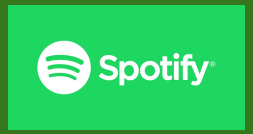
Node workshop

Creating an API for your next web application!



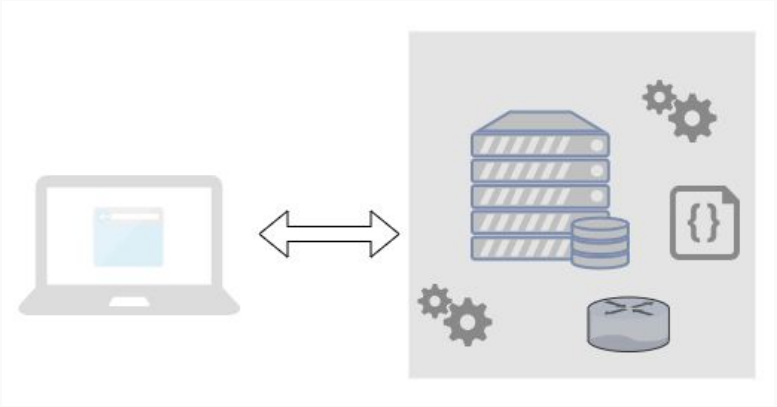
What is this about?

We will build a **web server** using **Node.js** and friends, focusing entirely on **backend development**.

The code and concepts here would be a **great foundation for your next web project!**

Topics include, but not limited to:

- Using Node and Express
- Routing, request and response
- Building a REST API
- Server-rendered templates
- Connecting to a NoSQL (mongo) database
- Using external APIs, such as Spotify



We need:

For Windows users:
Check out this doc if having problems with Node, npm installation
https://goo.gl/WN3LmA

1. WiFi!
2. Join the **Slack**: https://goo.gl/wz8GG2
3. Install Git and Sign up to Github (if you don't have one yet)
    - You'll need this to fork the Github Repo
4. Install **Chrome/Firefox**
5. Install **VSCode** - why? bash terminals built-in
6. Install node - install **latest LTS version**
    - Download and install from https://nodejs.org/en/download/
    - To test: https://nodejs.org/en/docs/guides/getting-started-guide/
7. Check out the cheat sheet https://goo.gl/m8i2gc for some quick info





Nice to haves! (optional but recommended)
Please install these before the workshop

- JSON Viewer browser plugin
  - plenty out there. this is my fave for Chrome

-
- Install mongodb (we will use mLab but local DB is good to have for testi
  - Follow https://docs.mongodb.com/manual/installation/

Git setup also in Slack #resources

1. Log-in to github
2. Go to https://github.com/lenmorld/**node_workshop**
3. In github, fork branch
    - You'll be asked to login to your github account

4. On local terminal, clone the repo

```
$ git clone https://github.com/<username>/node_workshop.git
```

or use SSH if you'd like

5. Checkout dev branch (which should be an empty slate)

```
$ cd node_workshop
```

```
$ git checkout dev
```

6. Open code editor with **node_workshop** as the root directory
7. **We're ready!** 🚀

Git: exploring workshop code

also in Slack #resources

Comparing branches

1. Each chapter/step is numbered, corresponding to a github branch
2. See diff at any step, e.g. to see code in step c1.2
   https://github.com/<user_name>/**node_workshop**/compare/c1.1...c1.2
   Or use this in Github

Checkout code at any step:

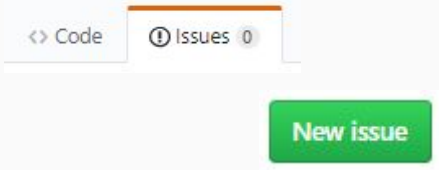
```
$ git stash                  // (and/or git clean -fd) (or commit + push in a new branch)
$ git checkout <chapter>   // e.g.    git checkout c4.1
$ npm install              // if needed, install deps to make code work
$ npm start                // if needed, restart server
```

Introduction

Let’s make this better!

After the workshop/before you leave
Please answer a short survey about the workshop
https://goo.gl/M6Bdc3

Please go to the code repo, and give it a star
(if it deserves one, of course!)
https://github.com/lenmorld/node_workshop



Any mistakes or improvements, please submit an issue
or even better, contribute to open-source! (ping me in Slack for details)

1 Node Basics

c1.0 Hello World console.log()

Execute *node server.js*

```
$ node server.js
>> Hello World!
```

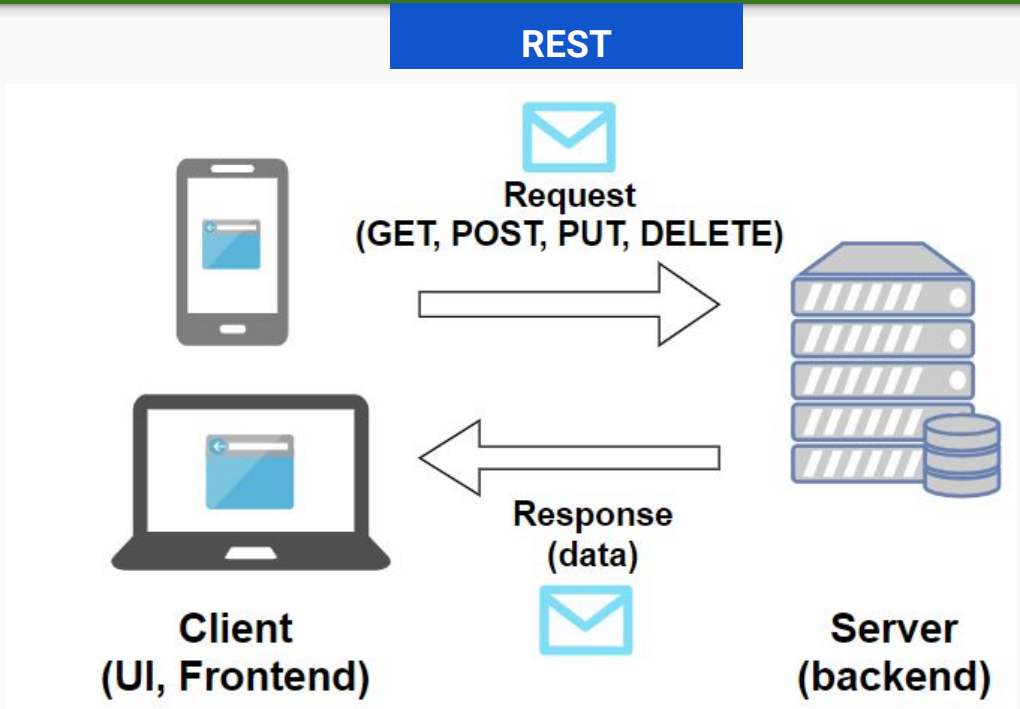
Sample directory structure:

```
node_workshop/
     server.js
     README.md
     ...
```

Review of Client-server architecture

REST allows stateless connection between client and server using HTTP protocols and methods/verbs

Since it's **stateless**, **client** can be in any language/platform that implements HTTP
(here we're using browser and CLI)

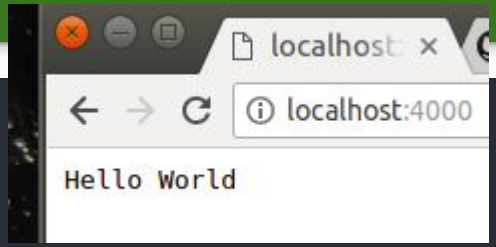


Since it's **stateless**, **server** can be in any language/platform that implements HTTP (here, we're using **Node**)

c1.1 - Hello World! server

with some ES6 syntax

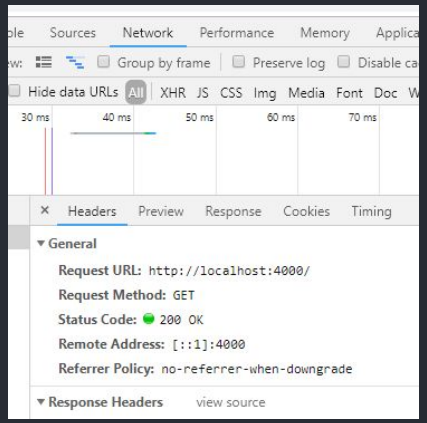
```
# to restart server
CRTL+C to stop server
$ node server.js
>> Server listening at 4000
```




```
// server.js

// import built-in Node package
const http = require('http');
const port = 4000;

const server = http.createServer(function (req, res) { // Callback function
   // Response header
   res.writeHead(200, { "Content-Type": "text/plain" });
   // send response
   res.end("Hello World\n");
});

server.listen(port, function () { // Callback function
   console.log(`Server listening at ${port}`);
});
```

ES6 const

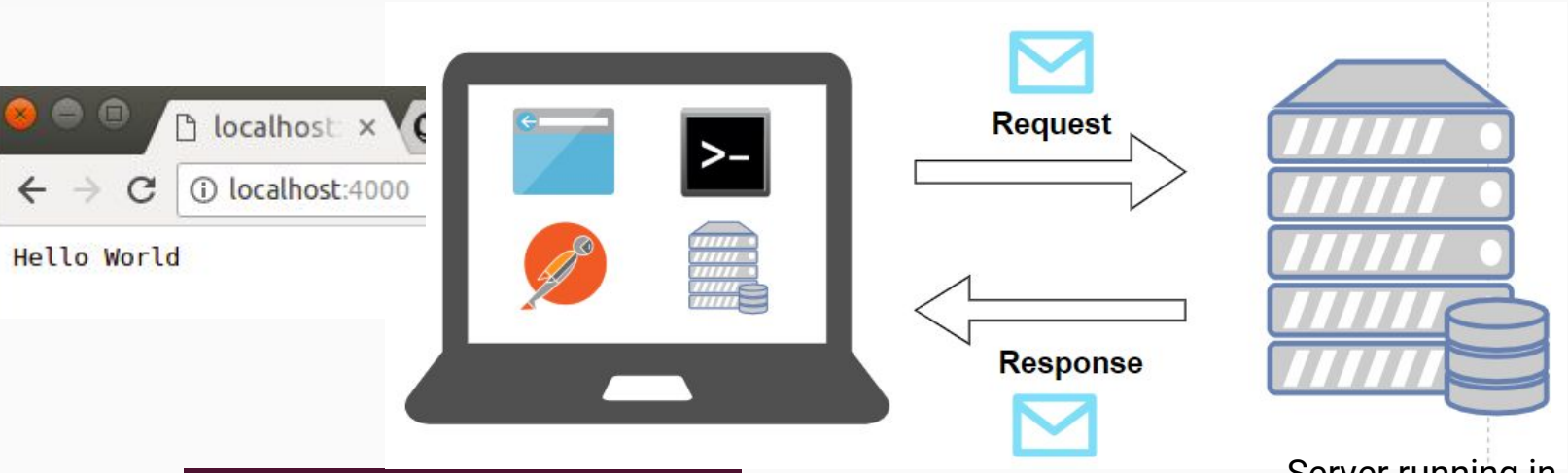
ES6 template literals



Client and Server setup

Server: runs on our machine as **localhost:4000**

Client: sends HTTP requests to the server, which can also process and display the response from the server.

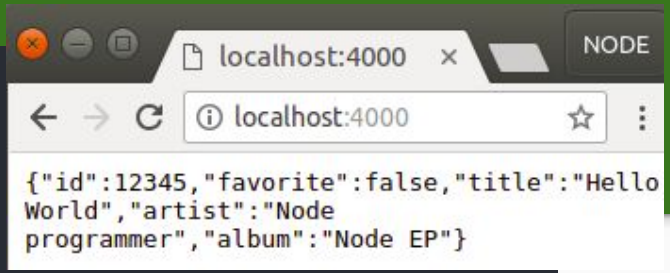



```
# CLIENT: on another terminal
$ curl localhost:4000
>> Hello World
```

C1.2 - Hello World - JSON

```
// server.js

const http = require('http');
const port = 4000;

const server = http.createServer((req, res) => {
   // Response header
   res.writeHead(200, { "Content-Type": "application/json" });

   // JSON object
   const song = {
       id: 12345,
       favorite: false,
       title: "Hello World",
       artist: "Node programmer",
       album: "Node EP"
   };

  // send JSON response to client
  res.end(JSON.stringify(song));      // JSON.stringify({a: 1}) -> '{"a":1}'
});

server.listen(port, function () { // Callback function
   console.log(`Server listening at ${port}`);
});
```

ES6 arrow function



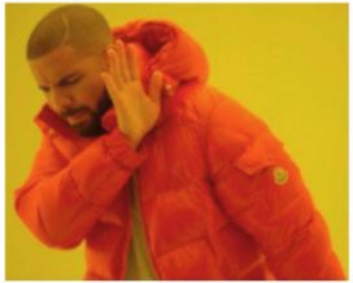

```
# to restart server
CRTL+C to stop server
$ node server.js
>> Server listening at 4000
```

why JSON?

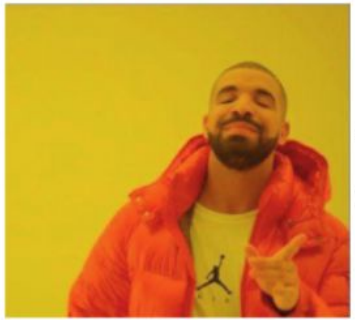
DevHumor | Popular | Categories | Privacy Policy | Shop

It's not hard to see why JSON won, really

Submitted By: admin on February 12, 2019 0 0 1876



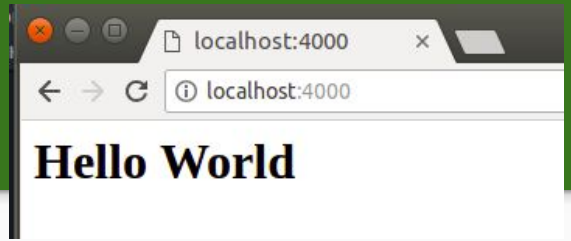
```
<?xml version="1.0"?>
<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
  <soap:Body xmlns:m="https://www.superglobamegacorp.example/stock">
    <m:GetStockPriceResponse>
      <m:StockSymbol>AAPL</m:Price>
      <m:Price>170.41</m:Price>
    </m:GetStockPriceResponse>
  </soap:Body>
</soap:Envelope>
```



```
{
  "stockSymbol": "AAPL",
  "price": 170.41
}
```

Source: www.reddit.com

C1.3 - Hello World - HTML




```
// server.js
...
   // Response header
   res.writeHead(200, { "Content-Type": "text/html" });

  // send HTML response to client
  res.end("<h1>Hello World</h1>");
});
...
```

```
# to restart server
# CRTL+C to stop server
$ node server.js
>> Starting server at 4000
```

How about serving HTML **files**?

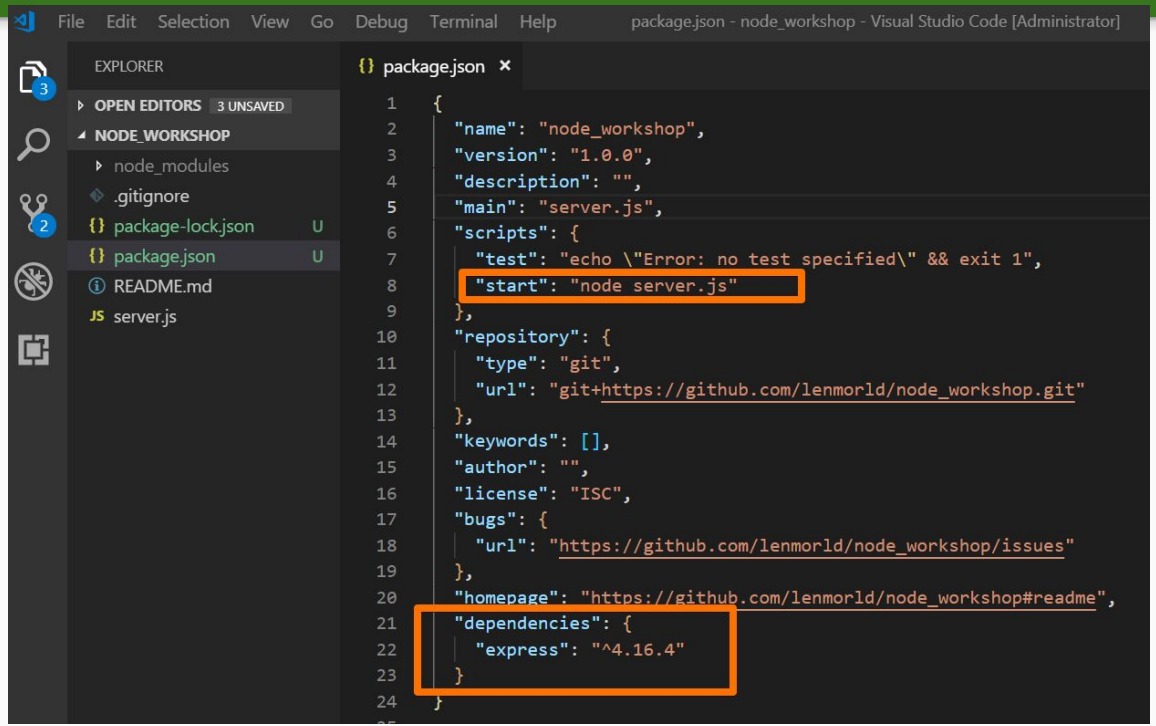
c1.4 - Using `npm` and installing `express`

Express simplifies web server stuff in Node
But first, to install any package (dependency, library) in Node, we need **npm**

```
$ npm init -y
$ npm install express
```

Examine ***package.json***

- installed dependency: express
- Npm scripts



Sidenote: Npm, package.json, node_modules

Node_modules contains all the packages locally

- **add to *.gitignore* !!!**

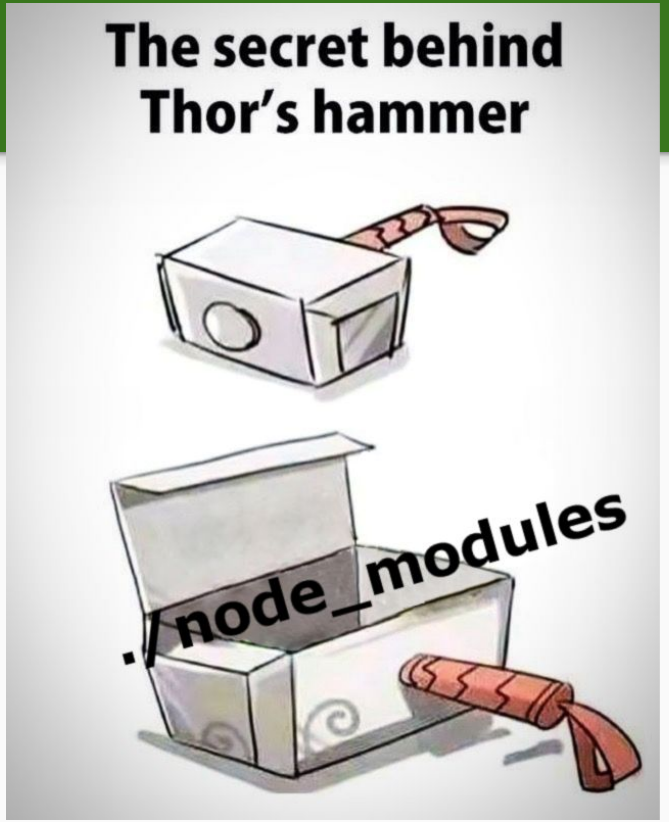
.gitignore

```
1   node_modules/
```

Directory structure:

```
node_workshop/
      .gitignore
      node_modules/
       server.js
      package.json
       package-lock.json
```

*include **package-lock.json** in the files you check in to git
This is useful in *semvar*



Express middlewares

Express is a ***routing*** and ***middleware*** framework***.***

What's a middleware?

- Middlewares are functions that has access to request (req) and response (res) objects
- We use different kinds of middleware to handle requests

c1.5 - Introduction to Routing using *express*

Directory structure:

```
node_workshop/
      server.js
      index.html
      ...
```

```
// server.js

// import built-in Node packages
const express = require('express'); // import express
const server = express();
const port = 4000;

server.get("/", (req, res) => {
   res.sendFile(__dirname + '/index.html');
});

server.get("/json", ({res}) => {
   res.send((JSON.stringify({ name: "Lenny" })));
});

server.listen(port, () => { // Callback function
...
```

ES6 arrow function

ES6 destructuring

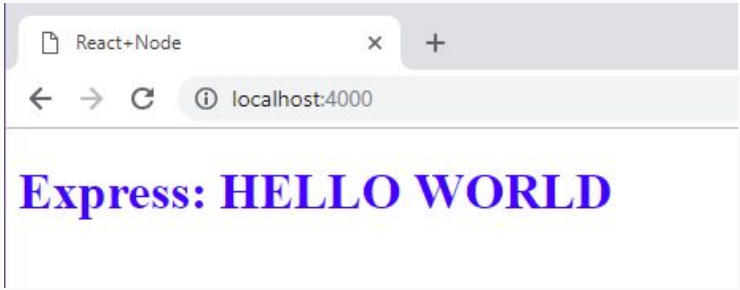
Now we can serve **index.html** using Express.
Create ***index.html*** in project root

```
<!DOCTYPE html>
<html lang="en">
 <head>
   <title>Node Workshop</title>
 </head>
 <body>
     <h1 style="color: blue;">
        Express: HELLO WORLD
     </h1>
 </body>
</html>
```

Running server and testing

Since we have npm now,
we can use `npm start`, instead of `node server.js`

```
# to restart server
# CRTL+C to stop server
$ npm start
```




```
# in another terminal tab/window
$ curl localhost:4000/json
>> {"name":"Lenny"}
```

c1.6 auto restart server.js on changes

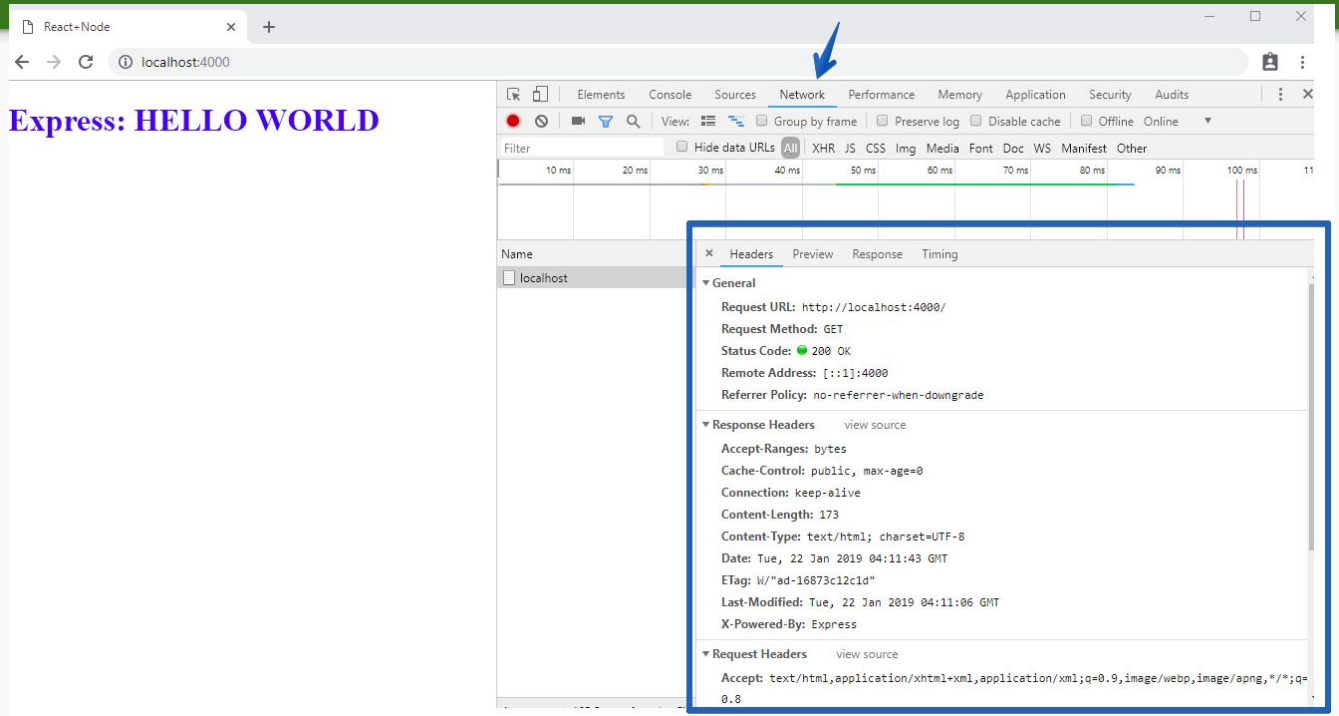
```
# install nodemon
$ npm install nodemon

# in package.json, change start script to
# nodemon server.js

# start server
$ npm start
```

```
// package.json
...
 "scripts": {
   "test": "...,
   "start": "nodemon server.js"
 }
...
```

*Note that doing **$ nodemon server.js** in your terminal doesn't work
but **$ npm start** works, because npm scripts have access to all the installed deps*

Browser dev tools

Network tab
gives details
on request and
response



Templating

Template engine (think JSP, PHP, rhtml but for Node)

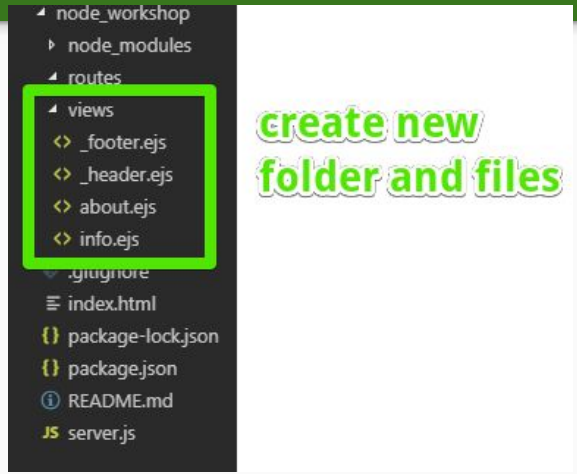
Here we will use **EJS** Template engine**,** since it's the closest to HTML

- Simple HTML-like **views**, but with extra features, like:
  - Passing data to view
    - `<h1><%= my_var %></h1>`
  - **Partials**: reusable parts, perfect for site Header, Footer, Nav, etc
    - `<header><% include ./header.ejs %></header>`
- whenever we need **server-rendered** pages (vs client-rendered)

c1.7 Using EJS templates

1. `$ npm install ejs`
2. Create new folder **views** and some files under it

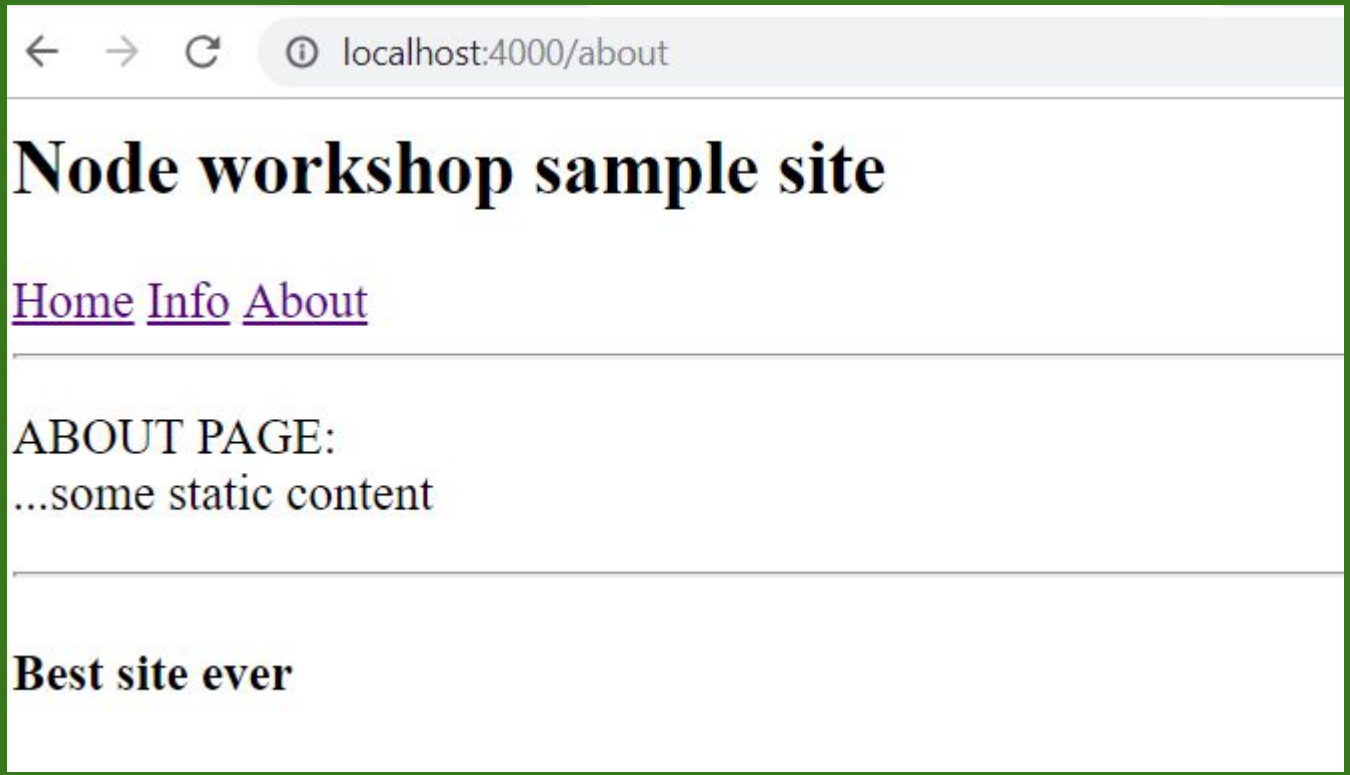
NOTE:
`res.render('about')`
expects a file **views/about.ejs**




```
// server.js
...
// set the view engine to ejs
server.set('view engine', 'ejs');
...
// template pages
server.get("/about", (req, res) => {
   res.render('about');
});

server.get("/info", (req, res) => {
   res.render('info', { message: 'Hello world' });
});
...
```

```
// views/info.ejs
<% include ./_header.ejs %>
<div>
   INFO PAGE <br />
   some data passed from server: <br/>
   <%= message  %>
</div>
<% include ./_footer.ejs %>
```

Complete view of code changes:
https://github.com/lenmorld/node_workshop/compare/c1.6...c1.7


localhost:4000/about
Node workshop sample site
Home Info About
ABOUT PAGE:
...some static content
Best site ever

c1.8 express.Router() - Organizing code

express.Router() middleware allows us to:

- group the route handlers into **route modules** for a particular part of a site together

express.Router() + server.use()

```
// routes/pages.js
...
const express = require('express');
const server = express.Router();

<ALL THE PAGE ROUTES>

module.exports = server;
```

```
// server.js
...
// import route modules
const pages = require('./routes/pages');
...
<REMOVE ALL THE PAGE ROUTES>
// template pages
server.use("/", pages);
...
```

c1.9 express.Router() - mounting route module in different route

express.Router() also allows mounting a route module to a **route prefix**

The good thing with reusable partials like *_header.ejs* is you only change it ones for those views that uses *header.ejs*.

express.Router() + server.use()

```
// views/header.ejs

<a href="/pages/info">Info</a>
<a href="/pages/about">About</a>

// index.html
...
  <a href="/pages/info">Info</a>
  <a href="/pages/about">About</a>
```

```
// server.js
...
server.use("/pages", pages);
```

CHECKPOINT:

Questions

Recap

What can you do with this knowledge?

- Server-rendered pages in HTML (static data) or EJS (dynamic data)
- Simple routing
- using Router() to organize route handlers

Next:

REST API

2 Building a REST server

aka RESTful (HTTP) API web (server)

We're building our own **API, not using an external API (more on that later)*

HTTP requests cardio 🏃‍♀️

Try out these commands to understand HTTP methods, JSON, URL params, request body, content-type

Sample commands:

```
$ curl https://jsonplaceholder.typicode.com/posts/1
$ curl https://jsonplaceholder.typicode.com/posts
$ curl -X POST -H "Content-Type: application/json" --data '{"title": "foo", "body": "bar",
"userId": 1}' https://jsonplaceholder.typicode.com/posts/
$ curl -X PUT -H "Content-Type: application/json" --data '{"title": "foo", "body": "bar",
"userId": 2}' https://jsonplaceholder.typicode.com/posts/1
$ curl -X DELETE https://jsonplaceholder.typicode.com/posts/1


Sample requests cheat sheet: Slack #node_resources “HTTP Snippets”
```

c2.0 setup: CRUD route module

Add a new route module **routes/crud.js**

Use from **server.js**

```
// server.js
...
// import route modules
const pages = require('./routes/pages');
const crud = require('./routes/crud');
...
server.use("/pages", pages);
// crud
server.use("/", crud);
```

```
// routes/crud.js

const express = require("express");
const server = express.Router();

module.exports = server;
```

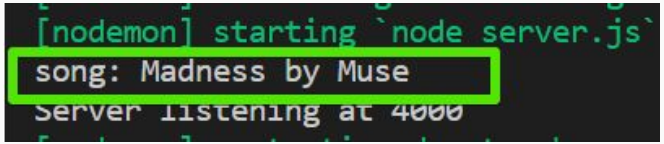
c2.1 The data we'll use

1. Create file ***/data.js***
   a. Copy from **Slack #node_snippets: data.js**
2. Import data.js from server.js
   a. Test by *console.log()*

* This data is in **JSON** format
* Node uses a **module system** to share code between files

```
// data.js

module.exports = {
  name: "Song List",
  owner: "Lenny",
  list: [
    {
      id: "0c4IEciLCDdXEhhKxj4ThA",
      artist: "Muse",
      title: "Madness",
      album: "The 2nd Law"
    },
    {
      id: "2QAHN4C4M8D8E8eiQvQW6a",
      artist: "One Republic",
      title: "I Lived",
      album: "Native"
    }
  ]
};
```

```
// routes/crud.js
...
// import data and controllers
const data = require('./data');
console.log(`song: ${data.list[0].title} by ${data.list[0].artist}`);
```



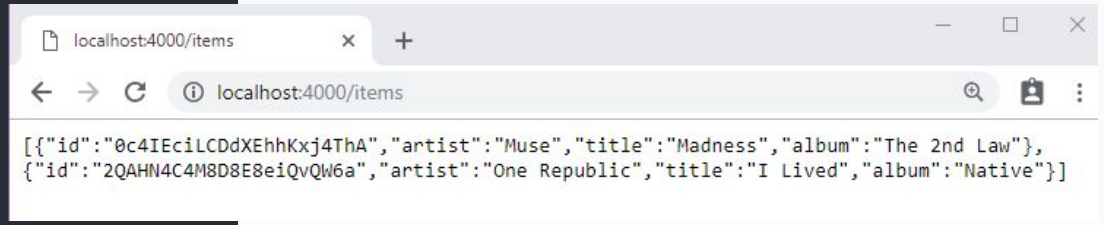
Implementing CRUD in REST HTTP

A REST HTTP web server API provides CRUD functionality to its clients

CRUD method	**HTTP Verb**
Create	POST
Read	GET
Update	PUT
Delete	DELETE

c2.2 Read all items: GET /items route

```
// routes/crud.js
...
// CRUD RESTful API routes
server.get("/items", (req, res) => {
   res.json(data.list);
});
...
```



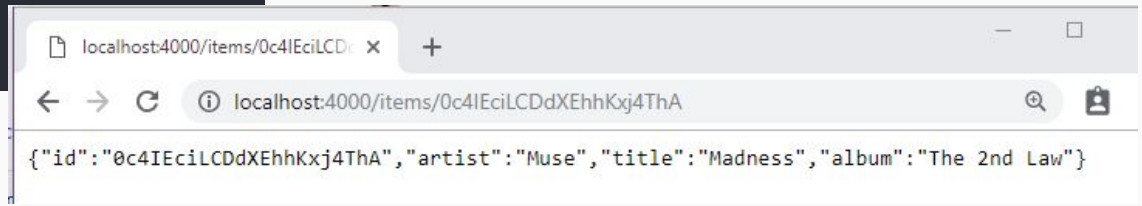

```
$ curl http://localhost:4000/items  Request
Response
[{"id":"0c4IEciLCDdXEhhKxj4ThA","artist":"Muse","title":"Madness","album":"The 2nd
Law"},{"id":"2QAHN4C4M8D8E8eiQvQW6a","artist":"One Republic","title":"I Lived","album":"Native"}]
```

c2.3 Read one item: GET /items/:id route

```
// routes/crud.js

...
// get an item identified by id
server.get("/items/:id", (req, res) => {
    const itemId = req.params.id;
    const item = data.list.find( _item => _item.id === itemId );
    res.json(item);
});
...
```

ES6 arrow function and implicit return


```
server.get("/items/:id", (req, res) => {
    const itemId = req.params.id;
```




```
$ curl http://localhost:4000/items/0c4IEciLCDdXEhhKxj4ThA
{"id":"0c4IEciLCDdXEhhKxj4ThA","artist":"Muse","title":"Madness","album":"The 2nd Law"}
```

Designing routes with HTTP methods

What are the methods and parameters we need for REST API?

Operation	Method	URL	URL params	Request body	example
Create	POST	/items		body: {id, title, artist, album}	POST /items body: {...item details}
Read one	GET	/items/:id	:id (item ID)		GET /items/12345
Read all	GET	/items			GET /items
Update	PUT	/items/:id	:id (item ID)	body: {id, title, artist, album}	PUT /items/12345 body: {...item details}
Delete	DELETE	/items/:id	:id (item ID)		DELETE /items/12345

c2.4 Create item: POST /items route

```
// need this to decode request body
$ npm install body-parser
```

1. Install and use *body-parser* middleware for the entire app
2. Inside POST /items route
   a. Obtain item from **req.body**
   b. Add item to array
   c. Return updated list

```
// server.js
...
cosnt server = express();
const body_parser = require('body-parser');
...
server.set('view engine', 'ejs');
server.use(body_parser.json()); // parse JSON (application/json content-type)
...
```

```
// routes/crud.js
...
// create/post new item
server.post("/items", (req, res) => {
   const item = req.body;
   console.log('Adding new item: ', item);

   // add new item to array
   data.list.push(item)

   // return updated list
   res.json(data.list);
});
...
```

Testing

*while we can use browser for GET requests,
for POST, PUT, DELETE we have to use cURL, Postman,
or any REST client (some browser extensions available too)

```
$ curl -X POST -H "Content-Type: application/json" --data '{"id": "abcd1234", "artist": "Meee", "title":
"sooong", "album": "Hello Word"}' http://localhost:4000/items

[{"id":"0c4IEciLCDdXEhhKxj4ThA","artist":"Muse","title":"Madness","album":"The 2nd
Law"},{"id":"2QAHN4C4M8D8E8eiQvQW6a","artist":"One Republic","title":"I
Lived","album":"Native"},{"id":"abcd1234","artist":"Meee","title":"sooong","album":"Hello Word"}]
```

```
Server listening at 4000
Adding new item:  { id: 'abcd1234',
  artist: 'Meee',
  title: 'sooong',
  album: 'Hello Word' }
```

c2.5 Update item: PUT /items/:id

Inside PUT /items/:id route

1. Obtain itemId from **req.params** and item from **req.body**
2. Replace target item in the list with new one**
3. Return list

*** many ways to implement this*

```
// routes/crud.js
...
// update an item
server.put("/items/:id", (req, res) => {
   const itemId = req.params.id;
   const item = req.body;
   console.log("Editing item: ", itemId, " to be ", item);

   const updatedListItems = [];
   // loop through list to find and replace one item
   data.list.forEach(oldItem => {
      if (oldItem.id === itemId) {
         updatedListItems.push(item);
      } else {
         updatedListItems.push(oldItem);
      }
   });

   // replace old list with new one
   data.list = updatedListItems;

   res.json(data.list);
});
```

Testing

```
$ curl -X PUT -H "Content-Type: application/json" --data '{"title":"Muse++", "artist":"Madness++",
"album":"The 3rd Law", "id":"0c4IEciLCDdXEhhKxj4ThA"}' http://localhost:4000/items/0c4IEciLCDdXEhhKxj4ThA

[{"title":"Muse++","artist":"Madness++","album":"The 3rd
Law","id":"0c4IEciLCDdXEhhKxj4ThA"},{"id":"2QAHN4C4M8D8E8eiQvQW6a","artist":"One Republic","title":"I
Lived","album":"Native"}]
```

After successful PUT, we can verify update with a GET request
(e.g. using a browser)

c2.6 Delete item: DELETE /items/:id

Inside DELETE /items route

1. Obtain itemId from **req.params** using ES6 object destructuring
2. Delete item from list**
3. Return list

*** many ways to implement this*

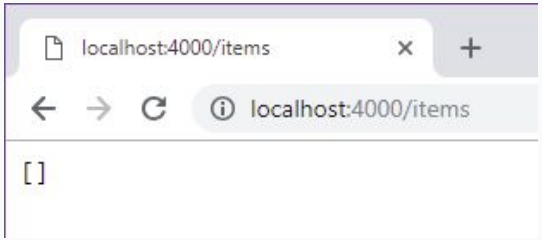
```
// server.js

...
// delete item from list
server.delete("/items/:id", (req, res) => {
   const { id } = req.params;
   const itemId = id;

   console.log("Delete item with id: ", itemId);

   // filter list copy, by excluding item to delete
   const filtered_list = data.list.filter(item => item.id !== itemId);

   // replace old list with new one
   data.list = filtered_list;

   res.json(data.list);
});
```

ES6 object destructuring

Testing

```
$ curl -X DELETE http://localhost:4000/items/0c4IEciLCDdXEhhKxj4ThA
[{"id":"2QAHN4C4M8D8E8eiQvQW6a","artist":"One Republic","title":"I Lived","album":"Native"}]

$ curl -X DELETE http://localhost:4000/items/2QAHN4C4M8D8E8eiQvQW6a
[]
```

After successful DELETE of all items, we can verify that list is empty, with a GET request



m1 (Milestone 1)
Building a server-rendered playlist app

https://github.com/<username>/node_workshop/tree/m1

In this activity, we implement:

- Playlist page that shows all items in the list
- Create and Edit page, linked to items in Playlist
- Delete function linked to items in Playlist

CHECKPOINT:

Questions

Recap

What can you do with this knowledge?

- CRUD pages for an information system in memory (no DB)

Next:

Data doesn't persist yet. When server restart/dies, our changes are lost 😖

Coming up: Persisting data into database

3 Connecting to a database: MongoDB

noSQL databases

```
// a document record sample in Mongodb mLab

{
    "_id": {
        "$oid": "5bafebc0c5fcaf59fed82980"
    },
    "id": "5VnDkUNyX6u5Sk0yZiP8XB",
    "title": "Thunder",
    "artist": "Imagine Dragons",
    "album": "Evolve"
}
```

What and why?

- Document-based
- Represents programming objects closer than relational databases
- Speed and flexibility
- Denormalization - e.g. get `item`, `item.artists`, `item.artist.stuffs` without joins
- Why not? Not good for ACID compliant purposes, e.g. joins

MongoDB database structure

NoSQL are usually structured in database > collections > documents

- in contrast to tables > rows in SQL databases

MongoDB is a noSQL database

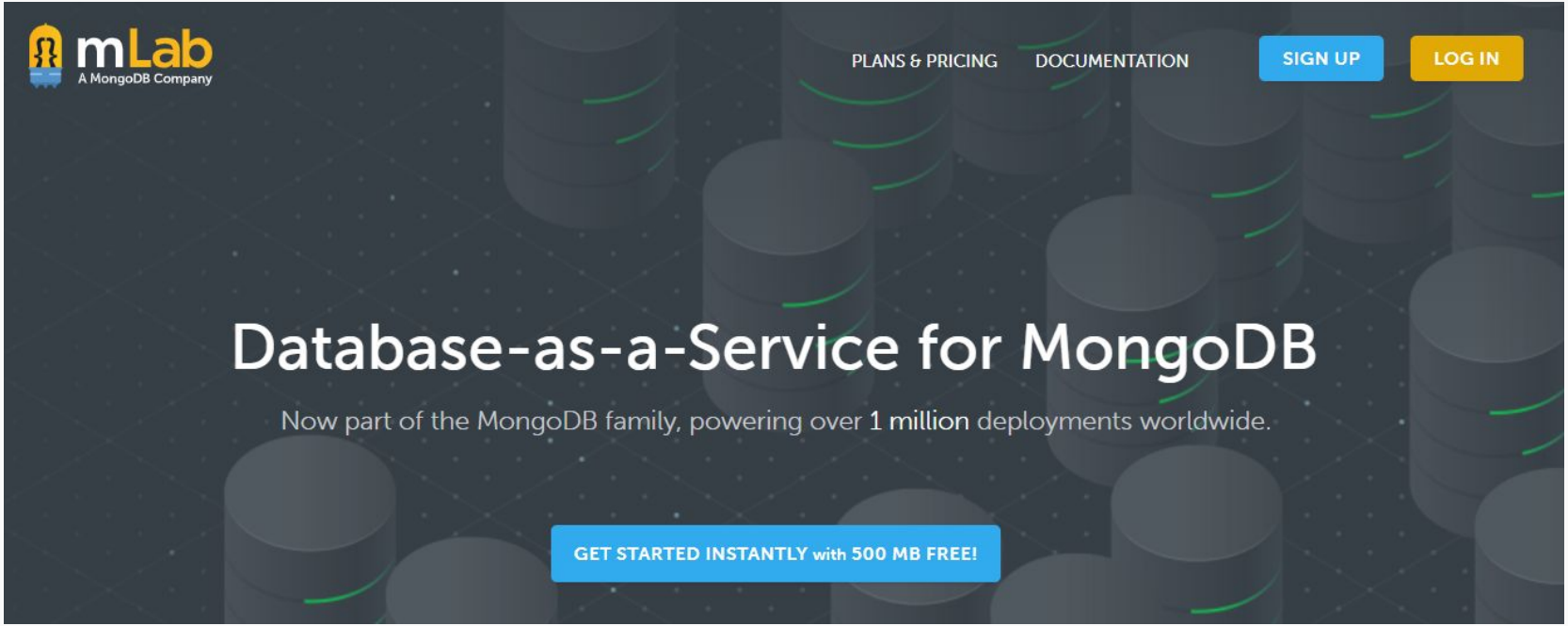
We will use mLab to have a mongoDB instance without any installation/config

```
database                node_workshop_db
    collections             items: [{}, {}, {}]
        documents                   {id: 1234, title: “my song”, ...}
```

mLab

mLab is the quickest way to setup a MongoDB instance

Free up to 0.5 GB



c3.1 Setting up mLab and MongoDB

For this workshop:

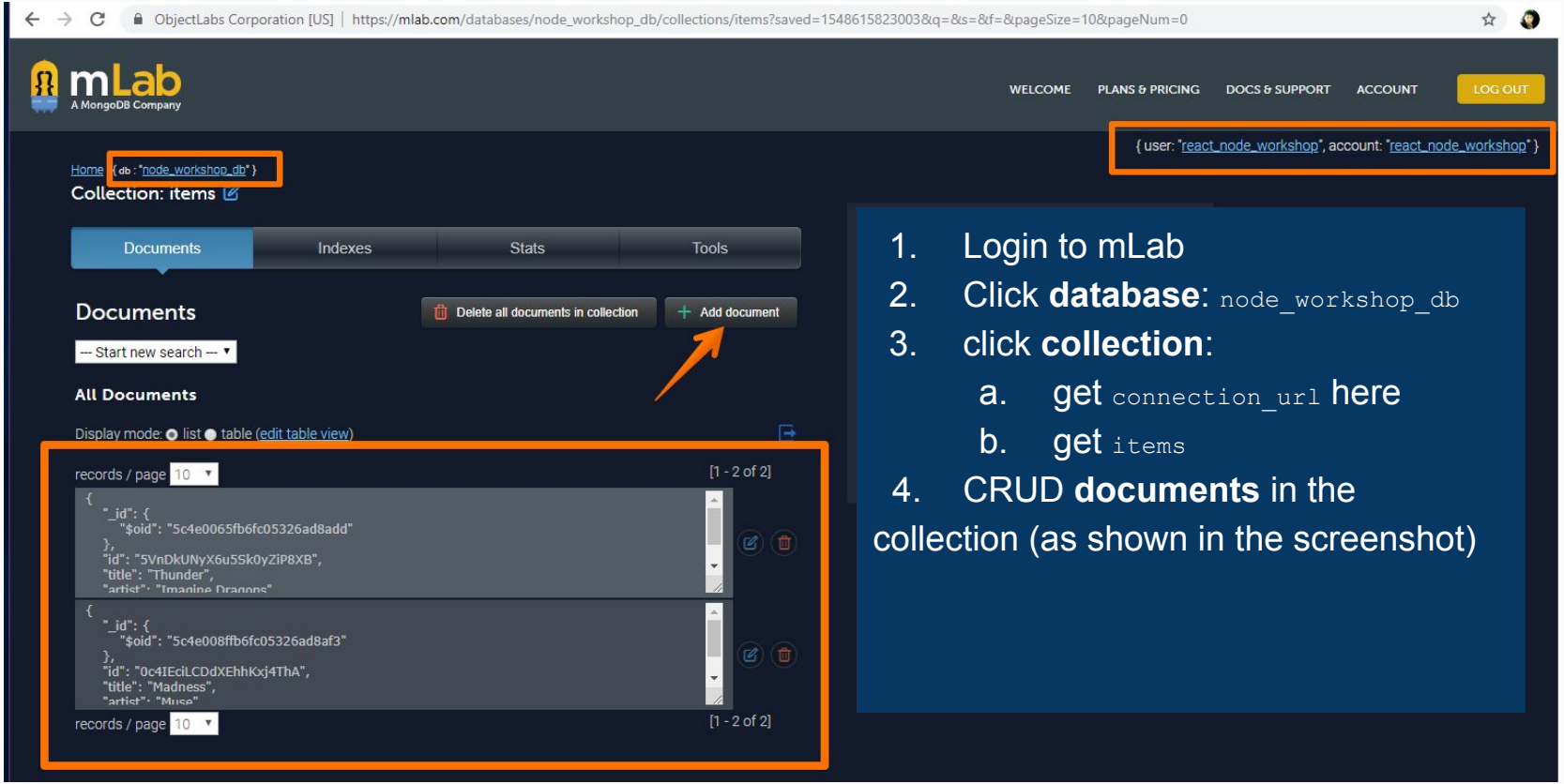
- We'll use my mLab db instance, with an initial user and some data
  I'll send the credentials through Slack
- Install mongodb driver to our project

```
# install mongodb to our project
$ npm install mongodb
```

** If you want to setup your own:

https://docs.mlab.com/

Setup and testing of mLab database



1. Login to mLab
2. Click **database**: `node_workshop_db`
3. click **collection**:
   a. get `connection_url` here
   b. get `items`
4. CRUD **documents** in the collection (as shown in the screenshot)

Optional: **local** MongoDB setup

***** This only works if you installed mongodb in your local machine as defined in https://docs.mongodb.com/manual/administration/install-community/***

1. On another terminal, start local mongodb server on your machine
   - Ubuntu: `sudo service mongod start`
   - Mac: `mongod`
   - Windows: *"C:\Program Files\MongoDB\Server\4.0\bin\mongod.exe" --dbpath="c:\data\db"*
2. On another terminal, start mongo client as described in previous slide

```
# after starting local mongodb server and
# after connecting to server using a mongodb client

# use db and start running mongodb commands
> use node_workshop_db

> db.items.insertOne( { id:"some_song_id", artist:"The Artist", "title": "Song
song", "album": "The Album" } )

> db.items.find()
# should see fake song we just added
```

Optional: Testing **local** DB connection

*** This only works if you installed mongodb in your local machine as defined in*

https://docs.mongodb.com/manual/administration/install-community/

```
# --- mongoDB CLIENT ----
# on another terminal, connect to db

# Linux
$ mongo <mongodb_connection_link>
# macOS
$ mongo --host <mongodb_connection_link>
# Windows
$ "C:\Program Files\MongoDB\Server\4.0\bin\mongo.exe"


# use db and start running mongodb commands
> use node_workshop_db

> db.items.find()
# should see some JSON-like objects here with song details
```

Async Programming with Promises

Time to brush up on Asynchronous Programming concepts!

Javascript (frontend and Node.js) runs asynchronously (no waiting and event-driven).

We've been doing it the whole time, using **callbacks** required by express methods.

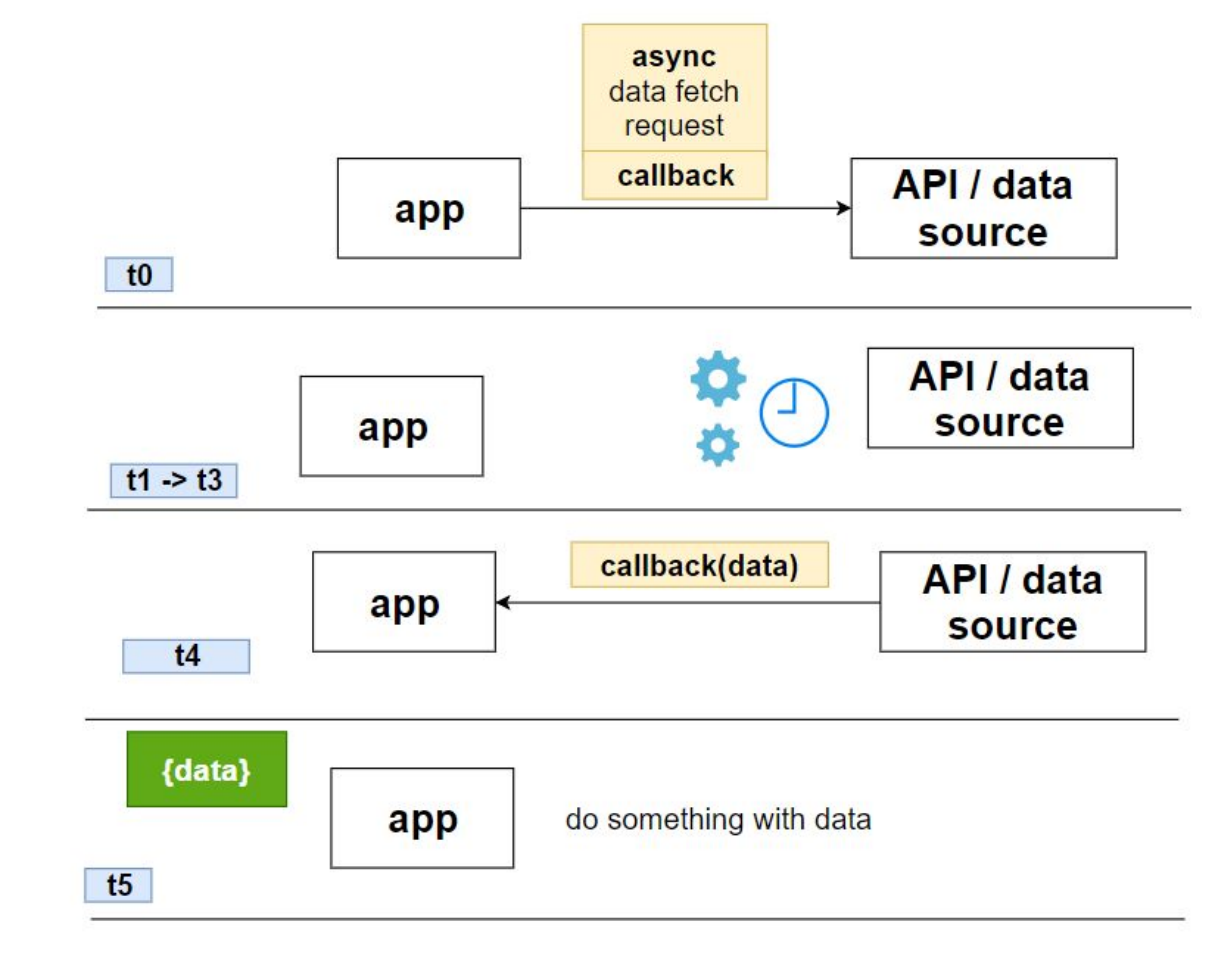
```
// routes/crud.js
...
server.put("/items/:id", function (req, res)  {...});
```

```
// server.js
...
server.listen(port, () => {...});
```

Callbacks

Sender sends a function along with request, which the service will “call back” at a later time to give what was requested

Analogy: when you leave your number at the answering machine; so they call you back later



Callbacks

```
function fetchData(successCallback, failureCallback) {
   doSomethingAsync();   // results to error or data
   if (error) {
       failureCallback(error);
   } else {
       successCallback(data);
   }
}

//ES5
fetchData( function(result){  // successCallback
   doSomething(result);
}, function(error) {          // failureCallback
   throw error;
});

// ES6
fetchData( result => {  // successCallback
   doSomething(result);
}, error => {          // failureCallback
   throw error;
});
```

c3.2 Mongo connection initDb using Callback

*** If using local mongodb, use the *localhost* version of *db_connection_url*

```
// mongo_db.js

... mLab CREDENTIALS: get from Slack #node_snippets

// CALLBACK VERSION
function initDb(dbCollectionName, successCallback, failureCallback) {
   const dbConnectionUrl = `mongodb://${dbUser}:${dbPass}@${dbHost}/${dbName}`;
   MongoClient.connect(dbConnectionUrl, function (err, dbInstance) {
       if (err) {
           console.log(`[MongoDB connection] ERROR: ${err}`);
           failureCallback(err);        // this should be "caught" by the calling function
       } else {
           const dbObject = dbInstance.db(dbName);
           const dbCollection = dbObject.collection(dbCollectionName);
           console.log("[MongoDB connection] SUCCESS");
           successCallback(dbCollection);
       }
   });
}
module.exports = { initDb };
```

c3.2 Server code for mongodb, using Callback and ES5

```
// routes/db_crud.js
const express = require("express");
const server = express.Router();
const mongo_db = require('../mongo_db');

// mongoDB connection
const collectionName = 'items';
// CALLBACK VERSION
mongo_db.initDb(collectionName, function (dbCollection) {   // successCallback
   server.get("/items", function (req, res) {
      dbCollection.find().toArray(function (err, result) {
         if (err) throw err;
         res.json(result);
      })
   });
}, function (err) {  // failureCallback
   throw (err);
});

module.exports = server;
```

```
// server.js
...
const db_crud = require('./routes/db_crud');
...
// server.use("/", crud);    // COMMENT Regular CRUD FOR NOW
// db crud
server.use("/", db_crud);
...
```

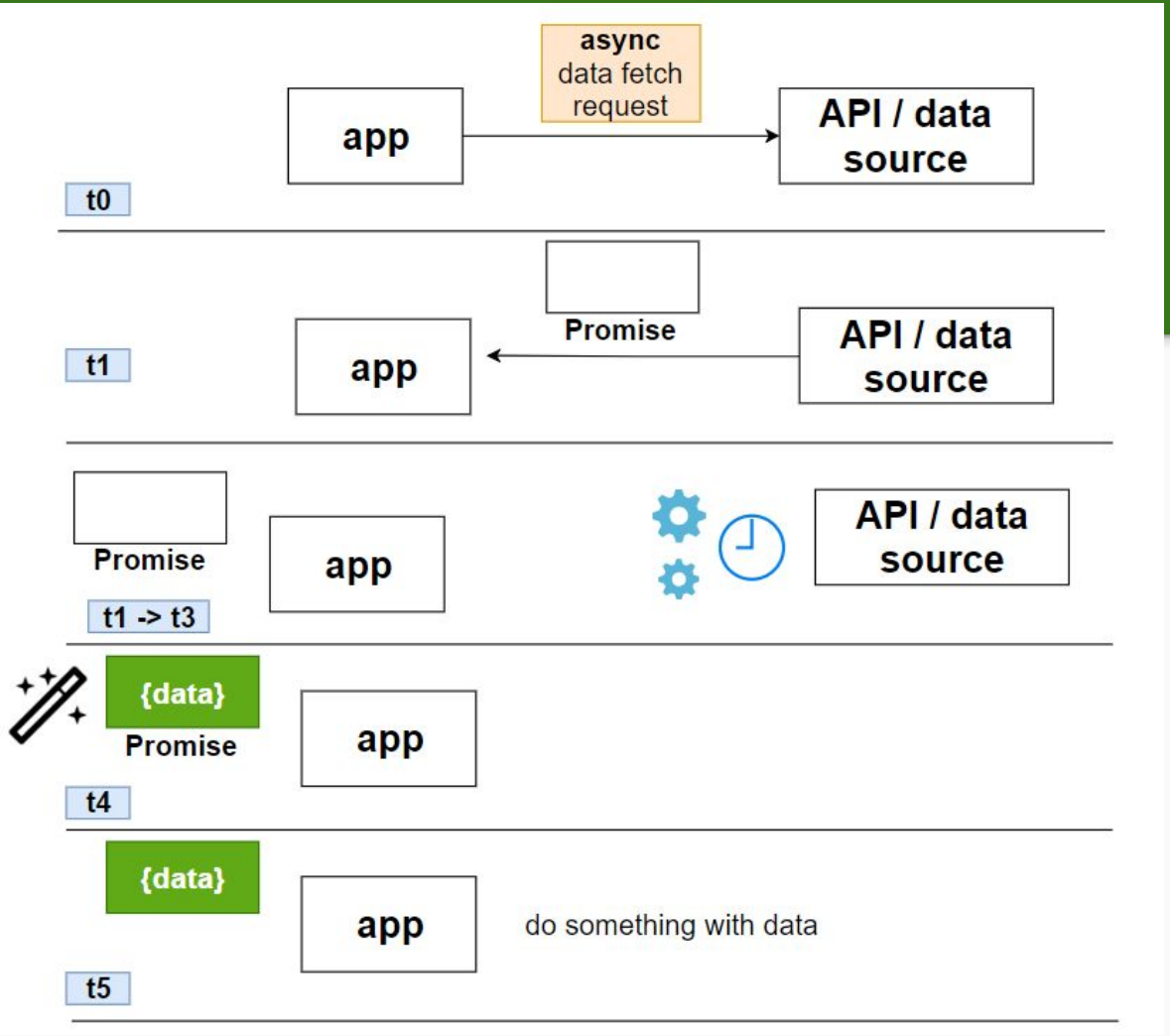
localhost:4000/items


```
[{"_id":"5c561232a9fac6055dbaab64","id":"15jRg8Ejs7X3JEHw38TSN3","title":"Teams"
{"_id":"5c561263a9fac6055dbaab65","id":"1NrJYpdAi7uosDRPmSYrsG","title":"Apologi
{"_id":"5c576cf7474c60c040c34c36","id":"7D49Iig0avHre9RFSUMkd2","title":"I Lived
{"_id":"5c576d9c474c60c040c34c38","id":"6rPO02ozF3bM7NnOV4h6s2","title":"Despaci
{"_id":"5c63954445699731ce378c81","id":"3TwtrR1yNLY1PMPsrGQpOp","title":"Superma
```

Promises

Service sends a Promise that later resolves to the data requested

Analogy: when someone gives you a prize/gift voucher only valid later



ES6 Promises

*Note that **promises** are not available in ES5
But you can still use ES5 *function() {}* syntax when using ES6 promises

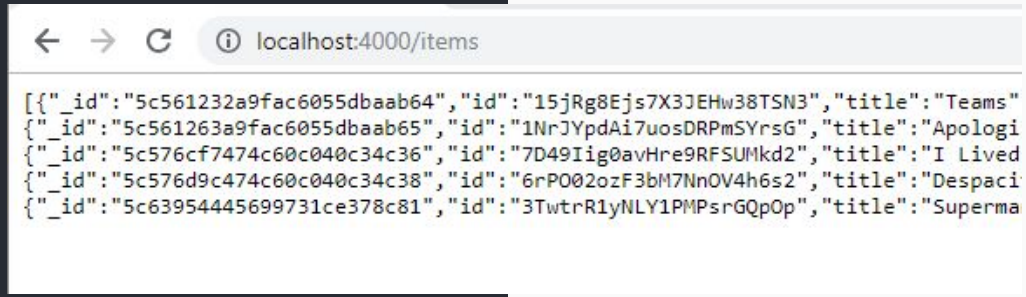
```
function fetchData() {
 return new Promise(function(resolve, reject) {
   doSomethingAsync();   // results to error or data
   if (error) {
       reject(error);
   } else {
       resolve(data);
   }
 });
}

//ES5 syntax
fetchData().then(function(result){
   doSomething(result);
}).catch(function(error) {
   throw error;
});

// ES6 syntax
fetchData().then(result => {
   doSomething(result);
}).catch(error => {
   throw error;
});
```

c3.3 initDb using Promise and ES6 (we're using this)

```
// mongo_db.js
...
// PROMISE VERSION
const initDb2 = (dbCollectionName) => {
   const dbConnectionUrl = `mongodb://${dbUser}:${dbPass}@${dbHost}/${dbName}`;
   // function returns a promise that resolves to a mongodb instance
   return new Promise((resolve, reject) => {
       MongoClient.connect(dbConnectionUrl, (err, dbInstance) => {
           if (err) {
               console.log(`[MongoDB connection] ERROR: ${err}`);
               reject(err);        // this should be "caught" by the calling function
           } else {
               const dbObject = dbInstance.db(dbName);
               const dbCollection = dbObject.collection(dbCollectionName);
               console.log("[MongoDB connection] SUCCESS");
               resolve(dbCollection);
           }
       });
   });
}
module.exports = { initDb, initDb2 };
```

c3.3 Server code for mongodb, using Callback

```
// routes/db_crud.js
...
// COMMENT OUT CALLBACK VERSION
// PROMISE VERSION
mongo_db.initDb2(collectionName).then(dbCollection => {
   // db-based CRUD RESTful API routes
   // get all items
   server.get("/items", (req, res) => {
      dbCollection.find().toArray((err, result) => {
         if (err) throw err;
         res.json(result);
      });
   });
}).catch(err => {
   throw (err);
});

module.exports = server;
```



MongoDB operations

Operation	Method	URL	mongoDB method
Create	POST	/items	collection.**insertOne**(obj, callback)
Read one	GET	/items/:id	collection.**findOne**(query, callback)
Read all	GET	/items	collection.**find**(query).toArray(callback)
Update	PUT	/items/:id	collection.**updateOne**(query, { $set: obj }, callback)
Delete	DELETE	/items/:id	collection.**deleteOne**(query, callback)

Examples:

```
obj
{
 id: "blah20",
 artist: "Artist",
 title: "Title",
 album: "Album"
}

query
{ id: "blah20" }

callback
function (err, result) {
  // process result
}
```

c3.4 db_collection.findOne({...params})

```
mongo_db.initDb2(collectionName).then(dbCollection => {
  . . .
   server.get("/items", (req, res) => {...}
   });

   // get an item identified by id
   server.get("/items/:id", (req, res) => {
      const itemId = req.params.id;
      dbCollection.findOne({ id: itemId }, function (err, result) {
         if (err) throw err;
         res.json(result);
      });
   });
  . . .
```

c3.5 db_collection.insertOne({...params})

```
mongo_db.init_db(db_connection_url).then(function(db_instance) {
  . . .
    // create/post new item
    server.post("/items", function(req, res) {
        var item = req.body;
        db_collection.insertOne(item, function(err, result) {
            if (err) throw err;
            // send back entire updated list after successful request
            db_collection.find().toArray(function(_err, _result) {
                if (_err) throw _err;
                res.json(_result);
            });
        });
    });

  . . .
```

check mLab as well

```
$ curl -X POST -H "Content-Type: application/json" --data '{"id": "abcd1234", "artist":
"Meee", "title": "sooong", "album": "Hello Word"}' http://localhost:4000/items

[{...}, {...},
{"_id":"5c4e712b4a36322c809f8594","id":"abcd1234","artist":"Meee","title":"sooong","album":"
Hello Word"}]
```

c3.6 db_collection.updateOne({...params})

```
mongo_db.init_db(db_connection_url).then(function(db_instance) {
  . . .
   // update an item
   server.put("/items/:id", function(req, res) {
       var item_id = req.params.id;
       var item = req.body;

       db_collection.updateOne({ id: item_id }, { $set: item }, function(err, result) {
           if (err) throw err;
           // send back entire updated list, to make sure frontend data is up-to-date
           db_collection.find().toArray(function(_err, _result) {
               if (_err) throw _err;
               res.json(_result);
           });
       });
   });
  . . .
```

c3.7 db_collection.deleteOne({...params})

```
mongo_db.init_db(db_connection_url).then(function(db_instance) {
  . . .
   // delete item from list
   server.delete("/items/:id", function(req, res) {
       var item_id = req.params.id;
       db_collection.deleteOne({ id: item_id }, function(err, result) {
           if (err) throw err;
           // send back entire updated list after successful request
           db_collection.find().toArray(function(_err, _result) {
               if (_err) throw _err;
               res.json(_result);
           });
       });
   });
  . . .
```

Testing Edit and Delete

```
$ curl -X PUT -H "Content-Type: application/json" --data '{"title":"Soong v2",
"artist":"meee and youuu", "album":"Hey World", "id":"abcd123456"}'
http://localhost:4000/items/abcd1234

[{...}, {...}
{"_id":"5c74c2ccdd14b6c6ca47ca2f","id":"abcd123456","artist":"meee and youuu","title":"Soong
v2","album":"Hey World"}]
```

check mLab as well

```
$ curl -X DELETE http://localhost:4000/items/abcd123456

... check that deleted item is gone, using cUrl or browser

$ curl http://localhost:4000/items/
```

TESTING

- App should work exactly as before, but changes are persisted now in the mLab DB

CHECKPOINT:

Questions

Recap

What can you do with this knowledge?

- CRUD system with a DB
- For reference of all mongoDb methods:
  - https://docs.mongodb.com/manual/reference/method/

Next:

REST API

m2 (Milestone 2)
Connecting m1 to mongodb

https://github.com/<username>/node_workshop/tree/m2

In this activity, we:

- use mongoDb connection to replace route handler logic with mongodb methods

4 Using external APIs

external API?
What is that?
Aren’t we doing an
API already?

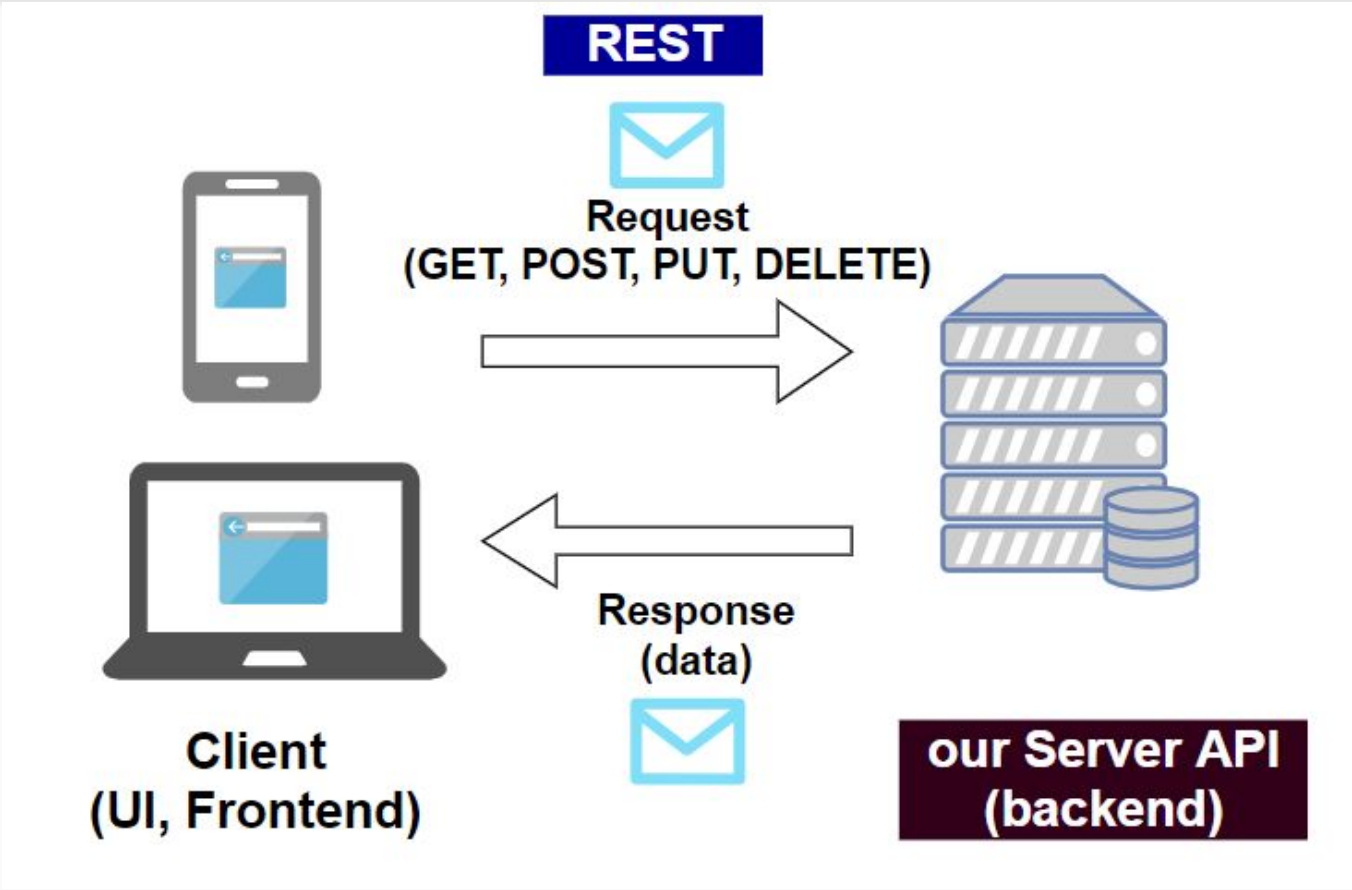
Communicating with APIs

We have been doing a **RESTful API** that a client (browser, cURL) can send requests to

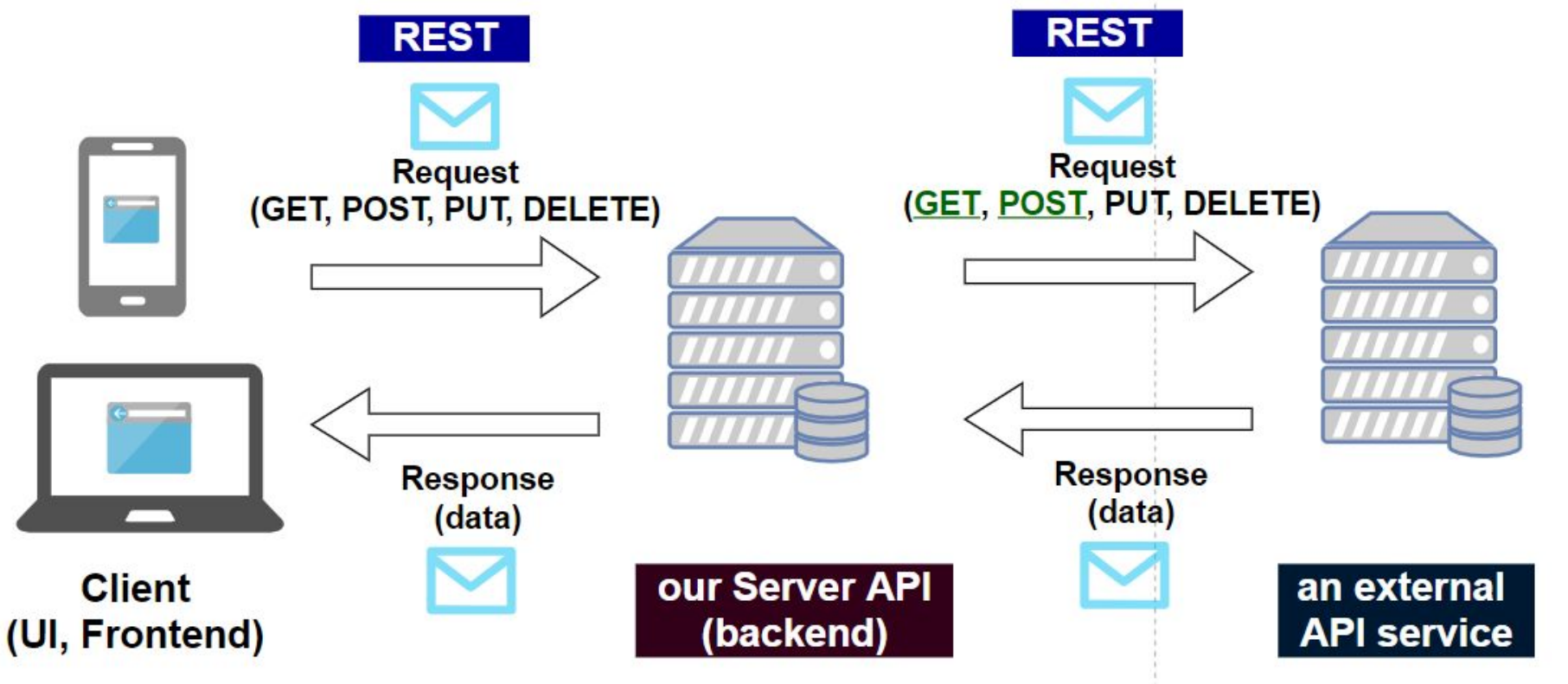
Now, our server will being the “client” and send requests to an external service e.g. searching a song using Spotify API, fetching latest tweets from Twitter API

How our server communicates with an **external API** is exactly the same as how our client communicates with our server, which is through HTTP requests, usually GET and POST 😊

Client <-> Server



Client <-> Server <-> external API



They have APIs for everything nowadays!

Using external APIs require varying degrees of authentication:

- **no auth**
- **API Key**
- **access tokens (OAuth)**
- etc

After we satisfy the auth scheme, we can successfully send HTTP requests to the API

Here is an exhaustive list of some public APIs, with varying degree of auth:
https://github.com/toddmotto/public-apis

💡 **Tip:** When looking at an API's website, look for examples, documentation, quick start on:

- auth method required, and how
- available endpoints
- available HTTP methods

Using APIs with **no auth**

we will use: *jsonplaceholder.typicode.com*

For APIs without auth, we can make an HTTP request directly!

To simplify sending a request from server, we use a library called **axios**

Example of some fun APIs without auth:

- jsonplaceholder: some sample data
- recipepuppy.com/api: recipes
- jobs.github.com/api: search jobs
- icanhazdadjoke.com/api: random jokes

```
# axios is a HTTP request library
$ npm install axios
```

Let’s try it out

Sample commands:

```
$ curl https://jsonplaceholder.typicode.com/posts/1
$ curl https://jsonplaceholder.typicode.com/posts
$ curl -X POST -H "Content-Type: application/json" --data '{"title": "foo", "body": "bar",
"userId": 1}' https://jsonplaceholder.typicode.com/posts/
$ curl -X PUT -H "Content-Type: application/json" --data '{"title": "foo", "body": "bar",
"userId": 2}' https://jsonplaceholder.typicode.com/posts/1
$ curl -X DELETE https://jsonplaceholder.typicode.com/posts/1
```

Sample requests cheat sheet: **Slack #node_resources “HTTP Snippets”**

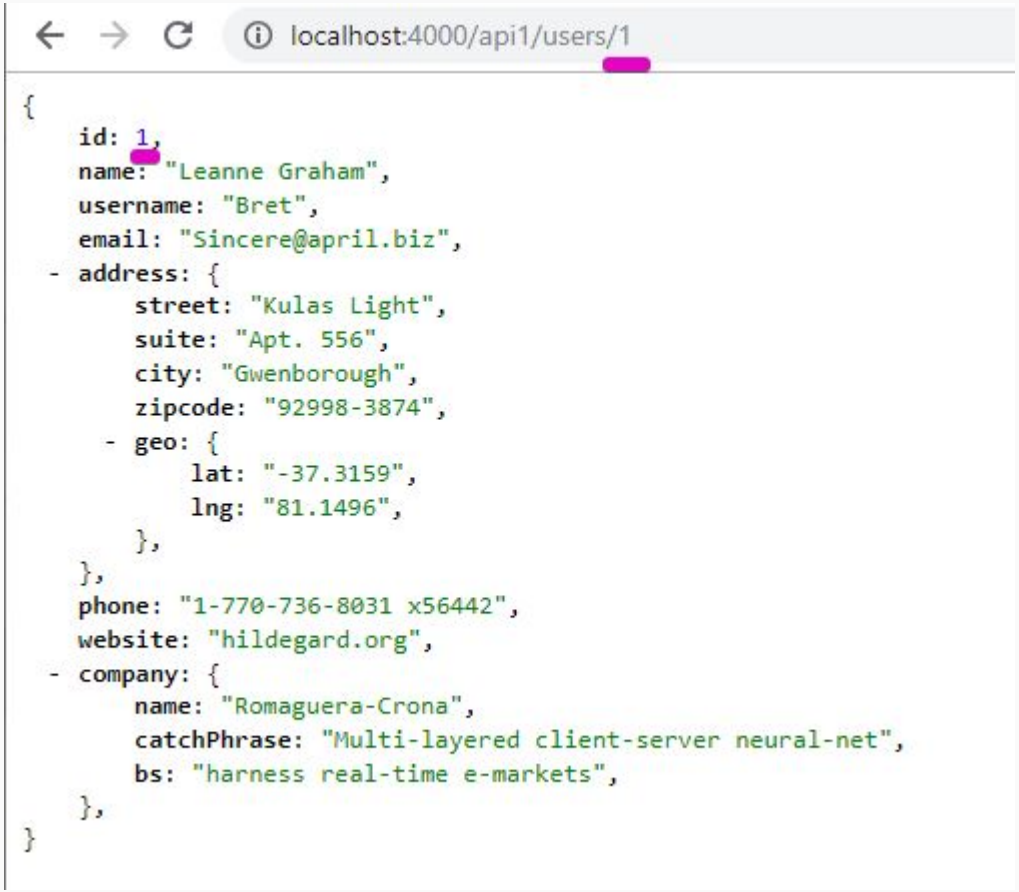
c4.1 sending requests to API with no auth

```
// routes/api1.js

var axios = require('axios');
var express = require('express');
var api1 = express.Router();

// external API routes
api1.get("/users/:id", function (req, res) {
  var id = req.params.id;
axios(`https://jsonplaceholder.typicode.com/users/${id}`)
  .then(response => {
    res.json(response.data);
  }).catch(err => {
    throw err;
  });
});

module.exports = api1;
```

1. create **routes/api1.js**
   a. send request
   b. *NOTE use of Promise
2. **server.js**
   a. import and ***mount*** to locahost:4000/api1

```
// server.js
...
const api1 = require('./routes/api1');
...
server.use('/api1', api1);
```

Testing - **install a Chrome JSON Viewer plugin to have this "prettified" output**



c4.2 Accessing *req.query*

e.g. (site.com/jobs?location=montreal&description=javascript)

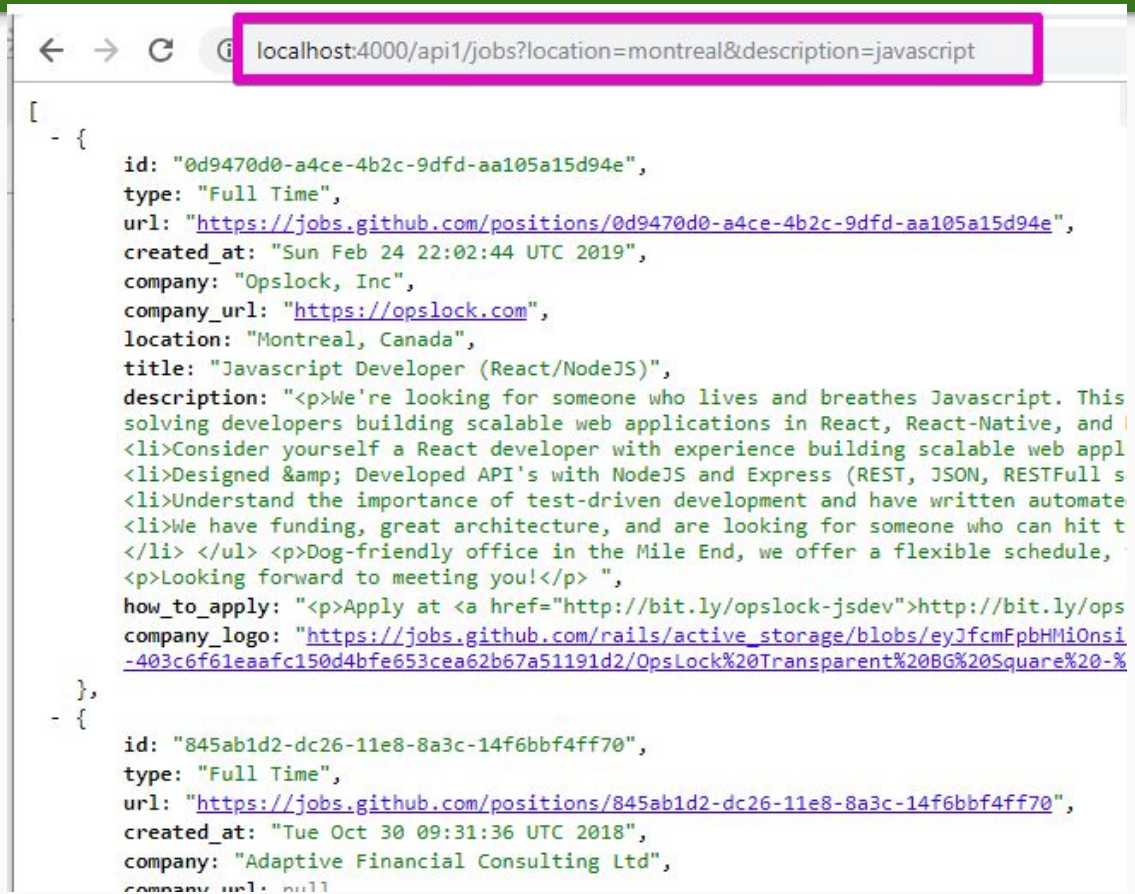
```
// routes/api1.js

api1.get("/jobs", function (req, res) {
   const { description, location } = req.query;
   let requestUrl = "https://jobs.github.com/positions.json";
   if (description || location) {
      const descriptionParam = description ? `description=${description}&` : '';
      const locationParam = location ? `location=${location}&` : '';
      requestUrl = `${requestUrl}?${descriptionParam}${locationParam}`;
   }
   console.log(`Request: ${requestUrl}`);
   axios(requestUrl).then(response => {
      res.json(response.data);
   }).catch(err => {
      throw err;
   });
});
```

ES6 object destructuring

ES6 let

ternary

Testing


localhost:4000/api1/jobs?location=montreal&description=javascript
[
- {
id: "0d9470d0-a4ce-4b2c-9dfd-aa105a15d94e",
type: "Full Time",
url: "https://jobs.github.com/positions/0d9470d0-a4ce-4b2c-9dfd-aa105a15d94e",
created_at: "Sun Feb 24 22:02:44 UTC 2019",
company: "Opslock, Inc",
company_url: "https://opslock.com",
location: "Montreal, Canada",
title: "Javascript Developer (React/NodeJS)",
description: " We're looking for someone who lives and breathes Javascript. This
solving developers building scalable web applications in React, React-Native, and
Consider yourself a React developer with experience building scalable web appl
Designed & Developed API's with NodeJS and Express (REST, JSON, RESTFull s
Understand the importance of test-driven development and have written automate
We have funding, great architecture, and are looking for someone who can hit t
  Dog-friendly office in the Mile End, we offer a flexible schedule,
Looking forward to meeting you! ",
how_to_apply: " Apply at <a href="http://bit.ly/opslock-jsdev">http://bit.ly/ops
company_logo: "https://jobs.github.com/rails/active_storage/blobs/eyJfcmFpbHMiOnsi
-403c6f61eaafc150d4bfe653cea62b67a51191d2/OpsLock%20Transparent%20BG%20Square%20-%
},
- {
id: "845ab1d2-dc26-11e8-8a3c-14f6bbf4ff70",
type: "Full Time",
url: "https://jobs.github.com/positions/845ab1d2-dc26-11e8-8a3c-14f6bbf4ff70",
created_at: "Tue Oct 30 09:31:36 UTC 2018",
company: "Adaptive Financial Consulting Ltd",

I could have entered the URL (API request) in my browser or from my frontend code!
Why do I need a server for that?

https://jobs.github.com/positions.json?description=javascript&location=montreal (without our server)
same result as
http://localhost:4000/jobs?description=javascript&location=montreal (using our server)

1. **Data manipulation**: post-processing, filtering, whatever you want to do with the data before sending it back to the client (some of these you can still do in frontend)
2. Allows you to add some **req.param, req.query** options that the **orig. API doesn't provide**
   a. e.g.**?description=javascript&location=montreal&titleContains=web&year=2019**
3. **CORS (Cross-Origin Resource Sharing)**
   a. If you're writing frontend code (JS), a lot of these external APIs won't allow you to communicate directly **(frontend JS <-> API)**.
   b. So, instead they require you to have a server who would act as a proxy/middleman
      (frontend JS <-> server <-> API)

c4.3 Manipulating results based on user's query

```
// routes/api1.js
...
//?description=javascript&location=montreal&titleContains=developer&year=2019
api1.get("/jobs2", function (req, res) {
   const { description, location, titleContains, year } = req.query;
   let requestUrl = "https://jobs.github.com/positions.json";
   if (description || location) {
      const descriptionParam = description ? `description=${description}&` : '';
      const locationParam = location ? `location=${location}&` : '';
      requestUrl = `${requestUrl}?${descriptionParam}${locationParam}`;
   }
   axios(requestUrl).then(response => {
      // post-processing of results
      const results = response.data;
      const matches = results.filter(r => {
         const { title, created_at } = r;
         const titleParam = titleContains ? titleContains.toLowerCase() : '';
         const yearParam = year || '';
         return title.toLowerCase().includes(titleParam) && created_at.includes(yearParam);
      });
      res.json(matches);
   })...
```

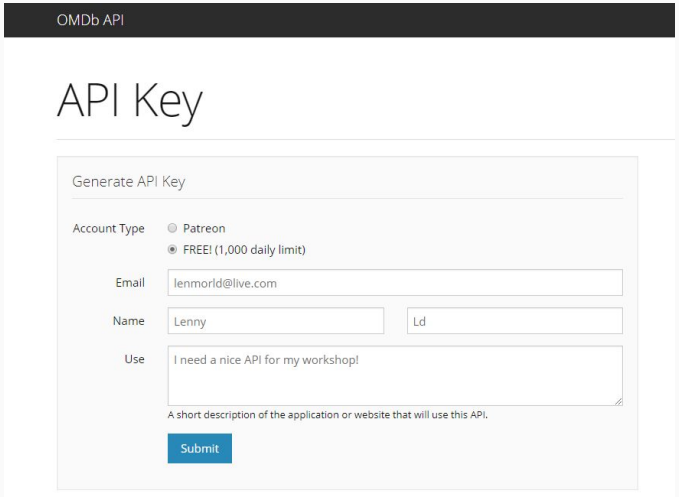
Testing

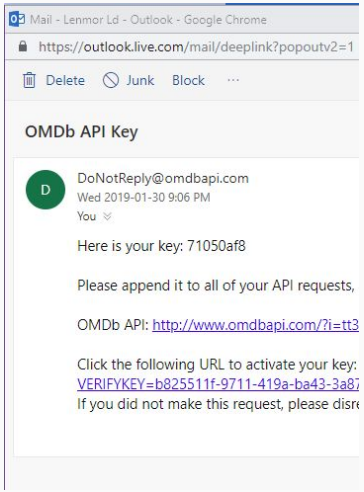

localhost:4000/api1/jobs2?location=montreal&description=javascript&year=2019&titleContains=developer
[
- {
id: "0d9470d0-a4ce-4b2c-9dfd-aa105a15d94e",
type: "Full Time",
url: "https://jobs.github.com/positions/0d9470d0-a4ce-4b2c-9dfd-aa105a15d94e",
created_at: "Sun Feb 24 22:02:44 UTC 2019",
company: "Opslock, Inc",
company_url: "https://opslock.com",
location: "Montreal, Canada",
title: "Javascript Developer (React/NodeJS)",
description: " We're looking for someone who lives and breathes Javascript. This person will be joining a team of pas
solving developers building scalable web applications in React, React-Native, and NodeJS. You should apply if yo
Consider yourself a React developer with experience building scalable web applications (React-Native experience is
Designed & Developed API's with NodeJS and Express (REST, JSON, RESTFull services). Know your way arou
Understand the importance of test-driven development and have written automated testing (Jest, Mocha, etc.)
We have funding, great architecture, and are looking for someone who can hit the ground running and join us for thi
  Dog-friendly office in the Mile End, we offer a flexible schedule, tons of autonomy and ownership, gym m
Looking forward to meeting you! ",
how_t
compa
-403c
}
]
localhost:4000/api1/jobs2?location=new+york&description=javascript&year=2019&titleContains=web
[
- {
id: "e8639715-c4da-4946-8519-4e7c5439f8d2",
type: "Full Time",
url: "https://jobs.github.com/positions/e8639715-c4da-4946-8519-4e7c5439f8d2",
created_at: "Wed Jan 23 16:13:18 UTC 2019",
company: "Datto",

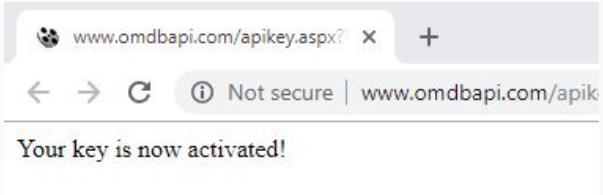
Using APIs that requires an **API key**

we will use: *omdbapi.com, Giphy API*

To **get an API key**, go to the API's website/documentation and follow instructions. Usually it involves signing up for an account/providing email.

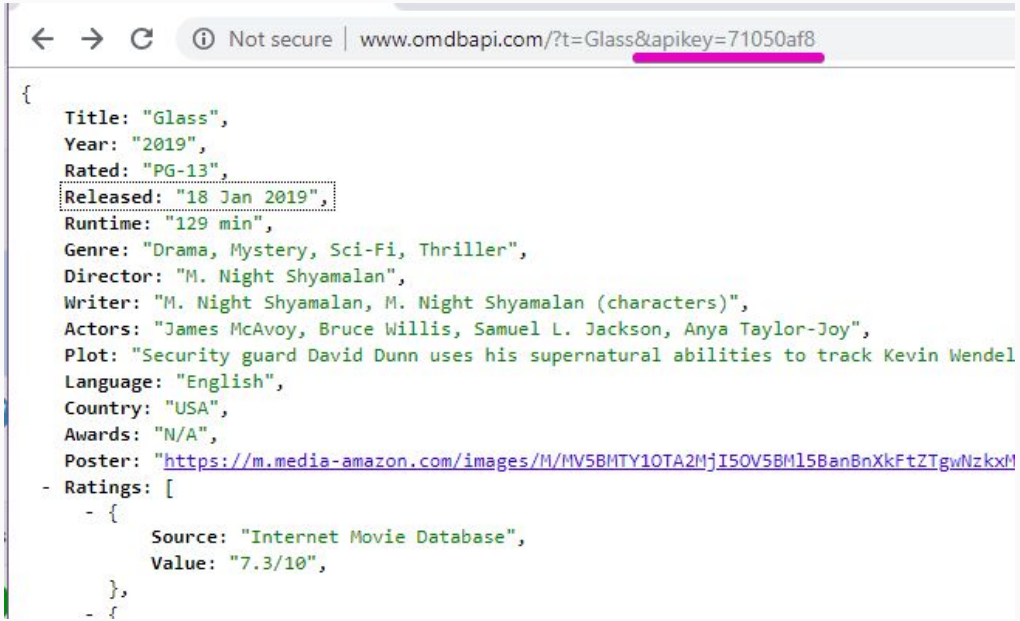






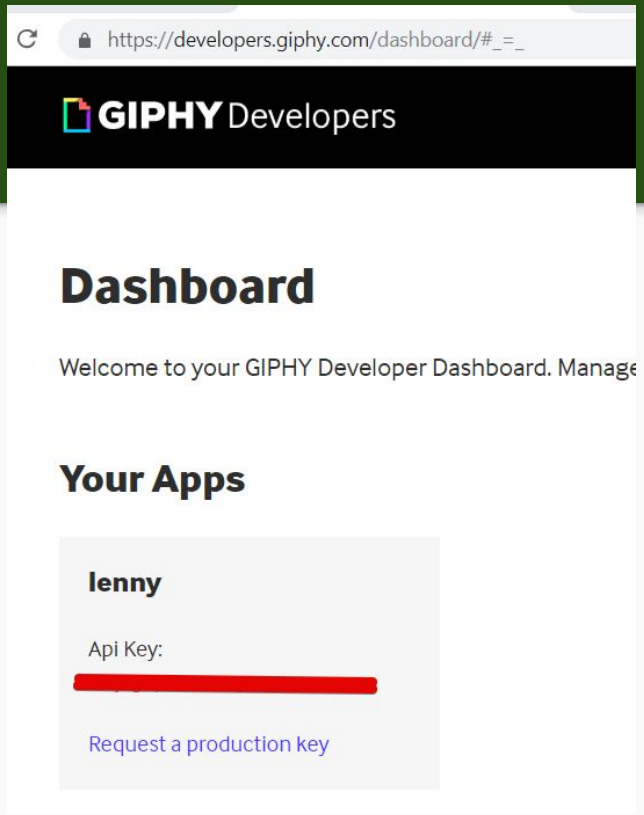
Testing API key

API key is usually attached as a param to the HTTP request.
Always follow the API's instructions on how to do this, since it varies everytime.



Bonus: Using the Giphy API

1. Login to https://developers.giphy.com/
2. Create an app
3. Get API key

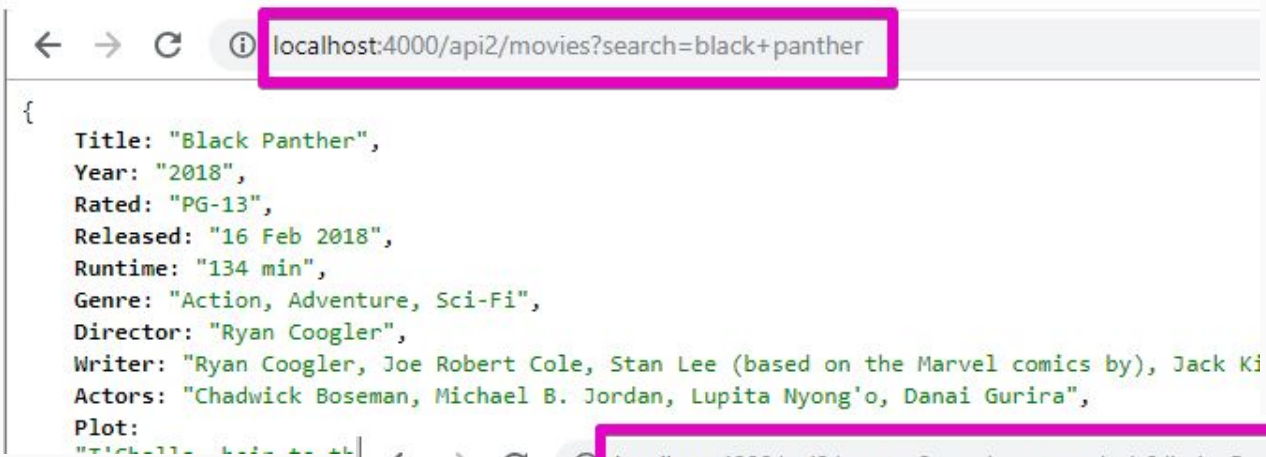


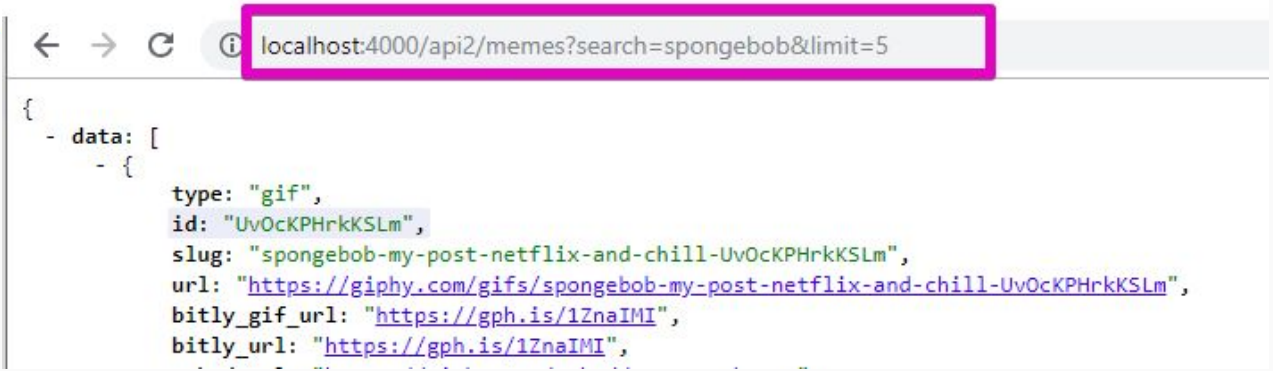
c4.4 adding new Router() and using **API key**

```
// routes/api2.js
...
const OMDB_API_KEY = <GET_YOUR_KEY>;
const GIPHY_API_KEY = <GET_YOUR_KEY>;
// /movies?search=black+panther
api2.get("/movies", (req, res) => {
   const search = req.query.search;
axios(`http://www.omdbapi.com/?t=${search}&apikey=${OMDB_API_KEY}`)
.then(response => {  res.json(response.data);
...
api2.get("/memes", (req, res) => {
   const { search, limit } = req.query;
axios(`http://api.giphy.com/v1/gifs/search?q=${search}&api_key=${GIPHY_API_KEY}&limit=${limit}`)
.then(response => {  res.json(response.data);
...
```

```
// server.js
...
const api2 = require('./routes/api2');
...
server.use('/api2', api2);
```

TESTING


localhost:4000/api2/movies?search=black+panther
{
Title: "Black Panther",
Year: "2018",
Rated: "PG-13",
Released: "16 Feb 2018",
Runtime: "134 min",
Genre: "Action, Adventure, Sci-Fi",
Director: "Ryan Coogler",
Writer: "Ryan Coogler, Joe Robert Cole, Stan Lee (based on the Marvel comics by), Jack Ki
Actors: "Chadwick Boseman, Michael B. Jordan, Lupita Nyong'o, Danai Gurira",
Plot:

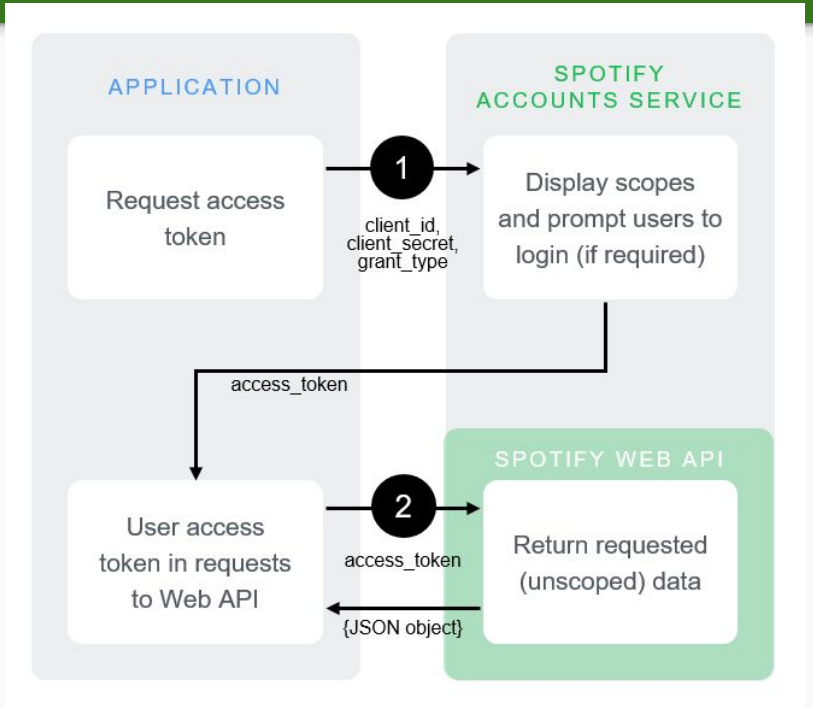

localhost:4000/api2/memes?search=spongebob&limit=5
{
- data: [
- {
type: "gif",
id: "UvOcKPHrkKSLm",
slug: "spongebob-my-post-netflix-and-chill-UvOcKPHrkKSLm",
url: "https://giphy.com/gifs/spongebob-my-post-netflix-and-chill-UvOcKPHrkKSLm",
bitly_gif_url: "https://gph.is/1ZnaIMI",
bitly_url: "https://gph.is/1ZnaIMI",

Using APIs implementing OAuth

A lot of APIs (e.g. Twitter, Spotify, Yelp) uses some form of **OAuth.**

We discuss here a common flow for server-server authentication, called **Client Credentials Flow** or **Access Tokens**. It goes like this:

access tokens are also called **bearer*



Optional: Setting up Spotify API

1. For simplicity, we will use the free Spotify developer account I setup before
   a. You could also setup your own developer account
      https://developer.spotify.com/dashboard

2. Authentication flow is using Client Credentials
   https://developer.spotify.com/documentation/general/guides/authorization-guide/#client-credentials-flow

* For those using my credentials, I will send the Base-64 string (composed of Client ID, Client secret) needed for authentication **through SLACK**

Testing out spotify credentials

Test in Terminal / Postman

```
$ curl -X "POST" -H "Authorization: Basic <base64_string>" -d
grant_type=client_credentials https://accounts.spotify.com/api/token
> {"access_token":".....","token_type":"Bearer","expires_in":3600,"scope":""}
```

We then use this access_token to do requests like

```
$ curl -H "Authorization: Bearer <access_token>"
"https://api.spotify.com/v1/search?query=gangnam%20style&type=track"

> { literally tons of data }
```

Our webapp server's job:

1. Prepare the **request**, including **encoding query params**
2. Process the **response**, which includes **filtering, preparing data for render in React**

c4.5 requesting for an access_token

```
# qs allow proper data encoding for HTTP
$ npm install qs
```

routes/api3.js

Define **getAccessToken()** which returns a Promise

- Prepare request config containing the base-64 encoded credentials in the header
- Send request and resolve Promise with *access_token* returned by Spotify

Setup route `'/songs'`

- Define .then(), send access_token for now to test

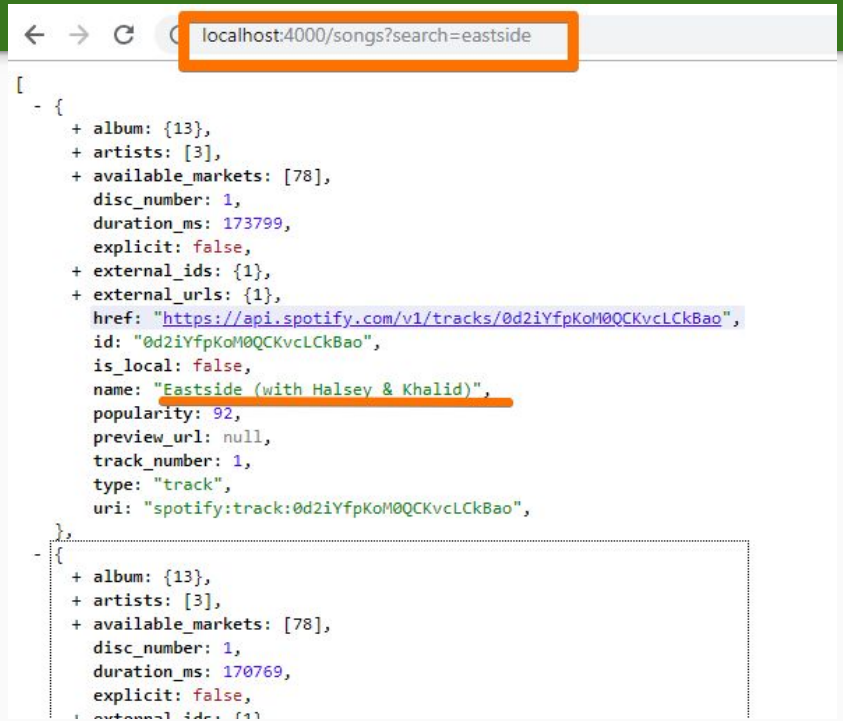
see code change at: https://github.com/lenmorld/node_workshop/compare/c4.4...c4.5

c4.6 GET request to search query string

We could now use the access_token to make a request

see full code change at: https://github.com/lenmorld/node_workshop/compare/c4.5...c4.6

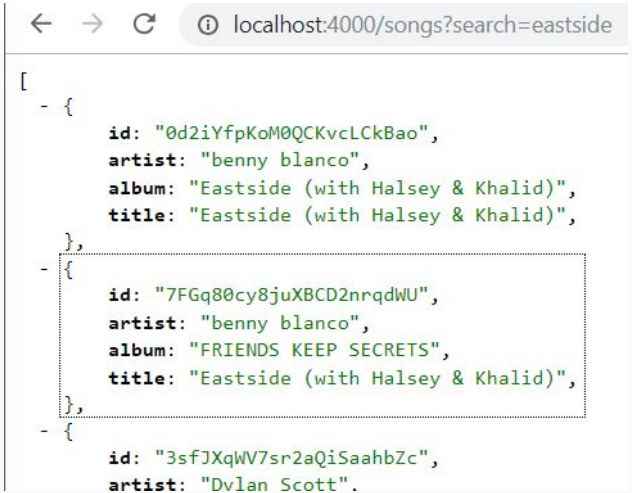
```
api3.get("/songs", (req, res) => {
   const search = req.query.search;
   console.log(`[SPOTIFY] : searching ${search}...`);
   getAccessToken().then( access_token => {
      const _url =
`https://api.spotify.com/v1/search?query=${search}&type=track`;
      axios({
         method: 'GET',
         url: _url,
         headers: {
            "Authorization": `Bearer ${access_token}`,
            "Accept": "application/json"
         }
      }).then( _res => {
         console.log(`search response: ${JSON.stringify(_res.data)}`);
         res.send(_res.data.tracks.items);
...
```



c4.7 “massaging” API’s response data using .map()

So far, this is the item format we have been using:

The object returned by Spotify is **too big**. We only need: **id, artist, title, album**.




```
api3.get("/songs", (req, res) => {
...
).then(function (_res) {
  // "massage" data so we only get the attributes we need
  const search_results = _res.data.tracks.items;
  const squashed_results = search_results.map( track => {
            return {
                id: track.id,
                artist: track.artists[0].name,
                album: track.album.name,
                title: track.name
            };
        });
  console.log(squashed_results);
  // res.send(_res.data.tracks.items);
  res.json(squashed_results);
});
...
```

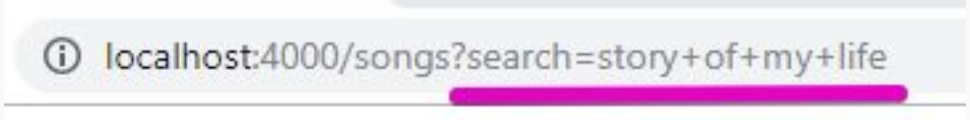
Testing:

Test our own API endpoints (which uses external API) to query jobs, songs, movies

How about querying spaces in URL?

💡 Use + instead of spaces.

Note that encoding stuff like these are
taken care of usually in frontend code, e.g. form inputs.



m3 (Milestone 3)
Integrating with Spotify to Playlist App

https://github.com/<username>/node_workshop/compare/m3...c4.7
Building up from m2, do the following:

- add a Find page that includes a form for searching Spotify tracks
- execute a POST `/playlist/search` route that uses the `req.body`
- provide a UI to load Spotify search results, and allow adding to Playlist
- refactor track searching into a Promise, such that both `/songs` and `/playlist/search` routes call the same function to search

BONUS m4: styling your template pages

https://github.com/<username>/node_workshop/compare/m3...m4

To style our server-rendered EJS template pages:

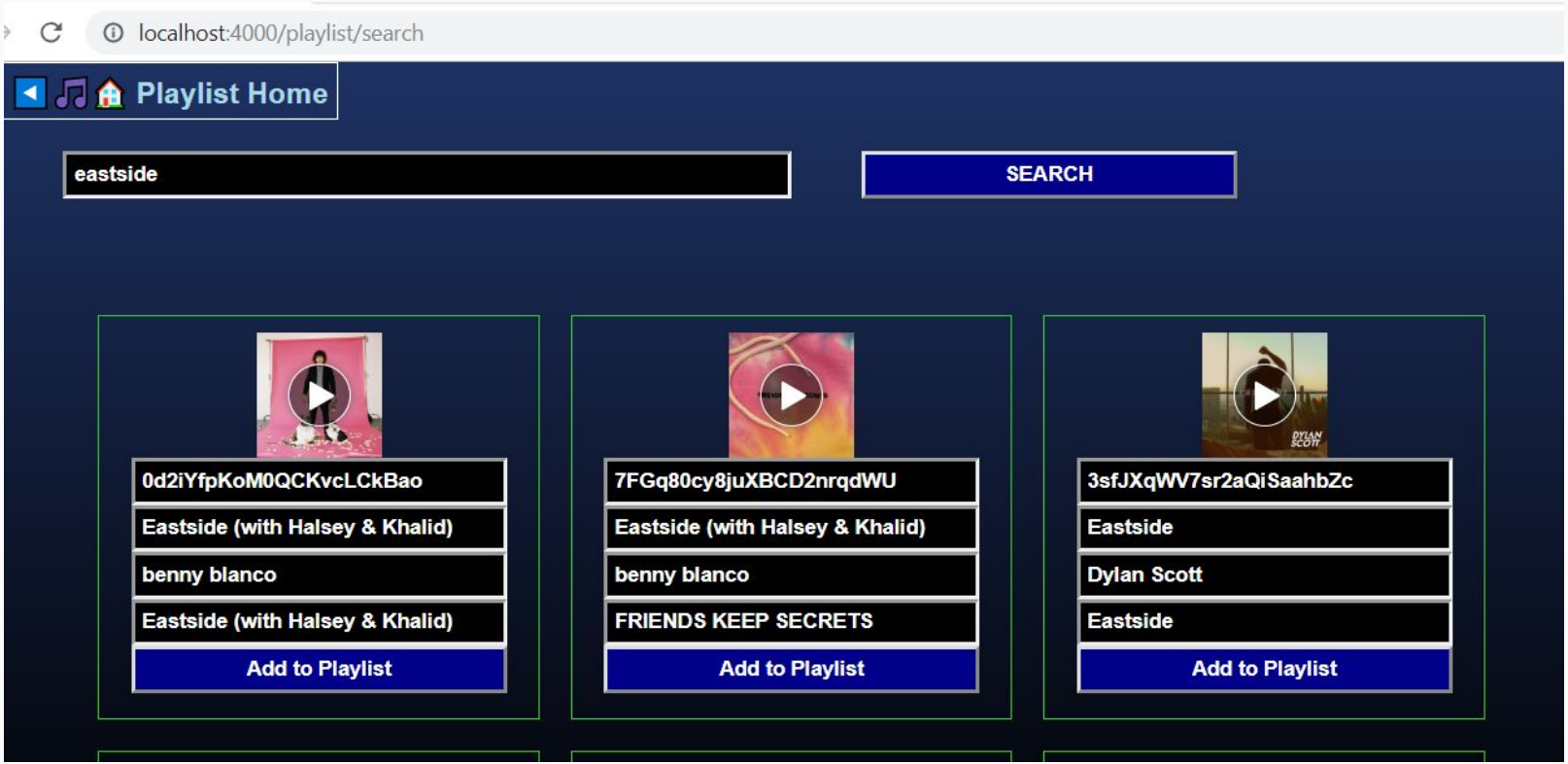
1. Create *public/styles.css*, and put all the CSS styles there
2. From *server.js*, serve the *public* folder with *server.use()*
    a. The **public** folder is the place to put all static files: Javascript, CSS, images

```
// server.js
...
server.use(express.static(__dirname + '/public'));
...
```

3. You can now use *styles.css* in your .ejs, .html files

```
// playlist/home.ejs
// → note it is important to use relative path /styles.css so it works in all the views, even the nested ones
<link href="/styles.css" rel="stylesheet" type="text/css">
...
```

Final project with Styling (and Emojis!) □🎨



We finished the material!

Questions?
Need help?

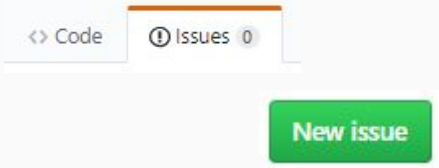
Coming up… Advanced Stuff and topics to explore

I need a little help...

Please answer a short survey about the workshop
https://goo.gl/M6Bdc3

Please go to the code repo, and give it a star
(if it deserves one, of course!)
https://github.com/lenmorld/node_workshop

Any mistakes or improvements, please submit an issue
or even better, contribute to open-source! (ping me in Slack for details)



Appendices

To reduce the complexity of these advanced topics,
the branches (e.g. a1, a2...) are independent from the previous branches c1...m4
They are self-contained and minimal
So it is better to view them independently as a branch, or use the github links given in the following chapters

Appendix 1

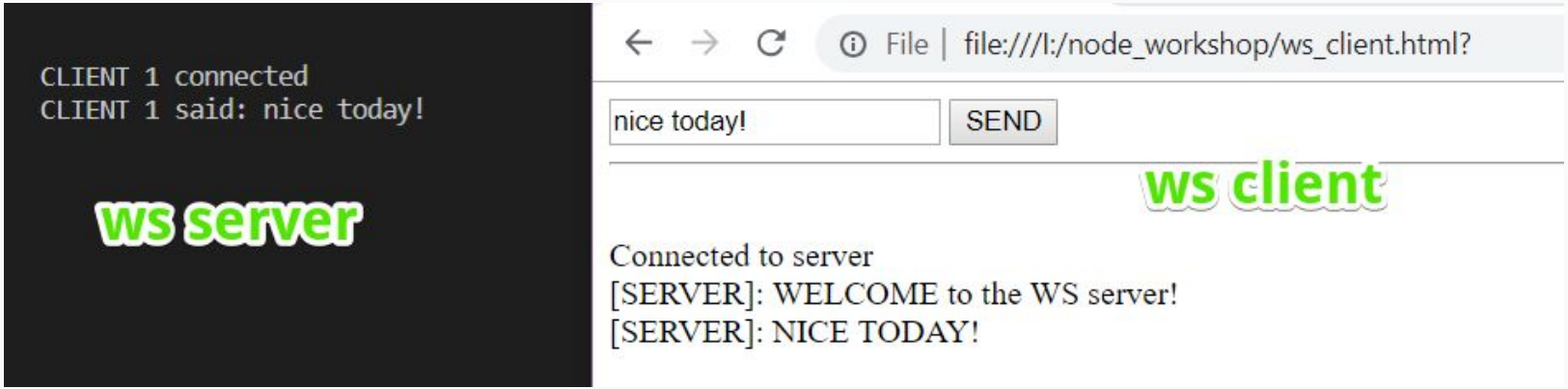
Intro to WebSockets for real-time apps

a1.1 Basic Client-Server

https://github.com/<username>/node_workshop/tree/a1.1

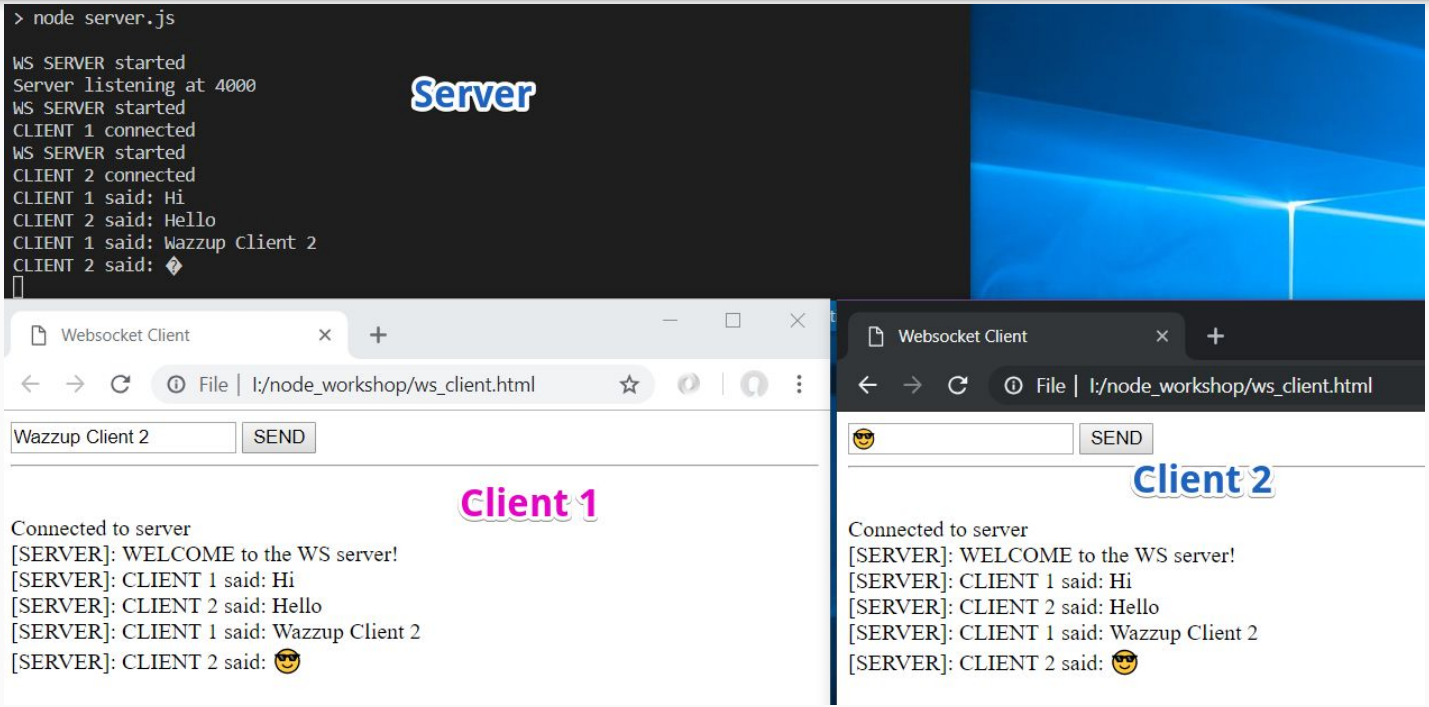
Websockets allow connection-oriented, bi-directional communication between client and server (in contrast to HTTP). In this example, the server CAPITALIZES whatever the client says and returns it.

TO TEST: Open file ***ws_client.html*** in a browser to serve as client



a1.2 Multi-client Chat Room

https://github.com/<user_name>/node_workshop/compare/a1.1...a1.2



Appendix 2

Intro to User Auth Implementation

- **Session-based using cookies**
- **Token-based using JWT**

Authentication Systems

- allows a user to access a resource only when user's identity is confirmed, e.g. by comparing the supplied credentials with the ones stored in the backend (usually a database)
    - To simplify here, we just use in-memory data. A good challenge is to persist this in the database as well using knowledge from Chapter 3
- can be **session-based** or **token-based**
- Usual scenario: when verifying user's identity when they login, use either a cookie, a JWT, or a more complete approach using OAuth

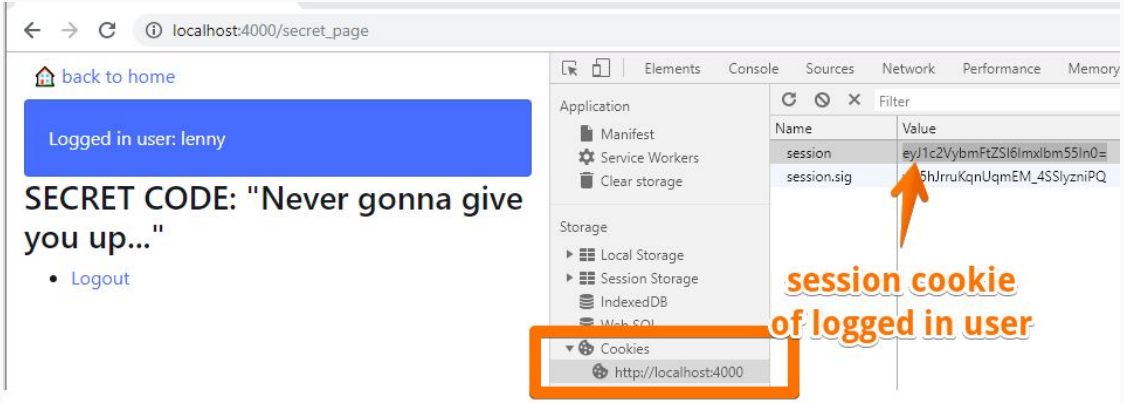
a2.1 Session-based Auth

AKA Cookie-based auth

- Traditional way of auth with sessions + cookies. It goes like this:
  1. When user logs in successfully on */login*, the *req.session.username* is set to the current user
  2. *req.session.username* must be set when accessing */secret_page* route
     a. otherwise, user is redirected to */login*
  3. on */logout*, *req.session.username* is cleared
  4. Users are also allowed to register

Branch code: https://github.com/<username>/node_workshop/tree/a2.1

Session/Cookie - based authentication



a2.2 Encrypting password

using bcrypt

Notice that we are storing user's password in-memory in **plaintext!** 👀👀👀

- A quick fix is to **hash** the password and save the hash.
- Since the hash is "one-way" (impossible to get password from hash), we can only compare if the **hash** of incoming password is similar to our saved hash.
- We also use a **salt** to add randomness, thus preventing brute-force attacks
- ***Bcrypt*** simplifies the hashing and comparing for us

Code changes:
https://github.com/<username>/node_workshop/compare/a2.1...a2.2

a2.3 Token-based Auth
using JWT

- Allows for “truly RESTful” or **stateless** auth, since no need to keep session data in server
- JWT (JSON Web Token) is a signed piece of data, sent along with every request
- It is used by the receipient to verify the sender’s authenticity

Code changes:
https://github.com/<username>/node_workshop/compare/a2.2...a2.3

more about authentication...

Some of the more advanced topics include:

- Storing tokens in ***localStorage***
- OAuth2 - a standard for token-based authentication and “access delagation”
- Passport.js - auth library that includes lots of reusable **strategies**
- Login with social media such as Github, Facebook, Google,Twitter, etc

...currently in the process of writing material for this.

Ping me on Slack if you need some help on this 😉

Appendix 3

Getting better at Node.js

Resources and tips to grasp the basics of Node.js and server web dev

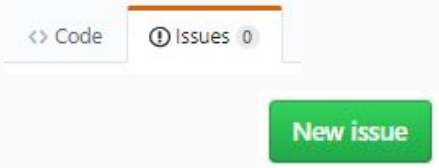
- For a more comprehensive guide on Node+Express:
  - MDN's Express web framework tutorial: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs
- For a deep dive on Node
  - Udemy course *All About Nodejs:* https://www.udemy.com/all-about-nodejs/
- For a quick refresh on Javascript
  - MDN's re-introduction to JS: https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

I need a little help...

Please answer a short survey about the workshop
https://goo.gl/M6Bdc3

Please go to the code repo, and give it a star
(if it deserves one, of course!)
https://github.com/lenmorld/node_workshop

Any mistakes or improvements, please submit an issue
or even better, contribute to open-source! (ping me in Slack for details)



Thank you!!!