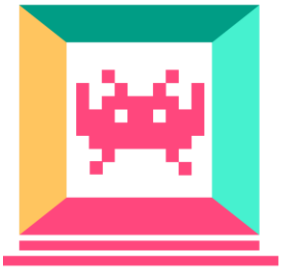


Learn and Master Node.js

By Ben Sparks in collaboration with Pablo Farias Navarro

Table of Contents

Part 1: Complete Beginner's Guide to Node.js

What is Node.js?

Node.js is a c++ engine built to manipulate OS systems that is scripted using JavaScript. The JavaScript engine is called V8, created by Google, and is the same engine that is used in Chromium (the browser technology powering Google Chrome). In simpler terms, it's a tool to write server side applications using javascript.

Often there can be confusion about what Node.js really is because in some ways it is very similar to Ruby, Python, PHP, LUA, or even Erlang. They are all c or c++ engines with bindings to a scripting language that can be JIT (Just in Time) compiled. Unlike those however, Node.js isn't also the name of the scripting language, but instead creates bindings to javascript.

Tutorial source code

You can download the scripts from this tutorial here.

Installing Node.js

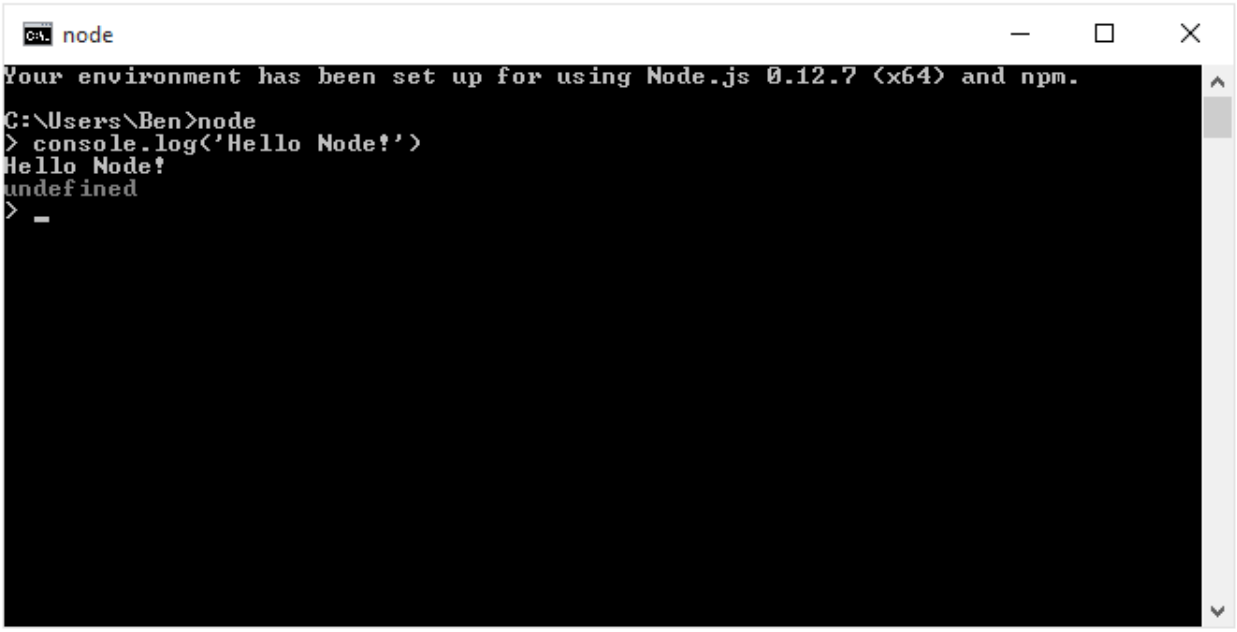
The easiest way to install Node.js is to just click on the "Install" button on the home page or grab the pre-built installer or binary from the downloads page.

Up and Running

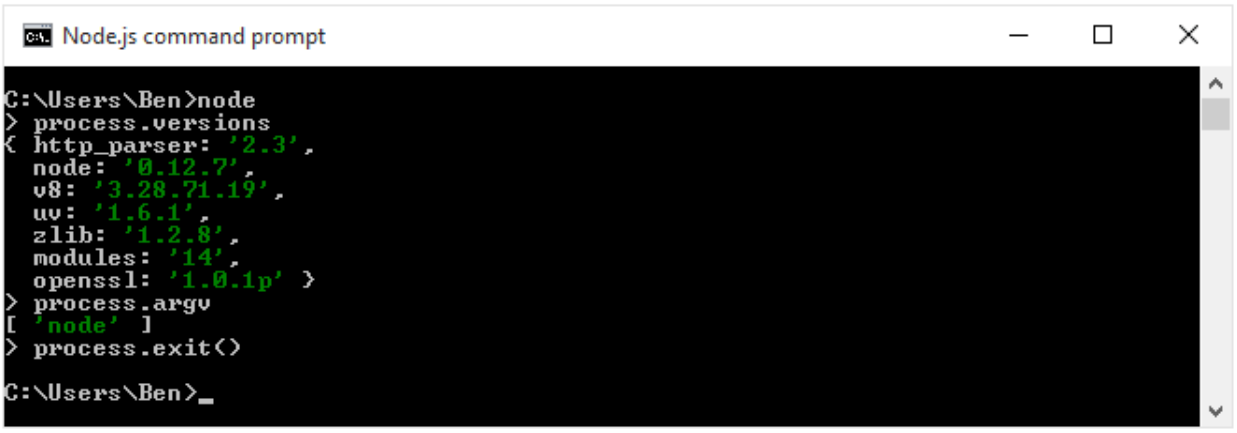
Once you've installed Node.js you should have a command `node` available in your terminal or command prompt. If you are on Windows, it will also install a Node.js Command Prompt window into your start menu. When you start the node executable you will be given a command prompt where you can type JavaScript code to be evaluated. This is very similar to the console that is found in the browser developer tools.

It's important to note, that unlike the browser, there is no DOM or even window object to work with. You will find some familiar things such as `console.log` and `setTimeout` that are not part

of the actual JavaScript language, but for the most part it's core language that you are working with.



What node does come built in with instead of the window object is an object called "process". In fact, if you need to exit the node.js REPL (Read Eval Print Loop), then you need to call `process.exit()`. In case that it isn't obvious to you why there is no window object, well, it's because there is no window! Only a process for the executeable that is managed by your OS. There are many useful attributes and methods attached to the process object such as node version, the version of the v8 javascript engine, and environment variables.

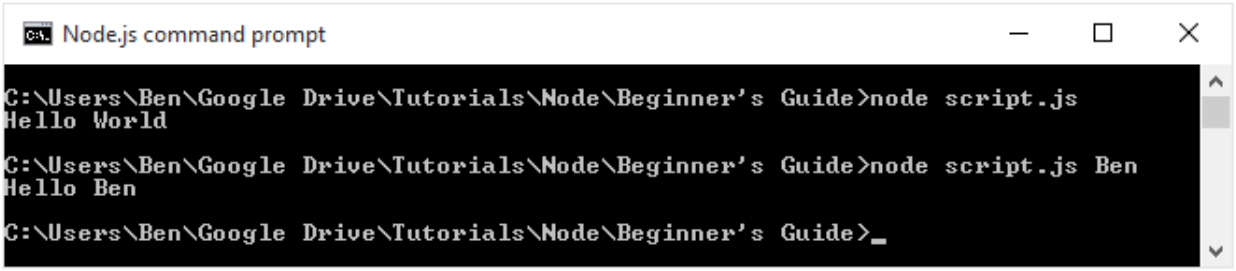


For now you can play around typing in javascript like `console.log('Hello Node!');` but what you'll want to do most of the time is load external js files and modules.

External Files & Modules

Node can load external js files, typically you will load a js file to start up your application. Create a file called “script.js”, and put the following code inside.

```
var who = 'World';
var greeting = 'Hello';

who = process.argv[2] || who;

console.log(greeting + ' ' + who);
```



Node.js uses a CommonJS module loading system. Instead of putting all of your code into a single js file you can load separate files using the `require` keyword. This is similar in a way to loading files using the script tag in the browser, except that the modules that are required are loaded from within other javascript files, and the script is not executed against the global context (window).

There are many modules built in to the node executable for access to the low level things such as networking and accessing the file system. When you are working with a built in module, you can simply `require` it. For those developed by third parties, you need to download the file first. There are thousands of node modules out there, and so there is another program, NPM (Node Package Manager) that is designed to help find and load them.

Common Uses

Ultimately you can build absolutely any type of application you want using Node.js as the core. Since you have the full power of integrating low level c++ modules there are no restrictions. Developers have built applications to control drones, MMO Game servers, REST APIs, web browsers, you name it. Some of the most common applications that you'll find (and likely want to

use Node.js for yourself) are HTTP Servers and build tools for software development (particularly web / javascript development).

HTTP Server

The absolute simplest HTTP server code is listed right on the home page of nodejs.org. While the total basics are great, of course you're going to want to do something a little more complicated and practical. Fortunately, there is a very popular library known as Express that is meant to give you all the power of a complicated web server, but still make it simple to setup and get running.

To install Express all it takes are a few simple steps. Create a folder for your project, or go into an existing folder. If there isn't already a node application initialized there, you can create one using `npm init`. The tool will prompt you through a series of questions, you should be able to just use the defaults for now, and then it will create a package.json file in your folder. This file serves as the manifest for your application. Once your project has been initialized for node, then run:

```
1 npm install express --save
```

This will not only install the express package in the node_modules folder, but adding the `--save` flag will add the entry to your package.json file as well. This is important because you can easily distribute your application and simply reinstall any dependencies, this way you don't have to store any of your dependencies in source control either. Later, when you want to install, you can simply run `npm install` inside the folder, it will use the dependencies listed in the file.

When developing even simple web pages with HTML5 features, you often need to host them via a web server rather than simply opening the index.html file in your browser. Ajax requests and other advanced features are blocked from scripts running locally by the browser's security sandbox. It's easy using Express to setup a simple static file server.

```
1 var express = require('express'),
2     app = express();
3
4 app.use(express.static('public'));
5
6 app.listen(3000);
```

Those lines will setup express to serve any files within the "public" folder (note, the folder relative to the script). Any html, images, css, json, txt, etc will all be accessible via

`http://localhost:3000` . The default page that it will look for is index.html. Now you can load things locally without running into cross domain issues (say if you wanted to open a json file locally).

Express is also handy to setup a REST API for your front end development projects. It's possible to host your back end API of course, but it's not always the case. Often times if you are a front end developer, the service API may not be developed by you, or even using Node.js. This doesn't have to slow down your development however, you can setup a mock server using Express relatively quickly and easily.

```
var foo = {
    data: [1, 2, 3],
    bar: 'baz'
};

app.get('/api/stuff.json', function(req, res) {
    res.send(foo);
});
```

You can of course respond to all of the HTTP verbs (GET, POST, PUT, DELETE, etc) as well as use query string parameters and post payloads. This way you can setup the client code to expect a certain API and simply swap out the host. You can also help out the service developer by letting them know what you are expecting on the front. With Express and Node.js you can create a fully featured and robust back end API and also do more traditional server side page rendering. It just depends on how far you want to take it.

Summary

Node.js is such an incredibly powerful tool with infinite possibilities, that we've just scratched the surface of what you can do. If you like JavaScript programming there is no better choice available right now to solve problems.

Part 2: How to Create a REST API with Node.js and Express

In this tutorial we will create a RESTful API to access leaderboard information. We'll be able to track players and scores for an unlimited number of boards. Whether you are developing a game or even just a website, chances are that you will want some way to work with your data

---

over HTTP. I'm going to show you how you can setup a server for this using node.js and express.

What is a REST API?

Chances are that if you're interested in creating one, that you are familiar at least a little bit with what they are. If however, you are not, I will take a brief moment now to explain them to you.

Sadly, a RESTful API is not going to help you get better sleep at night. REST stands for **Representational State Transfer** and it is the way that web browsers communicate.rest_joke The HTTP protocol uses a set of verbs (GET, POST, PUT, DELETE, etc) to manipulate various resources. Endpoints in a RESTful system represent specific resources. Instead of having something like "/api/getUserById?id=1234" which represents an action, we can say "/api/users/1234" represents a specific user. We use the HTTP verbs such as GET to define the action that we want to take on a particular resource.

One of the big advantages to doing it this way is that you provide consistent behavior for applications that will consume your API. You don't have to provide them extensive documentation with your specific domain language, instead they can simply use HTTP as it was intended.

About the source code

All of the source code to this tutorial is available here: Tutorial Source Code

Throughout the tutorial I will be using features from ES2015 (ES6) which is the latest version of JavaScript. If you are not familiar with these features, I can recommend this free ebook Exploring ES6. I'm only using features that are available in the latest stable version of node.js (as of this writing), nothing that needs to use a 3rd party "transpiler".

While we will be exploring how to create a REST API using node.js, I won't be covering the basics of using node.js. If you are unfamiliar with those then you may want to check out one of my other tutorials the Beginner's Guide to Node.js or Zenva also has a video course available: Node.js for Beginners.

Node.js online course

Check out Node.js from Zero to Hero on **Zenva Academy** if you are after a comprehensive Node.js online course that can get you project-ready with this awesome tool.

First Steps, Use the Verbs

First things first, lets create a folder for our project and intitialize it for node.js using npm.

```
npm init
```

Express is a web framework for node, and thus it can also be installed via npm.

```
npm install --save express
```

Express is a web server framework that provides a much higher abstraction from the raw HTTP communication available in node that makes it easy to host a web server. We'll start by creating the "Hello World" of web servers, a simple one that doesn't even host a web page, just says hello. Every route in Express is a function that gets passed the request and response as arguments. In this simple example, we are just using the response to send out a plain text string. It is also capable of sending complete html pages, json data, and more.

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

app.get('/', function (req, res) {
  res.send('Hello World!');
});

var server = app.listen(3000, function () {
  var host = server.address().address;
  host = (host === '::' ? 'localhost' : host);
  var port = server.address().port;

  console.log('listening at http://%s:%s', host, port);
});
```

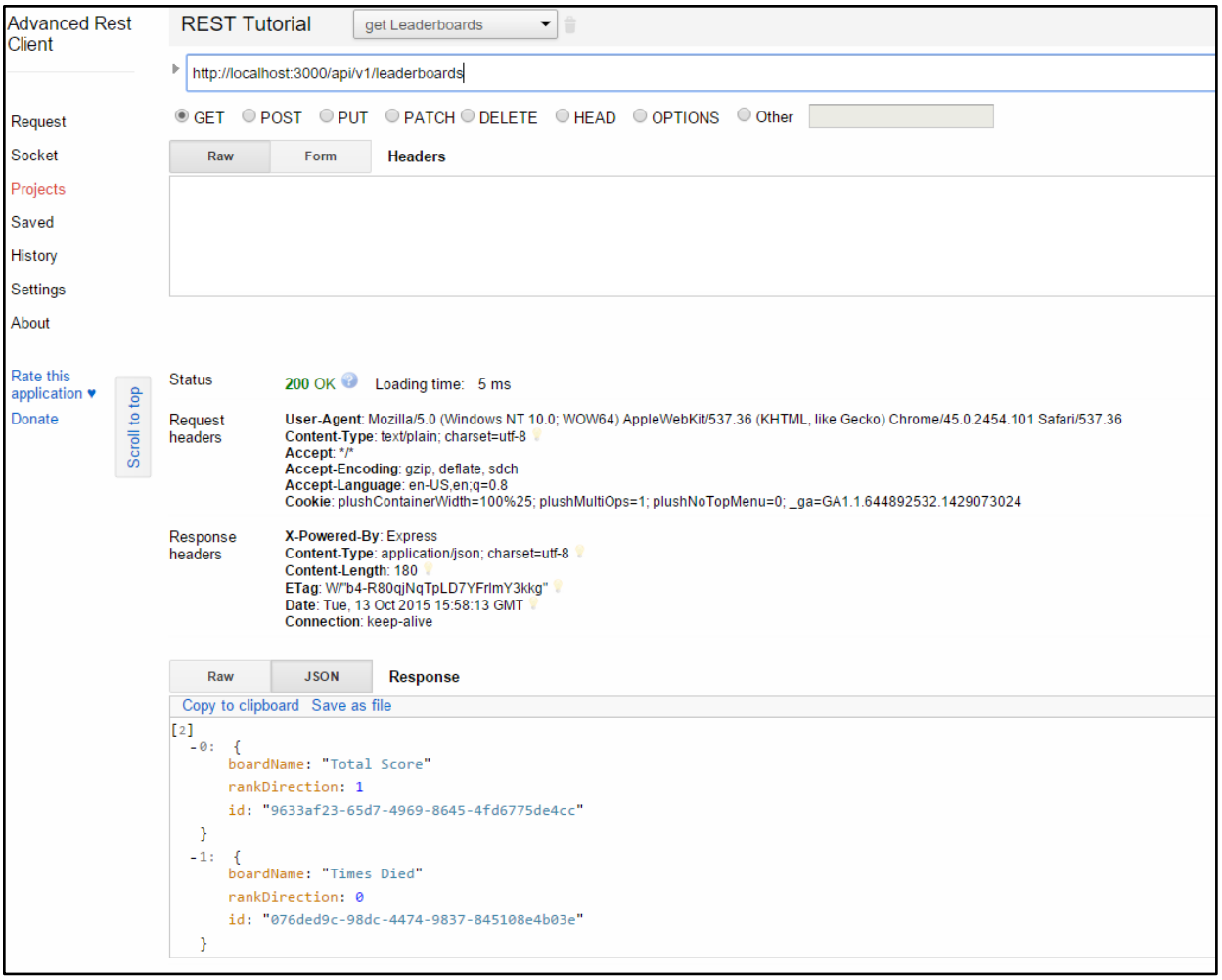
Now we can begin with REST with the most common verb, GET. We need to serve resources, so we'll do just that, create an array of objects with some arbitrary data in them. We've already seen ***get*** even in the first example, our resource was just a string "Hello World". This time we'll get a little more complicated, when we request the `/resources` url, we'll return the entire array. Using path variables, we can also request a specific resource (in this case by id). Path variables are automatically placed into the params object of the request. Then we can simply use the value of that variable to retrieve the element from our array and return it. If we don't find the item in our array, we should send a 404 code, the standard HTTP code for "Not Found". Now if we access our site using `/resources/1` we should get our item.

```
var resources = [
    {
        id: 1,
        name: 'Foo'
    }
];

app.get('/resources', function(req, res) {
    res.send(resources);
});

app.get('/resources/:id', function(req, res) {
    var id = parseInt(req.params.id, 10);
    var result = resources.filter(r => r.id === id)[0];

    if (!result) {
        res.sendStatus(404);
    } else {
        res.send(result);
    }
});
```

The next most common thing that we want to do with a REST API is create new resources. This is done primarily using the POST verb, the standard verb used with HTML forms to submit data. This request should have the same resource location as the entire collection that we used with GET, because we want to add an item to the collection. The data that we want to use to populate the new item will be located in the body of the request. Here is what our POST request looks like in Express.

```
app.post('/resources', function(req, res) {
    var item = req.body;

    if (!item.id) {
        return res.sendStatus(500);
    }

    resources.push(item);

    res.send('/resources/' + item.id);
});
```

In order to make POST requests, we need some sort of client, we can't just type a url into the browser (that's just GET). For this, you could do several things, one obvious choice is to host a website with a good JavaScript client library like jQuery and make all of the requests through the console (or even write custom scripts). Another option that IMO is much easier, is to use an application like Advanced REST Client which is a Chrome extension (if you don't use Chrome I imagine there are others out there). Like I mentioned earlier we also could create a HTML page with a form on it, but that is not really the way that we access an API, it's the way that we would build a website.



Another thing about POST requests when it comes to Express is that they can be sent in several different formats. The standard html form way to send data is in *application/multipart-form-data* and the Ajax way of doing it is using *application/json*. Because Express doesn't want to make any assumptions about how you consume your requests, it doesn't include a way to parse json automatically built it (the way that we are going to send data). For that we need to install a separate middleware module, the **body-parser**.

```
1 npm install --save body-parser
```

After it is installed we can reference it in our file, and tell express to parse the body of POST requests into json for us. Finally, when we POST to the `/resources` url it should create an item in our array. Once that is done, you should be able to reference it using the id that you posted.

```
var bodyParser = require('body-parser')

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

We also want to be able to update existing records, and for that we use the PUT verb. This one can be a little controversial, however you can also choose to support creating resources using PUT as well. The main idea is that your path is to a specific resource in the collection (by id in our case), if that resource does not exist, you can still consider it like updating an empty resource. In Express we use the ***put*** method and it is essentially the same as using POST. If we create a new resource we'll return status code 201 (Created). If we are updating an existing item we'll return 204 (No Response) because any 200 code is successful, and 204 speaks for itself.

```
app.put('/resources/:id', function(req, res) {
    var id = parseInt(req.params.id, 10);
    var existingItem = resources.filter(r => r.id === id)[0];

    if (!existingItem) {
        let item = req.body;
        item.id = id;
        resources.push(item);
        res.setHeader('Location', '/resources/' + id);
        res.sendStatus(201);
    } else {
        existingItem.name = req.body.name;
        res.sendStatus(204);
    }
});
```

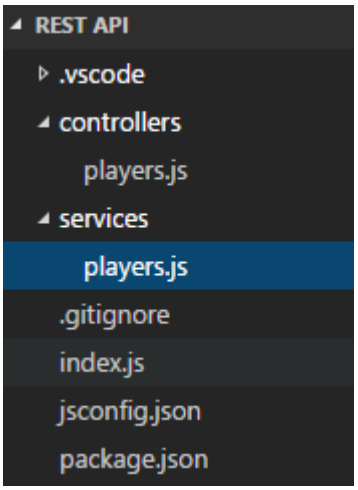
Finally we want to be able to remove records from the collection. For this we'll use the DELETE verb. In Express the method is also ***delete***, in this case I'm calling it using bracket notation because IDEs tend to dislike the use of the delete keyword. Delete is the same path as the GET request, if the item doesn't exist, we return 404 (Not Found), otherwise we remove it from our array and return 204 (No Response) indicating that it has been done and we have nothing else to say about it.

```
app['delete']('/resources/:id', function(req, res) {
    var id = parseInt(req.params.id, 10);
    var existingItem = resources.filter(r => r.id === id)[0];

    if (!existingItem) {
        return res.sendStatus(404);
    }

    resources = resources.filter(r => r.id !== id);
    res.sendStatus(204);
});
```

Well, there you have it, a complete REST API… Ok, well that was a simple and useless one, let's build a more real world example now, a Leaderboard API for games.

Creating a Leaderboard API

Writing all of your code in a single js file is fine for simple demos, but nothing that you want to do in real practice. The first thing that we'll want to do to build a more serious API is to create a folder structure to separate our code into controllers and services. Services will handle the business logic (validation, etc.) and the actual manipulation of data, and controllers will handle all of the interfacing with the HTTP API. Even though we are going to build a more complex API than our first example, I'm still not going to hook it up to a database this time. Because our services are separated from the controllers, we can more easily swap out services that interact with a database rather than in-memory data structures later, all we have to do is keep the same service API.



There are three main objects that we'll need to create for our leaderboards: Boards, Players, and Scores. Each board will track a ranking for a particular aspect of a game. It might be as simple as just "High Score" or it could be "Most Kills" or in the case of a golf game lowest score is better, and we can have a leaderboard for each hole as well as the entire match. We need players of course to know the name of who achieved the scores. Of course we also need to track scores, they are what we use to rank the players on the leaderboard.

Let's start with the players, we'll create a service class that will handle a collection (array) of players, and all of the methods needed to manipulate that collection. Each element in the collection needs to have a unique ID that we can use to look it up, if we were using a database, this would be some sort of GUID (globally unique identifier). Luckily, a package for node exists that can generate these for us: node-uuid. We can install it using npm, `npm install --save node-uuid` and then we'll reference it in our module. All of the methods in the service should be fairly self explanatory, we have something for each of the verbs: GET, POST, PUT, and DELETE that the controller will be using. We only want there to ever be one instance of our player service, so instead of exporting the class itself, we'll export a new instance of it. Module files in node are only ever executed once, so this effectively gives us a singleton.

```
'use strict';

var uuid = require('node-uuid');

class PlayersService {
    constructor() {
        this.players = [];
    }

    getPlayers() {
        return this.players;
    }

    getSinglePlayer(playerId) {
        var player = this.players.filter(p => p.id === playerId)[0];

        return player || null;
    }

    addPlayer(info) {
        // prevent a bit of bad/duplicate data
        if (!info || this.players.filter(p => (p.firstName === info.firstName && p.lastName === info.lastName)).length > 0) {
            return false;
        }

        info.id = uuid.v4();

        this.players.push(info);
        return true;
    }

    updatePlayer(playerId, info) {
        var player = this.getSinglePlayer(playerId);
        if (player) {
            player.firstName = info.firstName ? info.firstName : player.firstName;
            player.lastName = info.lastName ? info.lastName : player.lastName;
            player.displayName = info.displayName ? info.displayName : player.displayName;

            return true;
        }
        return false;
    }
}

module.exports = new PlayersService();
```

Next we'll build the controller for the API endpoints for the players. The controller will use a reference to the service, like I mentioned earlier, this makes it easy to swap out a completely different service as long as it follows the same interface.

In the constructor of our controller class we are going to take a reference to an Express Router object that we will hook our endpoints on to. In the registerRoutes method we are going to bind our REST verbs to the appropriate endpoints and corresponding class methods. Inside the class methods you will see that the code is very similar to our earlier example with "resources", we access the service to actually manipulate data, and these methods just handle the request and response.

```
'use strict';

var PlayersService = require('../services/players');

class PlayersController {
    constructor(router) {
        this.router = router;
        this.registerRoutes();
    }

    registerRoutes() {
        this.router.get('/players', this.getPlayers.bind(this));
        this.router.get('/players/:id', this.getSinglePlayer.bind(this));
        this.router.post('/players', this.postPlayer.bind(this));
        this.router.put('/players/:id', this.putPlayer.bind(this));
    }

    getPlayers(req, res) {
        var players = PlayersService.getPlayers();
        res.send(players);
    }

    getSinglePlayer(req, res) {
        var id = req.params.id;
        var player = PlayersService.getSinglePlayer(id);

        if (!player) {
            res.sendStatus(404);
        } else {
            res.send(player);
        }
    }
```

```
33
34      putPlayer(req, res) {
35          var id = parseInt(req.params.id, 10);
36          var existingPlayer = PlayersService.getSinglePlayer(id);
37
38          if (!existingPlayer) {
39              let playerInfo = req.body;
40              playerInfo.id = id;
41              if (PlayersService.addPlayer(playerInfo)) {
42                  res.setHeader('Location', '/players/' + id);
43                  res.sendStatus(201);
44              } else {
45                  res.sendStatus(500);
46              }
47          } else {
48              if (PlayersService.updatePlayer(id, req.body)) {
49                  res.sendStatus(204);
50              } else {
51                  res.sendStatus(404);
52              }
53          }
54      }
55
56      postPlayer(req, res) {
57          var playerInfo = req.body;
58
59          if (PlayersService.addPlayer(playerInfo)) {
60              res.setHeader('Location', '/players/' + playerInfo.id);
61              res.sendStatus(200);
62          } else {
63              res.sendStatus(500);
64          }
65      }
66  }
67
68  module.exports = PlayersController;
```

So far we have just been attaching everything to the main express app. Express however has a concept of a Router which is basically a “mini-app”. Just like the app, we can define endpoints with all of the same functionality as the main app. We can then tell the main app to use our router for a particular endpoint, and that will become the base path for our router’s endpoints.

This helps us produce modular code, and makes it so that when we pass in the router to our controller class we don't have to add the full path to each endpoint.

```
1 var apiRouter = express.Router();
2 app.use('/api/v1', apiRouter);
3
4 var PlayersController = require('./controllers/players');
5 var pc = new PlayersController(apiRouter);
```

The LeaderBoard service is going to be very similar to the player service. One thing to note about the leaderboard is that we need to track the "rankDirection" for each board. This is going to just be a simple flag that lets us know which way to sort the scores for each board (high score is better? or lowest score?).

```
'use strict';

var uuid = require('node-uuid');

class LeaderBoardService {
    constructor() {
        this.boards = [];
    }

    getBoards() {
        return this.boards;
    }

    getSingleBoard(boardId) {
        return this.boards.filter(b => b.id === boardId)[0] || null;
    }

    addBoard(boardName, rankDirection) {
        var existingBoard = this.boards.filter(b => b.boardName === boardName).length;
        if (existingBoard) {
            return null;
        }

        var board = { boardName: boardName, rankDirection: rankDirection };
        board.id = uuid.v4();

        this.boards.push(board);

        return board;
    }

    updateBoard(boardId, info) {
        var board = this.getSingleBoard(boardId);
        if (board) {
            board.boardName = info.boardName ? info.boardName : board.boardName;
            board.rankDirection = info.rankDirection ? info.rankDirection : board.rankDirection;

            return true;
        }
        return false;
    }
}

module.exports = new LeaderBoardService();
```

The controller for the boards is basically the same as the players controller. We wire up the endpoints to a router, calling the class methods which in turn utilize the service to manipulate the data. The controller class interacts with the HTTP request and response objects to provide access for our verbs.

```
'use strict';

var BoardsService = require('../services/boards');

class BoardsController {
    constructor(router) {
        this.router = router;
        this.registerRoutes();
    }

    registerRoutes() {
        this.router.get('/leaderboards', this.getBoards.bind(this));
        this.router.get('/leaderboards/:id', this.getSingleBoard.bind(this));
        this.router.post('/leaderboards', this.postBoard.bind(this));
        this.router.put('/leaderboards/:id', this.putBoard.bind(this));
    }

    getBoards(req, res) {
        var boards = BoardsService.getBoards();
        res.send(boards);
    }

    getSingleBoard(req, res) {
        var id = req.params.id;
        var board = BoardsService.getSingleBoard(id);

        if (!board) {
            res.sendStatus(404);
        } else {
            res.send(board);
        }
    }

    putBoard(req, res) {
        var id = req.params.id;
        var existingBoard = BoardsService.getSingleBoard(id);

        if (!existingBoard) {
            let board = BoardsService.addBoard(req.body.boardName, req.body.rankDirection);
            if (board) {
                res.setHeader('Location', '/leaderboards/' + board.id);
                res.sendStatus(201);
            } else {
                res.sendStatus(500);
            }
        } else {
            if (BoardsService.updateBoard(id, req.body)) {
                res.sendStatus(204);
            } else {
                res.sendStatus(404);
            }
        }
    }
```

Enr
developer.

```
54
55      postBoard(req, res) {
56          var board = BoardsService.addBoard(req.body.boardName, req.body.rankDirection);
57
58          if (board) {
59              res.setHeader('Location', '/leaderboards/' + board.id);
60              res.sendStatus(200);
61          } else {
62              res.sendStatus(500);
63          }
64      }
65  }
66
67  module.exports = BoardsController;
```

Again, just like the controller for Players, the Boards controller needs to be added to the main API Router.

```
1  var BoardsController = require('./controllers/boards');
2  var bc = new BoardsController(apiRouter);
```

Then just like before, we'll add a couple of boards for us to test things with, go back and remove them before you deploy your final service.

```
1  var BoardsService = require('./services/boards');
2  BoardsService.addBoard('Total Score', 1);
3  BoardsService.addBoard('Times Died', 0);
```

The Scores are going to be a little different than the other two objects that we have been working with so far. Scores are stored in relation to a particular leaderboard, so the scores service is going to need a reference to the boards service in order to look up the board information. When we get scores, we're automatically going to sort the results based on the rankDirection of the associated leaderboard. Also, we only need to add a score to the collection if it is better than the player's previous score. We're only going to track their *best* score (in relation to the leaderboard) not a history of every game session for that player.

```
'use strict';

var BoardsService = require('./boards');

class ScoresService {
    constructor() {
        this.scores = new Map();
    }

    isBetter(newScore, oldScore, direction) {
        if (direction === 1) { // higher is better
            return (newScore > oldScore);
        } else {
            return (newScore < oldScore);
        }
    }

    getScores(boardId) {
        var board = BoardsService.getSingleBoard(boardId);
        if (!board || !this.scores.has(board.id)) {
            console.log('board not found! ' + boardId)
            return [];
        }

        let results = [];
        for (let score of this.scores.get(board.id).values()) {
            results.push(score);
        }

        return results.sort((a, b) => this.isBetter(b.score, a.score, board.rankDirection));
    }

    addScore(boardId, playerId, score) {
        var board = BoardsService.getSingleBoard(boardId);
        if (!board) {
            console.log('addScore: can\'t find board: ', boardId);
            return false;
        }
```

```
39
40          var scoreObj = {score: score, submitted: new Date(), playerId: playerId};
41          if (!this.scores.has(board.id)) {
42              this.scores.set(board.id, new Map());
43              this.scores.get(board.id).set(playerId, scoreObj);
44          } else {
45              if (this.scores.get(board.id).has(playerId)) {
46                  if (this.isBetter(score, this.scores.get(board.id).get(playerId).score, board.rankDirection)) {
47                      this.scores.get(board.id).set(playerId, scoreObj);
48                  }
49              } else {
50                  this.scores.get(board.id).set(playerId, scoreObj);
51              }
52          }
53
54          return true;
55      }
56  }
57
58  module.exports = new ScoresService();
```

The Scores controller is similar to the other controllers, however because of the relationship to the leaderboards, the endpoints are based off a specific leaderboard id.

```
1   'use strict';
2
3   var ScoresService = require('../services/scores');
4
5   class ScoresController {
6       constructor(router) {
7           this.router = router;
8           this.registerRoutes();
9       }
10
11      registerRoutes() {
12          this.router.get('/leaderboards/:leaderboardId/scores', this.getScores.bind(this));
13          this.router.post('/leaderboards/:leaderboardId/scores', this.submitScore.bind(this));
14      }
15
16      getScores(req, res) {
17          var boardId = req.params.leaderboardId;
18          var scores = ScoresService.getScores(boardId);
19          res.send(scores);
20      }
21
22      submitScore(req, res) {
23          var boardId = req.params.leaderboardId;
24          ScoresService.addScore(boardId, req.body.playerId, req.body.score);
25
26          res.sendStatus(200);
27      }
28  }
29
30  module.exports = ScoresController;
```

The scores controller is added to the apiRouter in the same way as all of the other controllers.

```
var ScoresController = require('./controllers/scores');
var sc = new ScoresController(apiRouter);
```

Again, we'll add a few test data items so we can try out the scores without having to spend a lot of time adding them via the REST client every time we start the server.

```
var ScoresService = require('./services/scores');
ScoresService.addScore(b1.id, p1.id, 3000);
ScoresService.addScore(b1.id, p2.id, 2345);
ScoresService.addScore(b1.id, p3.id, 15238);
ScoresService.addScore(b2.id, p1.id, 33);
ScoresService.addScore(b2.id, p2.id, 7);
ScoresService.addScore(b2.id, p3.id, 67);
```

You may have noticed when we created the Scores controller, that we had to worry about (and repeat) the "/leaderboards/:leaderboardId" portion of the route. Routers actually can help here, we can modularize our endpoints much more than they are now. Just like we can add a router to handle a base url for the app, we can add a router to handle a base url for another router. We can refactor our code so that each controller only needs to know exactly about the resource that it's providing access to.

```
var apiRouter = express.Router();
app.use('/api', apiRouter);

var apiV1 = express.Router();
apiRouter.use('/v1', apiV1);

var playersApiV1 = express.Router();
apiV1.use('/players', playersApiV1);

var boardsApiV1 = express.Router();
apiV1.use('/leaderboards', boardsApiV1);

var PlayersController = require('./controllers/players');
var pc = new PlayersController(playersApiV1);
var BoardsController = require('./controllers/boards');
var bc = new BoardsController(boardsApiV1);
var ScoresController = require('./controllers/scores');
var sc = new ScoresController(boardsApiV1);
```

Separating out the version (v1) from the main apiRouter allows us to make “breaking” changes to our API later down the road, and simply increase the version. Since it’s all modular, we can do this even for just some of our resources, we don’t have to update the entire API to v2.

```
registerRoutes() {
    this.router.get('/', this.getPlayers.bind(this));
    this.router.get('/:id', this.getSinglePlayer.bind(this));
    this.router.post('/', this.postPlayer.bind(this));
    this.router.put('/:id', this.putPlayer.bind(this));
}
```

Then we refactor the rest of the controllers similarly, now if we wanted to change the endpoint from “/players” to “/gamers” in v2 we could! Routers are extremely powerful this way, and really help make your API be robust and forward thinking.

Documentation

A large part of a successful API is being well documented. Generally as programmers however, we don’t really enjoy writing documentation. Plus, if we write just a markdown file, it can quickly become stale and need to be updated. On top of that, having static documentation isn’t that great, the user still needs to use something like the Advanced REST Client to interact and try out the API. It would be great to have the documentation be interactive, so that everything you need to use an API is right there, and easy to follow. Fortunately, a tool has been created that will do this for you! It’s called Swagger, and while there are likely others, it’s one of the more popular tools.

Let’s start out by installing the node.js swagger tools.

```
npm install --save swagger-tools
```

Now we need to configure our web app to use the swagger tools. They are a set of middleware functions that we can pass to the express application.

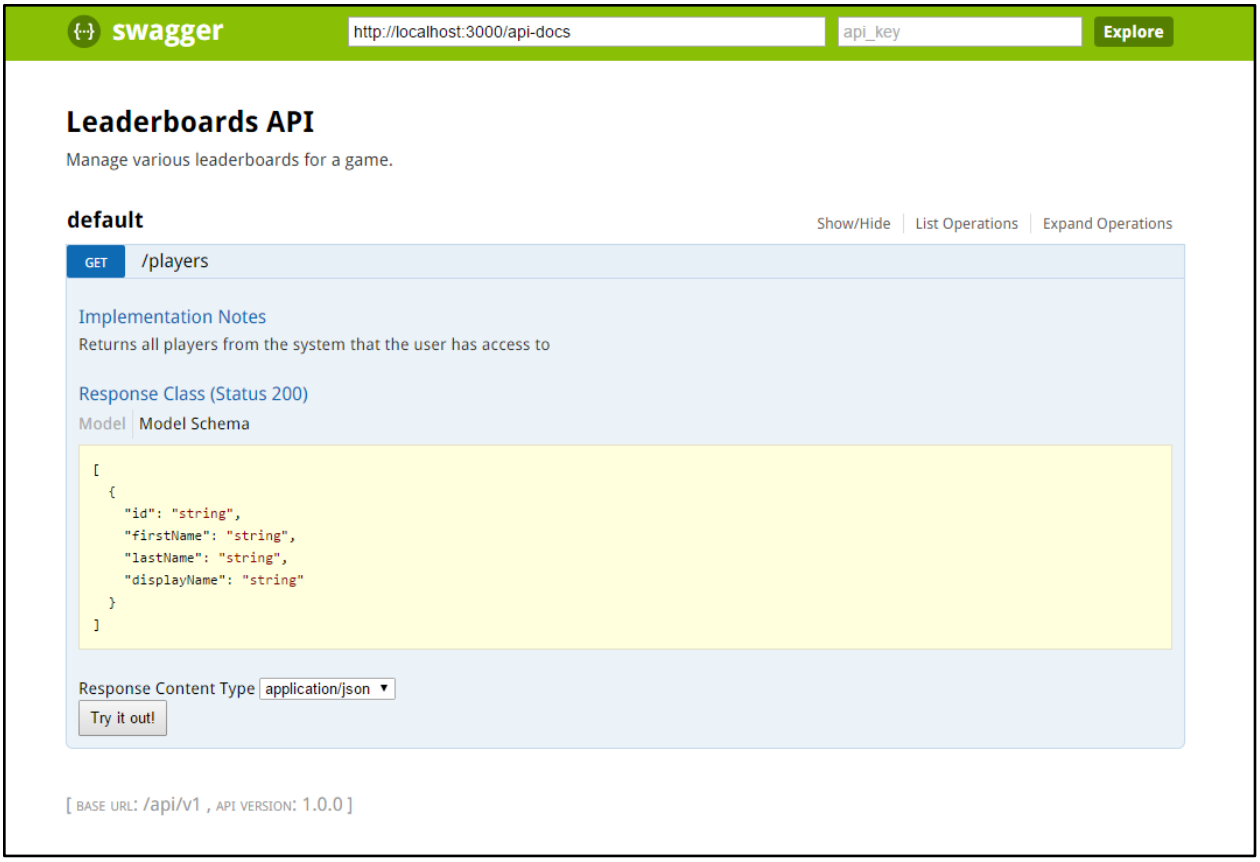
```
// Swagger Docs
var swaggerTools = require('swagger-tools');
// swaggerRouter configuration
var options = {
  swaggerUi: '/swagger.json',
  controllers: './controllers'
};

// The Swagger document (require it, build it programmatically, fetch it from a URL, ...)
var swaggerDoc = require('./swagger.json');

// Initialize the Swagger middleware
swaggerTools.initializeMiddleware(swaggerDoc, function (middleware) {
    // Interpret Swagger resources and attach metadata to request - must be first in swagger-tools middleware chain
    app.use(middleware.swaggerMetadata());

    // Validate Swagger requests
    app.use(middleware.swaggerValidator());

    // Route validated requests to appropriate controller
    app.use(middleware.swaggerRouter(options));

    // Serve the Swagger documents and Swagger UI
    app.use(middleware.swaggerUi());

    // start the server
    var server = app.listen(3000, function () {
        var host = server.address().address;
        host = (host === '::' ? 'localhost' : host);
        var port = server.address().port;

        console.log('listening at http://%s:%s', host, port);
    });
});
```

Swagger uses a JSON configuration to describe our API to it so that it can build the documentation. There is extensive documentation on the Swagger Spec that you can read to really flesh out a great service, there is also an online editor with examples that can help you get going more quickly. I used the examples to create a basic configuration for our API. I started with just the first GET request for the players collection.

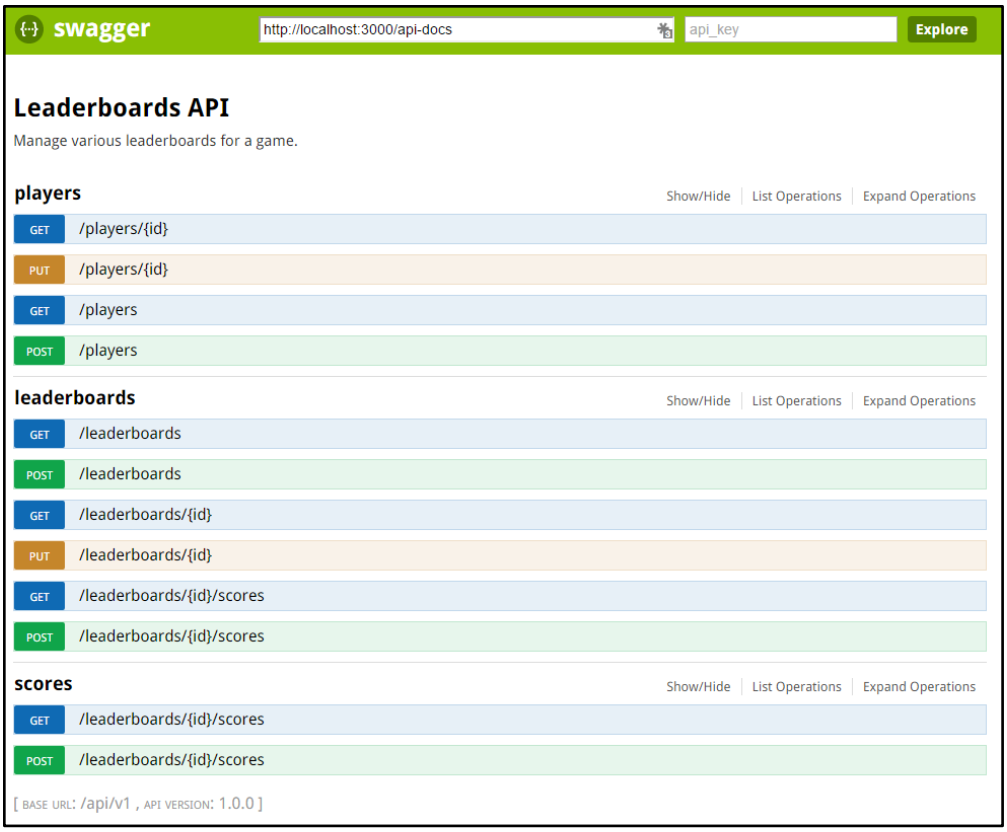
```
{
    "swagger": "2.0",
    "info": {
        "title": "Leaderboards API",
        "description": "Manage various leaderboards for a game.",
        "version": "1.0.0"
    },
    "produces": [
        "application/json"
    ],
    "host": "localhost:3000",
    "basePath": "/api/v1",
    "paths": {
        "/players": {
            "get": {
                "summary": "",
                "description": "Returns all players from the system that the user has access to",
                "x-swagger-router-controller": "players",
                "operationId": "getPlayers",
                "responses": {
                    "200": {
                        "description": "players response",
                        "schema": {
                            "type": "array",
                            "items": {
                                "$ref": "#/definitions/player"
                            }
                        }
                    }
                }
            }
        }
    },
    "definitions": {
        "player": {
            "type": "object",
            "required": [
                "id",
                "firstName",
                "lastName",
                "displayName"
            ],
            "properties": {
                "id": {
                    "type": "string"
                },
                "firstName": {
                    "type": "string"
                },
                "lastName": {
                    "type": "string"
                },
                "displayName": {
                    "type": "string"
                }
            }
        }
    }
}
```

Enroll in our Full-Stack Web Development Mini-Degree to go from zero to professional JavaScript developer.

You can see that this is much nicer already than having to deal with the Advanced REST Client generically, or interacting via a javascript library (if you've been doing that). Instead now we have a beautiful website where we can just click a button to try our API live and gain understanding of how it is used.



After that we simply need to go through and add configuration for each of our API endpoints until we have a fully documented service that looks like this.



Swagger has done all the work of creating a detailed and easy to use documentation site. The users of your API will surely appreciate this and consequently your API is more likely to gain popularity and be adopted by the community.

The online editor for Swagger actually has the capability of generating code from the documentation configuration files. Instead of building out the API and then documenting it like we have here, next time you could try describing the API to the documentation and then letting Swagger generate a Express (and others) based application for you!

All finished, what's next?

Now we have a fully functional Leaderboard API, the next obvious step would be to wire up an application to use it. Additionally there are a few things that might be a high priority left such as Authentication and Authorization using API keys, or wiring up the services to persist to a database rather than running in memory. You could add other fun features to the leaderboards, like user avatars, achievements, or even support multiple games.

I'll leave those things up to you to implement, you may have more ideas of your own. Please feel free to leave a comment about your ideas, or let me know how and where you used this tutorial (I like to play games too!).

Part 3: Node.js Quickies 1 – Working with MySQL

In this tutorial I'll give you a quick overview on how to work with MySQL and Node.js.

Firstly, you need to include the mysql package in your package.json file, under "dependencies". Make sure you run "npm install" to download all your project's dependencies.

```
"dependencies": {
    "node-mysql": "*"
},

```

Like with many things in life, you start by establishing a connection:

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

//mysql setup
var connection = mysql.createConnection({
  host     : 'localhost',
  user     : 'username',
  password : 'secret_password',
  database: 'database_name'
});

connection.connect();
```

This is how you can select data:

```
connection.query('SELECT * FROM user', function(err, rows, fields){
  if (err) throw err;

  rows.each(element, index) {
    console.log(element.firstName + " " + element.lastName);
  }
});
```

How about updating data:

```
connection.query('UPDATE user SET first_name = "'+ fName +'" WHERE username = "'+username+'"', function(err, result){
  console.log('updated user! ' + element.username);
});
```

Why you so asynchronous?

The main difference between working with MySQL in Node.js and other web application technologies such as PHP, is that in Node these calls are asynchronous. In PHP, once you do a query, the script execution will stop until the database has responded to the query. In Node, on the other hand, the query is an asynchronous call, so the script keeps on going.

If you are doing many queries and the asynchronous nature of it becomes too complex it's worth looking into using Promises. I'll cover promises using the Q package in another tutorial.

You can learn more about node-mysql by checking out it's Github repo.

Part 4: Node.js Quickies 2 – Loading a CSV File

In this short tutorial I'm gonna show you how to load a CSV file using Node.js.

Firstly, you need to include the ya-csv package in your package.json file, under "dependencies". Make sure you run "npm install" to download all your project's dependencies.

```
"dependencies": {
    "ya-csv": "*"
},
```

This is how you can load a csv file:

```
var csv = require('ya-csv');

//load csv file
var loadCsv = function() {
    var reader = csv.createCsvFileReader('data/ActualizaSC.csv', {
      'separator': ',',
      'quote': '"',
      'escape': '"',
      'comment': '',
  });

  var allEntries = new Array();

  reader.setColumnNames([ 'firstName', 'lastName', 'username' ]);

  reader.addListener('data', function(data) {
    //this gets called on every row load
    allEntries.push(data);
  });

  reader.addListener('end', function(data) {
    //this gets called when it's finished loading the entire file
    return allEntries;
  });
};

var myUsers = loadCsv();
```

You can learn more about this package by checking out it's Github repo.
