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.

Problem Explanation

Just like using `req.query` we can do `req.body` to get our data. This challenge is very similar to "Get Query Parameter Input from the Client."

In order to get data from a post request a general format is:

Hints

Hint 1

```
app.post(PATH, function(req, res) {  
  // Handle the data in the request  
});
```

Solutions

▼ Solution 1 (Click to Show/Hide)

```
app.post("/name", function(req, res) {  
  // Handle the data in the request  
  var string = req.body.first + " " + req.body.last;  
  res.json({ name: string });  
});
```

```
59 // OR you can destructure and rename the keys  
60 var { first: firstName, last: lastName } = req.query;  
61 // Use template literals to form a formatted string  
62 res.json({  
63   name: `${firstName} ${lastName}`  
64 });  
65 });  
66 /** 11) Get ready for POST Requests - the `body-parser` */  
67 // place it before all the routes !  
68  
69  
70 /** 12) Get data form POST */  
71 app.post("/name", function(req, res) {  
72   // Handle the data in the request  
73   var string = req.body.first + " " + req.body.last;  
74   res.json({ name: string });  
75 });  
76  
77  
78 // This would be part of the basic setup of an Express app  
79 // but to allow FCC to run tests, the server is already active  
80 /** app.listen(process.env.PORT || 3000 ); */  
81
```

Introduction to the MongoDB and Mongoose Challenges

MongoDB is a database that stores data records (documents) for use by an application. Mongo is a non-relational, "NoSQL" database. This means Mongo stores all associated data within one record, instead of storing it across many preset tables as in a SQL database. Some benefits of this storage model are:

- Scalability: by default, non-relational databases are split (or "shared") across many systems instead of only one. This makes it easier to improve performance at a lower cost.
- Flexibility: new datasets and properties can be added to a document without the need to make a new table for that data.
- Replication: copies of the database run in parallel so if one goes down, one of the copies becomes the new primary data source.

While there are many non-relational databases, Mongo's use of JSON as its document storage structure makes it a logical choice when learning backend JavaScript. Accessing documents and their properties is like accessing objects in JavaScript.

Mongoose.js is an npm module for Node.js that allows you to write objects for Mongo as you would in JavaScript. This can make it easier to construct documents for storage in Mongo.

Working on these challenges will involve you writing your code on Glitch on our starter project. After completing each challenge you can copy your public glitch url (to the homepage of your app) into the challenge 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 Glitch using [this link](#) or clone [this repository](#) on GitHub! If you use Glitch, remember to save the link to your project somewhere safe!

Use MongoDB Atlas to host a free mongodb instance for your projects

For the following challenges, we are going to start using MongoDB to store our data. To simplify the configuration, we are going to use MongoDB Atlas.

MongoDB Atlas is a MongoDB Database-as-a-Service platform, which basically means that they configure and host the database for you, making it so that the only responsibility you have is to populate your database with what matters: data! We are going to show you how to:

- Create a MongoDB Atlas account.
- Create a new cluster.
- Create a new user on the database.
- Whitelist your IP address.

- Connect to your cluster.

Create a MongoDB Atlas account

Let's start by [going to MongoDB Atlas](#).

Once you open the MongoDB Atlas page, you should sign up for a new account.

- Click the [Sign In](#) button in the top right corner to open the registration page.
- Click the [Register for a new account](#) link at the bottom of the sign in page.
- Fill the registration form with your information and press **continue**.
- You should now be logged into your new account and see a modal with a green **Build my first cluster** button, click on it.

Create a new cluster

- Go through the steps of building your first cluster by following the instructions they provide and clicking next after each step.
 - **Choose your cloud provider and region**, you can leave this as the default provided (typically AWS).
 - **Customize your cluster's specs**, you can also leave this as the default provided, M0 Sandbox (Shared RAM, 512 MB Storage) Encrypted.
 - **Give your cluster a name**, you can also leave this as the default provided, Cluster 0.
- Now click the green **Create Cluster** button at the bottom of the screen and verify the image captions they provide.
- You should now see the message, Your cluster is being created - New clusters take between 7-10 minutes to provision. Wait until the cluster is created before going to the next step.

Create a new user on the database

- You should be able to see the green **Get Started** button on the bottom left of your screen, you can click this button to see at which step of the process you are in, if you click on it now, you can see the next step is to **Create your first database user**, go ahead and click on that step.
 - Follow the instructions by clicking on the `Security` tab.
 - Click on the green **ADD NEW USER** button.
 - Enter a user name and password and then select **Read or write to any database** under user privileges, remember to store your username and password somewhere safe.
 - Click on the **ADD USER** green button in the bottom right of the modal.

Note: You can always upgrade your privileges to the **Admin** level, however, it is best practice to give permissions to a user on an as-needed basis for security reasons.

Whitelist your IP address

- If you now click on the green **Get Started** button in the bottom left of your screen, you should see the next step to take highlighted, **Whitelist your IP address**, click on it.
 - Follow the instructions by clicking on the `IP Whitelist` tab under the `Security` tab.
 - Click on the green **ADD IP ADDRESS** button.
 - In the modal, click the **ALLOW ACCESS FROM ANYWHERE** button and you should see `0.0.0.0/0` pre-filled for the whitelist entry field, click the green **Confirm** button.

Connect to your cluster

- Clicking on the green **Get Started** button in the bottom left of your screen should now show you the final step, **Connect to your cluster**, click on it.
 - Follow the instructions by clicking on the `Connect` button in the `Sandbox` section.
 - In the pop-up modal, click on **Connect Your Application**, a connection string will be displayed, you can copy that connection string by clicking on the `copy` button.
 - This will be the final URI that you will use to connect to your db, it will look something like this `mongodb+srv://<user>:<password>@<cluster#-dbname>.mongodb.net/test?retryWrites=true`, notice that the `user` and `cluster#-dbname` fields are already filled out for you, all you would need to replace is the `password` field with the one that you created in the previous step.
- That's it! You now have the URI you will add to your application to connect to your database. Keep this URI safe somewhere, so you can use it later!
- Feel free to create separate databases for different applications if they need a separate database. You just need to create a new project under your current MongoDB Atlas account, build a new cluster, add a new user, whitelist your IP addresses and finally connect to your cluster to obtain the new URI.

 <https://www.mongodb.com/cloud/atlas/register?v=1>



Get started free

No credit card required

How are you using MongoDB Atlas?

I'm learning MongoDB



Your Company (optional)

Your Work Email

rafasolis

First Name

RAFAEL

Last Name

SOLIS

Shared Clusters

For teams learning MongoDB or developing small applications.

- ✓ Highly available auto-healing cluster
- ✓ End-to-end encryption
- ✓ Role-based access control

Create a cluster


Starting at
FREE


Create a Starter Cluster


Welcome to MongoDB Atlas! We've recommended some of our most popular options, but feel free to customize your cluster to your needs. For more information, check our [documentation](#).

Cloud Provider & Region

AWS, N. Virginia (us-east-1) ▾









★ Recommended region ⓘ


NORTH AMERICA

EUROPE

ASIA

 N. Virginia (us-east-1) ★

 Ireland (eu-west-1) ★

 Singapore (ap-southeast-1) ★

FREE

Free forever! Your M0 cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

Back

Create Cluster

Cluster Tier

M0 Sandbox (Shared RAM, 512 MB Storage) >
 Encrypted

Additional Settings

MongoDB 4.2, No Backup >

Cluster Name

MyFirstCluster ▾

One time only: once your cluster is created, you won't be able to change its name.

MyFirstCluster

Cluster names can only contain ASCII letters, numbers, and hyphens.

FREE

Free forever! Your M0 cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

Back

Create Cluster

SPAIN

Access Manager

Support

Billing

All Clusters

RAFAEL ▾

Project 0

Atlas

Stitch

Charts

DATA STORAGE

Clusters

Triggers

Data Lake BETA

SECURITY

Database Access

Network Access

Advanced

Feature Requests

We are deploying your changes: 3 of 3 servers complete (current action: configuring MongoDB)

SPAIN > PROJECT 0

Clusters

Find a cluster...

SANDBOX

MyFirstCluster

Version 4.2.6

CONNECT METRICS COLLECTIONS ...

CLUSTER TIER

M0 Sandbox (General)

REGION

AWS / N. Virginia (us-east-1)

TYPE

Replica Set - 3 nodes

LINKED STITCH APP

None Linked

Your cluster is being created

New clusters take between 1-3 minutes to provision.

DATA STORAGE

Clusters

Triggers

Data Lake BETA

SECURITY

Database Access

Network Access

Advanced

Feature Requests

SPAIN > PROJECT 0 > CLUSTERS

MyFirstCluster

VERSION 4.2.6

REGION N. Virginia (us-east-1)

CLUSTER TIER M0 Sandbox (General)

OverviewReal TimeMetricsCollectionsProfilerPerformance AdvisorCommand Line Tools

SANDBOXNODESREPLICA SETCONNECTCONFIGURATION...

REGION N. Virginia (us-east-1)

myfirstcluster-shard-00-00...SECONDARY

myfirstcluster-shard-00-01...PRIMARY

myfirstcluster-shard-00-02...SECONDARY

Operations R: 0 W: 0100.0/s

Last 6 Hours

Logical Size 0.0 B512.0 MB max

Last 6 Hours

Connections 0500 max

Last 6 Hours

Enhance Your Experience

For dedicated throughput, richer metrics and enterprise security options, upgrade your cluster now!

Upgrade

SPAIN

Access ManagerSupportBilling

All Clusters

Project 0

AtlasStitchCharts

DATA STORAGE

Clusters

Triggers

Data Lake BETA

SECURITY

Database Access

Network Access

Advanced

Feature Requests

SPAIN > PROJECT 0

Database Access

Database UsersCustom Roles

Create a Database User

Set up database users, permissions, and authentication credentials in order to connect to your clusters.

Add New Database User

Feature Requests

Authentication Method

Password

Certificate
(M10 and up)

MongoDB uses [SCRAM](#) as its default authentication method.

Password Authentication

.....

SHOW

🔑 Autogenerate Secure Password

📋 Copy

Database User Privileges

Select a [built-in role](#) or [privileges](#) for this user.

Read and write to any database

⬆

Temporary User

This user is temporary and will be deleted after your specified duration of 6 hours, 1 day, or 1 week.

OFF

DATA STORAGE

Clusters

Triggers

Data Lake BETA

SECURITY

Database Access

Network Access

Advanced

We are deploying your changes (current action: configuring MongoDB)

SPAIN > PROJECT 0

Database Access

Database Users

Custom Roles

User Name ↕	Authentication Method ▲	MongoDB Roles
🔑 rZYT71ZYU2QMttii	SCRAM	readWriteAnyDatabase@admin

42

SPAIN

Access ManagerSupportBilling

Project 0

AtlasStitchCharts

DATA STORAGE

ClustersTriggersData Lake BETA

SECURITY

Database AccessNetwork AccessAdvanced

SPAIN > PROJECT 0

Network Access

IP WhitelistPeeringPrivate Endpoint

Whitelist an IP address

Configure which IP addresses can access your cluster.

Add IP Address

Learn more

Feature Requests

DATA STORAGE

ClustersTriggersData Lake BETA

SECURITY

Database AccessNetwork AccessAdvanced

We are deploying your changes (current action: configuring MongoDB)

SPAIN > PROJECT 0

Network Access

IP WhitelistPeeringPrivate Endpoint

+ ADD IP ADDRESS

You will only be able to connect to your cluster from the following list of IP Addresses:

IP Address	Comment	Status	Actions
0.0.0.0/0 (includes your current IP address)	pre-filled for the whitelist entry field	Pending	<button>Edit</button> <button>Delete</button>

×

Connect to MyFirstCluster


✓ Setup connection security

Choose a connection method

Connect


Choose a connection method [View documentation](#)

Get your pre-formatted connection string by selecting your tool below.




Connect with the mongo shell
Interact with your cluster using MongoDB's interactive Javascript interface

>



Connect your application
Connect your application to your cluster using MongoDB's native drivers

>



Connect using MongoDB Compass
Explore, modify, and visualize your data with MongoDB's GUI

>

Go Back

Close

44



Connect to MyFirstCluster

✓ Setup connection security
✓ Choose a connection method
Connect

1 Select your driver and version

DRIVER	VERSION
Node.js	3.6 or later

2 Add your connection string into your application code

☒ Include full driver code example

```
const MongoClient = require('mongodb').MongoClient;
const uri = "mongodb+srv://rZYT71ZYU2QMttii:<password>@myfirstclu
const client = new MongoClient(uri, { useNewUrlParser: true });
client.connect(err => {
  const collection = client.db("test").collection("devices");
  // perform actions on the collection object
  client.close();
});
```

Copy

Replace **<password>** with the password for the **rZYT71ZYU2QMttii** user. Replace **<dbname>** with the name of the database that connections will use by default. Ensure any option params are [URL encoded](#).

Hints

Hint 1

Timeout error

If the tests are timing out, check the `package.json` file. Ensure the final dependency does not end in a `,`.

For example, this will result in a timeout error:

```
"dependencies": {  
  "express": "^4.12.4",  
  "body-parser": "^1.15.2",  
},
```

Hint 2

add MONGO_URI to .env

- Insert a line that looks similar to:

```
MONGO_URI='mongodb+srv://<username>:<password>@<clustername>-vlas9.m
```

`<username>` and `<clustername>` will be automatically generated by MongoDB.

- Replace `<password>` with your password. There should be no `<>` characters (unless those are in your password).

Still having issues? Check the below hints:

- Remove spaces before and after `=`.
CORRECT:

```
MONGO_URI='mongodb...'
```

INCORRECT:

```
MONGO_URI = 'mongodb...'
```

- Do you have symbols or special characters in your password, e.g. `$&(@`? If so, you will need to translate these into unicode. MongoDB has instructions on how to do this. I would suggest changing your password to be letters and numbers only for simplicity.

Composición URI

mongodb+srv://<username>:<password>@myfirstcluster-
u48yr.mongodb.net/<dbname>?retryWrites=true&w=majority

MONGO_URI= mongodb+srv://useradmin:passuser123@myfirstcluster-
u48yr.mongodb.net/myfirstdb?retryWrites=true&w=majority

The screenshot displays a web-based development environment with two main panels. The top panel, titled "Environment Variables", shows a list of variables: GLITCH_DEBUGGER (true), MONGO_URI (mongodb+srv://useradmin:passu...), and PORT (1988). Each variable has a "copy" button and a close "x" icon. Below the list is an "Add a Variable" button. The bottom panel shows a file explorer on the left with files like .env, .hyperdev-assets, README.md, myApp.js, package.json, and server.js. The main editor area displays the content of package.json, which includes project metadata and dependencies for express, body-parser, mongodb, and mongoose. The mongoose dependency is highlighted in blue. Below the package.json, there is a snippet of JavaScript code for setting up mongoose with the MONGO_URI from the environment variables.

```
{
  "name": "fcc-mongo-mongoose-challenges",
  "version": "0.0.1",
  "description": "A boilerplate project",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.12.4",
    "body-parser": "^1.15.2",
    "mongodb": "^3.5.8",
    "mongoose": "^5.9.17"
  },
  "engines": {
    "node": "4.4.5"
  }
}
```

```
/** # MONGOOSE SETUP #
 * ===== */
const MongoClient = require('mongodb').MongoClient;
process.env.GLITCH_DEBUGGER=true;
/** 1) Install & Set up mongoose */
const mongoose = require('mongoose');
mongoose.connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true });
// Add mongodb and mongoose to the project's package.json. Then require
// mongoose. Store your Mongo Atlas database URI in the private .env file
// as MONGO_URI. Connect to the database using the following syntax:
mongoose.connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true });
```

MongoDB and Mongoose - 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.

Glitch 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!

```
/* Example */

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

▼ Solution 1 (Click to Show/Hide)

There are 3 things to do in this challenge. You can click each item to see the code.

Assign Mongoose Schema to a variable

This is not necessary but will make your code easier to read.

```
const Schema = mongoose.Schema;
```

See the [Mongoose docs](#) ⁸⁸ first where is a lot of useful stuff.

When you are building schema you can use either of three options for name validation

```
name: String
name: {type: String}
name: {type: String, required: true} //preferred
```

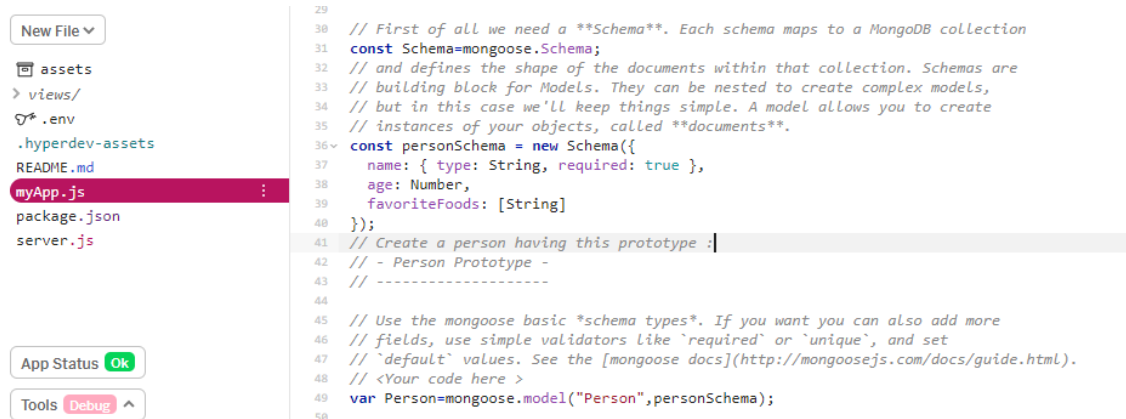
Create Person schema.

```
const personSchema = new Schema({
  name: { type: String, required: true },
  age: Number,
  favoriteFoods: [String]
});
```

Note: If you choose to skip the first step, you have to use `mongoose.Schema` instead of `Schema`.

Create Person model from the schema.

```
const Person = mongoose.model("Person", personSchema);
```

MongoDB and Mongoose - Create and Save a Record of a Model

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

Create a document instance using the `Person` constructor you built before. Pass to the constructor an object having the fields `name`, `age`, and `favoriteFoods`. Their types must conform to the ones in the `Person` Schema. Then call the method `document.save()` on the returned document instance. Pass to it a callback using the Node convention. This is a common pattern, all the following CRUD methods take a callback function like this as the last argument.

```
/* Example */

// ...
person.save(function(err, data) {
  // ...do your stuff here...
});
```

Hint 1

You need to do the following:

1. Create a model of a person, using the schema from exercise 2
2. Create a new person, including their attributes
3. Save the new person you created
4. Put your new person inside the `createAndSavePerson` function

```
/** 1) Install & Set up mongoose */

const mongoose = require('mongoose');
mongoose.connect(process.env.MONGO_URI);

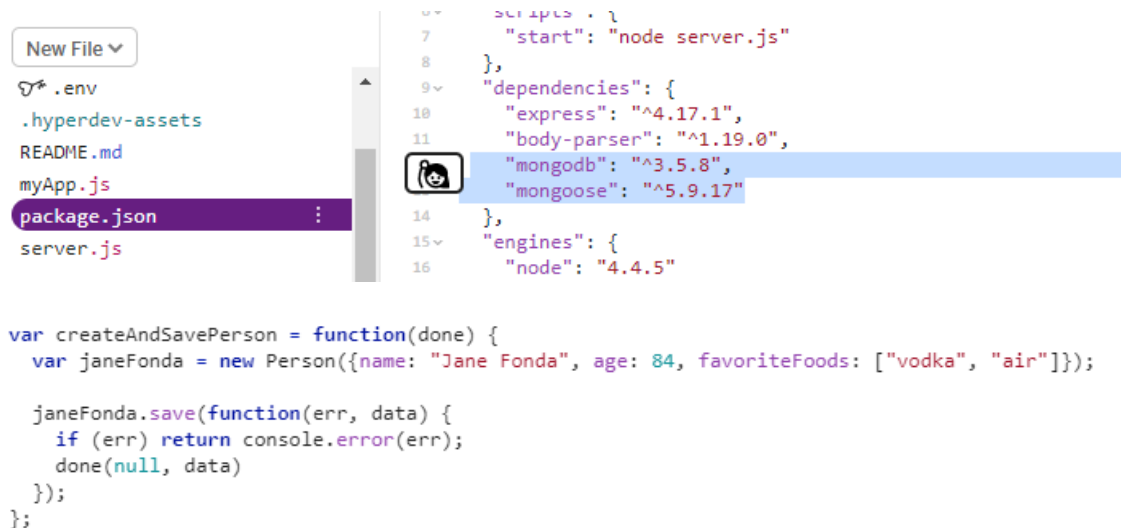
/** 2) Create a 'Person' Model */
var personSchema = new mongoose.Schema({
  name: String,
  age: Number,
  favoriteFoods: [String]
});
```

```
/** 3) Create and Save a Person */
var Person = mongoose.model('Person', personSchema);

var createAndSavePerson = function(done) {
  var janeFonda = new Person({name: "Jane Fonda", age: 84, favoriteFoods: ["vodka", "air"]});

  janeFonda.save(function(err, data) {
    if (err) return console.error(err);
    done(null, data)
  });
};
```

```
on({name: "Jane Fonda", age: 84, favoriteFoods: ["vodka", "air"]}]);
```



MongoDB and Mongoose - 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.

Create many people with `Model.create()`, using the function argument `arrayOfPeople`.

Hint 1

Create an array of objects. Each object has a name, age and an array of favorite foods. Use the variable name `arrayOfPeople`.

Hint 2

Create your many records (ie. create your people) inside the callback for `createManyPeople`.

Hint 3

Use `Model.create()` to create many records. You should replace `Model` with the name of the model you defined in the previous section. Most likely you called your model `Person`.

Hint 4

The `Model.create` function requires a callback, similar to the `person.save` function you used in the previous section.

```
/** 4) Create many People with `Model.create()` */
var arrayOfPeople = [
  {name: "Frankie", age: 74, favoriteFoods: ["Del Taco"]},
  {name: "Sol", age: 76, favoriteFoods: ["roast chicken"]},
  {name: "Robert", age: 78, favoriteFoods: ["wine"]}
];

var createManyPeople = function(arrayOfPeople, done) {
  Person.create(arrayOfPeople, function (err, people) {
    if (err) return console.log(err);
    done(null, people);
  });
};
```

```
var arrayOfPeople=[
  {name:"Peter",age:30,favoriteFoods:['Ice-cream','milk']},
  {name:"Ann",age:60,favoriteFoods:['limonade','wine']},
  {name:"John",age:45,favoriteFoods:['melon']}
];

var createManyPeople = function(arrayOfPeople, done) {

  Person.create(arrayOfPeople,function(err,people){
    if(err)return console.log(err);
    done(null,people);
  });
};
```

MongoDB and Mongoose - 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.

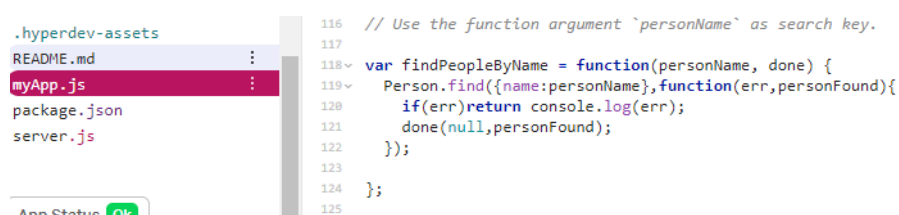
Hint 1

Replace `Model` with the name of your model from the previous sections. Most likely this is `Person`.

Hint 2

Use `{name: personName}` for the first argument in `Model.find()`.

```
/** 5) Use `Model.find()` */
var findPeopleByName = function(personName, done) {
  Person.find({name: personName}, function (err, personFound) {
    if (err) return console.log(err);
    done(null, personFound);
  });
};
```



The screenshot shows a code editor with a file explorer on the left. The file explorer lists: `.hyperdev-assets`, `README.md`, `myApp.js` (selected), `package.json`, and `server.js`. Below the file explorer is a status bar showing 'Ann Static' and a green 'OK' button. The code editor displays the following code:

```
116 // Use the function argument `personName` as search key.
117
118 var findPeopleByName = function(personName, done) {
119   Person.find({name:personName},function(err,personFound){
120     if(err)return console.log(err);
121     done(null,personFound);
122   });
123
124 };
125
```

MongoDB and Mongoose - 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.

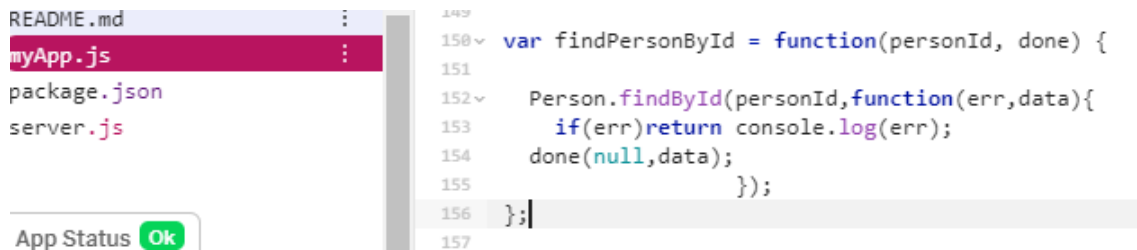
```
// ...
var findOneByFood = function(food, done) {
  Person.findOne({favoriteFoods: food}, function (err, data) {
    if (err) return console.log(err);
    done(null, data);
  });
};
```

MongoDB and Mongoose - 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.

Find the (only!!) person having a given `_id`, using `Model.findById()` -> `Person`. Use the function argument `personId` as the search key.

```
/** 7) Use `Model.findById()` */
var findPersonById = function(personId, done) {
  Person.findById(personId, function (err, data) {
    if (err) return console.log(err);
    done(null, data);
  });
};
```



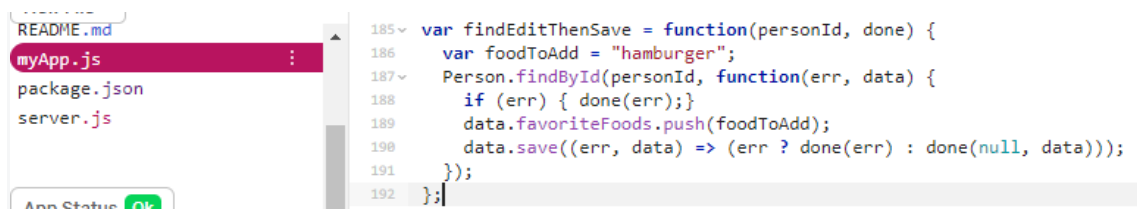
```
150~ var findPersonById = function(personId, done) {
151
152~   Person.findById(personId, function(err, data){
153     if(err) return console.log(err);
154     done(null, data);
155   });
156 };|
157
```

MongoDB and Mongoose - 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.

Find a person by `_id` (use any of the above methods) with the parameter `personId` as search key. Add "hamburger" to the list of the person's `favoriteFoods` (you can use `Array.push()`). Then - inside the find callback - `save()` the updated `Person`.

Note: This may be tricky, if in your Schema, you declared `favoriteFoods` as an Array, without specifying the type (i.e. `[String]`). In that case, `favoriteFoods` defaults to Mixed type, and you have to manually mark it as edited using `document.markModified('edited-field')`. See [Mongoose documentation](#)



```
185~ var findEditThenSave = function(personId, done) {
186   var foodToAdd = "hamburger";
187~   Person.findById(personId, function(err, data) {
188     if (err) { done(err); }
189     data.favoriteFoods.push(foodToAdd);
190     data.save((err, data) => (err ? done(err) : done(null, data)));
191   });
192 };|
```

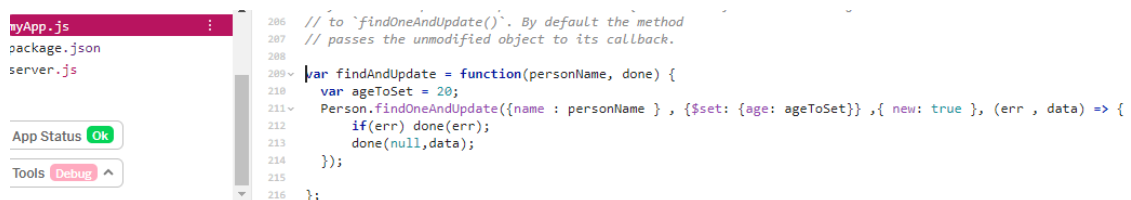
MongoDB and Mongoose - Perform New

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.

Find a person by `Name` and set the person's age to 20. Use the function parameter `personName` as search key.

Note: You should return the updated document. To do that you need to pass the options document `{ new: true }` as the 3rd argument to `findOneAndUpdate()`. By default these methods return the unmodified object.



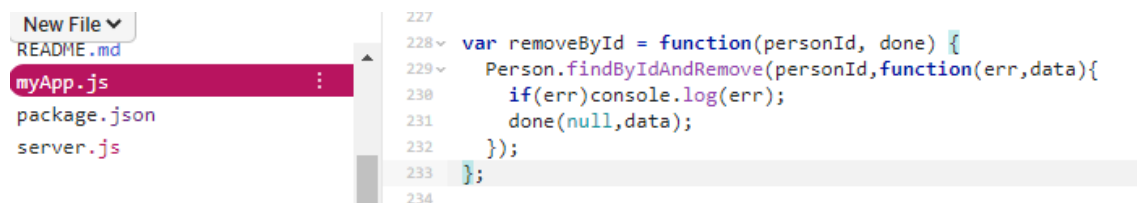
```

206 // to `findOneAndUpdate()`. By default the method
207 // passes the unmodified object to its callback.
208
209 var findAndUpdate = function(personName, done) {
210   var ageToSet = 20;
211   Person.findOneAndUpdate({name: personName}, {$set: {age: ageToSet}}, {new: true}, (err, data) => {
212     if(err) done(err);
213     done(null, data);
214   });
215
216 };
  
```

The screenshot shows a code editor with a file explorer on the left containing `myApp.js`, `package.json`, and `server.js`. The `myApp.js` file is selected, and the code shown is a function `findAndUpdate` that uses `Person.findOneAndUpdate` to update a person's age to 20 based on their name. The function returns the updated document. The editor also shows an 'App Status' indicator as 'Ok' and a 'Tools' dropdown set to 'Debug'.

MongoDB and Mongoose - 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.



```

227
228 var removeById = function(personId, done) {
229   Person.findByIdAndRemove(personId, function(err, data){
230     if(err) console.log(err);
231     done(null, data);
232   });
233 };
234
  
```

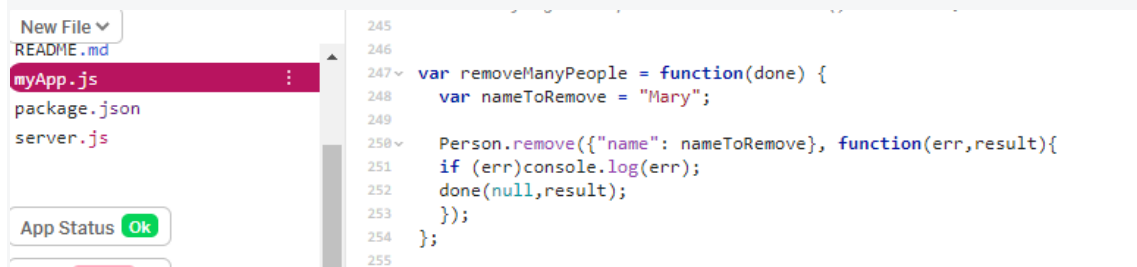
The screenshot shows a code editor with a file explorer on the left containing `myApp.js`, `package.json`, and `server.js`. The `myApp.js` file is selected, and the code shown is a function `removeById` that uses `Person.findByIdAndRemove` to delete a person by their ID. The function passes the removed document to the `done` callback. The editor also shows a 'New File' button and a 'README.md' file in the file explorer.

MongoDB and Mongoose - Delete Many Documents with `model.remove()`

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

Delete all the people whose name is "Mary", using `Model.remove()`. Pass it to a query document with the `name` field set, and of course a callback.

Note: The `Model.remove()` doesn't return the deleted document, but a JSON object containing the outcome of the operation, and the number of items affected. Don't forget to pass it to the `done()` callback, since we use it in tests.



```
245
246
247 var removeManyPeople = function(done) {
248   var nameToRemove = "Mary";
249
250   Person.remove({"name": nameToRemove}, function(err,result){
251     if (err)console.log(err);
252     done(null,result);
253   });
254 };
255
```

MongoDB and Mongoose - 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.

Find people who like `burrito`. Sort them by name, limit the results to two documents, and hide their age.

Chain `.find()`, `.sort()`, `.limit()`, `.select()`, and then `.exec()`. Pass the `done(err, data)` callback to `exec()`.

Hint 1

To create but not execute a find query

```
Model.find({ name: "Leah" });
```

Hint 2

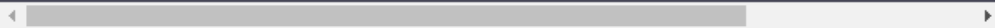
To store the find query into a variable for later use:

```
var findQuery = YourModel.find({ name: "Leah" });
```

Hint 3

To sort an array:

```
yourArray.sort({ age: 1 }); // Here: 1 for ascending order and -1
```



Hint 4

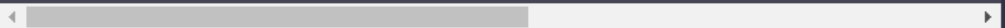
To limit an array's size:

```
yourArray.limit(5); // return array which has 5 items in it.
```

Hint 5

To hide certain property from the result:

```
yourArray.select({ name: 0, age: 1 }); // Here: 0 means false and thu
```



Hint 6

To execute this query, you can either:

1. Callback:

```
YourQuery.exec(function(err, docs) {  
  //do something here  
});
```

Hint 7

Or

2. Promise

```
YourQuery.exec.then(function(err, docs) {  
  //do something here  
});
```

Hint 8

Chain it all together:

```
Person.find({ age: 55 })  
  .sort({ name: -1 })  
  .limit(5)  
  .select({ favoriteFoods: 0 })  
  .exec(function(error, people) {  
    //do something here  
  });
```

assets

> views/

.env

.hyperdev-assets

README.md

myApp.js

package.json

server.js

```
274  
275~ var queryChain = function(done) {  
276   var foodToSearch = "burrito";  
277   const people = Person.find({favoriteFoods: foodToSearch})  
278     .sort({name: 1})  
279     .limit(2)  
280     .select({age: 0})  
281~   .exec(function(err, data){  
282~     if (err) {  
283       done(err);  
284     }  
285~     else {  
286       done(null, data);  
287     }  
288   })  
289 };  
290
```

PROYECT 1: API Project: Timestamp Microservice for FCC

API Project: Timestamp Microservice for FCC

User stories:

1. The API endpoint is GET [project_url]/api/timestamp/:date_string?
2. A date string is valid if can be successfully parsed by new Date(date_string) (JS) . Note that the unix timestamp needs to be an **integer** (not a string) specifying **milliseconds**. In our test we will use date strings compliant with ISO-8601 (e.g. "2016-11-20") because this will ensure an UTC timestamp.
3. If the date string is **empty** it should be equivalent to trigger new Date(), i.e. the service uses the current timestamp.
4. If the date string is **valid** the api returns a JSON having the structure {"unix": <date.getTime()>, "utc" : <date.toUTCString()> } e.g. {"unix": 1479663089000 , "utc": "Sun, 20 Nov 2016 17:31:29 GMT"}.
5. If the date string is **invalid** the api returns a JSON having the structure {"error" : "Invalid Date" }.

Example usage:

- <https://curse-arrow.glitch.me/api/timestamp/2015-12-25>
- <https://curse-arrow.glitch.me/api/timestamp/1451001600000>

Example output:

- {"unix":1451001600000, "utc":"Fri, 25 Dec 2015 00:00:00 GMT"}

Apis Microservices (freeCodeCamp)

```
index.html
.env
.gitconfig
.gitignore
package.json
server.js

App Status Ok
```

```
7     "start": "node server.js"
8   },
9   "dependencies": {
10     "express": "^4.17.1",
11     "cors": "^2.8.5",
12     "body-parser": "^1.19.0",
13     "dotenv": "^8.2.0"
14   }
15 }

3
4 // init project
5 var express = require('express');
6 var app = express();
7 require('dotenv').config({path: __dirname + './.env'});
8
9 // enable CORS (https://en.wikipedia.org/wiki/Cross-origin_resource_sharing)
10 // so that your API is remotely testable by FCC
11 var cors = require('cors');
12 app.use(cors({optionSuccessStatus: 200})); // some legacy browsers choke on 204
13
14 // http://expressjs.com/en/starter/static-files.html
15 app.use(express.static('public'));
16
17 // http://expressjs.com/en/starter/basic-routing.html
18 app.get("/", function (req, res) {
19   res.sendFile(__dirname + '/views/index.html');
20 });
21
22 // your first API endpoint...
23 app.get("/api/hello", function (req, res) {
24   res.json({greeting: 'hello API'});
25 });
26
27 // listen for requests :)
28 var listener = app.listen(process.env.PORT, function () {
29   console.log('Your app is listening on port ' + listener.address().port);
30 });
31
32 // Actual project code<
33
34 // seed req param whe we get the empty date
35 // app.get("/api/timestamp", function(req, res) {
36 //   let newDate = new Date();
37 //   res.redirect( "/api/timestamp/" + newDate.getFullYear() + "-" + (newDate.getUTCMonth() + 1) + "-" + newDate.getUTCDate() );
38 // });
39 // Changed to this beacuse not passing the test
40 app.get("/api/timestamp/", (req, res) => {
41   res.json({ unix: Date.now(), utc: Date() });
42 });
43
44 // Timestamp and Date endpoint to return formated date
45 app.get("/api/timestamp/:date_string", function(req, res, next){
46
47   let date;
48
49   if ( /\D/.test(req.params.date_string) ) {
50     date = new Date( req.params.date_string);
51   } else {
52     date = new Date( parseInt(req.params.date_string));
53   }
54
55   let utcDate = date.toUTCString();
56   let unixDate = date.getTime();
57
58   if (utcDate === "Invalid Date"){
59     res.json({ "error" : "Invalid Date" });
60   } else {
61     res.json({ "unix": unixDate, "utc": utcDate });
62   }
63 }
64 });
```

PROYECT 2. APIs and Microservices Projects - Request Header Parser Microservice

Build a full stack JavaScript app that is functionally similar to this: <https://dandelion-roar.glitch.me/>.

Apis Microservices (freeCodeCamp)

Working on this project will involve you writing your code on Glitch on our starter project. After completing this project you can copy your public glitch 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 Glitch using [this link](#) or clone [this repository](#) on GitHub! If you use Glitch, remember to save the link to your project somewhere safe!

The top screenshot shows the Glitch editor interface with the 'package.json' file selected in the file explorer on the left. The file content is as follows:

```
1 {
2   "name": "request_header",
3   "version": "0.0.1",
4   "description": "API project for freeCodeCamp",
5   "main": "server.js",
6   "scripts": {
7     "start": "node server.js"
8   },
9   "dependencies": {
10    "express": "^5.0.0-alpha.2",
11    "cors": "^2.8.1"
12  },
13  "engines": {
14    "node": "4.4.5"
15  },
16  "repository": {
17    "type": "git",
18    "url": "https://hyperdev.com/#!/project/welcome-project"
19  },
20  "keywords": [
21    "node",
22    "hyperdev",
23    "express"
24  ],
25  "license": "MIT"
26 }
```

The bottom screenshot shows the Glitch editor interface with the 'server.js' file selected in the file explorer on the left. The file content is as follows:

```
1 // enable CORS (https://en.wikipedia.org/wiki/cross-origin_resource_sharing)
2 // so that your API is remotely testable by FCC
3 var cors = require('cors');
4 app.use(cors({optionSuccessStatus: 200})); // some Legacy browsers choke on 204
5
6 // http://expressjs.com/en/starter/static-files.html
7 app.use(express.static('public'));
8
9 // http://expressjs.com/en/starter/basic-routing.html
10 app.get('/', function (req, res) {
11   res.sendFile(__dirname + '/views/index.html');
12 });
13
14 // your first API endpoint...
15 app.get("/api/hello", function (req, res) {
16   res.json({greeting: 'hello API'});
17 });
18
19 app.get('/', (req, res) => {
20   res.sendFile(__dirname + '/views/index.html');
21 });
22
23 app.get('/api/whoami', (req, res) => {
24   const ip =
25     req.headers['x-forwarded-for'].split(' ').pop() ||
26     req.connection.remoteAddress;
27   const languages = req.acceptsLanguages();
28   const userAgent = req.get('user-agent');
29
30   res.json({
31     ipaddress: ip.split(',')[0],
32     language: languages.join(','),
33     software: userAgent
34   });
35 });
36
37 // listen for requests :)
38 var listener = app.listen(process.env.PORT, function () {
39   console.log('Your app is listening on port ' + listener.address().port);
40 });
```

PROJECT 3. APIs and Microservices Projects - URL Shortener Microservice

Build a full stack JavaScript app that is functionally similar to this: <https://thread-paper.glitch.me/>.

Working on this project will involve you writing your code on Glitch on our starter project. After completing this project you can copy your public glitch 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 Glitch using [this link](#) or clone [this repository](#) on GitHub! If you use Glitch, remember to save the link to your project somewhere safe!

API Project: URL Shortener Microservice

User Story:

1. I can POST a URL to [project_url]/api/shorturl/new and I will receive a shortened URL in the JSON response.
Example: {"original_url": "www.google.com", "short_url": 1}
2. If I pass an invalid URL that doesn't follow the http(s)://www.example.com(/more/routes) format, the JSON response will contain an error like {"error": "invalid URL"}
HINT: to be sure that the submitted url points to a valid site you can use the function `dns.lookup(host, cb)` from the `dns` core module.
3. When I visit the shortened URL, it will redirect me to my original link.

Short URL Creation

example: POST [project_url]/api/shorturl/new - https://www.google.com

URL to be shortened

POST URL

Example Usage:

[\[this_project_url\]/api/shorturl/3](#)

Will Redirect to:

<https://www.freecodecamp.org/forum/>

by [freeCodeCamp](#)

Hints

Hint 1

Creating Short URL

- Connect to your database instance.

Note: It's important to check your Mongoose connection status before dive into the problem, just to check if everything is okay with your database configuration. This should help: `mongoose.connection.readyState`

- Receive a POST request containing an URL to be saved on Database.
- Check if it is a valid URL using `dns.lookup(url, callback)`
 - Remember you need to import the `dns` module with `required`.
- Generate some kind of identifier to save your original URL in database.
 - A SHA-1 hash could be used, or even the object ID when saving the element.
 - There is a bunch of samples over the internet how to generate some kind of identifier, try to explore it or create your own.
 - An example of how this should look like: `{ 'url': 'www.freecodecamp.org', 'hash': 'ef49fa8b4' }`

Hint 2

Retrieving Short URL

- Receive a GET request containing an identifier used to find a stored URL.
- Try to find one URL saved for this identifier
- Redirect user to URL.

Note: The `res.redirect(url)` function need that the given url, has a defined protocol (`http://`, `https://`), or it will just concatenate it as an extension of your current domain. eg: Good URL: `https://www.freecodecamp.org`, Bad URL: `www.freecodecamp.org`. Try it out.

- Remember to handle error situations with proper response like, ``res.json({ "error": "invalid URL" });``

1  share

Apis Microservices (freeCodeCamp)



```
1 {
2   "name": "shorturl",
3   "version": "0.0.2",
4   "description": "API project for freeCodeCamp",
5   "main": "server.js",
6   "scripts": {
7     "start": "node server.js"
8   },
9   "dependencies": {
10    "express": "^4.17.1",
11    "mongodb": "^3.3.3",
12    "mongoose": "^5.7.7",
13    "cors": "^2.8.5",
14    "body-parser": "^1.19.0"
15  },
16  "engines": {
17    "node": "10.17.0"
18  },
19  "repository": {
20    "type": "git",
21    "url": "https://hyperdev.com/#!/project/welcome-project"
22  },
23  "keywords": [
24    "node",
25    "hyperdev",
26    "express"
27  ],
28  "license": "MIT"
29 }
```



```
3 var express = require('express');
4 var mongo = require('mongodb');
5 var mongoose = require('mongoose');
6
7 var cors = require('cors');
8
9 var app = express();
10
11 // Basic Configuration
12 var port = process.env.PORT || 3000;
13
14 /** this project needs a db !! */
15 // mongoose.connect(process.env.DB_URI);
16
17 app.use(cors());
18
19 /** this project needs to parse POST bodies */
20 // you should mount the body-parser here
21
22 app.use('/public', express.static(process.cwd() + '/public'));
23
24 app.get('/', function(req, res){
25   res.sendFile(process.cwd() + '/views/index.html');
26 });
27
28
29 // your first API endpoint...
30 app.get("/api/hello", function (req, res) {
31   res.json({greeting: 'hello API'});
32 });
33
```


Apis Microservices (freeCodeCamp)

```
> public/  
> views/  
.gitconfig  
.hyperdev-assets  
README.md  
package.json  
server.js
```

```
34 var regex = new RegExp("^(http[s]?|\\|\\|\\|www\\.)?(ftp:\\|\\|\\|www\\.)?(www\\.){1}([0-9A-Z-a-z-\\.\\@:~%_+&=]+)(\\.[a-zA-Z]{2,3})+(\\/(-)*?\\(|?\\.|?)*$");  
35  
36 app.get('/:new:url(*)', function(req, res) {  
37   MongoClient.connect(mongoURL, function(err, db) {  
38     if (err) {  
39       console.log('Unable to connect to the mongoDB server. Error:', err)  
40     } else {  
41       console.log('Connection established to', mongoURL)  
42  
43       var collection = db.collection('links')  
44  
45       var Access = function(db, callback) {  
46         if (regex.test(req.params.url)) {  
47           collection.count().then(function(number) {  
48             var newElement = {  
49               original_url: req.params.url,  
50               short_url: "https://shrtm-me.herokuapp.com/" + (number + 1)  
51             }  
52             collection.insert([newElement])  
53             res.json({  
54               original_url: req.params.url,  
55               short_url: "https://shrtm-me.herokuapp.com/" + (number + 1)  
56             })  
57           })  
58         } else {  
59           res.json({  
60             'error': 'Please provide valid URL in order to shorten it'  
61           })  
62         }  
63       }  
64     }  
65   )  
66  
67   Access(db, function() {  
68     db.close()  
69   })  
70  
71  
72 app.get('/:shortid', function(req, res) {  
73   MongoClient.connect(mongoURL, function(err, db) {  
74     if (err) {  
75       console.log('Unable to connect to the mongoDB server. Error:', err)  
76     } else {  
77       var collection = db.collection('links')  
78  
79       var query = function(db, callback) {  
80         collection.findOne(  
81           "short_url": "https://shrtm-me.herokuapp.com/" + req.params.shortid  
82         ), {  
83           original_url: 1,  
84           _id: 0  
85         }, function(err, answer) {  
86           if (answer === null) {  
87             res.json({  
88               'error': "Provided URL is not found in our database"  
89             })  
90           } else {  
91             if (answer.original_url.split('').[0] == 'w') {  
92               res.redirect(301, 'http://' + answer.original_url)  
93             } else {  
94               res.redirect(301, answer.original_url)  
95             }  
96           }  
97         }  
98       })  
99  
100       query(db, function() {  
101         db.close()  
102       })  
103     }  
104   })  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736
```

PROYECT 4. APIs and Microservices Projects - Exercise Tracker

Build a full stack JavaScript app that is functionally similar to this: <https://nonstop-pond.glitch.me/>.

Working on this project will involve you writing your code on Glitch on our starter project. After completing this project you can copy your public glitch 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 Glitch using [this link](#) or clone [this repository](#) on GitHub! If you use Glitch, remember to save the link to your project somewhere safe!

Exercise tracker

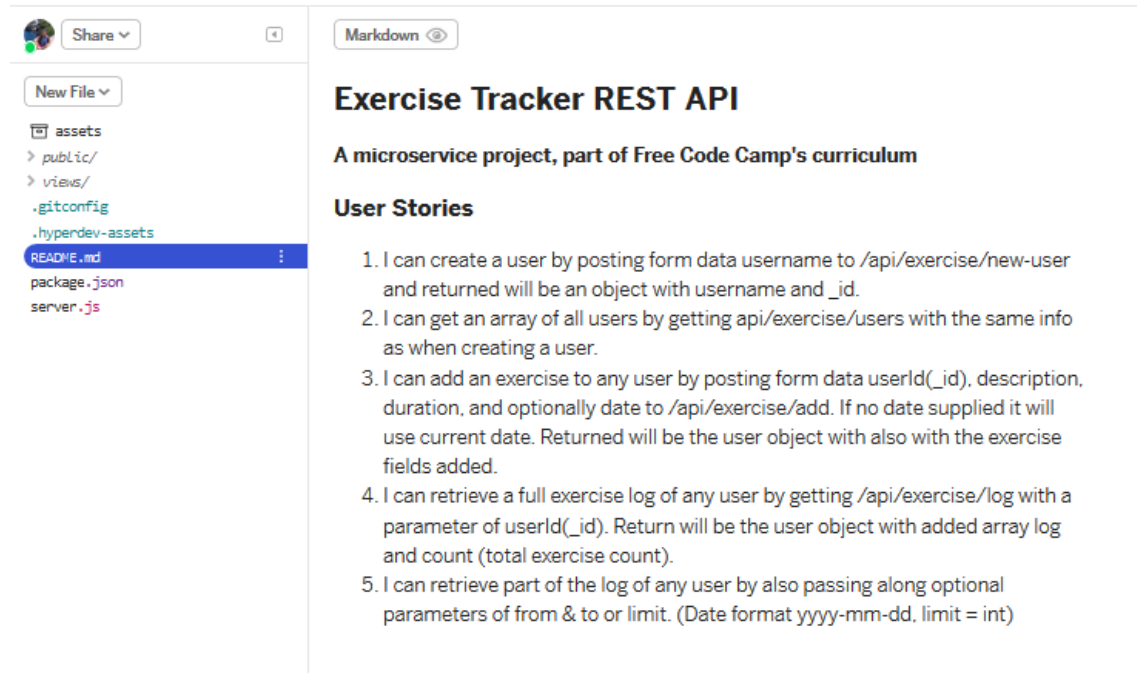
Create a New User

POST /api/exercise/new-user

Add exercises

POST /api/exercise/add

GET users's exercise log: GET /api/exercise/log?{userId} [&from] [&to] [&limit]
{ } = required, [] = optional
from, to = dates (yyyy-mm-dd); limit = number



Exercise Tracker REST API

A microservice project, part of Free Code Camp's curriculum

User Stories

1. I can create a user by posting form data username to `/api/exercise/new-user` and returned will be an object with username and `_id`.
2. I can get an array of all users by getting `api/exercise/users` with the same info as when creating a user.
3. I can add an exercise to any user by posting form data `userId(_id)`, description, duration, and optionally date to `/api/exercise/add`. If no date supplied it will use current date. Returned will be the user object with also with the exercise fields added.
4. I can retrieve a full exercise log of any user by getting `/api/exercise/log` with a parameter of `userId(_id)`. Return will be the user object with added array log and count (total exercise count).
5. I can retrieve part of the log of any user by also passing along optional parameters of `from` & `to` or `limit`. (Date format `yyyy-mm-dd`, `limit = int`)

<https://glitch.com/edit/#!/mighty-fortune-shock?path=server.js%3A107%3A40>

<https://glitch.com/edit/#!/hussey-fcc-microservices?path=middlewares%2Fvalidators.js%3A12%3A30>

PROJECT 5. APIs and Microservices Projects - File Metadata Microservice

Build a full stack JavaScript app that is functionally similar to this: <https://purple-paladin.glitch.me/>.

Working on this project will involve you writing your code on Glitch on our starter project. After completing this project you can copy your public glitch 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.

API Project: File Metadata Microservice for freeCodeCamp

User stories:

1. I can submit a form that includes a file upload.
2. The form file input field has the "name" attribute set to "upfile". We rely on this in testing.
3. When I submit something, I will receive the file name and size in bytes within the JSON response

Usage :

- Go to the main page, and upload a file using the provided form.

Hint:

- To handle the file uploading you should use the [multer](#) npm package.

API Project: File Metadata Microservice

User Stories

1. I can submit a form object that includes a file upload.
2. The from file input field has the "name" attribute set to "upfile". We rely on this in testing.
3. When I submit something, I will receive the file name, and size in bytes within the JSON response.

Usage

Please Upload a File ...

Examinar...

No se ha seleccionado ningún archivo.

Upload

Ver todos los proeyctos:

<https://glitch.com/edit/#!/hussey-fcc-microservices?path=server.js%3A323%3A4>

Apis Microservices (freeCodeCamp)

The image shows a code editor interface with two files open. The top editor displays the `package.json` file, and the bottom editor displays the `server.js` file. Both editors have a file explorer on the left showing the project structure.

package.json

```
1 {
2   "name": "file_metadata",
3   "version": "0.0.1",
4   "description": "API project for freeCodeCamp",
5   "main": "server.js",
6   "scripts": {
7     "start": "node server.js"
8   },
9   "dependencies": {
10    "express": "^5.0.0-alpha.2",
11    "cors": "^2.8.1"
12  },
13  "engines": {
14    "node": "4.4.5"
15  },
16  "repository": {
17    "type": "git",
18    "url": "https://hyperdev.com/#!/project/welcome-project"
19  },
20  "keywords": [
21    "node",
22    "hyperdev",
23    "express"
24  ],
25  "license": "MIT"
26 }
```

server.js

```
1 'use strict';
2
3 const express = require('express');
4 const cors = require('cors');
5 const multer = require('multer');
6 const upload = multer();
7 const app = express();
8
9 app.use(cors());
10 app.use(express.static('public'));
11
12 app.set('view engine', 'pug');
13
14 app.get('/', (req, res) => res.render('index'));
15
16 app.post('/api/fileanalyse', upload.single('upfile'), (req, res) => {
17   const { originalname, mimetype, size } = req.file;
18   return res.json({ name: originalname, type: mimetype, size: size })
19 });
20
21 const listener = app.listen(process.env.PORT || 3000, () => {
22   console.log(`Your app is listening on port ${listener.address().port}`);
23 });
```