# **1. Golden Rules**

1. **Always test your code!**
2. Use Google! **ESPECIALLY** if you don’t know or can’t remember something
   1. **DO NOT OVER-GOOGLE (do not pass page 2 of results)**
   2. **DO NOT USE CODE YOU DO NOT UNDERSTAND**
   3. **DO NOT COPY PASTE FROM GOOGLE! (read the code, understand it, and recreate it your own way)**
3. Make mistakes! **Seriously, don’t be scared of making mistakes**. The worst thing that can happen is you go back and fix it. Mistakes are a great way to learn (it’s called **trial and error** for a reason).
4. Good code is lazy and efficient! **Efficiency and laziness go hand in hand.**
5. Ask your classmates! There is no shame in not knowing something or asking for help. **Leverage the community!**
6. Hop in the queue! If you get stuck on something for 15 minutes or more, ask us.
7. Never overestimate how much time you have!
8. Don’t Panic!
9. **Don’t run terminal commands you don’t know!**
10. Static code is illegal code!

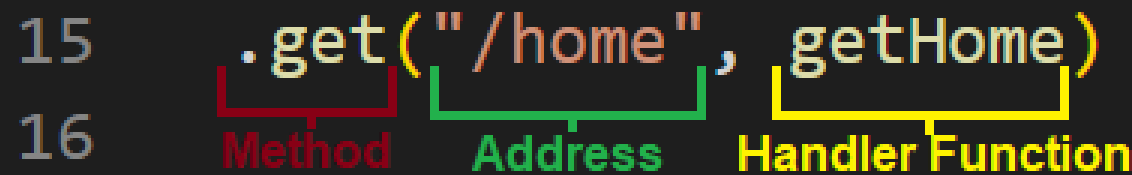
# **2. Node Concepts**

## **2.1 Endpoints**

### **2.1.1 What is an Endpoint?**

* An endpoint is just an address on a server that can be reached with a fetch (or other similar libraries / APIs).
* When an endpoint is called, it triggers a handler function that will:
  + Receive the request.
  + Parse the request.
  + Respond according to the request.
* An endpoint is made up of 3 things:
  + The [method](#_cv9szfu75x9q).
  + The [address](#_kk227t65h77z).
  + The [handler function](#_y0vlatd9e6u2).
* **AN ENDPOINT CAN ONLY SEND ONE RESPONSE!**

## **2.2. Anatomy of an Endpoint**

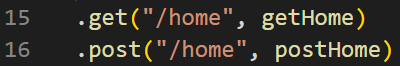
****

### **2.2.1 The Method**

The method is the “how” of an endpoint. It determines how an endpoint can be reached. There are 5 methods:

* GET: Retrieve data.
* POST: Create and / or submit data.
* PATCH: Update data (modify).
* PUT: Update data (rewrite).
* DELETE: Delete data.

The method also makes sure that each endpoint is unique when they share the same [address](#_kk227t65h77z). In the screenshot below, the endpoints are different from each other because one is a GET method and the other is a POST method.

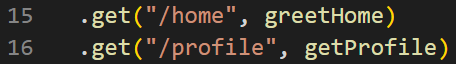


Only a fetch with method GET will reach the endpoint on line 15 and only a fetch with method POST will reach the endpoint on line 16.

### **2.2.2 The Address**

The address is there “where” of an endpoint. It determines at what server address an endpoint can be reached.

Similar to the [method](#_cv9szfu75x9q), the address also makes sure that each endpoint stays unique. When multiple endpoints share the same method, the address is what keeps them unique. In the screenshot below, the endpoints are different from each other because one goes to "/home", and the other goes to "/profile".



Only a fetch going to "/home" will reach the endpoint on line 15 and only a fetch going to "/profile" will reach the endpoint on line 16.

The address also has 2 extra properties:

* Url parameters
* Queries

#### **2.2.2.1 URL Parameters**

Also known as url params.

Similar to how we did it in [React](https://docs.google.com/document/d/1cNnWOKo7j7JppHCdMXsC0T5R_hW9a1J9qCPnXXYNEw4/edit#heading=h.y0m9xvpizwaw), the address on the server can also have url parameters. These are variables declared in the address string by placing a : in front of them.

Here you can see that the server expects to receive a profileId parameter in the address.



When a param is received, it is stored in the [req](#_n59csftqsg5p) object.

##### **⚠️ Important! ⚠️**

Order matters when it comes to the address!



In the screenshot above, the endpoint on line 20 will never trigger because the one on line 19 is too “loose”. A url param accepts anything as it’s value, it has no restrictions. If you were to try to go to "/profile", the word "profile" will be captured by the profileId param and used as its value.

There is a simple fix: invert the order!



Because code flows from top to bottom, the endpoint on line 19 will always be checked first, and because it can only be reached by putting the word "profile" in the address, it will be triggered when you try to go to "/profile". If you put anything else, it will be ignored by line 19 and caught by line 20 instead!

#### **2.2.2.2 Queries**

Unlike parameters, queries are not declared in the address string. They are sent from the frontend in the address but the server doesn’t need to do anything to receive them.

Given this fetch:



The endpoint will look like this:

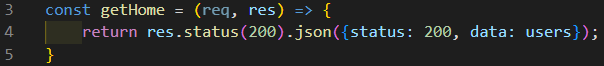


Node is smart enough and nice enough to handle the queries for us automatically. Like params, they are also stored in the [req](#_bxvfgepaa82r) object.

### **2.2.3 The Handler Function**

The handler function is the “what” of an endpoint. It determines what happens when the endpoint is triggered. The handler function is just a callback function, but it has a few unique twists:

* It can only accept 2 parameters:
  + The req object first.
  + The res object second.
* The order of the parameters must be respected.
* It has to send a response.
* The **return** statement is optional (but helpful).



#### **2.2.3.1 The req Object**

The req object is **ALWAYS** the **first parameter** in the [handler function](#_y0vlatd9e6u2). It is the entire request made by the frontend. Any data that was sent from the frontend will be stored in this object.

The main 3 containers are:

* [req.params](#_n59csftqsg5p).
* [req.query](#_bxvfgepaa82r).
* [req.body](#_8kd71lr1zm2t).

##### **2.2.3.1.1 req.params**

When a url param is received by the endpoint, it is stored inside req.params. It is an object that converts the param into a key, and the data passed in the address into the value.

If you fetch on "/rony", triggering this endpoint:



The req.params would look like this:

| { profileId: "rony" } |
| --- |

##### **2.2.3.1.2 req.query**

When a query is sent by a fetch, it is stored inside req.query. It is an object that converts the query into a key, and the data passed in the address into the value.

If you fetch on "/cars?color=blue", triggering this endpoint:



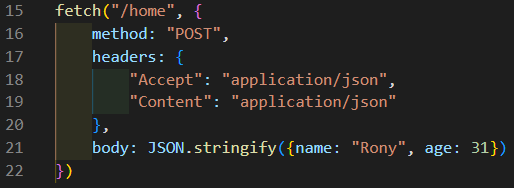
The req.query would look like this:

| { color: "blue" } |
| --- |

##### **2.2.3.1.3 req.body**

When a body is sent by a fetch, it is stored inside req.body. It is an object that holds all of the data passed along through the body. The data passed in the body must **ALWAYS** be a JSON string.

If you ran this fetch:



The req.body would look like this:

| { name: "Rony", age: 31 } |
| --- |

#### **2.2.3.2 The res Object**

The res object is **ALWAYS** the **second parameter** in the handler function. It is the entire response that will be made by the backend. Any data to be sent to the frontend will be sent through this object. By default, we’ll always send our data as JSON format.



##### **2.2.3.2.1 res.status()**

The res.status is literally the status of the response of the server. It lets the fetch request (also known as HTTP request) know what the status of the answer is; good (200), or bad (400 / 500).

Statuses ending in 00 are generic codes. For a full list of specific HTTP response status codes, check out this [link](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status).

##### **2.2.3.2.2 res.json({...})**

The res.json is the actual response of the server. It contains all of the data to be transmitted to the frontend. We use the res.json method because we always want to send our data in the JSON format.

Our convention also includes a status key and value inside the JSON object. The key:value pair inside the object is for the frontend developer to use if they need it. The data can be sent under any key, but typically we’ll call it data.

#### **2.2.3.3 The return Statement**

Technically, an endpoint’s handler function doesn’t need a **return** statement, it’s **entirely optional**. More often than not we tend to use it because we’re already used to it, and it can be helpful in preventing the endpoint from sending multiple responses. By putting a **return** in front of a res, we’re making sure that the handler function stops and sends only one response.

##### **⚠️ Important ⚠️**

**An endpoint can only send one response!**

The most common endpoint error you’ll run into is:

| Error: Can't set headers after they are sent to the client |
| --- |

This error only occurs when an endpoint sends more than one response to the frontend. To avoid this you should always make sure that:

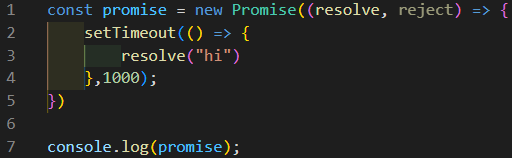
* Your responses are **NEVER** in a loop.
* If you have multiple possible responses, add conditions to make sure you only send one.
* Force the handler function to end its operations by putting a **return** statement.

## **2.3 Promises**

A promise is a piece of code that executes in parallel to JavaScript. Why in parallel? Because it’s asynchronous. JavaScript (and by extension, Node) is a synchronous language. What this means is that it executes the code one line at a time, and waits for the current line to finish before moving to the next line.

Asynchronous code runs all the lines at the same time. Nothing waits for anything to finish. It’s almost like a race to see who will finish first. When Asynchronous code is executed, JavaScript sets it aside in its own thread, and keeps reading the rest of the code synchronously as it normally would. When the asynchronous code finishes, it rejoins the main thread. Almost any asynchronous operation is using a promise under the hood.

Because promises are asynchronous, they cannot be handled normally. You have to set up some code to run only **AFTER** the promise has been fulfilled (finished). If you ran this code:

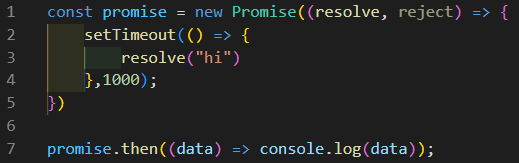


You would get:

| Promise { <pending> } |
| --- |

This happens because the promise is still running, it hasn’t had time to finish doing what it needs to do; in this case it needs to wait 1 second before resolving "hi".

The solution?



.then() is a promise-based method, but it waits for the code it’s attached to to return something. It receives the returned data into the parameter of its arrow function, and it can do something with it. In this case it will console.log:

| "hi" |
| --- |

### **2.3.1 resolve()**

The resolve statement of a promise is the equivalent of the **return** of a **function**. When a promise is fulfilled without problems, it resolves. It will return the data with a success flag, AKA resolve.

The data can be tested / verified in order to determine whether it should be resolved or not.

### **2.3.2 reject()**

The reject statement of a promise is the equivalent of **throw** **new** Error("error"). When the data is tested / verified in the promise, and it doesn’t pass the tests, the promise returns the data with an error flag, AKA reject.

### **2.3.3 .then()**

As mentioned above, the .then() is a promise-based method. It receives the data from the code it’s attached to and does something with it.

A unique feature of .then() is that it will also mimic and preserve the success / error flag of the returned data. If it’s attached to a promise that resolves, the .then() will receive the resolve flag, do something with the data, and return it with a resolve flag also. It does the same thing with the reject flag.

### **2.3.4 .catch()**

Just like .then(), .catch() is a promise-based method. The difference is that .catch() exclusively handles the reject of a promise. If a reject is detected in the promise chain, .catch() will take priority over any .then(). This is an advantage for a few reasons:

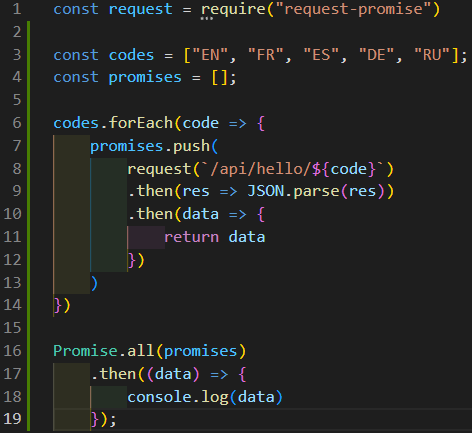
* No need to set up multiple .then() to handle the rejection correctly.
* It’s faster than passing it down multiple times.
* Because it’s exclusive to the rejection, you only really need one.

### **2.3.5** Promise.all()

Promise.all() is… you guessed it, a promise based method! Basically what it does is it takes an array of promises and waits for every single one to resolve or reject, then returns a single promise as a result.

In the screenshot below you’ll see that we’re looping over an array of codes with a forEach. In that loop, we’re pushing into an empty array the entire request (fetch) process. Because request is a promise based function, it will be a

Promise { <pending> } until it’s resolved. Promise.all() then waits for each pending promise in the array to fulfill before passing it down to its .then().

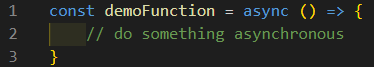


## **2.4 Async / Await**

Async / await is an alternative way of handling promises. Mostly because promises are gross and can get very messy very fast ([Pyramid of Doom](https://medium.com/dsc-srm/javascript-callback-hell-or-pyramid-of-doom-4f786d14b997)).

### **2.4.1 Async**

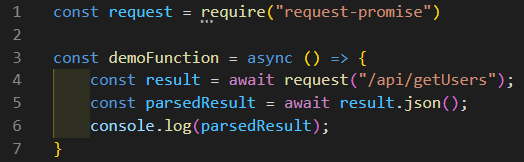
The **async** statement basically tells JavaScript (and Node by extension) that the function is asynchronous and will run asynchronous code.



It’s also a necessary keyword to be able to use **await**. Without **async**, **await** cannot exist.

### **2.4.2 Await**

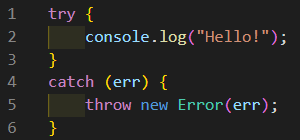
The **await** statement essentially forces JavaScript (and Node by extension) to wait for asynchronous code before moving on. It’s not like .then(). The .then() waits for asynchronous code to finish before executing. What **await** does is force JavaScript to wait before moving on to the next line; it forces asynchronous code to behave synchronously!



In the screenshot above, we’re telling JavaScript to wait for the request to finish, and store it in result. Then we tell it to wait again for the result to parse, and store it in parsedResult. After that it will console.log(). Because of the **await** statement, we don’t need to use any .then(), JavaScript is treating the code as if it was synchronous. Line 5 **WILL NOT** execute until line 4 is finished, and line 6 **WILL NOT** execute until line 5 is finished.

## **2.5 Try / Catch**

Try / catch is not exclusive to **async** / **await** functions, but because **async** / **await** doesn’t have any error handling, it’s almost always used with **try** / **catch**.



### **2.5.1 try**

The **try** block pretty much does what the name says: “try this code if it works”. It doesn’t really do anything special. It’s only there so that its partner, the **catch**, can trigger if something goes wrong. The try block cannot exist on its own. It **needs** to be accompanied by a **catch**, a **finally**, or both.

### **2.5.2 catch**

The **catch** block works like the .catch() method. If something in the **try** block fails, the **catch** block will be triggered and execute its code. The **catch** block **cannot** exist without the **try** block.

### **2.5.3 finally**

The **finally** block is the last thing that gets executed in the **try** / **catch** chain. It must come at the end of the chain. It’s mostly used for cleaning up or executing final operations that need to happen regardless of an error or not. The **finally** block cannot exist without the **try** block. The **finally** block **always** goes at the end of the **try** / **catch** chain.

## **2.6 Importing and Exporting**

### **2.6.1 Exporting**

Exporting in Node can only be done through an object. It’s the equivalent of named exports in React. Exporting in Node is pretty simple:

module.exports = { <variable>, <function> };

We’re exporting either a variable or a **function** from the module ( A.K.A the file) by placing them in an object. All the exported components will be exported in the object.

### **2.6.2 Importing**

Importing in Node is done through destructuring. It’s the equivalent of named imports in React. Importing in Node is pretty simple:

const { <variable>, <function> } = require("./handlers");

Because it’s a named import, we have to destructure the imported components. The names must match. The require keyword is what fetches the exported object from the module.

### **2.6.3 Importing JSON**

A JSON file does not have an export. Basically because they can only contain one data structure (either an object or an array), the file automatically exports its contents. Essentially it’s the Node equivalent of a default export in React. Importing from a JSON file is very similar to importing from a JavaScript file:

const data = require("./handlers");

We simply drop the {}, and give the imported data structure a name.

# **3. Mongo Concepts**

MongoDB is what is known as a NOSQL database. It stores data in object formats rather than in tables with specific rows and columns. It’s very OOP friendly.

## **3.1 Getting Started**

To get started with Mongo, you need to create an account, set up a database, and set up the connection protocols.

I strongly recommend going to your mongodb-P1 workshop and following the instructions in the README.md to get everything set up correctly.

## **3.2 Connecting to Mongo**

Assuming you set up everything correctly, you should have everything setup and your mongo connection string (called MONGO\_URI) should be in a .env file.

(continued on next page)

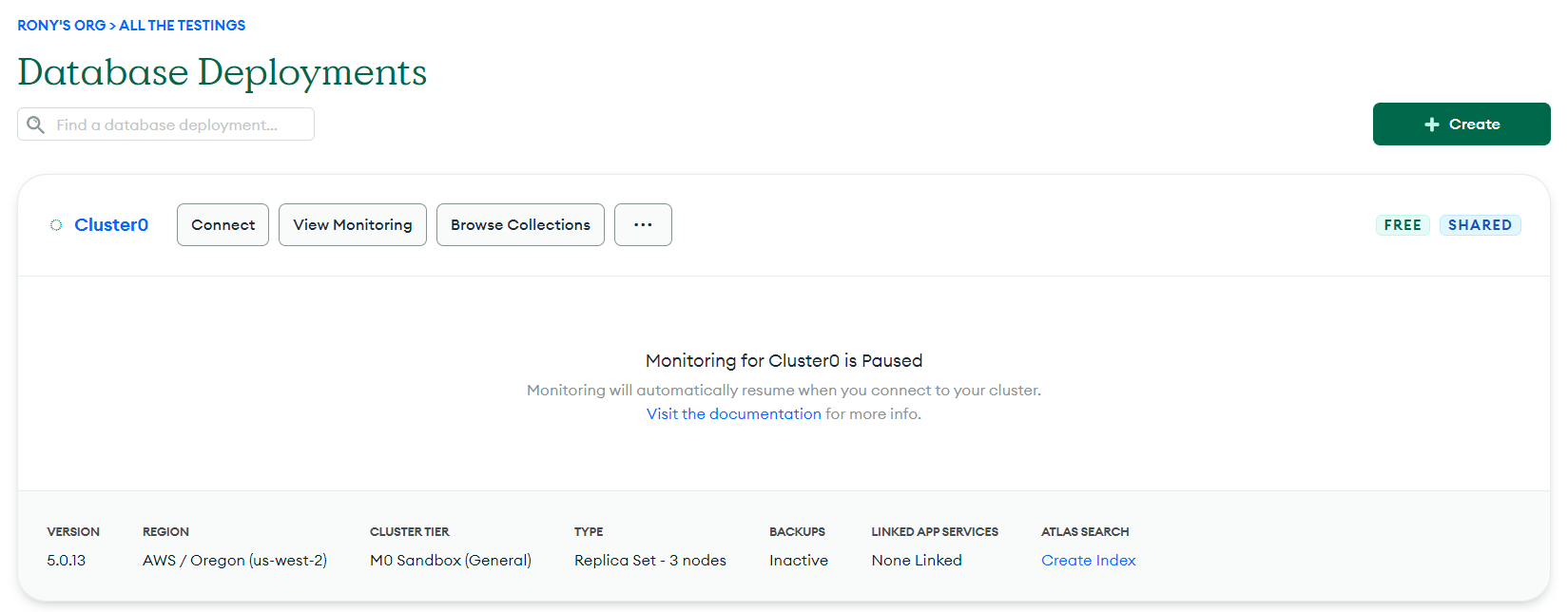
Here is a template code that you can copy and paste for a quick and easy function to connect to Mongo:

| // import the Mongo connection  const { MongoClient } = require("mongodb");  // import the connection key  require("dotenv").config();  const { MONGO\_URI } = process.env;  // setup the connection options  const options = {  useNewUrlParser: true,  useUnifiedTopology: true,  };  const demoFunction = async () => {  // creates a new connection client  const client = new MongoClient(MONGO\_URI, options);  // connect with the client  await client.connect();  // connect to the database  const db = client.db("demoDB");  // connect to the connection  const col = db.collection("demoCOL");  // close the connection to the database server  client.close();  };  demoFunction(); |
| --- |

## **3.3 Mongo Terminology**

### **3.3.1 Cluster**

A cluster is a collection of databases. Mongo provides us with clusters so we can group multiple databases together. For our intents and purposes, a cluster is a box where we keep our database.



### **3.3.2 Collection**

A collection is just what Mongo calls a database. They need to refer to it as a collection for their own systems. For us, a collection is a database.

### **3.3.3 Document**

A document is a single piece of data inside a database. Because Mongo follows OOP, all documents are simple objects.

### **3.3.4 \_id**

The \_id is a special property of Mongo that must be present in every document. If you don’t provide one, Mongo will. Why? So… even though everything in mongo is stored as an object, technically it’s [BSON](https://www.mongodb.com/json-and-bson) data. BSON data is just JSON data converted to binary structure. Because of this, the \_id is **REQUIRED**.

## **3.4 Mongo Methods**

**Most Mongo methods should have an await in front of them.**

The methods below will use the variable col based on the [template](#_tsduowtflgoy) above:

const col = db.collection("demoCOL");

### **3.4.1 client.db()**

The client.db() method takes a string as a parameter. It connects to the name of the database provided in the string. If the database doesn’t exist, it will create it first and then connect to it.

### **3.4.2 db.collection()**

The db.collection() method takes a string as a parameter. It connects to the name of the collection provided in the string. If the collection doesn’t exist, it will create it first and then connect to it.

### **3.4.3 .insertOne()**

The .insertOne() method inserts a new document into the collection. It takes an object as its parameter, often referred to as the data.

| await col.insertOne({name: "Rony"}); |
| --- |

### **3.4.4 .insertMany()**

The .insertMany() method inserts many documents into the collection. It takes an array of objects as its parameter, often referred to as the data.

| **const** documents = [{name: "Rony"}, {name: "Dipti"}];  await col.insertMany(documents); |
| --- |

### **3.4.5 .findOne()**

The .findOne() method returns the first document that matches its search parameter. It takes an object as its parameter, often referred to as the query. We almost always query on the \_id because it’s always unique.

| **await** col.findOne({\_id: 01}); |
| --- |

### **3.4.6 .find()**

The .find() method returns all the documents that match its search parameter. It takes an object as its parameter, often referred to as the query. If no query is provided, it will return all the documents. The .find() method returns some incompatible data, to make it usable we add .toArray() at the end to put all the results in an array.

| // returns everything  **await** col.find().toArray();  // returns all documents with this key:value pair  **await** col.find({grade: 75}).toArray(); |
| --- |

### **3.4.7 .updateOne()**

The .updateOne() method searches for and modifies the first document that matches its search parameter. It takes 2 objects for parameters in the following order:

1. A query object.
2. An update object.
   1. The update object takes a [$set](https://www.mongodb.com/docs/manual/reference/operator/update/set/#mongodb-update-up.-set) operator as a key, which is what applies the modification.
   2. The $set operator takes an object as its value, with the key:value pairs to replace in the original object.

We almost always query on the \_id because it’s always unique.

| **const** query = { \_id: 01 }; // document to modify **const** update = { $set: { name: "Dipti" } }; // fields to replace **await** col.updateOne(query, update); // replace |
| --- |

### **3.4.8 .update()**

⚠️ **This method is NOT recommended** ⚠️

The .update() method searches for and modifies all the documents that match its search parameter. It takes 2 objects for parameters in the following order:

1. A query object.
2. An update object.
   1. The update object takes a [$set](https://www.mongodb.com/docs/manual/reference/operator/update/set/#mongodb-update-up.-set) operator as a key, which is what applies the modification.
   2. The $set operator takes an object as its value, with the key:value pairs to replace in the original object.

| **const** query = { grade: 60 }; // document to modify **const** update = { $set: { status: "Pass" } }; // fields to replace **await** col.update(query, update); // replace |
| --- |

### **3.4.9 .deleteOne()**

The .deleteOne() method searches for and removes the first document that matches its search parameter. It takes an object as its parameter, often referred to as the query. We almost always query on the \_id because it’s always unique.

| **await** col.deleteOne({ \_id }); |
| --- |

### **3.4.10 .deleteMany()**

⚠️ **This method is NOT recommended** ⚠️

The .deleteMany() method searches for and removes all the documents that match its search parameter. It takes an object as its parameter, often referred to as the query.

| **await** col.deleteMany({ ingredient: "oatmeal" }); |
| --- |

## **3.5 Nested Documents**

⚠️ **This is an advanced topic** ⚠️

Sometimes in your database, you need to update an object that is nested in an array. Pretend you have a collection called "Classes", and in that collection you have several documents, each one representing a class. Here is a sample of one document.

{

\_id: ObjectId("507f1f77bcf86cd799439011"),

classId: "demo101",

students: [

{id: 001, grade: 75},

{id: 002, grade: 60},

{id: 003, grade: 88},

{id: 004, grade: 47},

{id: 005, grade: 91},

]

}

You have to change the grade of student 4, you accidentally wrote 47 instead of 74. You’ll need to [update](#_plct431mzga9) just that one object in the array. To do this, you’ll need to modify your query a bit:

{

classId: "demo101",

"students.id": 004

},

We’ve added a second query field, targeting the array called students, and even more specifically the key in the elements of the array called id with a value of 004.

(continued below)

The update object will also need a few changes:

{

$set: {"students.$.grade": 74}

}

The $ is what is known as a positional operator. It will target a specific key, in this case grade, using the new query field "students.id": 004 to know which position in the array to target and assign it a new value. Your operation will look like this:

col.updateOne(

// query object

{

classId: "demo101",

"students.id": 004

},

// update object

{

$set: {"students.$.grade": 74}

}

)

The collection will be modified to:

{

\_id: ObjectId("507f1f77bcf86cd799439011"),

classId: "demo101",

students: [

{id: 001, grade: 75},

{id: 002, grade: 60},

{id: 003, grade: 88},

{id: 004, grade: 74}, // this element has been updated

{id: 005, grade: 91},

]

}

# **4. Other Concepts**

## **4.1 The .env File**

The .env file is a very special file. It’s used to create, store, and use what we call "environment variables". Environment variables are variables that we literally put in the environment, rather than in the server. It’s accessible by the server, but it’s not directly available in the server because it exists in the environment.

Most of the time, environment variables are API keys, connection codes, encryption keys, etc. Valuable information that we must absolutely keep hidden, but also need in our server to be able to do things. So we put them in the .env file for safe keeping and call them with the [dotenv](https://www.npmjs.com/package/dotenv) package.

### **4.1.1 Where to put .env**

The .env file must exist in the same root folder as the files that are using the environment variables.

**For example:**

If the handlers.js file was calling a MONGO\_URI environment variable from the .env file, both these files have to be in the same folder. Usually that folder is called server or backend.

### ⚠️ **Important** ⚠️

We **NEVER** want to upload this file to GitHub. The .env file must stay secure at all times! To avoid this, make sure you have a [.gitignore](#_i1em3qtdquiz) file in the same folder as the .env file.

### **4.1.2 The .gitignore File**

The .gitignore file is a file that tells the git commands what should be ignored when uploading to GitHub. When you stage, commit, and push your changes to GitHub, it’s running git commands under the hood. The commands always look into the .gitignore file to make sure they don’t upload something that should not be uploaded.

### ⚠️ **Important** ⚠️

The 2 files we must make sure we **ALWAYS** have in the .gitignore file are:

* .env
* node\_modules

You can put anything else you want in the .gitignore file, but make sure you always have those 2 files.

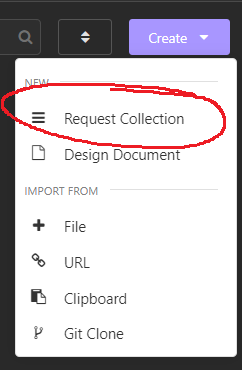
## **4.2 Insomnia**

Insomnia is a really good application for testing your server endpoints. Basically it runs a fetch request to a live endpoint. If your server isn’t running, nothing will happen.

There are a few of these applications out there, but we use Insomnia because it’s relatively easy to use. And also it’s free.

### **4.2.1 Creating a Collection**

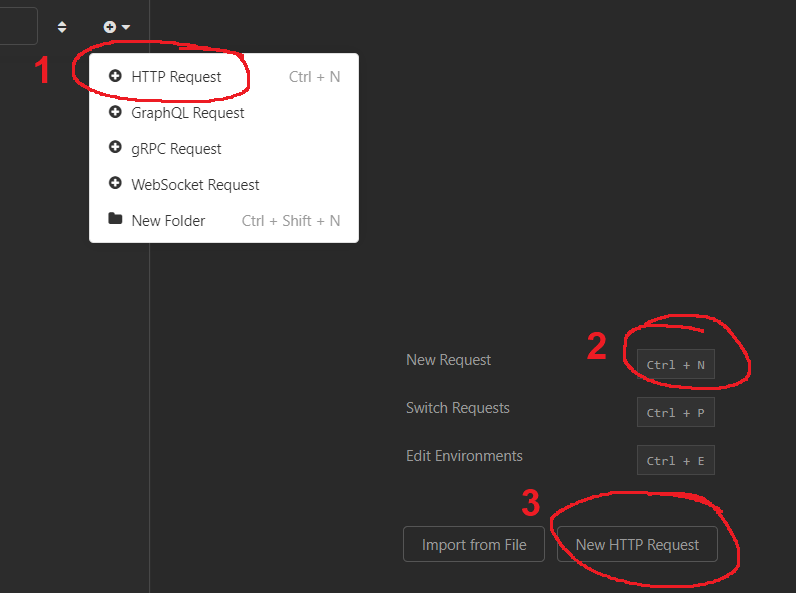
Before you can start testing endpoints, you need to create a Request Collection. On the Dashboard screen, the top right corner you should see a Create button. Click on it and then select Request Collection. A popup will prompt you to give it a name. You can call it whatever you want (though a clear name is always recommended).



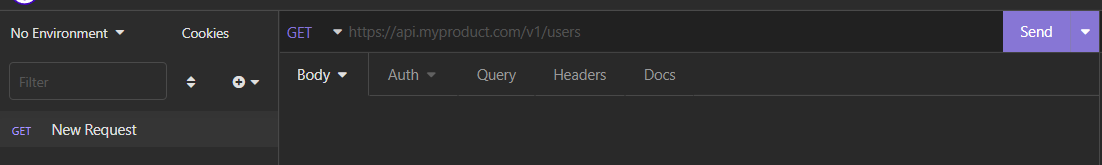
### **4.2.2 Creating a Request**

Next you’ll find yourself on a new screen. There are a few different ways to create a request. They all do the same thing:

1. Clicking the + sign, then clicking on HTTP Request.
2. Hitting CTRL + N (or CMD + N) on your keyboard.
3. Clicking on New HTTP Request.

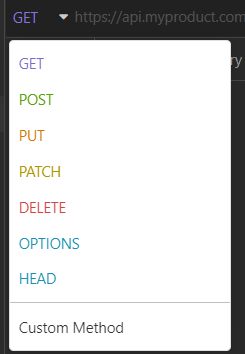


This will create a new GET request by default. You can create as many requests as you need.



### **4.2.3 Request Method**

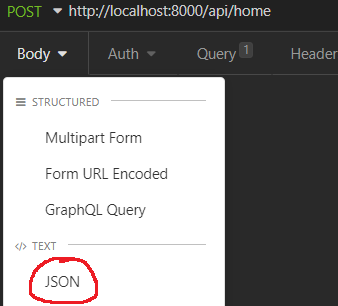
To change the request method, just click on the little arrow and select a new method.



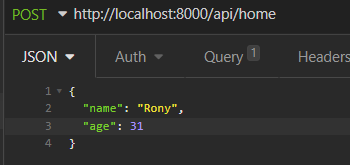
I have no idea what OPTIONS or HEAD do 🤷‍♂️.

### **4.2.4 Request Body**

To set a body in your fetch request, click on the Body option and choose JSON.

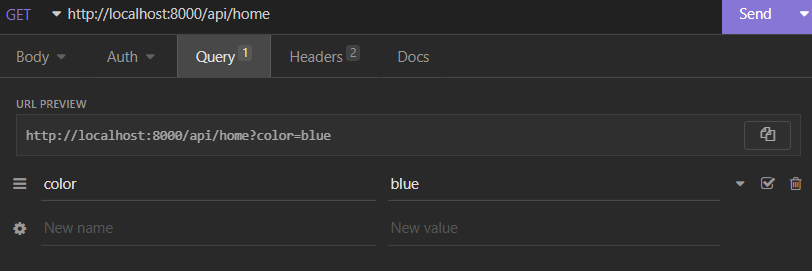


Just remember to write out the body in JSON format!



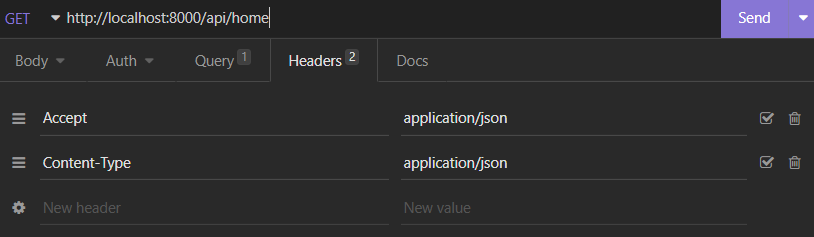
### **2.4.5 Request Queries**

To send some queries with your fetch request, go to the Query tab. You can also do it directly in the URL.



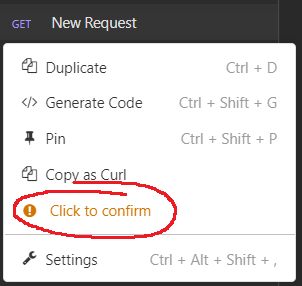
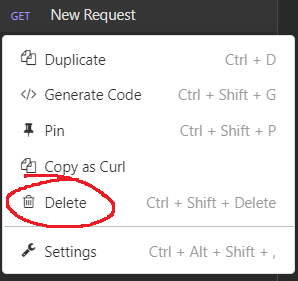
### **2.4.6 Request Headers**

To set up the headers for your fetch request, go to the Headers tab.



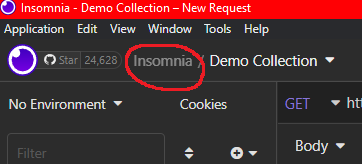
### **2.4.7 Deleting a Request**

To delete a request, right click on it and select the Delete option. You need to click on it twice. Once to start the deleting process, and a second time to confirm it.



### **2.4.8 Back to the Dashboard**

To get back to the dashboard, just click on the word Insomnia in the top left corner.



## **4.3 JSON Files**

A JSON file is a file that ends with the extension .json and contains a JSON object (groundbreaking revelations only in Rony’s cheat sheets). The JSON object can be either an object or an array (because arrays are technically objects). **A JSON file can only hold one object.**

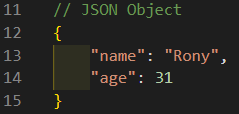
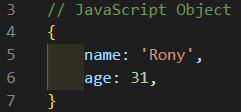
A JSON file is usually used for holding data, and since it can only have one object inside it, the data is always nested inside a parent object. Most of the time you’ll see an array of objects, or an object of objects. Data is very easy to store in the object format (see [Mongo](https://docs.google.com/document/d/1AzgSZu5jLeB_3qRUI-buSbShvdWji1UMruOI-O8nvdQ/edit#heading=h.vzky1wthjgyw)) and JSON is very OOP friendly.

### **4.3.1 JSON Formatting**

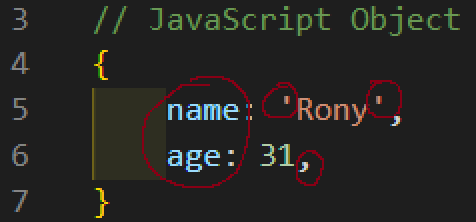
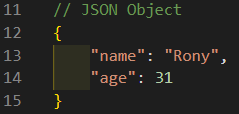
The JSON format is very similar to the JavaScript format, but it has a few unique rules added to it:

1. No trailing commas (, at the end that is not followed by another element).
2. All keys must be encased in "".
3. All strings must be encased in "".

**Example:**



**Differences:**

****

### **4.3.2 Importing JSON Data**

Because a JSON file can only hold one object, that object is always exported as default from the file. Which means when we want to import it, we must give a name to the imported data and target the file using the default import process.

**React:**

| **import** data **from** "./data.json"; |
| --- |

**Node:**

| **const** data = require("./data.json"); |
| --- |