

braintop
think.make.play

Node.js

Table Of Contents

- Chapter 1 - Introduction & Installation
- Chapter 2 - First Program
- Chapter 3 - First Server
- Chapter 4 - FS Module
- Chapter 5 - REST API
- Chapter 6 - Build a Server Without Express
- Chapter 7 - NPM
- Chapter 8 - Promises
- Chapter 9 - Build a Server Using Express
- Chapter 10 - middleware
- Chapter 11 - refactor to route and controller
- Chapter 12 - mongodb
- Chapter 13 - mongoose
- Chapter 14 - refactoring your mongoose project
- Chapter 15 - api security
- Chapter 16 - modeling data
- Chapter 17 - git source control
- Chapter 18 - git & heroku
- Chapter 19 - sql
- Chapter 20 - sql & mysql

Introduction

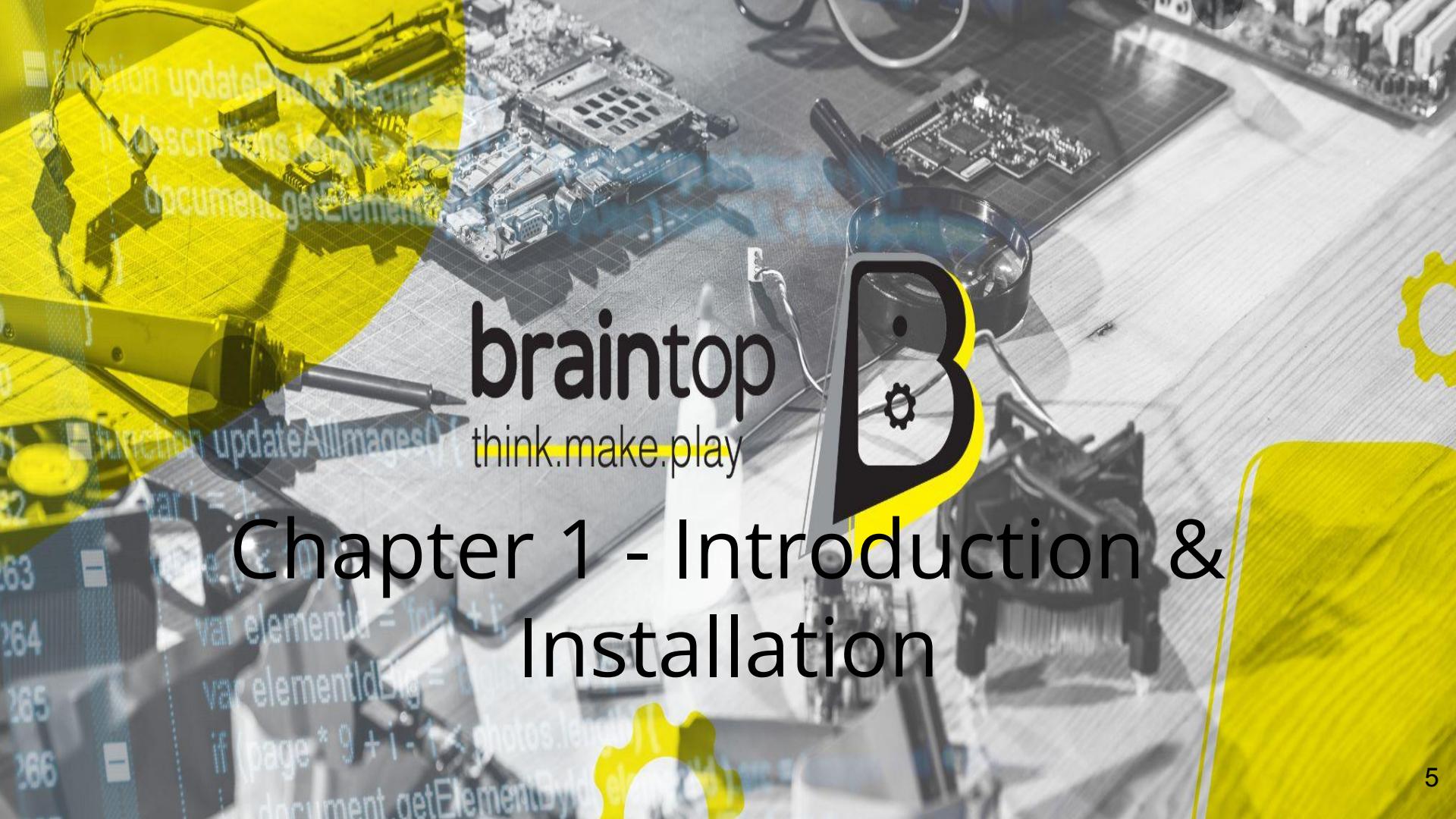
About node.js:

- Free
- Open Source Framework
- Runs on windows, linux, mac and more
- We will write in JavaScript

Introduction

Using node.js we can:

- Create dynamic web pages
- Open, close, delete and edit files on the server
- Collect data from forms
- Edit, create, add and delete data on databases



braintop
think.make.play

Chapter 1 - Introduction & Installation

Installation

Navigate to nodejs.org and install node.js

<https://nodejs.org/en/download/>

We also need a code editor. In this course we will use the platform Visual Studio Code.

<https://code.visualstudio.com/download>



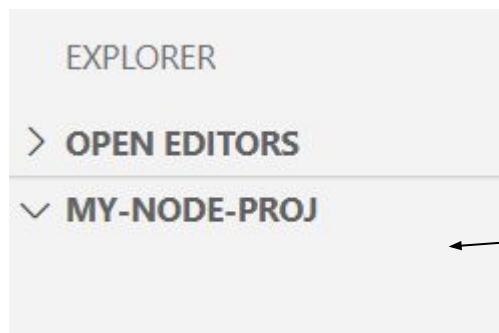
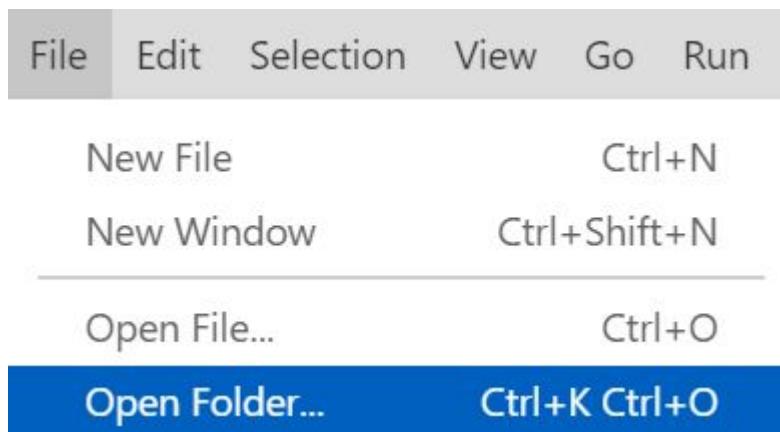
braintop
think.make.play

Chapter 2 - First Program

First Program

Create an empty folder on your computer, which later will contain your project.

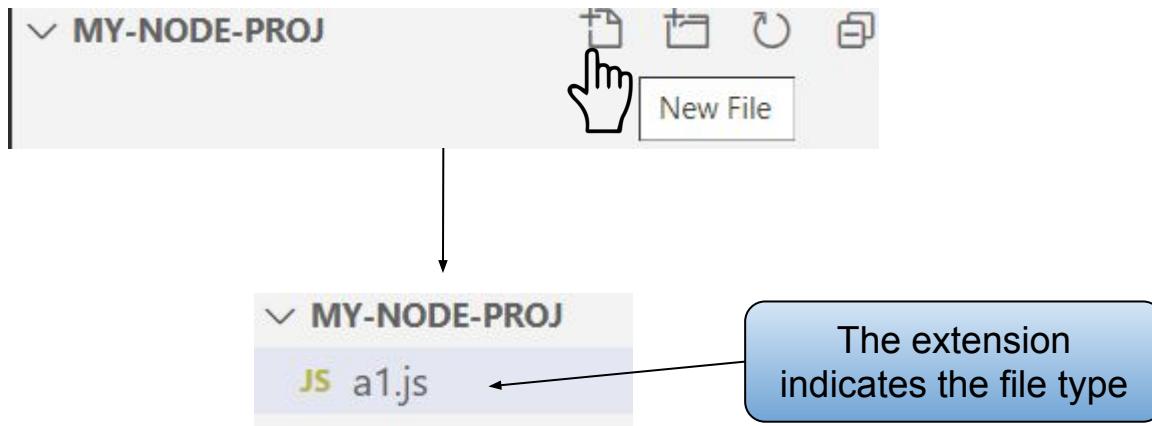
On VS Code - go to **file** -> **open folder** -> choose the folder you created



You can see the
project is empty

First Program

Create a new js file:



First Program

Let's start by logging "hello world":

```
console.log("Hello World");
```

Press Ctrl + s to save, or go to **file -> auto save**, to automatically save every edit in one of your files.

To run the project go to **terminal -> new terminal**.

In the terminal enter "node a1", where a1 is the file name.

```
PS C:\Users\X5-i7\OneDrive\Desktop\Programming\Asaf\my-node-proj> node a1
Hello World
```

First Module

In node we can create a module, which contains functions and exports them to use.

Create another js file:

MY-NODE-PROJ

JS a1.js

JS mymodule.js

First Module

In “mymodule.js” we will write functions and export them:

```
exports.plus = function(x, y) {  
    return x + y;  
}  
  
exports.minus = function(x, y) {  
    return x - y;  
}
```

The `module.exports` is a special object which is included in every JavaScript file in the Node.js application by default. The `module` is a variable that represents the current module, and `exports` is an object that will be exposed as a module. So, whatever you assign to `module.exports` will be exposed as a module.

First Module

And let's use the module's functions in another file, called "program.js":

In program.js:

```
var computeModule = require('./mymodule.js');

var sum = computeModule.plus(5, 6);
console.log("sum is " + sum);

var sub = computeModule.minus(5, 6);
console.log("sub is " + sub);
```

A variable that
contains 'mymodule'

```
PS C:\Users\X5-i7\OneDrive\Desktop\Programming\Asaf\my-node-proj> node program
sum is 11
sub is -1
```

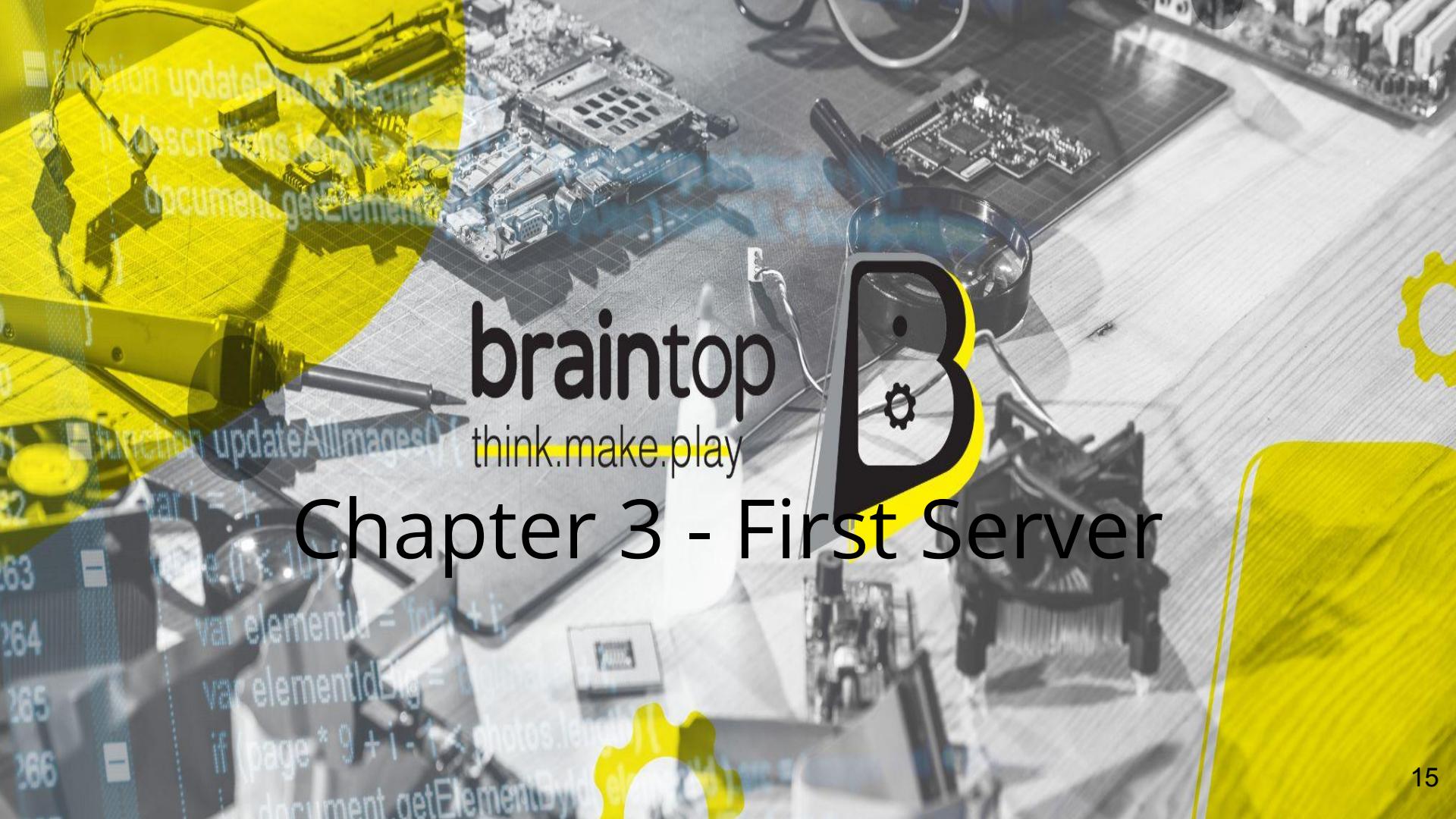
do it yourself 1

Export a Name Module. It has 2 functions:

1. A function that receives **first** and **last name** and returns it with “**hello**”.
2. A function that receives **sender name** , **message** and **subject** and prints them.

For Example: “Jack”, “Hello, how are you?”, “Dan”, the return message is: “**Dan**, you got a new message from **Jack**: Hello, how are you?”

Create a program and use it.



braintop
think.make.play

Chapter 3 - First Server

How Does The Internet Work?

Directory files

In this chapter you have 8 source files in the chapter folder

server1.js

server2.js

server3.js

server4.js

server5.js

server6.js

server7.js

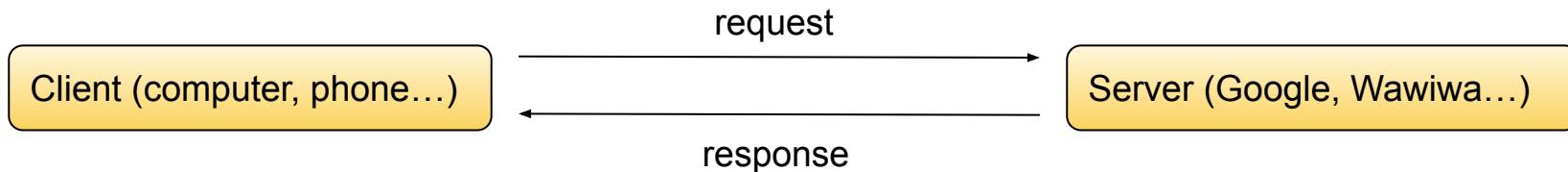
red/red1.js

red/red2.js

red/red3.js

How Does The Internet Work?

When we enter an address we receive a web page that contains text, images and more.



How Does The Internet Work?

Every address on the internet have an **ip** address.

ip address is a number that is used to identify clients and servers, and is dynamically allocated inside the local network.

There can be 2 address with the same ip, if they are not connected to the same local network.

That's why every computer also have a physical address that can't be changed, called **MAC** address.

How Does The Internet Work?

In order to find the ip of a server, open the cmd:

- Press win key + r and enter 'cmd'
- run the command 'ping':

```
C:\Users\X5-i7>ping google.co.il

Pinging google.co.il [216.58.206.3] with 32 bytes of data:
Reply from 216.58.206.3: bytes=32 time=70ms TTL=112
```

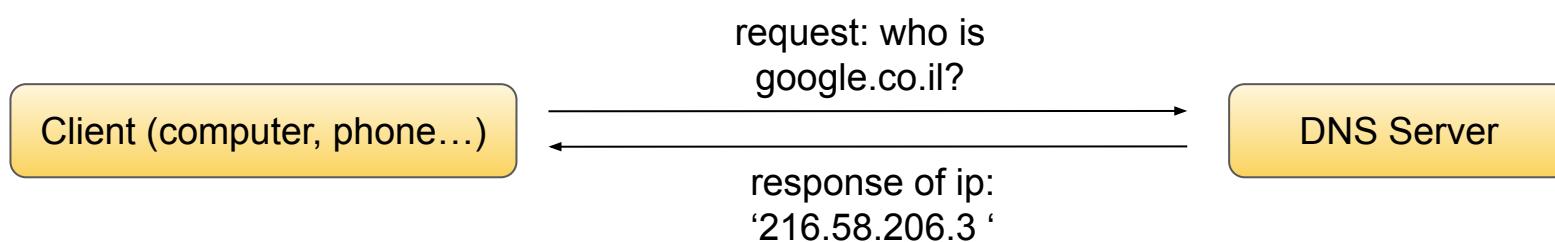
Notice that you can send data to yourself with ip '127.0.0.1'.

How Does The Internet Work?

DNS:

Domain Name System is a protocol for accessing databases, in order to let clients translate readable human addresses (urls) to domain names (ips)

More about dns : https://en.wikipedia.org/wiki/Domain_Name_System



How Does The Internet Work?

Port:

A specific process which programs can directly stream data through.

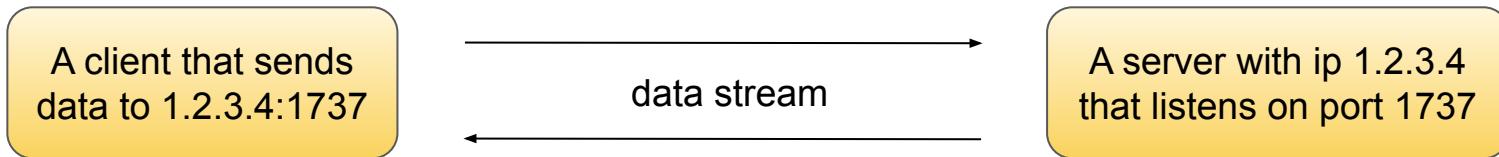
Let's say ip is the address of a building. To send a letter we need to know the address (ip) and apartment (port) too.

How Does The Internet Work?

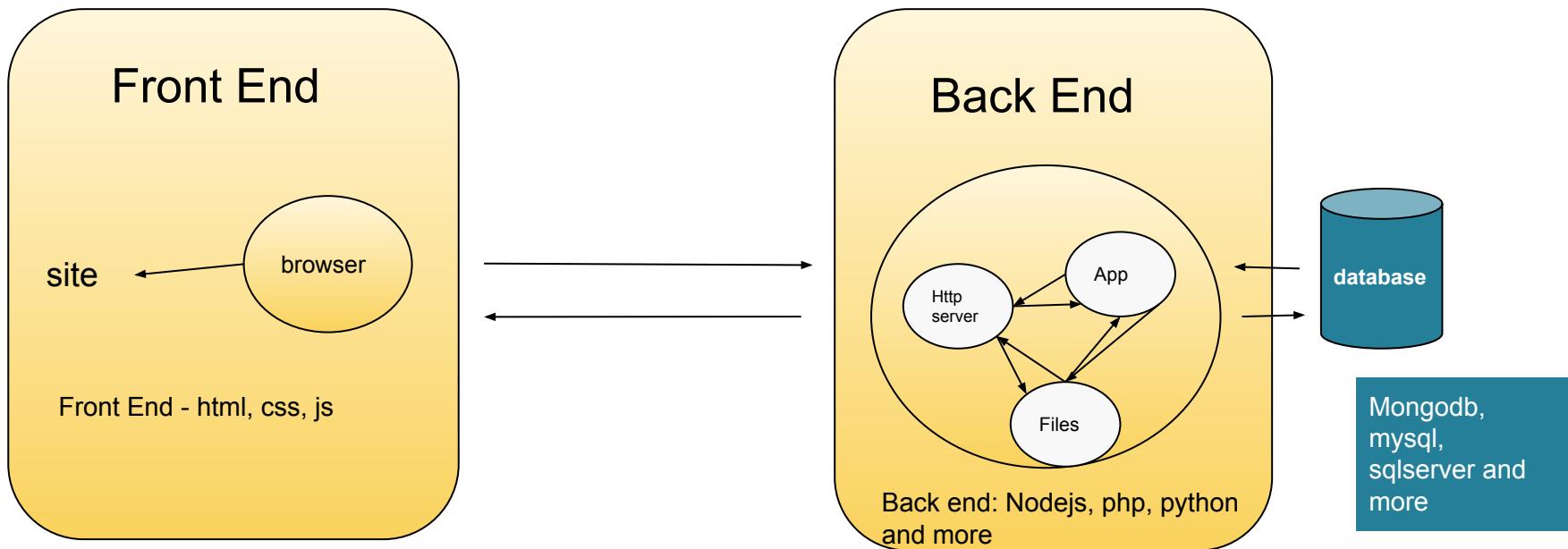
Socket:

A network socket is a client for streaming data between two processes in the network.

A socket address is the combination of the ip and port.

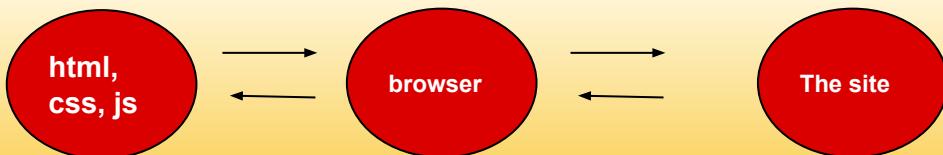


Front-End vs. Back-End

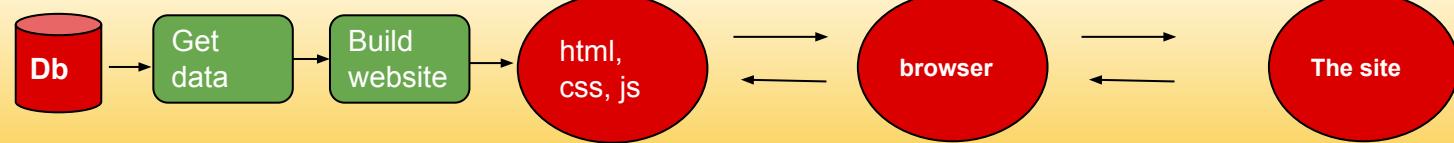


Static, Dynamic and Api

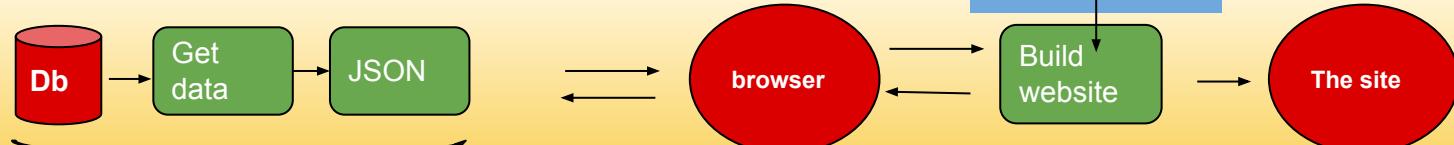
Static



Dynamic



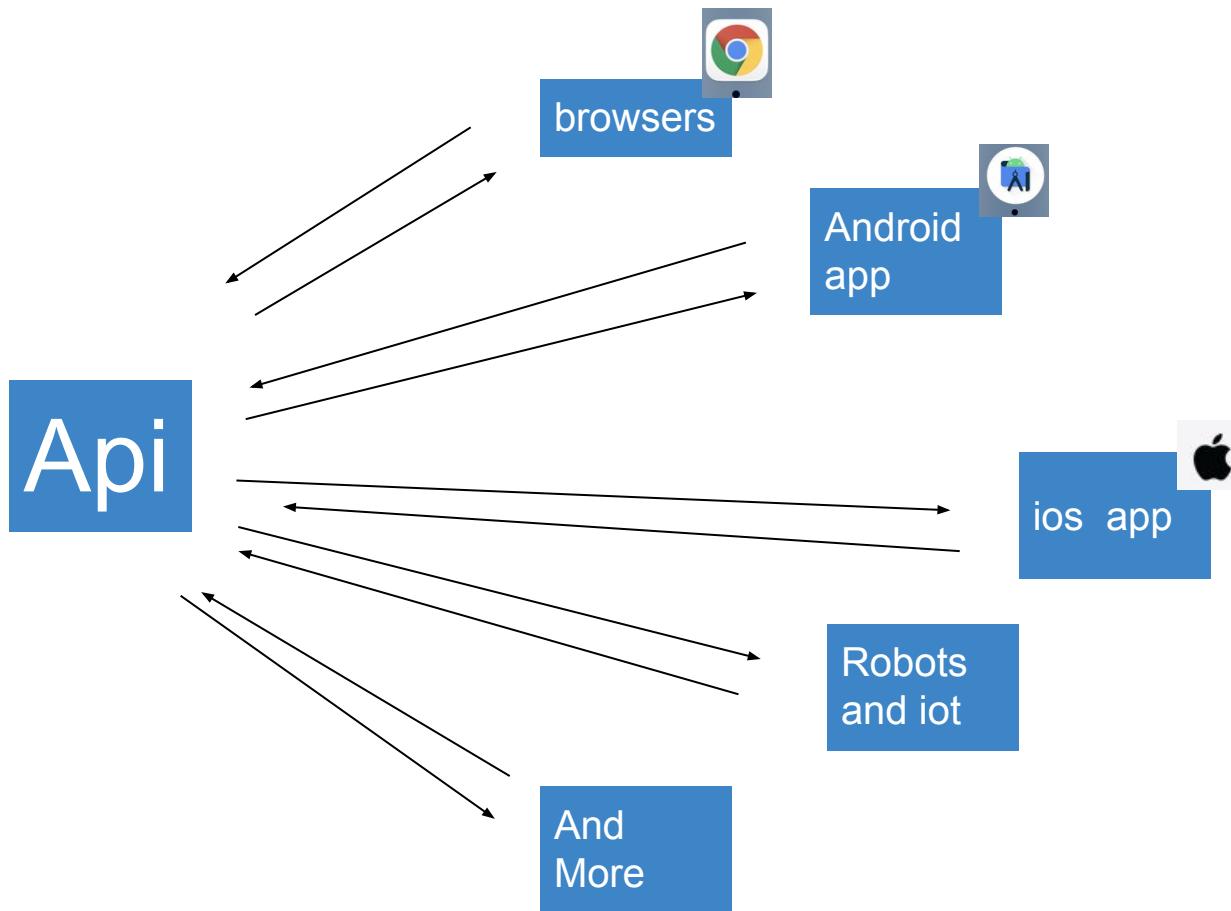
Api



Building Api

Consuming Api

Api



First Server

In this chapter we will build the first server in this guide, with http module.

In a new file called “server1.js”:

```
var http = require('http');
var server_function = function(req, res) {
  res.write('Hello World!');
  res.end();
}
var server = http.createServer(server_function);

server.listen(3000);
```

Use the module http,
which is built-in in node

The server responds with
“Hello World” no matter what
is the request

We create a server using
the http module, and pass
the server_function to it

The server will listen
on port 3000

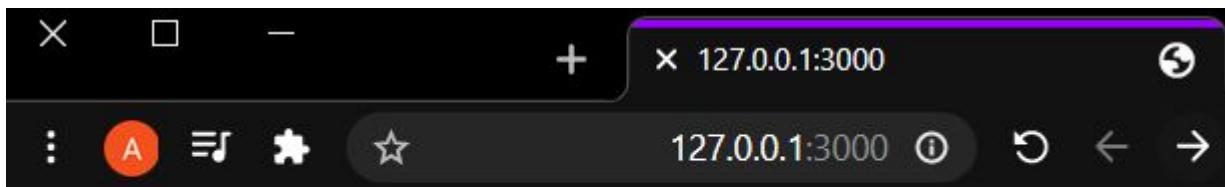
First Server

Because we created the server1.js file under a directory, we need to change the directory of the terminal to run the program:

```
chap12-mongodb % cd /Users/asafamir/Desktop/my-node-proj/chap3-firstserver  
chap3-firstserver % node server1
```

'cd' means
Change Directory

Now open a browser, and request our computer (127.0.0.1, or localhost) with port 3000:



Hello World!

First Server

For training let's build another server, in a new file called 'server2.js':

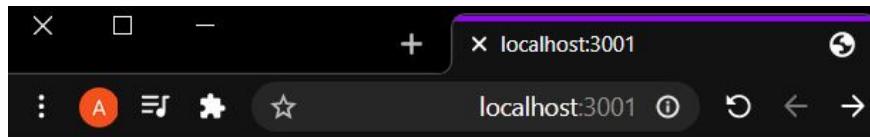
```
var http = require('http');

http.createServer(function(req, res) {
  res.end('Hello World 2!');
}).listen(3001);
```

This program acts the same as server1, but with fewer line

In the terminal press **ctrl + c** to exit the previous run (of server1), and run server 2.

Now open a browser, and request our computer with port 3001:



Hello World 2!

First Server

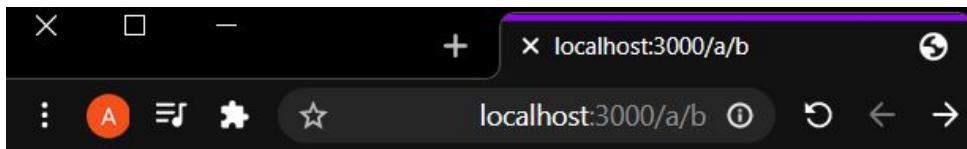
And the last server in this chapter ‘server3.js’:

```
var http = require('http'); ← http module

http.createServer(function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  res.write("client asked " + req.url);
  res.end();
}).listen(3000);
```

The response can have a header, which contains more data from the server.

In this header ‘200’ indicates the status code, which means ‘OK’ (the data the user requested was found). And the content type of the response ‘text/html’, which can also be css or js or whatever we want



client asked /a/b

We can see that the user asked to see the data in file ‘b’ that in directory ‘a’ in the server

Do it yourself 1

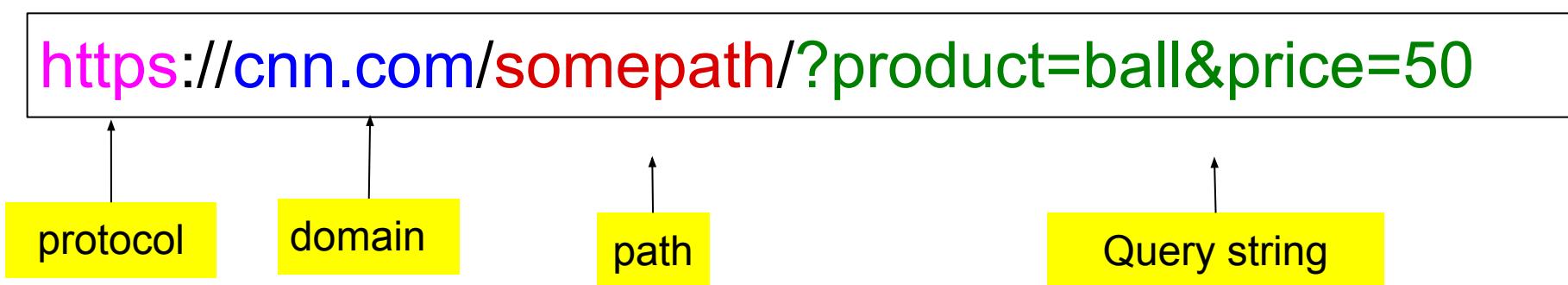
Create a server that responds to 127.0.0.1:3000 with your first name and last name

Do it 1 yourself solution

```
var http = require('http');

http.createServer(function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  res.write("My name is Mike Lewis");
  res.end();
}).listen(3000);
```

Url Structure

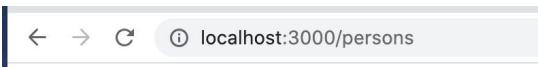
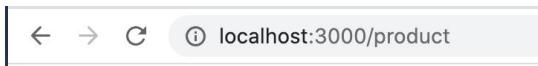
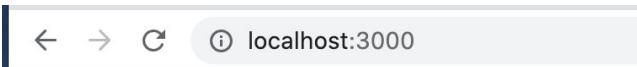


First Server - Routing

And the last server in this chapter ‘server4.js’:

```
var http = require('http');

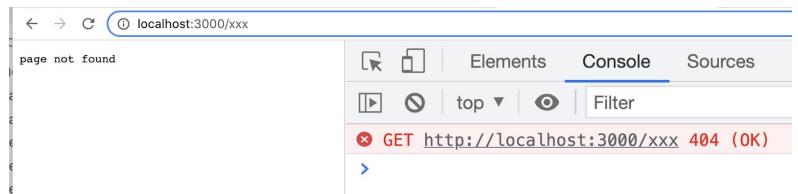
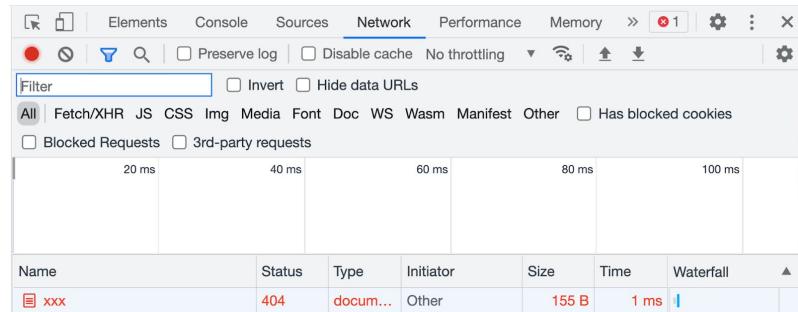
http.createServer(function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  let path = req.url;
  if(path === "/") {
    res.write("client asked for home page");
  } else if(path === "/product") {
    res.write("client asked for products");
  } else if(path === "/persons") {
    res.write("client asked for persons");
  }
  res.end();
}).listen(3000);
```



First Server - Routing

And the last server in this chapter 'server4.js':

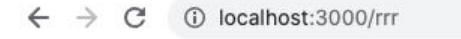
```
var http = require('http');
http.createServer(function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  let path = req.url;
  if(path === "/")
    res.write("client asked for home page");
  else if(path === "/product")
    res.write("client asked for products");
  else if(path === "/persons")
    res.write("client asked for persons");
  else {
    res.writeHead(404)
    res.write("page not found");
  }
  res.end();
}).listen(3000);
```



First Server - Routing

And the last server in this chapter ‘server5.js’:

```
var http = require('http');
http.createServer(function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  let path = req.url;
  if(path === "/")
    res.write("client asked for home page");
  else if(path === "/product")
    res.write("client asked for products");
  else if(path === "/persons")
    res.write("client asked for persons");
  else {
    res.writeHead(404, {
      'Content-Type': 'text/html'
    });
    res.write("<h1>page not found</h1>");
  }
  res.end();
}).listen(3000);
```



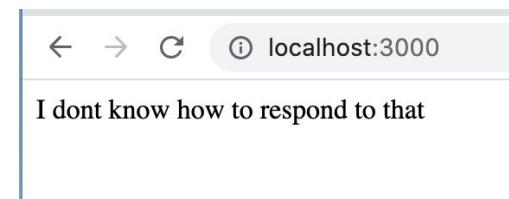
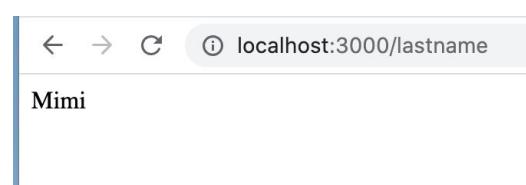
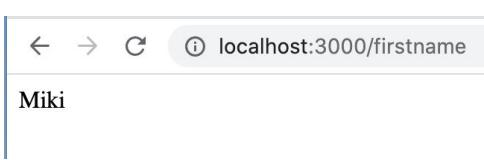
A screenshot of a web browser window. The address bar at the top shows the URL "localhost:3000/rrr". Below the address bar, the main content area displays the text "page not found" in a large, bold, black font.

page not found

Do it yourself 2

Build a server on port 3000 that responds to the user in the following way:

1. For the url ‘/firstname’ - return “Miki”
2. For the url ‘/lastname’ return “Mimi”
3. For every other request responds with “I don’t know how to respond to that”



Do it yourself 2 solution

```
var http = require('http');
http.createServer(function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  let path = req.url;
  if(path === "/firstname")
    res.write("Miki");
  else if(path === "/lastname")
    res.write("Mimi");
  else {
    res.write("I don't know how to respond to that");
  }
  res.end();
}).listen(3000);
```

SearchParams - server6.js:

```
var http = require('http');
var url = require('url');
http.createServer(function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  const baseURL = req.protocol + '://' + req.headers.host + '/';
  const reqUrl = new URL(req.url, baseURL);
  console.log(reqUrl)
  let pathname = reqUrl.pathname
  let searchParams = new URLSearchParams(reqUrl.searchParams);
  console.log(searchParams)
  console.log(searchParams.has('firstname') === true); //true
  console.log(searchParams.get('firstname') === "mike"); //true
  console.log(searchParams.getAll('age'));//[]
  console.log(searchParams.get('foo') === null); //true
  console.log(searchParams.toString()); //firstname=mike&lastname=even
  console.log(searchParams.set('age', '12'));
  console.log(searchParams.toString()); //firstname=mike&lastname=even&age=12
  searchParams.delete('firstname');
  console.log(searchParams.toString()); //lastname=even&age=12
  res.write("<h1>page not found</h1>");
  res.end();
}).listen(3000);
```



searchParams

```
URL {  
  href: 'undefined://localhost:3000/persons?firstname=mike&lastname=%22even%22',  
  origin: 'null',  
  protocol: 'undefined:',  
  username: '',  
  password: '',  
  host: 'localhost:3000',  
  hostname: 'localhost',  
  port: '3000',  
  pathname: '/persons',  
  search: '?firstname=mike&lastname=%22even%22',  
  searchParams: URLSearchParams { 'firstname' => 'mike', 'lastname' => '"even"' }  
  hash: ''  
}  
URLSearchParams { 'firstname' => 'mike', 'lastname' => '"even"' }  
[ 'firstname', 'mike' ]  
[ 'lastname', '"even"' ]  
true  
true  
[]  
true  
firstname=mike&lastname=%22even%22  
undefined  
firstname=mike&lastname=%22even%22&age=12  
lastname=%22even%22&age=12  
□
```

searchParams - iterate the search parameters- server7.js:

```
var http = require('http');
var url = require('url');

http.createServer(function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });

  const baseURL = req.protocol + '://' + req.headers.host + '/';
  const reqUrl = new URL(req.url, baseURL);

  let searchParams = new URLSearchParams(reqUrl.searchParams);
  //Iterate the search parameters.

  for (let p of searchParams) {
    console.log(p);
  }

  res.write("<h1>search parameters</h1>");
  res.end();
}).listen(3000);
```



localhost:3000/persons?firstname=mike&lastname="even"

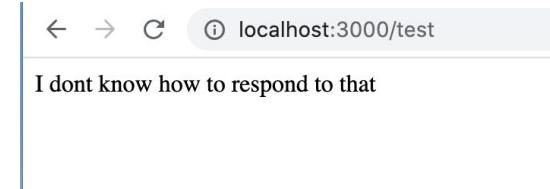
search parameters

```
[ 'firstname', 'mike' ]
[ 'lastname', '"even"' ]
```

Do it yourself 3

Build a server on port 3000 that responds to the user in the following way:

1. For the url '[/params](#)' - return all parameters
2. For every other request responds with "I don't know how to respond to that"



Do it yourself 3 solution

```
var http = require('http');
http.createServer(function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });
  const baseURL = req.protocol + '://' + req.headers.host + '/';
  const reqUrl = new URL(req.url, baseURL);
  let path = reqUrl.pathname
  if(path === "/params") {
    let searchParams = new URLSearchParams(reqUrl.searchParams);
    res.write(searchParams.toString());
  }
  else {
    res.write("I don't know how to respond to that");
  }
  res.end();
}).listen(3000);
```

Do it yourself 1

Create a server that responds to 127.0.0.1:3000 with your city name and your country name.

Do it yourself 2

Build a server on port 3000 that responds to the user in the following way:

1. For the url ‘/city’ - return “Paris”
2. For the url ‘/country’ return “Romania”
3. For every other request responds with “I don’t know how to respond to that”

Do it yourself 3

Build a server on port 3000 that responds to the user in the following way:

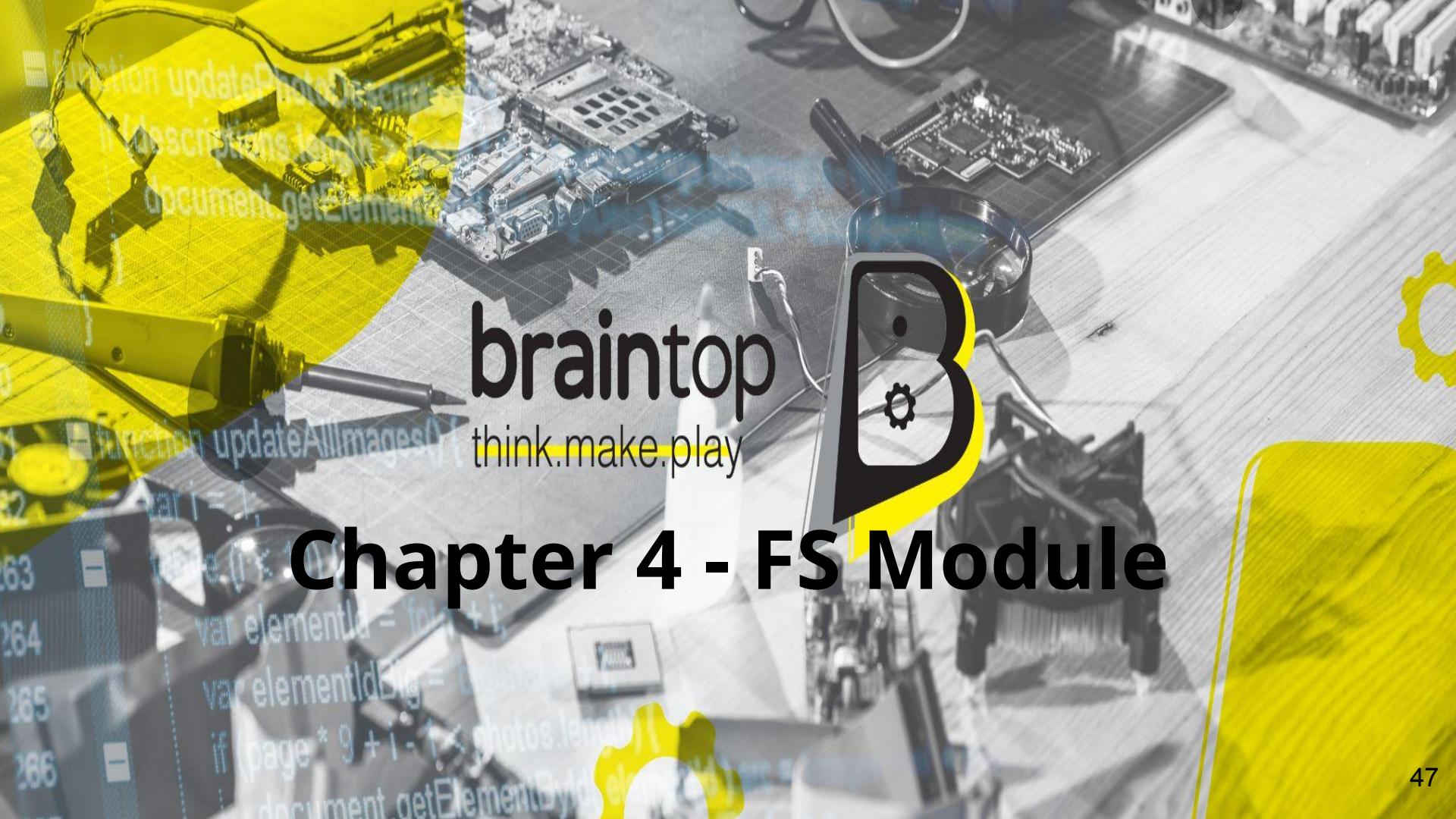
The server will receive two parameters a and b and

1. For the url `/plus?a=5&b=6` - return sum of parameters
2. For the url `/mult?a=5&b=6` - return multiplied parameter
3. For every other request responds with “I don’t know how to respond to that”

```
← → ⌂ ⓘ localhost:3000/plus?a=5&b=6
11
```

```
← → ⌂ ⓘ localhost:3000/mult?a=5&b=6
30
```

```
← → ⌂ ⓘ localhost:3000/mult?a=5
I dont know how to respond to that
```



braintop
think.make.play

Chapter 4 - FS Module

Reminder - try and catch [mdn](#)

The try...catch statement marks a block of statements to try and specifies a response should an exception be thrown.

```
try {  
    nonExistentFunction();  
} catch (error) {  
    console.error(error);  
    // expected output: ReferenceError: nonExistentFunction is not defined  
    // Note - error messages will vary depending on browser  
}
```

Reminder - try and catch [mdn](#)

```
try {  
    try_statements  
}  
  
catch (exception_var) {  
    catch_statements  
}  
  
finally {  
    finally_statements  
}
```

try_statements
The statements to be executed.

catch_statements
Statement that is executed if an exception is thrown in the try-block.

exception_var
An optional identifier to hold an exception object for the associated catch-block.

finally_statements
Statements that are executed after the try statement completes. These statements execute regardless of whether an exception was thrown or caught.

Unconditional catch block

When a catch-block is used, the catch-block is executed when any exception is thrown from within the try-block. For example, when the exception occurs in the following code, control transfers to the catch-block.

```
try {  
    throw 'myException'; // generates an exception  
} catch (e) {  
    // statements to handle any exceptions  
    logMyErrors(e); // pass exception object to error handler  
}
```

Conditional catch block

You can create "Conditional catch-blocks" by combining try...catch blocks with if...else if...else structures, like this:

```
try {
    myroutine(); // may throw three types of exceptions
} catch (e) {
    if (e instanceof TypeError) {
        // statements to handle TypeError exceptions
    } else if (e instanceof RangeError) {
        // statements to handle RangeError exceptions
    } else if (e instanceof EvalError) {
        // statements to handle EvalError exceptions
    } else {
        // statements to handle any unspecified exceptions
        logMyErrors(e); // pass exception object to error handler
    }
}
```

The exception identifier

When an exception is thrown in the try-block, exception_var (i.e., the e in catch (e)) holds the exception value. You can use this identifier to get information about the exception that was thrown. This identifier is only available in the catch-block scope. If you don't need the exception value, it could be omitted.

```
function isValidJSON(text) {  
  try {  
    JSON.parse(text);  
    return true;  
  } catch {  
    return false;  
  }  
}
```

The finally-block

The finally-block contains statements to execute after the try-block and catch-block(s) execute, but before the statements following the try...catch...finally-block. Note that the finally-block executes regardless of whether an exception is thrown. Also, if an exception is thrown, the statements in the finally-block execute even if no catch-block handles the exception.

The following example shows one use case for the finally-block. The code opens a file and then executes statements that use the file; the finally-block makes sure the file always closes after it is used even if an exception was thrown.

```
openMyFile();
try {
    // tie up a resource
    writeMyFile(theData);
} finally {
    closeMyFile(); // always close the resource
}
```

Reading a file content

Using node we can use the file system module to manage the files on our server.

The first program we will build reads file content.

Create a file called a.txt and write to the file the sentence ‘hello world’:



```
a.txt      X
chap4 - fs >  a.txt
1 hello world
```

A screenshot of a terminal window. The window title is 'a.txt'. The current directory is 'chap4 - fs >'. A file named 'a.txt' is open, showing the number '1' followed by the text 'hello world'.

Reading a File Content

f1.js

```
var fs = require("fs");

try {
  var data = fs.readFileSync("a.txt", 'utf-8');
  console.log(data);
} catch (e) {
  console.log("can't read from a.txt");
}
```

Read the files from a.txt.
Sync means we won't continue
until we finished reading

In case there was a problem reading the
files from lib (the lib doesn't exist, etc...)

```
asafamir@Asafs-MBP chap4-fs % node f1
hello world
```

More-Reading File Content async

In lib/first.html write "hello!", so later we can read it. f2_read_file_content_async.js

```
var fs = require("fs");
var path = require("path");

var filePath = path.join(__dirname, "lib", "first.html");

var stats = fs.statSync(filePath);

if (stats.isFile()) {
  fs.readFile(filePath, "UTF-8", function(err, contents) {
    console.log(contents);
  });
}
```

path is a built-in module in node for building a file path

__dirname is the current dir path.
So the final path is:
C:\Users\X5-i7\OneDrive\Desktop\Programming\Asaf\my-node-proj\chap3\lib\first.html

We need to wait for the path module to finish building our path before we continue

fs considers directories as files too, so we need to check that it really is a file

Writing To a File synchronous

f3_write_file_sync.js

```
var fs = require("fs");
var path = require("path");

var content = `

So
Much
Content!!!
`
```

The file name name will be a.txt

```
fs.writeFileSync("a.txt", content.trim())
```

The trim() method removes whitespace from both sides of a string.

Writing To a File asynchronous

f4_write_file_async.js

```
var fs = require("fs");
var path = require("path");

var content = `So
Much
Content!!!
`


var filePath = path.join(__dirname, "lib", "f1.txt");
fs.writeFile(filePath, content.trim(), function(err) {
  if (err) {
    console.log(err)
  } else {
    console.log("file created")
  }
})
```

The file name will be
'f1.txt' and it will be under lib

The trim() method removes
whitespace from both sides
of a string.

Writing To a File- append file

f5-append_content.js

And we can also append content to the end of an existing file:

```
var fs = require("fs");
var path = require("path");

var filePath = path.join(__dirname, "lib", "f1.txt");

fs.appendFile(filePath, "hi - how do u feel?", function(err) {
  if (err) {
    console.log(err)
  } else {
    console.log("appended content");
  }
})
```

Writing To a File

So the final result of f1.txt is:

So

Much

Content!!!hi - how do u feel?

Do it yourself 1

Create a program in node that [writes](#) **synchronously** to a file named test.txt with your full name.

Do it yourself 2

Create a program in node that **writes asynchronously** to a file named test.txt with your full name.

Do it yourself 3

Write a program that **reads synchronous** from a file named test.txt and prints the text to the console

Do it yourself 4

Write a program that **reads asynchronous** from a file called test.txt and prints the text to the console

Rename a File

f6_rename_sync.js

Synchronous:

```
var fs = require("fs");
fs.renameSync("./lib/f1.txt", "f1 after.txt");
console.log("f1 rename to 'f1 after'");
```

Attention!

We also changed the path,
from ./lib/ to our current dir

f6b_rename_async.js

Asynchronous:

```
var fs = require("fs");

fs.rename("./lib/f1.txt", "f1 after.txt", function(err) {
  if (err) {
    console.log(err)
  } else {
    console.log("file move");
  }
})
```

Delete a File

```
f7_delete_file_sync.js
```

Synchronous:

```
var fs = require("fs");

try {
  fs.unlinkSync("./f1.txt");
} catch (e) {
  console.log(e);
}
```

Delete a File

f8_delete_file_async.js

Asynchronous:

```
var fs = require("fs");

fs.unlink("f1.txt", function(err) {
  if (err) {
    console.log(err)
  } else {
    console.log("file deleted");
  }
});

console.log("hi");
```

Create a Directory

fs9_make_dir.js.js

```
var fs = require("fs");

if (!fs.existsSync("mydir")) {
  fs.mkdir("mydir", function(err) {
    if (err) {
      console.log(err);
    } else {
      console.log("directory created");
    }
  })
} else {
  console.log("directory exist");
}
```

Create the directory only if it isn't already exists
The path is relative so it searches and creates it
under 'chap 4', our current dir

Reading a File List from directory - synchronous

f10_read_file_names_sync.js

In fs1.js:

```
var fs = require("fs");

try {
    var files = fs.readdirSync("./lib");
    console.log(files);
} catch (e) {
    console.log("can't read from lib");
}

console.log("hello world");
```

Read the files from lib.
Sync means we won't continue
until we finished reading

In case there was a problem reading the
files from lib (the lib doesn't exist, etc...)

Reading a File List from directory - asynchronous

f11_read_file_names_sync.js

And another example. In fs2.js:

```
var fs = require("fs");

try {
  fs.readdir("./lib", function(err, files) {
    if (err) {
      throw err;
    }
    console.log(files);
  })
} catch (e) {
  console.log("can't read files");
}

console.log("hello world");
```

Read the files from lib.
This will be asynchronous so the
program will continue without
waiting for the files

In case of error while reading
we can handle it or throw it
and catch it outside

Reading a File List

So the output with asynchronous is:

```
hello world
[ 'first.html', 'second.html' ]
```

Because reading the files takes more time than logging.

Reading Files Content from directory

f12_read_files content_async.js

An example for reading the contents of all the files in a directory:

```
var fs = require("fs");
var path = require("path");

fs.readdir("./lib", function(err, files) {
  files.forEach(function(filename) {
    var filePath = path.join(__dirname, "lib", filename);
    var stats = fs.statSync(filePath);

    if (stats.isFile()) {
      fs.readFile(filePath, "UTF-8", function(err, contents) {
        console.log(contents);
      });
    }
  });
});
```

Output : hello from second too!
 hello!

Delete a Directory

f13_delete_dir.js

```
var fs = require("fs");

fs.readdirSync("./lib/data").forEach(function(filename) {
  fs.unlinkSync("./lib/data/" + filename);
});

fs.rmdir("./lib/data", function(err) {
  if (err) {
    console.log(err)
  } else {
    console.log("remove data directory")
  }
})
```

First delete all the files
in the directory 'data'

Delete the directory 'data'
now that it's empty

Do it yourself 1

Create a program in node that writes **synchronously** to a file named memory.txt with your city and your country

Do it yourself 2

Create a program in node that writes **asynchronously** to a file named memory.txt with your city and your country

Do it yourself 3

Write a program that reads **synchronous** from a file named test.txt and prints the text to the console

Create a File Server

f14_files_server.js

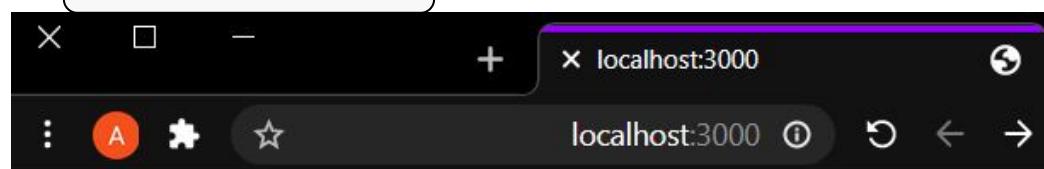
Return the client the content of a file:

```
var http = require('http');
var fs = require('fs');

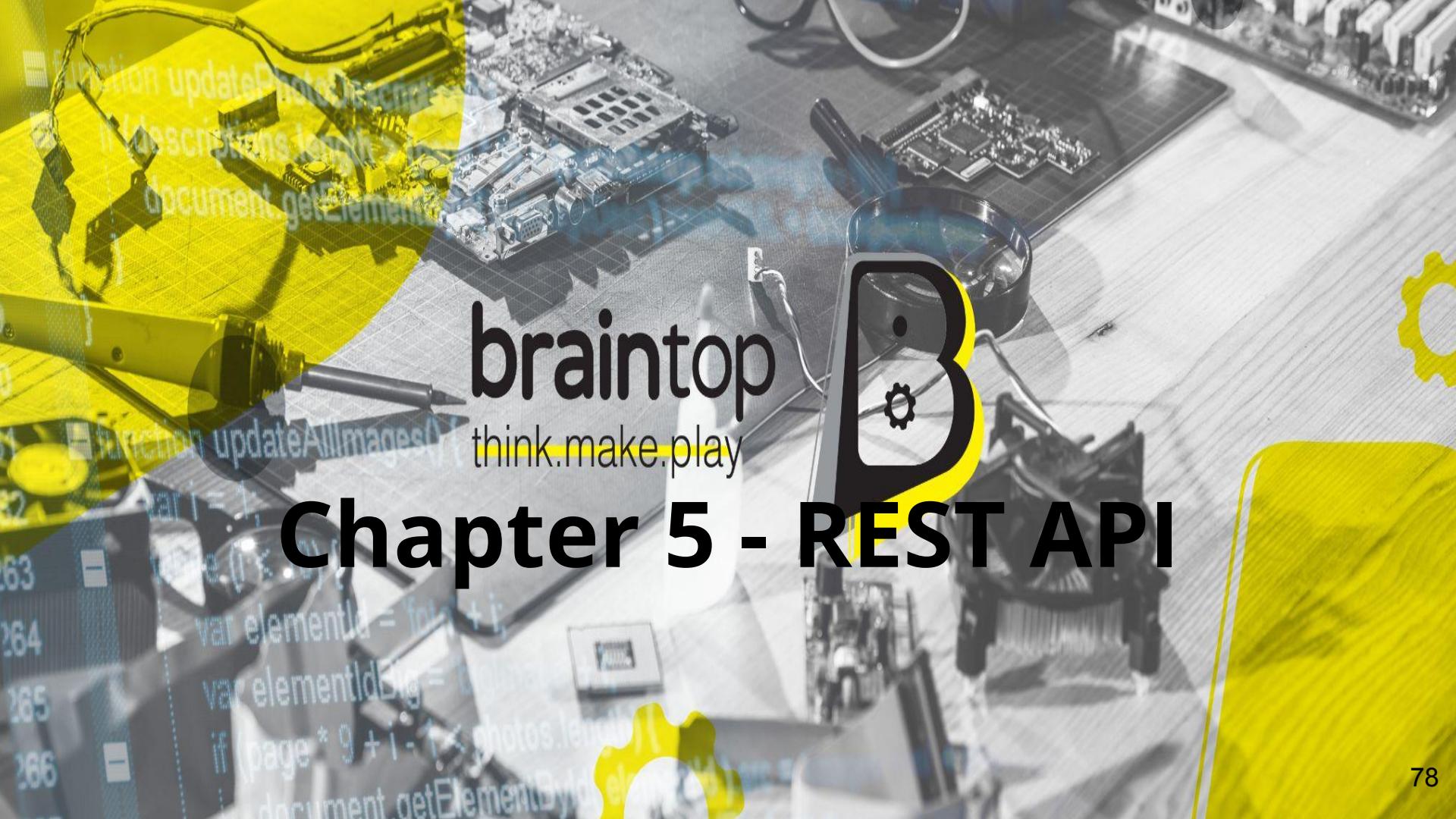
http.createServer(function(req, res) {
  fs.readFile("./lib/first.html", function(err, data) {
    res.writeHead(200, {
      'Content-Type': 'text/html'
    });
    res.write(data);
    res.end();
  })
}).listen(3000)
```

Read the file
'first.html' under 'lib'

Create a response



hello!



braintop
think.make.play

Chapter 5 - REST API

What is REST API?

REpresentational State Transfer (REST) is a standard for a software architecture for applications that typically use multiple web services.

In order to be used in a REST-based application, a web service needs to meet certain constraints. Such a web service is called RESTful.

In this chapter we will go through these standards, and understand how we can use RESTful web services in our own application.

REST API

HTTP-based RESTful APIs are defined with the following aspects:

- A base URI, such as `http://www.example.com/`;
- A standard HTTP methods (e.g., GET, POST, PUT, and DELETE);
- A media type that defines state transition data elements.

Semantics of HTTP methods

This is how HTTP methods are intended to be used in HTTP APIs:

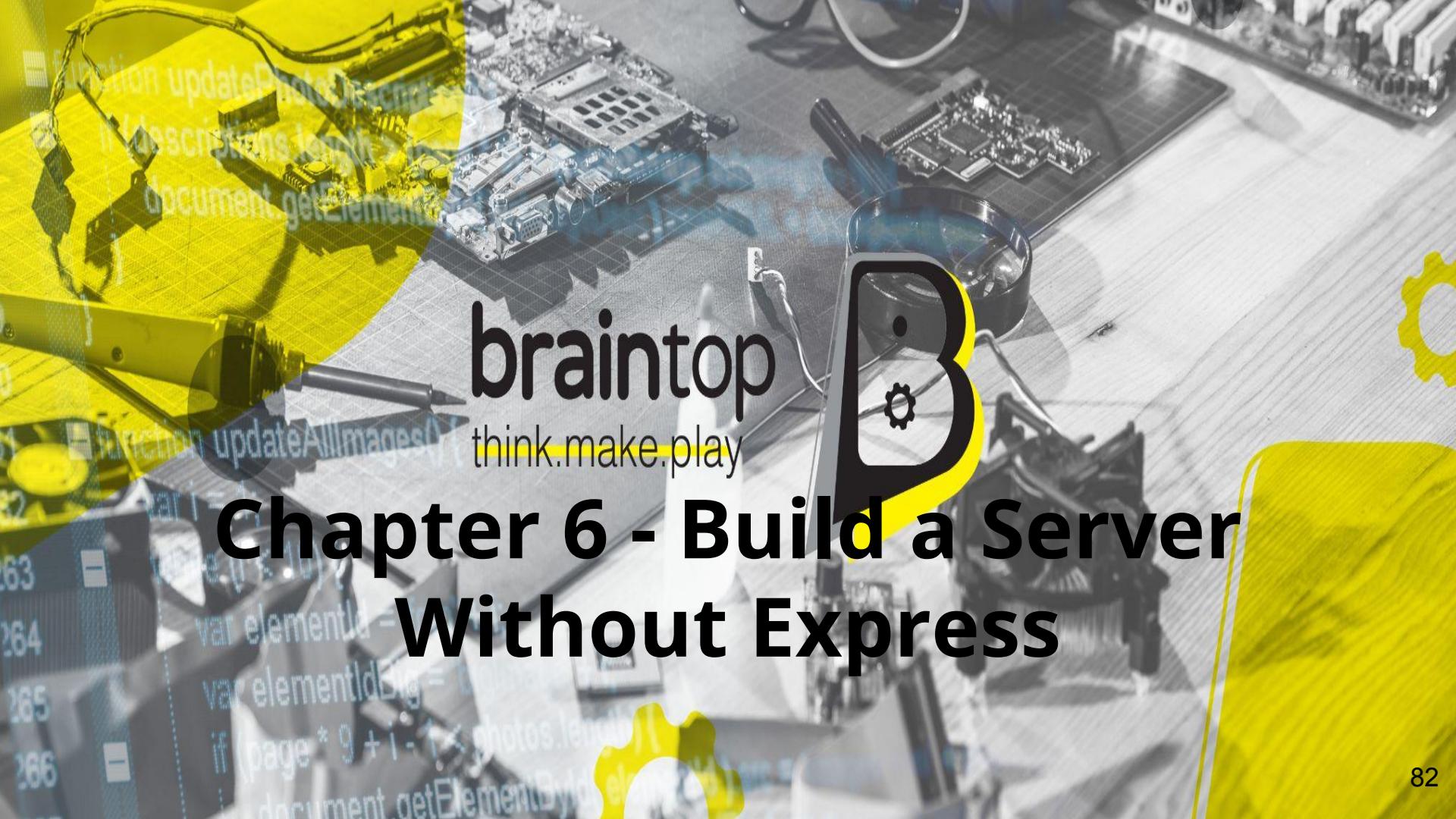
GET - Get a representation of the target resource's state (read only method).

POST - Let the target resource process the representation enclosed in the request.

PUT - Set the target resource's state to the state defined by the representation enclosed in the request.

DELETE - Delete the target resource's state.

- Bonus - Read about the Six guiding constraints that define a RESTful system.



braintop
think.make.play



Chapter 6 - Build a Server Without Express

Sources for this chapter

Folder - chap6-server-without-express

Reading file content

chap6-server-without-express/part1-teacher-demo/index1.js

```
var fs = require('fs');
var http = require('http');
http.createServer(function(req, res) {
  const baseURL = req.protocol + '://'+ req.headers.host + '/';
  const reqUrl = new URL(req.url,baseURL);
  let path = reqUrl.pathname
  if(path === "/")
    res.end("client asked for home page");
  else if(path === "/product")
    res.end("client asked for products");
  else if(path === "/persons")
    res.end("client asked for persons");
  else if(path === "/api"){
    fs.readFile(`$__dirname}/data/data.json`, 'utf-8', (err, data)=>{
      const productData = JSON.parse(data) ←
      console.log(productData)
      res.writeHead(200, {'Content-Type': 'application/json'});
      res.end(data);
    })
  }
  else {
    res.writeHead(404, {
      'Content-Type': 'text/html'
    });
    res.end("<h1>Page not found</h1>");
  }
}).listen(3000);
```

```
data.json
[
  {
    "title": "Yellow Ball",
    "description": "This is yellow 🟡",
    "image": "🟡"
  }
]
```

Reading the data from file

localhost:3000/api

```
[
  {
    "title": "Yellow Ball",
    "description": "This is yellow 🟡",
    "image": "🟡"
  }
]
```

Reading file content

chap6-server-without-express/part1-teacher-demo/index2.js

```
var fs = require('fs');
var http = require('http');
let pageData = fs.readFileSync(`$__dirname__/data/data.json`, 'utf-8'); Read the content only once
let pageobjData = JSON.parse(pageData);
console.log(pageobjData);
http.createServer(function(req, res) {
  const baseURL = req.protocol + '://' + req.headers.host + '/';
  const reqUrl = new URL(req.url, baseURL);
  let path = reqUrl.pathname
  if(path === "/") {
    res.end("client asked for home page");
  } else if(path === "/product") {
    res.end("client asked for products");
  } else if(path === "/persons") {
    res.end("client asked for persons");
  } else if(path === "/api") {
    res.writeHead(200, {'Content-Type': 'application/json'});
    res.end(products);
  } else {
    res.writeHead(404, {'Content-Type': 'text/html'});
    res.end("<h1>Page not found</h1>");
  }
}).listen(3000);
```

```
[  
  {  
    title: 'Yellow Ball',  
    description: 'This is yellow 🟡',  
    image: '🟡'  
  }  
]
```

Html file

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport"
content="width=device-width, initial-scale=1.0"
/>
    <meta http-equiv="X-UA-Compatible"
content="ie=edge" />
    <title>Products</title>
    <style>
      *
      {
        margin: 0;
        padding: 0;
        box-sizing: inherit;
      }

      html {
        box-sizing: border-box;
    }
```

```
      body {
        background:tomato;
      }
      .container{
        padding:20px;
      }
      .cards{
        padding: 5px;
      }
      .card{
        border: 5px solid royalblue;
        width: 25%;
        border-radius: 10px;
        padding: 5px;
        font-size: 20px;
        margin:10px 10px;
        float: left;
      }
    </style>
  </head>
```

Html file

chap6-server-without-express/part1-demo/templates/page.html

```
<body>
  <div class="container">
    <h1>⚡ { %TITLE% } </h1>
    <div>
      <p>Description : { %DESCRIPTION% } </p>
      <p>Image : { %IMAGE% } </p>
    </div>
  </div>
</body>
</html>
```

You can use any sign. (It is not mandatory to use the% sign)



The screenshot shows a red rectangular area containing the rendered HTML output. It includes a speaker icon, the placeholder text '{ %TITLE% }' followed by a ghost emoji, a description paragraph, and an image paragraph.

⚡ { %TITLE% } 🕸️

Description : { %DESCRIPTION% }

Image : { %IMAGE% }

replaceTemplate function

```
function replaceTemplate(template, data){  
  let output = template.replace(/{%TITLE%}/g,data.title)  
  output = output.replace(/{%DESCRIPTION%}/g,data.description)  
  output = output.replace(/{%IMAGE%}/g,data.image)  
  return output  
}
```

Template is the content of the html file

Data is the data.json content

We will replace the **TITLE**, **DESCRIPTION** and **IMAGE** with the content from the file

data[0].title, data[0].description, data[0].image

Reading html content

```
let templatePage = fs.readFileSync(`${__dirname}/templates/page.html`, 'utf-8');
```

Reading file content

chap6-server-without-express/part1-demo/templates/index3.html

```
var fs = require('fs');
var http = require('http');
let pageData = fs.readFileSync(`$__dirname__/data/data.json`, 'utf-8'); ← Read the content only once
let pageobjData = JSON.parse(pageData);
function replaceTemplate(template, data) {
  let output = template.replace(/%\{TITLE%\}/g, data.title)
  output = output.replace(/%\{DESCRIPTION%\}/g, data.description)
  output = output.replace(/%\{IMAGE%\}/g, data.image)
  return output
}
let templatePage = fs.readFileSync(`$__dirname__/templates/page.html`, 'utf-8');
http.createServer(function(req, res) {
  const baseURL = req.protocol + '://' + req.headers.host + '/';
  const reqUrl = new URL(req.url, baseURL);
  let path = reqUrl.pathname
  if(path === "/") {
    res.writeHead(200, {'Content-Type': 'text/html'});
    const output = replaceTemplate(templatePage, pageobjData[0]) ← Replace the contents of the html
    res.end(output); file with the json file
  }
})
```

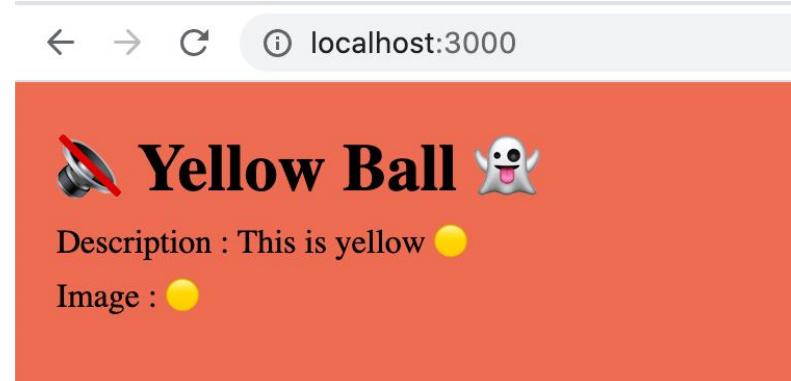
Continue Next page

```
[ {  
  title: "Yellow Ball",  
  description: "This is yellow 🟡",  
  image: '🟡'  
}]
```

Reading file content

```
else {
  res.writeHead(404, {
    'Content-Type': 'text/html'
  });
  res.end("<h1>Page not found</h1>");
}

}).listen(3000);
```



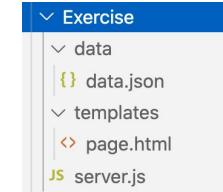
Do it yourself 1

Add this file to data folder

1. Create a folder with 2 directories
 - a. data that contain data.json

```
[  
 {  
   "product": "Heart",  
   "Description": "Pink Heart",  
   "price": "50",  
   "image": "\ud83d\udcbb"  
 }  
 ]
```

- b. templates - contain page.html - next slides content of page.html
2. Add to the folder file named server.js



page.html

Add this file to templates folder

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport"
content="width=device-width, initial-scale=1.0"
/>
    <meta http-equiv="X-UA-Compatible"
content="ie=edge" />
    <title>Products</title>
    <style>
      *
      {
        margin: 0;
        padding: 0;
        box-sizing: inherit;
      }

      html {
        box-sizing: border-box;
      }
    </style>
  </head>
```



```
  body {
    background:hsl(110, 100%, 64%);
  }
  .container{
    padding:20px;
  }
  .cards{
    padding: 5px;
  }
  .card{
    border: 5px solid royalblue;
    width: 25%;
    border-radius: 10px;
    padding: 5px;
    font-size: 20px;
    margin:10px 10px;
    float: left;
  }
</style>
</head>
```

page.html continue

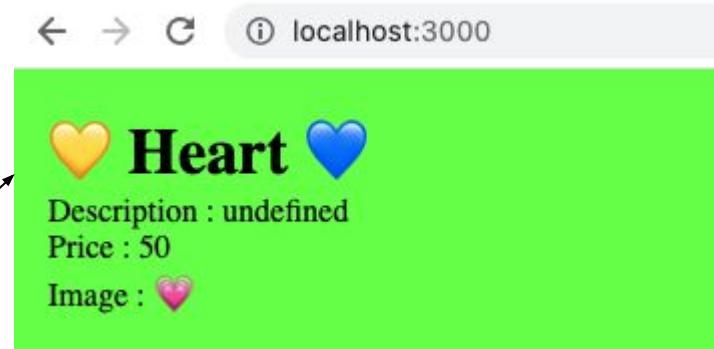
```
<body>
  <div class="container">
    <h1>💛 { %PRODUCT% } 💙 </h1>
    <div>
      <p>Description : { %DESCRIPTION% } </p>
      <p>Price : { %PRICE% } </p>
      <p>Image : { %IMAGE% } </p>
    </div>
  </div>
</body>
</html>
```

You can use any sign. (It is not mandatory to use the% sign)



Exercise continue

Create a server that listens to port 3000 and responds with the relevant data.json.



response

Build a Server Without Express good to know

In all the examples we saw, the user received the same response for every url he entered.

Let's build a server that we can control the pages that the user receives.

First, under the dir 'chap 4' create a dir called 'public', and inside a file called 'index.html' and give it some HTML content:

Build a Server Without Express

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>My Server</title>
</head>

<body>
  <h1>This is my server!</h1>
  And here is some more content
</body>

</html>
```

Build a Server Without Express

And under 'chap6' create a file called 'server-only-index.js':

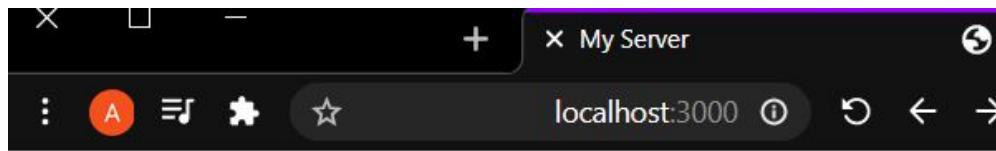
```
var http = require("http");
var fs = require("fs");
http.createServer(function(req, res) {
  console.log(` ${req.method} request for ${req.url}`);
  if (req.url === "/") {
    fs.readFile("./public/index.html", "UTF-8", function(err, html) {
      res.writeHead(200, {
        "Content-Type": "text/html"
      });
      res.end(html);
    });
  } else {
    res.writeHead(404, {"Content-Type": "text/plain"});
    res.end("404 file not found");
  }
}).listen(3000);
```

req.method indicates the REST method the user used to get to the data (GET, POST...)

For the url '/', which is the default, respond with 'index.html'. Else respond with 404 status code, which means 'page not found'.

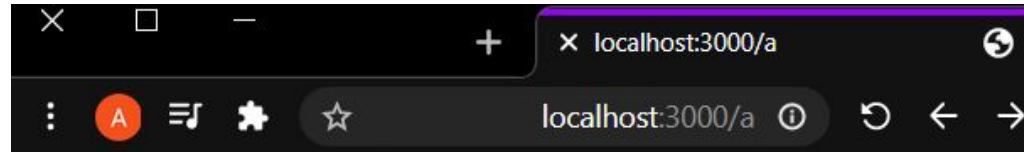
This way we can also respond to every url differently

Build a Server Without Express



This is my server!

And here is some more content



404 file not found

Explaining the Rendering process

Usually the web application also has css files which define the style for the html web pages.

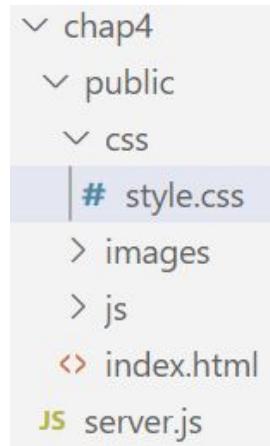
Under the directory 'public', create the following directories:

images

css

js

Under the css lib create a css file called 'style'



Explaining the Rendering process

In style.css:

```
h1 {  
    color: green;  
}
```

And add a link to style.css in index.html:

In the <head> element:

```
<link rel="stylesheet" href="./css/style.css">
```

Explaining the Rendering process

Go to localhost:3000/ and look in the logs:

```
GET request for /
GET request for /css/style.css
```

But who requested '/css/style.css'? We only requested '/'!

So the answer is that for every link we add to our web page, another request sent.

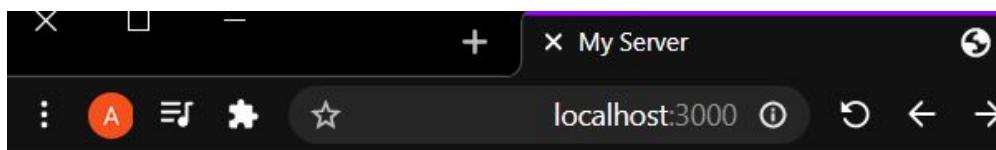
Try adding links to images or other css files and look what happens!

Explaining the Rendering process

Can you see the problem?

We only handle the requests for '/' in server.js. So the client won't get the css file!

The page will stay the same:



This is my server!

And here is some more content

Handling CSS Files & Images

So let's handle css files, so the user will get the page style.

```
var http = require("http");
var fs = require("fs");
var path = require("path");

http.createServer(function(req, res) {
  console.log(` ${req.method} request for ${req.url} `);

  if (req.url === "/") {
    fs.readFile("./public/index.html", "UTF-8", function(err, html) {
      res.writeHead(200, {
        "Content-Type": "text/html"
      });
      res.end(html);
    });
  }
}

// continues in the next slide
```

Same as we saw earlier

Handling CSS Files & Images

```
else if (req.url.match(/\.css$/)) {  
    var cssPath = path.join(__dirname, 'public', req.url);  
    var fileStream = fs.createReadStream(cssPath, "UTF-8");  
  
    res.writeHead(200, {  
        "Content-Type": "text/css"  
    });  
    fileStream.pipe(res);  
}  
else if (req.url.match(/\.png$/)) {  
    var imgPath = path.join(__dirname, 'public', req.url);  
    var imgStream = fs.createReadStream(imgPath);  
  
    res.writeHead(200, {  
        "Content-Type": "image/jpeg"  
    });  
    imgStream.pipe(res);  
}  
// continues in the next slide
```

Regex is a method for checking patterns in strings. This way we check if the req.url ends with '.css'

pipe is the response for streaming

Also return images. (only .png images in this example)

Handling CSS Files & Images

```
else {
  res.writeHead(404, {
    "Content-Type": "text/plain"
  });

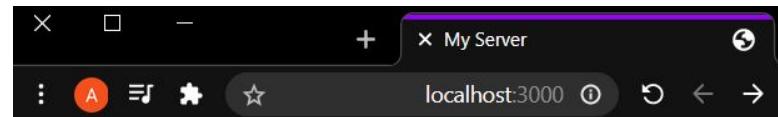
  res.end("404 file not found");
}

}).listen(3000);
```

And finally, we can see the title has color:

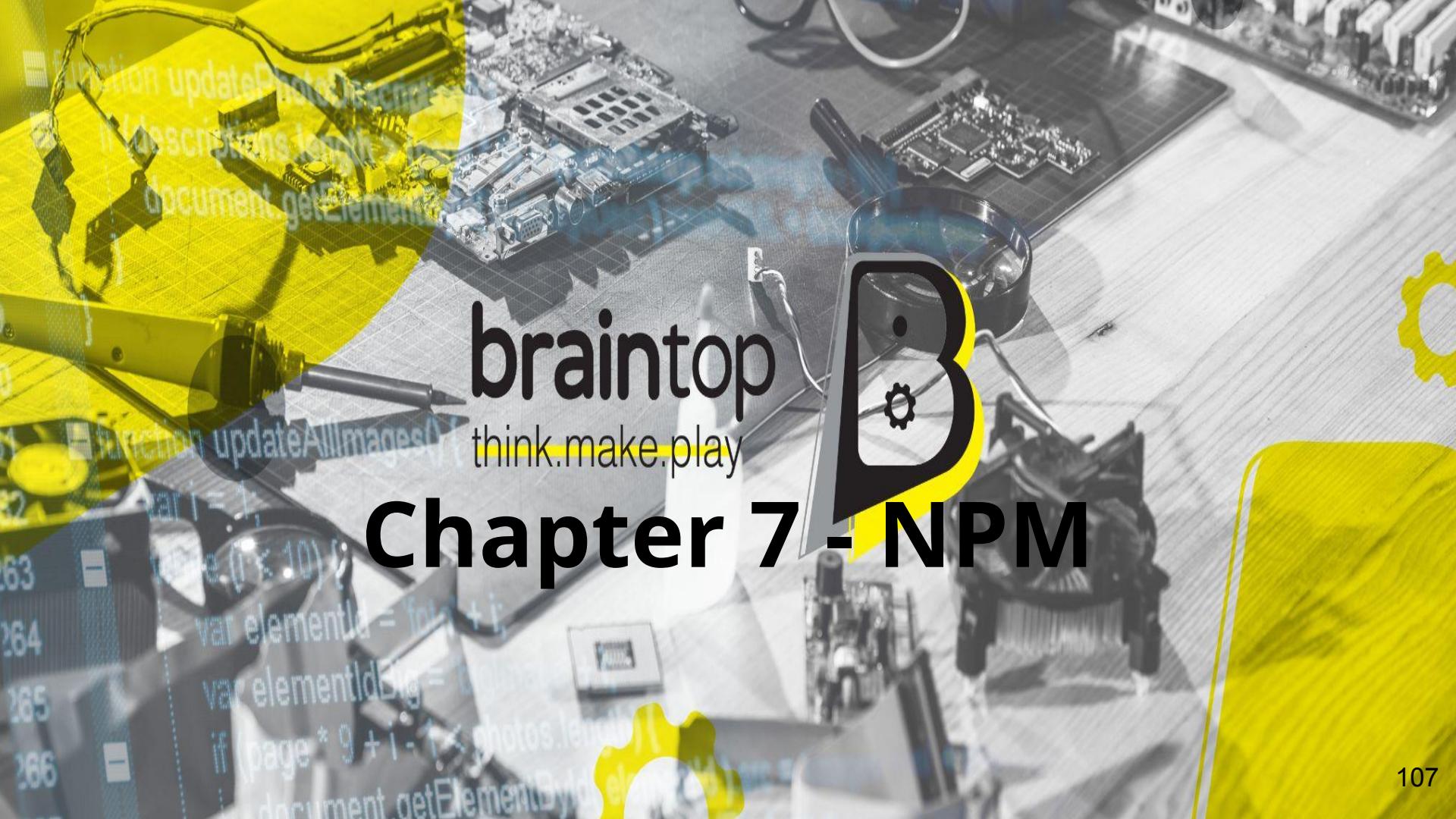
Try adding .img images too!

Same as we saw earlier



This is my server!

And here is some more content



braintop
think.make.play

Chapter 7 - NPM

What is NPM?

Node Package Manager is a tool for downloading and using packages and modules in node application.

www.npmjs.com hosts hundreds of thousands of free packages.

NPM is automatically downloaded while installing node.js

For example, in order to use the module ‘upper-case’, in the terminal enter:

npm install upper-case

And now you can use it!

What is NPM?

First run `npm init` in the terminal and initialize the relevant fields, or skip with enter.

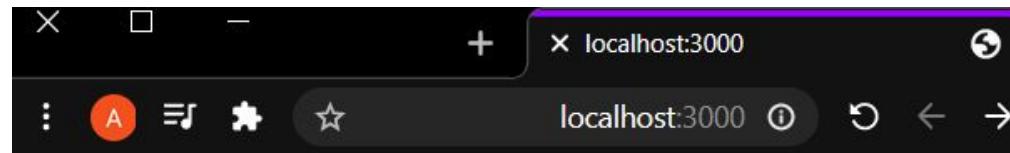
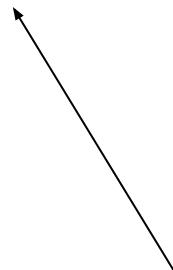
Run `npm install upper-case --save` (--save to add the relevant dependency)

We can build a program that uses it:

```
var http = require('http');
var uc = require('upper-case');

http.createServer(function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/html'
  });

  res.write(uc.upperCase("hello world!"));
  res.end();
}).listen(3000);
```





braintop
think.make.play

Chapter 8 - promises

call back in callback

index1-callback-in-callback.js

```
const fs = require('fs');
fs.readFile('./data/data.json','utf-8',(err, data)=>{
  if(err) return ("couldn't read file")
  data = JSON.parse(data)
  let product = "Ball"
  var exist = false
  data.forEach(element => {
    if(product==element.productName) exist=true;
  });
  if(exist){
    fs.writeFile('./data/data1.json','Ball Exist',err=>{
      if(err) console.log(err)
    })
  }
  else{
    fs.writeFile('./data/data1.json','Ball Not Exist',err=>{
      if(err) console.log(err)
    })
  }
})
```

cnapo-server-without-express > promises > data > 11 data.json

1 Ball Exist

Callback in callback - not good solution

data.json

data/data.json

```
[  
  {  
    "id": 0,  
    "productName": "Ball",  
    "image": "🟡",  
    "from": "Usa",  
    "quantity": "48 🟡",  
    "price": "8",  
    "description": "Jumping yellow ball"  
  },  
  {  
    "id": 0,  
    "productName": "Heart",  
    "image": "❤️",  
    "from": "London",  
    "quantity": "45",  
    "price": "10",  
    "description": "Orange Heart"  
  }]  
]
```

The solution is promises

index2-promise.js

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

let readFileData = file=>{
    return new Promise((resolve, reject)=>{
        fs.readFile(file, (err, data)=>{
            if(err) reject(err)
            resolve(data)
        })
    })
}

let writeDataToFile = (file, data)=>{
    return new Promise((resolve, reject)=>{
        fs.writeFile(file,data, err=>{
            if(err) reject(err)
            resolve(data)
        })
    })
}

//Continue next page
```

promises

```
readFileData('./data/data1.json')
.then(data=>{
    data=JSON.parse(data)
    for(let i=0;i<data.length;i++) {
        if(data[i].productName=="Ball")
            return data[i].productName
    }
    return false
})
.then(productName=>{
    if(productName!=false)
        return writeDataToFile('./data/data2.json','product ${productName} saved')
    else
        return writeDataToFile('./data/data2.json','product Not exists')
})
.then(()=>{
    console.log("data saved to file")
})
.catch(err=>{
    console.log(err)
})
```

Async and await

index3-await-async.js

```
const fs = require('fs');
let readFileData = (file)=>{
    return new Promise((resolve, reject)=>{
        fs.readFile(file, (err, data)=>{
            if(err) reject(err)
            resolve(data)
        })
    })
}

let writeDataToFile = (file, data)=>{
    return new Promise((resolve, reject)=>{
        fs.writeFile(file,data, err=>{
            if(err) reject(err)
            resolve(data)
        })
    })
}

}//Continue next page
```

Async and await

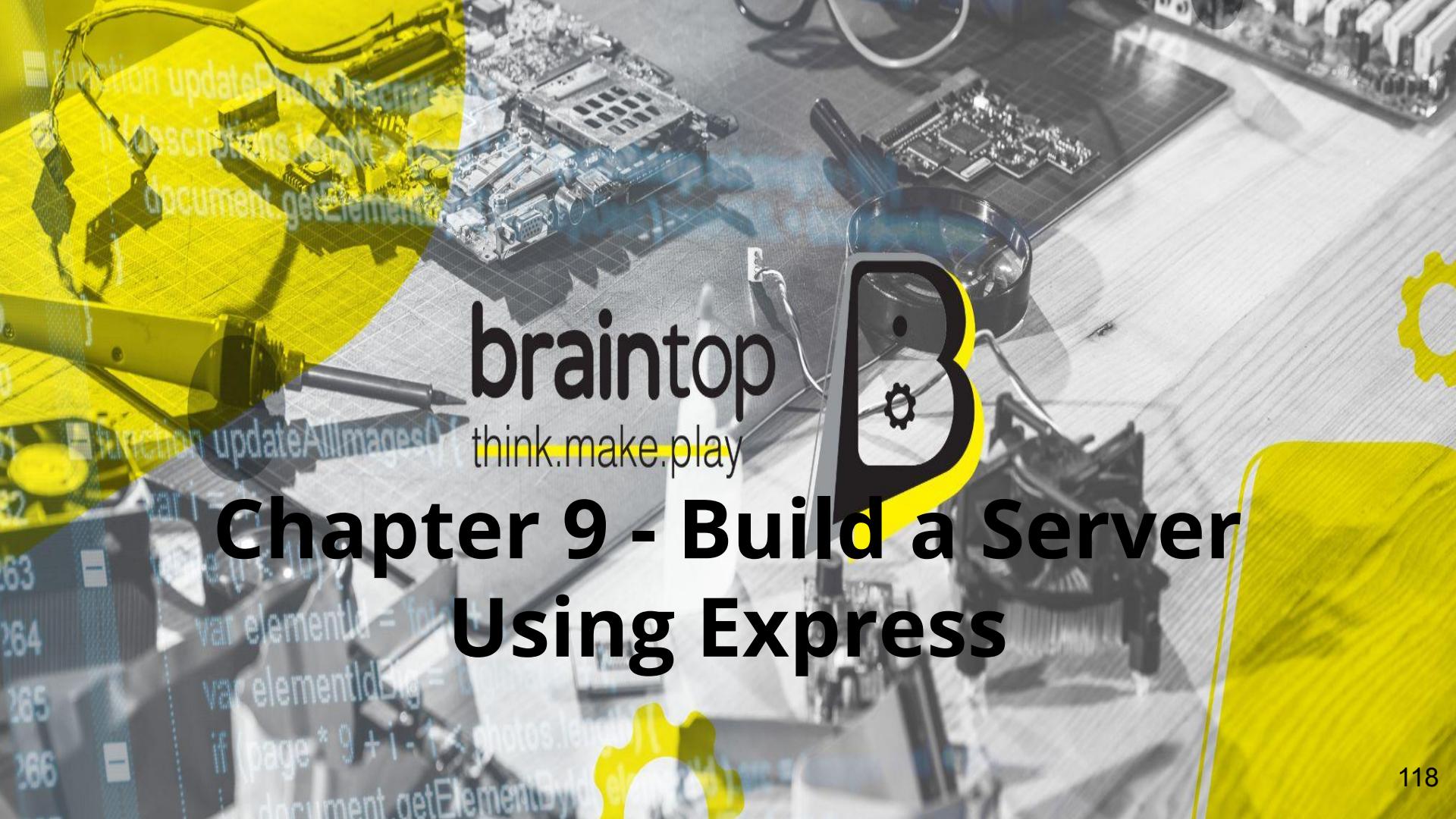
```
let writingDataToFile = async () => {
  try{
    let data = await readfileData('./data/data.json')
    data=JSON.parse(data)
    productName=false
    for(let i=0;i<data.length;i++){
      if(data[i].productName=="Ball"){
        productName= data[i].productName
      }
    let answer="";
    if(productName!=false)
      answer= await writeDataToFile('./data/data2.json', `product ${productName}
saved`)
    else
      answer= await writeDataToFile('./data/data2.json', `product Not exists`)
    return answer;//product ball saved
  }
  catch(e){
    console.log(e)
  }
}//Continue next page
```

Async and await

```
writingDataToFile()

.then(x => {
    console.log(x); //product Ball saved
    console.log('data saved to file!');
})

.catch(err => {
    console.log('ERROR');
}) ;
```



braintop
think.make.play

Chapter 9 - Build a Server Using Express

Before we start download Postman - Signing Up

Folder : chap9-express-introduction

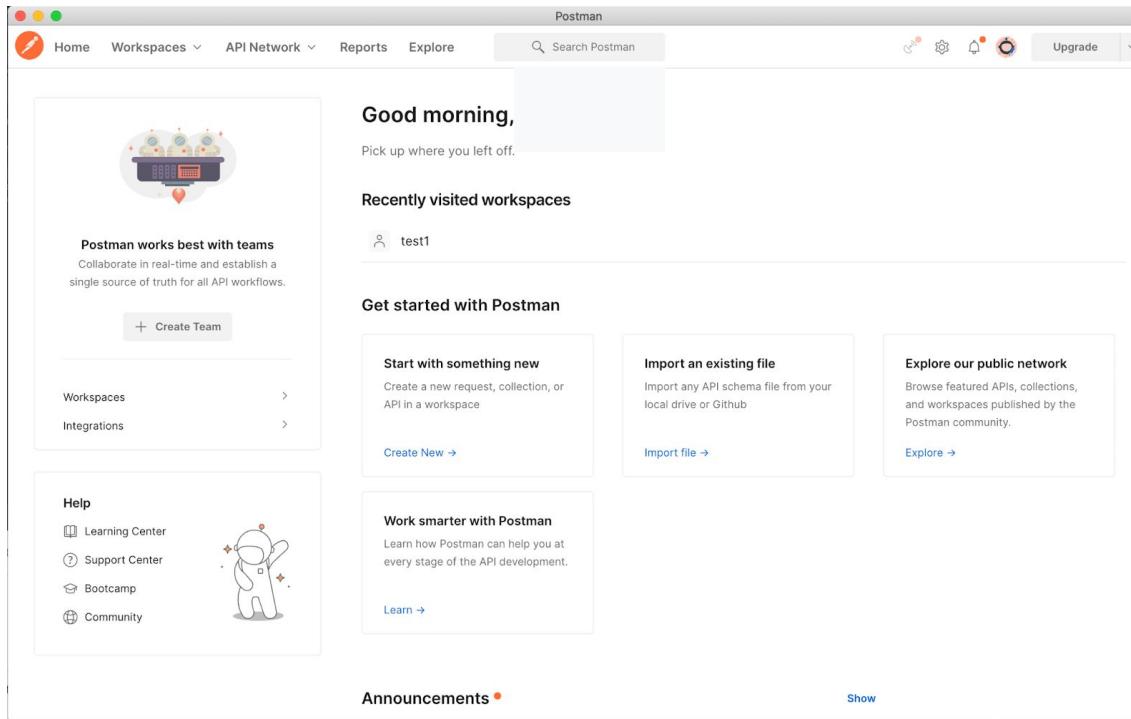
Postman is an API client that allows users to easily create and save simple and complex HTTP/s requests, as well as read their responses.

<https://www.postman.com/>

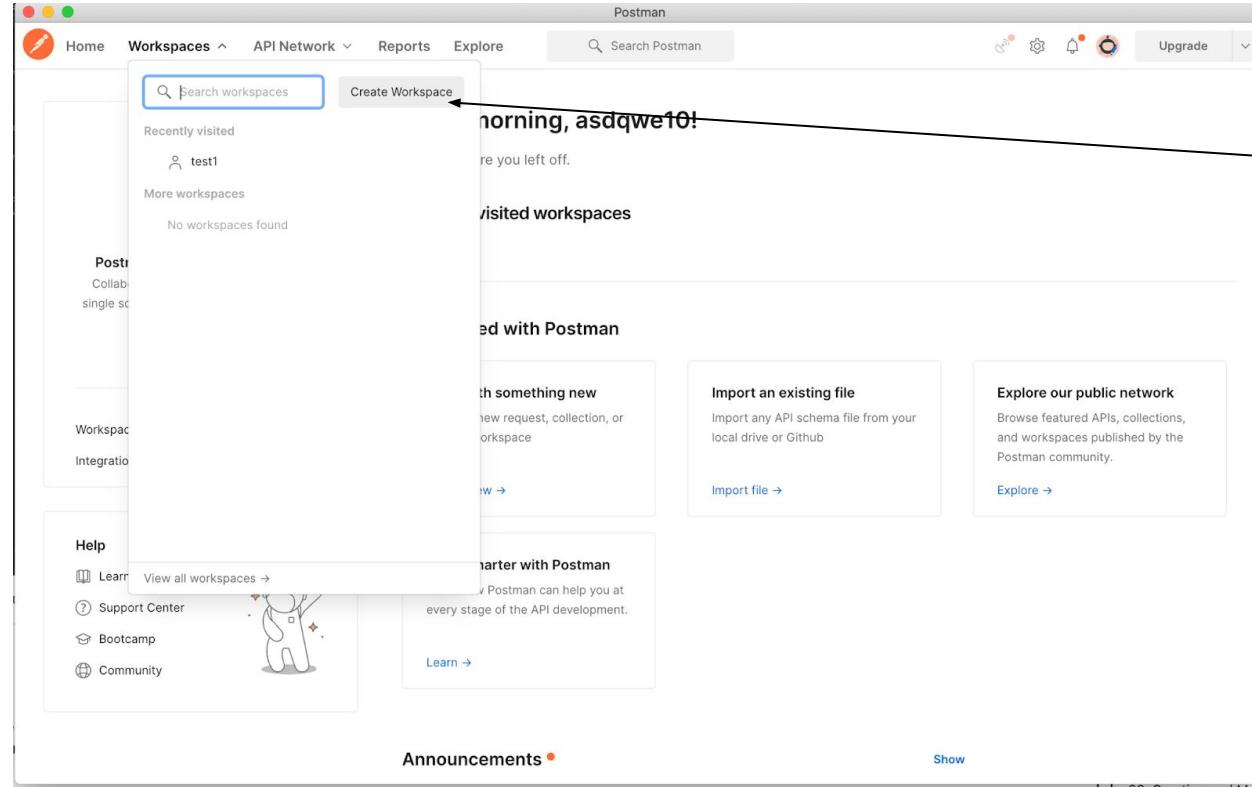
Download the application:

<https://www.postman.com/downloads/>

Before we start download Postman - Signing Up

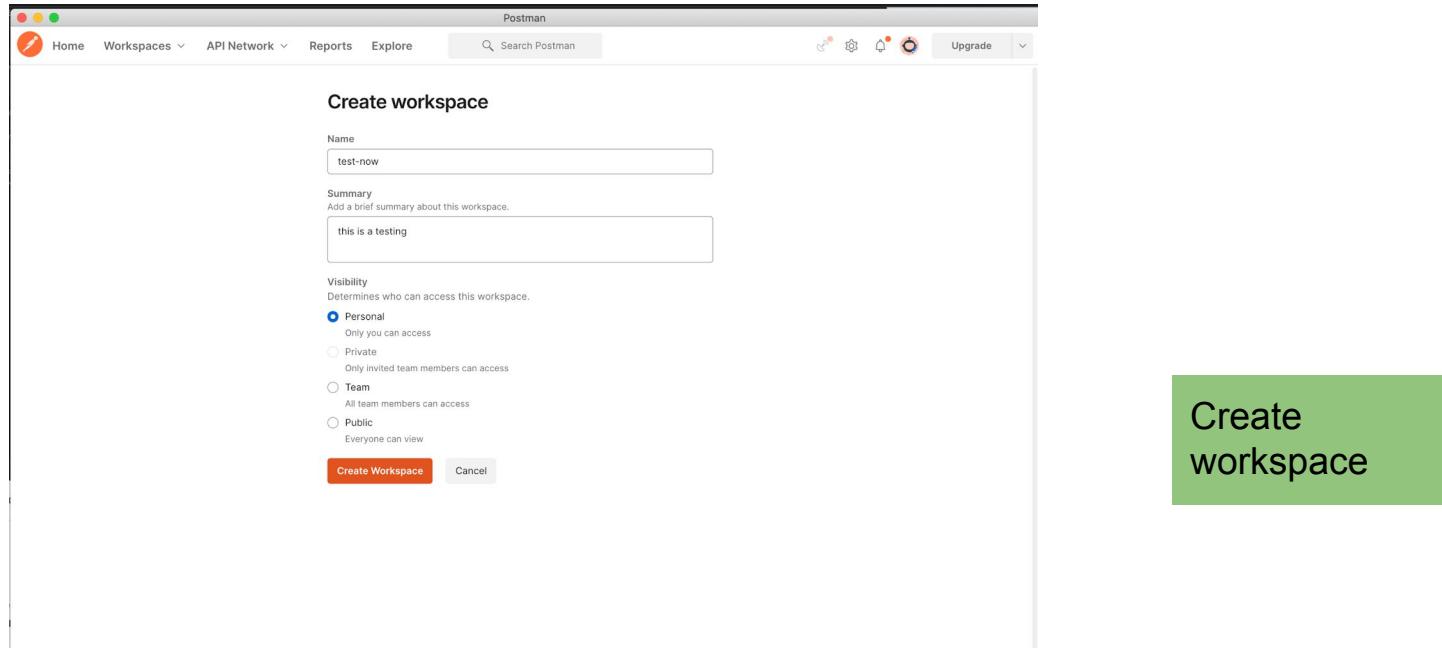


Before we start download Postman - Signing Up



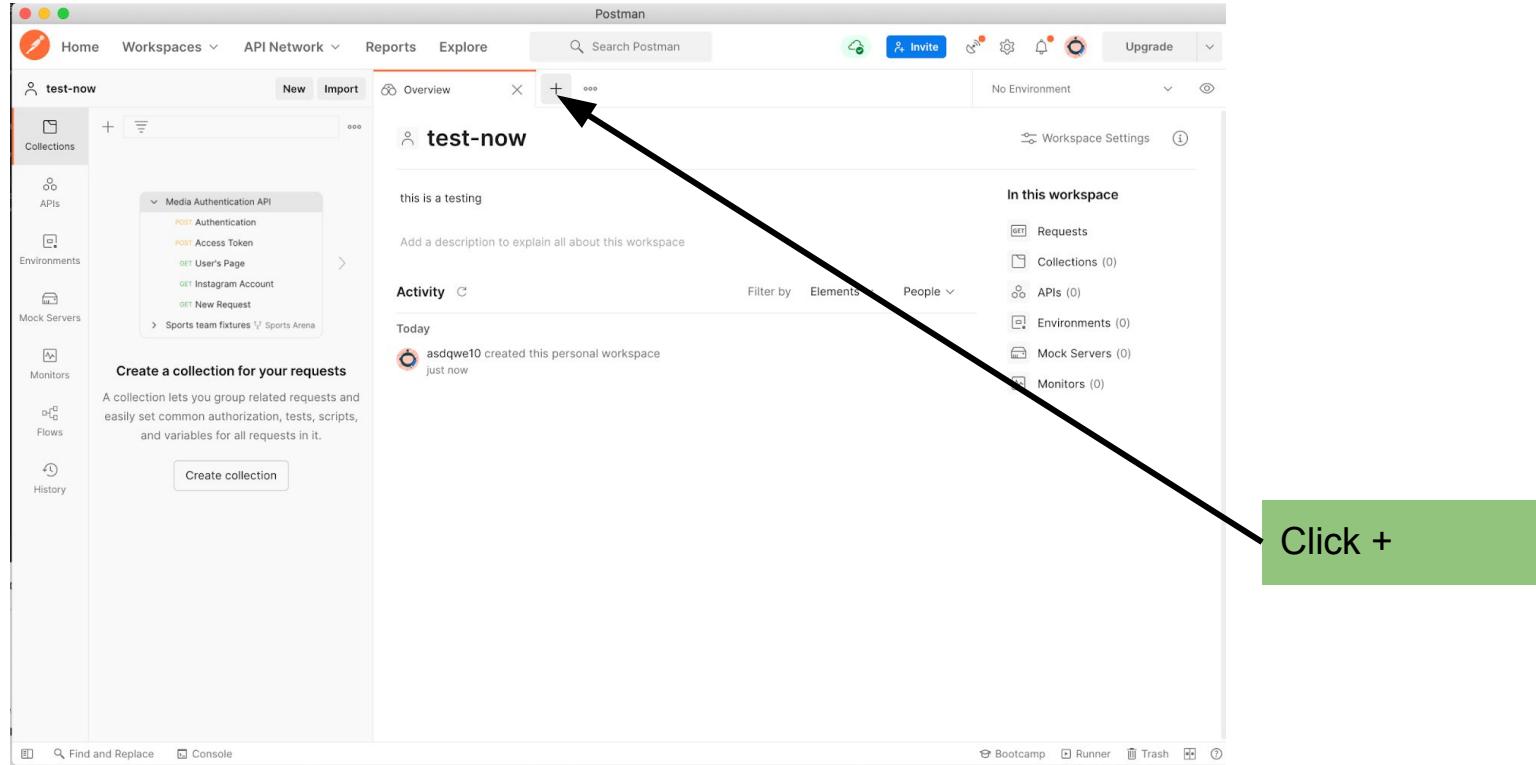
Create workspace

Before we start download Postman - Signing Up



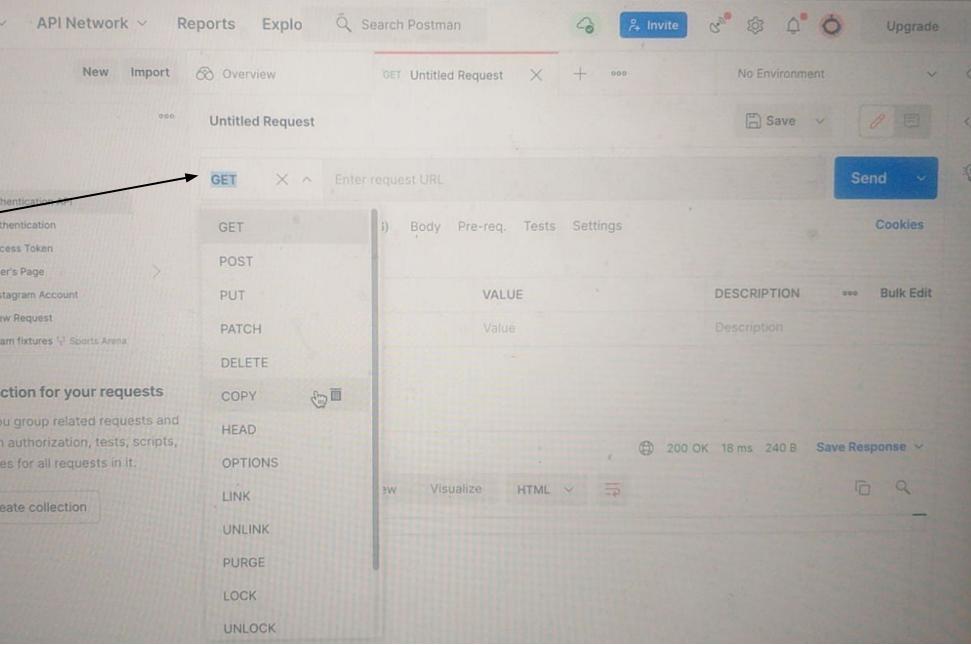
Create
workspace

Before we start download Postman - Signing Up



Before we start download Postman - Signing Up

Request :
get, post,
etc...

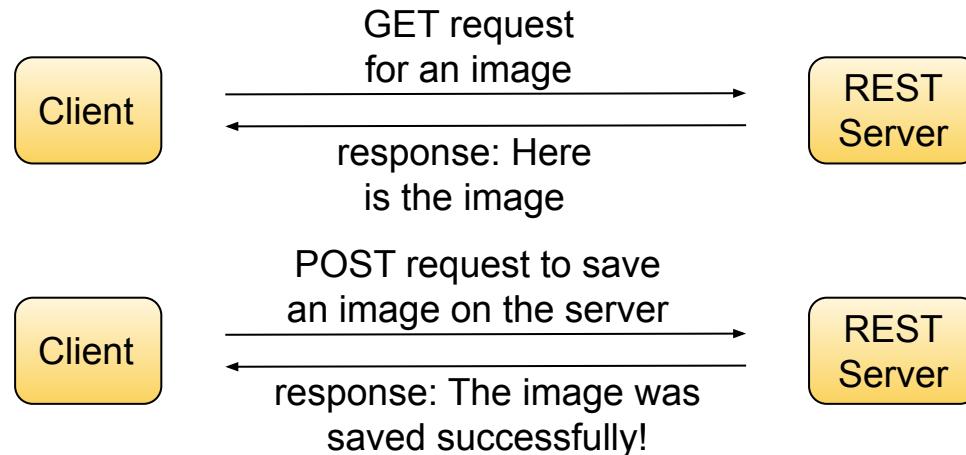


The screenshot shows the Postman application interface. On the left, there is a sidebar with a tree view of API collections, including 'Instagram Account' and 'Sports Arena'. The main area is titled 'Untitled Request' and shows a 'GET' method selected. Below the method, there are several other options: POST, PUT, PATCH, DELETE, COPY, HEAD, OPTIONS, LINK, UNLINK, PURGE, LOCK, and UNLOCK. To the right of these methods, there are columns for 'Body', 'Pre-req.', 'Tests', 'Settings', 'Value', 'Description', and 'Bulk Edit'. At the bottom of the request panel, there is a status bar showing '200 OK 18 ms 240 B' and a 'Save Response' button. The top navigation bar includes 'API Network', 'Reports', 'Explore', a search bar, and various account and settings icons.

What Is Express?

Express is a node module that allows us to building routing more simply.

We can build a server that listens to REST requests.



First app with express

Run **npm init** in the terminal, and then **npm install express**.

In **server.js**:

```
var express = require("express");
var app = express();
var port = 3000;
app.get('/', function(req, res) {
    res.status(200).send('hello express');
});
app.listen(port);
```

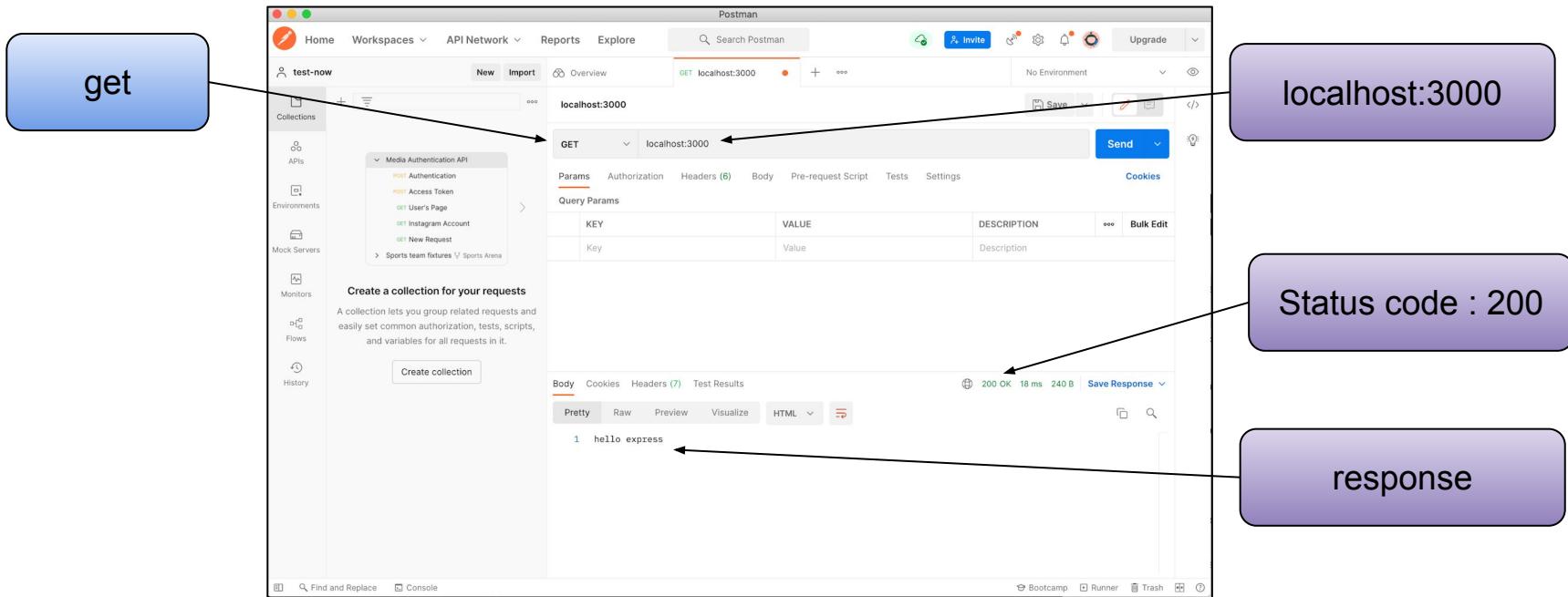
Initialize a variable to use
the express methods

Listen to get requests
with the url '/'

Status code

The response can
also be a json

Check it on postman



post request

post

Our app
doesn't
support post
request

The screenshot shows the Postman application interface. On the left, there's a sidebar with various sections like Home, Workspaces, API Network, Reports, Explore, Collections, APIs, Environments, Mock Servers, Monitors, Flows, and History. A green box labeled "post" is overlaid on the top-left area. Another green box labeled "Our app doesn't support post request" is overlaid on the bottom-left area. On the right, the main workspace shows a POST request to "localhost:3000". The "Headers" tab is selected, showing a single header "Content-Type: application/json". The "Body" tab contains a JSON payload: {"key": "value"}. The "Tests" tab has a script: `if (response.status === 200) { postman.setResponseHeader('x-test', 'success'); }`. The "Settings" tab has a "Timeout" set to 10s. Below the request details, a "Body" tab is open, showing the raw response content. The response is a simple HTML page with the following code:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>Error</title>
6 </head>
7 <body>
8   <pre>Cannot POST /</pre>
9 </body>
10 </html>
```

The status bar at the bottom indicates a 404 Not Found error with a 6 ms duration and 412 B size. A "Save Response" button is also present. A green box labeled "404" is overlaid on the bottom-right corner of the response preview area.

Now our app support post request

```
var express = require("express");
var app = express();
var port = 3000;
app.get('/', function(req, res) {
  res.status(200).send('hello express');
});
app.post('/', function(req, res) {
  res.status(200).send('we support post request');
});
app.listen(port);
```

Listen to **get** requests
with the url '/'

Listen to **post** requests
with the url '/'

Now our app support post request

The screenshot shows the Postman application interface. On the left, the sidebar includes 'Collections' (selected), 'APIs', 'Environments', 'Mock Servers', 'Monitors', 'Flows', and 'History'. A 'Create a collection for your requests' section with a 'Create collection' button is also present. The main workspace shows a 'localhost:3000' collection with a 'POST' request to 'localhost:3000'. The 'Headers' tab is selected, showing a 'Content-Type' header. Below it, the 'Body' tab displays the response body: '1 we support post request'. The status bar at the bottom indicates a 200 OK response with 17 ms latency and 251 B size.

Support post req

Res json example

```
var express = require("express");
var app = express();
var port = 3000;
app.get('/', function(req, res) {
    res.send('<html><head></head><body>Hello world</body></html>');
});
app.get('/api', function(req, res) {
    res.json({
        firstname: "asaf",
        lastname: "amir"
    });
});
app.listen(port);
```

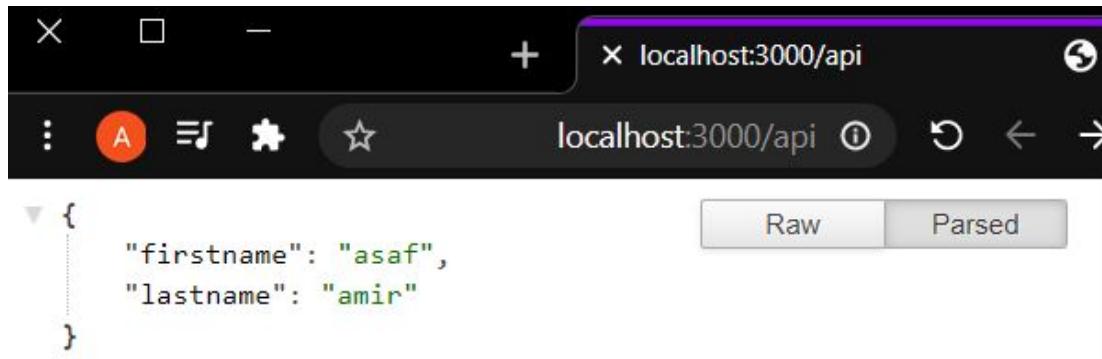
Initialize a variable to use the express methods

Listen to get requests with the url '/'

Listen to get requests with the url '/api'

The response can also be a json

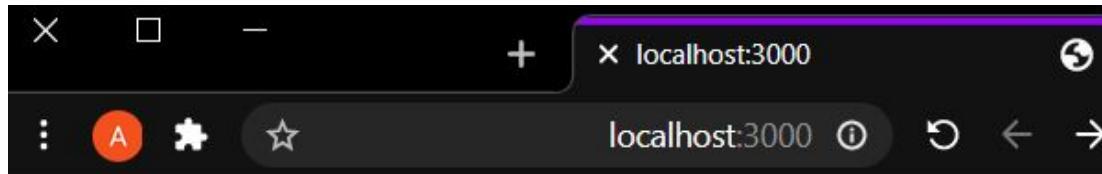
Build A Server Using Express



A screenshot of a browser window showing a JSON response. The URL in the address bar is `localhost:3000/api`. The content of the page is a JSON object:

```
{ "firstname": "asaf", "lastname": "amir" }
```

There are two buttons at the top right: "Raw" and "Parsed".



A screenshot of a browser window showing the string "Hello world". The URL in the address bar is `localhost:3000`.

Hello world

Do it yourself 1 - Build A Server Using Express

Create a server that responds to the following requests:

Request	Response	Method
<i>localhost:3000/api/name</i>	Your full name	get
<i>localhost:3000/students/number</i>	Random Number between 0 to 100	get
<i>localhost:3000/courses/n1ton2</i>	Random Number between 1000 to 2000	Post

Express - req.params

We can access the parameters that were sent with the request.

Example is in the next slide

Express - req.params

```
var express = require("express");
var app = express();
var port = 3000;

app.get('/', function(req, res) {
  res.send('<html><head></head><body>Hello world</body></html>');
});

app.get('/api', function(req, res) {
  res.json({ firstname: "asaf", lastname: "amir" });
});

app.get('/person/:id', function(req, res) {
  res.send('<html><head></head><body>hello person:' + req.params.id + '</body></html>');
});

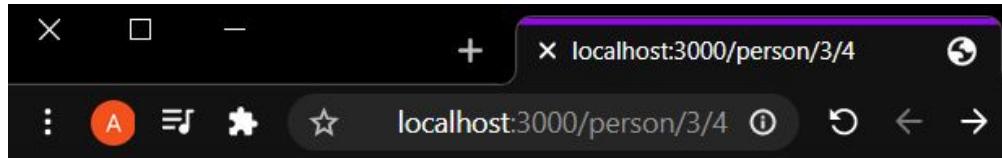
app.get('/person/:id/:page', function(req, res) {
  res.send('<html><head></head><body>hello page:' + req.params.page + '</body></html>');
});

app.listen(port);
```

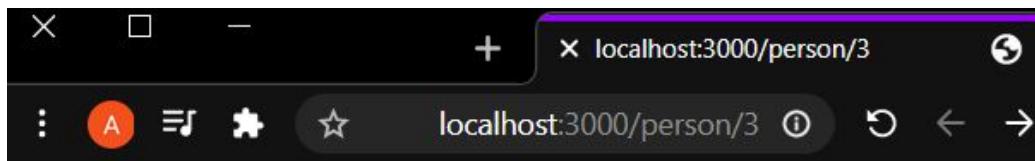
id is a parameter
that can change

There are 2 parameters in this request:
One after the second '/' and another after the third '/'

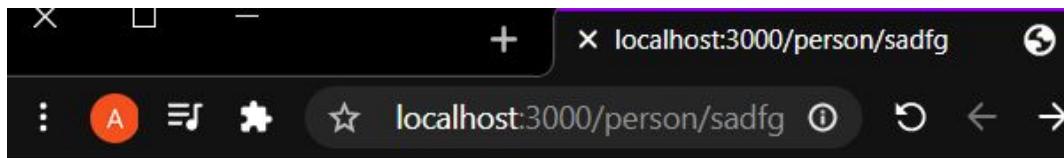
Express - req.params



hello page:4



hello person:3



hello person:sadfg

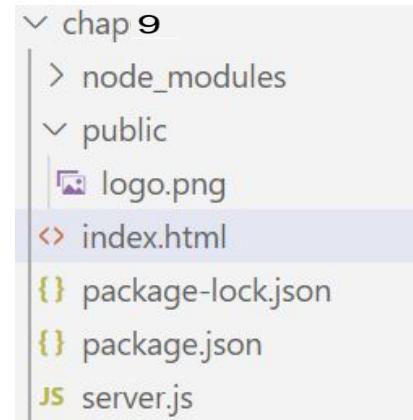
Render Static Files

Render static files from the server using express:

Create a file called ‘index.html’.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>hello</title>
</head>
<body>
  hello world
  
</body>
</html>
```

And create a folder called ‘public’, and save an image inside called ‘logo.png’



Render Static Files

Add to server.js:

```
var fs = require("fs");
app.use('/assets', express.static(__dirname + '/public'));
```

Tell express to search the static files under /public in the current directory, when receiving the /assets path

```
app.get('/firstpage', function(req, res) {
  fs.readFile("index.html", function(err, buffer) {
    var html = buffer.toString();
    res.setHeader('Content-Type', 'text/html');
    res.send(html);
  })
});
```

Save the html file as a string in a variable so we can use it later

Display the response content as an HTML text to the client

REST API ARCHITECTURE

HTTP METHODS

POST- **C**REATE

GET- **R**EAD

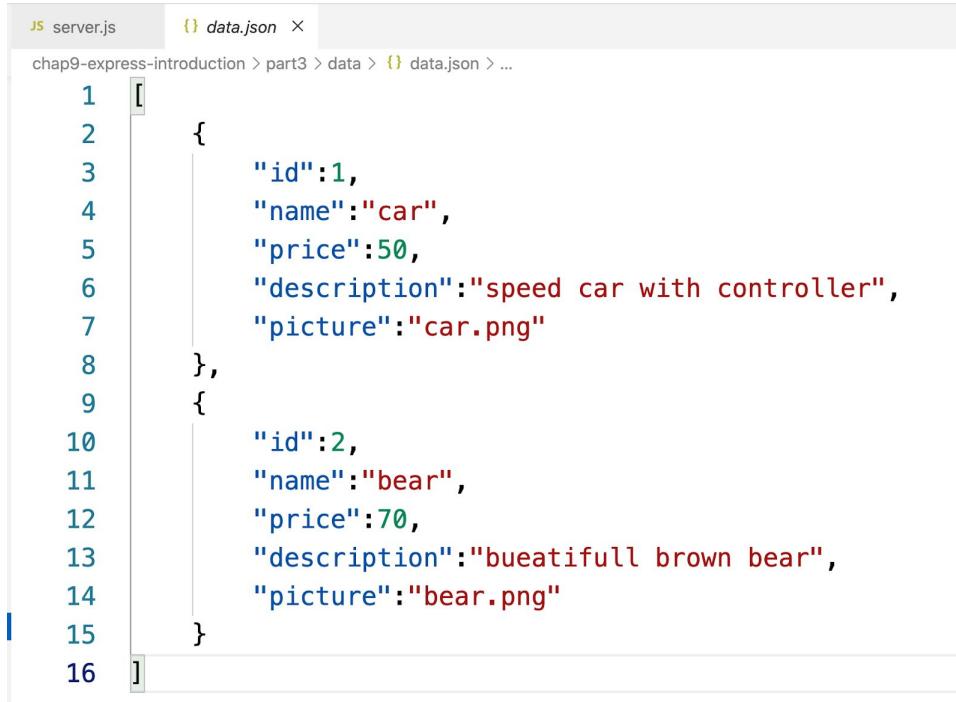
PUT- **U**PDATE

DELETE- **D**ELETE

CRUD - CREATE, READ, UPDATE, DELETE

Send data as json

Crud example



The screenshot shows a code editor with two tabs: "server.js" and "data.json". The "data.json" tab is active, displaying the following JSON data:

```
1 [  
2 {  
3     "id":1,  
4     "name":"car",  
5     "price":50,  
6     "description":"speed car with controller",  
7     "picture":"car.png"  
8 },  
9 {  
10    "id":2,  
11    "name":"bear",  
12    "price":70,  
13    "description":"bueatiful brown bear",  
14    "picture":"bear.png"  
15 }]  
16 ]
```

Crud example - get all toys

```
var express = require("express");
var fs = require("fs");
var app = express();
var port = 3000;
app.use('/assets', express.static(__dirname + '/public'));
let toys = JSON.parse(fs.readFileSync("./data/data.json", 'utf-8'))
app.get('/api/v1/toys', function(req, res) {
    res.status(200).json(
    {
        status:"success",
        data:toys
    })
});
app.listen(port);
```

localhost:3000/api/v1/toys

The screenshot shows the Postman application interface. The top navigation bar includes Home, Workspaces, API Network, Reports, Explore, a search bar, and various tool icons. The left sidebar contains sections for Collections, APIs, Environments, Mock Servers, Monitors, Flows, and History, along with a 'Create a collection for your requests' section and a 'Create collection' button. The main workspace displays a request configuration for a GET method to 'localhost:3000/api/v1/toys'. The 'Params' tab is selected, showing a single parameter 'Key' with the value 'Value'. Below the request details, the 'Body' tab is active, showing a JSON response with status 'success' and data containing two toy objects: a car and a bear.

```
1  "status": "success",
2  "data": [
3    {
4      "id": 1,
5      "name": "car",
6      "price": 50,
7      "description": "speed car with controller",
8      "picture": "car.png"
9    },
10   {
11     "id": 2,
12     "name": "bear",
13     "price": 70,
14     "description": "beautiful brown bear",
15     "picture": "bear.png"
16   }
17 ]
18
19 ]
```

Crud example - add new toy

```
var express = require("express");
var fs = require("fs");
var app = express();
app.use(express.json()) ←
var port = 3000;
app.use('/assets', express.static( dirname + '/public'));
let toys = JSON.parse(fs.readFileSync("./data/data.json", 'utf-8'))
app.post('/api/v1/toys', (req, res)=>{
    const id = toys[toys.length-1].id +1; ← Create id of next toy
    const newToy = Object.assign({id:id}, req.body) ← Create new object and push it to array
    toys.push(newToy)
    fs.writeFile("./data/data.json", JSON.stringify(toys), err=>{ ← Write the object to the file
        res.status(201).json(
            {
                status:"success",
                data:toys
            }
        );
    });
    app.listen(port);
})
```

Add the body to the request. req .body now will contain the post data

Create id of next toy

Create new object and push it to array

Write the object to the file

localhost:3000/api/v1/toys

1. Same url
2. But now its post req

raw -> body

res

The screenshot shows the Postman interface with a POST request to `localhost:3000/api/v1/toys`. The request body is set to `raw` and contains the following JSON:

```
1 {  
2   "name": "test car",  
3   "price": 500,  
4   "description": "very very test speed car with controller"  
}
```

The response status is `201 Created`, and the response body is:

```
1 {  
2   "status": "success",  
3   "data": [  
4     {  
5       "id": 0,  
6       "name": "car",  
7       "price": 50,  
8       "description": "speed car with controller",  
9       "picture": "car.png"  
10      },  
11      {  
12        "id": 1,  
13        "name": "bear",  
14        "price": 70,  
15        "description": "bueatiful brown bear",  
16        "picture": "bear.png"  
17      },  
18      {  
19        "id": 2,  
20        "name": "test car",  
21        "price": 500,  
22        "description": "very very test speed car with controller",  
23        "picture": "car.png"  
24      }  
25    ]  
}
```

Crud example - get toy by id

```
app.get('/api/v1/toys/:id', (req, res) => {  
    const id = Number(req.params.id)  
    const toy = toys.find(el=>el.id === id);  
    if(toy === undefined) { ←  
        return res.status(404).json({  
            status: 'fail',  
            message: "id not exist"  
        })  
    }  
    res.status(200).json({  
        status: 'success', ←  
        data: toy  
    })  
})
```

If no toy

Response the correct toy

localhost:3000/api/v1/toys/2

localhost:3000/api/v1/toys/2

GET localhost:3000/api/v1/toys/2

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Query Params

	KEY	VALUE	DES
	Key	Value	Des

Body Cookies Headers (7) Test Results Status: 200 OK

Pretty Raw Preview Visualize JSON ↻

```
1 "status": "success",
2 "data": {
3     "id": 2,
4     "name": "test car",
5     "price": 500,
6     "description": "very very test speed car with controller",
7     "picture": "car.png"
8 }
9
10 }
```

localhost:3000/api/v1/toys/200 - fail

Id 200 not exist

404

Response

The screenshot shows a Postman interface with the following details:

- Request URL:** localhost:3000/api/v1/toys/200
- Method:** GET
- Params:** (highlighted in red)
- Headers:** (8)
- Body:** (green dot)
- Pre-request Script:** (grey dot)
- Tests:** (grey dot)
- Settings:** (grey dot)
- Cookies:** (grey dot)
- Query Params:** (highlighted in red)

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

- Test Results:** (highlighted in red)
- Status:** 404 Not Found
- Time:** 4 ms
- Size:** 284 B
- Save Response:** (blue button)
- Pretty:** (selected)
- Raw:**
- Preview:**
- Visualize:**
- JSON:** (dropdown menu)
- Code:** (dropdown menu)
- Copy:** (button)
- Search:** (button)

The response body is displayed as:

```
1  {
2   "status": "fail",
3   "message": "id not exist"
4 }
```

Crud example - patch(update) toy by id

```
app.patch('/api/v1/toys/:id', (req, res) => {
    const id = Number(req.params.id)
    const toy = toys.find(el=>el.id === id);
    if(toy === undefined){
        return res.status(404).json({
            status: 'fail',
            message: "id not exist"
        })
    }
    toys[id].name = req.body.name
    toy.description = req.body.description
    toy.picture = req.body.picture
    toy.price = Number(req.body.price)
    toys[id] = toy
    fs.writeFile("./data/data.json", JSON.stringify(toys), err=>{
        res.status(200).json(
        {
            status: "success",
            data:toys[id]
        })
    });
})
```

If no toy

Update the toy

Writing to file

Response the correct toy

localhost:3000/api/v1/toys/1

1. Same url
2. But now its patch req

The screenshot shows the Postman interface with a PATCH request to `localhost:3000/api/v1/toys/1`. The request body is set to `raw` and contains the following JSON:

```
1 {"name": "update test car",  
2 "price": 500000,  
3 "description": "woooow",  
4 "picture": "car100.png"}  
5  
6
```

The response status is 200 OK with a response body:

```
1 {"status": "success",  
2 "data": {  
3     "id": 1,  
4     "name": "update test car",  
5     "price": 500000,  
6     "description": "woooow",  
7     "picture": "car100.png"  
8 }  
9 }  
10
```

A green box labeled `raw -> body` has an arrow pointing to the `Body` tab in the Postman interface. A green box labeled `res` has an arrow pointing to the response body in the bottom panel.

data.json

```
[  
  {"id":0,"name":"car","price":50,"description":"speed car with  
  controller","picture":"car.png"},  
  {"id":1,"name":"update test  
  car","price":500000,"description":"woooow","picture":"car100.png"},  
  {"id":2,"name":"test car","price":500,"description":"very very test speed car with  
  controller","picture":"car.png"}]
```

Crud example - delete toy by id

```
app.delete('/api/v1/toys/:id', (req, res) => {
    const id = Number(req.params.id)
    const toy = toys.findIndex(obj => obj.id === id);
    if(toy === undefined) {
        return res.status(404).json({
            status: 'fail',
            message: "id not exist"
        })
    }
    var index = toys.findIndex(obj => obj.id === id);
    toys.splice(index, 1) ← Find index of toy
    fs.writeFile("./data/data.json", JSON.stringify(toys), err => {
        res.status(200).json(
        {
            status: "success delete",
            data: toy
        })
    });
})
```

← Delete toy from array

← Response deleted toy

localhost:3000/api/v1/toys/1

1. Same url
2. But now it's delete req

The screenshot shows the Postman application interface. At the top, there is an 'Overview' tab, a URL field containing 'localhost:3000/api/v1/toys/1', and a 'Send' button. Below the URL field, the method is set to 'DELETE'. The 'Params' tab is selected, showing a single parameter 'Key' with 'Value'. Under the 'Body' tab, the response is displayed in 'Pretty' format:

```
1
2   "status": "success delete",
3   "data": {
4     "id": 1,
5     "name": "update test car",
6     "price": 500000,
7     "description": "woooow",
8     "picture": "car100.png"
9   }
10 }
```

A large arrow points from the text 'Deleted toy' to the 'status' key in the JSON response.

Deleted toy

do it yourself 2 - data/player.json

```
[  
  {  
    "id":0,  
    "firstname": "Andre",  
    "lastname": "Iguodala",  
    "age": 37,  
    "Team": "Warriors"  
  },  
  {  
    "id":1,  
    "firstname": "Carmelo",  
    "lastname": "Anthony",  
    "age": 37,  
    "Team": "Lakers"  
  }]  
]
```

Do it yourself 2 - Build A Server Using Express

Create an express server that responds for the following requests:

Request	Response	Method	Comment
/api/v1/players	All players	get	Get all players from players.json
/api/v1/players	The created player	Post	Create new Player
/api/v1/players/:id	Player by id	Get	Get player by id
/api/v1/players/:id	Update player	Patch	Update player by id
/api/v1/players/:id	delete player	Delete	Delete player by id

Refactoring code part 1

Refactoring

Code refactoring is defined as the process of restructuring computer code without changing or adding to its external behavior and functionality. There are many ways to go about refactoring, but it most often comprises applying a series of standardized.

Refactoring code getAllToys

```
var express = require("express");
var fs = require("fs");
var app = express();
app.use(express.json())
var port = 3000;
let getAllToys=(req, res)=>{
    res.status(200).json(
        {
            status:"success",
            data:toys
        }
)
app.get('/api/v1/toys', getAllToys);
app.listen(port);
```

Refactoring code part createToy

```
let createToy=(req, res)=>{
    const id = toys[toys.length-1].id +1;
    console.log(req.body)
    const newToy = Object.assign({id:id},req.body)
    toys.push(newToy)

fs.writeFile("./data/data.json",JSON.stringify(toys),err=>{
    res.status(201).json(
        {
            status:"success",
            data:toys
        }
    );
}
app.post('/api/v1/toys',createToy);
```

Refactoring code part getToyById

```
let getToyById=(req, res)=>{
    const id = Number(req.params.id)
    const toy = toys.find(el=>el.id ===id);
    if(toy==undefined) {
        return res.status(404).json({
            status:'fail',
            message:"id not exist"
        })
    }
    res.status(200).json({
        status:'success',
        data:toy
    })
}
app.get('/api/v1/toys/:id',getToyById);
```

Refactoring code part updateToyById

```
let updateToyById=(req, res)=>{
    const id = Number(req.params.id)
    const toy = toys.find(el=>el.id ===id);
    if(toy==undefined){
        return res.status(404).json({
            status:'fail',
            message:"id not exist"
        })
    }
    toys[id].name = req.body.name
    toy.description = req.body.description
    toy.picture = req.body.picture
    toy.price = Number(req.body.price)
    toys[id] = toy

    fs.writeFile("./data/data.json",JSON.stringify(toys),err=>{
        res.status(200).json(
        {
            status:"success",
            data:toys[id]
        })
    });
}
app.patch('/api/v1/toys/:id',updateToyById);
```

Refactoring code part deleteToyById

```
let deleteToyById=(req, res)=>{
    const id = Number(req.params.id)
    const toy = toys.findIndex(el=>el.id ===id);
    if(toy==undefined) {
        return res.status(404).json({
            status: 'fail',
            message:"id not exist"
        })
    }
    var index = toys.findIndex(obj => obj.id==id);
    toys.splice(index, 1)
    fs.writeFile("./data/data.json",JSON.stringify(toys),err=>{
        res.status(200).json(
        {
            status:"success delete",
            data:toy
        })
    });
}

app.delete('/api/v1/toys/:id',deleteToyById);
```

Final server after refactoring part 1

```
1 var express = require("express");
2 var fs = require("fs");
3 var app = express();
4 app.use(express.json())
5 var port = 3000;
6 > let getAllToys=(req, res)=>{ ...
12   }
13 > let createToy=(req, res)=>{ ...
25   }
26 > let getToyById=(req, res)=>{ ...
39   }
40
41 > let updateToyById=(req, res)=>{ ...
63   }
64 > let deleteToyById=(req, res)=>{ ...
82   }
83 app.get('/api/v1/toys', getAllToys);
84 app.post('/api/v1/toys',createToy);
85 app.get('/api/v1/toys/:id',getToyById);
86 app.patch('/api/v1/toys/:id',updateToyById);
87 app.delete('/api/v1/toys/:id',deleteToyById);
88 app.listen(port);
```

Do it yourself 3- refactor exercise 2

Create an express server that responds to the following requests:

Request	Response	Method	Comment
/api/v1/players	All players	get	Get all players from players.json
/api/v1/players	The created player	Post	Create new Player
/api/v1/players/:id	Player by id	Get	Get player by id
/api/v1/players/:id	Update player	Patch	Update player by id
/api/v1/players/:id	delete player	Delete	Delete player by id

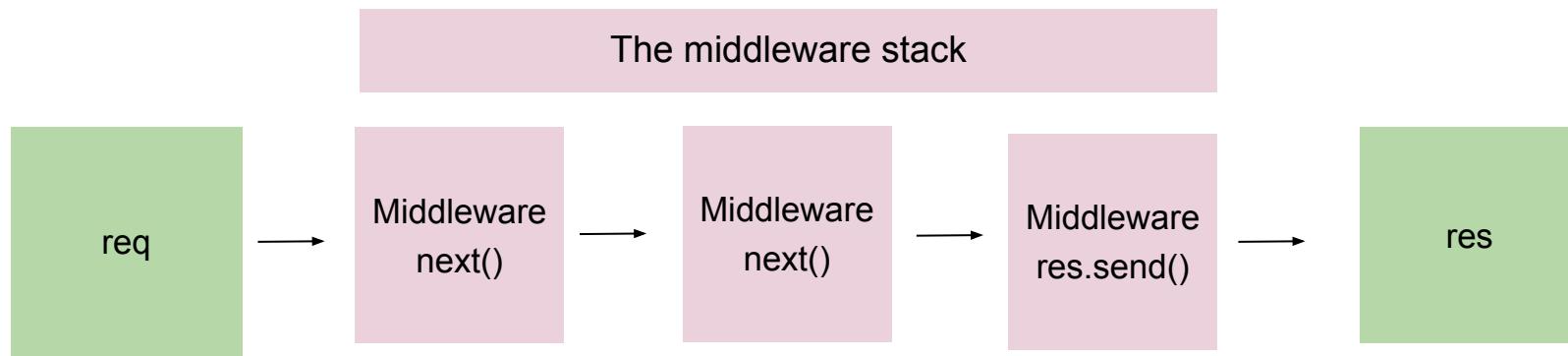


braintop
think.make.play



Chapter 10 - middleware

Request life cycle



Using middleware functions

Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named `next`.

Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.

Middleware functions can perform the following tasks:

- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware function in the stack.

If the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.

Application level middleware

Bind application-level middleware to an instance of the app object by using the app.use() and app.METHOD() functions, where METHOD is the HTTP method of the request that the middleware function handles (such as GET, PUT, or POST) in lowercase.

This example shows a middleware function with no mount path. The function is executed every time the app receives a request.

```
var express = require('express')
var app = express()

app.use(function (req, res, next) {
  console.log('Time:', Date.now())
  next()
})
app.listen(3000)
```

server1.js



Application level middleware

This example shows a middleware function mounted on the /user/:id path. The function is executed for any type of HTTP request on the /user/:id path.

```
var express = require('express')
var app = express()
app.use(function (req, res, next) {
  console.log('Time:', Date.now())
  next()
})
app.use('/user/:id', function (req, res, next) {
  console.log('Request Type:', req.method)
  next()
})
app.listen(3000)
```

The server stack there is no response

Application level middleware

This example shows a middleware function mounted on the /user/:id path. The function is executed for any type of HTTP request on the /user/:id path.

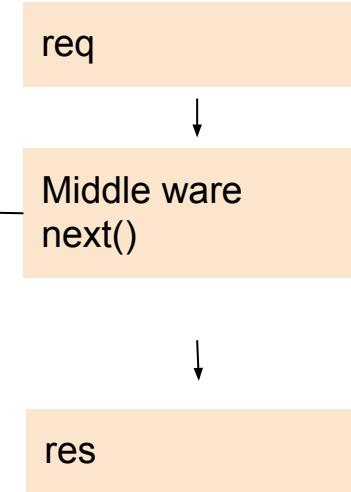
```
var express = require('express')
var app = express()
app.use(function (req, res, next) {
  console.log('Time:', Date.now())
  next()
})
app.use('/user/:id', function (req, res, next) {
  console.log('Request Type:', req.method)
  next()
})
app.listen(3000)
```

The server stack there is no response

Application level middleware

server2.js

```
var express = require('express')
var app = express()
app.use('/user/:id', function (req, res, next) {
  console.log('Request Type:', req.method)
  next() ←
})
app.get('/user/:id', function (req, res, next) {
  res.send('USER')
})
app.listen(3000)
```



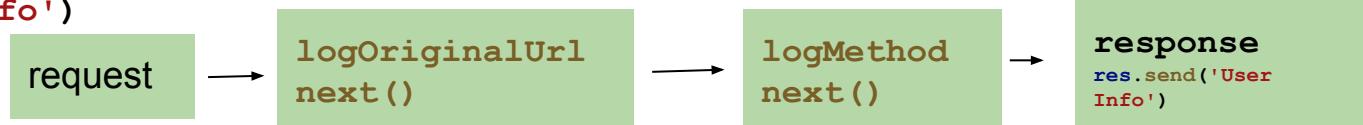
Array of middleware

server3.js

Middleware can also be declared in an array for reusability.

This example shows an array with a middleware sub-stack that handles GET requests to the /user/:id path

```
var express = require('express')
var app = express()
function logOriginalUrl (req, res, next) {
  console.log('Request URL:', req.originalUrl)
  next()
}
function logMethod (req, res, next) {
  console.log('Request Type:', req.method)
  next()
}
var logStuff = [logOriginalUrl, logMethod]
app.get('/user/:id', logStuff, function (req, res, next) {
  res.send('User Info')
})
app.listen(3000)
```



Third party middleware

Use third-party middleware to add functionality to Express apps.

Install the Node.js module for the required functionality, then load it in your app at the application level or at the router level.

The following example illustrates installing and loading the cookie-parsing middleware function cookie-parser.

```
Npm install cookie-parser
```

express.Router()

```
1 var express = require("express");
2 var fs = require("fs");
3 var app = express();
4 var toyRouter = express.Router(); ←
5 app.use('/api/v1/toys', toyRouter); ←
6 app.use(express.json())
7 var port = 3000;
8 app.use('/assets', express.static(__dirname + '/public'));
9 let toys = JSON.parse(fs.readFileSync("./data/data.json", 'utf-8'))
10 > let getAllToys=(req, res)=>{ ...
11   }
12 > let createToy=(req, res)=>{ ...
13   }
14 > let getToyById=(req, res)=>{ ...
15   }
16 > let updateToyById=(req, res)=>{ ...
17   }
18 > let deleteToyById=(req, res)=>{ ...
19   }
20   }
21   toyRouter.get('/', getAllToys);
22   toyRouter.post('/toys',createToy);
23   toyRouter.get('/:id',getToyById);
24   toyRouter.patch('/:id',updateToyById);
25   toyRouter.delete('/:id',deleteToyById);
26   }
27   app.listen(port);
```

part5/server.js

Declare toyRouter

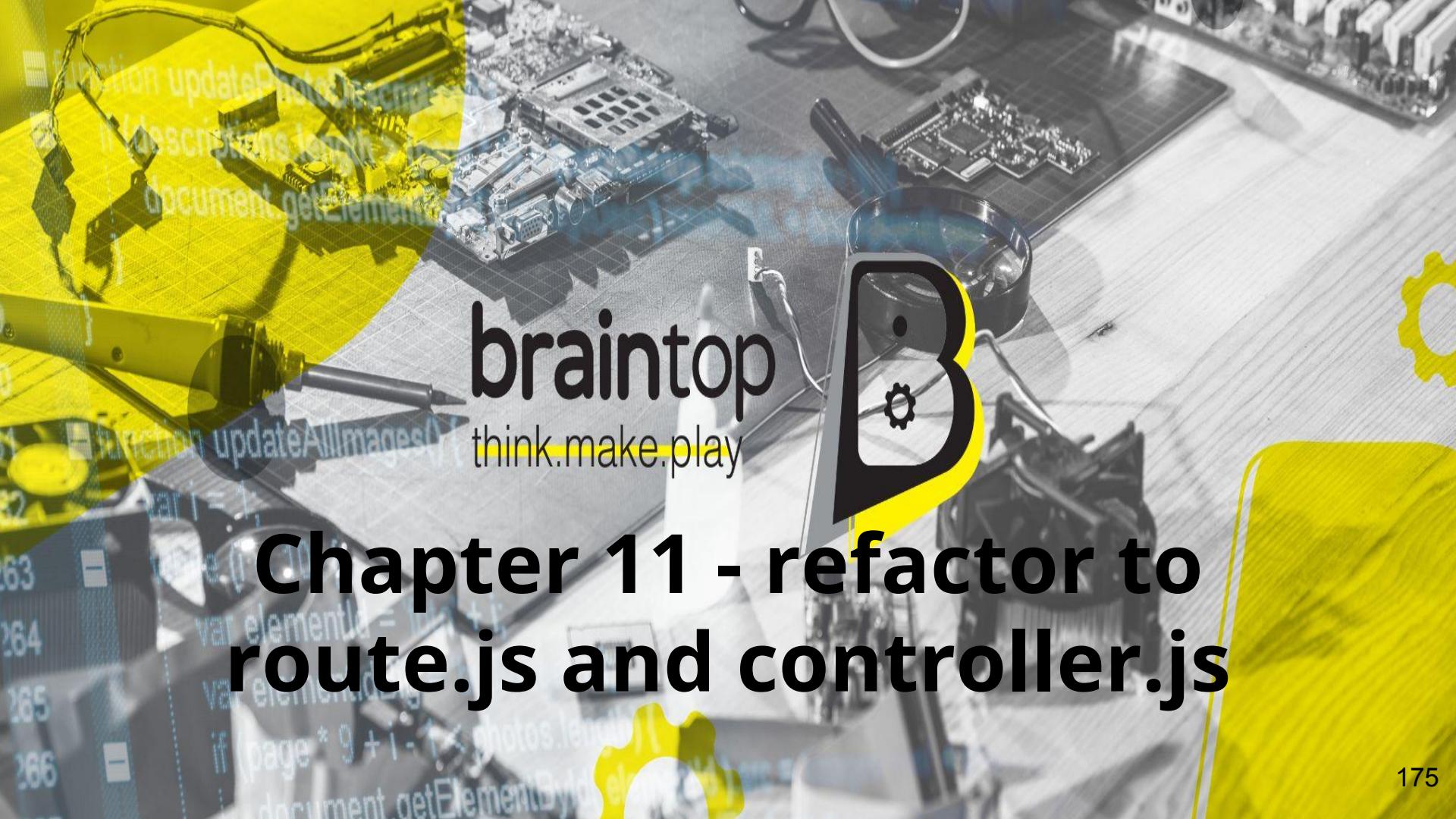
use toyRouter when user ask
for /api/v1/toys

You can delete /api/v1/toys
Because we mentioned it on
toyRouter

Do it yourself 1- refactor exercise 3 from chapter 11 and add a router according to last slide

Create an express server that responds for the following requests:

Request	Response	Method	Comment
/api/v1/players	All players	get	Get all players from players.json
/api/v1/players	The created player	Post	Create new Player
/api/v1/players/:id	Player by id	Get	Get player by id
/api/v1/players/:id	Update player	Patch	Update player by id
/api/v1/players/:id	delete player	Delete	Delete player by id



braintop
think.make.play

Chapter 11 - refactor to route.js and controller.js

Refactoring

Code refactoring is defined as the process of restructuring computer code without changing or adding to its external behavior and functionality. There are many ways to go about refactoring, but it most often comprises applying a series of standardized.

Create folders

1. Create folder api

a. Create file toyController.js

b. Create file toyRoute.js

```
✓ part6-middle
  ✓ api
    JS toyController.js
    JS toyRoute.js
  > assets/toys
  > data
  > node_modules
  {} package-lock.json
  {} package.json
  JS server.js
```

toyRouter.js

part6-middle/api/toyController.js

```
const express = require('express')

var toyRouter = express.Router();

var toyController = require('./toyController')

toyRouter.get('/', toyController.getAllToys);

toyRouter.post('/toys',toyController.createToy);

toyRouter.get('/:id',toyController.getToyById);

toyRouter.patch('/:id',toyController.updateToyById);

toyRouter.delete('/:id',toyController.deleteToyById);

module.exports = toyRouter;
```

toyRouter.js

chap11-refactor > part6-middle > api > `JS` toyController.js > ...

```
1 const fs = require('fs')
2 let toys = JSON.parse(fs.readFileSync("./data/data.json", 'utf-8'))
3 > exports.getAllToys=(req, res)=>{ ...
9   }
10 > exports.createToy=(req, res)=>{ ...
22   }
23 > exports.getToyById=(req, res)=>{ ...
36   }
37 > exports.updateToyById=(req, res)=>{ ...
59   }
60 > exports.deleteToyById=(req, res)=>{ ...
78   }
79 
```

Exports all
functions



server.js

```
var express = require("express");
var app = express();
var toyRouter = require("./api/toyRoute") ← require toyRoute.js
app.use('/api/v1/toys', toyRouter); ← use toyRouter when user
app.use(express.json()) request /api/v1/toys
var port = 3000;
app.use('/assets', express.static(__dirname +
'/public'));
app.listen(port);
```

Use `express.json` for body content

Refactor exercise 3 chapter 3 - to router.js and controller.js

Request	Response	Method	Comment
/api/v1/players	All players	get	Get all players from players.json
/api/v1/players	The created player	Post	Create new Player
/api/v1/players/:id	Player by id	Get	Get player by id
/api/v1/players/:id	Update player	Patch	Update player by id
/api/v1/players/:id	delete player	Delete	Delete player by id



braintop
think.make.play



Chapter 12 - mongoDb

What is mongodb

MongoDB Atlas is a multi-cloud database service by the same people that build MongoDB. Atlas simplifies deploying and managing your databases while offering the versatility you need to build resilient and performant global applications on the cloud providers of your choice.

Database, collection, document

Database

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Collection

Collection is a group of MongoDB documents. A collection exists within a single database. Documents within a collection can have different fields. MongoDB stores documents in collections. Collections are analogous to tables in relational databases.

Document

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data. MongoDB stores data records as BSON documents. BSON is a binary representation of [JSON](#) documents, though it contains more data types than JSON. For the BSON spec, see bsonspec.org. See also [BSON Types](#).

MongoDB - Signing Up

1. Go to <https://docs.atlas.mongodb.com/tutorial/create-atlas-account/> and create account
2. Deploy free cluster <https://docs.atlas.mongodb.com/tutorial/deploy-free-tier-cluster/>
3. Add your ip address to access list <https://docs.atlas.mongodb.com/security/add-ip-address-to-list/>

Document Structure

Document structure

```
{  
    field1: value1,  
    field2: value2,  
    field3: value3,  
    ...  
    fieldN: valueN  
}
```

Document structure example

```
var mydoc = {  
    _id: ObjectId("5099803df3f4948bd2f98391") ,  
    name: { first: "Alan", last: "Turing" } ,  
    birth: new Date('Jun 23, 1912') ,  
    death: new Date('Jun 07, 1954') ,  
    contribs: [ "Turing machine", "Turing test", "Turingery" ] ,  
    views : NumberLong(1250000)  
}
```

Field Names are strings

The above fields have the following data types:

_id holds an ObjectId - `_id` is a 12 bytes hexadecimal number which assures the uniqueness of every document.

name holds an embedded document that contains the fields `first` and `last`.

birth and `death` hold values of the `Date` type.

contribs holds an array of strings.

views holds a value of the `NumberLong` type.

Document-array fields

To specify or access an element of an array by the zero-based index position, concatenate the array name with the dot (.) and zero-based index position, and enclose in quotes: "`<array>.<index>`"

For example, given the following field in a document:

```
{  
  ...  
  contribs: [ "Turing machine", "Turing test", "Turingery" ],  
  ...  
}
```

Embedded documents

To specify or access a field of an embedded document with dot notation, concatenate the embedded document name with the dot (.) and the field name, and enclose in quotes: "`<embedded document>.<field>`"

For example, given the following field in a document:

```
{  
  ...  
  name: { first: "Alan", last: "Turing" },  
  contact: { phone: { type: "cell", number: "111-222-3333" } },  
  ...  
}
```

- To specify the field named `last` in the `name` field, use the dot notation "`name.last`".
- To specify the number in the `phone` document in the `contact` field, use the dot notation "`contact.phone.number`".

Mongodb crud introduction

CRUD operations *create, read, update, and delete*

Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.

Databases and Collections

MongoDB stores data records as documents (specifically BSON documents) which are gathered together in collections. A database stores one or more collections of documents.

Create database

Create a Database

If a database does not exist, MongoDB creates the database when you first store data for that database. As such, you can switch to a non-existent database and perform the following operation in `mongosh`

You can browse to <https://docs.mongodb.com/manual/tutorial/getting-started/>

For training online : you can : switch database, insert, find all and filter data

Click inside the shell to connect. Once connected, you can run the examples in the shell above.

[1. Switch Database](#)

[2. Insert](#)

[3. Find All](#)

[4. Filter Data](#)

[5. Project Fields](#)

[6. Aggregate](#)

Create database on mongodb
More mongodb - [link](#)

After signup created

List of users and pass that have access to database

List of ip's that have access to database

The screenshot shows the MongoDB Atlas interface for a project named 'Project 0'. The left sidebar has sections for Deployment, Databases (selected), Data Lake, Data Services, Triggers, Data API (with a 'PREVIEW' button), and Security (Database Access, Network Access, Advanced). The main area is titled 'Database Deployments' and shows 'Cluster0'. It includes metrics like R: 0, W: 0, Connections: 0, In: 0.0 B/s, Out: 0.0 B/s, and Data Size: 512.0 MB. A graph shows traffic over time. Below the metrics is a table with columns: VERSION, REGION, CLUSTER TIER, TYPE, BACKUPS, LINKED REALM APP, and ATLAS SEARCH. The table data is: VERSION 4.4.10, REGION AWS / N. Virginia (us-east-1), CLUSTER TIER M0 Sandbox (General), TYPE Replica Set - 3 nodes, BACKUPS Inactive, LINKED REALM APP None Linked, and ATLAS SEARCH Create Index. There are also 'FREE' and 'SHARED' buttons, an 'Enhance Your Experience' section, and an 'Upgrade' button.

VERSION	REGION	CLUSTER TIER	TYPE	BACKUPS	LINKED REALM APP	ATLAS SEARCH
4.4.10	AWS / N. Virginia (us-east-1)	M0 Sandbox (General)	Replica Set - 3 nodes	Inactive	None Linked	Create Index

After signup create

Click on connect

The screenshot shows the MongoDB Atlas interface for a project named 'PROJECT 0'. On the left, a sidebar lists 'DEPLOYMENT' sections: 'Databases' (selected), 'Data Lake', 'DATA SERVICES' (Triggers, Data API PREVIEW), and 'SECURITY' (Database Access, Network Access, Advanced). A green callout box with the text 'Click on connect' has an arrow pointing to the 'Connect' button for 'Cluster0'.

The main area is titled 'Database Deployments' and shows 'Cluster0'. It includes a search bar, a 'Create' button, and tabs for 'Cluster0' (selected), 'Connect', 'View Monitoring', 'Browse Collections', and '...'. Below this, there's a summary section with metrics: R 0, W 0 (Last 16 minutes), Connections 0 (Last 16 minutes), In 0.0 B/s, Out 0.0 B/s (Last 11 minutes), Data Size 512.0 MB, and a 'FREE' badge.

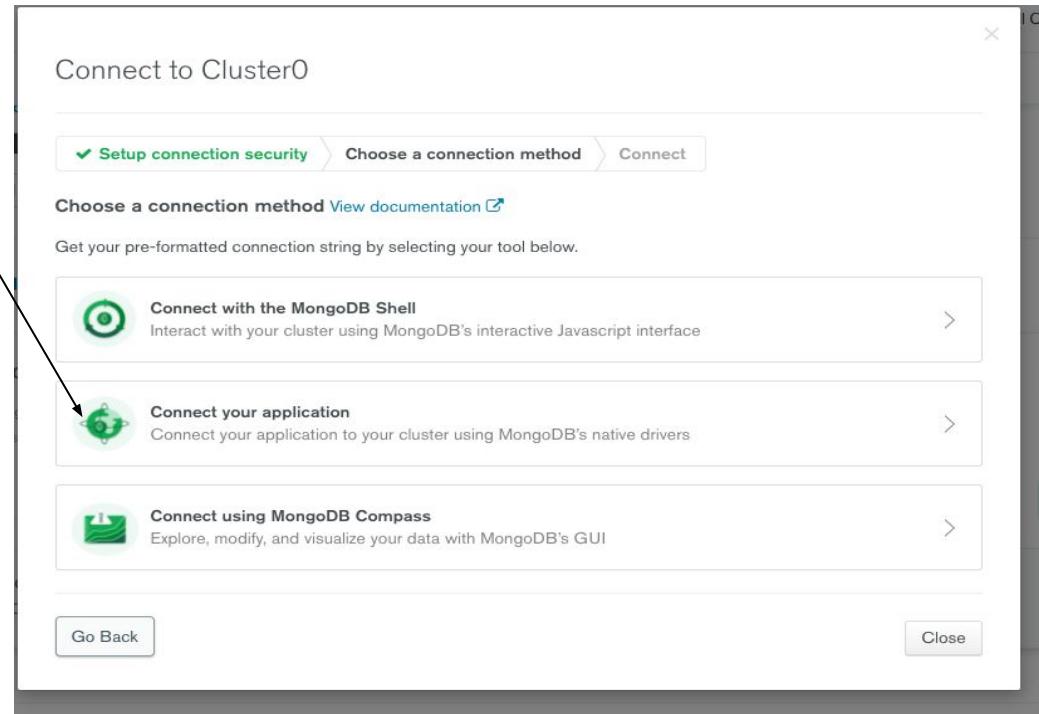
On the right, there's an 'Enhance Your Experience' section with a 'Upgrade' button, and a table with columns: VERSION, REGION, CLUSTER TIER, TYPE, BACKUPS, LINKED REALM APP, and ATLAS SEARCH. The table data is:

VERSION	REGION	CLUSTER TIER	TYPE	BACKUPS	LINKED REALM APP	ATLAS SEARCH
4.4.10	AWS / N. Virginia (us-east-1)	M0 Sandbox (General)	Replica Set - 3 nodes	Inactive	None Linked	Create Index

A small circular icon with a person icon is located at the bottom right.

After signup create

We will connect with
nodejs



After signup create

We will connect with
nodejs

Connect to Cluster0

✓ Setup connection security ✓ Choose a connection method Connect

1 Select your driver and version

DRIVER VERSION

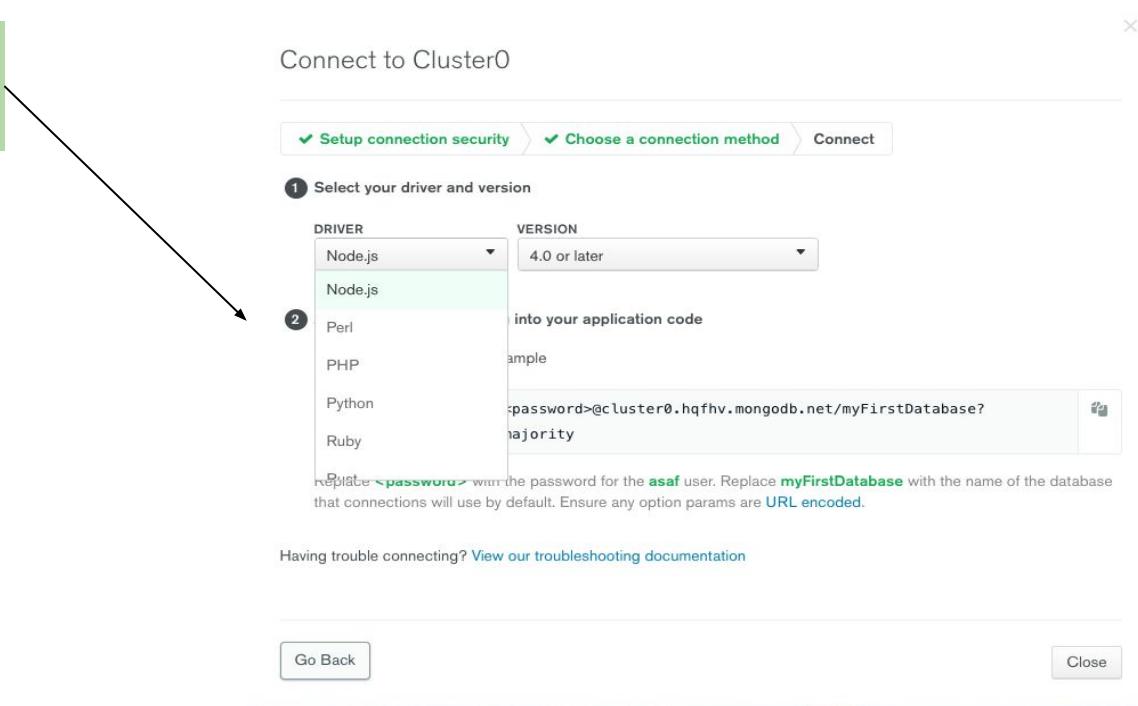
Node.js	4.0 or later
Node.js	into your application code
Perl	example
PHP	:password>@cluster0.hqfhv.mongodb.net/myFirstDatabase?
Python	majority
Ruby	

2

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

Having trouble connecting? [View our troubleshooting documentation](#)

Go Back Close



After signup create

Copy the code

The screenshot shows the MongoDB Atlas connection setup process. It includes steps for 'Setup connection security', 'Choose a connection method', and 'Connect'. Step 1: 'Select your driver and version' (DRIVER: Node.js, VERSION: 4.0 or later). Step 2: 'Add your connection string into your application code' (checkbox checked for 'Include full driver code example'). The code example is:

```
const { MongoClient } = require('mongodb');
const uri = "mongodb+srv://asaf:<password>@cluster0.hqfhv.mongodb.net/myFirstDatabase?retryWrites=true&w=majority";
const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true });
client.connect(err => {
  const collection = client.db("test").collection("devices");
  // perform actions on the collection object
  client.close();
});
```

Below the code, instructions say: 'Replace <password> with the password for the **asaf** user. Replace **myFirstDatabase** with the name of the database that connections will use by default. Ensure any option params are **URL encoded**.'

Having trouble connecting? [View our troubleshooting documentation](#)

Create a1.js and paste the code

Browse data

The screenshot shows the MongoDB Atlas interface for a project named 'Project 0'. On the left, there's a sidebar with sections for DEPLOYMENT (highlighted), DATA SERVICES (Triggers, Data API PREVIEW), and SECURITY (Database Access, Network Access, Advanced). The main area is titled 'Database Deployments' and shows a single cluster named 'Cluster0'. The cluster card displays metrics like R 0, W 0, Connections 1.0, and Data Size 0.0 B. It also features a 'Browse Collections' button, which is the target of a black arrow from the 'Browse data' callout. To the right of the cluster card, there are buttons for 'FREE' and 'SHARED', and a green 'Upgrade' button. Below the cluster card, a table provides detailed information: VERSION (4.4.10), REGION (AWS / N. Virginia (us-east-1)), CLUSTER TIER (M0 Sandbox (General)), TYPE (Replica Set - 3 nodes), BACKUPS (Inactive), LINKED REALM APP (None Linked), and ATLAS SEARCH (Create Index). A small circular icon with a person icon and a speech bubble is located in the bottom right corner.

VERSION	REGION	CLUSTER TIER	TYPE	BACKUPS	LINKED REALM APP	ATLAS SEARCH
4.4.10	AWS / N. Virginia (us-east-1)	M0 Sandbox (General)	Replica Set - 3 nodes	Inactive	None Linked	Create Index

Create a1.js and paste the code

Project 0 Atlas Realm Charts

DEPLOYMENT

Databases Data Lake

DATA SERVICES

Triggers Data API PREVIEW

SECURITY

Database Access Network Access Advanced

BRAIN'S ORG - 2021-12-19 > PROJECT 0 > DATABASES

Cluster0

VERSION 4.4.10 REGION AWS N. Virginia (us-east-1)

Overview Real Time Metrics **Collections** Search Profiler Performance Advisor Online Archive Command

DATABASES: 0 COLLECTIONS: 0

[VISUALIZE YOUR DATA](#) [REFRESH](#)

 Explore Your Data

- **Find:** run queries and interact with documents
- **Indexes:** build and manage indexes
- **Aggregation:** test aggregation pipelines
- **Search:** build search indexes

[Load a Sample Dataset](#) [Add My Own Data](#)

Learn more in Docs and Tutorials ↗

Feedback icon

Create a1.js and paste the code

Create new folder named mongo and Run **npm init** in the terminal, and then **npm install mongodb**. And add the this code.

```
const { MongoClient } = require('mongodb');
const uri =
  "mongodb+srv://asaf:<password>@cluster0.hqfhv.mongodb.net/myFirstDatabase?retryWrites=true&
  w=majority";
const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true });
client.connect(err => {
  const collection = client.db("test").collection("devices");
  // perform actions on the collection object
  client.close();
});
```

Connection string

Add your password to the connection
string

Db name

Collection name

Create a1.js and paste the code

```
const { MongoClient } = require('mongodb');
const uri =
"mongodb+srv://asaf:asaf@cluster0.hqfhv.mongodb.net/myFirstDatabase?retryWrites=true&w=majority" ;
const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true });
client.connect(async (err) => {
  if(err) console.log(err)
  const db = client.db("test"); ← Create db called test
  // Use the collection "people"
  const col = db.collection("people"); ← Create collection named people
  // Construct a document
  let personDocument = { ← Create document : personDocument
    "name": { "first": "Alan", "last": "Turing" },
    "birth": new Date(1912, 5, 23), // June 23, 1912
    "death": new Date(1954, 5, 7), // June 7, 1954
    "contribs": [ "Turing machine", "Turing test", "Turingery" ],
    "views": 1250000
  }
  const p = await col.insertOne(personDocument); ← Insert to people collection
  client.close();
});
```

Create **db** called test

Create **collection** named people

Create **document** : personDocument

Insert to people **collection**

Create a1.js and paste the code

The screenshot shows the MongoDB Atlas interface for a project named "Project 0". The "Atlas" tab is selected in the top navigation bar. The main view is for "Cluster0", which is running version 4.4.10 in the AWS N. Virginia (us-east-1) region. The "Collections" tab is active, showing a message: "DATABASES: 0 COLLECTIONS: 0". Below this, there's a section titled "Explore Your Data" with a list of features: "Find", "Indexes", "Aggregation", and "Search". At the bottom, there are buttons for "Load a Sample Dataset" and "Add My Own Data". A red arrow points from the text "Create a1.js and paste the code" in the slide to the "Add My Own Data" button.

Project 0

Atlas REALM Charts

DEPLOYMENT

Databases

Data Lake

DATA SERVICES

Triggers

Data API PREVIEW

SECURITY

Database Access

Network Access

Advanced

BRAIN'S ORG - 2021-12-19 > PROJECT 0 > DATABASES

Cluster0

VERSION 4.4.10 REGION AWS N. Virginia (us-east-1)

Overview Real Time Metrics Collections Search Profiler Performance Advisor Online Archive Command

DATABASES: 0 COLLECTIONS: 0

Explore Your Data

- Find: run queries and interact with documents
- Indexes: build and manage indexes
- Aggregation: test aggregation pipelines
- Search: build search indexes

Load a Sample Dataset Add My Own Data

Learn more in Docs and Tutorials ↗

◀

↗

Create a1.js and paste the code

The screenshot shows the MongoDB Atlas interface. The top navigation bar includes Project 0, Atlas (selected), Realm, Charts, and various icons for monitoring and support. Below the navigation is a secondary menu with DEPLOYMENT, DATABASES (selected), DATA SERVICES, SECURITY, and a PREVIEW section. The DATABASES section shows 'Data Lake'. The DATA SERVICES section shows 'Triggers', 'Data API' (with a PREVIEW button), and 'SECURITY' (Database Access, Network Access, Advanced). The main content area displays 'test.people' collection details: COLLECTION SIZE: 180B, TOTAL DOCUMENTS: 1, INDEXES TOTAL SIZE: 4KB. It features tabs for Find, Indexes, Schema Anti-Patterns, Aggregation, and Search Indexes. A FILTER bar contains the query '{ field: 'value' }'. Below it, a table shows the single document with fields: _id, name, birth, death, contribs, and views. At the bottom, System Status is shown as 'All Good'.

DATABASES: 1 COLLECTIONS: 1

+ Create Database

NAMESPACES

Find Indexes Schema Anti-Patterns Aggregation Search Indexes

FILTER { field: 'value' }

OPTIONS Apply Reset

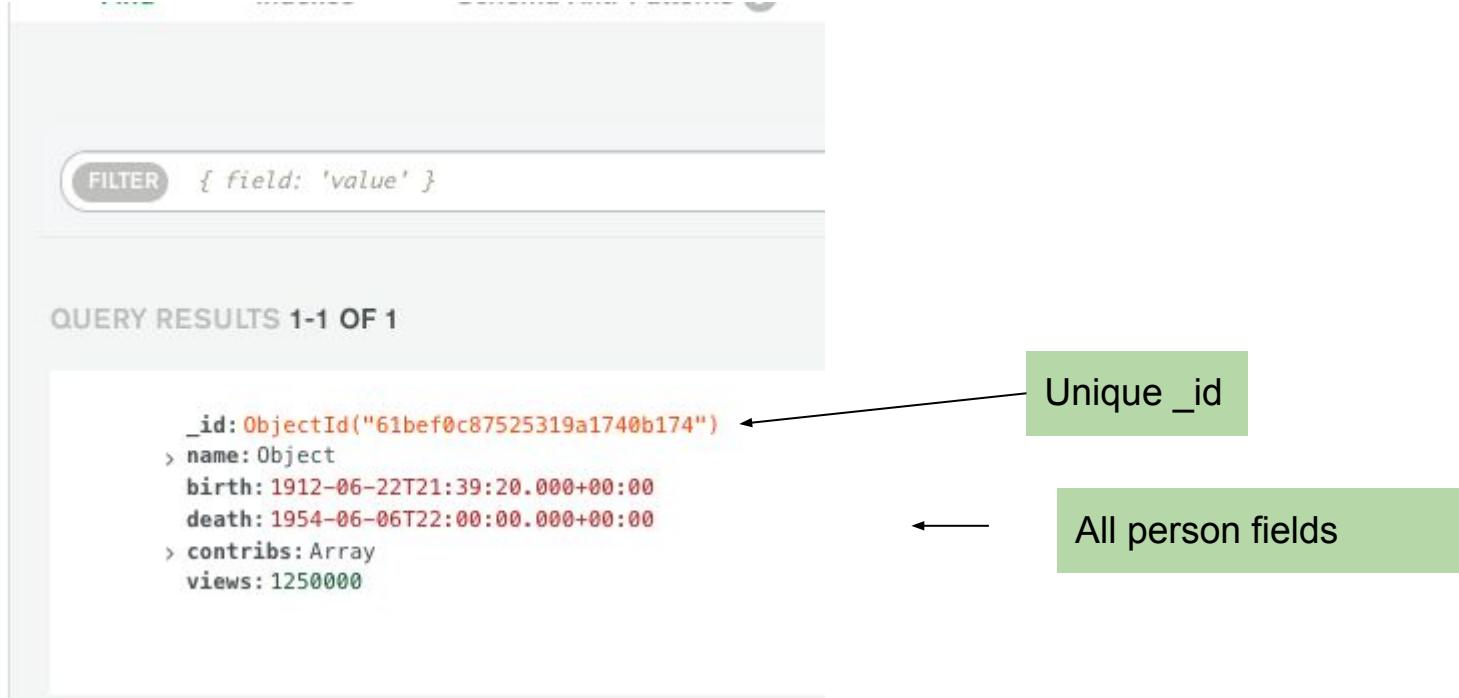
QUERY RESULTS 1-1 OF 1

_id: ObjectId("61bef0c87525319a1740b174")
> name: Object
birth: 1912-06-22T21:39:20.000+00:00
death: 1954-06-06T22:00:00.000+00:00
> contribs: Array
views: 1250000

System Status: All Good

©2021 MongoDB, Inc. Status Terms Privacy Atlas Blog Contact Sales

Create a1.js and paste the code



The screenshot shows the MongoDB Compass interface. At the top, there is a filter bar with the text `{ field: 'value' }`. Below it, the results section is titled "QUERY RESULTS 1-1 OF 1". A single document is listed:

```
_id: ObjectId("61bef0c87525319a1740b174")
> name: Object
  birth: 1912-06-22T21:39:20.000+00:00
  death: 1954-06-06T22:00:00.000+00:00
> contribs: Array
  views: 1250000
```

Two annotations are present on the right side of the document:

- A green box labeled "Unique _id" with an arrow pointing to the `_id` field.
- A green box labeled "All person fields" with an arrow pointing to the `name`, `birth`, `death`, and `contribs` fields.

Final code with try and catch

```
const { MongoClient } = require('mongodb');
const uri = "mongodb+srv://asaf:asaf@cluster0.hqfhv.mongodb.net/myFirstDatabase?retryWrites=true&w=majority";
const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true });
client.connect(async (err) => {
  if(err) console.log(err)
  try
  {
    const db = client.db("test");
    // Use the collection "people"
    const col = db.collection("people");
    // Construct a document
    let personDocument = {
      "name": { "first": "Alan", "last": "Turing" },
      "birth": new Date(1912, 5, 23), // June 23, 1912
      "death": new Date(1954, 5, 7), // June 7, 1954
      "contribs": [ "Turing machine", "Turing test", "Turingery" ],
      "views": 1250000
    }
    const p = await col.insertOne(personDocument);
  }
  catch (err) {
    console.log(err.stack);
  }
  finally {
    await client.close();
  }
});
```

Final code with try and catch

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Construct a document
  let personDocument = {
    "name": { "first": "Alan", "last": "Turing" },
    "birth": new Date(1912, 5, 23), // June 23, 1912
    "death": new Date(1954, 5, 7), // June 7, 1954
    "contribs": [ "Turing machine", "Turing test", "Turingery" ],
    "views": 1250000
  }
  const p = await col.insertOne(personDocument);
  // Find one document
  const myDoc = await col.findOne();
  // Print to the console
  console.log(myDoc);
}
```

-----stdout -----

```
{
  _id: new ObjectId("61bef30d9126e644fe102cfb"),
  name: { first: 'Alan', last: 'Turing' },
  birth: 1912-06-22T21:39:20.000Z,
  death: 1954-06-06T22:00:00.000Z,
  contribs: [ 'Turing machine', 'Turing test', 'Turingery' ],
  views: 1250000
}
```

Lets insert 2 persons

```
let personDocument1 = {  
    "name": { "first": "Mike", "last": "Lewis" },  
    "birth": new Date(1942, 5, 22),  
    "death": new Date(1980, 6, 8),  
    "contribs": ["Turingery" ],  
    "views": 130000  
}
```

```
let personDocument2 = {  
    "name": { "first": "Lisa", "last": "Mine" },  
    "birth": new Date(1971, 8, 22),  
    "death": new Date(1967, 9, 9),  
    "contribs": [ "Turing machine"],  
    "views": 145000  
}
```

After insert

The screenshot shows the MongoDB Compass interface with a database named 'test' and a collection named 'people'. The interface includes tabs for 'Find', 'Indexes', 'Schema Anti-Patterns', 'Aggregation', and 'Search Indexes'. A 'FILTER' button with the query '{ field: 'value' }' is present. The results section displays three documents:

```
_id: ObjectId("61bef30d9126e644fe102cfb")
~name: Object
  first: "Alan"
  last: "Turing"
  birth: 1912-06-23T21:39:20.000+00:00
  death: 1954-06-05T22:08:00.000+00:00
> contribs: Array
  views: 1250000

_id: ObjectId("61bf1133176f849e321250d0")
~name: Object
  first: "Mike"
  last: "Lewis"
  birth: 1942-06-21T21:00:00.000+00:00
  death: 1980-07-07T22:00:00.000+00:00
> contribs: Array
  views: 130000

_id: ObjectId("61bf1145651d237d271b528c")
~name: Object
  first: "Lisa"
  last: "Mine"
  birth: 1971-09-21T22:00:00.000+00:00
  death: 1987-10-08T22:00:00.000+00:00
> contribs: Array
  views: 145000
```

Insert

insertOne

insert into collection one document

```
const p = await col.insertOne(personDocument);
```

insertOne vs insertMany

The `insert()` method is deprecated in major driver so you should use the the `.insertOne()` method whenever you want to insert a single document into your collection and the `.insertMany` when you want to insert multiple documents into your collection.

More about insertOne and insertMany

insertOne - [link](#)

insertMany - [link](#)

Do it yourself 1

The sequence available in the next slides (till the end of the chapter) will only work if we perform all the steps described in the previous slides - connection to DB, try, Catch etc

1. Open an account in mongodb
2. After registering, make sure you have a user in the account's user list who has access to the account
3. Make sure your ip is in the ip list if not then add it
4. Create a js file named b1.js
5. Click connect and then connect your application and select nodejs version 4 Copy the code and paste it in b1.js Run the file by node b1

Do it yourself 2

Create a product object with the following attributes in the b1.js file:

```
let product = {  
    "title": "ball" ,  
    "description": "Big blue ball" ,  
    "tags": [ "circle", "toy", "kids" ] ,  
    "age": 12 ,  
    "price": 20  
}
```

Add the correct commands to put this product into mongodb using the insert command
Run the b1 file and make sure a product logs in

Do it yourself 3

Create 3 additional objects that describe different products in the b1.js file

Add the correct commands to insert into mongodb using the insert command

Run the b1 file and make sure a product logs in

find()

```
find({})  
Return all documents  
try  
{  
  const db = client.db("test");  
  // Use the collection "people"  
  const col = db.collection("people");  
  // Find one document  
  const myDoc = await col.find().toArray();  
  // Print to the console  
  console.log(myDoc);  
}
```

```
[  
  {  
    _id: new ObjectId("61bef30d9126e644fe102cfb"),  
    name: { first: 'Alan', last: 'Turing' },  
    birth: 1912-06-22T21:39:20.000Z,  
    death: 1954-06-06T22:00:00.000Z,  
    contribs: [ 'Turing machine', 'Turing test', 'Turingery' ],  
    views: 1250000  
  },  
  {  
    _id: new ObjectId("61bf1133176f849e321250d0"),  
    name: { first: 'Mike', last: 'Lewis' },  
    birth: 1942-06-21T21:00:00.000Z,  
    death: 1980-07-07T22:00:00.000Z,  
    contribs: [ 'Turingery' ],  
    views: 130000  
  },  
  {  
    _id: new ObjectId("61bf1145651d237d271b528c"),  
    name: { first: 'Lisa', last: 'Mine' },  
    birth: 1971-09-21T22:00:00.000Z,  
    death: 1967-10-08T22:00:00.000Z,  
    contribs: [ 'Turing machine' ],  
    views: 145000  
  }  
]
```

Do it yourself 4

Open a file named b4.js and write a suitable code that will print to the console all the records in mongodb

Help with the command:

```
find().toArray();
```

Find and filter ({})

```
find({views:130000})  
Return the document that the views are 130000 (Mike Lewis)  
try  
{  
  const db = client.db("test");  
  // Use the collection "people"  
  const col = db.collection("people");  
  // Find one document  
  const myDoc = await col.find({ "views":130000 }).toArray();  
  // Print to the console  
  console.log(myDoc);  
}
```

Only people with 130,000 views

```
{  
  _id: new ObjectId("61bf1133176f849e321250d0"),  
  name: { first: 'Mike', last: 'Lewis' },  
  birth: 1942-06-21T21:00:00.000Z,  
  death: 1980-07-07T22:00:00.000Z,  
  contribs: [ 'Turingery' ],  
  views: 130000  
}
```

Do it yourself 5

Open a file named b5.js and write a suitable code that will print to the console all the records that are in mongodb that cost 20.

Help with the command:

```
find().toArray();
```

Popular operators - link to more operators

<code>\$eq</code>	Matches values that are equal to a specified value.
<code>\$gt</code>	Matches values that are greater than a specified value.
<code>\$gte</code>	Matches values that are greater than or equal to a specified value.
<code>\$in</code>	Matches any of the values specified in an array.
<code>\$lt</code>	Matches values that are less than a specified value.
<code>\$lte</code>	Matches values that are less than or equal to a specified value.
<code>\$ne</code>	Matches all values that are not equal to a specified value.
<code>\$nin</code>	Matches none of the values specified in an array

Find and filter ({} - \$gt

The following operation uses the `$gt` operator returns all the documents from the `people` collection where `views` is greater than 135000

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await col.find({ "views": { $gt: 135000 } }).toArray();
  // Print to the console
  console.log(myDoc);
}
```

↓

```
[
  {
    _id: new ObjectId("61bef30d9126e644fe102cfb"),
    name: { first: 'Alan', last: 'Turing' },
    birth: 1912-06-22T21:39:20.000Z,
    death: 1954-06-06T22:00:00.000Z,
    contribs: [ 'Turing machine', 'Turing test', 'Turingery' ],
    views: 1250000
  },
  {
    _id: new ObjectId("61bf1145651d237d271b528c"),
    name: { first: 'Lisa', last: 'Mine' },
    birth: 1971-09-21T22:00:00.000Z,
    death: 1967-10-08T22:00:00.000Z,
    contribs: [ 'Turing machine' ],
    views: 145000
  }
]
```

Do it yourself 6

Open a file named b6.js and write a suitable code that will print to the console all the records that are in mongodb that cost more than 20.

Help with the command:

```
find().toArray();
```

Find and filter ({} - \$lt

The following operation uses the `$lt` operator returns all the documents from the `people` collection where `views` is less than 135000

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await col.find({ "views": {$lt: 135000} }).toArray();
  // Print to the console
  console.log(myDoc);
}
```

```
[{"_id": new ObjectId("61bf1133176f849e321250d0"),
  "name": { first: "Mike", last: "Lewis" },
  "birth": "1942-06-21T21:00:00.000Z",
  "death": "1980-07-07T22:00:00.000Z",
  "contribs": [ "Turingery" ],
  "views": 130000}]
```

Do it yourself 7

Open a file named b7.js and write a suitable code that will print to the console all the records that are in mongodb that cost less than 20.

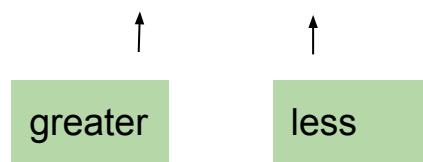
Help with the command:

```
find().toArray();
```

Find and filter ({} - between

Combine comparison operators to specify ranges for a field. The following operation returns from the people collection documents where views is **between** 100,000 and 130,000

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await col.find({ "views": { $gt: 100000, $lt: 150000 } }).toArray();
  // Print to the console
  console.log(myDoc);
}
```



The diagram consists of two green rectangular boxes. The left box contains the word "greater" and has an upward-pointing arrow above it, which points to the "\$gt" operator in the code. The right box contains the word "less" and has an upward-pointing arrow above it, which points to the "\$lt" operator in the code.

Do it yourself 8

Open a file named b8.js and write a suitable code that will print to the console all the records that are in mongodb that cost between 20 and 40.

Help with the command:

```
find().toArray();
```

Query multiple condition

The following operation returns all the documents from the people_collection where views field is greater than 100000 and birth field is greater 01/01/1920:

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await col.find({
    "views": {$gt:100000},
    "birth": {$gt: new Date('1920-01-01') } }) .toArray();
  // Print to the console
  console.log(myDoc);
}
```

Multiple condition

Do it yourself 9

Open a file called b9.js and write a suitable code that will print to the console all the records that are in mongodb that are priced larger than 20 and are also suitable for ages 12
Help with the command:

```
find().toArray();
```

Query embedded Documents

The following operation returns documents in the people where the embedded document name is exactly { first: "Lisa", last: "Mine" }, including the order:

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await col.find({name: { first: "Lisa", last: "Mine" }}).toArray();
  // Print to the console
  console.log(myDoc);
}
```



Embedded query

Query fields of embedded Documents

The following operation returns documents in the collection where the embedded document name contains a field first with the value "Lisa" and a field last with the value "Mine". The query uses dot **notation** to access fields in an embedded document:

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await col.find(
  {
    "name.first": "Lisa",
    "name.last": "Mine"
  }).toArray();
  // Print to the console
  console.log(myDoc);
}
```

Query Arrays

The following operation returns documents in the people collection where the array field `contribs` contains the element "Turing test":

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await col.find(
    {"contribs": "Turing test" }).toArray();
  // Print to the console
  console.log(myDoc);
}
```

```
[{"_id": new ObjectId("61bef30d9126e644fe102cfb"), "name": { first: "Alan", last: "Turing" }, "birth": 1912-06-22T21:39:20.000Z, "death": 1954-06-06T22:00:00.000Z, "contribs": [ "Turing machine", "Turing test", "Turingery" ], "views": 1250000}]
```

Query Arrays

The following operation returns documents in the people collection where the array field `contribs` contains the element "Turingery" or "Lisp":

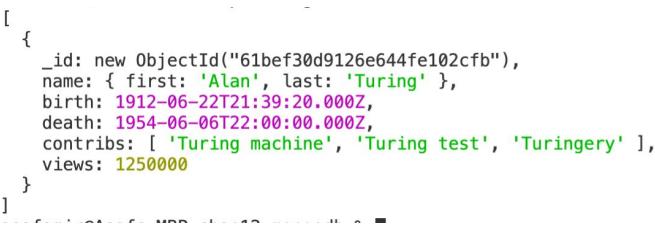
```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await col.find(
    {"contribs": "Turing test" }).toArray();
  // Print to the console
  console.log(myDoc);
}
```

```
[{"_id": new ObjectId("61bef30d9126e644fe102cfb"),
  name: { first: 'Alan', last: 'Turing' },
  birth: 1912-06-22T21:39:20.000Z,
  death: 1954-06-06T22:00:00.000Z,
  contribs: [ 'Turing machine', 'Turing test', 'Turingery' ],
  views: 1250000
},
 {"_id": new ObjectId("61bf1133176f849e321250d0"),
  name: { first: 'Mike', last: 'Lewis' },
  birth: 1942-06-21T21:00:00.000Z,
  death: 1980-07-07T22:00:00.000Z,
  contribs: [ 'Turingery' ],
  views: 130000
}]
```

Query Arrays

The following operation use the `$all` query operator to return documents in the collection where the array field `contribs` contains both the elements "Turingery" and "Turing test":

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await col.find(
    { contribs: { $all: [ "Turingery", "Turing test" ] } }).toArray();
  // Print to the console
  console.log(myDoc);
}
```



```
[{"_id": new ObjectId("61bef30d9126e644fe102cfb"), "name": { "first": "Alan", "last": "Turing" }, "birth": 1912-06-22T21:39:20.000Z, "death": 1954-06-06T22:00:00.000Z, "contribs": [ "Turing machine", "Turing test", "Turingery" ], "views": 1250000}]
```

Query Arrays

The following operation uses the \$size operator to return documents in the bios collection where the array size of contribs is 4:

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await col.find(
    { contribs: { $size: 3 } }).toArray();
  // Print to the console
  console.log(myDoc);
}
```

```
[
  {
    _id: new ObjectId("61bef30d9126e644fe102cfb"),
    name: { first: 'Alan', last: 'Turing' },
    birth: 1912-06-22T21:39:20.000Z,
    death: 1954-06-06T22:00:00.000Z,
    contribs: [ 'Turing machine', 'Turing test', 'Turingery' ],
    views: 1250000
  }
]
```

sort

The `sort()` method orders the documents in the result set. The following operation returns documents in the `people` collection sorted in ascending order by the `name.first` field:

ascending (1) vs descending (-1) order

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await col.find().sort({ 'name.first': 1 }).toArray();
  // Print to the console
  console.log(myDoc);
}
```

```
[
  {
    _id: new ObjectId("61bef30d9126e644fe102cfb"),
    name: { first: 'Alan', last: 'Turing' },
    birth: 1912-06-22T21:39:20.000Z,
    death: 1954-06-06T22:00:00.000Z,
    contribs: [ 'Turing machine', 'Turing test', 'Turingery' ],
    views: 125000
  },
  {
    _id: new ObjectId("61bf1145651d237d271b528c"),
    name: { first: 'Lisa', last: 'Mine' },
    birth: 1971-09-21T22:00:00.000Z,
    death: 1967-10-08T22:00:00.000Z,
    contribs: [ 'Turing machine' ],
    views: 145000
  },
  {
    _id: new ObjectId("61bf1133176f849e321250d0"),
    name: { first: 'Mike', last: 'Lewis' },
    birth: 1942-06-21T21:00:00.000Z,
    death: 1980-07-07T22:00:00.000Z,
    contribs: [ 'Turingery' ],
    views: 130000
  }
]
```

Limit

The limit() method limits the number of documents in the result set. The following operation returns at most 2 documents in the collection:

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await col.find().limit(2).toArray();
  // Print to the console
  console.log(myDoc);
}
```

```
[  
  {  
    _id: new ObjectId("61bef30d9126e644fe102cfb"),  
    name: { first: 'Alan', last: 'Turing' },  
    birth: 1912-06-22T21:39:20.000Z,  
    death: 1954-06-06T22:00:00.000Z,  
    contribs: [ 'Turing machine', 'Turing test', 'Turingery' ],  
    views: 1250000  
  },  
  {  
    _id: new ObjectId("61bf1133176f849e321250d0"),  
    name: { first: 'Mike', last: 'Lewis' },  
    birth: 1942-06-21T21:00:00.000Z,  
    death: 1980-07-07T22:00:00.000Z,  
    contribs: [ 'Turingery' ],  
    views: 130000  
  }  
]
```

Do it yourself 10

Open a file named b10.js and write a suitable code that will print to the console all the records, limit the number of records to 2.

Help with the command:

```
find().toArray();
```

Delete - link

The deleteOne() method:

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await
  col.deleteOne({_id:ObjectId("61bef30d9126e644fe102cfb")})
    // Print to the console
  console.log(myDoc);
  await client.close();
}
```

Before

QUERY RESULTS 1-3 OF 3

	_id	name	birth	death	contribs	views
1	_id: ObjectId("61bef30d9126e644fe102cfb")					
2	_id: ObjectId("61bf1133176f849e321250d0")	Object	1912-06-22T21:39:20.000+00:00	1954-06-06T22:00:00.000+00:00	Array	1250000
3	_id: ObjectId("61bf1145651d237d271b528c")	Object	1942-06-21T21:00:00.000+00:00	1988-07-07T22:00:00.000+00:00	Array	130000

delete

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  const myDoc = await
col.deleteOne({ id:ObjectId("61bef30d9126e644fe102cfb") })
  // Print to the console
  console.log(myDoc);
  await client.close();
}
```

After

QUERY RESULTS 1-2 OF 2

```
_id: ObjectId("61bf1133176f849e321250d0")
> name: Object
  birth: 1942-06-21T21:00:00.000+00:00
  death: 1980-07-07T22:00:00.000+00:00
> contribs: Array
  views: 130000
```

```
_id: ObjectId("61bf1145651d237d271b528c")
> name: Object
  birth: 1971-09-21T22:00:00.000+00:00
  death: 1967-10-08T22:00:00.000+00:00
> contribs: Array
  views: 145000
```

Update synchronous

The update method:

```
try
{
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  var myquery = { "views":130000};
  var newvalues = { $set: {views:172123} } ;
  await col.updateOne(myquery,newvalues)
  // Print to the console
  console.log("updated");
  await client.close();
}
```

before

```
_id: ObjectId("61bf1133176f849e321250d0")
> name: Object
  birth: 1942-06-21T21:00:00.000+00:00
  death: 1980-07-07T22:00:00.000+00:00
> contribs: Array
  views: 130000
```

```
_id: ObjectId("61bf1145651d237d271b528c")
> name: Object
  birth: 1971-09-21T22:00:00.000+00:00
  death: 1967-10-08T22:00:00.000+00:00
> contribs: Array
  views: 145000
```

After

```
_id: ObjectId("61bf1133176f849e321250d0")
> name: Object
  birth: 1942-06-21T21:00:00.000+00:00
  death: 1980-07-07T22:00:00.000+00:00
> contribs: Array
  views: 173123
```

```
_id: ObjectId("61bf1145651d237d271b528c")
> name: Object
  birth: 1971-09-21T22:00:00.000+00:00
  death: 1967-10-08T22:00:00.000+00:00
> contribs: Array
  views: 145000
```

Async function - Reminder from the js -link

An async function is a function declared with the `async` keyword, and the `await` keyword is permitted within it. The `async` and `await` keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

Update asynchronous

```
client.connect((err) => {
  if(err) console.log(err)
  const db = client.db("test");
  // Use the collection "people"
  const col = db.collection("people");
  // Find one document
  var myquery = { "views":130000 };
  var newvalues = { $set: {views:172123} };
  col.updateOne(myquery,newvalues,function(err, res) {
    if (err) throw err;
    console.log("1 document updated");
    client.close();
  })
  // Print to the console
})
```

before

```
_id:ObjectId("61bf1133176f849e321250d0")
> name:Object
birth:1942-06-21T21:00:00.000+00:00
death:1980-07-07T22:00:00.000+00:00
> contribs:Array
views:130000
```

```
_id:ObjectId("61bf1145651d237d271b528c")
> name:Object
birth:1971-09-21T22:00:00.000+00:00
death:1967-10-08T22:00:00.000+00:00
> contribs:Array
views:145000
```

After

```
_id:ObjectId("61bf1133176f849e321250d0")
> name:Object
birth:1942-06-21T21:00:00.000+00:00
death:1980-07-07T22:00:00.000+00:00
> contribs:Array
views:173123
```

```
_id:ObjectId("61bf1145651d237d271b528c")
> name:Object
birth:1971-09-21T22:00:00.000+00:00
death:1967-10-08T22:00:00.000+00:00
> contribs:Array
views:145000
```

Do it yourself 11

Open a file named b11.js and write an appropriate code that will update the records that cost 20 to 30. Help with updateMany or updateOne:



braintop
think.make.play

Chapter 13 - Mongoose

Mongoose queries - mongoose

Link to Mongoose Queries

The screenshot shows the Mongoose documentation website. The left sidebar includes links for Quick Start, Guides, Schemas, SchemaTypes, Connections, Models, Documents, Subdocuments, and Queries. Under Queries, there are sub-links for Query Casting, findOneAndUpdate, The Lean Option, Validation, Middleware, Populate, Discriminators, Plugins, Transactions, TypeScript, and API. A callout for AG Grid is present. The main content area is titled 'Queries' and contains a list of static helper functions for Mongoose models. Below the list, it explains how to execute a query and provides examples of promises.

Queries

Mongoose [models](#) provide several static helper functions for [CRUD operations](#). Each of these functions returns a [mongoose `Query` object](#).

- `Model.deleteMany()`
- `Model.deleteOne()`
- `Model.find()`
- `Model.findById()`
- `Model.findByIdAndUpdate()`
- `Model.findByIdAndDelete()`
- `Model.findByIdAndUpdate()`
- `Model.findByIdAndUpdateAndRemove()`
- `Model.findByIdAndUpdateAndReplace()`
- `Model.findOne()`
- `Model.findOneAndDelete()`
- `Model.findOneAndRemove()`
- `Model.findOneAndReplace()`
- `Model.findOneAndUpdate()`
- `Model.replaceOne()`
- `Model.updateMany()`
- `Model.updateOne()`

A mongoose query can be executed in one of two ways. First, if you pass in a `callback` function, Mongoose will execute the query asynchronously and pass the results to the `callback`.

A query also has a `.then()` function, and thus can be used as a promise.

- Executing
- Queries are Not Promises
- References to other documents
- Streaming
- Versus Aggregation

Queries

Mongoose [models](#) provide several static helper functions for [CRUD operations](#). Each of these functions returns a [mongoose `Query` object](#).

- `Model.deleteMany()`
- `Model.deleteOne()`
- `Model.find()`
- `Model.findById()`
- `Model.findByIdAndUpdate()`
- `Model.findByIdAndUpdateAndDelete()`
- `Model.findByIdAndUpdateAndRemove()`
- `Model.findByIdAndUpdateAndReplace()`
- `Model.findOne()`
- `Model.findOneAndDelete()`
- `Model.findOneAndRemove()`
- `Model.findOneAndReplace()`
- `Model.findOneAndUpdate()`
- `Model.replaceOne()`
- `Model.updateMany()`
- `Model.updateOne()`

Schema

Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.

Schema

Create new folder named (select a name) and Run **npm init** in the terminal, and then **npm install mongoose** and **npm install express**.

Create file **app.js**

Create file **ProductModel.js**

MongoDB Schema example - ProductModel.js

```
var mongoose = require("mongoose");
var Schema = mongoose.Schema;
var ProductSchema = new Schema({
  title:String,
  Description:String,
  price:Number
  created: Date
});
module.exports = mongoose.model('product', ProductSchema);
```

Create a new file with the model name you want to save in the db (ProductModel in our case):

JS ProductModel.js

Schema define the fields of the model

Export the schema so other modules could use it

The collection name in mongoDB will be 'product'

MongoDB Schema Add Validators - ProductModel.js

```
var mongoose = require("mongoose");
var Schema = mongoose.Schema;
var ProductSchema = new Schema({
  title: {
    type:String,
    required:[true,'A product must have title'],
    unique:true,
    trim:true
  },
  description:{
    type:String,
    minlength:[5,'Description is minimum 20 characters'],
    maxlength:[1000,'Description is maximum 1000 characters']
  },
  price:{
    type:Number,
    required:[true, 'A product must have price'],
    min:[0,'price must be above 0'],
    max:[10000,'price must be below 10000']
  },
  created: Date
});
module.exports = mongoose.model('product', ProductSchema);
```

You can add
validators

Validators

Insert Data to MongoDB - app.js

```
var express = require('express');
var app = express();
app.use(express.json());
var CurrentProduct = require('./ProductModel');
var mongoose = require('mongoose');
const strConnect =
"mongodb+srv://asaf:asaf@cluster0.hqfhv.mongodb.net/myFirstDatabase?retryWrites=true&w=majority";
const OPT = {
  useNewUrlParser: true
};
```

Get the ProductModel
we created

Use mongoDB
API - [link](#)

The MongoDB Node.js driver rewrote the tool it uses to parse [MongoDB connection strings](#). Because this is such a big change, they put the new connection string parser behind a flag. To turn on this option, pass the `useNewUrlParser` option to `mongoose.connect()` or `mongoose.createConnection()`.

Continue next slide

This should be **your own**
connection string to the db
(go to the first slide in this
chapter to see how to get it)

Insert Data to MongoDB -app.js

```
app.post('/api/v1/products', function(req, res, next) {  
  let p1= req.body;  
  console.log(req.body)  
  var newItem = new CurrentProduct(p1);  
  newItem.save().then(item=>{  
    res.json({item:item})  
  }).catch(err=>{  
    console.log("error 🤦 :" +err)  
  });  
});
```

Continue next slide

Insert Data to MongoDB - app.js

```
mongoose.connect(strConnect, OPT);  
var port = process.env.PORT || 3000;  
app.listen(port, function() {  
    console.log("Running on port " + port);  
})
```

Connect to mongoDB
using mongoose

Insert Data to MongoDB

The screenshot shows a Postman interface for a POST request to `localhost:3000/api/v1/products`. The request body is a JSON object:

```
1 | {
2 |   "title": "Red ball",
3 |   "description": "This is a red ball",
4 |   "created": "10/10/2023",
5 |   "price": 300
6 | }
```

The response status is 200 OK with a response time of 189 ms and a size of 390 B. The response body is also a JSON object:

```
1 | {
2 |   "item": {
3 |     "title": "Red ball",
4 |     "description": "This is a red ball",
5 |     "price": 300,
6 |     "created": "2023-10-09T21:00:00.000Z",
7 |     "_id": "61dc1f9e8693b53f3e4f461f",
8 |     "__v": 0
9 |   }
10 | }
```

Run your application using the command **node app** in the terminal.
`localhost:3000/api/v1/products`

Insert Data to MongoDB

Under your project in the MongoDB website go to [clusters](#) -> [collections](#).

And you can see the newly added product under products:

Insert Data to MongoDB using await and async

```
app.post('/api/v1/products', async function(req, res, next) {  
  try{  
    let p1= req.body;  
    var newItem = await CurrentProduct.create(p1);  
    res.status(201).json({  
      status:"success",  
      data:newItem  
    })  
  }  
  catch(err){  
    res.status(400).json({  
      status:"fail",  
      message:"error: "+err  
    })  
  }  
});
```

Do it yourself 1

1. Create a folder that contains an app.js file
2. Add to the folder a model named PersonModel.js that contains the following fields: **first name, family, city, country, salary**
3. Add a new function that insert person to db
4. Run the application and insert 3 persons to db
5. Check in mongodb that the entries have been entered

Get All Data from MongoDB

Add a new function to retrieve the data from the db: **Rerun the application and navigate to /getall**

```
app.get('/api/v1/products', function(req, res, next){  
  CurrentProduct.find({}).then(function(data){  
    res.status(200).json({  
      status:"success",  
      data:data  
    })  
  }).catch(err=>{  
    res.status(404).json({  
      status:"fail",  
      message:"error:😱" + err  
    })  
  })  
})
```

The find function is empty because we don't want to filter the data, but get it all

Get All Data from MongoDB

Add a new function to retrieve the data from the db: **Rerun the application and navigate to** `localhost:3000/api/v1/products`

The screenshot shows a Postman request configuration and its resulting JSON response.

Request Configuration:

- Method: GET
- URL: `localhost:3000/api/v1/products`
- Send button is visible

Query Params:

KEY	VALUE	DESCRIPTION
Key	Value	Description

Response Headers:

- Body
- Cookies
- Headers (7)
- Test Results

Status: 200 OK | 1013 ms | 411 B | Save Response

Pretty **Raw** **Preview** **Visualize** **JSON**

JSON Response:

```
1 "status": "success",
2 "data": [
3     {
4         "_id": "61dc1f9e8693b53f3e4f461f",
5         "title": "Red ball",
6         "description": "This is a red ball",
7         "price": 300,
8         "created": "2023-10-09T21:00:00.000Z",
9         "__v": 0
10    }
11 ]
12 ]
13 ]
```

Do it yourself 2

1. Add a new function to app.js to retrieve the data from the db
2. Rerun the application and navigate to localhost:3000/api/v1/products

Get Data by id from MongoDB

Add a new function to retrieve the data from the db: Rerun the application and navigate to localhost:3000/api/v1/products/61dc1f9e8693b53f3e4f461f

```
app.get('/api/v1/products/:id', function(req, res, next) {
  let id = req.params.id
  CurrentProduct.find({_id:id}).then(function(data) {
    res.status(200).json({
      status:"success",
      data:data
    })
  }).catch(err=>{
    res.status(404).json({
      status:"fail",
      message:"error: "+err
    })
  })
})
```

Filter by id

Get Data by id from MongoDB

Add a new function to retrieve the data from the db: **Rerun the application and navigate to** `localhost:3000/api/v1/products/61dc1f9e8693b53f3e4f461f`

GET `localhost:3000/api/v1/products/61dc1f9e8693b53f3e4f461f` Send

Params Authorization Headers (8) Body ● Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (7) Test Results

Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "status": "success",
3   "data": [
4     {
5       "_id": "61dc1f9e8693b53f3e4f461f",
6       "title": "Red ball",
7       "description": "This is a red ball",
8       "price": 300,
9       "created": "2023-10-09T21:00:00.000Z",
10      "__v": 0
11    }
12  ]
13 }
```

Write here id of your object

Do it yourself 3

1. Add a new function to app.js to retrieve the person by _id from the db
2. Rerun the application and navigate to localhost:3000/api/v1/products/_----- (select an id)

Update Data

Add a new function to retrieve the data from the db: **Rerun the application and navigate to**

```
app.patch('/api/v1/products/:id', function(req, res, next){  
  let id = req.params.id  
  CurrentProduct.findByIdAndUpdate(id, req.body, {new:true, runValidators:true})  
  .then(function(data){  
    res.status(200).json({  
      status:"success",  
      data:data  
    })  
  }).catch(err=>{  
    res.status(404).json({  
      status:"fail",  
      message:"error: "+err  
    })  
  })  
})
```

New true and runvalidators true means its return new object after update parameters

Read more about `findByIdAndUpdate`:
https://mongoosejs.com/docs/api.html#model_Model.findByIdAndUpdate

Update Data

Add a new function to update the data from the db: Rerun the application and navigate to localhost:3000/api/v1/products/61dc1f9e8693b53f3e4f461f

Update the price to 400

Write here id of your object

The screenshot shows a Postman interface for a PATCH request to the URL `localhost:3000/api/v1/products/61dc1f9e8693b53f3e4f461f`. The 'Body' tab is selected, containing the following JSON payload:

```
1 {  
2   ...  
3     "title": "Red ball",  
4     "description": "This is a red ball",  
5     "created": "10/10/2023",  
6     "price": 400  
7 }
```

The response section shows a 200 OK status with a response time of 196 ms and a response body of 409 B. The response body is:

```
1 {  
2   "status": "success",  
3   "data": {  
4     "_id": "61dc1f9e8693b53f3e4f461f",  
5     "title": "Red ball",  
6     "description": "This is a red ball",  
7     "price": 400,  
8     "created": "2023-10-09T21:00:00.000Z",  
9     "__v": 0  
10 }  
11 }
```

Do it yourself 4

1. Add a new function to app.js to update the person by _id
2. Rerun the application and navigate to localhost:3000/api/v1/products/_----- (select an id and run from postman patch request and update the salary to 10000 for id selected)

Delete Data

Add a new function to retrieve the data from the db: **Rerun the application and navigate to**

```
app.delete('/api/v1/products/:id', function(req, res, next){  
  let id = req.params.id  
  CurrentProduct.findByIdAndDelete(id)  
    .then(function(data){  
      res.status(404).json({  
        status:"success",  
        data:null  
      })  
    }).catch(err=>{  
      res.status(404).json({  
        status:"fail",  
        message:"error:😱" + err  
      })  
    })  
})
```

Delete Data

Add a new function to delete the data from the db: **Rerun the application and navigate to** <localhost:3000/api/v1/products/61dc1f9e8693b53f3e4f461f>

Delete : 61dc1f9e8693b53f3e4f461f

Write here id of your object

The screenshot shows the Postman interface with a DELETE request to the URL `localhost:3000/api/v1/products/61dc1f9e8693b53f3e4f461f`. The Body tab is selected, displaying the following JSON payload:

```
1
2   ...
3   "title": "Red ball",
4   ...
5   "description": "This is a red ball",
6   ...
7   "created": "10/10/2023",
8   ...
9   "price": 400
```

The response section shows a 404 Not Found status with a timestamp of 1402 ms and a size of 274 B. The response body is:

```
1
2   "status": "success",
3   "data": null
4
```

Do it yourself 5

1. Add a new function to app.js to delete person by _id
2. Rerun the application and navigate to localhost:3000/api/v1/products/_----- (select an id and run from postman delete request and delete the selected person)

Read and filter Data

Update get `/api/v1/products` function to filter the data from the db: Rerun the application and navigate to

- In the future we would like to filter by limit, page, sort and more!
- For example:
`localhost:3000/api/v1/products?price=500&sort=price&fields=title,price&limit=10&page=2`
- In the query above we want all products **price** = 500 ,**sort** according to price, on **page** 2 and a **limit** of 10.
- Mongoose does not know limit, sort, fields and page, so we need to remove them from the queryObj.
- Limit - The number of records for each page
- Page - Represent the page we want
- Sort - sort according to specific field
- Fields - fields we want to retrieve

Read and filter Data

Update get `/api/v1/products` function to filter the data from the db: Rerun the application and navigate to

Example query : localhost:3000/api/v1/products?price=500&page=2

```
app.get('/api/v1/products',async function(req, res, next) {
  console.log("hello")
  let queryObj = {...req.query}
  let withOutFields = ['page', 'sort', 'limit', 'fields']
  withOutFields.forEach(el => {
    delete queryObj[el]
  });
  CurrentProduct.find(queryObj).then(function(data) {
    res.status(200).json({
      status:"success",
      data:data
    })
  }).catch(err=>{
    res.status(404).json({
      status:"fail",
      message:"error:💀 :" + err
    })
  })
})
```



Remove
sort,limit,
page,fields
from query

Read and filter Data

Update get `/api/v1/products` function to filter the data from the db: Rerun the application and navigate to

Example query : localhost:3000/api/v1/products?price=500&page=2

```
let queryObj = { ...req.query } ← Retrieve array of param
```

```
console.log(queryObj) // { price: '500', page: '2' }
```

```
let withOutFields = ['page', 'sort', 'limit', 'fields']
withOutFields.forEach(el => {
  delete queryObj[el] ← Remove page,sort,limit,fields from queryObj.
});
```

```
console.log(queryObj) // { price: '500' }
```

{ price: '500', page: '2' } ← Before without fields ← Later we will use 'page' field too.
{ price: '500' } ← After without fields

Read and filter Data

Update get `/api/v1/products` function to filter the data from the db: Rerun the application and navigate to

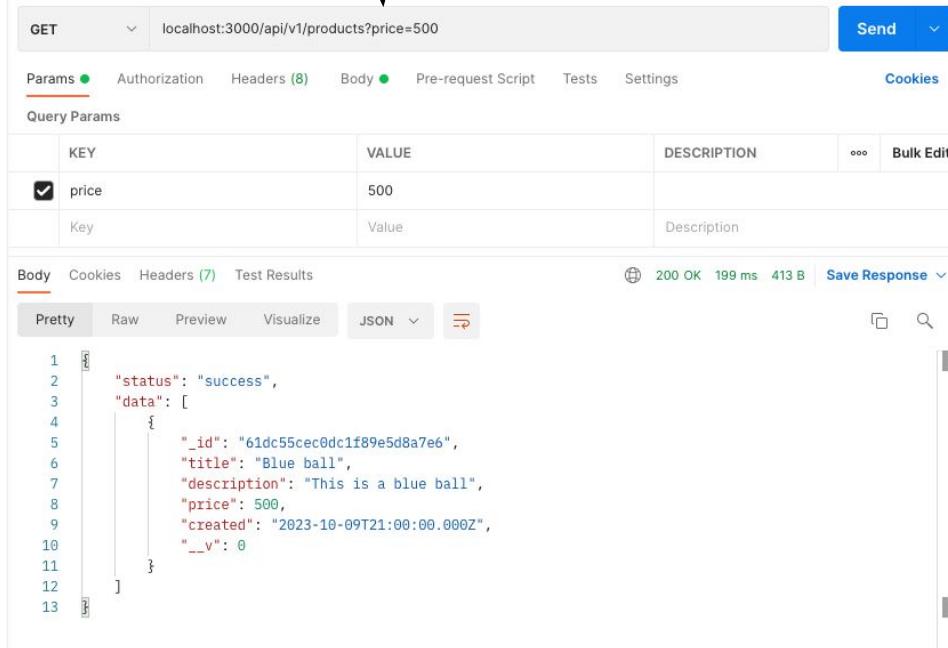
Example query : localhost:3000/api/v1/products?price=500&page=2

```
app.get('/api/v1/products',async function(req, res, next) {
  let queryObj = {...req.query}//
  let withOutFileds = ['page', 'sort', 'limit', 'fields']
  withOutFileds.forEach(el => {
    delete queryObj[el]
  });
  CurrentProduct.find(queryObj).then(function(data) {
    res.status(200).json({
      status:"success",
      data:data
    })
  }).catch(err=>{
    res.status(404).json({
      status:"fail",
      message:"error: 😱 :" + err
    })
  })
})
```

Filter data

Rerun the application and navigate to `localhost:3000/api/v1/products?price=500`

All products with price 500



The screenshot shows the Postman interface with a GET request to `localhost:3000/api/v1/products?price=500`. The 'Params' tab is selected, showing a single query parameter `price` with a value of `500`. The response body is a JSON object:

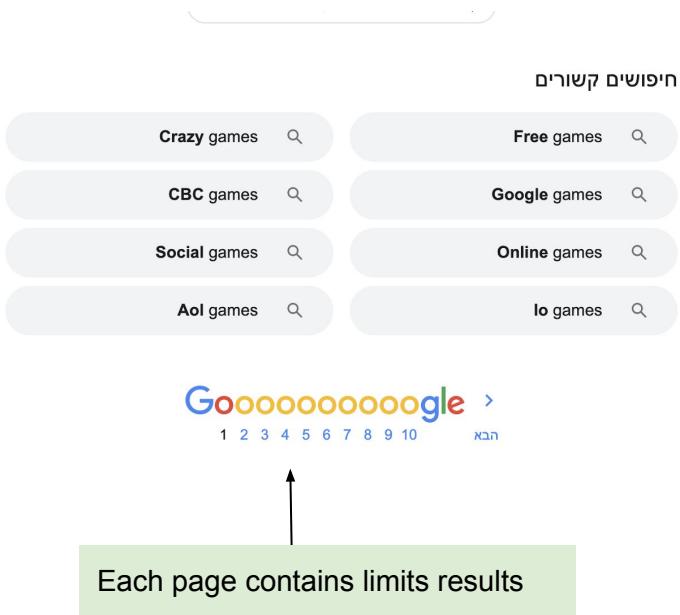
```
1 {  
2   "status": "success",  
3   "data": [  
4     {  
5       "_id": "61dc55cec0dc1f89e5d8a7e6",  
6       "title": "Blue ball",  
7       "description": "This is a blue ball",  
8       "price": 500,  
9       "created": "2023-10-09T21:00:00.000Z",  
10      "__v": 0  
11    }  
12  ]  
13 }
```

Do it yourself 6

1. Update the function from exercise 2 and to filter data from db.
2. Rerun the application and navigate to localhost:3000/api/v1/persons/ and retrieve all person with salary 1000
3. Rerun the application and navigate to localhost:3000/api/v1/persons/ and retrieve all person with salary firstname Mike

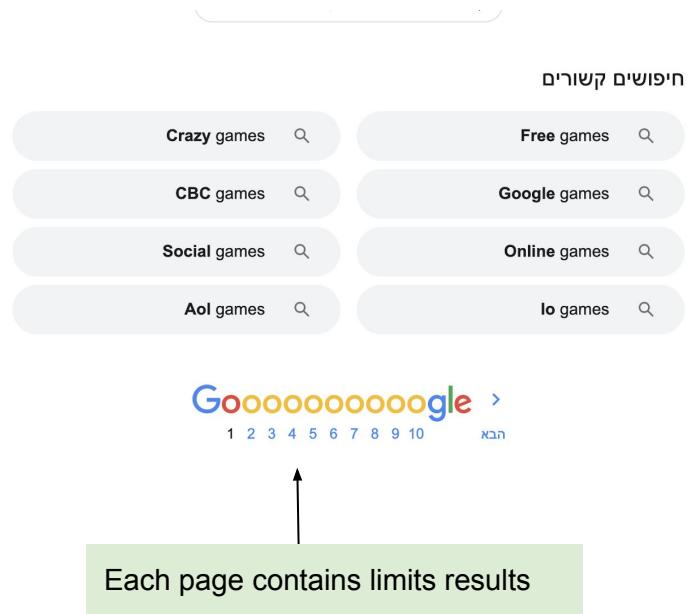
pagination & limit

Pagination is a method of dividing web content into discrete pages, thus presenting content in a limited and digestible manner. Google search results page is a typical example of such a search.



Mongoose - pagination & limit

Mongodb does not know how to handle pagination and sort. During the following slides we will learn how to handle pagination, limit, sort and more fields.



Advance Read and filter Data

Update get `/api/v1/products` function to filter the data from the db: Rerun the application

```
app.get('/api/v1/products',async function(req, res, next){  
    //phase 1 - filtering  
    let queryObj = {...req.query}  
    let withOutFileds = ['page', 'sort', 'limit', 'fields']  
    withOutFileds.forEach(el => {  
        delete queryObj[el]  
    });  
});
```

Mongodb queries doesn't know how handle : page, sort limit and fields

Lets remove `'page', 'sort',
'limit', 'fields'` from `queryObj`

Advance Read and filter Data

Update get `/api/v1/products` function to filter the data from the db: **Rerun the application**

```
app.get('/api/v1/products',async function(req, res, next){  
    //phase 1 - filtering  
    let queryObj = {...req.query}  
    let withOutFileds = ['page', 'sort', 'limit', 'fields']  
    withOutFileds.forEach(el => {  
        delete queryObj[el]  
    });  
    //phase 2 - advance filtering  
    let strQuery = JSON.stringify(queryObj)  
    strQuery = strQuery.replace(/\b(gt|gte|lt|lte)\b/g,match =>`$$ ${match}`) ← Regular  
    queryObj = JSON.parse(strQuery)  
    console.log(queryObj)  
    CurrentProduct.find(queryObj).then(function(data){  
        res.status(200).json({  
            status:"success",  
            data:  
        })  
    }).catch(err=>{  
        res.status(404).json({  
            status:"fail",  
            message:"error: " + err  
        })  
    })  
})
```

Before replace :{ price: { gte: '70', lt: '400' } }

Regular expression

After replace :{ price: { '\$gte': '70', '\$lt': '400' } }

Advance Read and filter Data

Update get `/api/v1/products` function to filter the data from the db: Rerun the application

```
app.get('/api/v1/products',async function(req, res, next){  
    //phase 1 - filtering  
    let queryObj = {...req.query}          { price: { gte: '70', lt: '400' } }  
    let withOutFileds = ['page', 'sort', 'limit', 'fields']  
    withOutFileds.forEach(el => {  
        delete queryObj[el]  
    });  
    //phase 2 - advance filtering  
    let strQuery = JSON.stringify(queryObj)  
    strQuery = strQuery.replace(/\b(gt|gte|lt|lte)\b/g,match =>`$${match}`)  
    queryObj = JSON.parse(strQuery)  
    console.log(queryObj)//{ price: { $gte: '70', $lt: '400' } }  
  
    CurrentProduct.find(queryObj).then(function(data) {  
        res.status(200).json({  
            status:"success",  
            data:data  
        })  
    }).catch(err=>{  
        res.status(404).json({  
            status:"fail",  
            message:"error: 😱 : " + err  
        })  
    })  
})
```

We need to add \$ before gte and lt

In order to add \$ sign we will use regular expression

Now we have \$ sign before gte and lt. And we can run find function;

Advance Filter data

Rerun the application and navigate to `localhost:3000/api/v1/products?price[gte]=70&price[lt]=400`

All products greater or equal to 70 and less than 400

The screenshot shows the Postman application interface. At the top, there is a URL bar with the URL `localhost:3000/api/v1/products?price[gte]=70&price[lt]=400`. Below the URL bar, the method is set to `GET`, and the full URL is displayed again. To the right of the URL, there are `Save`, `Edit`, and `Send` buttons. The `Send` button is highlighted in blue.

Below the URL bar, there are tabs for `Params`, `Authorization`, `Headers (8)`, `Body`, `Pre-request Script`, `Tests`, and `Settings`. The `Params` tab is selected, showing two query parameters: `price[gte]` with value `70` and `price[lt]` with value `400`.

Under the `Body` tab, the response is displayed as a JSON object:

```
1
2   "status": "success",
3   "data": [
4     {
5       "_id": "61dc55b4c0dc1f89e5d8a7e1",
6       "title": "Red ball",
7       "description": "This is a red ball",
8       "price": 300,
9       "created": "2023-10-09T21:00:00.000Z",
10      "__v": 0
11    }
12  ]
```

The response status is `200 OK`, the time taken is `177 ms`, and the size is `411 B`. There is also a `Save Response` button.

Do it yourself 7

1. Update the function from exercise 5 and to advance filter data from db.
2. Rerun the application and navigate to localhost:3000/api/v1/persons/ and retrieve all persons with salary above 1000
3. Rerun the application and navigate to localhost:3000/api/v1/persons/ and add parameters to the query that retrieve all persons with salary below 1000
4. Rerun the application and navigate to localhost:3000/api/v1/persons/ and add parameters to the query that retrieve all persons with salary between 1000 to 2000

Sort Data

Update get `/api/v1/products` function to filter the data from the db: Rerun the application and navigate to

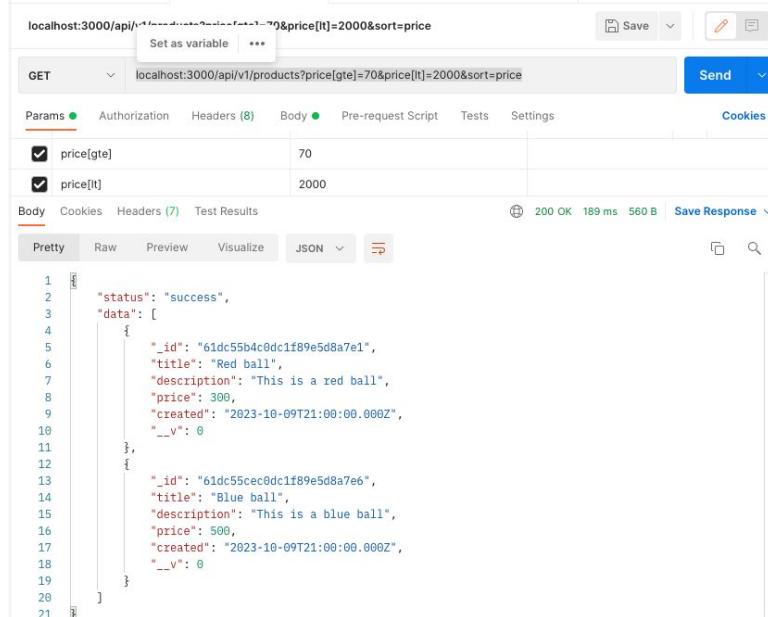
```
app.get('/api/v1/products',async function(req, res, next){  
    //phase 1 - filtering  
    let queryObj = {...req.query}  
    let withOutFileds = ['page', 'sort', 'limit', 'fields']  
    withOutFileds.forEach(el => {  
        delete queryObj[el]  
    });  
    //phase 2 - advance filtering  
    let strQuery = JSON.stringify(queryObj)  
    strQuery = strQuery.replace(/\b(gte|gt|lte|lt)\b/g,match =>`$${match}`)  
    queryObj = JSON.parse(strQuery)  
    let sort="";  
    if(req.query.sort){  
        sort = req.query.sort.split(',').join(' ')// add more sorts ←  
    }  
    CurrentProduct.find(queryObj).sort(sort).then(function(data){  
        res.status(200).json({  
            status:"success",  
            data:data  
        })  
    }).catch(err=>{  
        res.status(404).json({  
            status:"fail",  
            message:"error: 😬 :" + err  
        })  
    })  
})
```

If sort - maybe you will want more than one sort.

Sort data

Rerun the application and navigate to `localhost:3000/api/v1/products?price[gte]=70&price[lt]=2000&sort=price`

All products greater or equal to 70 and less than 500 and sort according to price



```
1
2     "status": "success",
3     "data": [
4         {
5             "_id": "61dc55b4c0dc1f89e5d8a7e1",
6             "title": "Red ball",
7             "description": "This is a red ball",
8             "price": 300,
9             "created": "2023-10-09T21:00:00.000Z",
10            "_v": 0
11        },
12        {
13            "_id": "61dc55cec0dc1f89e5d8a7e6",
14            "title": "Blue ball",
15            "description": "This is a blue ball",
16            "price": 500,
17            "created": "2023-10-09T21:00:00.000Z",
18            "_v": 0
19        }
20    ]
21 ]
```

Do it yourself 8

1. Update the function from exercise 6 and to and add the option to sort.
2. Rerun the application and navigate to localhost:3000/api/v1/persons/ and retrieve all persons sorting according to salary

Remove fields from response data

Update get function `/api/v1/products` to filter the data from the db

```
app.get('/api/v1/products',async function(req, res, next){  
    //phase 1 - filtering  
    let queryObj = {...req.query}  
    let withOutFileds = ['page', 'sort', 'limit', 'fields']  
    withOutFileds.forEach(el => {  
        delete queryObj[el]  
    });  
    //phase 2 - advance filtering  
    let strQuery = JSON.stringify(queryObj)  
    strQuery = strQuery.replace(/\b(gte|gt|lte|lt)\b/g,match :  
    queryObj = JSON.parse(strQuery)  
    console.log(queryObj)  
    let sort="";  
    let selected = "";  
    if(req.query.sort){  
        sort = req.query.sort.split(',').join(' ')// add more sorts  
    }  
}
```

If sort - maybe you will want more than one sort.

Continue ->

Remove fields from response data

```
if(req.query.fields){  
    selected = req.query.fields.split(',') .join(' ') // show fields  
    //--fields will remove a fields  
}  
  
CurrentProduct.find(queryObj).select(selected).sort(sort).then(function(data){  
    res.status(200).json({  
        status:"success",  
        data:data  
    })  
}).catch(err=>{  
    res.status(404).json({  
        status:"fail",  
        message:"error: 🤦‍♂️ :" + err  
    })  
})  
})
```

Show the fields

Select - which to
fields to show

Remove fields from response data

Update get function to retrieve the data from the db: [Rerun the application and navigate to](#)

localhost:3000/api/v1/advancefilterproducts?price[gte]=70&price[lt]=200&sort=price&**fields=title,price**

The screenshot shows the Postman interface with a GET request to `localhost:3000/api/v1/advancefilterproducts`. The URL includes filters for price (gte 70, lt 200), sorting by price, and specifying fields as title and price. The response body is a JSON object with a "status" key and a "data" array containing two items, each with an _id, title, and price.

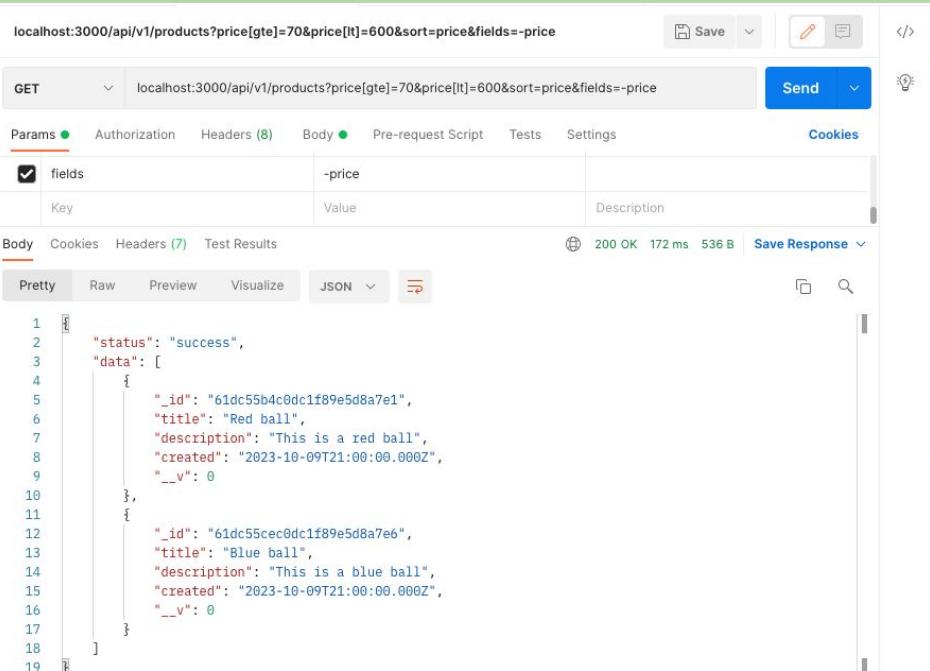
```
1 "status": "success",
2 "data": [
3     {
4         "_id": "61dc55b4c0dc1f89e5d8a7e1",
5         "title": "Red ball",
6         "price": 300
7     },
8     {
9         "_id": "61dc55cec0dc1f89e5d8a7e6",
10        "title": "Blue ball",
11        "price": 500
12    }
13 ]
14
15 ]
```

Show only : title and price

Remove fields from response data

Update get function to retrieve the data from the db: **Rerun the application and navigate to**

localhost:3000/api/v1/advancefilterproducts?price[gte]=70&price[lt]=200&sort=price&**fields=-price**



localhost:3000/api/v1/products?price[gte]=70&price[lt]=600&sort=price&fields=-price

GET localhost:3000/api/v1/products?price[gte]=70&price[lt]=600&sort=price&fields=-price Send

Params (8) Authorization Headers (8) Body (1) Pre-request Script Tests Settings Cookies

<input checked="" type="checkbox"/> fields	-price	
Key	Value	Description

Body Cookies Headers (7) Test Results 200 OK 172 ms 536 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "status": "success",
3   "data": [
4     {
5       "_id": "61dc55b4c0dc1f89e5d8a7e1",
6       "title": "Red ball",
7       "description": "This is a red ball",
8       "created": "2023-10-09T21:00:00.000Z",
9       "__v": 0
10    },
11    {
12      "_id": "61dc55cec0dc1f89e5d8a7e6",
13      "title": "Blue ball",
14      "description": "This is a blue ball",
15      "created": "2023-10-09T21:00:00.000Z",
16      "__v": 0
17    }
18  ]
19 }
```

Minus price : don't show price to the user

pagination

Update get `/api/v1/products` function to filter the data from the db

```
app.get('/api/v1/products',async function(req, res, next){
  //phase 1 - filtering
  let queryObj = {...req.query}
  let withOutFileds = ['page', 'sort', 'limit', 'fields']
  withOutFileds.forEach(el => {
    delete queryObj[el]
  });
  //phase 2 - advance filtering
  let strQuery = JSON.stringify(queryObj)
  strQuery = strQuery.replace(/\b(gte|gt|lte|lt)\b/g,match =>`${match}`)
  queryObj = JSON.parse(strQuery)
  console.log(queryObj)
  let sort="";
  let selected = "";
  if(req.query.sort){
    sort = req.query.sort.split(',').join(' ')// add more sorts
  }
  if(req.query.fields){
    selected = req.query.fields.split(',').join(' ')// show fields
  }
})
```

Continue ->

pagination

Update get `/api/v1/products` function to filter the data from the db

```
let limit = req.query.limit || 100 ← Limit per page - default 100
let page = req.query.page || 1 ← Selected page - default 1
let skip = (page-1)*limit
let documents = await CurrentProduct.countDocuments()
if(skip>=documents){
    res.status(404).json({
        status:"fail",
        data:"no data on this page and limit"
    })
}
return
```

Return 404 if the page doesn't exist

pagination

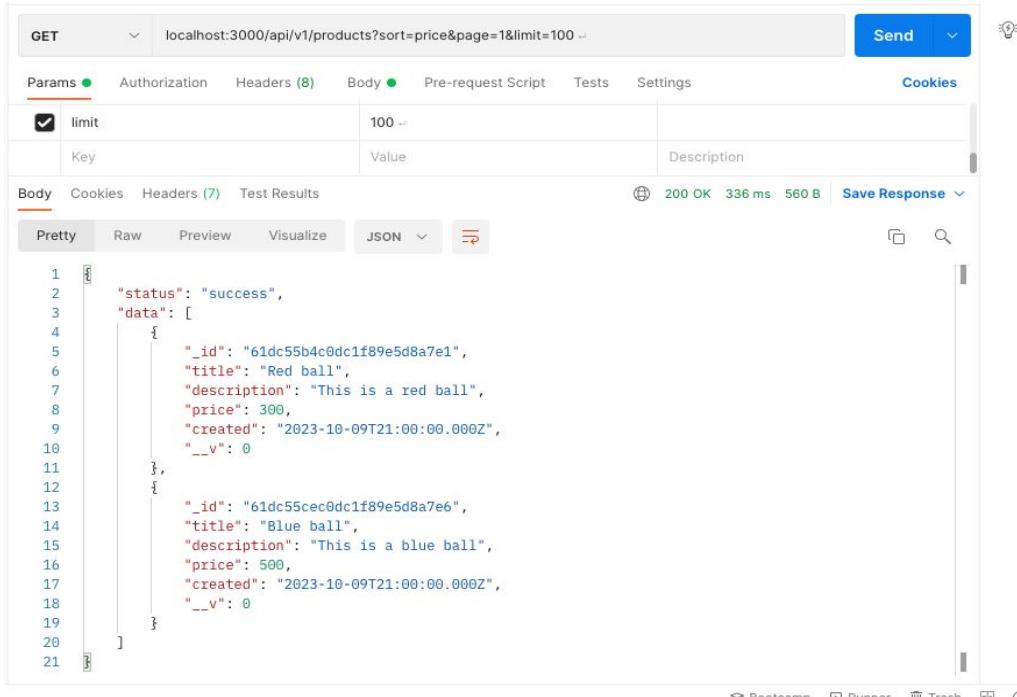
Update get `/api/v1/products` function to filter the data from the db

```
CurrentProduct.find(queryObj).select(selected).sort(sort).then(function(data) {
  res.status(200).json({
    status:"success",
    data:data
  })
}).catch(err=>{
  res.status(404).json({
    status:"fail",
    message:"error:😱 :" + err
  })
})
})
```

pagination

Add a new function to retrieve the data from the db: **Rerun the application and navigate to**

`localhost:3000/api/v1/products?sort=price&page=1&limit=100`



The screenshot shows a Postman request for a GET endpoint at `localhost:3000/api/v1/products?sort=price&page=1&limit=100`. The 'Params' tab has a checked 'limit' parameter set to 100. The response body is a JSON object:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
{
  "status": "success",
  "data": [
    {
      "_id": "61dc55b4c0dc1f89e5d8a7e1",
      "title": "Red ball",
      "description": "This is a red ball",
      "price": 300,
      "created": "2023-10-09T21:00:00.000Z",
      "__v": 0
    },
    {
      "_id": "61dc55cec0dc1f89e5d8a7e6",
      "title": "Blue ball",
      "description": "This is a blue ball",
      "price": 500,
      "created": "2023-10-09T21:00:00.000Z",
      "__v": 0
    }
  ]
}
```

aggregate

aggregate mongodb

Aggregation operations process multiple documents and return computed results. You can use aggregation operations to:

- Group values from multiple documents together.
- Perform operations on the grouped data to return a single result.
- Analyze data changes over time.

Aggregate - mongodb

The `$match` stage:

- Filters the documents to those with a status of urgent.
- Outputs the filtered documents to the `$group` stage.

The `$group` stage:

- Groups the input documents by `productName`.
- Uses `$sum` to calculate the total quantity for each `productName`, which is stored in the `sumQuantity` field returned by the aggregation pipeline.



Aggregate - link

Add new function for statistic `/api/v1/products/get/statistic`: Rerun the application and navigate to

```
app.get('/api/v1/products/get/statistic', function(req, res, next) {
  CurrentProduct.aggregate([
    {
      $match : {price:{$gt:30}} ←
    },
    {
      $group:{ ←
        _id:null,
        count:{$sum:1},
        avgPrice:{$avg:'$price'},
        minPrice:{$min:'$price'},
        maxPrice:{$max:'$price'}
      }
    },
    {
      $sort:{avgPrice:1} // 1 its asc, 0 its decn ←
    }
  ])
})
```

\$sort = sorting according to field : descending order or ascending order

The query - for example all products that price above 30

Group includes the statistic we want

\$sum = count of group

\$avg = the average of price

\$min = the min price

\$max = the max price

Add new function for statistic `/api/v1/products/get/statistic`: Rerun the application and navigate to

```
.then(function(data) {
  console.log(data)
  res.status(200).json({
    status:"success",
    data:data
  })
}).catch(err=>{
  res.status(404).json({
    status:"fail",
    message:"error: "+err
  })
})
```

The screenshot shows the Postman interface with the following details:

- URL:** localhost:3000/api/v1/products/get/statistic
- Method:** GET
- Body:** PRE-REQUEST SCRIPT (Body tab selected)
- Response:** PRETTY (Response tab selected)

```
1
2   "status": "success",
3   "data": [
4     {
5       "_id": null,
6       "count": 2,
7       "avgPrice": 400,
8       "minPrice": 300,
9       "maxPrice": 500
10    }
11  ]
12 ]
```
- Performance Metrics:** 200 OK 185 ms 332 B

Do it yourself 9

1. Add a function that displays statistics about people. The function will return the average salary, the minimum salary and the maximum salary
2. Rerun the application and navigate to `localhost:3000/api/v1/get/statistic/` and retrieve the average, max salary and max salary

Add middleware to MongoDB Schema

middleware mongodb

Middleware (also called pre and post *hooks*) are functions which are passed control during execution of asynchronous functions. Middleware is specified on the schema level and is useful for writing plugins.

Mongoose has 4 types of middleware:

1. document middleware,
2. query middleware
3. aggregate middleware,
4. model middleware,

document middleware mongodb

Document middleware is supported for the following document functions. In document middleware functions, `this` refers to the document.

- [validate](#)
- [save](#)
- [remove](#)
- [updateOne](#)
- [deleteOne](#)
- init (note: init hooks are [synchronous](#))

Query middleware mongodb

Query middleware is supported for the following Model and Query functions. In query middleware functions, **this** refers to the query.

- `count`
- `countDocuments`
- `deleteMany`
- `deleteOne`
- `estimatedDocumentCount`
- `find`
- `findOne`
- `findOneAndDelete`
- `findOneAndRemove`
- `findOneAndReplace`
- `findOneAndUpdate`
- `remove`
- `replaceOne`
- `update`
- `updateOne`
- `updateMany`

aggregate middleware mongodb

- Aggregate middleware is for MyModel.aggregate(). Aggregate middleware executes when you call exec() on an aggregate object. In aggregate middleware, this refers to the aggregation object.

model middleware mongodb

- Model middleware is supported for the following model functions. In model middleware functions, `this` refers to the model.

`insertMany`

Middleware mongodb

All middleware types support **pre** and **post** hooks. How pre and post hooks work is described in more detail below.

Note: If you specify `schema.pre('remove')`, Mongoose will register this middleware for `doc.remove()` by default. If you want to your middleware to run on `Query.remove()` use `schema.pre('remove', { query: true, document: false }, fn)`.

Note: Unlike `schema.pre('remove')`, Mongoose registers `updateOne` and `deleteOne` middleware on `Query#updateOne()` and `Query#deleteOne()` by default. This means that both `doc.updateOne()` and `Model.updateOne()` trigger `updateOne` hooks, but `this` refers to a query, not a document. To register `updateOne` or `deleteOne` middleware as document middleware, use `schema.pre('updateOne', { document: true, query: false })`.

Note: The `create()` function fires `save()` hooks.

Examples

Add middleware to MongoDB Schema

```
var mongoose = require("mongoose");
var Schema = mongoose.Schema;
var ProductSchema = new Schema({
  title: {
    type:String,
    required:[true,'A product must have title'],
    unique:true,
    trim:true
  },
  description:{
    type:String,
    minlength:[5,'Description is minimum 20 characters'],
    maxlength:[1000,'Description is maximum 1000 characters']
  },
  price:{
    type:Number,
    required:[true, 'A product must have price'],
    min:[0,'price must be above 0'],
    max:[10000,'price must be below 10000']
  },
  created: Date
});
module.exports = mongoose.model('product', ProductSchema);
```

Document middleware - pre save

This middleware run before save commands or create commands

```
ProductSchema.post('save', function(data, next) {  
    console.log(data)  
    next();  
})
```

Document middleware - pre save

This middleware run after save commands or create commands

```
ProductSchema.post('save', function(data, next) {  
    console.log(data)  
    next();  
})
```

Query middleware - pre save

This middleware run before find command

```
//query middleware pre - all the query start with find  
ProductSchema.pre('find', function(next) {  
    this.find({price : {$gt:0}})  
    next();  
});
```

The this here relate to the query

Query middleware - pre save

If you want all queries that start with find, use regular expression

This middleware run before save commands or create commands

```
//query middleware pre - all the query start with find
```

```
ProductSchema.pre(/^find/, function(next) {
```

```
    this.find({price : {$gt:0} } )
```

```
    next();
```

```
} );
```

Regular expression

Query middleware - post find

This middleware run before find command

```
//query middleware pre - all the query start with find  
ProductSchema.post('find', function(next) {  
  this.find({price : {$gt:0}})  
  next();  
});
```

The this here relate to the query

Query middleware - post save

If you want all queries that start with find, use regular expression

This middleware run after save commands or create commands

```
//query middleware post - all the query start with find
ProductSchema.post(/^find/, function(next) {
    this.find({price : {$gt:0} } )
    next();
}) ;
```

Regular expression

aggregate middleware - post save

This middleware run before 'aggregate' command

```
//query middleware post - all the query start with find  
ProductSchema.pre('aggregate', function(next) {  
    console.log(this.pipeline())  
    this.pipeline().unshift({$match:{price:{$gt:0}}}) // add to array  
    next();  
});
```



braintop
think.make.play

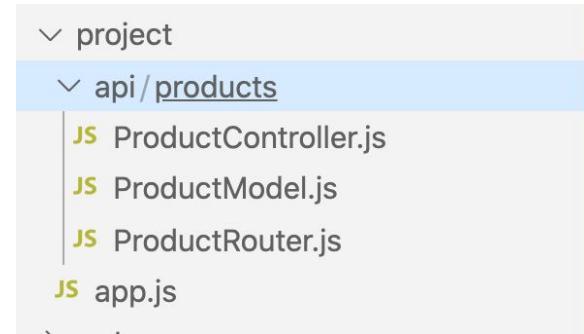
Chapter 14 - Refactoring your mongoose project

Refactoring

Code refactoring is defined as the process of restructuring computer code without changing or adding to its external behavior and functionality. There are many ways to go about refactoring, but it most often comprises applying a series of standardized basic micro-refactorings, each of which is (usually) a tiny change in a computer program's source code that either preserves the behaviour of the software, or at least does not modify its conformance to functional requirements.

Create folders

1. Create folder project
2. In project folder create api folder
 - a. In api folder
 - i. Create file `productController.js`
 - ii. Create file `productRoute.js`
 - iii. Create file `productModel.js`
 - b. Add `app.js` to project



productRouter.js

project/api/product/productRouter.js

```
const express = require('express')
var productRouter = express.Router();
var productController = require('./ProductController')
productRouter.get('/', productController.getProducts);
productRouter.post('/',productController.createProduct);
productRouter.get('/:id',productController.getProductById);
productRouter.patch('/:id',productController.updateProductById);
productRouter.delete('/:id',productController.deleteProductById);
productRouter.get('/get/statistic',productController.getStatistic);

module.exports = productRouter;
```

productController.js

project/api/product/productController.js

```
1 var CurrentProduct = require('./ProductModel')
2 > exports.createProduct = async function(req, res, next) { ...
17 };
18 //read by id
19 > exports.getProductById =  function(req, res, next){ ...
32 }
33 //update
34 > exports.updateProductById= function(req, res, next){ ...
48 }
49 //delete
50 > exports.deleteProductById = function(req, res, next){ ...
64 }
65 //get
66 > exports.getProducts = async function(req, res, next){ ...
110 }
111 > exports.getStatistic = function(req, res, next){ ...
140 }
```

ProductModel.js

project/api/product/productModel.js

```
var mongoose = require("mongoose");
var Schema = mongoose.Schema;
var ProductSchema = new Schema({
  title: {
    type:String,
    required:[true,'A product must have title'],
    unique:true,
    trim:true
  },
  description:{
    type:String,
    minlength:[5,'Description is minimum 20 characters'],
    maxlength:[1000,'Description is maximum 1000 characters']
  },
  price:{
    type:Number,
    required:[true, 'A product must have price'],
    min:[0,'price must be above 0'],
    max:[10000,'price must be below 10000']
  },
  created: Date
});
module.exports = mongoose.model('product', ProductSchema);
```

app.js

project/app.js

```
var express = require('express');
var app = express();
app.use(express.json())
var mongoose = require('mongoose');
var productRouter = require("./api/products/ProductRouter")
app.use('/api/v1/products', productRouter);
const strConnect = "mongodb+srv://asaf:asaf@cluster0.hqfhv.mongodb.net/myFirstDatabase?retryWrites=true&w=majority";
const OPT = {
  useNewUrlParser: true
};
mongoose.connect(strConnect, OPT);
var port = process.env.PORT || 3000;
app.listen(port, function() {
  console.log("Running on port " + port);
})
```

Use `express.json` for body content

use `productRouter` when user request `/api/v1/products`

Refactor exercise 1-8 - to personRouter.js ,personmodel and personController.js

Request	Response	Method	Comment
/api/v1/products	All products	get	Get all products
/api/v1/product	The created product	Post	Create new product
/api/v1/products/:id	product by id	Get	Get product by id
/api/v1/products/:id	Update product	Patch	Update product by id
/api/v1/products/:id	delete products	Delete	Delete product by id



braintop
think.make.play

Chapter 15 - API Security

API Security

You can work on the last project **or** create new folder(project)

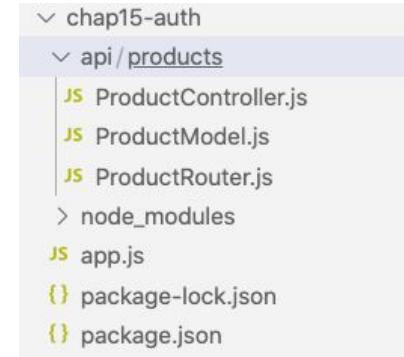
I selected to create new project :so i created folder named chap15-auth.

Create new folder named (select a name) and Run `npm init` in the terminal, and then `npm install mongoose` and `npm install express`.

Copy file named `app.js` and copy `api` folder from the previous project and paste it

Run the server and check that everything work.

After node app: Running on port 3000

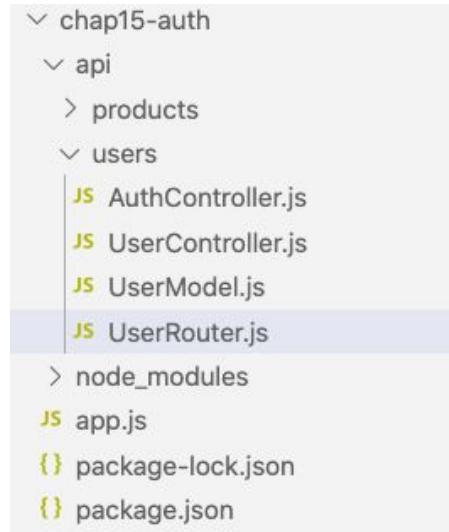


Create User

In api folder create folder named users

In users create 4 files

- UserModel.js
- UserController.js
- UserRouter.js
- AuthController.js



UserModel.js

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var UserSchema = new Schema({
  email: {
    type: String,
    unique: [true, 'please provide email'],
    required: true,
    lowercase:true
  },
  password: {
    type: String,
    required: [true, 'please provide password'],
    minlength:6,
  },
  firstname: String,
  lastname: String,
  photo:String,
  created: Date,
  modified: Date,
  permission: {}
});
module.exports = mongoose.model('user', UserSchema);
```

This is the schema
for a user

We will later see
how to use the
permission array

UserController.js

```
var CurrentUser = require('./UserModel')
exports.createUser = async function(req, res, next) {
  try{
    let p1= req.body;
    var newItem = await CurrentUser.create(p1);
    res.status(201).json({
      status:"success",
      data:newItem
    })
  }
  catch(err){
    res.status(400).json({
      status:"fail",
      message:"error: "+err
    })
  }
};

};
```

Read - UserController.js

```
//read by id
exports.getUserById =  function(req, res, next){
  let id = req.params.id
  CurrentUser.find({_id:id}).then(function(data) {
    res.status(200).json({
      status:"success",
      data:data
    })
  }).catch(err=>{
    res.status(404).json({
      status:"fail",
      message:"error:😱" + err
    })
  })
}
```

Update UserController.js

```
//update
exports.updateUserById= function(req, res, next){
    let id = req.params.id
    CurrentUser.findByIdAndUpdate(id, req.body, {new:true,runValidators:true})
    .then(function(data) {
        res.status(200).json({
            status:"success",
            data:data
        })
    }).catch(err=>{
        res.status(404).json({
            status:"fail",
            message:"error: "+err
        })
    })
}
```

Delete - UserController.js

```
exports.deleteUserById = function(req, res, next) {
  let id = req.params.id
  CurrentUser.findByIdAndDelete(id)
    .then(function(data) {
      res.status(404).json({
        status: "success",
        data: null
      })
    })
    .catch(err=>{
      res.status(404).json({
        status: "fail",
        message: "error: " + err
      })
    })
}
```

Get - UserController.js

```
//get
exports.getUsers = async function(req, res, next) {
    //phase 1 - filtering
    let queryObj = {...req.query}
    let withOutFileds = ['page', 'sort', 'limit', 'fields']
    withOutFileds.forEach(el => {
        delete queryObj[el]
    });
    //phase 2 - advance filtering
    let strQuery = JSON.stringify(queryObj)
    strQuery = strQuery.replace(/\b(gt|gte|lt|lte)\b/g, match =>`${match}`)
    queryObj = JSON.parse(strQuery)
    console.log(queryObj)
    let sort="";
    let selected = "";
    if(req.query.sort){
        sort = req.query.sort.split(',').join(' ')// add more sorts
    }
}
```

Get - UserController.js

```
if(req.query.fields){
    selected = req.query.fields.split(',') .join(' ') // show fields
}
let limit = req.query.limit || 100
let page = req.query.page || 1
let skip = (page-1)*limit
let documents = await CurrentUser.countDocuments()
if(skip>=documents){
    res.status(404).json({
        status:"fail",
        data:"no data on this page and limit"
    })
    return
}
CurrentUser.find(queryObj).skip(skip).limit(limit).select(selected).sort(sort).then(function(data)
{
    res.status(200).json({
        status:"success",
        data:data
    })
}).catch(err=>{
    res.status(404).json({
        status:"fail",
        message:"error: "+err
    })
})
}
```

Signup - AuthController.js

```
let User = require('./UserModel')
exports.signup = function(req, res, next) {
  var newUser = new User(req.body);
  newUser.created = new Date();
  newUser.modified = new Date();
  newUser.save().then(function(user) {
    res.status(201).json(
      {
        status:"success",
        user:user
      });
  }).catch(err=>{
    res.status(404).json({
      status:"fail",
      message:"error: "+err
    })
  })
}
```

UserRouter.js

```
const express = require('express')

var userRouter = express.Router();

var userController = require('./UserController')
var authController = require('./AuthController')

userRouter.post('/signup', authController.signup); ← signup
userRouter.post('/',userController.createUser);

userRouter.get('/:id',userController.getUserById);

userRouter.patch('/:id',userController.updateUserById);

userRouter.delete('/:id',userController.deleteUserById);

module.exports = userRouter;
```

app.js

```
var express = require('express');
var app = express();
app.use(express.json())
var mongoose = require('mongoose');
var productRouter = require("./api/products/ProductRouter")
var userRouter = require("./api/users/UserRouter") ← Add user router
app.use('/api/v1/products', productRouter);
app.use('/api/v1/users', userRouter); ←
const strConnect =
"mongodb+srv://asaf:asaf@cluster0.hqfhv.mongodb.net/myFirstDatabase?retryWrites=true
&w=majority";
const OPT = {
  useNewUrlParser: true
};
mongoose.connect(strConnect, OPT);
var port = process.env.PORT || 3000;
app.listen(port, function() {
  console.log("Running on port " + port);
})
```

Add user router

Run the app and try to signup a user

localhost:3000/api/v1/users/signup

POST localhost:3000/api/v1/users/signup

Send

Params Authorization Headers (8) Body **Body** Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 "email": "a1@gmail.com",
2 "password": "abcdefwqwqwqwq",
3 "firstname": "david",
4 "lastname": "lewis"
```

Body Cookies Headers (7) Test Results

201 Created 194 ms 477 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "status": "success",
3   "user": {
4     "email": "a1@gmail.com",
5     "password": "abcdefwqwqwqwq",
6     "firstname": "david",
7     "lastname": "lewis",
8     "_id": "61dea4c8b817c6787e0efa6f",
9     "created": "2022-01-12T09:52:08.115Z",
10    "modified": "2022-01-12T09:52:08.115Z",
11    "__v": 0
12  }
13}
```

Bootcamp Runner Trash

localhost:3000/api/v1/users/signup

UserModel.js

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var UserSchema = new Schema({
  email: {
    type: String,
    unique: [true, 'please provide email'],
    required: true,
    lowercase:true
  },
  password: {
    type: String,
    required: [true, 'please provide password'],
    minlength:6,
  },
  firstname: String,
  lastname: String,
  photo:String,
  created: Date,
  modified: Date,
  permission: {}
});
module.exports = mongoose.model('user', UserSchema);
```

This is the schema
for a user

We will later see
how to use the
permission array

UserModel.js - add hash to password

```
UserSchema.pre('save', function(next) {  
  if (!this.isModified('password')) {  
    return next();  
  }  
  
  var salt = bcrypt.genSaltSync(12);  
  this.password = bcrypt.hashSync(this.password, salt);  
  next();  
});  
  
<!-- Continues in the next slide -->  
Don't forget to add var bcrypt = require('bcryptjs')
```

For the implementation of encryptPassword() add:

```
var bcrypt = require('bcryptjs');
```

And run:

```
npm install bcryptjs
```

Before calling the
save() method, this
function will be called

Check that the password
hasn't change

Encrypt the password
(we will see the
implementation in the
next slide)

UserModel.js

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var bcrypt = require('bcryptjs')
var UserSchema = new Schema({
  email: {
    type: String,
    unique: [true, 'please provide email'],
    required: true,
    lowercase:true
  },
  password: {
    type: String,
    required: [true, 'please provide password'],
    minlength:6,
  },
  firstname: String,
  lastname: String,
  photo:String,
  created: Date,
  modified: Date,
  permission: {}
});
module.exports = mongoose.model('user', UserSchema);
```

```
UserSchema.pre('save', function(next) {
  if (!this.isModified('password')) {
    return next();
  }
  var salt = bcrypt.genSaltSync(12);
  this.password = bcrypt.hashSync(this.password,
salt);
  next();
});
```

Run the app and try to signup a user

The screenshot shows a Postman interface for a POST request to `localhost:3000/api/v1/users/signup`. The request body is a JSON object:

```
1 {  
2   "email": "a2@braingtop.io",  
3   "password": "123456",  
4   "firstname": "Daniela",  
5   "lastname": "Lewis"  
6 }  
7 }
```

The response status is 201 Created, with a response time of 231 ms and a size of 528 B. The response body is:

```
1 {  
2   "status": "success",  
3   "user": {  
4     "email": "a2@braingtop.io",  
5     "password": "$2a$10$jdzQYYArA19xpu2jr07S8.3g.9M5JkHIV3r9GM0wFIJEufbPaTiEK",  
6     "firstname": "Daniela",  
7     "lastname": "Lewis",  
8     "_id": "61deaf590d68a472d982e93a",  
9     "created": "2022-01-12T10:37:13.328Z",  
10    "modified": "2022-01-12T10:37:13.328Z",  
11    "__v": 0  
12  }  
13 }
```

A callout box highlights the encrypted password in the response body with the text "Password encrypted".

Do it yourself

According to slides 323 till 339 Add to project from the previous chapter in slide 321:

In api folder create folder named users and add to it 4 files:

- UserModel.js
- UserController.js
- UserRouter.js
- AuthController.js

In addition add the options to make signup : api/v1/users/signup

Jwt - json web token <https://jwt.io/>

JSON Web Token (JWT) is an open standard ([RFC 7519](#)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA** or **ECDSA**.

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on *signed* tokens. Signed tokens can verify the *integrity* of the claims contained within it, while encrypted tokens *hide* those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

Jwt - json web token <https://jwt.io/>

When should you use JSON Web Tokens?

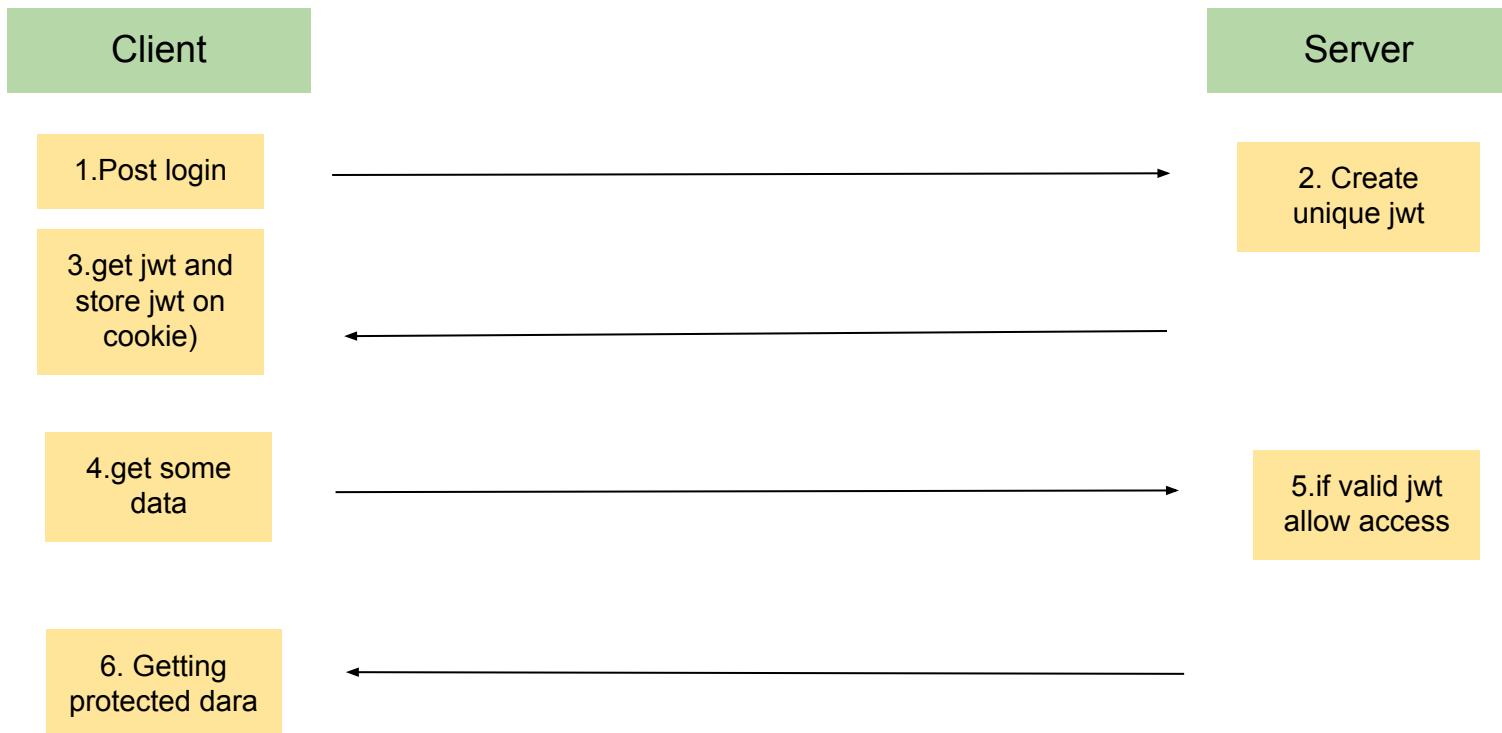
Here are some scenarios where JSON Web Tokens are useful:

- **Authorization:** This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.
- **Information Exchange:** JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

Encryption

Encryption is a means **of securing digital data using one or more mathematical techniques**, along with a password or "key" used to decrypt the information. The encryption process translates information using an algorithm that makes the original information unreadable.

Jwt - json web token - <https://jwt.io/>



Add config.env

```
var configValues = {  
  "uname": "asaf",  
  "pwd": "asaf"  
};  
  
var config = {  
  dev: 'development',  
  test: 'testing',  
  prod: 'production',  
  port: process.env.PORT || 3000,  
  //ten days in minutes  
  expireTime: 60 * 60 * 1000,  
  getDbConnectionString: function() {  
    return '';  
  },  
  secrets: {  
    jwt: process.env.JWT || "mysecret"  
  },  
};  
  
module.exports = config;
```

Static values for checks

Our application's running modes

Default port

Expiration time of a token

So we won't need to search the dbConnectionString we put in the app.js every time we want to connect to the db

A secret for the tokens - my secret is very bad secret 😊

Signup - AuthController.js

```
let User = require('./UserModel')
let jwt = require('jsonwebtoken') ← Add require jsonwebtoken
let config = require('../config') ← Add require config
exports.signup = function(req, res, next) {
  var newUser = new User(req.body);
  newUser.created = new Date();
  newUser.modified = new Date();
  newUser.save().then(function(user) {
    let token = jwt.sign({id:user._id}, config.secrets.jwt, {expiresIn:config.expireTime})
    res.status(201).json(← Returns a new token for the given id.
      {
        status:"success",
        token,
        user:user
      });
  }).catch(err=>{
    res.status(404).json({
      status:"fail",
      message:"error:❗️" + err
    })
  })
}
```

Returns a new token for the given id.
On sign-in the user will receive a new token with expiration time

Response token

UserModel authenticate Login

Add this `UserSchema.methods` to `UserModel`

```
UserSchema.methods = {  
  
  authenticate: function(plainTextPword) {  
  
    console.log("this password is " + this.password);  
  
    return bcrypt.compareSync(plainTextPword, this.password);  
  
  },  
  
  toJson: function() {  
  
    var obj = this.toObject();  
  
    delete obj.password;  
  
    return obj;  
  
  }  
  
};
```

Delete user password property
before response

Authcontroller Login

Add this `login` function to `AuthController.js`

```
exports.login = function(req, res, next){  
  let {email, password} = req.body;  
  if(!email || !password){  
    res.status(400).send('you need email and password');  
    return;  
  }  
  User.findOne({ email: email })  
  .then(function(user) {  
    if (!user) {  
      res.status(401).send('No user with the given username');  
      return  
    }  
  })  
  .catch(function(error) {  
    res.status(500).send('Something went wrong');  
  })  
};
```

Stop if there is not user with the given email or password

Find the user with the given email in the db

Authcontroller Login

```
else {
    if (!user || !user.authenticate(password)) {
        res.status(401).send('Wrong password');
        return
    }
    else{
        let token = signToken(user._id)
        res.status(200).json({
            status:"success",
            token:token
        });
    }
}
```

Check the password

Retrieve token

Response token to client

Authcontroller Login

Add this `signToken` function to `AuthController.js`

```
let signToken = id =>  {
  return jwt.sign({
    id: id
  },
  config.secrets.jwt, {
    expiresIn: config.expireTime
  }
}
```

Phase 1 Find the user

Run the app and try to login a user

The screenshot shows a Postman interface with two requests. The left request is a POST to `localhost:3000/api/v1/users/login` with JSON body:

```
1
2   "email": "c1@braingtop.io",
3   "password": "123456"
```

The right request is also to `localhost:3000/api/v1/users/login`. The response body is:

```
1
2   "status": "success",
3   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
4       eyJpZCI6IjYxZGVjYTYxNTMzM2E0OTVjYjNmZCIsImlhCI6MTY0MTk5MTA5MiwiZXhwIjoxNjQ1NTkxMDkyfQ.
5       fIbs7C4yaHudU95saInN1r4Wa4UDKort_DwCMQ7RrF4"
```

Do it yourself

According to slides 340 till 351 Add to project

- config.js
- Login + token - api/v1/users/login

In addition add the options to make signup : api/v1/users/signup

Protecting data -

Add the function `exports.protectSystem` to `AuthController.js`

```
exports.protectSystem = async function(req, res, next) {  
    //phase 1 - get token  
    //phase 2 - verification token  
    //phase 3 - check if user exist  
    //phase 4 - check if user does not change passwords  
}
```

Add protectSystem middleware to productRouter.js

```
const express = require('express')
var productRouter = express.Router();
var productController = require('./ProductController')

var authController = require('../users/AuthController')

productRouter.get('/',authController.protectSystem, productController.getProducts);

productRouter.post('/' ,productController.createProduct);
productRouter.get('/:id',productController.getProductById);
productRouter.patch('/:id',productController.updateProductById);
productRouter.delete('/:id',productController.deleteProductById);
productRouter.get('/get/statistic',productController.getStatistic);
module.exports = productRouter;
```

Protecting data - Phase 1 Get Token

```
exports.protectSystem = async function(req, res, next) {
    //phase 1 - get token
    let token = ""
    arrAuthorization = req.headers.authorization.split(' ')
    if(arrAuthorization[0]==='Bearer' && arrAuthorization[1])
        token = arrAuthorization[1]
    console.log(token)
    if(!token) {
        res.status(401).json({fail:"You are not login again"})
        return
    }
}
```

localhost:3000/api/v1/products

The screenshot shows the Postman interface with a GET request to `localhost:3000/api/v1/products`. The Authorization tab is active, displaying a Bearer token. The response status is 200 OK.

```
1 "status": "success",
2 "data": [
3     {
4         "_id": "61dc55b4c0dc1f09e5d8",
5         "title": "Red ball",
6         "description": "This is a red ball",
7         "price": 300,
8         "created": "2023-10-09T21:00:00.000Z",
9         "__v": 0
10     },
11     {
12         ...
13     }
14 ]
```

token

Running on port 3000

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYxZTNkOWZlZGY3ZmU5ODhiMDJlNDRmNyIsImlhcdCI6MTY0MjMyNDczMCwiZXhwIjoxNjQ1OTI0NzMwfQ.3ZLz5ugvVTu55DBd5vbpcKVH0BS8kQJqwR076MTEgWU

Add promisify to AuthController.js - link

The `util.promisify()` method basically takes a function as an input that follows the common Node.js callback style, i.e., with a `(err, value)` and returns a version of the same that returns a promise instead of a callback.

```
// Importing the fs and util modules
const fs = require('fs');
const util = require('util');

// Reading the file using a promise & printing its text
let file = util.promisify(fs.readFile);
file('./promisify.js', 'utf8') // Reading the same file
  .then((txt) => {
    console.log(txt);
  })

// Printing error if any
  .catch((err) => {
    console.log('Error', err);
  });
});
```

In the example, we have used the **fs** module to read the files. We have used **util.promisify()** method to convert the **fs.readFile** to a promise-based method. Now, the above method gives us a promise instead of a callback.

Add promisify to AuthController.js - link

The util.promisify() method basically takes a function as an input that follows the common Node.js callback style, i.e., with a (err, value) and returns a version of the same that returns a promise instead of a callback.

```
let User = require('./UserModel')
const {promisify} = require('util')
let jwt = require('jsonwebtoken')
let config = require('../config')
```

Protecting data -

Add the function `exports.protectSystem` to `AuthController.js`

```
exports.protectSystem = async function(req, res, next) {
  .
  .
  //phase 2 - verification token
  let decoded = ""
  try{
    decoded = await promisify(jwt.verify)(token, config.secrets.jwt)
    console.log(decoded)
  }
  catch(err){
    console.log(err)
    res.status(401).json({fail:"verification token failed please login
again:" + err})
    return
  }
}
```

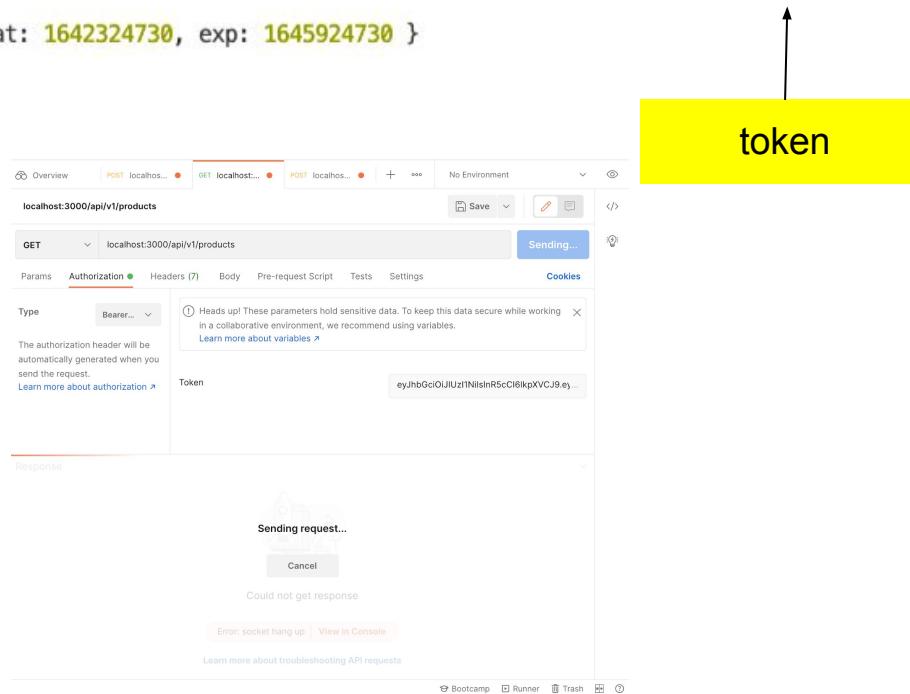
localhost:3000/api/v1/products

Running on port 3000

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYxZTNkOWZiZGY3ZmU5ODhiMDJlNDRmNyIsImhlhdCI6MTY0MjMyNDczMCwiZXhwIjoxNjQ1OTI0NzMwfQ.3ZLz5ugvVTu55
DBd5vbpcKVH0BS8kQJqwR076MTEgWU

{ id: '61e3d9fbdf7fe988b02e44f7', iat: 1642324730, exp: 1645924730 }

↑
decoded user id



Protecting data - phase 3 - check if user exist

```
exports.protectSystem = async function(req, res, next) {
    .
    .
    .
    // phase 3 check if user exist
    const currentUser = await User.findById(decoded.id)
    console.log(currentUser)
    if(!currentUser) {
        res.status(401).json({fail:"User not login please login again"})
        return
    }
}
```

The diagram illustrates the flow of the `User_id` variable. It starts with a yellow box labeled `User_id` pointing to a dotted line. This line leads to another yellow box labeled `User_id`, which then points to a JSON object: `{ id: '61e3d9fbdf7fe988b02e44f7', iat: 1642324730, exp: 1645924730 }`. There are also two arrows pointing upwards from the JSON object towards the second `User_id` box.

Protecting data - phase 4 - Add **passwordChangedAt** to userModel.js

```
var UserSchema = new Schema({
  email: {
    type: String,
    unique: [true, 'please provide email'],
    required: true,
    lowercase:true
  },
  password: {
    type: String,
    required: [true, 'please provide password'],
    minlength:6,
  },
  firstname: String,
  lastname: String,
  photo:String,
  created: Date,
  modified: Date,
  passwordChangedAt:Date,
  permission: {}
});
```

Protecting data - phase 4 - Add to userModel.js

```
UserSchema.methods = {
  authenticate: function(plainTextPword) {
    console.log("this password is " + this.password);
    return bcrypt.compareSync(plainTextPword, this.password);
  },
  changePasswordAfter:function(JWTTimeStamp) {
    if(this.changePasswordAt){
      let changedTimeStamp = parseInt(this.passwordChangedAt.getTime()/1000)
      return JWTTimeStamp < changedTimeStamp
    }
    return false; //password didn't change
  },
  toJson: function() {
    var obj = this.toObject();
    delete obj.password;
    return obj;
  }
};
```

Protecting data - protectSystem Phase 4

```
exports.protectSystem = async function(req, res, next){  
  .  
  .  
  .  
  
  //phase 4 - check if user changed passwords  
  if(currentUser.changePasswordAfter(decoded.iat))  
  {  
    res.status(401).json({fail:"User changed passwords, please login again"})  
    return  
  }  
  req.user = currentUser;  
  next();  
}
```

Attach user to req and call
next()

Permissions - Add permissions to UserModel.js

```
var UserSchema = new Schema({
  email: {
    type: String,
    unique: [true, 'please provide email'],
    required: true,
    lowercase:true
  },
  password: {
    type: String,
    required: [true, 'please provide password'],
    minlength:6,
  },
  firstname: String,
  lastname: String,
  photo:String,
  created: Date,
  modified: Date,
  passwordChangedAt:Date,
  permission: {
    type:String,
    enum:['user', 'author', 'admin'],
    default:'user'
  }
});
```

Permissions - Add isAdmin middleware

```
exports.isAdmin = function(req, res, next){  
  if(req.user && req.user.permission && req.user.permission=='admin')  
    next()  
  else{  
    res.status(401).json({  
      message:'you don't have permission'  
    })  
  }  
}
```

Permissions - Add isAdmin middleware

```
const express = require('express')
var productRouter = express.Router();
var productController = require('./ProductController')
var authController = require('../users/AuthController')
productRouter.get('/',authController.protectSystem,authController.isAdmin, productController.getProducts);
productRouter.post('/',productController.createProduct);
productRouter.get('/:id',productController.getProductById);
productRouter.patch('/:_id',productController.updateProductById);
productRouter.delete('/:_id',productController.deleteProductById);
productRouter.get('/get/statistic',productController.getStatistic);
module.exports = productRouter;
```

Now only admin users can
getProducts

Sign up user

The screenshot shows a Postman interface with the following details:

- Method:** POST
- URL:** localhost:3000/api/v1/users/signup
- Body:** JSON (selected)
- Request Body:**

```
1  {
2    "email": "c1@gmail.com",
3   "password": "123456",
4   "firstname": "as",
5   "lastname": "last",
6   "passwordChangedAt": "10/10/2022"
7 }
8 }
```

- Response Status:** 201 Created
- Response Time:** 276 ms
- Response Size:** 768 B
- Response Content:**

```
1  {
2     "status": "success",
3     "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
4         eyJpZC16IjYxZTNmYjBkZmQyM2NiMjQzOTVhNWE1YiIsImhdCI6MTY0MjMzMdg5MywiZXhwIjoxNjQ1OTMwODkzfQ.
5         A9GazmrOTIk-qKLvrHL5oL3udNM7XILWXQBbVyrCCzw",
6     "user": {
7         "email": "c1@gmail.com",
8         "password": "$2a$10$KENNgH9MWLbtNPm/LiYCmu51BZ5mv2dKE58danZN1shDhCmH2tWow",
9         "firstname": "as",
10        "lastname": "last",
11        "passwordChangedAt": "2022-10-09T21:00:00.000Z",
12        "permission": "user",
13        "_id": "61e3fb0dfdf23cb24395a5a5b",
14        "created": "2022-01-16T11:01:33.471Z",
15        "modified": "2022-01-16T11:01:33.471Z",
16        "__v": 0
17     }
18 }
```

Login the user

The screenshot shows the Postman interface for making a POST request to `localhost:3000/api/v1/users/login`. The **Body** tab is selected, containing the following JSON payload:

```
1
2   ...
3     "email": "c1@gmail.com",
4     "password": "123456"
```

The response status is 200 OK, with a response body containing:

```
1
2   {
3     "status": "success",
4     "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJpZCI6IjYxZTNmYjBkZmQyM2NjMjQzOTVnWE1iisImIhdCI6MTY0MjMzMTA1MywiZXhwIjoxNjQ1OTMxMDUzfQ.  
ajdEM4BxdmA75pejk0yFEYYi7NNjmRD6Q0Qfnerj0-Q"
```

A handwritten annotation with an arrow points from the text "copy the token" to the `token` field in the response body.

copy the token

Try to get products

The screenshot shows a Postman request for `localhost:3000/api/v1/products/` using the `GET` method. The `Authorization` tab is selected, showing a `Bearer Token` input field. A warning message states: `Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables.`. The token value is highlighted with a yellow box and an annotation pointing to it with the text `Paste the token`.

The response status is `401 Unauthorized`, with a message: `"message": "you dont have permission"`.

Change to admin

```
_id: ObjectId("61e3fb0df23cb24395a5a5b")
email: "c1@gmail.com"
password: "$2a$10$KENNgH9MWLbtNPm/LiYCmu5iBZ5mv2dKE58danZNlshDhCmH2tWoW"
firstname: "as"
lastname: "last"
passwordChangedAt: 2022-10-09T21:00:00.000+00:00
permission: "admin"
created: 2022-01-16T11:01:33.471+00:00
modified: 2022-01-16T11:01:33.471+00:00
__v: 0
```

Change to admin

Change to admin

```
_id: ObjectId("61e3fb0df23cb24395a5a5b")
email: "c1@gmail.com"
password: "$2a$10$KENNgH9MWLbtNPm/LiYCmu5iBZ5mv2dKE58danZNlshDhCmH2tWoW"
firstname: "as"
lastname: "last"
passwordChangedAt: 2022-10-09T21:00:00.000+00:00
permission: "admin"
created: 2022-01-16T11:01:33.471+00:00
modified: 2022-01-16T11:01:33.471+00:00
__v: 0
```

Change to admin

Now you have permission

The screenshot shows the Postman interface with a successful API call. The URL is `localhost:3000/api/v1/products/`. The **Authorization** tab is selected, showing a **Bearer Token** input field containing a long string of characters. A tooltip message states: "Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. [Learn more about variables](#)". The response body is displayed in **Pretty** format:

```
1  "status": "success",
2  "data": [
3    {
4      "_id": "61dc55cec0dc1f89e5d8a7e6",
5      "title": "Blue ball",
6      "description": "This is a blue ball",
7      "price": 500,
8      "created": "2023-10-09T21:00:00.000Z",
9      "__v": 0
10    }
11  ]
12]
13]
```

The status bar at the bottom indicates: Status: 200 OK Time: 605 ms Size: 413 B. A yellow callout box labeled "Response products" points to the "data" field in the JSON response.

Response products

Do it yourself

According to slides 352 till 374 Add to project protect

- Only admin users can delete data

Sign up to <https://mailtrap.io/>

After sign up you can see
your
Host
Port
Username
Password

The screenshot shows the 'My Inbox' settings page on Mailtrap. At the top, there are tabs for 'SMTP Settings', 'Email Address', 'Auto Forward', 'Manual Forward', and 'Team Members'. The 'Total messages sent: 0' is displayed on the right.

SMTP / POP3 (Reset Credentials)

Use these settings to send messages directly from your email client or mail transfer agent.

ⓘ Don't disclose your username or password as this may result in your inbox getting filled up with spam.

[Hide Credentials ^](#)

SMTP

Host:	smtp.mailtrap.io
Port:	25 or 465 or 587 or 2525
Username:	0c67e8550f88ec
Password:	0e3b2c63bc66a0
Auth:	PLAIN, LOGIN and CRAM-MD5
TLS:	Optional (STARTTLS on all ports)

POP3

Host:	pop3.mailtrap.io
Port:	1100 or 9950
Username:	0c67e8550f88ec
Password:	0e3b2c63bc66a0
Auth:	USER/PASS, PLAIN, LOGIN, APOP and CRAM-MD5
TLS:	Optional (STARTTLS on all ports)

Integrations (curl)

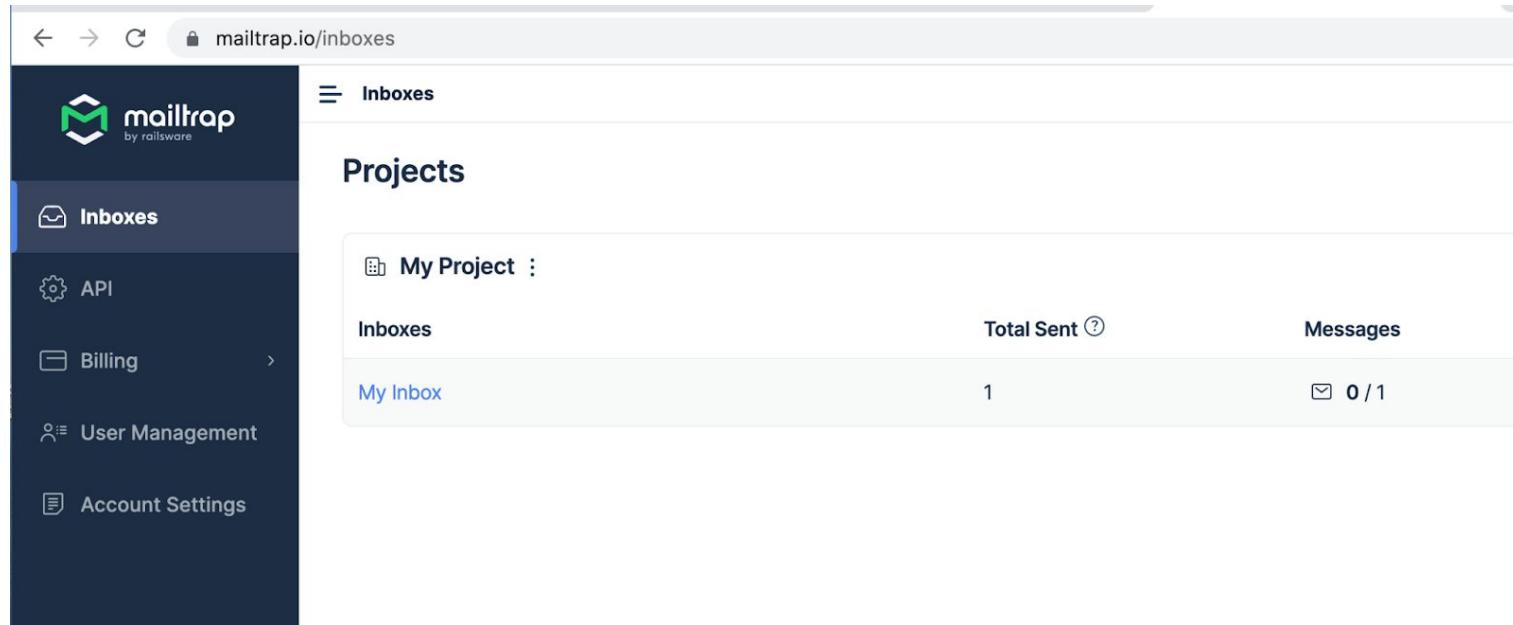
```
curl --ssl-reqd \
--url 'smtp://smtp.mailtrap.io:2525' \
--user '0c67e8550f88ec:0e3b2c63bc66a0' \
--mail-from from@example.com \
--mail-to to@example.com \
--upload-file - <<EOF
From: Magic Elves <from@example.com>
To: Mailtrap Inbox <to@example.com>
Subject: You are awesome!
Content-Type: multipart/alternative; boundary="boundary-string"

--boundary-string
Content-Type: text/plain; charset="utf-8"
Content-Transfer-Encoding: quoted-printable

```

[Copy](#)

Go to your inbox



The screenshot shows the Mailtrap dashboard at mailtrap.io/inboxes. The left sidebar has a dark blue background with white text and icons. It includes links for 'Inboxes' (selected), 'API', 'Billing', 'User Management', and 'Account Settings'. The main area has a light gray background. At the top, there's a header with a back arrow, forward arrow, refresh icon, and a lock icon followed by the URL. Below the header, the word 'Inboxes' is displayed next to a three-line menu icon. The main content area is titled 'Projects' and shows a card for 'My Project'. The card contains the following information:

Inboxes	Total Sent	Messages
My Inbox	1	✉ 0 / 1

Forgot password

```
var config = {
  configValues : {
    "uname": "asaf",
    "pwd": "asaf"
  },
  EMAIL_USERNAME : "0c67e8550f88ec",
  EMAIL_PASSWORD : "0e3b2c63bc66a0",
  EMAIL_HOST : "smtp.mailtrap.io",
  EMAIL_PORT : 25,
  dev: 'development',
  test: 'testing',
  prod: 'production',
  port: process.env.PORT || 3000,
  //ten days in minutes
  expireTime: 60 * 60 * 1000,
  getDbConnectionString: function() {
    return '';
  },
  secrets: {
    jwt: process.env.JWT || "mysecret"
  },
};
module.exports = config;
```



Add to config.js

```
EMAIL_USERNAME,EMAIL_PASSWORD,EMAIL_HOST,EMAIL_PORT
```

Email.js module

```
let nodemailer = require('nodemailer')
let config = require('../config')
const sendEmail = async options=>{
    let transporter = nodemailer.createTransport({
        host:config.EMAIL_HOST,
        port:config.EMAIL_PORT,
        auth:{
            user:config.EMAIL_USERNAME,
            pass:config.EMAIL_PASSWORD
        }
    })
    const mailOptions = {
        from: '<test mail>no-reply',
        to:options.email,
        subject:options.subject,
        text:options.message
    }
    await transporter.sendMail(mailOptions)
}
module.exports = sendEmail
```

Add send email to this module and export it.

AuthController.js export.forgotPassword

```
exports.forgotPassword = async function(req, res, next) {  
    //phase 1 - get user by email  
    let user = await User.findOne({email:req.body.email})  
    if(!user) {  
        res.status(404).json({  
            message:'please send email'  
        })  
        return  
    }  
}
```

AuthController.js export.forgotPassword

```
try{
  //phase 2 - create random reset token
  let resetToken = user.createNewPasswordToken()
  await user.save({validateBeforeSave:false})
  //phase 3 - send it to user email
  let resetUrl = req.protocol + "://" + req.get('host') +"/api/v1/users/resetPassword/" +
resetToken
  let message = "click here to make new password " + resetUrl
  await sendEmail({email:user.email, subject:'your password reset token ', message})
  res.status(200).json({status:"success", message:'token sent to your email'})
}
catch(err){
  user.passwordResetToken=undefined
  user.passwordResetExpires = undefined
  await user.save({validateBeforeSave:false})
  res.status(500).json({status:"failed", message:'error sending email'})
}
}
```

Send token to email

localhost:3000/api/v1/users/forgotPassword

The screenshot shows the Postman application interface. At the top, the URL is set to `localhost:3000/api/v1/users/forgotPassword`. The method is selected as `POST`. In the `Body` tab, the `JSON` section contains the following payload:

```
1 {  
2   "email": "c1@gmail.com"  
3 }
```

Below the request, the response is displayed. The status is `200 OK`, and the message body is:

```
1 {  
2   "status": "success",  
3   "message": "token sent to your email"  
4 }
```

Email accept

Inboxes > My Inbox > your password reset token

Search...     

your password reset token
to: <c1@gmail.com> a few seconds ago

[Show Headers](#)

HTML HTML Source **Text** Raw Spam Analysis Tech Info

click here to make new password <http://localhost:3000/api/v1/users/resetPassword/14j2p9rr5yuwjc453e3azmeo9qyvh8br2kzw6>


You are limited to last 50 messages,
1 inbox and 1 user on the free forever plan.

[Click here to upgrade](#)

Not now, thanks

Reset password

```
exports.resetPassword= async function(req, res, next){  
    //phase 1 - get user  
    let user = await User.findOne(  
    {  
        passwordResetToken:req.params.token,  
        passwordResetExpires:{$gt:Date.now()}  
    })  
    //phase 2 - check if token not expired  
    if(!user){  
        res.status(404).json({status:"failed", message:'invalid token'})  
    }  
    user.password = req.body.password  
    //phase 3  
    user.passwordResetToken=undefined  
    user.passwordResetExpires=undefined  
    await user.save()  
    let token = signToken(user._id)  
    res.status(200).json({  
        status:"success",  
        token:token  
    });  
}
```

Sending reset password

The screenshot shows a POST request in Postman to the endpoint `localhost:3000/api/v1/users/resetPassword/renmdtj4ym2iqd1ks3j4ubhn6ios3doqsknysd2`. The request body is a JSON object with a single field `"password": "1234567"`. The response status is 200 OK, indicating success.

```
localhost:3000/api/v1/users/resetPassword/renmdtj4ym2iqd1ks3j4ubhn6ios3doqsknysd2
```

```
PATCH localhost:3000/api/v1/users/resetPassword/renmdtj4ym2iqd1ks3j4ubhn6ios3doqsknysd2
```

```
Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies Beautify
```

```
none form-data x-www-form-urlencoded raw binary GraphQL JSON
```

```
1 "password": "1234567"
```

```
Body Cookies Headers (7) Test Results Status: 200 OK Time: 508 ms Size: 438 B Save Response
```

```
Pretty Raw Preview Visualize JSON
```

```
1 "status": "success",
2 "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
3 eyJpZCI6IjYxZTNmYjBkZmQyM2NiMjQzOTVhNWE1YiIsImlhCI6MTY0MjUwODEwMiwiZXhwIjoxNjQ2MTA4MTAyfQ.
4 eyNhHys3BJ-bQkXophSSxDR1ji6Bsud9gZBnQEtqM"
```

New token

Sending reset password

The screenshot shows a POST request in Postman to the URL `localhost:3000/api/v1/users/resetPassword/renmdtj4ym2iqd1ks3j4ubhn6ios3doqsknysd2`. The request body is a JSON object with a single field `"password": "1234567"`. The response status is 200 OK, and the response body is a JSON object with fields `"status": "success"` and `"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYxZTNmYjBkZmQyM2NiMjQzOTVhNWE1YiIsImlhCI6MTY0MjUwODEwMiwiZXhwIjoxNjQ2MTA4MTAyfQ.eyJHYS3BJ-bQkxophSSxDR1ji6Bsud9gZBnQEtqM"`.

```
localhost:3000/api/v1/users/resetPassword/renmdtj4ym2iqd1ks3j4ubhn6ios3doqsknysd2
```

```
PATCH localhost:3000/api/v1/users/resetPassword/renmdtj4ym2iqd1ks3j4ubhn6ios3doqsknysd2
```

```
Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies Beautify
```

```
none form-data x-www-form-urlencoded raw binary GraphQL JSON
```

```
1 "password": "1234567"
```

```
Body Cookies Headers (7) Test Results Status: 200 OK Time: 508 ms Size: 438 B Save Response
```

```
Pretty Raw Preview Visualize JSON
```

```
1 "status": "success",  
2 "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
3 eyJpZCI6IjYxZTNmYjBkZmQyM2NiMjQzOTVhNWE1YiIsImlhCI6MTY0MjUwODEwMiwiZXhwIjoxNjQ2MTA4MTAyfQ.  
eyNHYS3BJ-bQkxophSSxDR1ji6Bsud9gZBnQEtqM"
```

New token

Update my password

```
exports.updatePassword =async function (req, res, next){  
    //get user  
    let user = await User.findById(req.user._id);  
    //check current password  
    if(!(await user.matchPassword(req.body.currentPassword)))  
    {  
        req.status(401).json({  
            status:"failed",  
            message:"password not correct"  
        })  
        return  
    }  
    //if the password correct update it  
    user.password = req.body.password  
    await user.save()  
    //login user by sending jwt token  
    let token = signToken(user._id)  
    res.status(200).json({  
        status:"success",  
        token:token  
    });  
}
```

Update my password

```
const express = require('express')
var userRouter = express.Router();
var userController = require('./UserController')
var authController = require('./AuthController')

userRouter.post('/signup', authController.signup);
userRouter.post('/login', authController.login);
userRouter.post('/forgotPassword', authController.forgotPassword);
userRouter.patch('/resetPassword/:token', authController.resetPassword);
userRouter.patch('/updateMyPassword',authController.protectSystem ,authController.updatePassword);

userRouter.post('/',userController.createUser);
userRouter.get('/:id',userController.getUserById);
userRouter.patch('/:id',userController.updateUserById);
userRouter.delete('/:id',authController.isAdmin ,userController.deleteUserById);
module.exports = userRouter;
```

Add this function to model methods

```
UserSchema.methods = {
  authenticate: function(plainTextPword) {
    console.log("this password is " + this.password);
    return bcrypt.compareSync(plainTextPword, this.password);
  },
  changePasswordAfter:function(JWTTimeStamp){
    if(this.changePasswordAt){
      let changedTimeStamp = parseInt(this.passwordChangedAt.getTime()/1000)
      return JWTTimeStamp < changedTimeStamp
    }
    return false; //password dont changed
  },
  createNewPasswordToken: function(){
    let str ="abcdefghijklmnopqrstuvwxyz1234567890"
    this.passwordResetToken=""
    for(var i=0;i<40;i++){
      let n = Math.floor(Math.random() * (str.length - 1));
      this.passwordResetToken+=str[n]
    }
    this.passwordResetExpires = Date.now() + 30*60*1000 // half hour
    return this.passwordResetToken;
  },
  matchPassword:function(pass){
    if(bcrypt.compareSync(pass, this.password))
      return true
    return false
  },
  toJson: function() {
    var obj = this.toObject();
    delete obj.password;
    return obj;
  }
}
```

Check if user sent is password, before he change it.

localhost:3000/api/v1/users/updateMyPassword

The screenshot shows a POSTMAN interface with the following details:

- Method:** PATCH
- URL:** localhost:3000/api/v1/users/updateMyPassword
- Body (JSON):**

```
1 {
2   ...
3     "currentPassword": "1234567",
4     "password": "654321"
5 }
```
- Response Status:** 200 OK
- Response Time:** 665 ms
- Response Size:** 438 B
- Response Body (Pretty JSON):**

```
1 {
2   "status": "success",
3   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
4       eyJpZCI6IjYxZTNmYjBkZmQyM2NiMjQzOTVhNWE1YiIsImhdCI6MTY0MjY40TMyNSwiZXhwIjoxNjQ2Mjg5MzI1fQ.
      uuloaWkwBAuJCID7wxD04xF6hYFBw0YsuM3bUrQK5aw"
```

Update user data

```
exports.updateMe = async function(req, res, next) {
  // create error if user post the password
  if(req.body.password)
  {
    res.status(400).json({
      status:"failed",
      message:"can't update password from here"
    });
    return
  }
  //update user
  let filterBody = allowedObj(req.body, 'firstname', 'lastname','email')
  let user = await User.findByIdAndUpdate(req.user._id,filterBody,
  {new:true, runValidators:true})
  res.status(200).json({
    status:"success",
    user:user
  });
}
```

The filled that you allowed to update. For example you don't want that the user will update the token or the permission

Update user data

```
let allowedObj = function(obj, ...allowedFields) {  
  let newobj = {}  
  Object.keys(obj).forEach(el =>{  
    if(allowedFields.includes(el))  
      newobj[el] = obj[el]  
  }) ;  
  return newobj  
}
```

Return allowed object to update.

localhost:3000/api/v1/users/updateMe

The screenshot shows a Postman interface with the following details:

- URL:** localhost:3000/api/v1/users/updateMe
- Method:** PATCH
- Body:** JSON (selected)
- Request Body Content:**

```
1 "firstname": "oren",
2 "lastname": "as",
3 "permission": "user"
```

- Response Status:** 200 OK
- Response Time:** 315 ms
- Response Size:** 582 B
- Response Body (Pretty JSON):**

```
1 {
2   "status": "success",
3   "user": {
4     "_id": "61e3fb0dffd23cb24395a5a5b",
5     "email": "c1@gmail.com",
6     "password": "$2a$10$jamT0zY1GpVyaPkFkLd2u5iB3nKsm2EPnUJNuj9xs8lk1xKyzwey",
7     "firstname": "oren",
8     "lastname": "as",
9     "passwordChangedAt": "2022-01-18T12:25:11.714Z",
10    "permission": "admin",
11    "created": "2022-01-16T11:01:33.471Z",
12    "modified": "2022-01-16T11:01:33.471Z",
13    "__v": 0
14  }
15 }
```

Annotations:

- A green box on the left contains the text: "Firstname and last name changed". It has two arrows pointing to the "firstname" and "lastname" fields in the response body.
- A green box at the bottom contains the text: "The permission will not updates". It has two arrows pointing to the "permission" field in the response body.

Deleting user - we set user active to false

```
var UserSchema = new Schema({  
  email: {  
    type: String,  
    unique: [true, 'please provide email'],  
    required: true,  
    lowercase: true  
  },  
  password: {  
    type: String,  
    required: [true, 'please provide password'],  
    minlength: 6,  
  },  
  firstname: String,  
  lastname: String,  
  photo: String,  
  created: Date,  
  modified: Date,  
  passwordChangedAt: Date,  
  permission: {  
    type: String,  
    enum: ['user', 'author', 'admin'],  
    default: 'user'  
  },  
  passwordResetToken: String,  
  passwordResetExpires: Date,  
  active: {  
    type: Boolean,  
    default: true  
  }  
});
```

Add active property to user

Add pre save function to user schema

```
UserSchema.pre(/^find/, function(next) {  
  this.find({active:true})  
  next()  
})
```

Add deleteMe function to authcontroller.js

```
exports.deleteMe = async function(req, res, next) {  
    //delete user  
  
    let user = await User.findByIdAndUpdate(req.user._id, {active:false})  
    res.status(204).json({  
        status:"success",  
        user:null  
    }) ;  
}
```

Add deleteMe function to UserRouter.js

```
const express = require('express')
var userRouter = express.Router();
var userController = require('./UserController')
var authController = require('./AuthController')

userRouter.post('/signup', authController.signup);
userRouter.post('/login', authController.login);
userRouter.post('/forgotPassword', authController.forgotPassword);
userRouter.patch('/resetPassword/:token', authController.resetPassword);
userRouter.patch('/updateMyPassword',authController.protectSystem ,authController.updatePassword);
userRouter.patch('/updateMe',authController.protectSystem ,authController.updateMe);
userRouter.patch('/deleteMe',authController.protectSystem ,authController.deleteMe);

userRouter.post('/',userController.createUser);
userRouter.get('/:id',userController.getUserById);
userRouter.patch('/:id',userController.updateUserById);
userRouter.delete('/:id',authController.isAdmin ,userController.deleteUserById);
module.exports = userRouter;
```

localhost:3000/api/v1/users/signup

The screenshot shows a Postman interface with the following details:

- URL:** localhost:3000/api/v1/users/signup
- Method:** POST
- Body:** JSON (selected)
- Request Body Content:**

```
1 {
2   "email": "test@gmail.com",
3   "password": "123456",
4   "firstname": "mikel",
5   "lastname": "lewis"
6 }
```

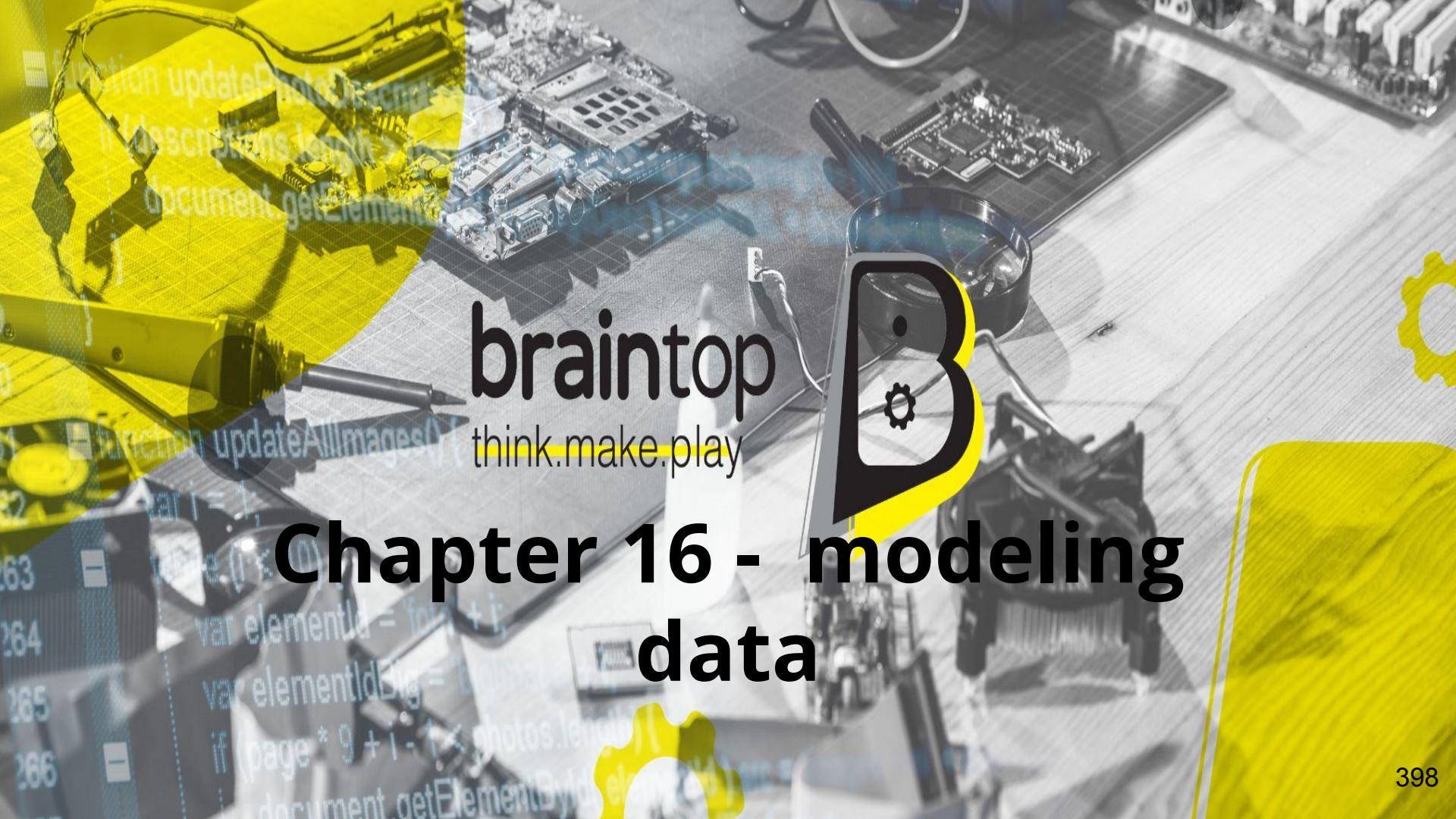
- Response Status:** 201 Created
- Response Time:** 1164 ms
- Response Size:** 741 B
- Response Body (Pretty JSON):**

```
1 {
2   "status": "success",
3   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
4   eyJpZC16IjYxZTk40wEyMjd1NGYwMGJiNDM3NzQ0YsisImhdCI6MTY0MjY5NTA3NiwiZXhwIjoxNjQ2Mjk1MDc2fQ.
5   2Lu-ptLvlqPZZUDilxc5JkKPBylbeY5IHQcl82ye8",
6   "user": {
7     "email": "test@gmail.com",
8     "password": "$2a$10$1zihPFGuC/7f1n2MN7020S.UgKUKEYm0CjjaeTLEn0Shh60IngBjy",
9     "firstname": "mikel",
10    "lastname": "lewis",
11    "permission": "user",
12    "active": true,
13    "id": "61e989a227edf00bb437744a",
14    "created": "2022-01-20T16:11:14.889Z",
15    "modified": "2022-01-20T16:11:14.889Z",
16    "__v": 0
17 }
```

localhost:3000/api/v1/users/deleteMe

```
_id: ObjectId("61e989a227e4f00bb437744a")
email: "test@gmail.com"
password: "$2a$10$1zhPFGuc/7f1nZMN7020S.UgKUKEYmDCjjaeTLEn0ShZh60IngBjy"
firstname: "mikel"
lastname: "lewis"
permission: "user"
active: false
created: 2022-01-20T16:11:14.889+00:00
modified: 2022-01-20T16:11:14.889+00:00
__v: 0
```





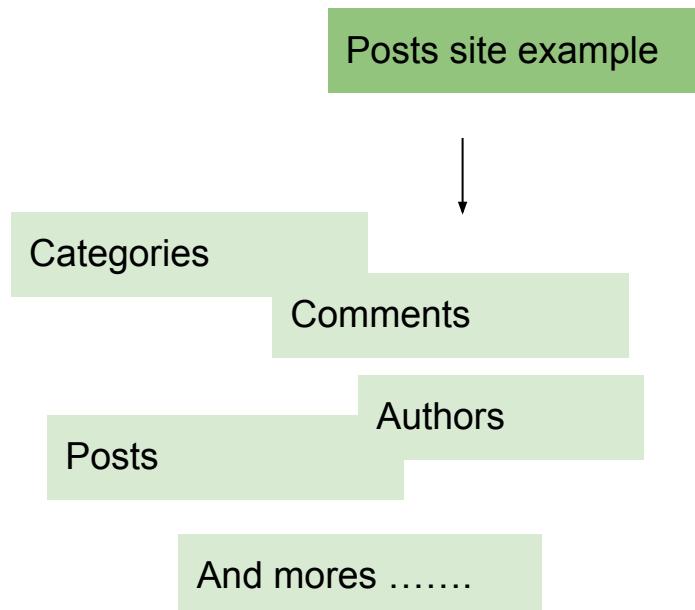
braintop
think.make.play

Chapter 16 - modeling data

Data modeling - link

The key challenge in data modeling is balancing the needs of the application, the performance characteristics of the database engine, and the data retrieval patterns. When designing data models, always consider the application usage of the data (i.e. queries, updates, and processing of the data) as well as the inherent structure of the data itself.

Data modeling - link



In this section we learned about

1. Relationships between data
2. Referencing vs embedding data
3. Type of referencing

Embedded data

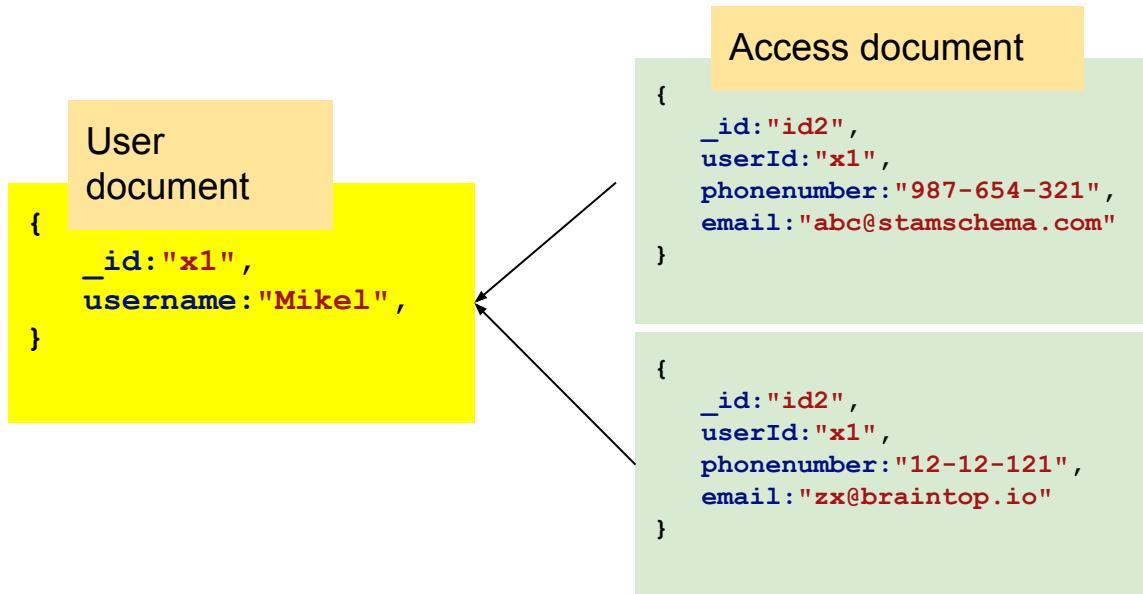
Embedded documents capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures in a field or array within a document. These denormalized data models allow applications to retrieve and manipulate related data in a single database operation.

```
{  
    _id: "123123123",  
    username: "Mikel",  
    password: "123",  
    contact: {  
        phonenumer: "987-654-321",  
        email: "abc@stamschema.com"  
    },  
    permission: {  
        type: "admin",  
        index: 1  
    }  
}
```

The diagram illustrates the structure of a MongoDB document. It shows a main document with fields: _id, username, password, contact, and permission. The contact field is highlighted with a green box and labeled 'Embedded sub-document'. The permission field is also highlighted with a green box and labeled 'Embedded sub-document'. Arrows point from the labels to their respective fields in the JSON code.

references data

References store the relationships between data by including links or *references* from one document to another. Applications can resolve these references to access the related data. Broadly, these are *normalized* data models.



Relationships between data

1:1 course -> title 1 course can have only one title

1:many course -> comments -1 course can have many comments

many:many course -> lesson 1 lesson can have many lessons and one lessons can be in many courses

1:many - link

1:many(**few**) course -> award -1 course can have some awards

1:many(**many**) course -> comment -1 course can have many comments

1:many(**ton**) course -> logs -1 course can have millions of logs

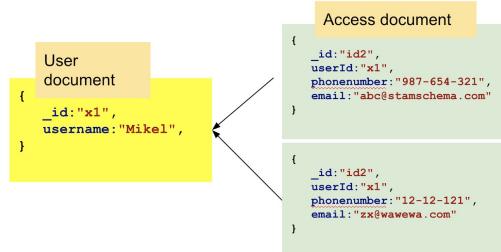
denormalized

With MongoDB, you may embed related data in a single structure or document. These schema are generally known as "denormalized" models, and take advantage of MongoDB's rich documents.

Consider the following diagram:

```
{  
  _id:"123123123",  
  username:"Mikel",  
  password:"123",  
  contact:{  
    phonenumber:"987-654-321",  
    email:"abc@stamschema.com"  
  },  
  permission:{  
    type:"admin",  
    index:1  
  }  
}
```

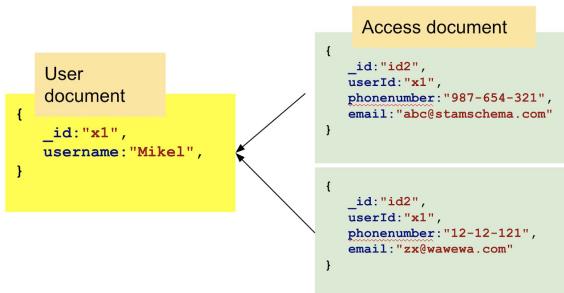
Normalized data models describe relationships using references between documents.



Referencing vs. embedding (denormalization->)

Reference

- 😊 Performance : it's easier to query each document on its own
- 😢 We need 2 query to get the data



denormalization

Embedding

- 😊 Performance : we can get all data in one query
- 😢 impossible to query the embedded document on its own

```
{
  _id:"123123123",
  username:"Mikel",
  password:"123",
  contact:{
    phonenumber:"987-654-321",
    email:"abc@stamschema.com"
  },
  permission:{
    type:"admin",
    index:1
  }
}
```

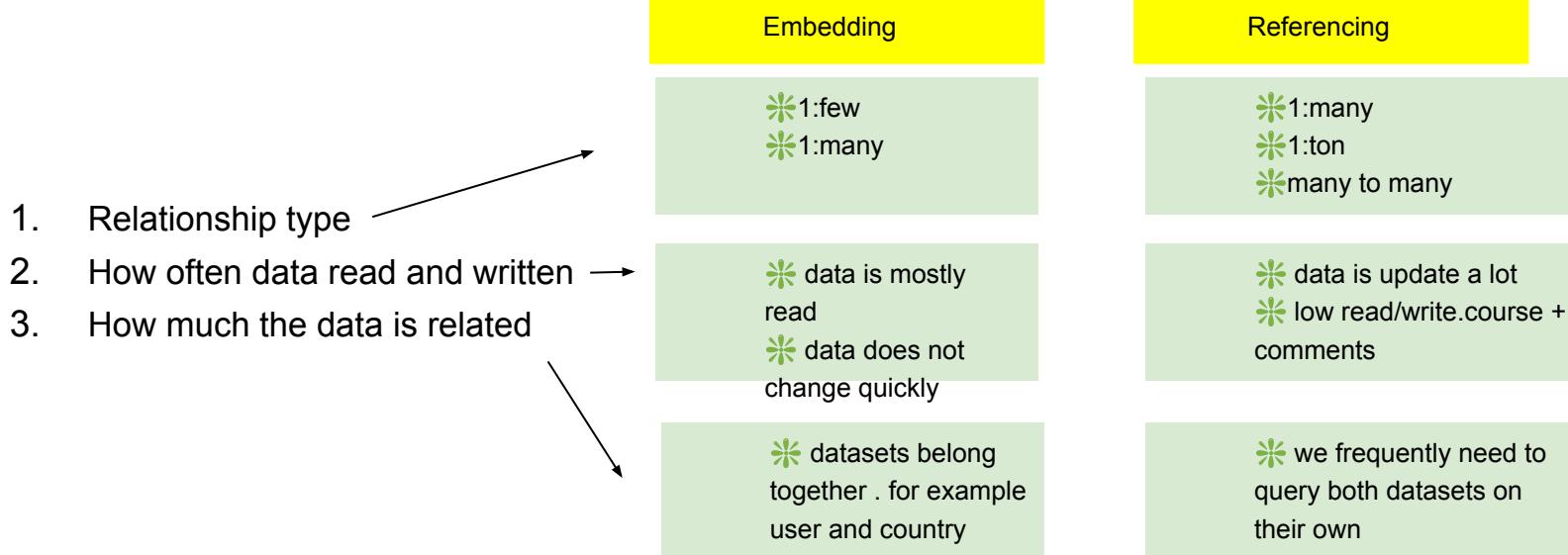
Normalization

How to decide if embed or reference ?

In general, use normalized data models:

- when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- to represent more complex many-to-many relationships.
- to model large hierarchical data sets.

How to decide if embed or reference ?



Types of referencing

Child referencing

parents referencing

two-way referencing

Example data model

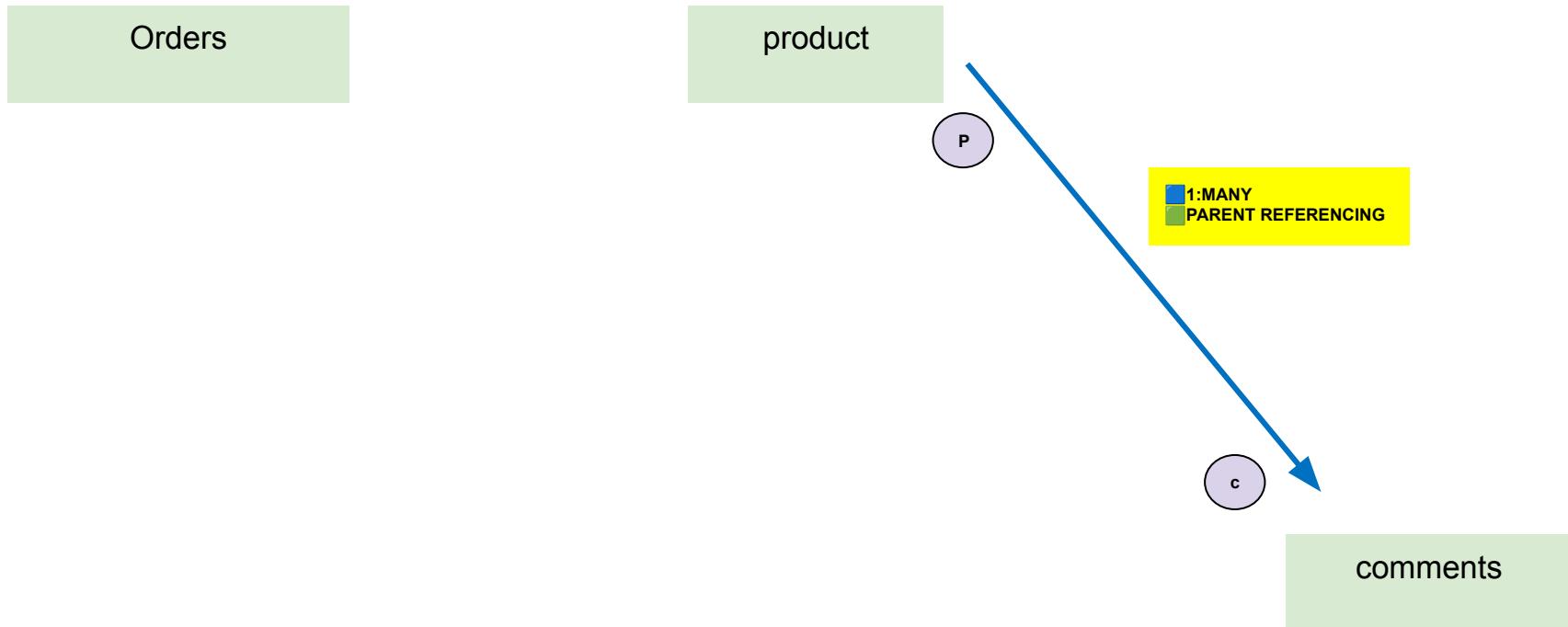
Orders

product

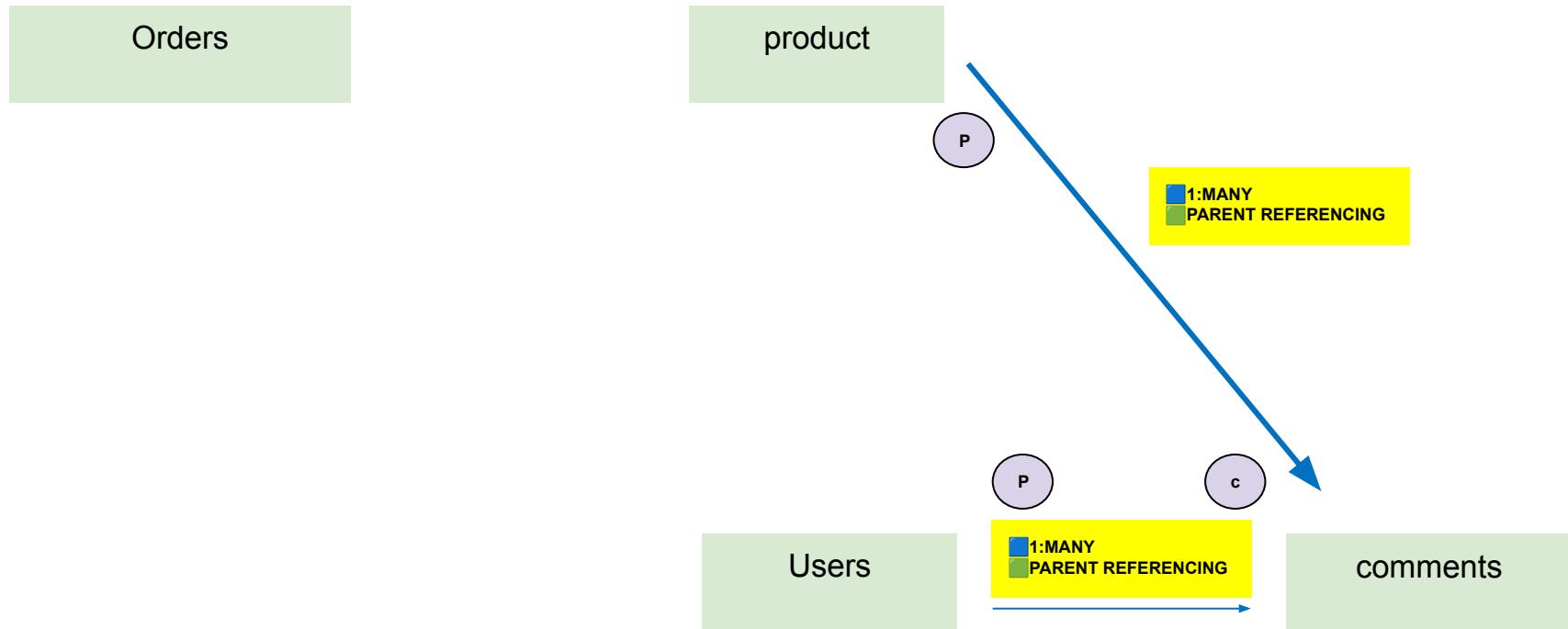
Users

comments

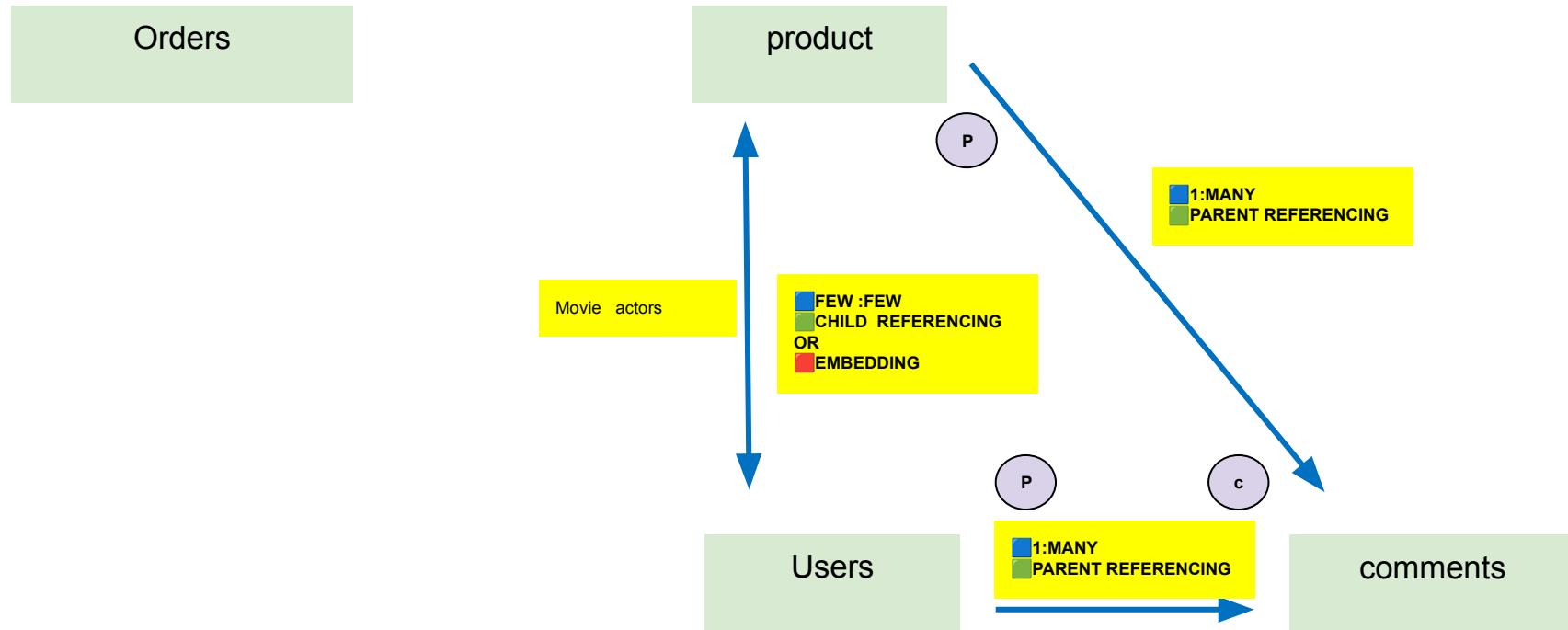
Example data model



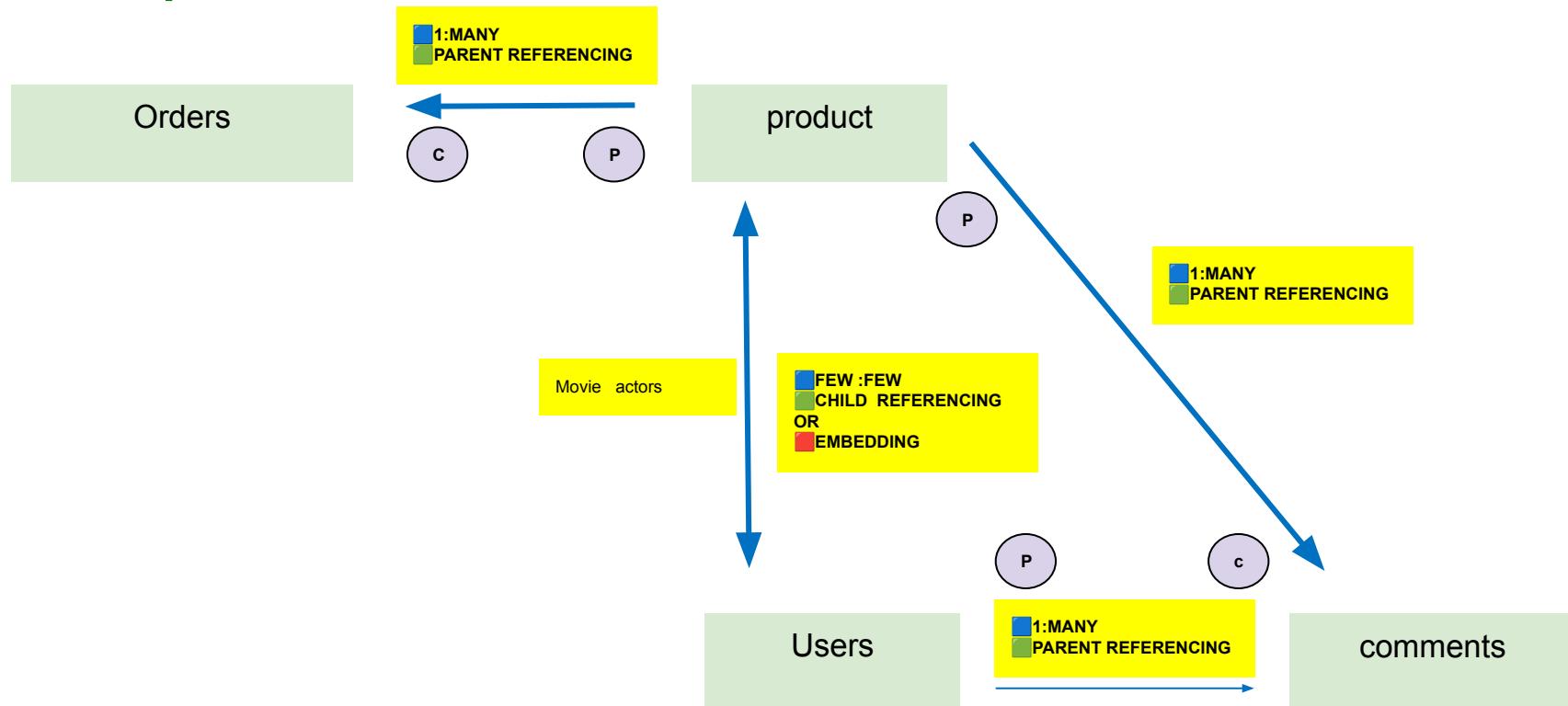
Example data model



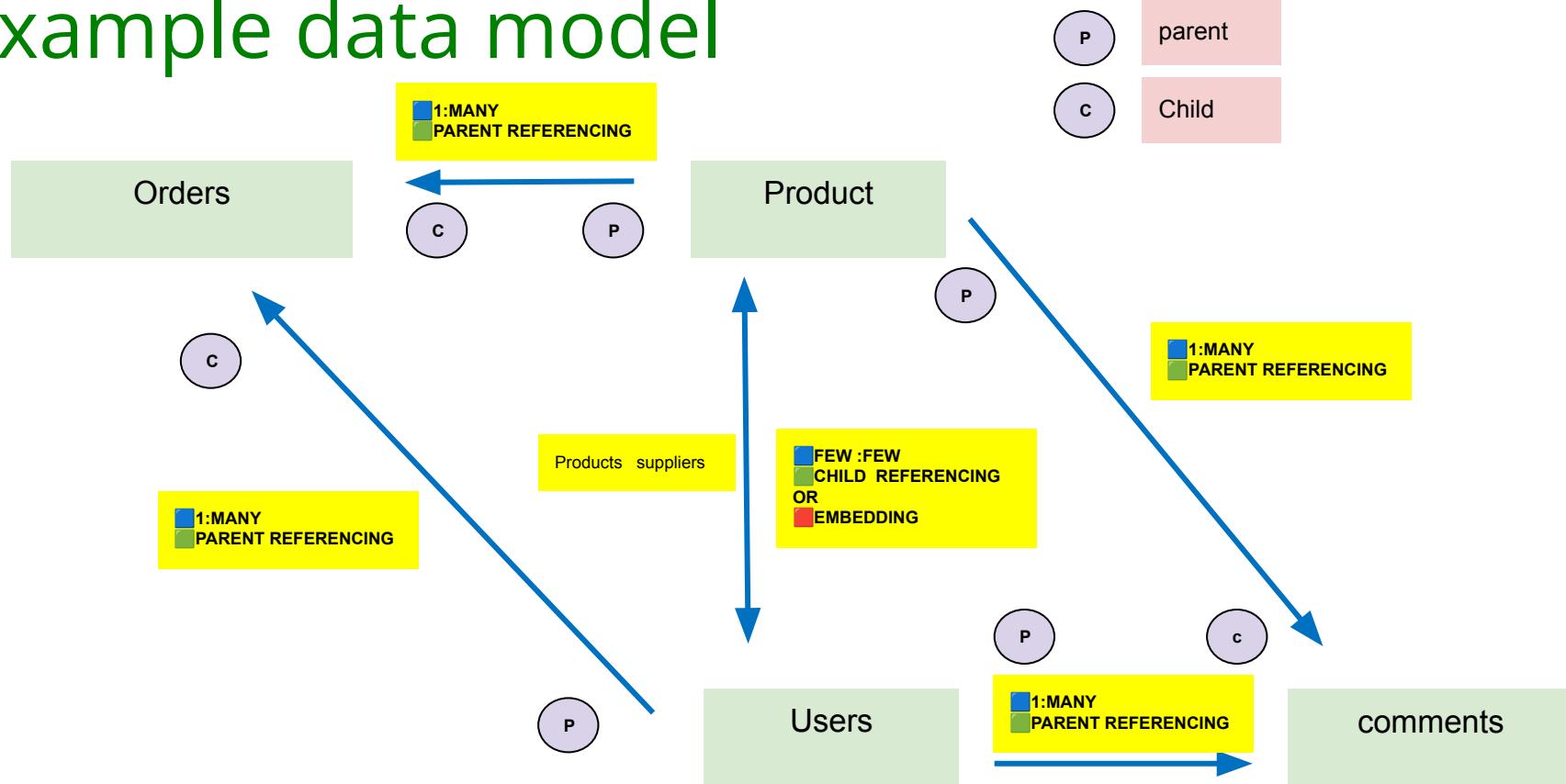
Example data model -



Example data model



Example data model



Old product model

```
var mongoose = require("mongoose");
var Schema = mongoose.Schema;
var ProductSchema = new Schema({
  title: {
    type:String,
    required:[true,'A product must have title'],
    unique:true,
    trim:true
  },
  description:{
    type:String,
    minlength:[5,'Description is minimum 20 characters'],
    maxlength:[1000,'Description is maximum 1000 characters']
  },
  price:{
    type:Number,
    required:[true, 'A product must have price'],
    min:[0,'price must be above 0'],
    max:[10000,'price must be below 10000']
  },
  module.exports = mongoose.model('product', ProductSchema);
```

Modelling product - **child** referencing

Add these fields to Old product model

```
imageCover:{  
  type:String,  
  required:[true,'A tour must have a cover image']  
},  
images:[String],  
suppliers:[  
  {  
    type:mongoose.Schema.ObjectId,  
    ref:'user'  
  }  
],//array of suppliers (supplier is _id of user)  
  
module.exports = mongoose.model('product', ProductSchema);
```

Main product image

More images of product

Array of product suppliers - child referencing

Example of 2 users

```
_id: ObjectId("61e989a227e4f00bb437744a") ←  
email: "test@gmail.com"  
password: "$2a$10$1zhPFGuc/7f1nZMN7020S.UgKUKEYm0CjjaeTLEn0ShZh60IngBjy"  
firstname: "mikel"  
lastname: "lewis"  
permission: "user"  
active: true  
created: 2022-01-20T16:11:14.889+00:00  
modified: 2022-01-20T16:11:14.889+00:00  
__v: 0
```

```
_id: ObjectId("61efb0546f1fe4bb97941438") ←  
email: "test@gmail.com"  
password: "$2a$10$LIB.mB80twSfa9rmenKQNeFWU.Yl37ZP7k7HIhtBQ7dZVERmuk1Ee"  
firstname: "mikel"  
lastname: "lewis"  
permission: "admin"  
active: true  
created: 2022-01-25T08:09:56.480+00:00  
modified: 2022-01-25T08:09:56.480+00:00  
__v: 0
```

Post new product

```
title: "ball-10",
description: "This is a white ball",
price: 1000,
created: "2023-10-09T21:00:00.000Z",
suppliers:[ "61e989a227e4f00bb437744a", "61efb0546f1fe4bb97941438" ],
imageCover:"c.png"
```

Array of product suppliers - child referencing

Post new product

POST localhost:3000/api/v1/products

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 | {
2 |   "title": "ball-10",
3 |   "description": "This is a white ball",
4 |   "price": 1000,
5 |   "created": "2023-10-09T21:00:00.000Z",
6 |   "suppliers": ["61e989a227e4f00bb437744a", "61efb0546f1fe4bb97941438"],
7 |   "imageCover": "c.png"
8 | }
```

Body Cookies Headers (7) Test Results

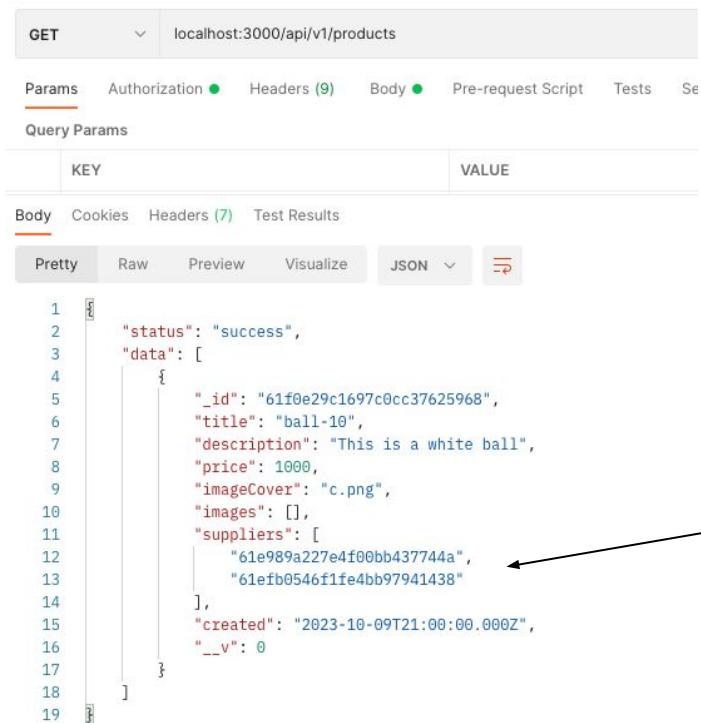
Pretty Raw Preview Visualize JSON

```
1 | {
2 |   "status": "success",
3 |   "data": {
4 |     "title": "ball-10",
5 |     "description": "This is a white ball",
6 |     "price": 1000,
7 |     "imageCover": "c.png",
8 |     "images": [],
9 |     "suppliers": [
10 |       "61e989a227e4f00bb437744a",
11 |       "61efb0546f1fe4bb97941438"
12 |     ],
13 |     "created": "2023-10-09T21:00:00.000Z",
14 |     "_id": "61f0e29c1697c0cc37625968",
15 |     "__v": 0
16 |   }
17 | }
```

Post - Array of product suppliers - child referencing

This product has 2 suppliers

Retrieve product



GET <localhost:3000/api/v1/products>

Params Authorization Headers (9) Body Pre-request Script Tests Se

Query Params

KEY	VALUE
-----	-------

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON ↻

```
1
2   "status": "success",
3   "data": [
4     {
5       "_id": "61f0e29c1697c0cc37625968",
6       "title": "ball-10",
7       "description": "This is a white ball",
8       "price": 1000,
9       "imageCover": "c.png",
10      "images": [],
11      "suppliers": [
12        "61e989a227e4f00bb437744a",
13        "61efb0546f1fe4bb97941438"
14      ],
15      "created": "2023-10-09T21:00:00.000Z",
16      "__v": 0
17    }
18  ]
19 }
```

We want to populate user details

Populate

MongoDB has the join-like `$lookup` aggregation operator in versions ≥ 3.2 . Mongoose has a more powerful alternative called **populate()**, which lets you reference documents in other collections.

Population is the process of automatically replacing the specified paths in the document with document(s) from other collection(s). We may populate a single document, multiple documents, a plain object, multiple plain objects, or all objects returned from a query. Let's look at some examples.

Our Product model has its `suppliers` field set to an array of **ObjectIds**. The `ref` option is what tells Mongoose which model to use during **population**, in our case the user model. All `_ids` we store here must be document `_ids` from the Story model.

Note: `ObjectId`, `Number`, `String`, and `Buffer` are valid for use as refs. However, you should use `ObjectId` unless you are an advanced user and have a good reason for doing so.

Add populate

```
//read by id
exports.getProductId = function(req, res, next) {
  let id = req.params.id
  CurrentProduct.find({_id:id}).populate('user').then(function(data) {
    res.status(200).json({
      status: "success",
      data:data
    })
  }).catch(err=>{
    res.status(404).json({
      status: "fail",
      message: "error: 😱" + err
    })
  })
}
```

This will not work

```
{
  "status": "fail",
  "message": "error: 😱 MongooseError: Cannot populate path
  'user' because it is not in your schema. Set the `strictPopulate`
  option to false to override."
}
```

Add populate

```
//read by id
exports.getProductId = function(req, res, next) {
  let id = req.params.id
  CurrentProduct.find({_id:id}).populate({
    path:'suppliers', ←
    select:'-password -passwordChangedAt -passwordResetToken'
  })
  .then(function(data) {
    res.status(200).json({
      status:"success",
      data:data
    })
  })
  .catch(err=>{
    res.status(404).json({
      status:"fail",
      message:"error:💀" + err
    })
  })
}
```

Path

You can select fields to remove or to add

More about populate and path : [link](#)

Response with user fields

localhost:3000/api/v1/products/61f0e29c1697c0cc37625968

```
GET localhost:3000/api/v1/products/61f0e29c1697c0cc37625968
```

Params Authorization Headers (9) Body Pre-request Script Tests Settings

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON ↻

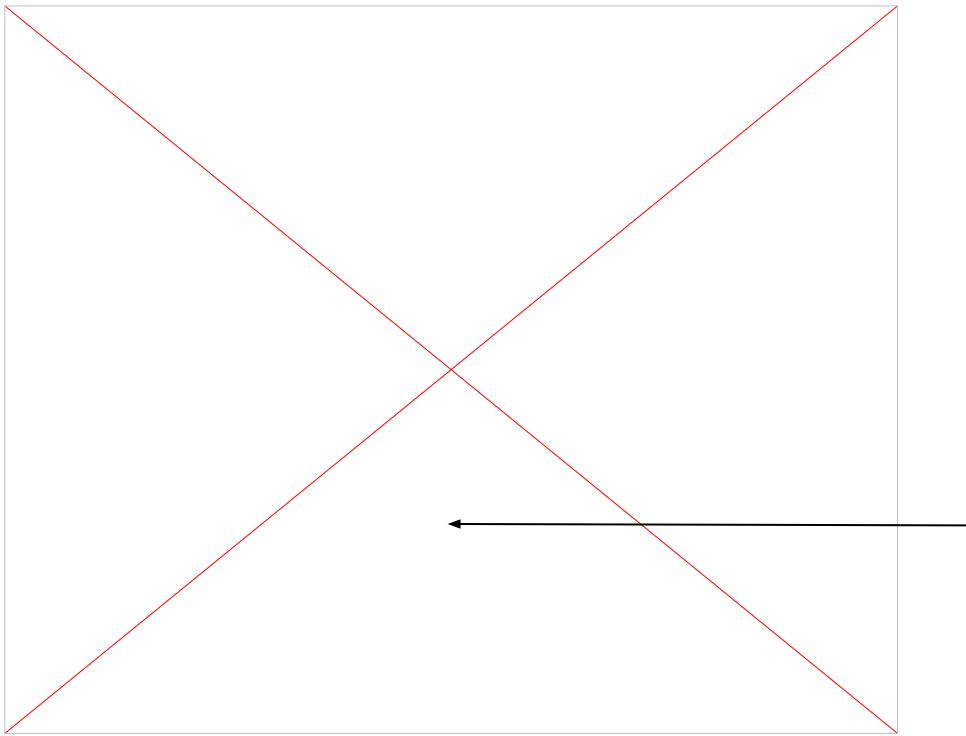
```
8   "price": 1000,
9   "imageCover": "c.png",
10  "images": [],
11  "suppliers": [
12    {
13      "_id": "61e989a227e4f00bb437744a",
14      "email": "test@gmail.com",
15      "firstname": "mikel",
16      "lastname": "lewis",
17      "permission": "user",
18      "active": true,
19      "created": "2022-01-20T16:11:14.889Z",
20      "modified": "2022-01-20T16:11:14.889Z",
21      "__v": 0
22    },
23    {
24      "_id": "61efb0546f1fe4bb97941438",
25      "email": "test1@gmail.com",
26      "firstname": "mikel",
27      "lastname": "lewis",
28      "permission": "admin",
29      "active": true,
30      "created": "2022-01-25T08:09:56.480Z",
31      "modified": "2022-01-25T08:09:56.480Z",
32      "__v": 0
33    }
]
```

Adding middleware for all get queries

```
ProductSchema.pre(/^find/, async function(next) {
  this.populate({
    path: 'suppliers',
    select: '-password -passwordChangedAt -passwordResetToken'
  })
  next();
})
```

Now all **find** queries return suppliers fields

Get all products



Modelling comments - parent referencing

Modelling comments - parent referencing

```
var mongoose = require("mongoose");
var Schema = mongoose.Schema;
var CommentSchema = new Schema({
  title: {
    type:String,
    required:[true,'A comment must have title' ],
  },
  description:{
    type:String,
    minlength:[5,'Description is minimum 20 characters'],
    maxlength:[1000,'Description is maximum 1000 characters']
  },
  rating:{
    type:Number,
    required:[true, 'A comment must have rating'],
    min:1,
    max:5
  },
  product:{
    type:mongoose.Schema.ObjectId, ←
    ref:'product',
    required:[true, 'Product must belong to a comment']
  },
  user:{
    type:mongoose.Schema.ObjectId, ←
    ref:'user',
    required:[true, 'Product must belong to a user']
  },
  created: Date
});
module.exports = mongoose.model('comment', CommentSchema);
```

Parent referencing

Parent referencing

Add CommentController.js

```
var CurrentComment = require('./CommentModel')
exports.createComment = async function(req, res, next) {
  try{
    let p1= req.body;
    var newItem = await CurrentComment.create(p1);
    res.status(201).json({
      status:"success",
      data:newItem
    })
  }
  catch(err){
    res.status(400).json({
      status:"fail",
      message:"error:😱" + err
    })
  }
};

};
```

Add CommentController.js

```
exports.getAllComments = async function(req, res, next) {
  try{
    const comments = await CurrentComment.find()
    res.status(200).json({
      status:"success",
      data:comments
    })
  }
  catch(err){
    res.status(400).json({
      status:"fail",
      message:"error:😱" + err
    })
  }
}
```

Add CommentRouter.js

```
const express = require('express')

var CommentRouter = express.Router();

var CommentController = require('./CommentController')
var authController = require('../users/AuthController')

CommentRouter.post('/', authController.protectSystem, CommentController.createComment);
CommentRouter.get('/', CommentController.getAllComments);

module.exports = CommentRouter;
```

remember: ⚡ Only login user can add comment

Add commentRouter to app.js

```
var express = require('express');
var app = express();
app.use(express.json())
var mongoose = require('mongoose');

var productRouter = require("./api/products/ProductRouter")
var userRouter = require("./api/users/UserRouter")
var commentRouter = require("./api/comments/CommentRouter")

app.use('/api/v1/products', productRouter);
app.use('/api/v1/users', userRouter);
app.use('/api/v1/comments', commentRouter);
```

Post comment - don't forget to be login

POST localhost:3000/api/v1/comments

Params **Authorization** ● Headers (9) Body ● Pre-request Script Tests Settings

Type **Bearer Token**

The authorization header is generated when you send a POST request. [Learn more about authorization](#)

!

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. [Learn more about variables](#)

Don't forget Bearer token

eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9eyJpZCI6IjYxZWZiMDU0NmYxZmU0Yml5Nzk0MTQzOClsImhdCl6MTY0MzE4MzEzMzNSwiZXhwIjoxNjQ2NzgzMTM1fQ.gSDXV4JfCS5GH9oQzwmV4L7klvLHpZYvMPI5mF7Syow

Post comment

POST localhost:3000/api/v1/comments

Params Authorization ● Headers (9) **Body** ● Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {  
2   "title": "great product",  
3   "decription": "comment description",  
4   "rating": 5,  
5   "product": "61f0e29c1697c0cc37625968",  
6   "user": "61e989a227e4f00bb437744a"  
7 }  
8 }
```

Body Cookies Headers (7) Test Results Status: 201 Created

Pretty Raw Preview Visualize JSON

```
1 {  
2   "status": "success",  
3   "data": {  
4     "title": "great product",  
5     "rating": 5,  
6     "product": "61f0e29c1697c0cc37625968",  
7     "user": "61e989a227e4f00bb437744a",  
8     "_id": "61f0fc48bbb7b1c6782fe346",  
9     "__v": 0  
10 }  
11 }
```

Parent references

Populating comments - add to CommentModel.js

```
CommentSchema.pre(/^find/, async function(next) {
```

```
    this.populate({
```

```
        path: 'product',
```

Parent referencing

```
        select: 'title -suppliers'
```

Select only title, delete suppliers field

```
    }).populate({
```

```
        path: 'user',
```

Parent referencing

```
        select: 'firstname lastname -_id'
```

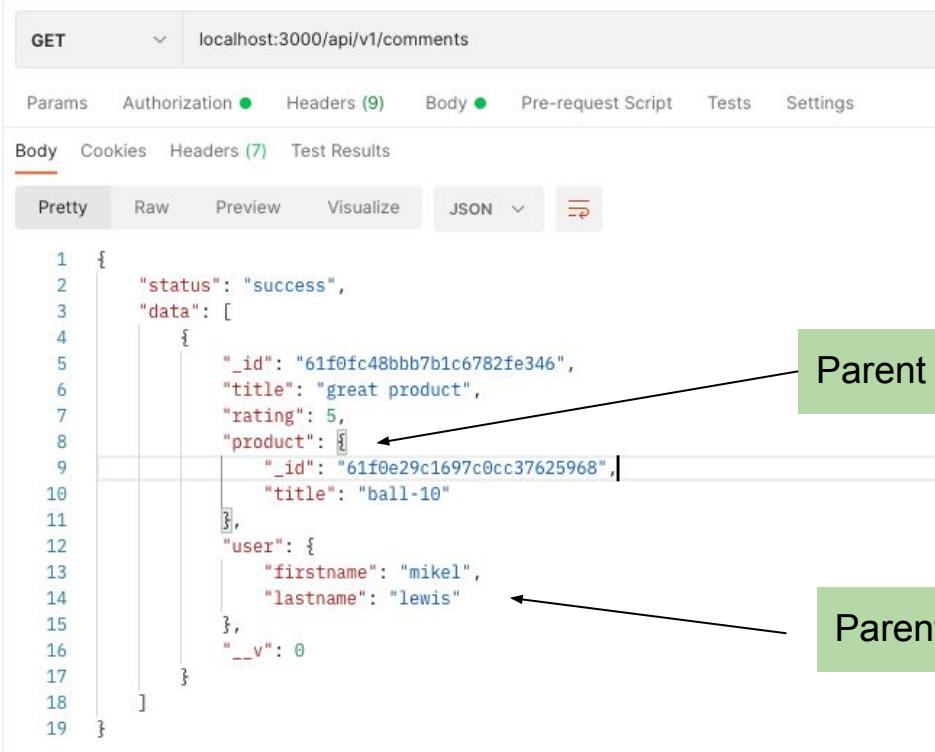
-Select only firstname and lastname
-delete _id field

```
})
```

```
next();
```

```
})
```

Populating comments



The screenshot shows a Postman request for `localhost:3000/api/v1/comments` using the GET method. The response body is displayed in Pretty JSON format, showing an array of comment data. Two annotations with arrows point to specific fields:

- An arrow points from the text "Parent referencing" to the `"product": {}` field in a comment object, which contains a nested object with `_id`, `title`, and `rating` fields.
- An arrow points from the text "Parent referencing" to the `"user": {}` field in a comment object, which contains a nested object with `firstname` and `lastname` fields.

```
1  {
2    "status": "success",
3    "data": [
4      {
5        "_id": "61f0fc48bbb7b1c6782fe346",
6        "title": "great product",
7        "rating": 5,
8        "product": {},
9        "comment": {
10          "_id": "61f0e29c1697c0cc37625968",
11          "title": "ball-10"
12        },
13        "user": {
14          "firstname": "mikel",
15          "lastname": "lewis"
16        },
17        "__v": 0
18      }
19    ]
}
```

Parent referencing

Parent referencing



braintop
think.make.play



Chapter 17 - git Source control

Source control

1. A system that stores all project code
2. Holds historical changes to their code and documentation
 - a. Stores all the versions we released
3. Allows multiple developers to work on the code
 - a. You can see who is responsible for which part of the code
 - b. Prevents collisions and helps deal with them
4. Allows you to simultaneously develop different capabilities

<https://git-scm.com>



The homepage features a large central image showing a network of six white boxes representing repositories, connected by red and blue lines representing branches and merges. To the left of the diagram is a brief introduction to Git's features.

About
The advantages of Git compared to other source control systems.

Documentation
Command reference pages, Pro Git book content, videos and other material.

Downloads
GUI clients and binary releases for all major platforms.

Community
Get involved! Bug reporting, mailing list, chat, development and more.

Pro Git by Scott Chacon and Ben Straub is available to [read online for free](#). Dead tree versions are available on [Amazon.com](#).

Latest source Release
2.35.1
[Release Notes \(2022-01-29\)](#)

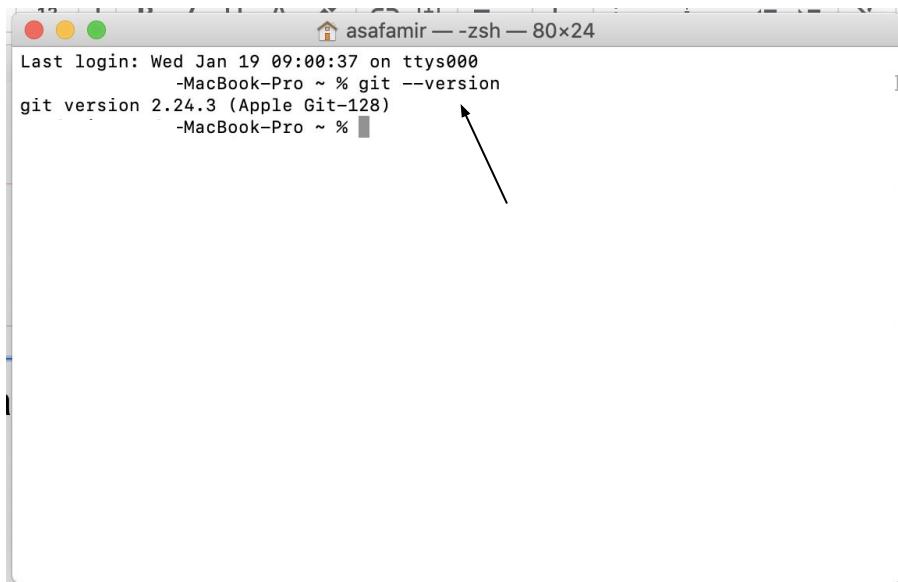
[Download for Mac](#)

[Mac GUIs](#) [Tarballs](#)
[Windows Build](#) [Source Code](#)

Git -- version

Go to terminal and check if git already installed

Write on terminal : git --version



The screenshot shows a terminal window titled "asafamir — zsh — 80x24". The window contains the following text:

```
Last login: Wed Jan 19 09:00:37 on ttys000
-MacBook-Pro ~ % git --version
git version 2.24.3 (Apple Git-128)
-MacBook-Pro ~ %
```

A black arrow points from the bottom right towards the bottom of the terminal window.

Installing Git

Git is a version-control system for tracking changes in any set of files, and coordinating work among programmers during software development.

Before installing the heroku CLI you also need to install git on your computer.

<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

Next, open a terminal and enter config git with **your own** credentials:

```
git config --global user.name "John Doe"
```

```
git config --global user.email johndoe@example.com
```

Working with git

How to work with source control

Using git and github

Why is it good to work with git

- Suppose we develop a system and we want to add some capability to it and we change something in the code and now it has stopped working
- Adding a new feature to the system
- what can be done ?

Why is it good to work with git

If we work with git we can manage versions and go back to previous versions in the code.

Example:

We started working on a new module in the code, and found a bug in the code and it is not working. We can update the code in one version backwards and return to the employee code, at the same time correcting the code and updating again.

Why is it good to work with git

Usually in development projects more than one developer works on the code. Using git we can synchronize between all developers.

Version management

Think of versions of Android operating system development



Source control

- A system that contains all the project code
- The system contains a change history of all code, including documentation.
- Multiple developers can work on the same system and develop different products simultaneously

Basic concepts in source control

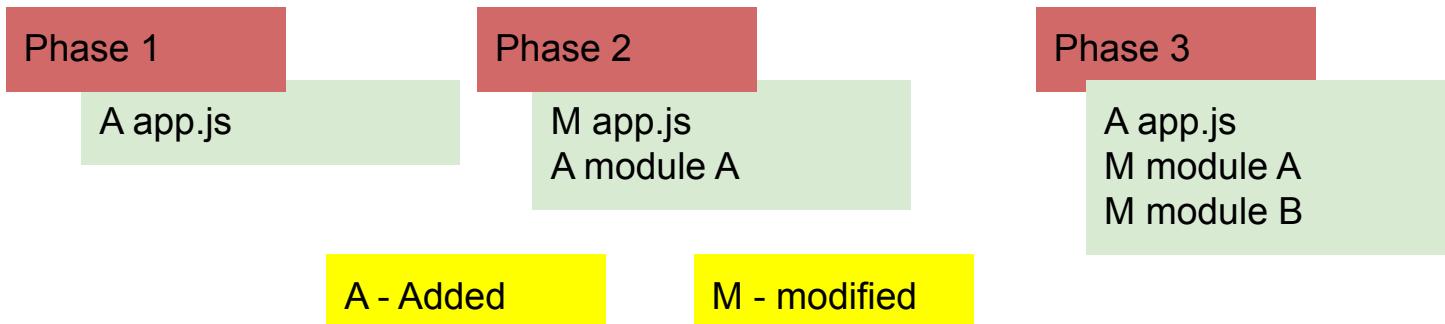
Repository

- For each project we will create a repository
- The repository will hold the change history in the code.
Which include the operation operation, versions, parallel development axes

changeset

Collection of changes between two sub-tasks

Changes can be - adding files, updating or editing a file, deleting files.



- Each set of changes has a number called - revision and is unique - represents the code version
- And every revision has a name called a tag

Repository

1. Managed code
2. For each project - we will create a Repository
 - a. In a company with 10 projects - will hold 10 repositories
3. The repository holds the change history in the code
 - a. All changes (delete / create / change) in files
 - b. Who made the change
 - c. Versions we released
 - d. Past and present development axes (branches)

Source control types

1. There are two main types of Source Control:
2. 1. Centralized Source Control
 - a. For example: cvs, svn
 - b. All change history is stored on the server (and only on it)
 - c. Each operation will be performed directly in front of the server
 - d. Such work requires from us an active connection all the time
 - i. Every action is performed in front of the network
 - ii. This may slow down the work

Source control types

2. Distributed Source Control

- Each user has a full copy of the repository.
- Working in front of a local database on the PC
- At the end of the work update on the server
 - Push to a central server
- In order to receive changes - updated from the server
 - Perform a pull from the main server
- Main players:
 - git and the Github code storage service
 - Mercurial and Bitbucket code storage service

git

- Distributed VC
- The use is made using the git tool
- There are a number of graphical interfaces developed
- Recommended graphical interfaces:
 - SourceTree (Recommended!)
 - Tortoise git
- There is integration with git for most IDEs (eclipse, visual studio,...)

Common commands

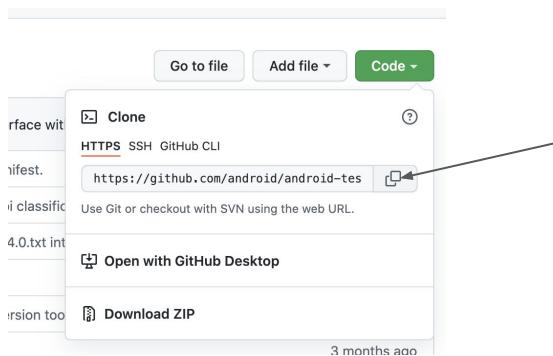
1. Clone
2. Commit
3. Add
4. Commit
5. Push
6. Pull
7. Update
8. merge
9. branch

Common Commands

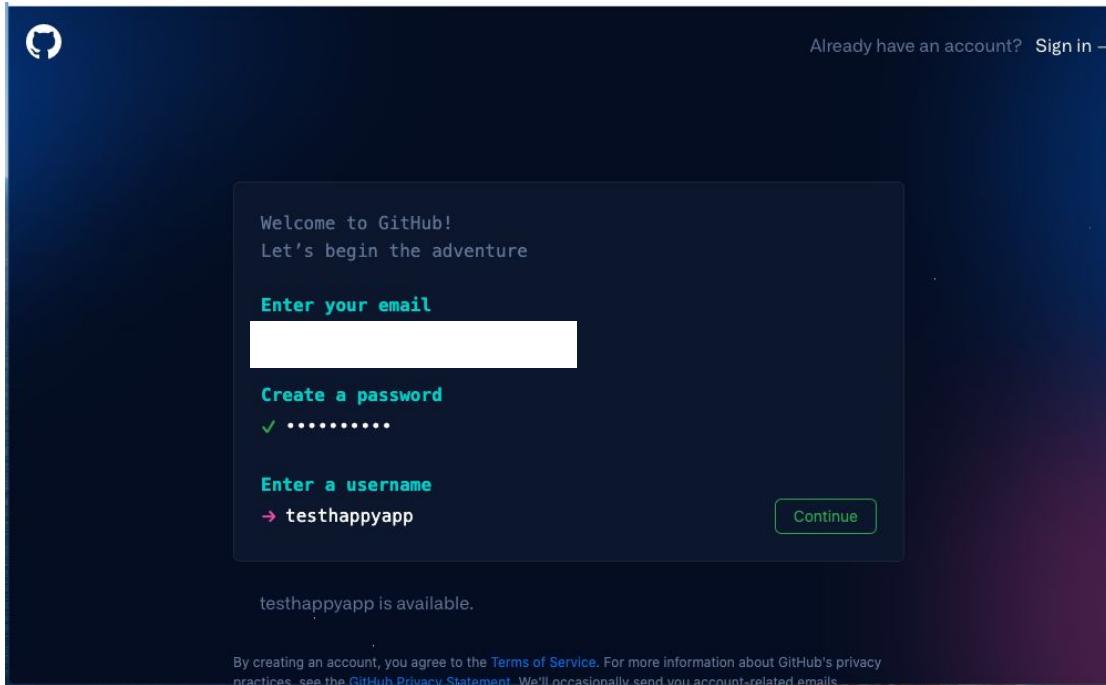
- touch <filename> : create file
- rm <filename> delete file
- git add .
- git commit -m"first"
- git diff - different between last version
- git status
- git log

git clone

- Each user holds a full copy of the Repository
- The clone operation creates the above copy on the computer
- The command will create a folder on the computer, and inside it will sit a complete mirror image of all the latest code.
- Command structure:
 - `git clone https://user@server.com/repo-name`



Signup to github.com



Create first github repository

The screenshot shows a GitHub user interface with a dark header bar containing the GitHub logo, a search bar, and navigation links for Pull requests, Issues, Marketplace, and Explore.

Create your first project
Ready to start building? Create a repository for a new idea or bring over an existing repository to keep contributing to it.

Create repository (button) | **Import repository** (link)

Recent activity
When you take actions across GitHub, we'll provide links to that activity here.

Learn Git and GitHub without any code! (with an 'x' icon to close the box)
Using the Hello World guide, you'll create a repository, start a branch, write comments, and open a pull request.

Read the guide (button) | **Start a project** (button)

All activity

Introduce yourself
The easiest way to introduce yourself on GitHub is by creating a README in a repository about you! You can start here:

testhappyapp / README.md

```
1 - 🌟 Hi, I'm @testhappyapp
2 - 🌐 I'm interested in ...
3 - 🌱 I'm currently learning ...
4 - 💬 I'm looking to collaborate on ...
5 - 📧 How to reach me ...
6
```

Dismiss this | **Continue** (button)

First app

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Owner * **Repository name** *

testhappyapp / firstapp ✓

Great repository names are short and memorable. Need inspiration? How about [solid-memory](#)?

Description (optional)

my first app

Public
Anyone on the internet can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Initialize this repository with:

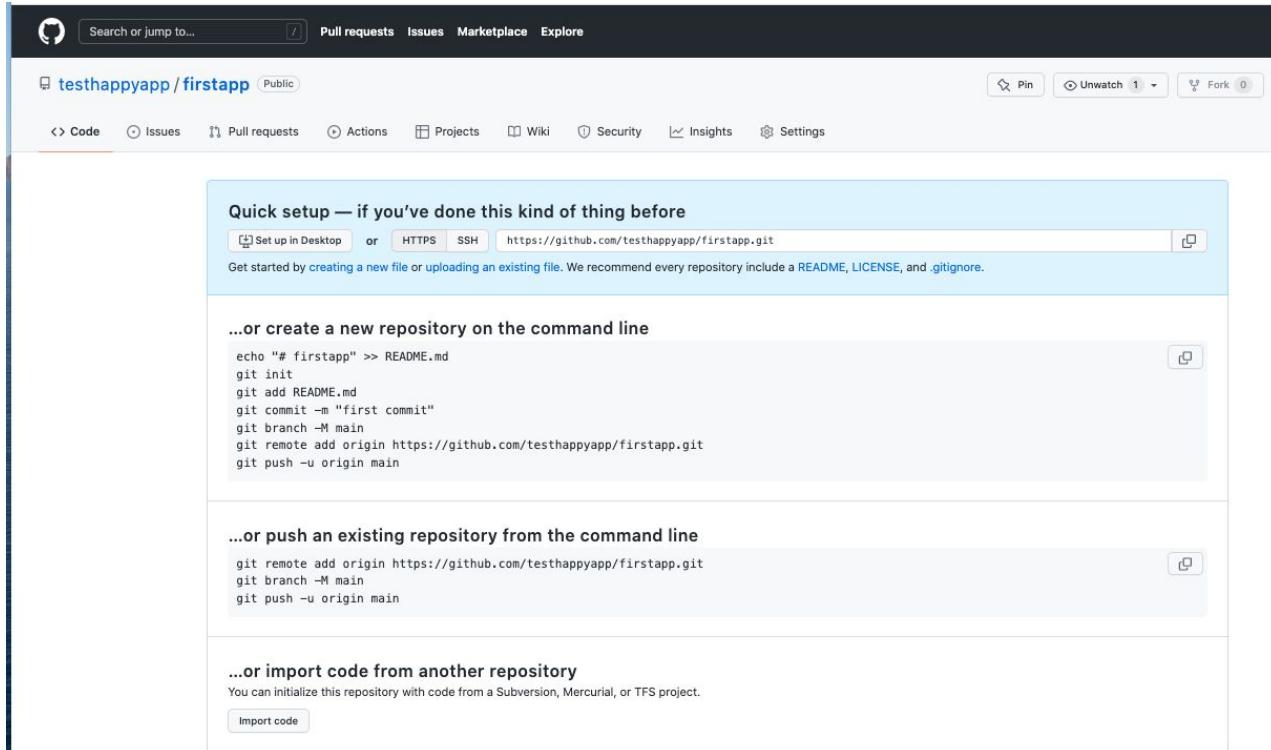
Skip this step if you're importing an existing repository.

Add a README file
This is where you can write a long description for your project. [Learn more](#).

Add .gitignore
Choose which files not to track from a list of templates. [Learn more](#).

Choose a license
A license tells others what they can and can't do with your code. [Learn more](#).

Create repository



The screenshot shows the GitHub interface for creating a new repository. At the top, there's a search bar and navigation links for Pull requests, Issues, Marketplace, and Explore. Below that, the repository name "testhappyapp / firstapp" is shown, along with a Public link, a Pin button, an Unwatch button (with 1 watch), a Fork button (with 0 forks), and a Settings button.

The main content area has a light blue header titled "Quick setup — if you've done this kind of thing before". It includes a "Set up in Desktop" button, an "HTTPS" button, an "SSH" button, and a URL "https://github.com/testhappyapp/firstapp.git". A note below says "Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore." There are three expandable sections:

- ...or create a new repository on the command line**

```
echo "# firstapp" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/testhappyapp/firstapp.git
git push -u origin main
```
- ...or push an existing repository from the command line**

```
git remote add origin https://github.com/testhappyapp/firstapp.git
git branch -M main
git push -u origin main
```
- ...or import code from another repository**

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

[Import code](#)

Create new folder on visual studio

1. From visual studio create new folder and select name for the folder.
2. From terminal cd to the folder

On terminal write command

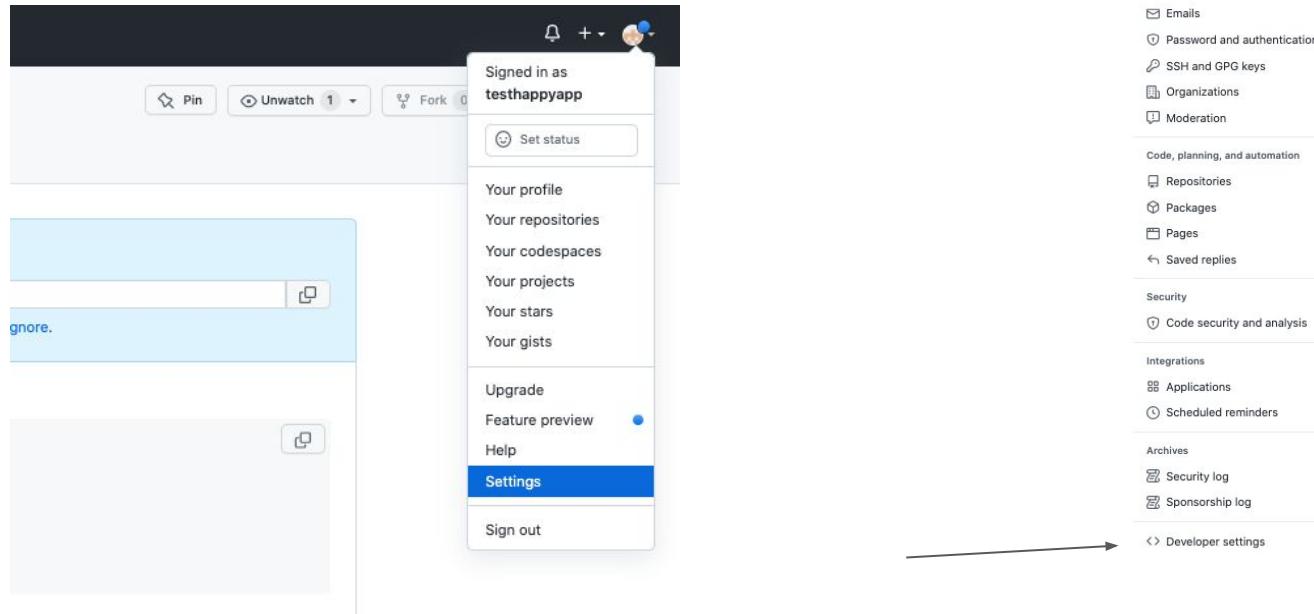
```
echo "# test" >> README.md  
git init  
git add README.md  
git commit -m "first commit"  
git branch -M main  
git remote add origin  
https://github.com/testhappyapp/firstapp.git  
git push -u origin main
```

README.md file
created

chap17-firstapp
| README.md

If you can't push, you have to create token

Make token - go to github



Create token

Settings / Developer settings

GitHub Apps
OAuth Apps
Personal access tokens

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note
test

What's this token for?

Expiration *
30 days The token will expire on Tue, Mar 1 2022

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input type="checkbox"/> repo:status	Access commit status
<input type="checkbox"/> repo_deployment	Access deployment status
<input type="checkbox"/> public_repo	Access public repositories
<input type="checkbox"/> repo:invite	Access repository invitations
<input type="checkbox"/> security_events	Read and write security events
<input checked="" type="checkbox"/> workflow	Update GitHub Action workflows
<input checked="" type="checkbox"/> write:packages	Upload packages to GitHub Package Registry
<input type="checkbox"/> read:packages	Download packages from GitHub Package Registry
<input checked="" type="checkbox"/> delete:packages	Delete packages from GitHub Package Registry
<input checked="" type="checkbox"/> admin:org	Full control of orgs and teams, read and write org projects
<input type="checkbox"/> write:org	Read and write org and team membership, read and write org projects
<input type="checkbox"/> read:org	Read org and team membership, read org projects
<input checked="" type="checkbox"/> admin:public_key	Full control of user public keys

Create token

Settings / Developer settings

New personal access token

GitHub Apps
OAuth Apps
Personal access tokens

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note
test

What's this token for?

Expiration *
30 days The token will expire on Tue, Mar 1 2022

Select scopes
Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input type="checkbox"/> repo:status	Access commit status
<input type="checkbox"/> repo_deployment	Access deployment status
<input type="checkbox"/> public_repo	Access public repositories
<input type="checkbox"/> repo:invite	Access repository invitations
<input type="checkbox"/> security_events	Read and write security events
<input checked="" type="checkbox"/> workflow	Update GitHub Action workflows
<input checked="" type="checkbox"/> write:packages	Upload packages to GitHub Package Registry
<input type="checkbox"/> read:packages	Download packages from GitHub Package Registry
<input checked="" type="checkbox"/> delete:packages	Delete packages from GitHub Package Registry
<input checked="" type="checkbox"/> admin:org	Full control of orgs and teams, read and write org projects
<input type="checkbox"/> write:org	Read and write org and team membership, read and write org projects
<input type="checkbox"/> read:org	Read org and team membership, read org projects
<input checked="" type="checkbox"/> admin:public_key	Full control of user public keys

Copy the token

Some of the scopes you've selected are included in other scopes. Only the minimum set of necessary scopes has been saved. [X](#)

Settings / Developer settings

GitHub Apps

OAuth Apps

Personal access tokens

Personal access tokens

Generate new token

Revoke all

Tokens you have generated that can be used to access the [GitHub API](#).

Make sure to copy your personal access token now. You won't be able to see it again!

✓ ghp_50xt0YkJitQcsP6cIv3MSNCDI0w2uR48pQvF [Copy](#) [Delete](#)

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).



© 2022 GitHub, Inc.

[Terms](#)

[Privacy](#)

[Security](#)

[Status](#)

[Docs](#)

[Contact GitHub](#)

[Pricing](#)

[API](#)

[Training](#)

[Blog](#)

[About](#)

On terminal write command

On terminal :

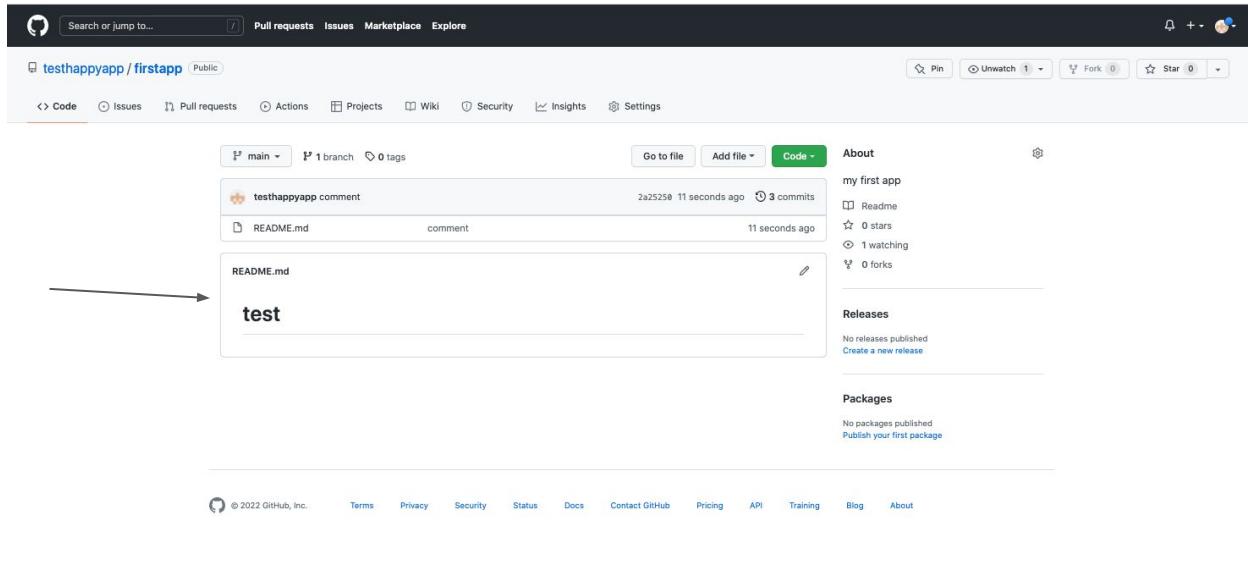
```
git remote set-url origin https://<TOKEN>@github.com/<USERNAME>/<REPOSITORYNAME>.git  
git remote set-url origin https://ghp_50xt0YkJitQcsP6cIv3MSNCDI0w2uR48pQvF@github.com/testhappyapp/firstapp.git
```

Now try to push :

```
git push -u origin main  
asafamir@Asafs-MBP chap17-firstapp % git push -u origin main  
s: 6, done.  
Counting objects: 100% (6/6), done.  
Delta compression using up to 16 threads  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (6/6), 436 bytes | 436.00 KiB/s, done.  
Total 6 (delta 0), reused 0 (delta 0)  
To https://github.com/testhappyapp/firstapp.git  
 * [new branch]      main -> main  
Branch 'main' set up to track remote branch 'main' from 'origin'.  
asafamir@Asafs-MBP chap17-firstapp %
```

Change github.com

The file we create



Change file README.md content



The screenshot shows a terminal window with the following content:

```
 README.md ×
chap17-firstapp > README.md > # test hello world
1 | # test hello world

git add .
git commit -am "second commit"
git push -u origin main

MBP chap17-firstapp % git add . ←
MBP chap17-firstapp % git commit -am "second commit" ←

[main 7ad3140] second commit
 1 file changed, 1 insertion(+), 1 deletion(-)
MBP chap17-firstapp % git push -u origin main ←

Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 263 bytes | 263.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/testhappyapp/firstapp.git
 2a25250..7ad3140  main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
MBP chap17-firstapp %
```

Red arrows point to the command `git add .`, the commit message `"second commit"`, and the command `git push -u origin main`.

After push to github

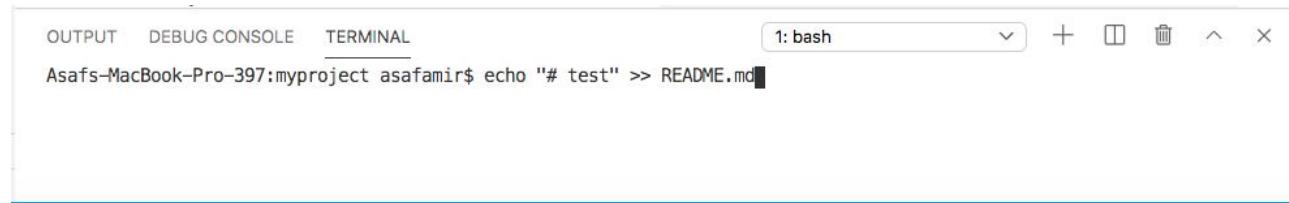
The screenshot shows a GitHub repository page for the repository 'testhappyapp/firstapp'. The repository is public. The main navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, and Wiki. Below the navigation bar, there are status indicators: 'main' branch selected (1 branch, 0 tags), 'Go to file' button, and an 'Add' button. A commit history section shows one commit from 'testhappyapp' titled 'second commit' with hash '7ad3140' and a timestamp of '3 minutes ago'. The commit details show the file 'README.md' was updated with the content 'second commit'. The file content view shows the text 'test hello world'.

Let's repeat what we did

on the terminal :

```
echo "# test" >> README.md
```

Create readme file



A screenshot of a terminal window. At the top, there are tabs labeled "OUTPUT", "DEBUG CONSOLE", and "TERMINAL". The "TERMINAL" tab is underlined. To the right of the tabs is a dropdown menu showing "1: bash" and various icons for new tabs, windows, and closing the terminal. Below the tabs, the terminal prompt shows "Asafs-MacBook-Pro-397:myproject asafamir\$". A command is typed into the terminal: "echo '# test' >> README.md". The command has been partially executed, as indicated by the cursor at the end of the line.

on the terminal :

```
git init
```

Initialize git



The screenshot shows a terminal window with three tabs: OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is selected and contains the following text:

```
Asafs-MacBook-Pro-397:myproject asafamir$ echo "# test" >> README.md
Asafs-MacBook-Pro-397:myproject asafamir$ git init ←
Initialized empty Git repository in /Users/asafamir/Desktop/myproject/.git/
Asafs-MacBook-Pro-397:myproject asafamir$ █
```

A red arrow points to the command `git init`.

on the terminal :

```
git add README.md
```

Add file only on my computer

OUTPUT DEBUG CONSOLE TERMINAL

1: bash



```
Asafs-MacBook-Pro-397:myproject asafamir$ git add README.md
Asafs-MacBook-Pro-397:myproject asafamir$ git commit -m "first commit" ←
[master (root-commit) a52092b] first commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
Asafs-MacBook-Pro-397:myproject
```

on the terminal

```
git commit -m "first commit"
```

Add change set

OUTPUT DEBUG CONSOLE TERMINAL

1: bash

```
Asafs-MacBook-Pro-397:myproject asafamir$ git commit -m "first commit" ←
[master (root-commit) a52092b] first commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
Asafs-MacBook-Pro-397:myproject          it branch -M main
Asafs-MacBook-Pro-397:myproject
```

git branch -M main

on the terminal

git branch -M main

The screenshot shows a terminal window with the following content:

```
OUTPUT DEBUG CONSOLE TERMINAL
1: bash + □ □ ×
[master (root-commit) a52092b] first commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
Asafs-MacBook-Pro-397:myproject git branch -M main ←
Asafs-MacBook-Pro-397:myproject git remote add origin https://github.com/appschool1/test.git
Asafs-MacBook-Pro-397:myproject
```

A red arrow points to the command `git branch -M main`.

Or set with token

```
git remote set-url origin https://ghp_50xt0YkJitQcsP6cIv3MSNCDI0w2uR48pQvF@github.com/testhappyapp/firstapp.git
```

Git status

The git status command shows us all the changes that have been made since the last version. The changes will not be valid until we commit. The commit command saves our git changes to the computer.

Git add

When we create a file we must use the git add command. The command will add all the files.

```
Asafs-MacBook-Pro-397:git-lesson asafamir$ git add index.html ←
Asafs-MacBook-Pro-397:git-lesson asafamir$ git status
On branch main
Your branch is up-to-date with 'origin/main'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   index.html

Asafs-MacBook-Pro-397:git-lesson asafamir$ █
```

Git commit

When we finish working on a file, we can, we want to give the collection of changes a name. The change collection reminder is **changeset**

Example :

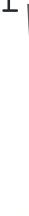
```
Git commit -m"my description"
```

```
Git commit -am"my description"
```

```
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)
```

```
  new file:  index.html
```

```
Asafs-MacBook-Pro-397:git-lesson asafamir$ git commit -am"description 1"  
[main 660858b] description 1  
 1 file changed, 5 insertions(+)  
 create mode 100644 index.html  
Asafs-MacBook-Pro-397:git-lesson asafamir$
```



Git rm

The command will delete a file

Example:

```
git rm index.html
```

```
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)  
  
      new file:   index.html  
  
Asafs-MacBook-Pro-397:git-lesson asafamir$ git commit -am"description 1"  
[main 660858b] description 1  
 1 file changed, 5 insertions(+)  
  create mode 100644 index.html  
Asafs-MacBook-Pro-397:git-lesson asafamir$ git rm index.html   
rm 'index.html'  
Asafs-MacBook-Pro-397:git-lesson asafamir$ █
```

Git log

View change history

Example:

```
git log
```

```
Asafs-MacBook-Pro-397:git-lesson asafamir$ git log
commit 660858b1bd0bee6f261c2544f6d7d09a0fb0d12db (HEAD -> main)
Author: yaya games <yayagames100@gmail.com>
Date: Mon Nov 2 07:39:22 2020 +0200

    description 1

commit 0dce20a6a68ac8c5f87ecbd0a457062c1c89c342 (origin/main)
Author: yaya games <yayagames100@gmail.com>
Date: Mon Nov 2 07:33:40 2020 +0200

    first commit
Asafs-MacBook-Pro-397:git-lesson asafamir$ █
```

Git diff

Shows the changes in the files

Example:

git diff

```
Asafs-MacBook-Pro-397:git-lesson asafamir$ git diff
diff --git a/index.html b/index.html
index 427f49b..97d009e 100644
--- a/index.html
+++ b/index.html
@@ -1,5 +1,5 @@
<html>
  <body>
-    hello
+    hello, hi
  </body>
</html>
\ No newline at end of file
Asafs-MacBook-Pro-397:git-lesson asafamir$ █
```

More commands

<https://github.com/joshnh/Git-Commands>



braintop
think.make.play



Chapter 18 - git & heroku

<https://git-scm.com>

If you
installed git,
skip to slide
486

The screenshot shows the official website for Git. At the top left is the Git logo (a red octopus icon) followed by the word "git". Below it is the tagline "-local-branching-on-the-cheap". A search bar with the placeholder "Search entire site..." is positioned at the top right. The main content area features a 3D diagram of several white cubes representing repositories, connected by red and blue lines to illustrate branching and merging. To the left of the diagram, there's a brief introduction to what Git is and how it compares to other systems. Below the diagram are four main navigation links: "About", "Documentation", "Downloads", and "Community". Each link has a corresponding icon and a brief description. On the right side, there's a section for the "Latest source Release 2.35.1" with a "Download for Mac" button. At the bottom, there are links for "Mac GUIs", "Tarballs", "Windows Build", and "Source Code".

git -local-branching-on-the-cheap

Search entire site...

Git is a [free and open source](#) distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is [easy to learn](#) and has a [tiny footprint with lightning fast performance](#). It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like [cheap local branching](#), convenient [staging areas](#), and [multiple workflows](#).

About
The advantages of Git compared to other source control systems.

Documentation
Command reference pages, Pro Git book content, videos and other material.

Downloads
GUI clients and binary releases for all major platforms.

Community
Get involved! Bug reporting, mailing list, chat, development and more.

Latest source Release
2.35.1
[Release Notes \(2022-01-29\)](#)

[Download for Mac](#)

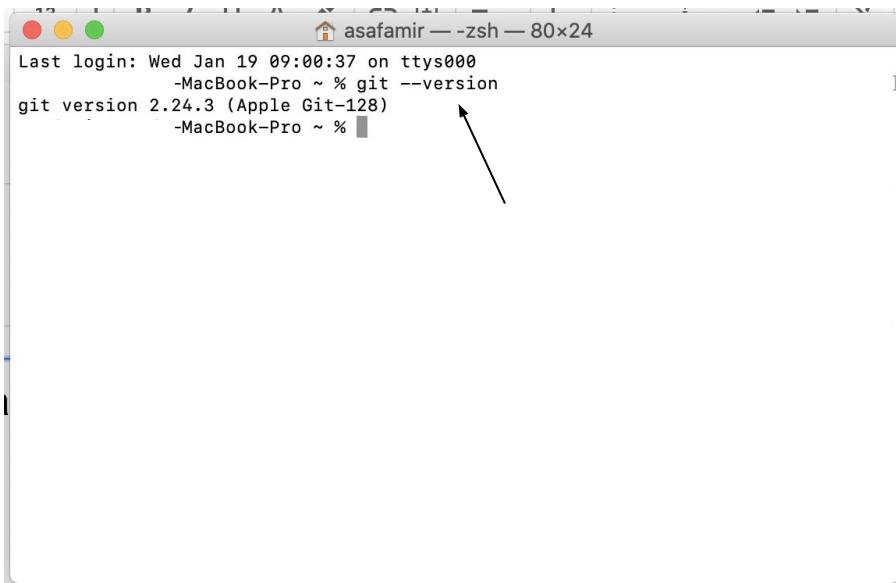
Pro Git by Scott Chacon and Ben Straub is available to [read online for free](#). Dead tree versions are available on [Amazon.com](#).

[Mac GUIs](#) [Tarballs](#)
[Windows Build](#) [Source Code](#)

Git installed ?

Go to terminal and check if git already installed

Write on terminal : git --version



The screenshot shows a terminal window titled "asafamir — zsh — 80x24". The window contains the following text:

```
Last login: Wed Jan 19 09:00:37 on ttys000
-MacBook-Pro ~ % git --version
git version 2.24.3 (Apple Git-128)
-MacBook-Pro ~ %
```

A black arrow points from the bottom right towards the command "git --version" in the terminal output.

Installing Git

Git is a version-control system for tracking changes in any set of files, and coordinating work among programmers during software development.

Before installing the heroku CLI you also need to install git on your computer.

<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

Next, open a terminal and enter config git with **your own** credentials:

```
git config --global user.name "John Doe"
```

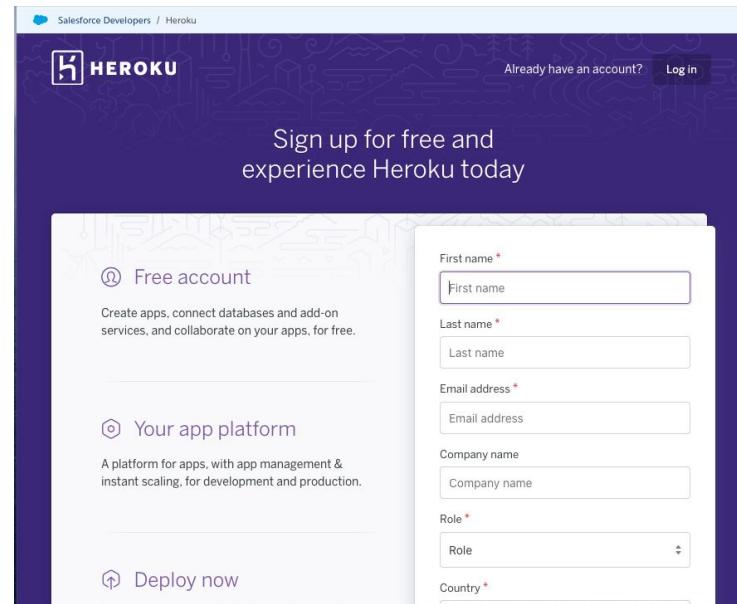
```
git config --global user.email johndoe@example.com
```

Heroku - Signing Up

Heroku is a platform as a service (PaaS) that enables developers to build, run, and operate applications entirely in the cloud.

In order to use heroku, first you need to sign up

<https://www.heroku.com/>



Recommendation - Setup

Download cli and set up heroku - [Link](#)

Create new app

The screenshot shows the Heroku dashboard interface. At the top, there's a header bar with the Salesforce Platform logo, the Heroku logo, and a search bar labeled "Jump to Favorites, Apps, Pipelines, Spaces...". On the right side of the header are icons for settings and user profile. Below the header, the main area has a purple background with a sunburst graphic. A prominent message says "Welcome to Heroku" with a small "h" icon, followed by "Now that your account has been set up, here's how to get started." There's a "Dismiss" button in the bottom right corner of this message area. The main content area contains two sections: "Create a new app" (with a hexagonal icon) and "Create a team" (with a people icon). Both sections have descriptive text and a "Create [action]" button at the bottom.

Salesforce Platform

HEROKU

Jump to Favorites, Apps, Pipelines, Spaces...

Personal

New

Welcome to Heroku

Now that your account has been set up, here's how to get started.

Dismiss

Create a new app

Create your first app and deploy your code to a running dyno.

Create new app

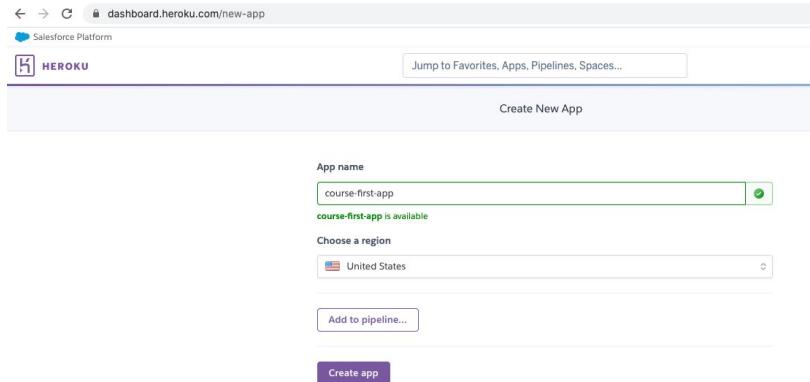
Create a team

Create teams to collaborate on your apps and pipelines.

Create a team

Heroku - Open a New Project

Go to the dashboard screen



The screenshot shows the Heroku dashboard at dashboard.heroku.com/new-app. The page has a header with the Salesforce Platform logo and a 'Jump to Favorites, Apps, Pipelines, Spaces...' button. Below the header is a 'Create New App' button. The main form fields are: 'App name' (containing 'course-first-app'), which is highlighted with a green border; 'course-first-app is available' (in green text); 'Choose a region' (with 'United States' selected); and a 'Create app' button at the bottom.

Create a new app

Enter the required credentials

Installing Heroku CLI

Download the heroku CLI:

<https://devcenter.heroku.com/articles/heroku-cli#download-and-install>

Open a terminal (you may need to close the terminal and open again, if the terminal was open before installing) and enter **heroku login**

cd to your project and enter **git init**

And **heroku git:remote -a course-first-app**

And then **heroku create**

Until now

```
asafamir@Asafs-MBP chap18-heroku-app % heroku login
heroku: Press any key to open up the browser to login or q to exit.
Opening browser to https://cli-auth.herokuapp.com/auth/cli/browser/
jGp6p-AW.
Logging in... done
Logged in as app com
asafamir@Asafs-MBP chap18-heroku-app % git init
Reinitialized existing Git repository in /Use /Desktop/my-node-proj/chap18-heroku-app/.
asafamir@Asafs-MBP chap18-heroku-app % heroku git:remote -a course-first-app
set git remote heroku to https://git.heroku.com/course-first-app.git
asafamir@Asafs-MBP chap18-heroku-app % heroku create
Creating app... done, ⬤ tranquil-plains-25185
https://tranquil-plains-25185.herokuapp.com/ | https://git.heroku.com/tranquil-plains-25185.git
asafamir@Asafs-MBP chap18-heroku-app %
```

Using Heroku

Under you project's dir create a new file called '.gitignore':

```
❖ .gitignore
```

In this file you can enter the files and folders name that you want git to ignore (and also heroku). This way those files and folders won't be uploaded.

npm init

npm init express

We want to ignore the node_modules folder, because it's hugh and takes a lot of space. Heroku has its own way

```
❖ .gitignore
```

to do this, so your .gitignore files should have the following content:

```
1 node_modules
```

After npm init

```
✓ chap18-heroku-app      ●  
◆ .gitignore             U  
{} package.json           U  
ⓘ README.md
```

Using heroku

How will heroku know what modules our application needs?

When we use **npm install <module_name> -s** automatically a dependency is added to the package.json file of our application.

It might look something like that:

```
"dependencies": {  
  "express": "^4.17.2"  
}
```

So before running our application heroku checks what modules we need and downloads them.

Using Heroku

We also need to tell heroku what is the start point of the application.

In the package.json file, under the scripts section, add the following:

```
"start": "<file_name>"
```

So it looks something like:

```
"scripts": {  
  "test": "echo \\"$Error: no test specified\\" && exit 1",  
  "start": "node server.js"  
},
```

Create server.js

```
const express = require('express')

const app = express()

const PORT = process.env.PORT || 5000

app.get('/', (req, res) => {
  res.send('Hello Heroku!')
})

app.listen(PORT, () => {
  console.log(`Example app listening on port ${PORT}`)
})
```

Localhost:5000

← → C ⓘ localhost:5000

Hello Heroku!

Push the code to heroku

Using Heroku

In a terminal in your project dir enter **git add .**

And enter **git commit -m "An informative message"**

Next push the changes you committed from your local repository to the remote repository using

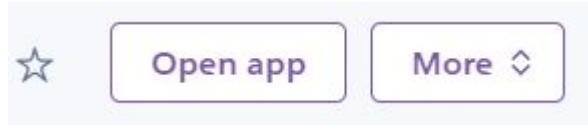
git push heroku master

If git push heroku master does not work => git push heroku main

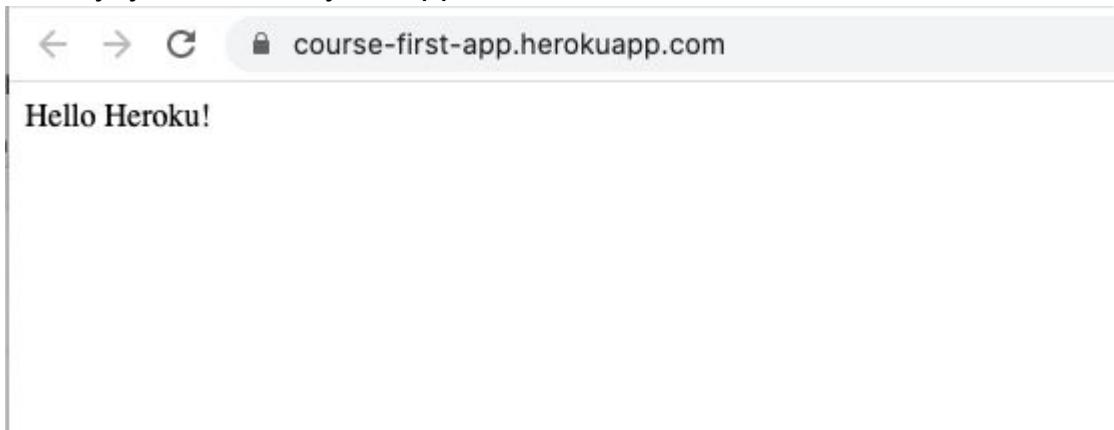
While pushing the project, heroku launches the application in a remote server.

Using Heroku

Navigate to your project in the heroku dashboard, and choose 'open app'



Finally, you can can your application!



Using Heroku

Now every time you want to update this project, you only need to:

1. Add the files to the next commit: **git add .**
2. Commit the changes to your local repository, with an informative message:

git commit -m "Added: new button to home screen"

1. Push the changes from your local repository to the remote repository:

git push heroku master

That's it! (git and heroku have more great features, check them out!)

Recommendation

Recommendation - [Follow this lesson](#)

<https://devcenter.heroku.com/articles/getting-started-with-nodejs#view-logs>



braintop
think.make.play



Chapter 19 - sql

What is sql

A language that enables the handling of tables of a tabular database and the execution of any operation on them (reading, updating, adding). The American Standards Institute has defined sql as a standard language for working with table databases.

What is a table?

Example of a table containing

5 columns - personId, fname, lname, salary, birthday

The table contains 4 rows. Each line represents one person and shows the details about him.

personId	fname	lname	salary	birthday
1	oren	uziel	2000	12/10/1987
2	david	azulay	3000	15/6/1990
3	oren	kabuli	1500	15/4/1984
4	shula	afula	5000	10/3/1995

A row in a table

The table contains 4 rows. Each line represents one person and shows the details about him.

personId	fname	lname	salary	birthday
1	oren	uziel	2000	12/10/1987
2	david	azulay	3000	15/6/1990
3	oren	kabuli	1500	15/4/1984
4	shula	afula	5000	10/3/1995

Sample row in
table



A column in a table

Sample column

The table contains 5 columns

Sample column in table



personId	fname	lname	salary	birthday
1	oren	uziel	2000	12/10/1987
2	david	azulay	3000	15/6/1990
3	oren	kabuli	1500	15/4/1984
4	shula	afula	5000	10//3/1995

Cell in table

Sample cell

The table contains 20 cells

Example of a cell

personId	fname	lname	salary	birthday
1	oren	uziel	2000	12/10/1987
2	david	azulay	3000	15/6/1990
3	oren	kabuli	1500	15/4/1984
4	shula	afula	5000	10//3/1995

Sql command to create a table

```
create Table tblCities(cityId INTEGER PRIMARY KEY, name varchar(50));
```

cityId	name
1	oren
2	david
3	oren
4	shula

Sql command to create a table

create Table if not exists tblCities(cityId INTEGER PRIMARY KEY autoincrement, name varchar(50));



הסבר	פירוש
create Table tblCities	Create a table named tblCities
cityId INTEGER PRIMARY KEY	The first column of the table is a master key. That is, a value that cannot be repeated. The word Integer signifies a number type field
name varchar(50);	The second column in the table indicates that this is a string type field of 50 at most

INSERT to tblCities

Using the INSERT command we enter data into the table

INSERT INTO `tblCities (cityId, name) VALUES (1,'Tel-Aviv');`

Will insert another row in the table.

cityId	name
1	Tel-Aviv

This will work too : **INSERT INTO** `tblCities (name) VALUES ('Bucharest');`

The cityId is autoincrement so its 2.

cityId	name
1	Tel-Aviv
2	'Bucharest'

INSERT to tblCities

If we delete line 2

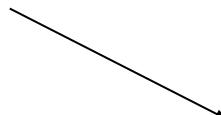
cityId	name
1	Tel-Aviv

And add line : **INSERT INTO tblCities (name) VALUES ('Moscow');**

The cityId is autoincrement so its 3.

The sql system manages the id and even if we deleted the id the system remembers the deleted cityId.

Let sql manage the id



cityId	name
1	Tel-Aviv
3	Moscow

Do it yourself

1. Create the table named tblPerson with the fields -personId, fname, lname, age
2. Insert a row in the table
3. Insert additional rows in the table

Select

Select command structure

select	Defines the columns, fixed them and removable. Required command
from	The names of the tables from which the information will be extracted - mandatory in the ordinance
where	The retrieval conditions that will determine the rows that will be retrieved
group by	Allows grouping of list
Having	Defines logical conditions on the grouped rows
Order By	Displays the order in which the retrieved records will be displayed

Suppose we set up the following table with 4 rows

tblPersons(personId, fname, lname, salary, birthday)

personId	fname	lname	salary	birthday
1	oren	uziel	2000	12/10/1987
2	david	azulay	3000	15/6/1990
3	oren	kabuli	1500	15/4/1984
4	shula	afula	5000	10//3/1995

Retrieving the personId, fname, lname column

Select personId, fname, lname from **tblPersons**

personId	fname	lname
1	oren	uziel
2	david	azulay
3	oren	kabuli
4	shula	afula

Retrieve All Columns: An asterisk indicates all columns

Select * from **tblPersons**

personId	fname	Iname
1	oren	uziel
2	david	azulay
3	oren	kabuli
4	shula	afula

Operators

=	equal
\neq , \neq , !=	Not equal
>	Bigger than
\geq	Bigger or equal
<	Smaller
\leq	Smaller or equal

Get all the records that the salary is higher than 3000

```
Select personId, fname, lname, salary, birthday  
from tblPersons  
Where salary > 3000
```

Salary retrieval higher than 3000

personId	fname	lname	salary	birthday
4	shula	afula	5000	10/3/1995

Pull out all the rows and all the columns that the salary is higher than 3000

Select *

from tblPersons

Where salary > 3000

Pulling out people whose wages are higher than 3000

personId	fname	lname	salary	birthday
4	shula	afula	5000	10/3/1995

Retrieving records in the table of all people born after 14.5

```
select personId, fname, lname, salary, birthday  
from tblPersons  
where birthday > '14.5.1999'
```

personId	fname	lname	salary	birthday
3	oren	kabuli	1500	15/6/1999
4	shula	afula	5000	10/3/1995

Between

```
select personId, fname, lname, salary, birthday  
from tblPersons  
where birthday BETWEEN '1.1.90' AND '31.12.96'
```

personId	fname	lname	salary	birthday
2	david	azulay	3000	15/6/1990
4	shula	afula	5000	10/3/1995

Not Between

```
select personId, fname, lname, salary, birthday  
from tblPersons  
where birthday NOT BETWEEN '1.1.90' AND '31.12.96'
```

personId	fname	lname	salary	birthday
1	oren	uziel	2000	12/10/1987
3	oren	kabuli	1500	15/4/1984

Distinct

Its job is to eliminate duplicate lines

```
select distinct fname  
from tblPerson
```

fname
oren
david
shula

Update

UPDATE TABLE_NAME

SET COULUMN1 = 'VALUE1',COLUMN2 = 'VALUE2'

WHERE SOME_COLUMN='SMOE_VALUE'

```
UPDATE tblPerson  
SET fname='shoosha' lname='gusha'  
WHERE fname='uzi';
```

Go to all lines where the name is uzi and change the first name to shoosha and the last name to gusha.

Update

```
UPDATE tblPerson  
SET fname='shoosha' lname='gusha'  
WHERE fname='uzi';
```

Go to all lines where the name is uzi and change the first name to shoosha and the last name to gusha.

Delete

```
DELETE FROM TABLE NAME  
WHERE SOME_COLUMN=SOME_VALUE
```

```
DELETE FROM tblPerson  
WHERE salary>2000  
Will delete all lines where the salary is higher than 2000.
```

Delete

DELETE FROM tblPersons

WHERE salary>2000

.Will delete all lines where the salary is higher than 2000

Hands On

1. Create the table named tblProducts with the productId, name, description, price, category, quantity fields

tblProducts(productId,name,description,price,category,quantity)

```
create Table tblProducts(productId INTEGER PRIMARY KEY, name varchar(50),description  
varchar(200), price INTEGER,category varchar(50),quantity INTEGER);
```

2. Insert a row in the table

```
INSERT INTO tblProducts (productId,name,description,price,category,quantity) VALUES (1,'Ball','Red  
ball',10,'Sport',123);
```

3. Insert additional rows in the table

Hands On

1. After setting up the table in mysql. Build the following queries
2. tblProducts (productId, name, description, price, category, dateProducuon, quantity)
3. Insert 4 products into the table using the insert command
4. See all products in the table.
5. View all categories
6. Show only the productId, price fields.
7. See all products whose category is 8
8. Show all products whose code is 10 or 15 Price between 100 200
9. See all product names that contain the word "game"
10. View all products sorted by year of issue.
11. Delete all products that have a quantity equal to 0
12. Upload all products for NIS 100 using an update query
13. Add a city table. Show product name, price, city name (help in inner join)

More rules for sql

(List of values) IN	Equal to one of the entries in the list
not in	Not equal to any of the entries in the list
LIKE string pattern	Compared to a string pattern
IS NULL	The value is null
IS NOT NULL	The value is not null

Example of - like

```
select personId , fname  
from tblPerson  
where fname like 'a%'
```

Displays all the people whose names begin with the letter a

```
select personId , fname  
from tblPerson  
where fname like '%a'
```

Displays all the people whose names end in the letter a

Example of null

A NULL value in the field means that the value is unknown, missing. List all people who are missing a last name in the table

```
select personId , fname  
from tblPerson  
where fname  is null
```

Example of not null

Displays all the people who do not have a last name in the table

```
select personId , fname  
from tblPerson  
where fname is not null
```

An example of a complex condition

An example of a complex condition

Show all the people who call them Uzi and their salary is higher than 2000

```
select personId , fname  
from tblPerson  
where salary > 2000 AND fname = 'uzi'
```

An example of a complex condition

An example of a complex condition

Show all people who call themselves Uzi or whose salary is higher than 2000

```
select personId , fname  
from tblPerson  
where salary>2000 OR fname = 'uzi'
```

Displays rows in a certain order

Will show all employees with a salary higher than 2000 in **descending** order of salary:

```
select lname , fname  
from tblPerson  
where salary>2000  
ORDER BY salary desc
```

Normalizing tables

Suppose you have the following table:

We would like to normalize the table.

That is, change the cityname field

To cityId for two reasons:

- 1.) Memory saving - we can keep a city number in memory
- 2.) Prevention of future mistakes

tblPersons

personId
name
age
cityname

personId	Name	Age	Cityname
1	Lewis	30	Bucharest
2	Adam	32	Moscow
3	Or	26	Bucharest

Normalizing tables - Room for error

Suppose we want all the people who live in Bucharest but in personId number 3 a typo was made in the Cityname, That is, we will not get it in the result

personId	Name	Age	Cityname
1	Lewis	30	Bucharest
2	Adam	32	Moscow
3	Or	26	Buchrest 

Normalizing tables

tblPersons

personId
name
age
cityname

tblCities

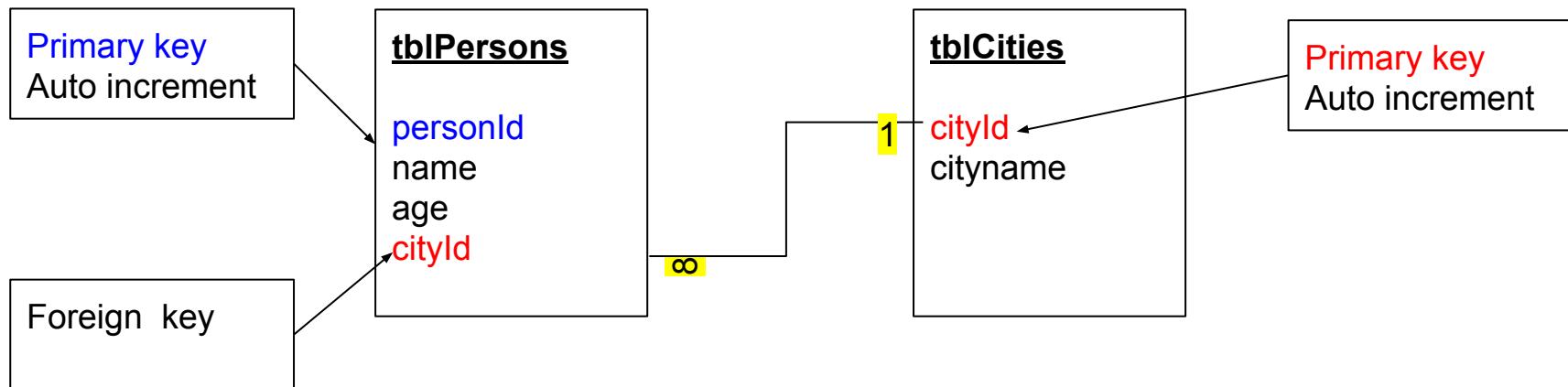
cityId
cityname

and We will change the cityname field to cityId
create a city table

personId	Name	Age	cityId
1	Lewis	30	1
2	Adam	32	2
3	Or	26	1

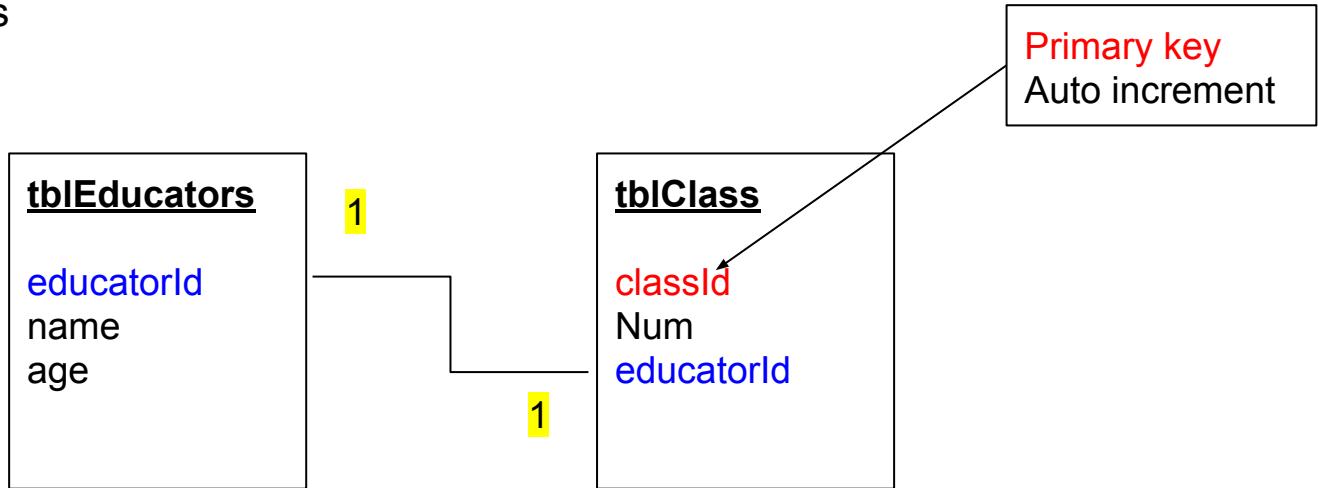
Relationship - one to many

Single connection to many: Every person lives in one city, but in one city many people live
Foreign key - Main key in another table



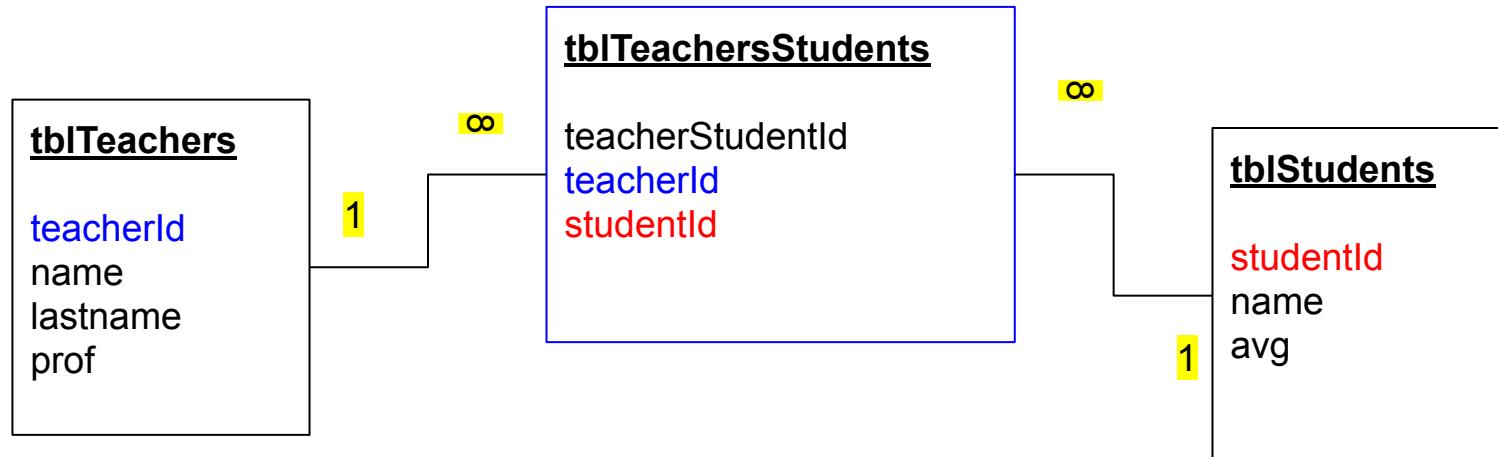
Relationship - one to one

Individual-to-individual relationship: educator and class. Each class has one educator, and each educator has one class



Relationship - many to many

Many to many: teacher and students. Every teacher has a lot of students, and every student has a lot of teachers



Products - many to many

tblStore(storeId, name, address)

tblProducts(productId, name, price)

tblProductStore(ProductStoreId,storeId, productId)

	storeId	name	address
1	a	telaviv	
2	b	hadera	

ProductStoreId, storeId, productId

1	1	1
2	1	2
3	2	1

	productId	name	price
1	ball	5	
2	TS	10	

Inner join

Combines records from two tables whenever there are matching values in a common field.

Syntax :

```
FROM table1 INNER JOIN table2 ON table1.field1 composer table2.field2
```

The INNER JOIN operation has these parts:

Part	Description
<i>table1, table2</i>	The names of the tables from which records are combined.
<i>field1, field2</i>	The names of the fields that are joined. If they are not numeric, the fields must be of the same data type and contain the same kind of data, but they do not have to have the same name.
<i>comopr</i>	Any relational comparison operator: "=," "<," ">," "<=," ">=," or "<>."

Inner join

You can use an INNER JOIN operation in any FROM clause. This is the most common type of join. Inner joins combine records from two tables whenever there are matching values in a field common to both tables.

You can use INNER JOIN with the Departments and Employees tables to select all the employees in each department. In contrast, to select all departments (even if some have no employees assigned to them) or all employees (even if some are not assigned to a department), you can use a LEFT JOIN or RIGHT JOIN operation to create an outer join.

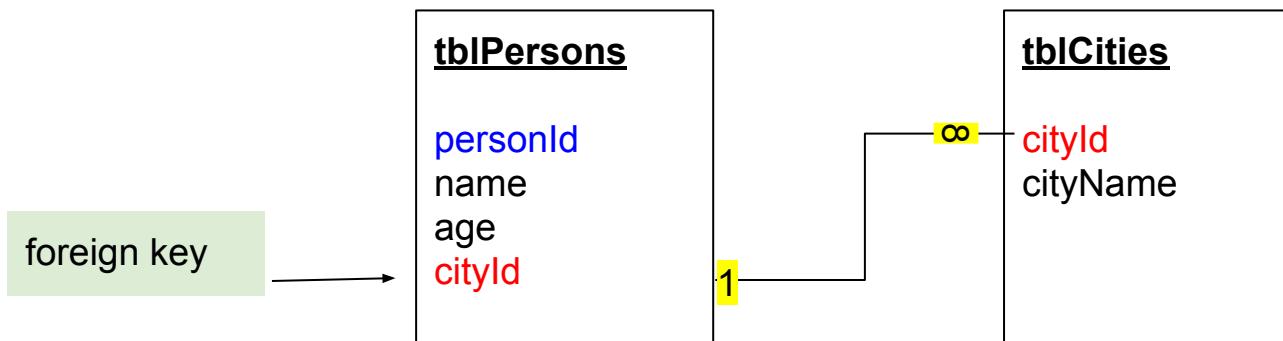
If you try to join fields containing Memo or OLE Object data, an error occurs.

You can join any two numeric fields of like types. For example, you can join on AutoNumber and Long fields because they are like types. However, you cannot join Single and Double types of fields.

Inner join

```
select tblPerson.name,tblPerson.age,tblCity.cityName  
FROM tblPersons  
INNER JOIN tblCities  
on tblCities.cityId = tblPerson.cityId
```

Will display the columns first name,age cityName



Do it yourself

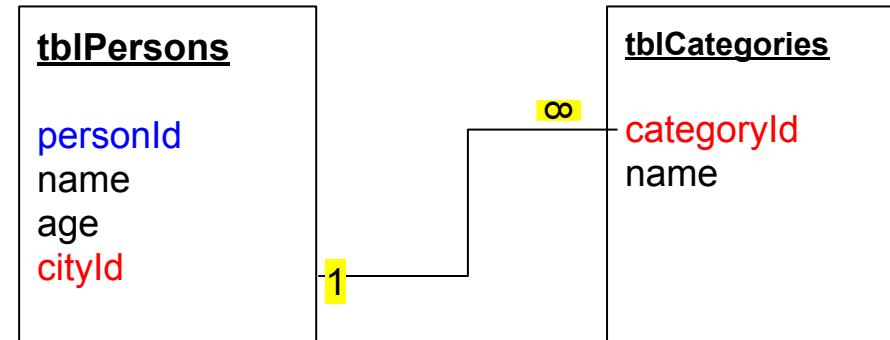
Set up 2 tables

tblProducts(productId, name, description, price, categoryId, dateProduction, quantity)

tblCategories(categoryId, name)

Add 3 line to tblProduct

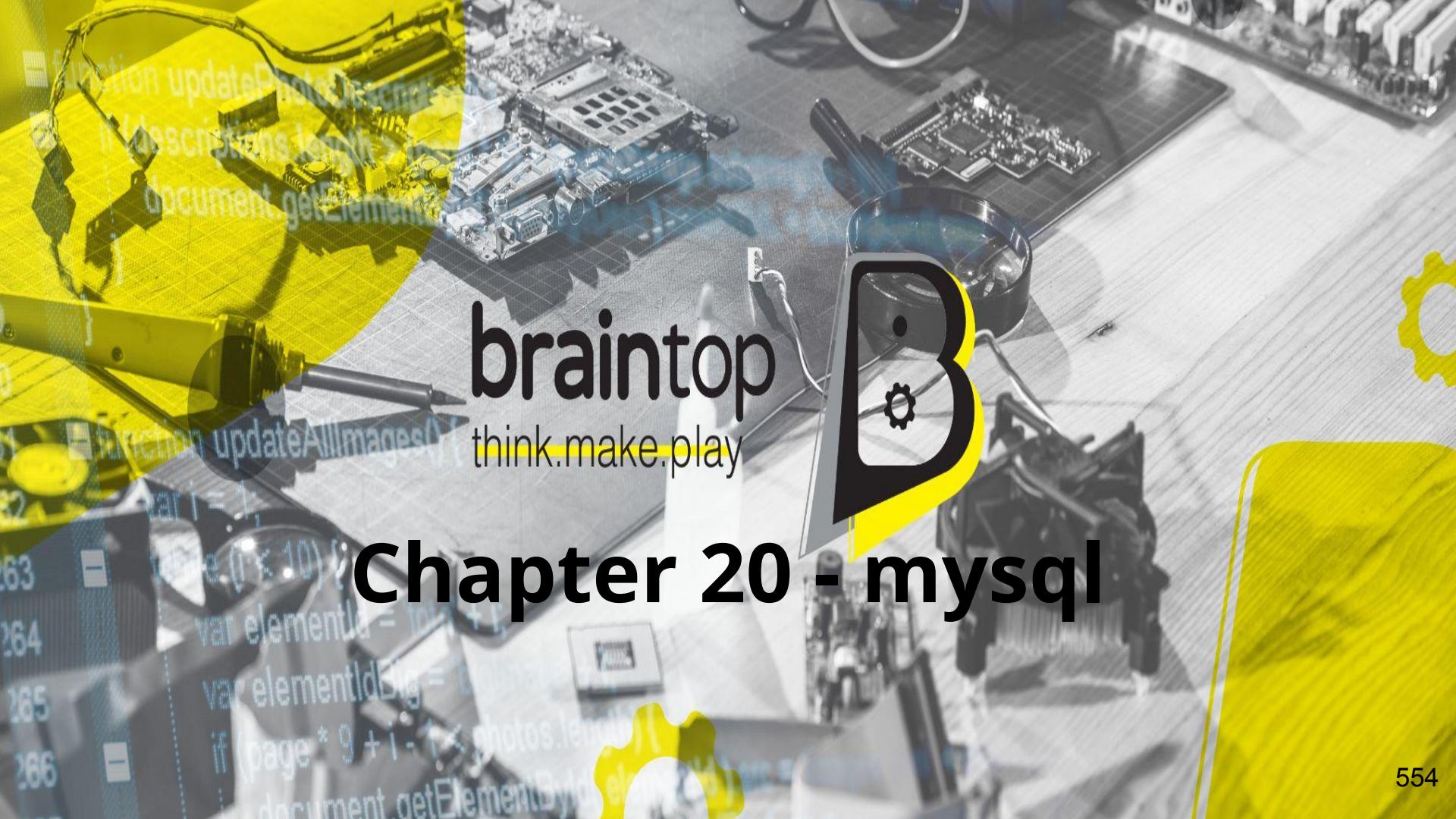
1	ball	blue ball	50	1	12/12/1978	5
2	ball	t shirt	60	2	12/12/1978	2
3	ball	red ball	100	1	12/10/1980	4



Add 2 categories

1	toys
2	clothing

Retrieve the product name, description, price and category from the tables (join command)



braintop
think.make.play

Chapter 20 - mysql

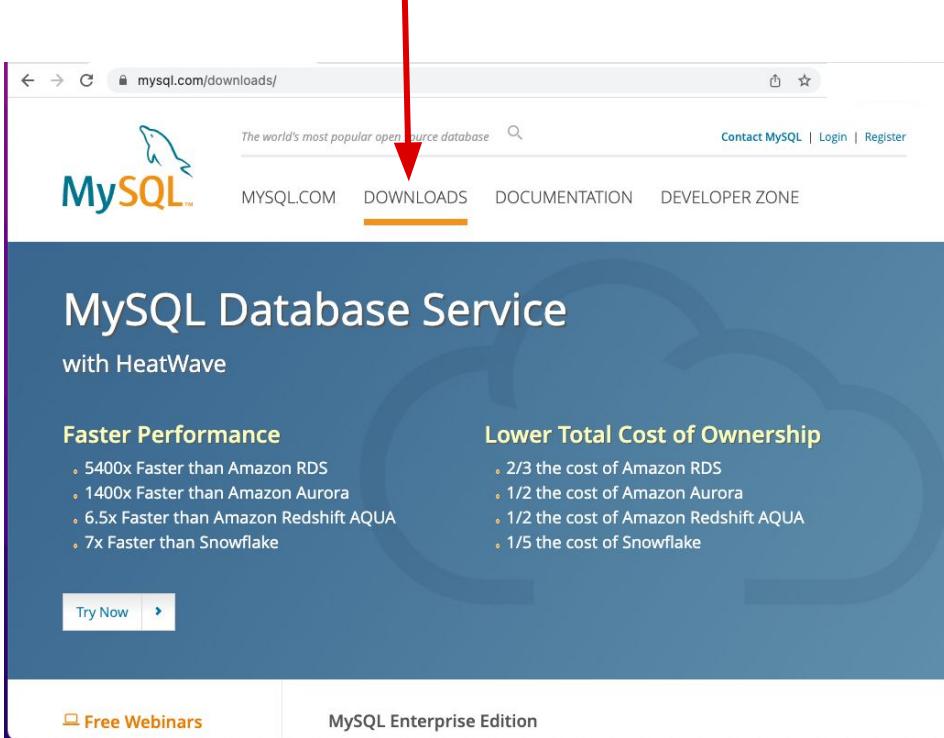
Mysql database

MySQL is a relational database management system based on SQL - Structured Query Language. ... The most common use for mySQL however, is for the purpose of a web database. It can be used to store anything from a single record of information to an entire inventory of available products for an online store.

Mysql.com

Download mysql

Go to mysql.com



The screenshot shows the MySQL website at mysql.com/downloads/. A red arrow points to the "DOWNLOADS" menu item, which is underlined, indicating it is the active section. The page features the MySQL logo and a banner for the MySQL Database Service with HeatWave. Below the banner, there are two columns: "Faster Performance" and "Lower Total Cost of Ownership", each with a list of benefits. At the bottom, there are links for "Free Webinars" and "MySQL Enterprise Edition".

The MySQL Database Service with HeatWave

Faster Performance

- 5400x Faster than Amazon RDS
- 1400x Faster than Amazon Aurora
- 6.5x Faster than Amazon Redshift AQUA
- 7x Faster than Snowflake

Lower Total Cost of Ownership

- 2/3 the cost of Amazon RDS
- 1/2 the cost of Amazon Aurora
- 1/2 the cost of Amazon Redshift AQUA
- 1/5 the cost of Snowflake

Try Now >

Free Webinars MySQL Enterprise Edition

Download community server

Go to [mysql.com](https://www.mysql.com)

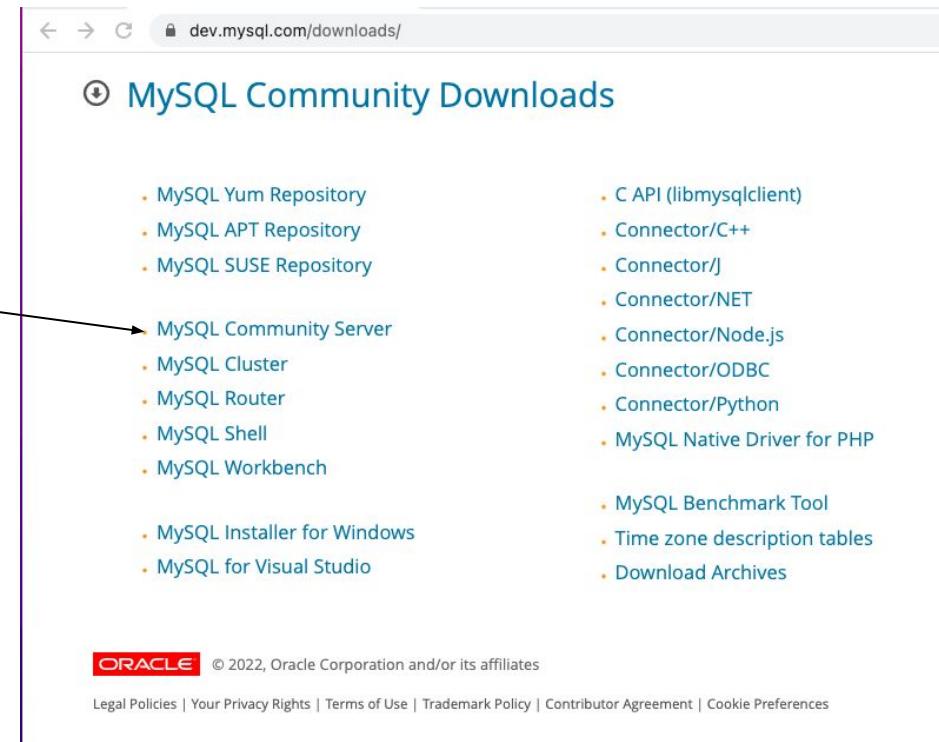
The screenshot shows the MySQL website homepage. At the top, there's a navigation bar with links for 'MySQL.COM', 'DOWNLOADS' (which is highlighted in orange), 'DOCUMENTATION', and 'DEVELOPER ZONE'. There's also a search bar and social media links for Facebook, Twitter, LinkedIn, and YouTube. The main content area features a large blue background with a white cloud icon. The title 'MySQL Database Service' is at the top, followed by 'with HeatWave'. Below this, there are two sections: 'Faster Performance' and 'Lower Total Cost of Ownership', each with a bulleted list of benefits. At the bottom of this section is a 'Try Now' button. On the left side of the page, there are several sidebar sections: 'Free Webinars' (listing events like 'Guide to NoSQL and SQL with Grade MySQL'), 'MySQL Enterprise Edition' (describing it as the most comprehensive set of advanced features), 'MySQL Cluster CGE' (describing it as a real-time open source transactional database), 'Contact Sales' (providing phone numbers for USA, Canada, Germany, France, Italy, and UK), and 'MySQL Community (GPL) Downloads' (with an arrow pointing from the text above to this link).

Click on: [MySQL Community \(GPL\) Downloads »](#)

Download community server

Go to mysql.com

Download : MySQL Community Server



The screenshot shows a web browser window with the URL dev.mysql.com/downloads/ in the address bar. The page title is "MySQL Community Downloads". A large list of download links is displayed in two columns. An arrow points from the text "Download : MySQL Community Server" to the "MySQL Community Server" link in the list.

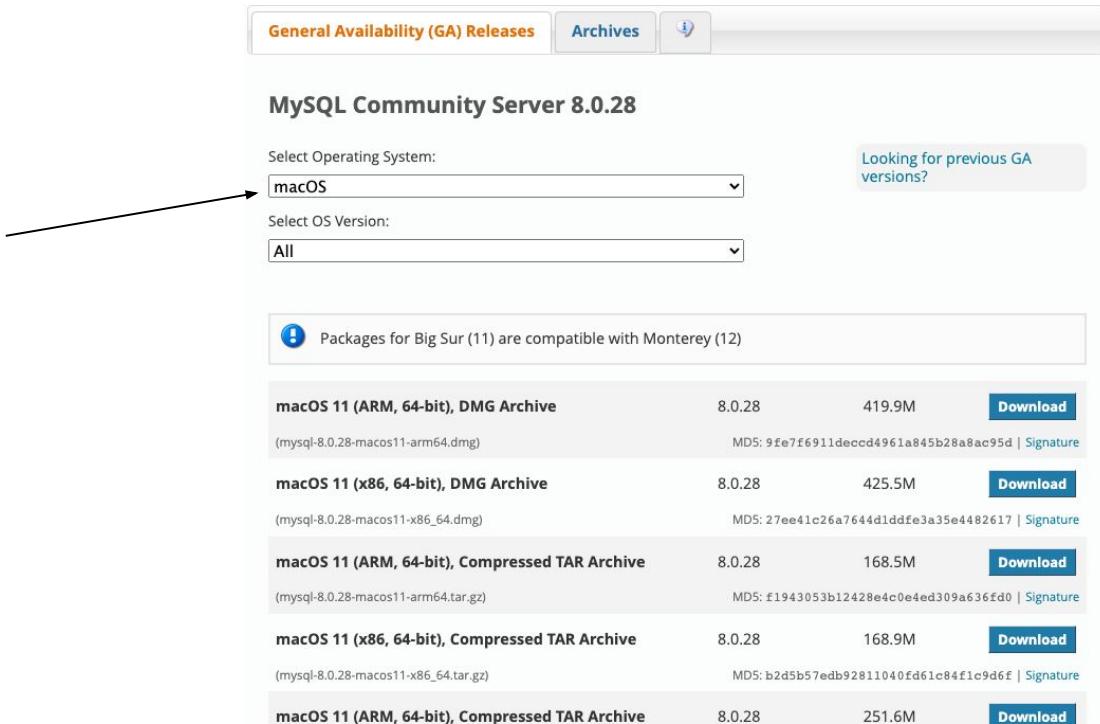
- MySQL Yum Repository
- MySQL APT Repository
- MySQL SUSE Repository
- MySQL Community Server
- MySQL Cluster
- MySQL Router
- MySQL Shell
- MySQL Workbench
- MySQL Installer for Windows
- MySQL for Visual Studio
- C API (libmysqlclient)
- Connector/C++
- Connector/J
- Connector/.NET
- Connector/Node.js
- Connector/ODBC
- Connector/Python
- MySQL Native Driver for PHP
- MySQL Benchmark Tool
- Time zone description tables
- Download Archives

ORACLE © 2022, Oracle Corporation and/or its affiliates
Legal Policies | Your Privacy Rights | Terms of Use | Trademark Policy | Contributor Agreement | Cookie Preferences

Download mysql

Go to mysql.com

Select your operating system and download :



General Availability (GA) Releases Archives 🔍

MySQL Community Server 8.0.28

Select Operating System:

macOS

Select OS Version:

All

Looking for previous GA versions?

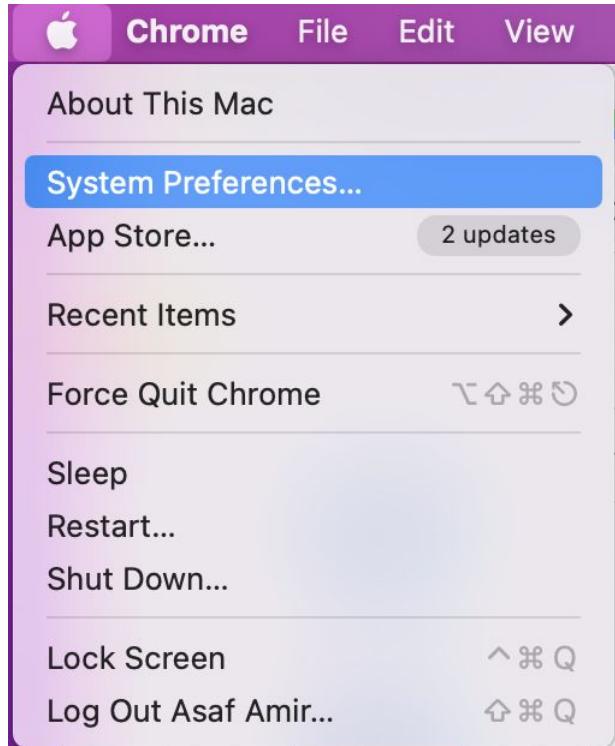
Packages for Big Sur (11) are compatible with Monterey (12)

macOS 11 (ARM, 64-bit), DMG Archive	8.0.28	419.9M	Download
(mysql-8.0.28-macos11-arm64.dmg)			MD5: 9fe7f6911dec4d4961a845b28a8ac95d Signature
macOS 11 (x86, 64-bit), DMG Archive	8.0.28	425.5M	Download
(mysql-8.0.28-macos11-x86_64.dmg)			MD5: 27ee41c26a7644d1ddfe3a35e4482617 Signature
macOS 11 (ARM, 64-bit), Compressed TAR Archive	8.0.28	168.5M	Download
(mysql-8.0.28-macos11-arm64.tar.gz)			MD5: f1943053b12428e4c0e4ed309a636fd0 Signature
macOS 11 (x86, 64-bit), Compressed TAR Archive	8.0.28	168.9M	Download
(mysql-8.0.28-macos11-x86_64.tar.gz)			MD5: b2d5b57edb92811040fd61c84f1c9d6f Signature
macOS 11 (ARM, 64-bit), Compressed TAR Archive	8.0.28	251.6M	Download

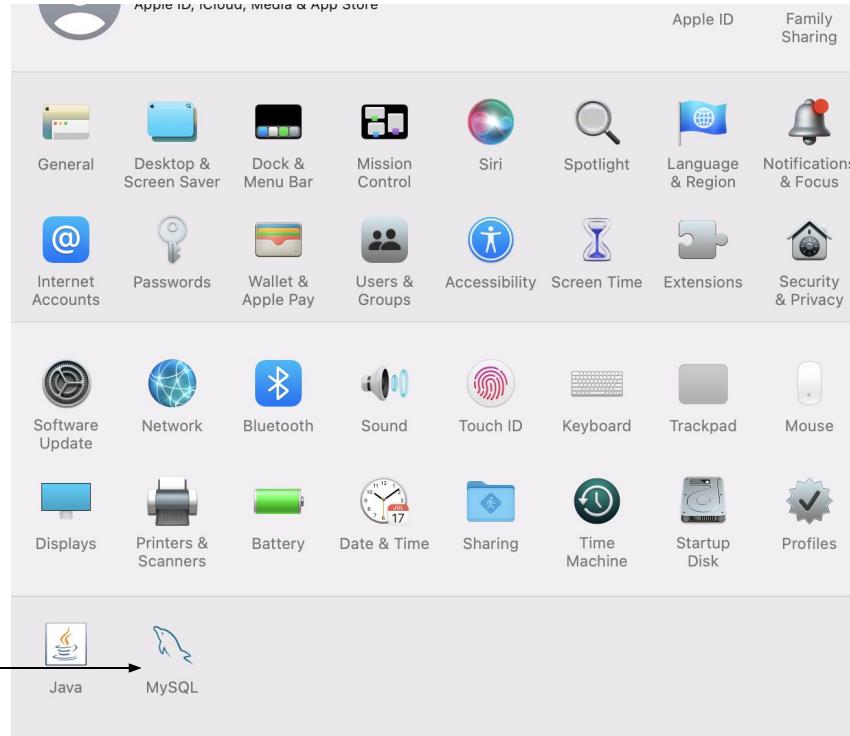
Start the server

From mac :

Go to

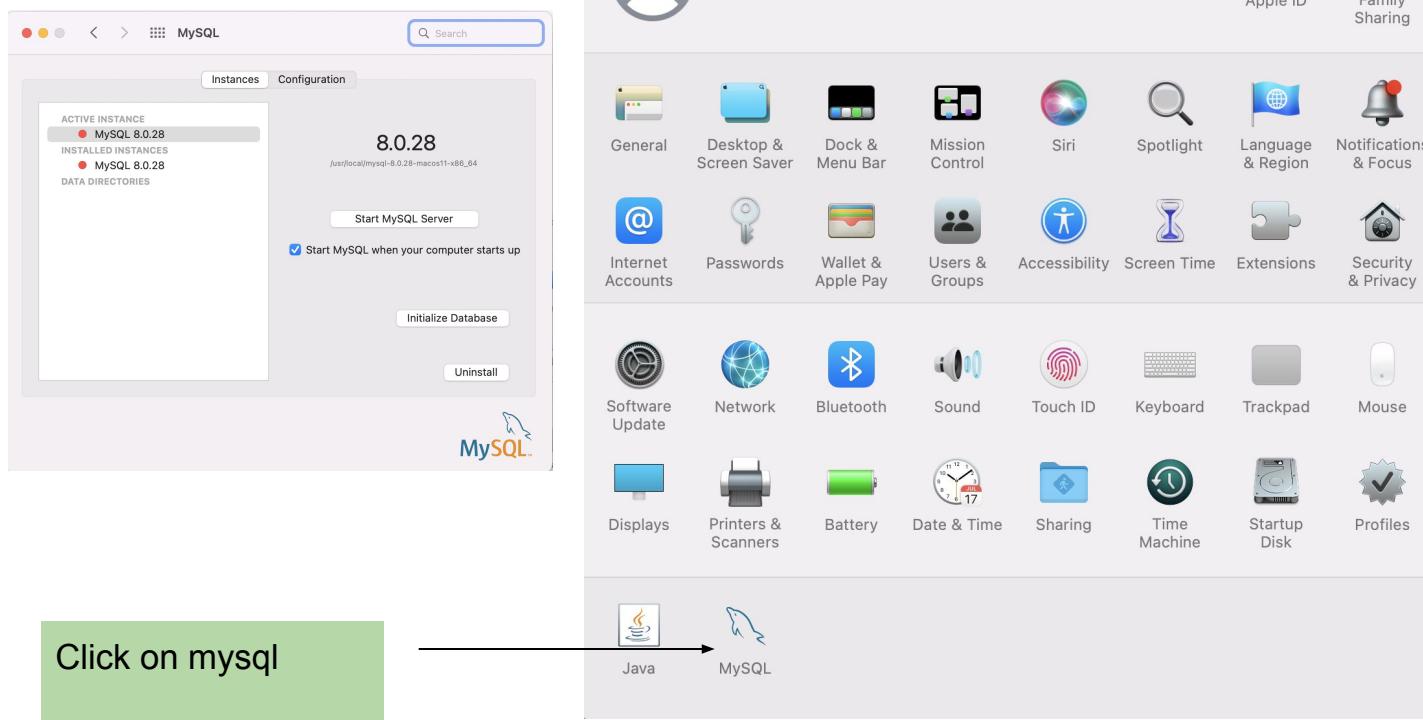


Start the server



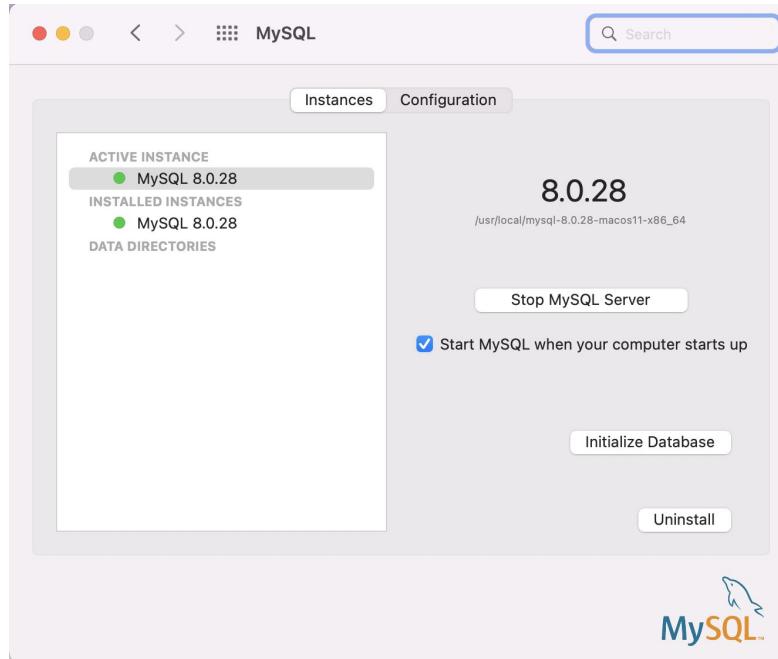
Click on mysql

Start the server



After installing start server

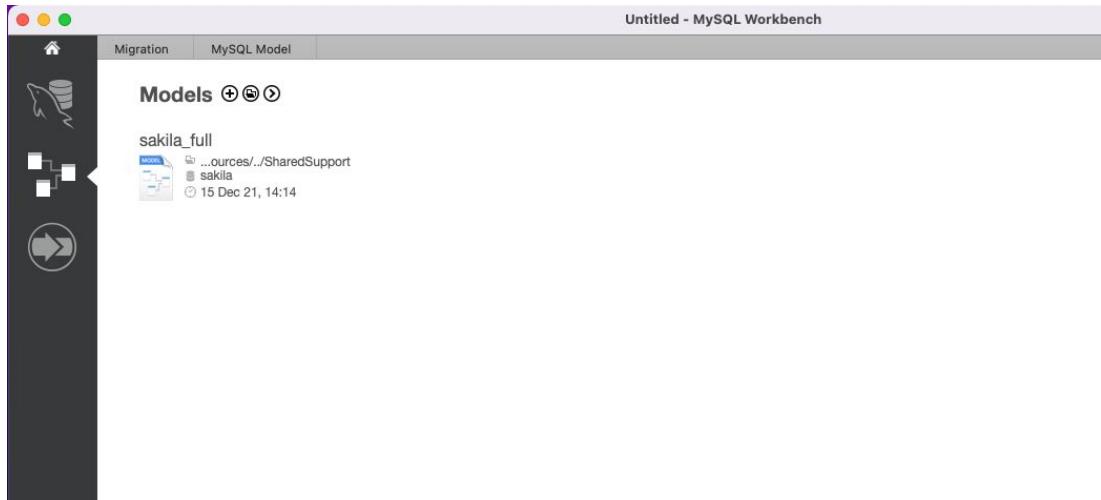
Go to mysql.com



Mysql - download workbench

<https://dev.mysql.com/downloads/workbench/>

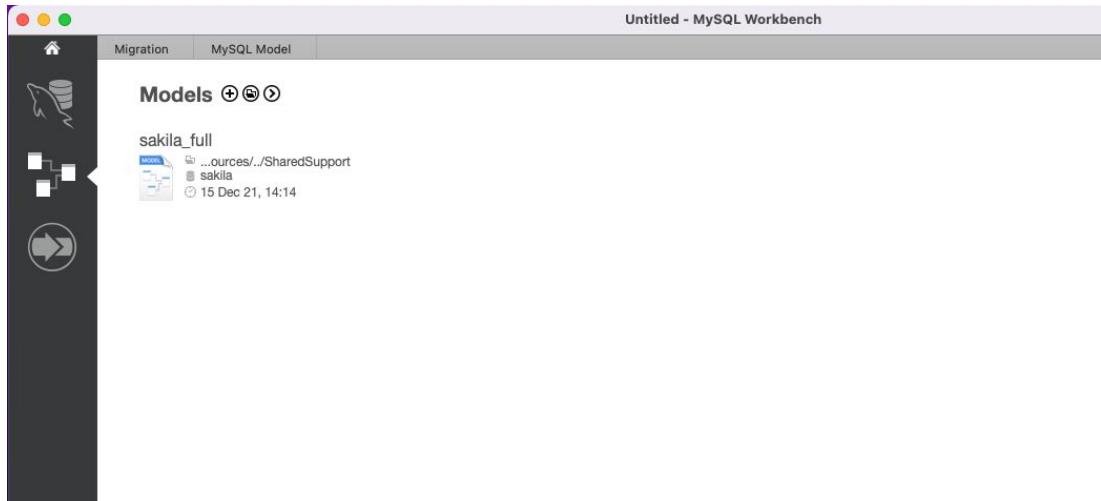
After you install open it :



Mysql

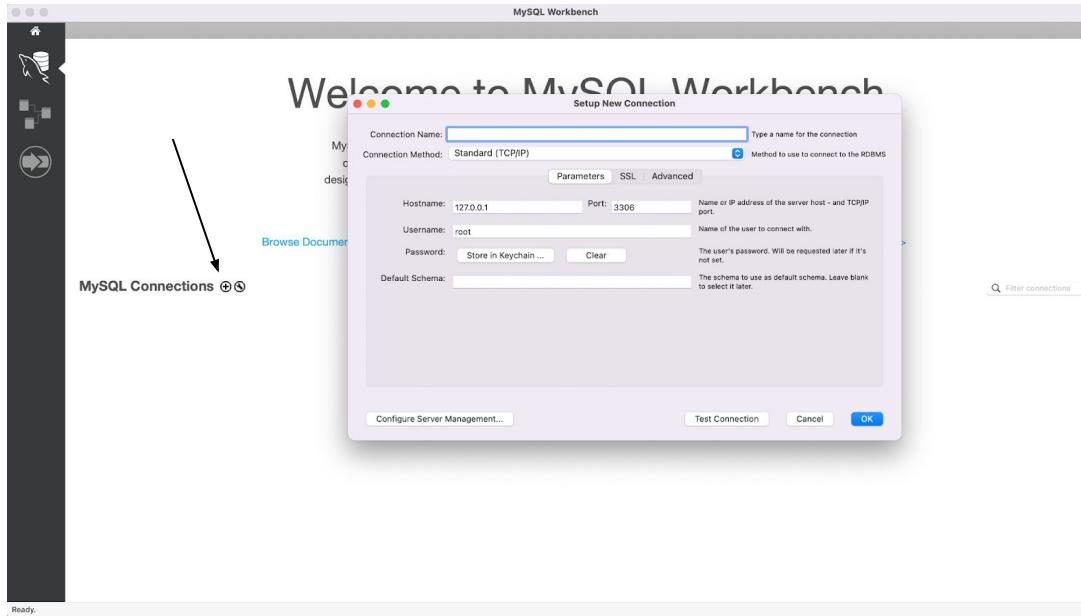
<https://dev.mysql.com/downloads/workbench/>

After you install open it :



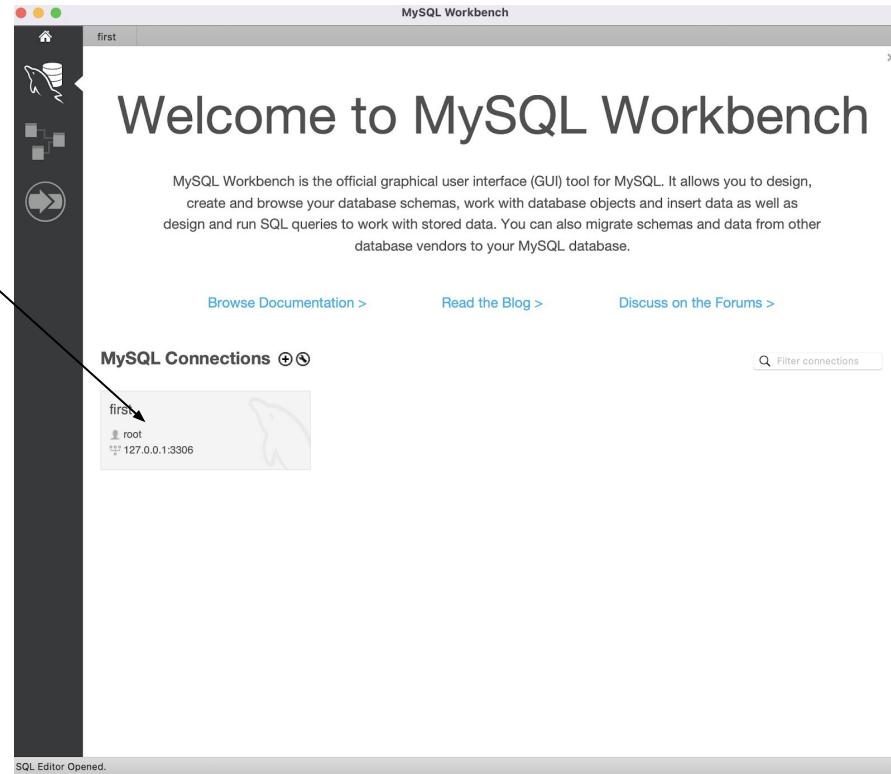
Add module

Click on + icon and add module



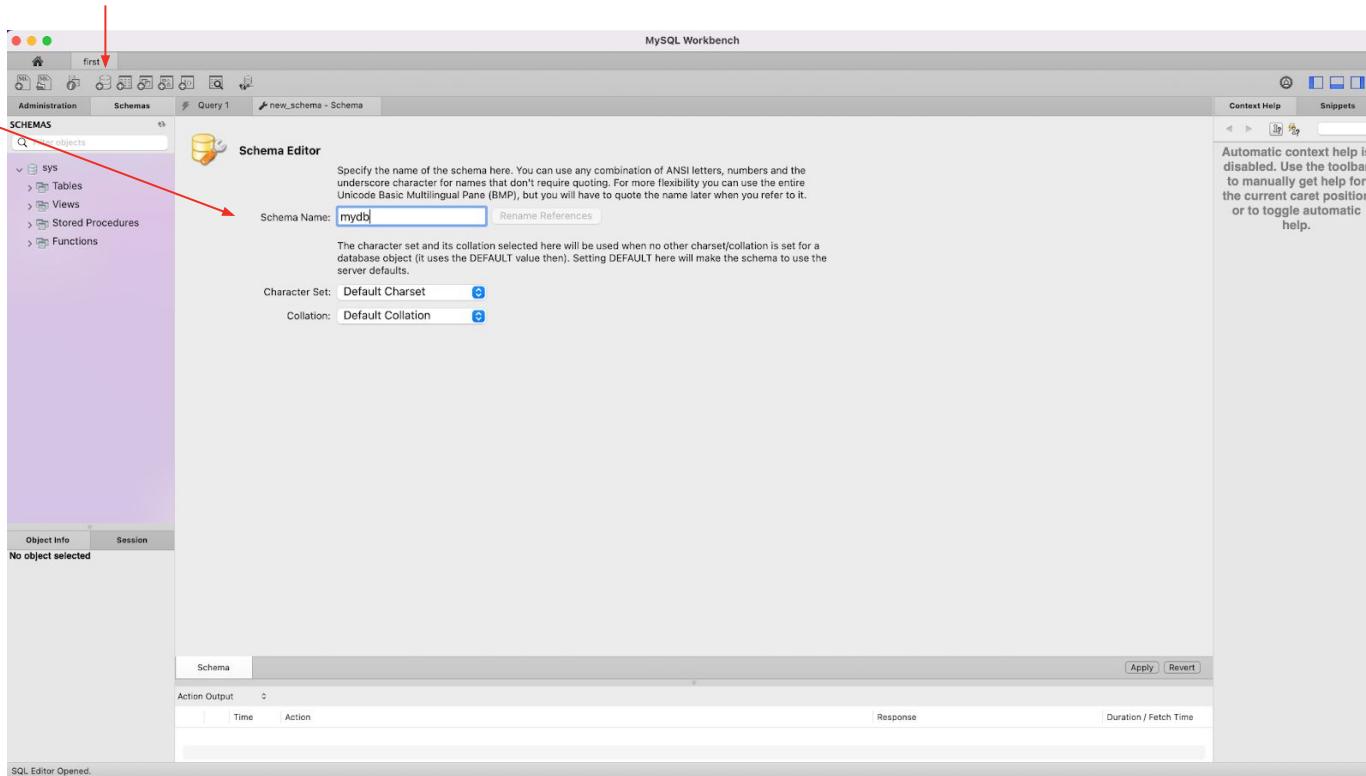
Add connection

Click the connection

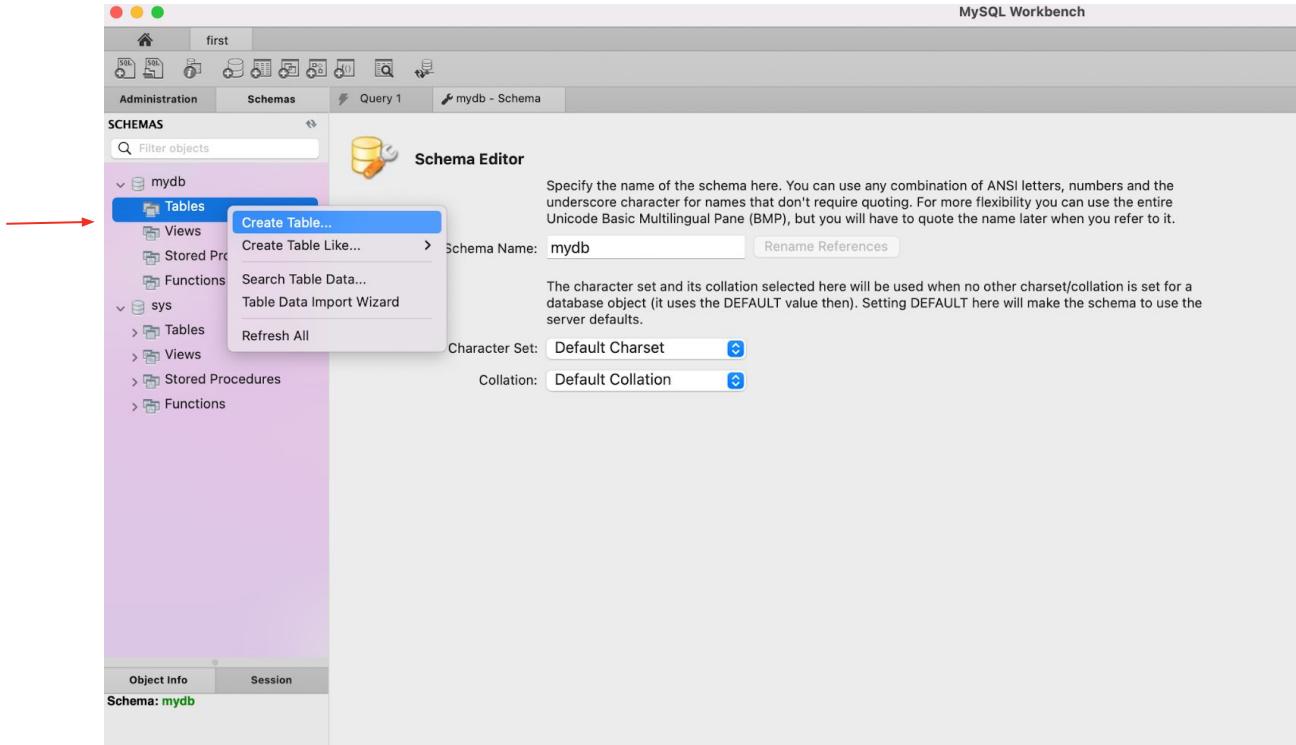


Create db

Create db



Create table



Add fields

MySQL Workbench

first

Administration Schemas

Filter objects

mydb

- Tables
- Views
- Stored Procedures
- Functions

sys

- Tables
- Views
- Stored Procedures
- Functions

Name: `tblPerson`

Column	Datatype	PK	NN	UQ	B...	UN	ZF	AI	G	Default / Expression
<code>personId</code>	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
<code>firsname</code>	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
<code>lastname</code>	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
<code>salary</code>	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

<click to edit>

INT
VARCHAR()
DECIMAL()
DATETIME
BLOB

Column details 'salary'

Column Name: salary

Charset/Collation: Default Charset Default Collation

Comments:

Object Info Session

Schema: mydb

570

Add fields

MySQL Workbench

Query 1 mydb - Schema

Name: tbPerson

Column Datatype

- personId INT
- firsname VARCHAR(45)
- lastname VARCHAR(45)
- salary VARCHAR(45)

<click to edit>

Column details 'salary'

Column Name: salary

Charset/Collation: Default Charset

Comments:

Review SQL Script Apply SQL Script

Please review the following SQL script that will be applied to the database.
Note that once applied, these statements may not be revertible without losing some of the data.
You can also manually change the SQL statements before execution.

Online DDL

Algorithm: Default Lock Type: Default

```
1 CREATE TABLE `mydb`.`tblPerson` (
2     `personId` INT NOT NULL AUTO_INCREMENT,
3     `firsname` VARCHAR(45) NULL,
4     `lastname` VARCHAR(45) NULL,
5     `salary` VARCHAR(45) NULL,
6     PRIMARY KEY (`personId`));
7
```

100% 1:1

Go Back Apply

Nodejs & Mysql

Create connection

```
const express = require('express')
var mysql = require('mysql');
const app = express()
const PORT = process.env.PORT || 3000
app.get('/', (req, res) => {
  res.send('Hello Heroku!')
})
var con = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "asaf7452702"
});

con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
})

app.listen(PORT, () => {
  console.log(`Example app listening on port ${PORT}`)
})
```

```
chap18-heroku-app % node server
Example app listening on port 3000
Connected!
```

Create database

We created db on mysql, but you can do it from nodejs code too.

```
con.connect(function(err) {  
  if (err) throw err;  
  
  console.log("Connected!");  
  
  con.query("CREATE DATABASE mydb", function (err, result) {  
    if (err) throw err;  
  
    console.log("Database created");  
  });  
});
```

Create table

We created `tblPerson` on mysql, but you can do it from nodejs code too.

```
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
  createTablePerson()  
});  
  
function createTablePerson(){  
  let sql = `CREATE TABLE if not exists mydb.tblPersons(personId INT NOT  
NULL AUTO_INCREMENT,  
  firstname VARCHAR(45) NULL, lastname VARCHAR(45) NULL,  
  salary INT NULL, PRIMARY KEY (personId))`;  
  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log("Table created");  
  });  
}
```

;-MBP chap18-heroku-app % node server
Example app listening on port 3000
Connected!
Table created

Insert record

```
app.post('/api/v1/persons', function(req, res, next) {  
  try{  
    let sql = `INSERT INTO mydb.tblPersons (firstname, lastname, salary) VALUES  
('${req.body.firstname}', '${req.body.lastname}', '${req.body.salary}')`;  
    con.query(sql, function (err, result) {  
      if (err) throw err;  
      res.status(201).json({  
        status:"success",  
        data:result  
      })  
    });  
  }  
  catch(err){  
    res.status(400).json({  
      status:"fail",  
      message:"error:😱" + err  
    })  
  }  
}) ;
```

The screenshot shows the Postman application interface. At the top, it says "localhost:3000/api/v1/persons". Below that, there's a "POST" button and the URL again. Under "Body", the "JSON" tab is selected, and the body content is:

```
1 ... "firstname": "David",  
2 ... "lastname": "Debkin",  
3 ... "salary": 200  
4  
5
```

At the bottom, under "Body", the "Pretty" tab is selected, showing the JSON response:

```
1 {  
2   "status": "success",  
3   "data": {  
4     "fieldCount": 0,  
5     "affectedRows": 1,  
6     "insertId": 2,  
7     "serverStatus": 2,  
8     "warningCount": 0,  
9     "message": "",  
10    "protocol41": true,  
11    "changedRows": 0  
12  }  
13}
```

Update record

```
app.patch('/api/v1/persons', function(req, res, next) {
  console.log(req.body)
  try{
    var sql = `UPDATE mydb.tblPersons SET firstname = '${req.body.firstname}' WHERE personId
= ${req.body.personId}`;
    console.log(sql)
    con.query(sql, function (err, result) {
      if (err) throw err;
      res.status(201).json({
        status:"success",
        data:result
      })
    });
  }
  catch(err){
    res.status(400).json({
      status:"fail",
      message:"error: "+err
    })
  }
});
```

Before Update record

The screenshot shows the MySQL Workbench interface. On the left, the 'Schemas' tree shows a database named 'mydb' containing a table named 'tblPersons'. A context menu is open over the 'tblPersons' table, with the 'Select Rows - Limit 1000' option highlighted. The main area displays the results of the query `SELECT * FROM mydb.tblPersons;`. The result grid shows two rows of data:

personid	firstname	lastname	salary
1	Mikel	Lewis	100
2	David	Debkin	200
NULL	NULL	NULL	NULL

Select all records

```
app.get('/api/v1/persons', (req, res) => {
  try{
    let sql = `SELECT * FROM mydb.tblPersons;`;
    con.query(sql, function (err, result) {
      if (err) throw err;
      res.status(201).json({
        status:"success",
        data:result
      })
    });
  }
  catch(err){
    res.status(400).json({
      status:"fail",
      message:"error:😱" + err
    })
  }
});
```

localhost:3000/api/v1/persons/

GET localhost:3000/api/v1/persons/

Params Authorization Headers (7) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
1 { "status": "success",
2   "data": [
3     {
4       "personId": 1,
5       "firstname": "Ron",
6       "lastname": "Lewis",
7       "salary": 100
8     },
9     {
10       "personId": 2,
11       "firstname": "David",
12       "lastname": "Debkin",
13       "salary": 200
14     }
15   ]
16 }
17 }
```

Delete

```
app.delete('/api/v1/persons', function(req, res, next) {
  console.log(req.body)
  try{
    var sql = `delete from mydb.tblPersons WHERE personId = ${req.body.personId}`;
    con.query(sql, function (err, result) {
      if (err) throw err;
      res.status(201).json({
        status:"success",
        data:result
      })
    });
  }
  catch(err){
    res.status(400).json({
      status:"fail",
      message:"error: 😱 " + err
    })
  }
});
```

localhost:3000/api/v1/persons/

DELETE localhost:3000/api/v1/persons/

Params Authorization Headers (9) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

1 "personId":1

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
1 "status": "success",
2 "data": {
3   "fieldCount": 0,
4   "affectedRows": 1,
5   "insertId": 0,
6   "serverStatus": 2,
7   "warningCount": 0,
8   "message": "",
9   "protocol41": true,
10  "changedRows": 0
11 }
```

Delete

```
app.delete('/api/v1/persons', function(req, res, next) {
  console.log(req.body)
  try{
    var sql = `delete from mydb.tblPersons WHERE personId = ${req.body.personId}`;
    con.query(sql, function (err, result) {
      if (err) throw err;
      res.status(201).json({
        status:"success",
        data:result
      })
    });
  }
  catch(err){
    res.status(400).json({
      status:"fail",
      message:"error: 😱 " + err
    })
  }
});
```

The screenshot shows a Postman request to `localhost:3000/api/v1/persons` using the DELETE method. The Body tab contains the following JSON:

```
1
2   "personId":1
3
```

The response body is displayed in the JSON tab:

```
1
2   "status": "success",
3   "data": {
4     "fieldCount": 0,
5     "affectedRows": 1,
6     "insertId": 0,
7     "serverStatus": 2,
8     "warningCount": 0,
9     "message": "",
10    "protocol41": true,
11    "changedRows": 0
12  }
13
```

More examples

```
con.query("SELECT * FROM mydb.tblPersons ORDER BY firstname", function (err, result) {})
```

```
con.query("DROP TABLE IF EXISTS mydb.tblPersons", function (err, result) {})
```

```
con.query("SELECT * FROM mydb.tblPersons LIMIT 10\"", function (err, result) {})
```

Join examples

```
con.query("SELECT mydb.tblPersons.firstname AS firstname, mydb.tblPersons.lastname AS lastname  
FROM mydb.tblPersons JOIN cities ON mydb.tblPersons.cityId = mydb.cities.cityId", function (err,  
result) {})
```

Do it yourself

Enter the following table in mysql,

tblProduct (productId, name, description, price, category, quantity)

Write the following queries on nodeJs:

1. Enter 5 products in the table by adding a command
2. Show all products in the table.
3. View all categories
4. Show only product ID, price.
5. Show all products in their category 8
6. Show all products whose code is 10 or 15 and priced between 100 200
7. View all products sorted in order by name.
8. Delete all products whose quantity is equal to 0