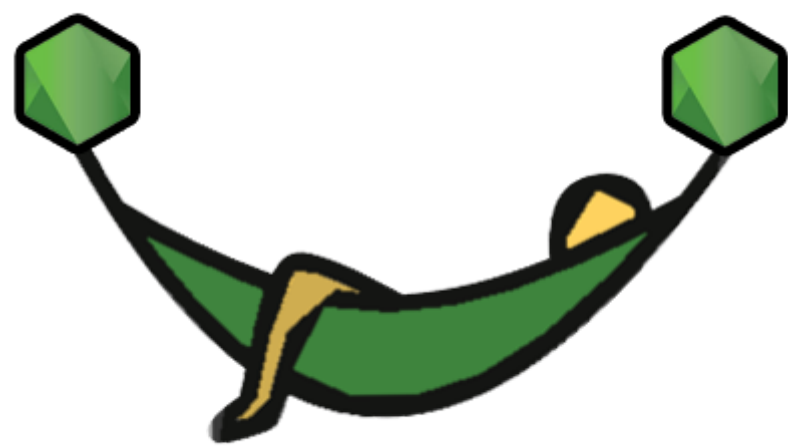


Node.js Express Rest Api Tutorial



Let's see how to build a simple REST api using Express to allow for all of the

CRUD operations. REST is referring to Representational State Transfer. It is a convention for building HTTP services via a client and server architecture. In REST, the HTTP protocol is used to facilitate Create, Read, Update, and Delete of resources on the server. These operations can be referred to collectively as CRUD operations. Each operation uses a different HTTP verb. Create uses a POST request. Read uses a GET request. Update uses a PUT request. Finally, Delete uses a DELETE request. In this tutorial, we will create all of these services using [Node.js](#) and [Express](#).

Express Web Server

We've learned how to scaffold out an Express project already, so let's go ahead and create a new one so we can build out our new REST api. Enter the following commands to get started.

- `node $mkdir express-rest`
- `node $cd express-rest`
- `express-rest $npm init --yes`

P-SILVER CORE

INTEL® XEON® SILVER 4110



 1 CPU (8C/16T)
@2.1 GHZ

 1 GBPS
BANDWIDTH

 2 x 240 GB
SSD

 64 GB RAM
DDR4

FROM

€169⁹⁹ / MTH
• EX. VAT












DISCOVER

ikoula

we host with care

www.ikoula.com

Advertise Here

- [How To Send Email To New Users](#)
- [Interface Segregation Principle](#)
- [How To Highlight New Content For Returning Visitors](#)
- [HTML Form Tutorial](#)
- [How to add a WordPress Blog to your Laravel Application](#)
- [Creating C# Class Properties And Tests](#)
- [CSS Crash Course Tutorial For Beginners](#)
- [What's Next For Bootstrap](#)
- [Basic Laravel Routing and Views](#)
- [Conditional Rendering In React](#)
- [How To Compare Two Arrays of](#)

```

node $mkdir express-rest
node $cd express-rest
express-rest $npm init --yes
Wrote to C:\node\express-rest\package.json:

{
  "name": "express-rest",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

express-rest $npm i express

```

We also need an index.js file in the root of the project, so we can create that too. Once that is complete, add the following code. We are requiring the express module, and then calling the express() function and assigning the result to the `app` constant. The result is an Object, and by convention it is typically named `app`. Visual Studio Code informs us of the following about `express()`. "Creates an Express application. The `express()` function is a top-level function exported by the `express` module."

```

1 const express = require('express');
2 const app = express();
3
4 app.get('/', (req, res) => {
5   res.send('Oh Hi There!');
6 });
7
8 app.listen(3000, () => console.log('Listening on port 3000'))

```

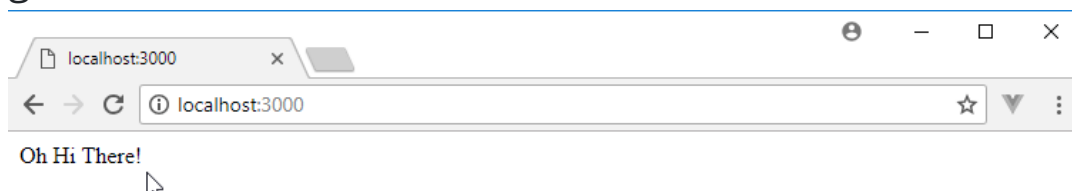
We can launch the web server by typing `node index.js` at the command prompt.

```

express-rest $ls
index.js  node_modules/  package-lock.json  package.json
express-rest $node index.js
Listening on port 3000

```

Now that the server is running, we can load up `http://localhost:3000` in the browser to see that all is good.



Now we will add a new route that will simulate an API. We want to be able to visit the `/api/games` endpoint and see all the games in the system. At this point, we have no

Data and Calculate Position Differences



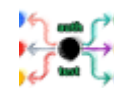
[Introduction To Laravel 5](#)



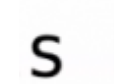
[Combine PHP Functions To Make Your Own](#)



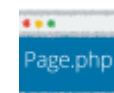
[How To Add Post Meta In WordPress](#)



[Using Test Authentication To Allow Logged In Users To Post Replies](#)



[What Is HTML](#)



[How To Create Custom Templates For WordPress Pages](#)



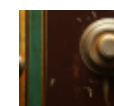
[ES6 Generators](#)



[Create A Navbar Component In React](#)



[Laravel Validation](#)



[Laravel hasMany and belongsTo Tutorial](#)



[jQuery Selectors and Filters](#)



[A User hasMany Games and also hasMany Reviews](#)



[Document Object Model Tutorial](#)



[Getting Your Database Set Up For Use With Laravel](#)

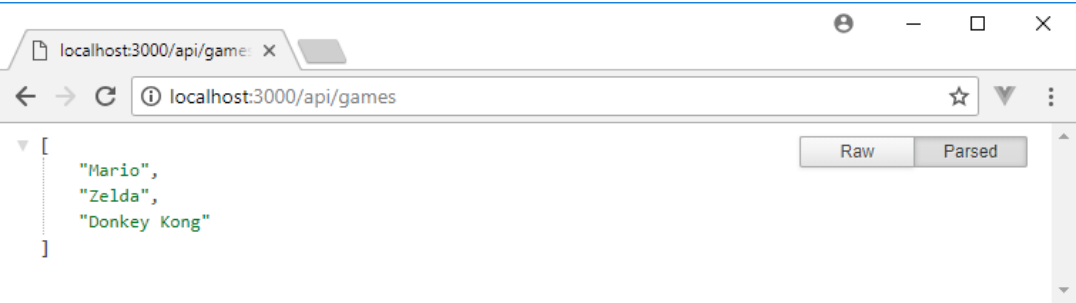


[Open Closed Principle](#)

database so we'll just populate a simple array as an example.

```
1 const express = require('express');
2 const app = express();
3
4 app.get('/', (req, res) => {
5   res.send('Oh Hi There!');
6 });
7
8 app.get('/api/games', (req, res) => {
9   res.send(['Mario', 'Zelda', 'Donkey Kong']);
10 });
11
12 app.listen(3000, () => console.log('Listening on port 3000'));
```

In order to test this, you'll need to restart the web server at the command line. First type CTRL-C to halt the server, then re launch the server again with node `index.js`. Now, go ahead and visit `http://localhost:3000/api/games` in the browser and have a look!

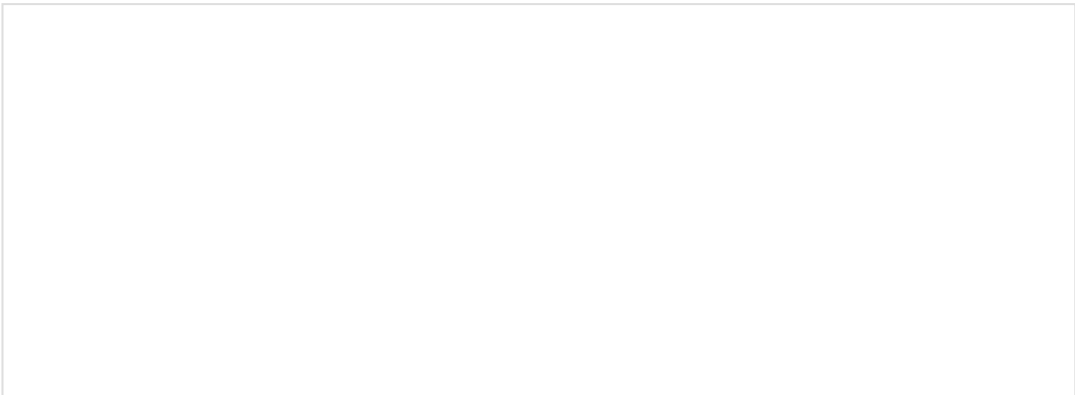





Nodemon Express

As we move along, it is going to be tiresome to manually restart the server on every update we make. We can easily deal with this problem by making use of nodemon when launching the server. Nodemon is likely installed if you checked out our [rendering html with node](#) tutorial. If it's not installed, just go ahead and type `npm i -g nodemon` at the command line. Once ready, halt the server and re launch it using `nodemon index.js`.

```
express-rest $nodemon index.js
[nodemon] 1.17.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
Listening on port 3000
```

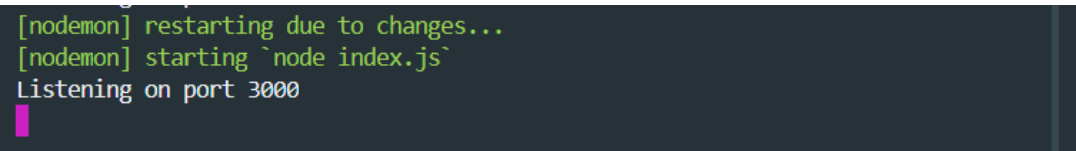
Now when you make changes to your files like in line 5 here,



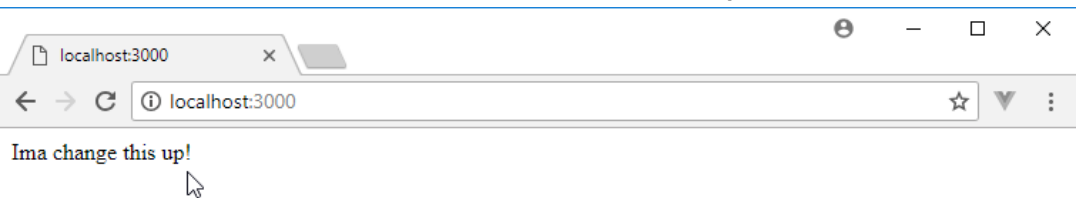
- [Laravel API Resource Tutorial](#)
- [27 Awesome VueJS Libraries](#)
- [WordPress Post Formats](#)
- [Install Laravel Homestead on Windows](#)
- [Liskov Substitution Principle](#)
- [Check Authorization With Policies Before Delete Function](#)
- [Underscore JS sortBy Function](#)
- [Most Useful JavaScript Array Functions](#)
- [The Top 15 Most Popular JavaScript String Functions](#)
- [VueJs Parent Child Communication](#)
- [Vuejs Form Example](#)
- [What Is Goutte?](#)
- [Twitter Bootstrap Navigation Elements](#)
- [The 27 Most Popular File Functions in PHP](#)
- [Angular Parent Child Communication](#)
- [AngularJS Component](#)

```
1 const express = require('express');
2 const app = express();
3
4 app.get('/', (req, res) => {
5   res.send('Ima change this up!');
6 });
7
8 app.get('/api/games', (req, res) => {
9   res.send(['Mario', 'Zelda', 'Donkey Kong']);
10 });
11
12 app.listen(3000, () => console.log('Listening on port 3000'));
```

You will no longer need to stop and then restart the server. Go ahead and make your updates, and you'll see the command line reflect this automatically.



Of course the result in the browser is updated as well.



Setting Port with an Environment Variable

So far we have been hard coding the values for different application variables such as the listening port. You're not going to want to do that once your building anything bigger than a tutorial app. The more appropriate way to handle this would be to check for an environment variable with the name of PORT, and if it is set, use that value. If it is not set, you can fall back to a sensible default. Note the new code here.



Tutorial



[How To Filter Models By Another Model](#)



[Drawing Scalable Vector Graphics With D3 JavaScript](#)



[PHP Booleans and Constants Tutorial](#)



[Douglas Crockford The Good Parts Examples](#)



[What Is An Assembly In C#?](#)



[What are Getters and Setters?](#)



[Node Package Manager Tutorial](#)



[Axios Powered VueJS Form Component](#)



[How To Use An Interface In Angular](#)



[Using Operators in PHP](#)



[Angular Data Binding](#)



[How To Validate Form](#)

[Submissions In Laravel](#)



[Install Twitter Bootstrap](#)



[Introduction To AngularJS](#)



[How Do Linux Permissions Work?](#)



[Getting Started with Data](#)


```
1 const express = require('express');
2 const app = express();
3
4 app.get('/', (req, res) => {
5   res.send('I'ma change this up!');
6 });
7
8 app.get('/api/games', (req, res) => {
9   res.send(['Mario', 'Zelda', 'Donkey Kong']);
10 });
11
12 const port = process.env.PORT || 3000;
13 app.listen(port, () => console.log(`Listening on port $
```

Now, the port can be set and read from the environment variable. Here, we set the port to 4000 instead of the 3000 we were using. Note that when we relaunch nodemon, the server is now listening on port 4000.

```
npm
express-rest $export PORT=4000
express-rest $nodemon index.js
[nodemon] 1.17.5
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
Listening on port 4000...
```

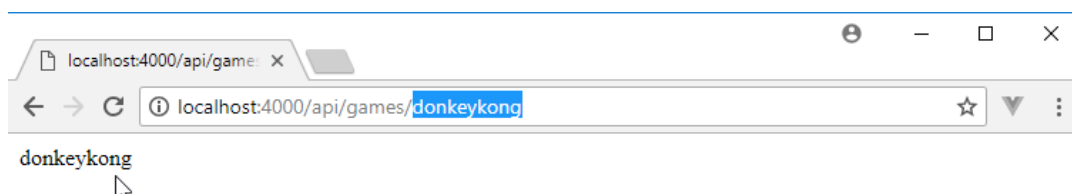
Express Route Parameters

Single Params

Route parameters are of supreme importance. It is by passing route parameters that a user can specify to the server what resource it wants to fetch. Consider the following code.

```
1 app.get('/api/games/:id', (req, res) => {
2   res.send(req.params.id);
3 });
```

What this says is that anytime a get request is made to `/api/games`, anything that comes after that is a dynamic route parameter. So for example if we visit `http://localhost:4000/api/games/25`, then 25 is the route parameter. To fetch this in the code, you can use **req.params.id**. Here we visit `http://localhost:4000/api/games/donkeykong` and note we simply display the value of the route parameter in the browser.



Manipulation Language in MySQL



[Vuejs Router Tutorial](#)



[Testing JavaScript With Jest](#)



[Laravel Subscription System Tutorial](#)



[What Is Guzzle PHP?](#)



[Flash Messages in Laravel](#)



[Setting Up A Database With Seeding](#)



[PHP Variables and Strings Tutorial](#)



[Vue.js Tutorial](#)



[The Art of Creating a WordPress Post](#)



[How To Create A Child Component In VueJS](#)



[What is the IoC Container in Laravel?](#)



[Create React App Tutorial](#)



[HTML Hyperlinks Tutorial](#)

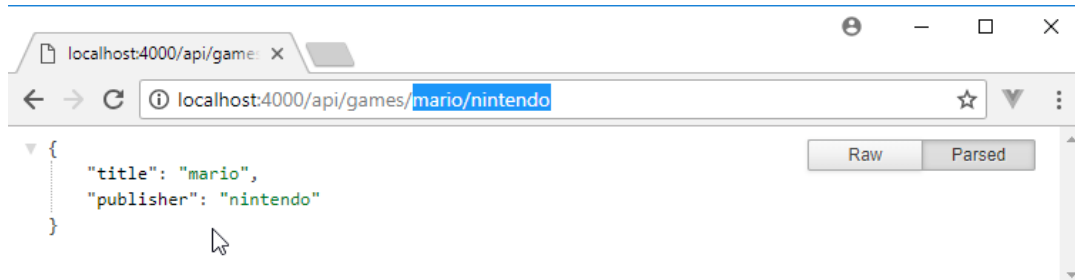
Multiple Params

You can have more than one route parameter in the url. Consider this code.

```
1 app.get('/api/games/:title/:publisher', (req, res) => {  
2   res.send(req.params);  
3 });
```

Now if we visit

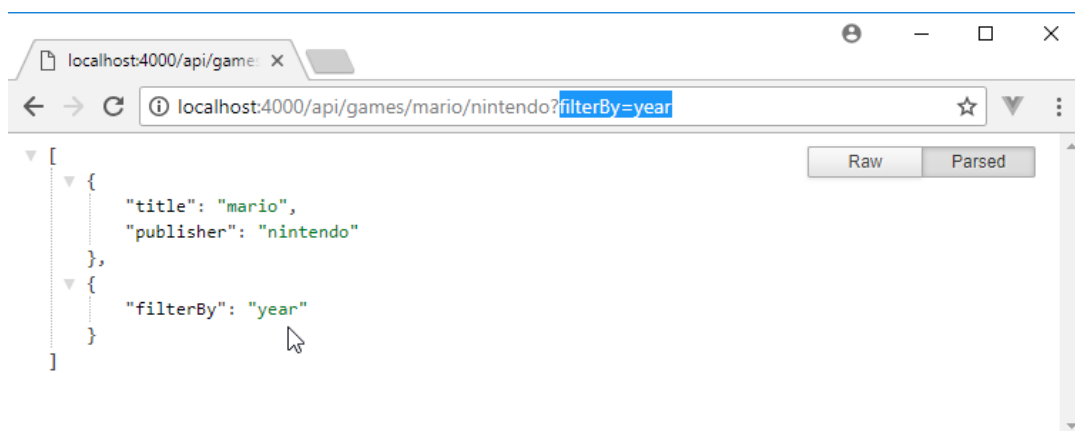
http://localhost:4000/api/games/mario/nintendo here is the result we get.



Query Parameters

We can also access query parameters in the url. Here we update the code to allow for both route parameters and query parameters.

```
1 app.get('/api/games/:title/:publisher', (req, res) => {  
2   res.send([req.params, req.query]);  
3 });
```



Route parameters are used for primary and needed values for working with a resource. Query parameters on the other hand can be thought of as more optional in nature.

HTTP GET Requests

Now we can set up some get requests for fetching games from the server. This corresponds to the *Read* of crud in a rest api. We are simply using an array of games, as again there is no database just yet. So let's set up an array of games like we see here.

```
1 const games = [{
2   id: 1,
3   title: 'Mario'
4 },
5 {
6   id: 2,
7   title: 'Zelda'
8 },
9 {
10  id: 3,
11  title: 'Donkey Kong'
12 }
13 ];
```

Typically in the RESTful convention, if you make a get request to the api with no route parameters specified, then you should get back all resources. So if we visit `/api/games`, then we should see all games. This code here should do the trick.

```
1 // get all games
2 app.get('/api/games', (req, res) => {
3   res.send(games);
4 });
```

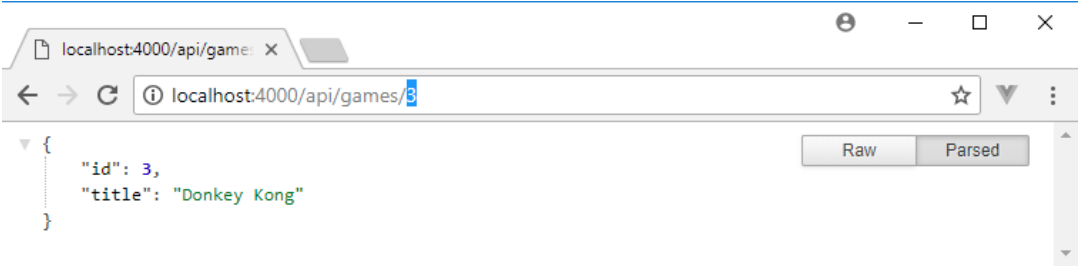
Looks good!



Now we want to be able to find a specific game only using a route parameter. For this, we can use the [find function](#).

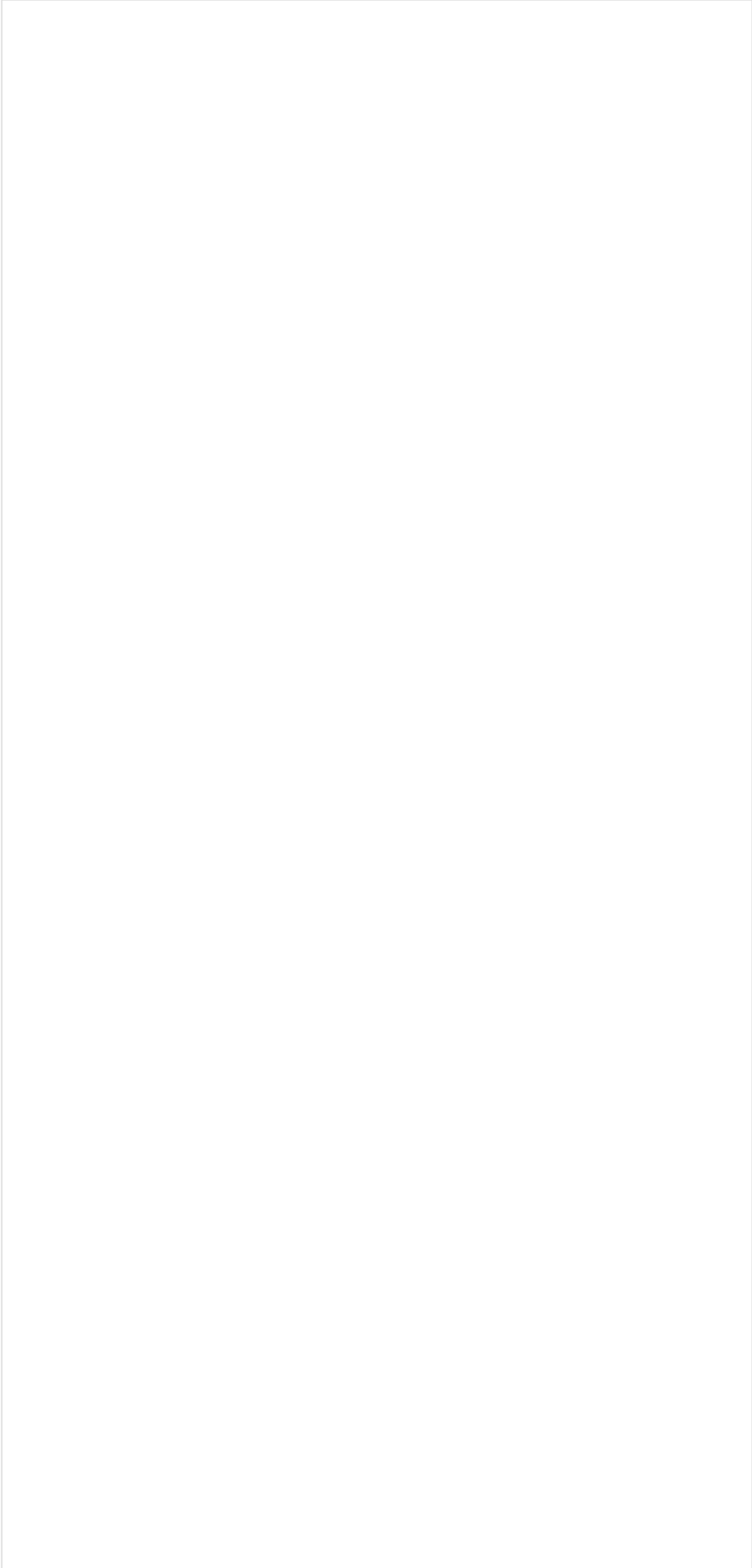
```
1 // get game by id
2 app.get('/api/games/:id', (req, res) => {
3   const game = games.find(g => g.id === parseInt(req.params.id));
4   if (!game) return res.status(404).send('The game was not found');
5   res.send(game);
6 });
```

This is also working nicely! When we provide the route parameter of `3`, we get the **Donkey Kong** game back.



HTTP POST Requests

Now we need to set up the code that will allow our web server to respond to http post requests. This corresponds to the *Create* of crud in a rest api. We can use a post request to add a new game to the system. Note the additional code here.




```
1 const express = require('express');
2 const app = express();
3
4 app.use(express.json());
5
6 const games = [{
7   id: 1,
8   title: 'Mario'
9 },
10 {
11   id: 2,
12   title: 'Zelda'
13 },
14 {
15   id: 3,
16   title: 'Donkey Kong'
17 }
18 ];
19
20 // get all games
21 app.get('/api/games', (req, res) => {
22   res.send(games);
23 });
24
25 // get game by id
26 app.get('/api/games/:id', (req, res) => {
27   const game = games.find(g => g.id === parseInt(req.params.id));
28   if (!game) return res.status(404).send('The game does not exist');
29   res.send(game);
30 });
31
32 // add a game
33 app.post('/api/games', (req, res) => {
34   const game = {
35     id: games.length + 1,
36     title: req.body.title
37   }
38   games.push(game);
39   res.send(game);
40 });
41
42 const port = process.env.PORT || 3000;
43 app.listen(port, () => console.log(`Listening on port ${port}`));
```

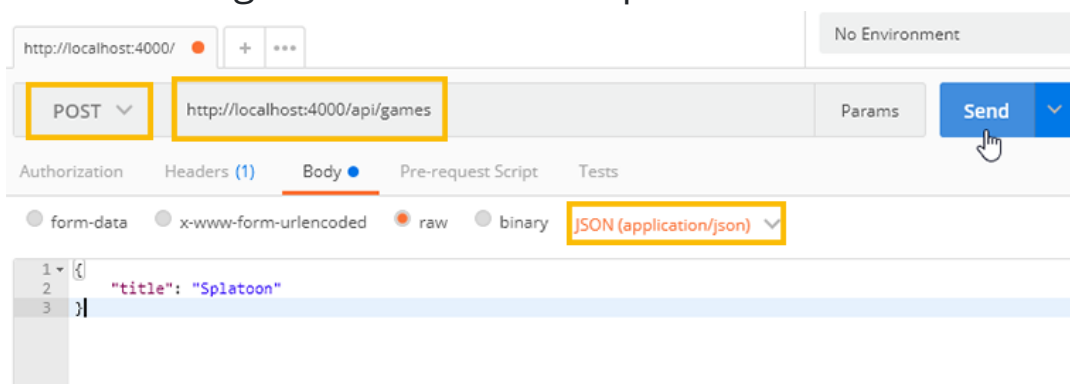
The first thing we notice in the snippet above is the `app.use(express.json());` line. This specifies that we are adding in a piece of middleware to our application. The reason we need this is because we need to parse the title property from the body of the post request,

req.body.title. Express needs to be able to parse json objects in the body of the request so this middleware turns that on.

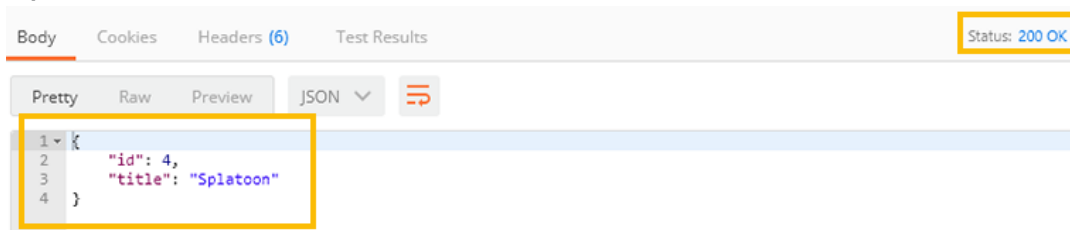
Next, we see that we are posting to the collection, or in other words to /api/games. This example is a bit of a hack since we are not actually working with a database, but it gets the idea across. That is why we are getting the id by using games.length + 1. The title will be parsed from the json object in the post request via req.body.title. Finally, we simply push the new game onto our games array, and then send back the game as a response by convention.

Testing Endpoints With Postman

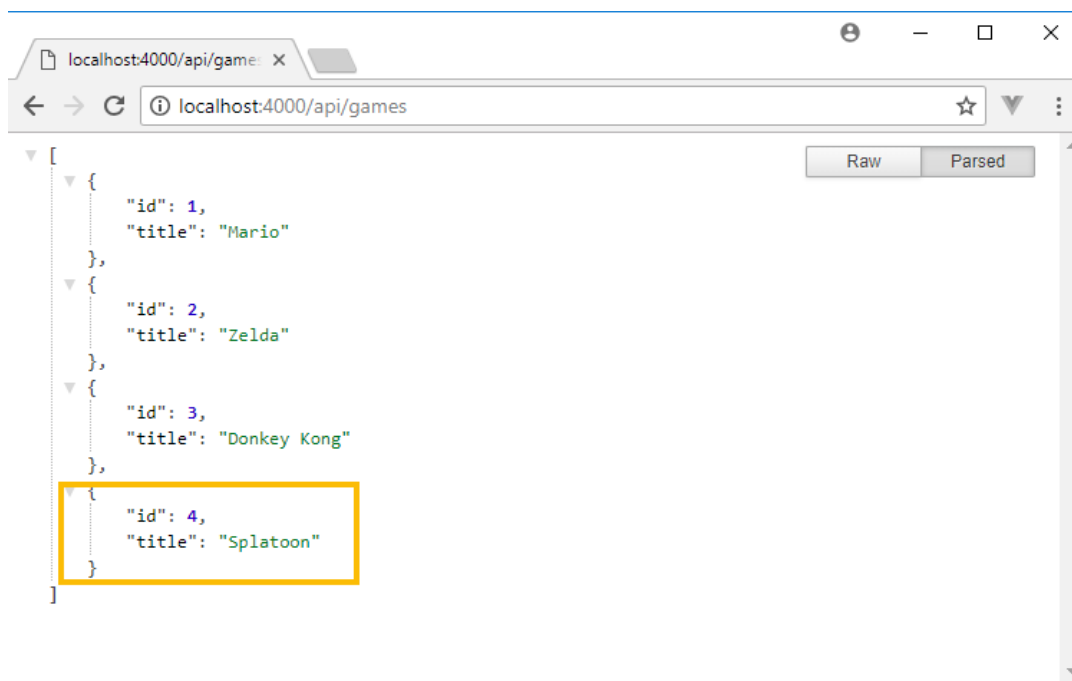
The way we can test out sending a post request with a json object in the body of the request is by using [postman](#). We specify the request type as POST, provide the route for the games collection at http://localhost:4000/api/games, and set the json object to include a game with a title of Splatoon.



Once we click Send, we should get a response back like we see here with a new id of 4 and the title of the game Splatoon.



So it looks like everything worked perfectly! This means that if we now make simple a GET request to that same collection api of api/games, then we should now see 4 games instead of the 3 we originally had and that is exactly what we get.



Joi Input Validation

When setting up an api, it is important to make sure to validate any data that gets sent to the api. You can not trust any data that a user or another application might be sending to your api. A nice way to set up input validation when using express is with the popular [Joi package](#).

Let's [install it with npm](#)!

```
express-rest $npm i joi
npm WARN express-rest@1.0.0 No description
npm WARN express-rest@1.0.0 No repository field.

+ joi@13.3.0
added 5 packages from 1 contributor in 4.342s
[+] no known vulnerabilities found [135 packages audited]

express-rest $
```

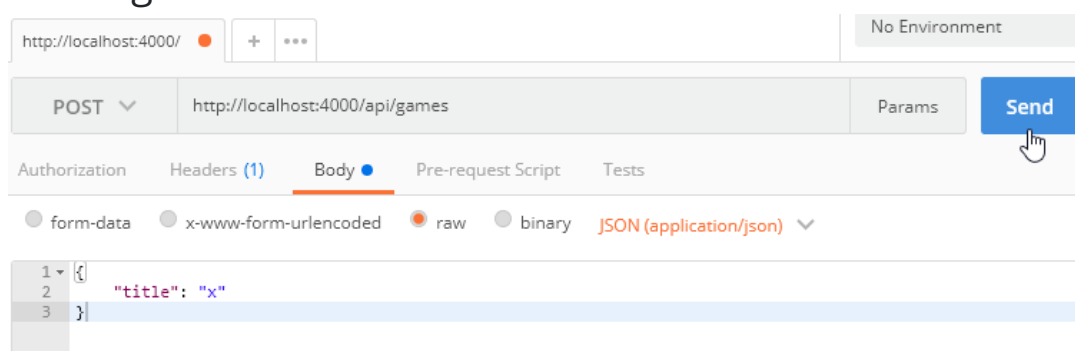
Once Joi is installed, require it in to the file like so.

```
1 const Joi = require('joi');
```

Now we can update the app.post() call like this. First, we define the schema or validation rules we need. Here we simply say the title must be at least 2 characters. Then, we make that call to the validate() method passing in the request body and the schema. From there, we just check for errors and if there are any, send them right back as a response.

```
1 // add a game
2 app.post('/api/games', (req, res) => {
3   const schema = {
4     title: Joi.string().min(2).required()
5   };
6
7   const result = Joi.validate(req.body, schema);
8   if (result.error) {
9     res.status(400).send(result.error)
10  }
11
12  const game = {
13    id: games.length + 1,
14    title: req.body.title
15  }
16  games.push(game);
17  res.send(game);
18 });
```

Great! Now, we can send a bad request using postman once again and see the result.



The response we get back is an error, just like we expected!

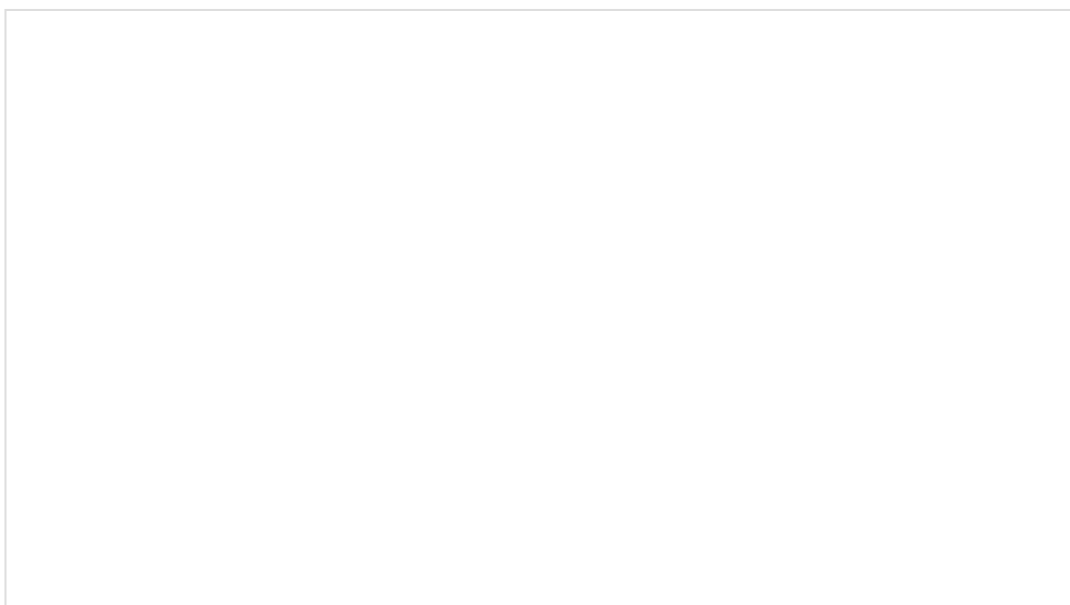
```
1 {
2   "isJoi": true,
3   "name": "ValidationError",
4   "details": [
5     {
6       "message": "\"title\" length must be at least 2",
7       "path": [
8         "title"
9       ],
10      "type": "string.min",
11      "context": {
12        "limit": 2,
13        "value": "x",
14        "key": "title",
15        "label": "title"
16      }
17    }
18  ],
19  "_object": {
20    "title": "x"
21  }
22 }
```

HTTP PUT Requests

To update an existing resource on the server you can use a PUT request. This corresponds to the *Update* of crud in a rest api. Let's see how to set up the code to handle a PUT request so we can update a game in the application. This one is just slightly more complicated. Let's review the steps we need to complete first.

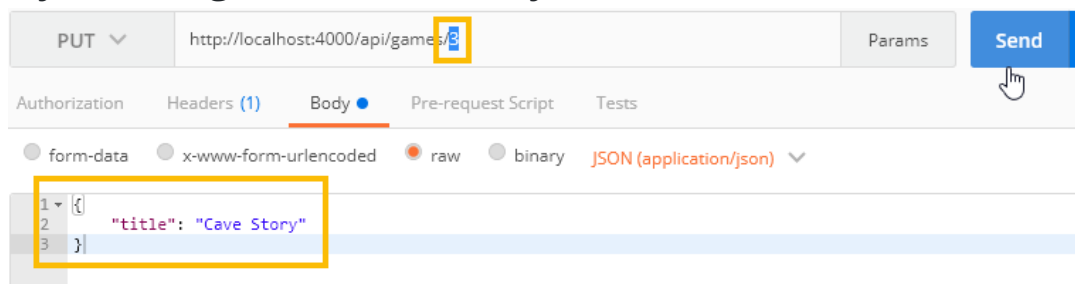
- Look up the game in the application
- If it is not found, return a 404 error
- Validate the data being sent to the server
- If that data is invalid, send an error 400 bad request
- If all checks out, update the game
- Send the updated game back as a response

This would translate into something like this in our code.

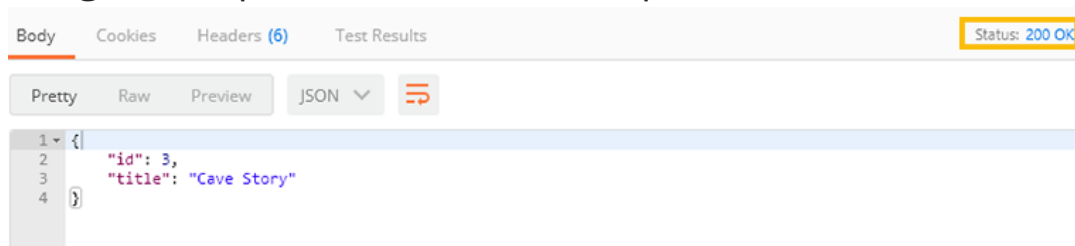



```
1 // update a game
2 app.put('/api/games/:id', (req, res) => {
3   const game = games.find(g => g.id === parseInt(req.params.id))
4   if (!game) return res.status(404).send('The game does not exist')
5
6   const schema = {
7     title: Joi.string().min(2).required()
8   };
9
10  const result = Joi.validate(req.body, schema);
11  if (result.error) {
12    res.status(400).send(result.error)
13  }
14
15  game.title = req.body.title;
16  res.send(game);
17 });
```

Let's test it out! We will send a PUT request to the server specifying the id of 3, and passing a json object in the body of the request with a new title for this game. Right now, the game with the id of 3 is 'Donkey Kong'. We will try to change it to 'Cave Story'.



We get a response back like we expect.



Finally, we just want to again make a GET request to the collection in our browser and game 3 should now be 'Cave Story'.

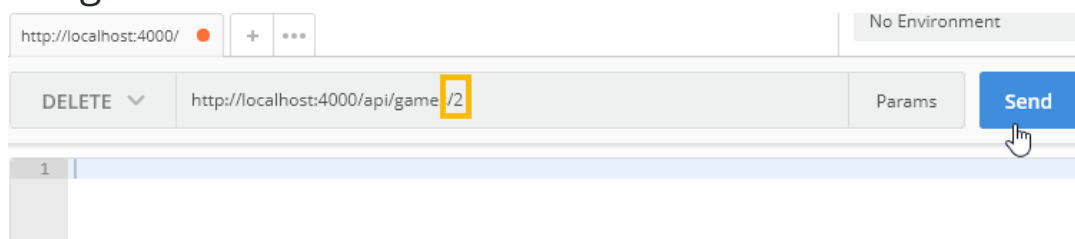


HTTP Delete Requests

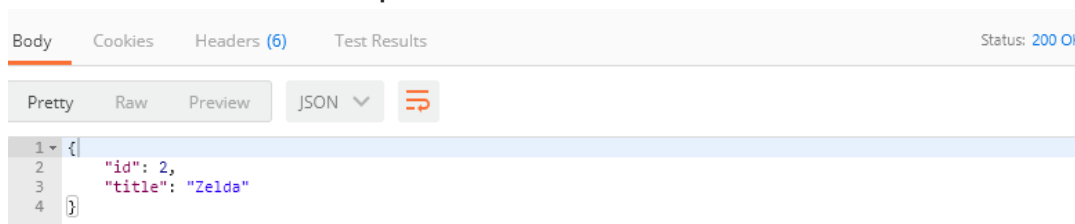
Finally, we will learn how to implement the *Delete* of crud operations in our rest api. We can follow a similar logic as updating a course. Here is the code we can add.

```
1 // delete a game
2 app.delete('/api/games/:id', (req, res) => {
3   const game = games.find(g => g.id === parseInt(req.params.id));
4   if (!game) return res.status(404).send('The game does not exist');
5
6   const index = games.indexOf(game);
7   games.splice(index, 1);
8
9   res.send(game);
10 });
```

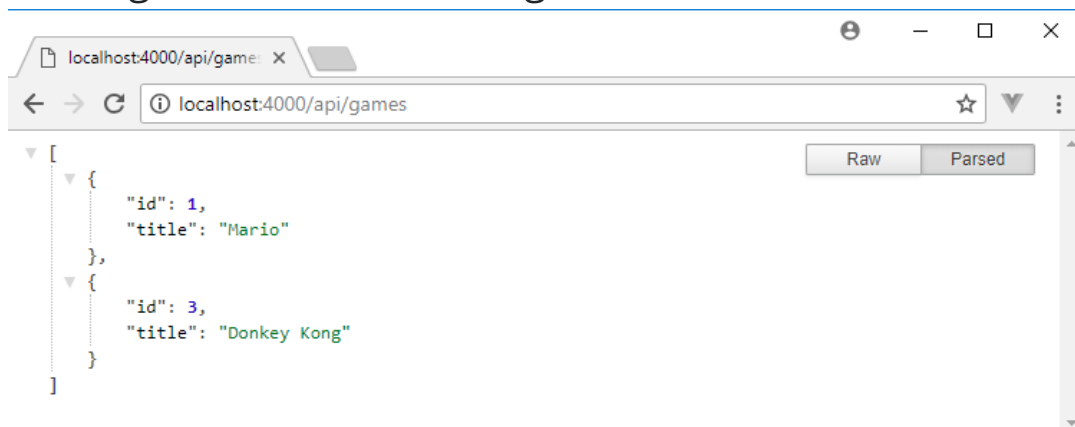
Now, let's send a DELETE request in postman to delete the game with an id of 2.



We get back the game which was deleted as a response, which is what we expect.



Finally, we once again make a GET request to the api to list all of our games. We can see that game 2 is now missing. Oh no! Zelda don't go!



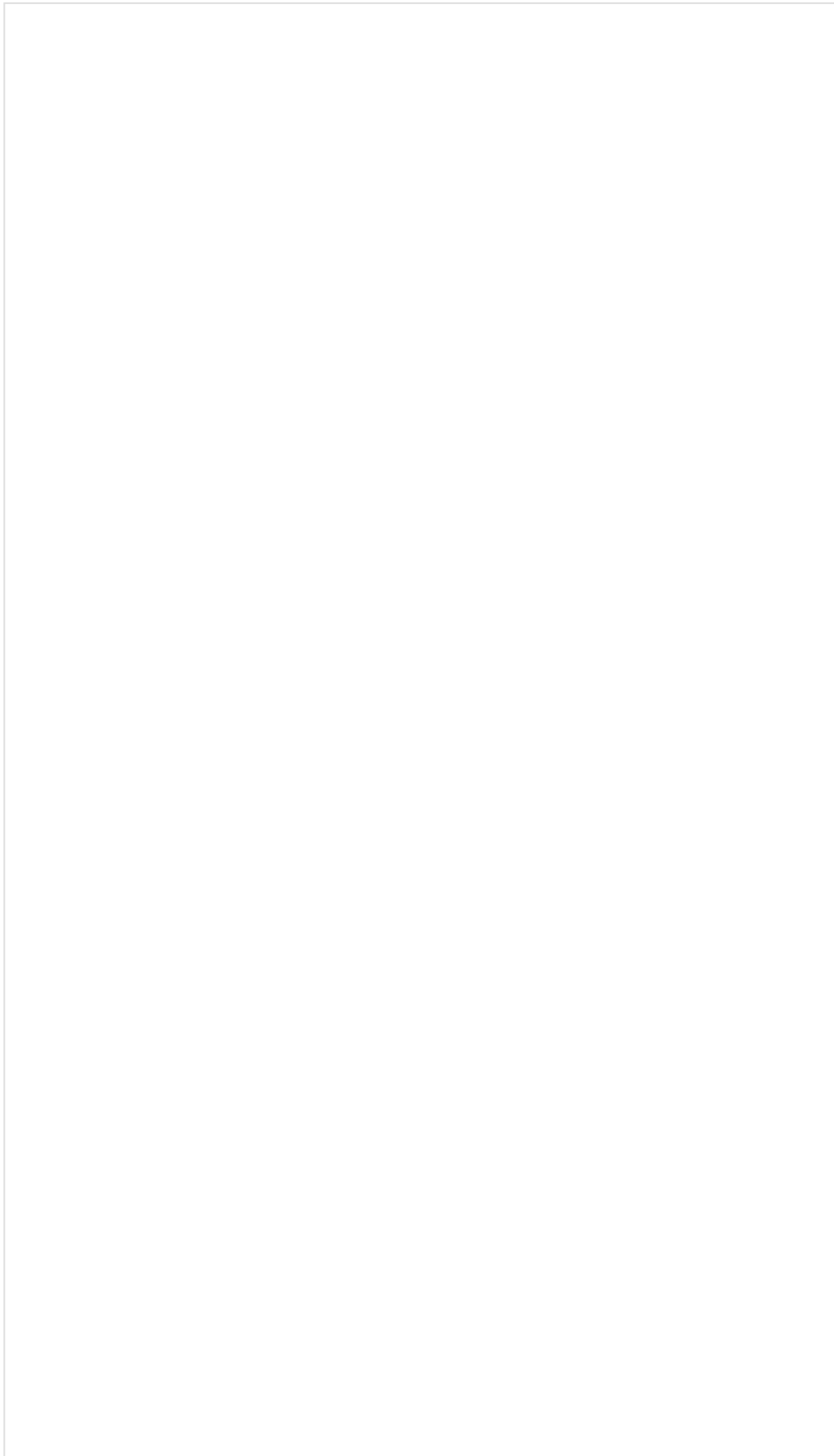
Node.js Express Rest Api Tutorial Summary

In this tutorial we learned all about setting up a simple REST api using Node.js and Express together. We covered the main verbs to use such as GET, POST, PUT, and DELETE, as well as all of the CRUD operations. Express makes it pretty easy to set these up with `app.get()`, `app.post()`, `app.put()`, and `app.delete()`. In addition, we can use [Mongoose for CRUD](https://vegibit.com/node-js-express-rest-api-tutorial/).

Some Key Points To Remember

- **REST** defines a set of conventions for creating HTTP services:
 - **POST**: to *create* a resource
 - **GET**: to *read* a resource
 - **PUT**: to *update* a resource
 - **DELETE**: to *delete* a resource
- You can use **Express** for building web servers with Node.js.
- **Nodemon** is a great way to watch for changes in files and automatically restart the node process.
- Environment variables can store various settings for an application. To read an environment variable, use **process.env**.
- Never trust data sent by the client. Perform input validation using **Joi** instead.

Useful Snippets For Express and REST



```
1 // Creating a web server
2 const express = require('express');
3 const app = express();
4
5 // Creating a resource
6 app.post('/api/resources', (req, res) => {
7   // Create the resource and return the resource object
8   resn.send(resource);
9 });
10
11 // Getting all the resources
12 app.get('/api/resources', (req, res) => {
13   // To read query string parameters (?sortBy=title)
14   const sortBy = req.query.sortBy;
15   // Return the resources
16   res.send(resources);
17 });
18
19 // Getting a single resource
20 app.get('/api/resources/:id', (req, res) => {
21   const resourceId = req.params.id;
22   // Lookup the resource and if not found, return 404
23   res.status(404).send('Resource not found. ');
24   // Else, return the resource object
25   res.send(resource);
26 });
27
28 // Updating a resource
29 app.put('/api/resources/:id', (req, res) => {
30   // If resource not found, return 404, otherwise update
31   // and return the updated object.
32 });
33
34 // Deleting a resource
35 app.delete('/api/resources/:id', (req, res) => {
36   // If resource not found, return 404, otherwise delete
37   // and return the deleted object.
38 });
39
40 // Listen on port 4000
41 app.listen(4000, () => console.log('Listening...'));
42
43 // Reading the port from an environment variable
44 const port = process.env.PORT || 5000;
45 app.listen(port);
```

