Managing Packages with Npm

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

Description

The file package.json is the center of any Node.js project or npm package. It stores information about your project just like the <head>-section in a HTML document describes the content of a webpage. The package.json consists of a single JSON-object where information is stored in "key": value-pairs. There are only two required fields in a minimal package.json - name and version - but it's a good practice to provide additional information about your project that could be useful to future users or maintainers. The author-field If you go to the Glitch project that you set up previously and look at on the left side of your screen, you'll find the file tree where you can see an overview of the various files in your project. Under the file tree's back-end section, you'll find package.json - the file that we'll be improving in the next couple of challenges. One of the most common pieces of information in this file is the author-field that specifies who's the creator of a project. It can either be a string or an object with contact details. The object is recommended for bigger projects but in our case, a simple string like the following example will do. `"author": "Jane Doe",` Instructions Add your name to the author-field in the package.json of your Glitch project. Remember that you're writing JSON. All field-names must use double-quotes ("), e.g. "author" All fields must be separated with a comma (,)

Instructions

Challenge Seed

Solution

```
// solution required
```

2. Add a Description to Your package.json

Description

The next part of a good package.json 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, remember that 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 for your normal Node.js-projects 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. Let's add something similar to this: `"description": "A project that does something awesome",` Instructions Add a description to the package.json in your Glitch project. Remember to use double-quotes for field-names (") and commas (,) to separate fields.

Instructions

Challenge Seed

Solution

```
// solution required
```

3. Add Keywords to Your package.json

Description

The keywords-field is where you can describe your project using related keywords. Example `"keywords": [ "descriptive", "related", "words" ],` As you can see, this field is structured as an array of double-quoted strings. Instructions Add an array of suitable strings to the keywords-field in the package.json of your Glitch project. One of the keywords should be freecodecamp.

Instructions

Challenge Seed

Solution

```
// solution required
```

4. Add a License to Your package.json

Description

The license-field is where you inform users of your project what they are allowed to do with it. Some common licenses for open source projects include MIT and BSD. http://choosealicense.com is a great resource if you want to learn more about what license could fit your project. License information is not required. 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. Example `"license": "MIT",` Instructions Fill the license-field in the package.json of your Glitch project as you find suitable.

Instructions

Challenge Seed

Solution

```
// solution required
```

5. Add a Version to Your package.json

Description

The version is together with name one of the required fields in a package.json. This field describes the current version of your project. Example `"version": "1.2",` Instructions Add a version to the package.json in your Glitch project.

Instructions

Challenge Seed

Solution

```
// solution required
```

6. Expand Your Project with External Packages from npm

Description

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. In the dependencies-section, packages your project require are stored using the following format: `"dependencies": { "package-name": "version", "express": "4.14.0" }` Instructions Add version 2.14.0 of the package moment to the dependencies-field of your package.json Moment is a handy library for working with time and dates.

Instructions

Challenge Seed

Solution

```
// solution required
```

7. Manage npm Dependencies By Understanding Semantic Versioning

Description

Versions of the npm packages in the dependencies-section of your package.json 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 that projects who depend on the package can expect if they update. SemVer doesn't make sense in projects without public APIs - so unless your project is similar to the examples above, use another versioning format. So why do you need to understand SemVer? Knowing SemVer can be useful when you develop software that use 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. This is how Semantic Versioning works according to the official website: Given a version number MAJOR.MINOR.PATCH, increment the: MAJOR version when you make incompatible API changes, MINOR version when you add functionality in a backwards-compatible manner, and PATCH version 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. Example A semantic version number: 1.3.8 Instructions In the dependencies-section of your package.json, change the version of moment to match MAJOR version 2, MINOR version 10 and PATCH version 2

Instructions

Challenge Seed

Solution

```
// solution required
```

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

Description

In the last challenge, we 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 a npm dependency to get updated to the latest PATCH-version, you can prefix the dependency's version with the tilde-character (~). In package.json, our current rule for how npm may upgrade moment is to use a specific version only (2.10.2), but we want to allow the latest 2.10.x-version. Example `"some-package-name": "~1.3.8" allows updates to any 1.3.x version.` Instructions 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 that the version numbers themselves should not be changed.

Instructions

Challenge Seed

Solution

```
// solution required
```

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

Description

Similar to how the tilde (~) we learned about in the last challenge allow 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. At the moment, your current version of moment should be ~2.10.2 which allows npm to install to the latest 2.10.x-version. If we instead were to use the caret (^) as our version prefix, npm would instead be allowed to update to any 2.x.x-version. Example `"some-package-name": "^1.3.8" allows updates to any 1.x.x version.` Instructions Use the caret-character (^) to prefix the version of moment in your dependencies and allow npm to update it to any new MINOR release. Note that the version numbers themselves not should be changed.

Instructions

Challenge Seed

Solution

```
// solution required
```

10. Remove a Package from Your Dependencies

Description

Now you've tested a few ways you can manage dependencies of your project by using the package.json's dependencies-section. You've included external packages by adding them to the file and even told npm what types of versions you want by using special characters 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 from your dependencies. This same method applies to removing other fields in your package.json as well Instructions Remove the package moment from your dependencies. Make sure you have the right amount of commas after removing it.

Instructions

Challenge Seed

Solution

```
// solution required
```

Basic Node and Express

1. Meet the Node console

Description

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' (top-left, under the app name). To get started, just print the classic "Hello World" in the console. We recommend to keep the log panel open while working at these challenges. Reading the logs you can be aware of the nature of the errors that may occur.

Instructions

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

Challenge Seed

Solution

```
// solution required
```

2. Start a Working Express Server

Description

In the first two lines of the file myApp.js you can see how it's easy to create an Express app object. This object has several methods, and we 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 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. Be sure that your code works by looking at the logs, then see the results in your browser, clicking the button 'Show Live' in the Glitch UI.

Instructions

Challenge Seed

Solution

```
// solution required
```

3. Serve an HTML File

Description

We can respond with a file using the method `res.sendFile(path)`. 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. e.g. `absolutePath = __dirname + relativePath/file.ext`. The file to send is `/views/index.html`. Try to 'Show Live' your 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 the routes from top to bottom. It executes the handler for the first match. You have to comment out the preceding solution, or the server will keep responding with a string.

Instructions

Challenge Seed

Solution

```
// solution required
```

4. Serve Static Assets

Description

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 parameter is the absolute path of the folder containing the assets. If you don't know what a middleware is, don't worry. We'll discuss about it later in details. Basically middlewares 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 the requests. Mount the `express.static()` middleware for all the 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!

Instructions

Challenge Seed

Solution

```
// solution required
```

5. Serve JSON on a Specific Route

Description

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 the GET requests to the route `/json`. Then point your browser to your-app-url/json, you should see the message on the screen.

Instructions

Challenge Seed

Solution

```
// solution required
```

6. Use the .env File

Description

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"}`.

Instructions

Challenge Seed

Solution

```
// solution required
```

7. Implement a Root-Level Request Logger Middleware

Description

Before we introduced 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 informations to the request or response objects. They can also end the cycle sending the 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 is triggered calling the 3rd argument `next()`. More information in the express documentation. Look at the following example :

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

Let's suppose we 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 we are going to build a root-level middleware. As we have seen in challenge 4, to mount a middleware function at root level we can use the method `app.use(<mware-function>)`. In this case the function will be executed for all the requests, but you can also set more specific conditions. For example, if you want a function to be executed only for POST requests, you could use `app.post(<mware-function>)`. Analogous methods exist for all the http verbs (GET, DELETE, PUT, ...). Build a simple logger. For every request, it should log in the console a string taking the following format: `method path - ip`. An example would look like: `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 either side. 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... Hint: 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.

Instructions

Challenge Seed

Solution

```
// solution required
```

8. Chain Middleware to Create a Time Server

Description

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 = getTheUserSync(); // 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}`. Hint: 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.

Instructions

Challenge Seed

Solution

```
// solution required
```

9. Get Route Parameter Input from the Client

Description

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

Instructions

Challenge Seed

Solution

```
// solution required
```

10. Get Query Parameter Input from the Client

Description

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 cannot be in URLs, they 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`. TIP: In the following exercise we 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.

Instructions

Challenge Seed

Solution

```
// solution required
```

11. Use body-parser to Parse POST Requests

Description

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 the REST convention POST is used to send data to create new items in the database (a new user, or a new blog post). We don't have a database in this project, but we 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 they are 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 we can also use JSON to be able 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 we will use an urlencoded body. To parse the data coming from POST requests, you have to install a package: the body-parser. This package allows you to use a series of middleware, which can decode data in different formats. See the docs here. 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 url encoded data is returned by `bodyParser.urlencoded({extended: false})`. `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 allows more data flexibility, but it is outmatched by JSON. 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.

Instructions

Challenge Seed

Solution

```
// solution required
```

12. Get Data from POST Requests

Description

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.

Instructions

Challenge Seed

Solution

```
// solution required
```

MongoDB and Mongoose

1. Install and Set Up Mongoose

Description

Add mongodb and mongoose to the project's package.json. Then require mongoose. Store your mLab database URI in the private .env file as MONGO_URI. Connect to the database using mongoose.connect()

Instructions

Challenge Seed

Solution

```
// solution required
```

2. Create a Model

Description

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. Create a person having this prototype : `- Person Prototype - -- name : string [required] age : number favoriteFoods : array of strings (*)` Use the mongoose basic schema types. If you want you can also add more fields, use simple validators like required or unique, and set default values. See the mongoose docs. [C]RUD Part I - CREATE Note: 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); };`

Instructions

Challenge Seed

Solution

```
// solution required
```

3. Create and Save a Record of a Model

Description

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 be conformant 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... });`

Instructions

Challenge Seed

Solution

```
// solution required
```

4. Create Many Records with model.create()

Description

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.

Instructions

Challenge Seed

Solution

```
// solution required
```

5. Use model.find() to Search Your Database

Description

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.

Instructions

Challenge Seed

Solution

```
// solution required
```

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

Description

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. Find just one person which has a certain food in her favorites, using Model.findOne() -> Person. Use the function argument food as search key.

Instructions

Challenge Seed

Solution

```
// solution required
```

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

Description

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 search key.

Instructions

Challenge Seed

Solution

```
// solution required
```

8. Perform Classic Updates by Running Find, Edit, then Save

Description

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 binded 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 her favoriteFoods (you can use Array.push()). Then - inside the find callback - save() the updated Person. [*] Hint: This may be tricky if in your Schema you declared favoriteFoods as an Array, without specifying the type (i.e. [String]). In that casefavoriteFoods defaults to Mixed type, and you have to manually mark it as edited using document.markModified('edited-field'). (http://mongoosejs.com/docs/schematypes.html - #Mixed)

Instructions

Challenge Seed

Solution

```
// solution required
```

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

Description

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 her age to 20. Use the function parameter personName as search key. Hint: We want you to 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.

Instructions

Challenge Seed

Solution

```
// solution required
```

10. Delete One Document Using model.findByIdAndRemove

Description

Delete one person by her _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 cb. As usual, use the function argument personId as search key.

Instructions

Challenge Seed

Solution

```
// solution required
```

11. Delete Many Documents with model.remove()

Description

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: 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.

Instructions

Challenge Seed

Solution

```
// solution required
```

12. Chain Search Query Helpers to Narrow Search Results

Description

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(). 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().

Instructions

Challenge Seed

Solution

```
// solution required
```

Apis and Microservices Projects

1. Timestamp Microservice

Description

Build a full stack JavaScript app that is functionally similar to this: https://curse-arrow.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!

Instructions

Challenge Seed

Solution

```
// solution required
```

2. Request Header Parser Microservice

Description

Build a full stack JavaScript app that is functionally similar to this: https://dandelion-roar.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!

Instructions

Challenge Seed

Solution

```
// solution required
```

3. URL Shortener Microservice

Description

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!

Instructions

Challenge Seed

Solution

```
// solution required
```

4. Exercise Tracker

Description

Build a full stack JavaScript app that is functionally similar to this: https://fuschia-custard.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!

Instructions

Challenge Seed

Solution

```
// solution required
```

5. File Metadata Microservice

Description

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. 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!

Instructions

Challenge Seed

Solution

```
// solution required
```