**# The ExpressJS Way To Write APIs

This article will cover the basics of express from the perspective of a beginner without concerning its self with the underlying mechanisms...

The ExpressJS Way To Write APIs

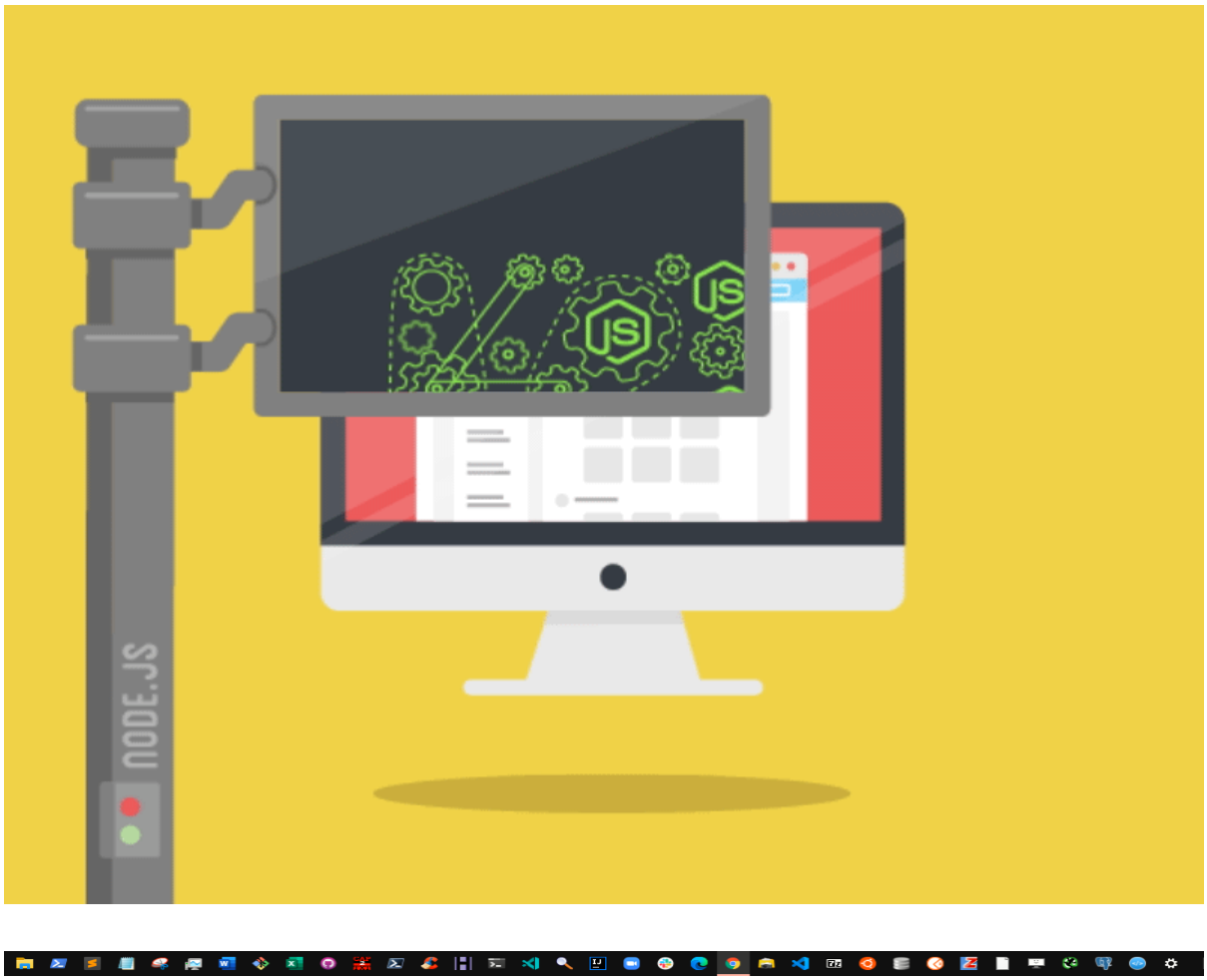
This article will cover the basics of express from the perspective of a beginner without concerning its self with the underlying mechanisms and theory that underlies the application of the framework.

For starters, what is express JS¿

When introduced, node.js gave developers the chance to use JavaScript to write software that, up to that point, could only be written using lower level languages like C, C++, Java, Python...

This tutorial will cover how to write **web services** that can communicate with clients (the front end application) using **J**ava**S**cript **O**bject **N**otation (JSON).

- JavaScript is asynchronous, which allows us to take full advantage of the processor it's running on. Taking full advantage of the processor is crucial because the node process will be running on a single CPU.
- Using JavaScript gives us access to the npm repository. This repository is the largest ecosystem of useful libraries (most of them free to use) in **npm modules**.



Explain what Node.js is and its core features

Traditionally, developers only used the JavaScript language in web browsers. But, in 2009, **Node.js** was unveiled, and with it, the developer tool kit expanded greatly. Node.js gave developers the chance to use JavaScript to write software that, up to that point, could only be written using C, C++, Java, Python, Ruby, C#, and the like.

We will use Node to write server code. Specifically, web services that can communicate with clients using the JavaScript Object Notation (JSON) format for data interchange.

Some of the advantages of using Node.js for writing server-side code are:

- Uses the same programming language (JavaScript) and paradigm for both client and server. Using the same language, we minimize context switching and make it easy to share code between the client and the server.
- JavaScript is single-threaded, which removes the complexity involved in handling multiple threads.
- JavaScript is asynchronous, which allows us to take full advantage of the processor it's running on. Taking full advantage of the processor is crucial because the node process will be running on a single CPU.
- Using JavaScript gives us access to the npm repository. This repository is the largest ecosystem of useful libraries (most of them free to use) in **npm modules**.

Some of the disadvantages of using Node.js for writing server-side code are:

- By strictly using JavaScript on the server, we lose the ability to use the right tool (a particular language) for the job.
- Because JavaScript is single-threaded, we can't take advantage of servers with multiple cores/processors.
- Because JavaScript is asynchronous, it is harder to learn for developers that have only worked with languages that default to synchronous operations that block the execution thread.
- In the npm repository, there are often too many packages that do the same thing. This excess of packages makes it harder to choose one and, in some cases, may introduce vulnerabilities into our code.

To write a simple web server with `Node.js`:

1. Use Node's `HTTP` module to abstract away complex network-related operations.
2. Write the single ***request handler*** function to handle all requests to the server.

The request handler is a function that takes the `request` coming from the client and produces the `response`. The function takes two arguments: 1) an object representing the `request` and 2) an object representing the `response`.

This process works, but the resulting code is verbose, even for the simplest of servers. Also, note that when using only Node.js to build a server, we use a single request handler function for all requests.

Try It Out:

Using only Node.js, let's write a simple web server that returns a message. Create a folder for the server and add an `index.js` file inside.

Next, add the following code to the `index.js` file:

```
const http = require("http"); // built in node.js module to handle http traffic

const hostname = "127.0.0.1"; // the local computer where the server is running
const port = 3000; // a port we'll use to watch for traffic

const server = http.createServer((req, res) => {
    // creates our server
    res.statusCode = 200; // http status code returned to the client
    res.setHeader("Content-Type", "text/plain"); // inform the client that we'll
be returning text
    res.end("Hello World from Node\\n"); // end the request and send a response
with the specified message
});
```

```
    server.listen(port, hostname, () => {
        // start watching for connections on the port specified
        console.log(`Server running at <http://$>{hostname}:${port}/`);
```

```
});
```

Now navigate to the folder in a terminal/console window and type: `node index.js` to execute your file. A message that reads "*Server running at http://127.0.0.1:3000*" should be displayed, and the server is now waiting for connections.

Open a browser and visit: `http://localhost:3000`. `localhost` and the IP address `127.0.0.1` point to the same thing: your local computer. The browser should show the message: "*Hello World from Node*". There you have it, your first web server, built from scratch using nothing but `Node.js`.

Explain what Express is and its core features:

Node's built-in `HTTP` module provides a powerful way to build web applications and services. However, it requires a lot of code for everyday tasks like sending an HTML page to the browser.

Introducing Express, a light and unopinionated framework that **sits on top of Node.js**, making it easier to create web applications and services. Sending an HTML file or image is now a one-line task with the `sendFile` helper method in `Express`.

Ultimately, Express is **just a Node.js module** like any other module.

What can we do with Express? So many things! For example:

- Build web applications.
- Serve *Single Page Applications* (SPAs).
- Build RESTful web services that work with JSON.
- Serve static content, like HTML files, images, audio files, PDFs, and more.
- Power real-time applications using technologies like **Web Sockets** or **WebRTC**.

Some of the benefits of using Express are that it is:

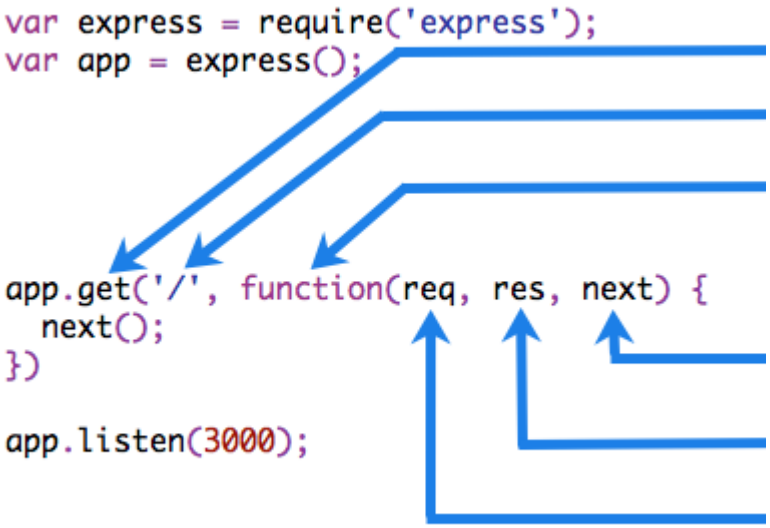
- Simple
- Unopinionated
- Extensible
- Light-weight
- Compatible with connect middleware (Links to an external site.). This compatibility means we can tap into an extensive collection of modules written for `connect`.
- All packaged into a clean, intuitive, and easy-to-use API.
- Abstracts away common tasks (writing web applications can be verbose, hence the need for a library like this).

Some of the drawbacks of Express are:

- It's not a one-stop solution. Because of its simplicity, it does very little out of the box. Especially when compared to frameworks like **Ruby on Rails** and **Django**.
- We are forced to make more decisions due to the flexibility and control it provides.

Main Features of Express

Writing middleware for use in Express apps
*Middleware functions are functions that have access to the request object (req), the response object (res), and the...*expressjs.com



Middleware

Middleware functions can get the request and response objects, operate on them, and (when specified) trigger some action. Examples are logging or security.

Express's middleware stack is an array of functions.

Middleware *can* change the request or response, but it doesn't have to.

Routing

Routing is a way to select which request handler function is executed. It does so based on the URL visited and the HTTP method used. Routing provides a way to break an application into smaller parts.

Routers for Application Modularity

We can break up applications into routers. We could have a router to serve our SPA and another router for our API. Each router can have its own middleware and routing. This combination provides improved functionality.

Convenience Helpers

Express has many helpers that provide out of the box functionality to make writing web applications and API servers easier.

A lot of those helpers are extension methods added to the request and response objects.

Examples from the Api Reference (Links to an external site.) include: `response.redirect()`, `response.status()`, `response.send()`, and `request.ip`.

Views

Views provide a way to dynamically render HTML on the server and even generate it using other languages.

Try It:

Let's write our first server using Express:

- Create a new file called `server.js` to host our server code.
- Type `npm init -y` to generate a `package.json`.
- Install the `express` npm module using: `npm install express`.

Inside `server.js` add the following code:

```
const express = require('express'); // import the express package

const server = express(); // creates the server

// handle requests to the root of the api, the / route
server.get('/', (req, res) => {
  res.send('Hello from Express');
});

// watch for connections on port 5000
server.listen(5000, () =>
  console.log('Server running on <http://localhost:5000>')
);
```

Run the server by typing: `node server.js` and visit `http://localhost:5000` in the browser.

To stop the server, type `Ctrl + c` at the terminal window.

Create an API that can respond to GET requests

The steps necessary to build a simple Web API that returns the string "Hello World" on every incoming `GET` request. The program should return the string every time a request comes into the root route ("/"). For now, you don't need to code along, just read through the steps.

To make things easier, we'll use an existing repository as the base for our API. Later in the week, as we learn more about Node.js and Express, we'll create an API from scratch.

To build our first API, we will:

1. clone the node-express-mini repository (Links to an external site.) to a folder on our computer.
2. Navigate into the folder using `cd`.

3. Use `npm install` to download all dependencies.
4. Add a file called `index.js` at the folder's root, next to the `package.json` file.
5. Open the `index.js` file using our favorite code editor.
6. Add the following code:

```
    // require the express npm module, needs to be added to the project using "npm
install express"
    const express = require('express');



    // creates an express application using the express module
    const server = express();



    // configures our server to execute a function for every GET request to "/"
    // the second argument passed to the .get() method is the "Route Handler
Function"
    // the route handler function will run on every GET request to "/"
    server.get('/', (req, res) => {
      // express will pass the request and response objects to this function
      // the .send() on the response object can be used to send a response to the
client
      res.send('Hello World');
    });



    // once the server is fully configured we can have it "listen" for connections
on a particular "port"
    // the callback function passed as the second argument will run once when the
server starts
    server.listen(8000, () => console.log('API running on port 8000'));
```

make sure to save your changes to `index.js`.

We are using the `express` npm module in our code, so we need to add it as a dependency to our project. To do this:

- Open a terminal/console/command prompt window and navigate to the root of our project.
- Add express to our `package.json` file by typing `npm install express`.

Now we're ready to test our API!

In the terminal, still at the root of our project:

- Type: `npm run server` to run our API. The message "*Api running on port 8000*" should appear on the terminal.
- Open a web browser and navigate to "http://localhost:8000".

There we have it, our first API!

A lot is going on in those few lines of code (only six lines if we remove the comments and white space). We will cover every piece of it in detail over the following modules, but here is a quick rundown of the most important concepts.

First, we used `require()` to **import** the `express module` and make it available to our application. `require()` is similar to the `import` keyword we have used before. The line `const express = require('express');` is equivalent to `import express from 'express';` if we were using ES2015 syntax.

The following line creates our Express application. The return of calling `express()` is an instance of an Express application that we can use to configure our **server** and, eventually, start listening for and responding to requests. Notice we use the word server, not API. An Express application is generic, which means we can use it to serve static content (HTML, CSS, audio, video, PDFs, and more). We can also use an Express application to serve dynamically generated web pages, build real-time communications servers, and more. We will use it statically to accept requests from clients and respond with data in JSON format.

An Express application publishes a set of methods we can use to configure functions. We are using the `.get()` method to set up a **route handler** function that will run on every `GET` request. As a part of this handler function, we specify the URL which will trigger the request. In this case, the URL is the site's root (represented by a `/`). There are also methods to handle the `POST`, `PUT`, and `DELETE` HTTP verbs.

The first two arguments passed by `express` to a route handler function are 1) an object representing the `request` and 2) an object representing the `response`. Express expands those objects with a set of useful properties and methods. Our example uses the `.send()` method of the response object to specify the data we will send to the client as the response body. You can call the first two arguments anything you want, but it is prevalent to see them dubbed `req` and `res`.

That's all the configuring we need to do for this first example We'll see other ways of configuring our server as we go forward.

After configuring the server, it's time to turn it on. We use the `.listen()` method to monitor a port on the computer for any incoming connections and respond to those we have configured. Our server will only respond to `GET` requests made to the `/` route on port `8000`.

That's it for our first Web API, and now it's time for you to follow along as we add a new **endpoint** to our server that returns JSON data!

Try It Out:

Let's try returning JSON instead of just a simple string.

Please follow the steps outlined on the overview, but, to save time, copy and paste the content of `index.js` instead of typing it. Then run your API through a browser to make sure it works.

Now follow along as we code a new *endpoint* that returns an array of movie characters in JSON format.

The first step is to define a new *route handler* to respond to `GET` requests at the `/hobbits` endpoint.

```
server.get('/hobbits', (req, res) => {
  // route handler code here
});
```

```
Next, we define the return data that our endpoint will send back to the client. We
do this inside the newly defined route handler function.
```js
    const hobbits = [
      {
        id: 1,
        name: 'Samwise Gamgee',
      },
      {
        id: 2,
        name: 'Frodo Baggins',
      },
    ];
```

Now we can return the `hobbits` array. We could use `.send(hobbits)` as we did for the string on the `/` endpoint, but this time we'll learn about two other useful methods we find in the response object.

```
    res.status(200).json(hobbits);
```

We should provide as much useful information as possible to the clients using our API. One such piece of data is the `HTTP status code` that reflects the client's operation outcome. In this case, the client is trying to get a list of a particular `resource`, a `hobbits` list. Sending back a `200 OK` status code communicates to the client that the operation was successful.

We will see other status codes as we continue to build new endpoints during this week. You can see a list by following this link to the documentation about HTTP Response Codes on the Mozilla Developer Network site (Links to an external site.).

We can use the `.status()` method of the response object to send any valid `HTTP status code`.

We are also chaining the `.json()` method of the response object. We do this to communicate to both the client making the request and the next developer working with this code that we intend to send the data in `JSON format`.
```

> The complete code for `index.js` should now look like so:

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

```
const server = express();


server.get('/', (req, res) => {
  res.send('Hello World');
});

server.get('/hobbits', (req, res) => {
  const hobbits = [
    {
      id: 1,
      name: 'Samwise Gamgee',
    },
    {
      id: 2,
      name: 'Frodo Baggins',
    },
  ];

  res.status(200).json(hobbits);
});


server.listen(8000, () => console.log('API running on port 8000'));
```

```
#### Now we can visit `http://localhost:8000/hobbits` in our browser, and we
should get back our JSON array.

If you are using the Google Chrome browser, <a
href="https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefha
kgolnmc" class="markup--anchor markup--p-anchor">this extension (Links to an
external site.)</a> can format the JSON data in a more readable fashion.

Congratulations! You just built an API that can return data in JSON format.

<figure><img src="https://cdn-images-
1.medium.com/max/1200/1*nGyJHK1Q_sSB6fjbBbF3xA.png" class="graf-image" /></figure>

### Let's look at a basic example of routing in action.

First, to make our Express application respond to `GET` requests on different
URLs, add the following endpoints:
```

```
```js

    // this request handler executes when making a GET request to /about
    server.get('/about', (req, res) => {
      res.status(200).send('<h1>About Us</h1>');
    });

```js

    // this request handler executes when making a GET request to /contact
    server.get('/contact', (req, res) => {
      res.status(200).send('<h1>Contact Form</h1>');
    });
```

Two things to note:

> 1.) We are using the same HTTP Method on both endpoints, but express looks at the URL and executes the corresponding request handler.

> 2.) We can return a string with valid HTML!

Open a browser and navigate to the `/about` and `/contact` routes. The appropriate route handler will execute.

Please follow along as we write endpoints that execute different request handlers on the same URL by changing the HTTP method.

Let's start by adding the following code after the `GET` endpoint to `/hobbits`:

```
    // this request handler executes when making a POST request to /hobbits
    server.post('/hobbits', (req, res) => {
      res.status(201).json({ url: '/hobbits', operation: 'POST' });
    });
```

Note that we return HTTP status code 201 (created) for successful `POST` operations.

Next, we need to add an endpoint for `PUT` requests to the same URL.

```
    // this request handler executes when making a PUT request to /hobbits
    server.put('/hobbits', (req, res) => {
      res.status(200).json({ url: '/hobbits', operation: 'PUT' });
    });
```

For successful `PUT` operations, we use HTTP Status Code 200 (OK).

Finally, let's write an endpoint to handle `DELETE` requests.

```
// this request handler executes when making a DELETE request to /hobbits
server.delete('/hobbits', (req, res) => {
  res.status(204);
});
```

We are returning HTTP Status Code 204 (No Content). Suppose you are returning any data to the client, perhaps the removed resource, on successful deletes. In that case, you'd change that to be 200 instead.

You may have noticed that we are not reading any data from the request, as that is something we'll learn later in the module. We are about to learn how to use a tool called `Postman` to test our `POST`, `PUT`, and `DELETE` endpoints.

Reading and Using Route Parameters

Let's revisit our `DELETE` endpoint.

```
server.delete('/hobbits', (req, res) => {
  res.status(204);
});
```

How does the client let the API know which hobbit should be deleted or updated? One way, the one we'll use, is through `route parameters`. Let's add support for route parameters to our `DELETE` endpoint.

We define route parameters by adding it to the URL with a colon (`:`) in front of it. Express adds it to the `.params` property part of the request object. Let's see it in action:

```
server.delete('/hobbits/:id', (req, res) => {
  const id = req.params.id;
  // or we could destructure it like so: const { id } = req.params;
  res.status(200).json({
    url: `/hobbits/${id}`,
    operation: `DELETE for hobbit with id ${id}`,
  });
});
```

This route handler will execute every `DELETE` for a URL that begins with `/hobbits/` followed by any value. So, `DELETE` requests to `/hobbits/123` and `/hobbits/frodo` will both trigger this request handler. The value passed after `/hobbits/` will end up as the `id` property on `req.params`.

The value for a route parameter will always be `string`, even if the value passed is numeric. When hitting `/hobbits/123` in our example, the type of `req.params.id` will be `string`.

Express routing has support for multiple route parameters. For example, defining a route URL that reads `/hobbits/:id/friends/:friendId`, will add properties for `id` and `friendId` to `req.params`.

Using the Query String

The query string is another strategy using the URL to pass information from clients to the server. The query string is structured as a set of key/value pairs. Each pair takes the form of `key=value`, and pairs are separated by an `&`. To mark the beginning of the query string, we add `?` and the end of the URL, followed by the set of key/value pairs.

An example of a query string would be: `https://www.google.com/search?q=lambda&tbo=1`. The query string portion is `?q=lambda&tbo=1` and the key/value pairs are `q=lambda` and `tbo=1`.

Let's add sorting capabilities to our API. We'll provide a way for clients to hit our `/hobbits` and pass the field they want to use to sort the responses, and our API will sort the data by that field in ascending order.

Here's the new code for the `GET /hobbits` endpoint:

```
server.get('/hobbits', (req, res) => {
  // query string parameters get added to req.query
  const sortField = req.query.sortby || 'id';
  const hobbits = [
    {
      id: 1,
      name: 'Samwise Gamgee',
    },
    {
      id: 2,
      name: 'Frodo Baggins',
    },
  ];


  // apply the sorting
  const response = hobbits.sort(
    (a, b) => (a[sortField] < b[sortField] ? -1 : 1)
  );


  res.status(200).json(response);
});
```

Visit `localhost:8000/hobbits?sortby=name`, and the list should be sorted by `name`. Visit `localhost:8000/hobbits?sortby=id`, and the list should now be sorted by `id`. If no `sortby` parameter is provided, it should default to sorting by `id`.

To read values from the query string, we use the `req.query` object added by Express. There will be a key and a value in the `req.query` object for each key/value pair found in the query string.

The parameter's value will be of type `array` if more than one value is passed for the same key and `string` when only one value is passed. For example, in the query string `?id=123`, `req.query.id` will be a string, but for `?id=123&id=234`, it will be an array.

Another gotcha is that the names of query string parameters are case sensitive, `sortby` and `sortBy` are two different parameters.

The rest of the code sorts the array before sending it back to the client.

Reading Data from the Request Body

We begin by taking another look at the `POST /hobbits` endpoint. We need to read the hobbit's information to add it to the `hobbits` array. Let's do that next:

```
// add this code right after const server = express();
server.use(express.json());


let hobbits = [
  {
    id: 1,
    name: 'Bilbo Baggins',
    age: 111,
  },
  {
    id: 2,
    name: 'Frodo Baggins',
    age: 33,
  },
];
let nextId = 3;

// and modify the post endpoint like so:
server.post('/hobbits', (req, res) => {
  const hobbit = req.body;
  hobbit.id = nextId++;

  hobbits.push(hobbit);
```

```
    res.status(201).json(hobbits);
  });
```

To make this work with the hobbits array, we first move it out of the get endpoint into the outer scope. Now we have access to it from all route handlers.

Then we define a variable for manual id generation. When using a database, this is not necessary as the database management system generates ids automatically.

To read data from the request body, we need to do two things:

- Add the line: `server.use(express.json());` after the express application has been created.
- Read the data from the body property that Express adds to the request object. Express takes all the client's information from the body and makes it available as a nice JavaScript object.

Note that we are skipping data validation to keep this demo simple, but in a production application, you would validate before attempting to save to the database.

Let's test it using Postman:

- Change the method to POST.
- Select the `Body` tab underneath the address bar.
- Click on the `raw` radio button.
- From the dropdown menu to the right of the `binary` radio button, select `JSON (application/json).
- Add the following JSON object as the body:

```
{
  "name": "Samwise Gamgee",
  "age": 30
}
```

Click on `Send`, and the API should return the list of hobbits, including Sam!

Try It:

Please code along as we implement the `PUT` endpoint and a way for the client to specify the sort direction.

Implement Update Functionality

Let's continue practicing reading route parameters and information from the request body. Let's take a look at our existing PUT endpoint:

```
server.put('/hobbits', (req, res) => {
  res.status(200).json({ url: '/hobbits', operation: 'PUT' });
});
```

We start by adding support for a route parameter the clients can use to specify the id of the hobbit they intend to update:

```
server.put('/hobbits/:id', (req, res) => {
  res.status(200).json({ url: '/hobbits', operation: 'PUT' });
});
```

Next, we read the hobbit information from the request body using `req.body` and use it to update the existing hobbit.

```
server.put('/hobbits/:id', (req, res) => {
  const hobbit = hobbits.find(h => h.id == req.params.id);

  if (!hobbit) {
    res.status(404).json({ message: 'Hobbit does not exist' });
  } else {
    // modify the existing hobbit
    Object.assign(hobbit, req.body);
    res.status(200).json(hobbit);
  }
});
```

Concentrate on the code related to reading the `id` from the `req.params` object and reading the hobbit information from `req.body`. The rest of the code will change as this is a simple example using an in-memory array. Most production APIs will use a database.

TBC.....

If you found this guide helpful feel free to checkout my GitHub/gists where I host similar content:

bgoonz's gists
Instantly share code, notes, and snippets. Web Developer, Electrical Engineer JavaScript | CSS | Bootstrap | Python |...gist.github.com

bgoonz --- Overview
Web Developer, Electrical Engineer JavaScript | CSS | Bootstrap | Python | React | Node.js | Express | Sequelize...github.com

Discover More:

Web-Dev-Hub

*Memoization, Tabulation, and Sorting Algorithms by Example Why is looking at runtime not a reliable method of...*bgoonz-blog.netlify.app

Update(Bonus Best Practices):

Things to do in your code

Here are some things you can do in your code to improve your application's performance:

- Use gzip compression
- Don't use synchronous functions
- Do logging correctly
- Handle exceptions properly

Use gzip compression

Gzip compressing can greatly decrease the size of the response body and hence increase the speed of a web app. Use the compression middleware for gzip compression in your Express app. For example:

```
let compression = require('compression')
let express = require('express')
let app = express()
app.use(compression())
```

For a high-traffic website in production, the best way to put compression in place is to implement it at a reverse proxy level (see Use a reverse proxy). In that case, you do not need to use compression middleware. For details on enabling gzip compression in Nginx, see Module ngx_http_gzip_module in the Nginx documentation.

Don't use synchronous functions

Synchronous functions and methods tie up the executing process until they return. A single call to a synchronous function might return in a few microseconds or milliseconds, however in high-traffic websites, these calls add up and reduce the performance of the app. Avoid their use in production.

Although Node and many modules provide synchronous and asynchronous versions of their functions, always use the asynchronous version in production. The only time when a synchronous function can be justified is upon initial startup.

If you are using Node.js 4.0+ or io.js 2.1.0+, you can use the `--trace-sync-io` command-line flag to print a warning and a stack trace whenever your application uses a synchronous API. Of course, you wouldn't want to use this in production, but rather to ensure that your code is ready for production. See the node command-line options documentation for more information.

Do logging correctly

In general, there are two reasons for logging from your app: For debugging and for logging app activity (essentially, everything else). Using `console.log()` or `console.error()` to print log messages to the terminal is common practice in development. But these functions are synchronous when the destination is a terminal or a file, so they are not suitable for production, unless you pipe the output to another program.

For debugging

If you're logging for purposes of debugging, then instead of using `console.log()`, use a special debugging module like debug. This module enables you to use the DEBUG environment variable to control what debug messages are sent to `console.error()`, if any. To keep your app purely asynchronous, you'd still want to pipe `console.error()` to another program. But then, you're not really going to debug in production, are you?

For app activity

If you're logging app activity (for example, tracking traffic or API calls), instead of using `console.log()`, use a logging library like Winston or Bunyan. For a detailed comparison of these two libraries, see the StrongLoop blog post Comparing Winston and Bunyan Node.js Logging.

Handle exceptions properly

Node apps crash when they encounter an uncaught exception. Not handling exceptions and taking appropriate actions will make your Express app crash and go offline. If you follow the advice in Ensure your app automatically restarts below, then your app will recover from a crash. Fortunately, Express apps typically have a short startup time. Nevertheless, you want to avoid crashing in the first place, and to do that, you need to handle exceptions properly.

To ensure you handle all exceptions, use the following techniques:

- Use try-catch
- Use promises

Before diving into these topics, you should have a basic understanding of Node/Express error handling: using error-first callbacks, and propagating errors in middleware. Node uses an "error-first callback" convention for returning errors from asynchronous functions, where the first parameter to the callback function is the error object, followed by result data in succeeding parameters. To indicate no error, pass null as the first parameter. The callback function must correspondingly follow the error-first callback convention to meaningfully handle the error. And in Express, the best practice is to use the next() function to propagate errors through the middleware chain.

For more on the fundamentals of error handling, see:

- Error Handling in Node.js
- Building Robust Node Applications: Error Handling (StrongLoop blog)

What not to do

One thing you should *not* do is to listen for the `uncaughtException` event, emitted when an exception bubbles all the way back to the event loop. Adding an event listener for `uncaughtException` will change the default behavior of the process that is encountering an exception; the process will continue to run despite the

exception. This might sound like a good way of preventing your app from crashing, but continuing to run the app after an uncaught exception is a dangerous practice and is not recommended, because the state of the process becomes unreliable and unpredictable.

Additionally, using `uncaughtException` is officially recognized as crude. So listening for `uncaughtException` is just a bad idea. This is why we recommend things like multiple processes and supervisors: crashing and restarting is often the most reliable way to recover from an error.

We also don't recommend using domains. It generally doesn't solve the problem and is a deprecated module.

Use try-catch

Try-catch is a JavaScript language construct that you can use to catch exceptions in synchronous code. Use try-catch, for example, to handle JSON parsing errors as shown below.

Use a tool such as JSHint or JSLint to help you find implicit exceptions like reference errors on undefined variables.

Here is an example of using try-catch to handle a potential process-crashing exception. This middleware function accepts a query field parameter named "params" that is a JSON object.

```
app.get('/search', function (req, res) {
  // Simulating async operation
  setImmediate(function () {
    let jsonStr = req.query.params
    try {
      let jsonObj = JSON.parse(jsonStr)
      res.send('Success')
    } catch (e) {
      res.status(400).send('Invalid JSON string')
    }
  })
})
```

However, try-catch works only for synchronous code. Because the Node platform is primarily asynchronous (particularly in a production environment), try-catch won't catch a lot of exceptions.

Use promises

Promises will handle any exceptions (both explicit and implicit) in asynchronous code blocks that use `then()`. Just add `.catch(next)` to the end of promise chains. For example:

```
app.get('/', function (req, res, next) {
  // do some sync stuff
  queryDb()
    .then(function (data) {
      // handle data
```

```
        return makeCsv(data)
      })
      .then(function (csv) {
        // handle csv
      })
      .catch(next)
  })

  app.use(function (err, req, res, next) {
    // handle error
  })
```

Now all errors asynchronous and synchronous get propagated to the error middleware.

However, there are two caveats:

1. All your asynchronous code must return promises (except emitters). If a particular library does not return promises, convert the base object by using a helper function like Bluebird.promisifyAll().
2. Event emitters (like streams) can still cause uncaught exceptions. So make sure you are handling the error event properly; for example:
3. const wrap = fn => (...args) => fn(...args).catch(args[2])
4. app.get('/', wrap(async (req, res, next) => { const company = await getCompanyById(req.query.id) const stream = getLogoStreamById(company.id) stream.on('error', next).pipe(res) }))

The `wrap()` function is a wrapper that catches rejected promises and calls `next()` with the error as the first argument. For details, see Asynchronous Error Handling in Express with Promises, Generators and ES7.

For more information about error-handling by using promises, see Promises in Node.js with Q---An Alternative to Callbacks.

Things to do in your environment / setup

Here are some things you can do in your system environment to improve your app's performance:

- Set NODE_ENV to "production"
- Ensure your app automatically restarts
- Run your app in a cluster
- Cache request results
- Use a load balancer
- Use a reverse proxy

Set NODE_ENV to "production"

The NODE_ENV environment variable specifies the environment in which an application is running (usually, development or production). One of the simplest things you can do to improve performance is to set NODE_ENV to "production."

Setting NODE_ENV to "production" makes Express:

- Cache view templates.

- Cache CSS files generated from CSS extensions.
- Generate less verbose error messages.

Tests indicate that just doing this can improve app performance by a factor of three!

If you need to write environment-specific code, you can check the value of NODE_ENV with `process.env.NODE_ENV`. Be aware that checking the value of any environment variable incurs a performance penalty, and so should be done sparingly.

In development, you typically set environment variables in your interactive shell, for example by using `export` or your `.bash_profile` file. But in general you shouldn't do that on a production server; instead, use your OS's init system (systemd or Upstart). The next section provides more details about using your init system in general, but setting NODE_ENV is so important for performance (and easy to do), that it's highlighted here.

With Upstart, use the `env` keyword in your job file. For example:

```
# /etc/init/env.conf
 env NODE_ENV=production
```

For more information, see the Upstart Intro, Cookbook and Best Practices.

With systemd, use the `Environment` directive in your unit file. For example:

```
# /etc/systemd/system/myservice.service
Environment=NODE_ENV=production
```

For more information, see Using Environment Variables In systemd Units.

Ensure your app automatically restarts

In production, you don't want your application to be offline, ever. This means you need to make sure it restarts both if the app crashes and if the server itself crashes. Although you hope that neither of those events occurs, realistically you must account for both eventualities by:

- Using a process manager to restart the app (and Node) when it crashes.
- Using the init system provided by your OS to restart the process manager when the OS crashes. It's also possible to use the init system without a process manager.

Node applications crash if they encounter an uncaught exception. The foremost thing you need to do is to ensure your app is well-tested and handles all exceptions (see handle exceptions properly for details). But as a fail-safe, put a mechanism in place to ensure that if and when your app crashes, it will automatically restart.

Use a process manager

In development, you started your app simply from the command line with `node server.js` or something similar. But doing this in production is a recipe for disaster. If the app crashes, it will be offline until you restart it. To ensure your app restarts if it crashes, use a process manager. A process manager is a "container" for

applications that facilitates deployment, provides high availability, and enables you to manage the application at runtime.

In addition to restarting your app when it crashes, a process manager can enable you to:

- Gain insights into runtime performance and resource consumption.
- Modify settings dynamically to improve performance.
- Control clustering (StrongLoop PM and pm2).

The most popular process managers for Node are as follows:

- StrongLoop Process Manager
- PM2
- Forever

For a feature-by-feature comparison of the three process managers, see http://strong-pm.io/compare/. For a more detailed introduction to all three, see Process managers for Express apps.

Using any of these process managers will suffice to keep your application up, even if it does crash from time to time.

However, StrongLoop PM has lots of features that specifically target production deployment. You can use it and the related StrongLoop tools to:

- Build and package your app locally, then deploy it securely to your production system.
- Automatically restart your app if it crashes for any reason.
- Manage your clusters remotely.
- View CPU profiles and heap snapshots to optimize performance and diagnose memory leaks.
- View performance metrics for your application.
- Easily scale to multiple hosts with integrated control for Nginx load balancer.

As explained below, when you install StrongLoop PM as an operating system service using your init system, it will automatically restart when the system restarts. Thus, it will keep your application processes and clusters alive forever.

Cache request results

Another strategy to improve the performance in production is to cache the result of requests, so that your app does not repeat the operation to serve the same request repeatedly.

Use a caching server like Varnish or Nginx (see also Nginx Caching) to greatly improve the speed and performance of your app.

Use a load balancer

No matter how optimized an app is, a single instance can handle only a limited amount of load and traffic. One way to scale an app is to run multiple instances of it and distribute the traffic via a load balancer. Setting up a load balancer can improve your app's performance and speed, and enable it to scale more than is possible with a single instance.

A load balancer is usually a reverse proxy that orchestrates traffic to and from multiple application instances and servers. You can easily set up a load balancer for your app by using Nginx or HAProxy.

With load balancing, you might have to ensure that requests that are associated with a particular session ID connect to the process that originated them. This is known as *session affinity*, or *sticky sessions*, and may be addressed by the suggestion above to use a data store such as Redis for session data (depending on your application). For a discussion, see Using multiple nodes.

Use a reverse proxy

A reverse proxy sits in front of a web app and performs supporting operations on the requests, apart from directing requests to the app. It can handle error pages, compression, caching, serving files, and load balancing among other things.

Handing over tasks that do not require knowledge of application state to a reverse proxy frees up Express to perform specialized application tasks. For this reason, it is recommended to run Express behind a reverse proxy like Nginx or HAProxy in production.