Introduction to the Basic Node and Express Challenges

Node.js is a JavaScript tool that allows developers to write backend (server-side) programs in JavaScript. Node.js comes with a handful of built-in modules—small, independent programs—that help facilitate this purpose. Some of the core modules include:

* HTTP: a module that acts as a server
* File System: a module that reads and modifies files
* Path: a module for working with directory and file paths
* Assertion Testing: a module that checks code against prescribed constraints

Express, while not included with Node.js, is another module often used with it. Express runs between the server created by Node.js and the frontend pages of a web application.

**Express also handles an application's routing. Routing directs users to the correct page based on their interaction with the application.**  
  
While there are alternatives to using Express, its simplicity makes it a good place to begin when learning the interaction between a backend powered by Node.js and the frontend.

## Start a Working Express Server

In the first two lines of the file myApp.js you can see how it’s easy to create an Express app object. This object has several methods, and we will learn many of them in these challenges. One fundamental method is app.listen(port). It tells your server to listen on a given port, putting it in running state. You can see it at the bottom of the file. It is inside comments because for testing reasons we need the app to be running in background. All the code that you may want to add goes between these two fundamental parts. Glitch stores the port number in the environment variable process.env.PORT. Its value is 3000.

Let’s serve our first string! In Express, routes takes the following structure: app.METHOD(PATH, HANDLER).

METHOD is an http method in lowercase.

PATH is a relative path on the server (it can be a string, or even a regular expression). HANDLER is a function that Express calls when the route is matched.

Handlers take the form function(req, res) {...}, where req is the request object, and res is the response object. For example, the handler

function(req, res) {  
res.send('Response String');  
}

will serve the string 'Response String'.

Use the app.get()method to serve the string Hello Express, to GET requests matching the / root path. Be sure that your code works by looking at the logs, then see the results in your browser, clicking the button ‘Show Live’ in the Glitch UI.

## Serve an HTML File

You probably need to comment out the last challenge. If you have a website and want to serve an index.html file you probably want to put this in a public folder. This is to ensure the public doesn’t see something you dont want them to, and it sometimes is called “public” or “views,” but you can technically call it whatever you want.

To serve an index.html in a folder called “public” at the root domain you would do so like this:

app.get("/", function(req, res) {

res.sendFile( \_\_dirname + "/public/index.html");

});

Note: \_\_dirname returns the root directory is a best practice for node developers.

## Basic Node and Express - Serve Static Assets

An HTML server usually has one or more directories that are accessible by the user. You can place there the static assets needed by your application (stylesheets, scripts, images). In Express you can put in place this functionality using the middleware express.static(path), where the parameter is the absolute path of the folder containing the assets.

If you don’t know what a middleware is, don’t worry. We’ll discuss about it later in details. Basically middlewares are functions that intercept route handlers, adding some kind of information. A middleware needs to be mounted using the method app.use(path, middlewareFunction). The first path argument is optional. If you don’t pass it, the middleware will be executed for all the requests.

Mount the express.static()middleware for all the requests with app.use(). The absolute path to the assets folder is \_\_dirname + /public.

Now your app should be able to serve a CSS stylesheet. From outside the public folder will appear mounted to the root directory. Your front-page should look a little better now!

## Serve JSON on a Specific Route

While an HTML server serves (you guessed it!) HTML, an API serves data. A REST (REpresentational State Transfer) API allows data exchange in a simple way, without the need for clients to know any detail about the server.

The client only needs to know where the resource is (the URL), and the action it wants to perform on it (the verb). The GET verb is used when you are fetching some information, without modifying anything. These days, the preferred data format for moving information around the web is JSON. Simply put, JSON is a convenient way to represent a JavaScript object as a string, so it can be easily transmitted.

Let's create a simple API by creating a route that responds with JSON at the path /json. You can do it as usual, with the app.get() method. Inside the route handler use the method res.json(), passing in an object as an argument. This method closes the request-response loop, returning the data.

Behind the scenes it converts a valid JavaScript object into a string, then sets the appropriate headers to tell your browser that you are serving JSON, and sends the data back. A valid object has the usual structure {key: data}. Data can ba a number, a string, a nested object or an array. Data can also be a variable or the result of a function call; in which case it will be evaluated before being converted into a string.

Serve the object {"message": "Hello json"} as a response in JSON format, to the GET requests to the route /json. Then point your browser to your-app-url/json, you should see the message on the screen.

## Basic Node and Express - Use the .env File

The .env file is a hidden file that is used to pass environment variables to your application. This file is secret, no one but you can access it, and it can be used to store data that you want to keep private or hidden. For example, you can store API keys from external services or your database URI. You can also use it to store configuration options. By setting configuration options, you can change the behavior of your application, without the need to rewrite some code.

The environment variables are accessible from the app as process.env.VAR\_NAME. The process.env object is a global Node object, and variables are passed as strings. By convention, the variable names are all uppercase, with words separated by an underscore. The .envis a shell file, so you don’t need to wrap names or values in quotes. It is also important to note that there cannot be space around the equals sign when you are assigning values to your variables, e.g. VAR\_NAME=value. Usually, you will put each variable definition on a separate line.

## Implement a Root-Level Request Logger Middleware(Node2.js)

Before we introduced the express.static()middleware function. Now it’s time to see what middleware is, in more detail. Middleware functions are functions that take 3 arguments: the request object, the response object, and the next function in the application’s request-response cycle.

These functions execute some code that can have side effects on the app, and usually add informations to the request or response objects. They can also end the cycle sending the response, when some condition is met. If they don’t send the response, when they are done they start the execution of the next function in the stack. This is triggered calling the 3rd argument next(). More information in the [express documentation](http://expressjs.com/en/guide/using-middleware.html).

Look at the following example :

function(req, res, next) {  
console.log("I'm a middleware...");  
next();  
}

Let’s suppose we mounted this function on a route. When a request matches the route, it displays the string “I’m a middleware…”. Then it executes the next function in the stack.

In this exercise we are going to build a root-level middleware. As we have seen in challenge 4, to mount a middleware function at root level we can use the method app.use(<mware-function>). In this case the function will be executed for all the requests, but you can also set more specific conditions. For example, if you want a function to be executed only for POST requests, you could use app.post(<mware-function>). Analogous methods exist for all the http verbs (GET, DELETE, PUT, …).

Build a simple logger. For every request, it should log in the console a string taking the following format: method path - ip. An example would look like: GET /json - ::ffff:127.0.0.1. Note that there is a space between method and path and that the dash separating path and ip is surrounded by a space on either side. You can get the request method (http verb), the relative route path, and the caller’s ip from the request object, using req.method, req.path and req.ip. Remember to call next()when you are done, or your server will be stuck forever. Be sure to have the ‘Logs’ opened, and see what happens when some request arrives…

Hint: Express evaluates functions in the order they appear in the code. This is true for middleware too. If you want it to work for all the routes, it should be mounted before them.

## Chain Middleware to Create a Time Server

Middleware can be mounted at a specific route using app.METHOD(path, middlewareFunction). Middleware can also be chained inside route definition.

Look at the following example:

app.get('/user', function(req, res, next) {  
req.user = getTheUserSync(); // Hypothetical synchronous operation  
next();  
}, function(req, res) {  
res.send(req.user);  
})

This approach is useful to split the server operations into smaller units. That leads to a better app structure, and the possibility to reuse code in different places. This approach can also be used to perform some validation on the data.

At each point of the middleware stack you can block the execution of the current chain and pass control to functions specifically designed to handle errors. Or you can pass control to the next matching route, to handle special cases. We will see how in the advanced Express section.

In the route app.get('/now', ...) chain a middleware function and the final handler. In the middleware function you should add the current time to the request object in the req.time key. You can use new Date().toString(). In the handler, respond with a JSON object, taking the structure {time: req.time}.

Hint: The test will not pass if you don’t chain the middleware. If you mount the function somewhere else, the test will fail, even if the output result is correct.

## Get Route Parameter Input from the Client(Node2.js)

When building an API, we have to allow users to comunicate us what they want to get from our service. For example, if the client is requesting information about a user stored in the database, they need a way to let us know which user they're interested in. One possible way to achieve this result is using route parameters. Route parameters are named segments of the URL, delimited by slashes (/). Each segment captures the value of the part of the URL which matches its position. The captured values can be found in the req.params object.

route\_path: '/user/:userId/book/:bookId'  
actual\_request\_URL: '/user/546/book/6754'   
req.params: {userId: '546', bookId: '6754'}

Build an echo server, mounted at the route GET /:word/echo. Respond with a JSON object, taking the structure {echo: word}. You can find the word to be repeated at req.params.word. You can test your route from your browser's address bar, visiting some matching routes, e.g. your-app-rootpath/freecodecamp/echo

## Get Query Parameter Input from the Client(Node3.js)

Another common way to get input from the client is by encoding the data after the route path, using a query string. The query string is delimited by a question mark (?), and includes field=value couples.

Each couple is separated by an ampersand (&). Express can parse the data from the query string, and populate the object req.query. Some characters cannot be in URLs, they have to be encoded in a [different format](https://en.wikipedia.org/wiki/Percent-encoding) before you can send them. If you use the API from JavaScript, you can use specific methods to encode/decode these characters.

route\_path: '/library'  
actual\_request\_URL: '/library?userId=546&bookId=6754'   
req.query: {userId: '546', bookId: '6754'}

Use body-parser to Parse POST Requests

Besides GET there is another common http verb, it is POST. POST is the default method used to send client data with HTML forms. In the REST convention POST is used to send data to create new items in the database (a new user, or a new blog post). We don’t have a database in this project, but we are going to learn how to handle POST requests anyway.

In these kind of requests the data doesn’t appear in the URL, it is hidden in the request body. This is a part of the HTML request, also called payload. Since HTML is text based, even if you don’t see the data, it doesn’t mean that they are secret. The raw content of an HTTP POST request is shown below:

POST /path/subpath HTTP/1.0  
From: john@example.com  
User-Agent: someBrowser/1.0  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 20  
name=John+Doe&age=25

As you can see the body is encoded like the query string. This is the default format used by HTML forms. With Ajax we can also use JSON to be able to handle data having a more complex structure. There is also another type of encoding: multipart/form-data. This one is used to upload binary files.

In this exercise we will use an urlencoded body.

To parse the data coming from POST requests, you have to install a package: the body-parser. This package allows you to use a series of middleware, which can decode data in different formats. See the docs [here](https://github.com/expressjs/body-parser).

Install the body-parser module in your package.json. Then require it at the top of the file. Store it in a variable named bodyParser.

The middleware to handle url encoded data is returned by bodyParser.urlencoded({extended: false}). extended=false is a configuration option that tells the parser to use the classic encoding. When using it, values can be only strings or arrays. The extended version allows more data flexibility, but it is outmatched by JSON.

Pass to app.use()the function returned by the previous method call. As usual, the middleware must be mounted before all the routes which need it.

## Get Data from POST Requests

Mount a POST handler at the path /name. It’s the same path as before. We have prepared a form in the html frontpage. It will submit the same data of exercise 10 (Query string). If the body-parser is configured correctly, you should find the parameters in the object req.body. Have a look at the usual library example:

route: POST '/library'  
urlencoded\_body: userId=546&bookId=6754   
req.body: {userId: '546', bookId: '6754'}

Respond with the same JSON object as before: {name: 'firstname lastname'}. Test if your endpoint works using the html form we provided in the app frontpage.

Tip: There are several other http methods other than GET and POST. And by convention there is a correspondence between the http verb, and the operation you are going to execute on the server. The conventional mapping is:

POST (sometimes PUT) - Create a new resource using the information sent with the request,

GET - Read an existing resource without modifying it,

PUT or PATCH (sometimes POST) - Update a resource using the data sent,

DELETE => Delete a resource.

There are also a couple of other methods which are used to negotiate a connection with the server. Except from GET, all the other methods listed above can have a payload (i.e. the data into the request body). The body-parser middleware works with these methods as well.