Express JS

A Fast, unopinionated, minimalist web framework for [Node.js](https://nodejs.org/en/)

# Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

# Installation

npm install express –SE

# Hello World

**var** express = require('express');  
**var** app = express();  
  
app.get('/', **function** (req, res) {  
 res.send({"message": "Hello World!"});  
});

The above example is a very simple example of a definition of route handler.

With app = express() we define a new instance of an express app. The next 3 lines define a path (‘/’) and a route to handle GET (app.get) requests to that route.

# The request object (req)

The request (usually *req* in express) object is the first parameter (usually) passed to a handler. It contains all information regarding the incoming request. Some of these are:

* The requested path
* The query string
* Request headers
* Protocol
* Hostname
* Etc

Basically if it has anything to do with the request it is in this object.

<http://expressjs.com/en/4x/api.html#req>

# The response object (res)

The response/res object is a similar to the request object. Except it contains information and helper methods pertaining to the response.

Some of the properties and methods are:

* Methods to set cookie
* Headers
* Methods to set headers
* The status code (get and set)
* res.send
* and more

<http://expressjs.com/en/4x/api.html#res> for more

# Routing

TODO

# Express.Router()

* Express.Router module allow us to create modular, isolated handlers.
* These handlers are then attached or mounted to the main app.
* This allows us to keep the main app clean and abstract by delegating actual routing to Router instances.

For example, we can encapsulate all functionality pertaining to blog posts, we can initialize an instance of Router and delegate any routes related to blog post resources to this router.

That router might look something like this:

router.route("/")  
 .get(**function** (req, res) {  
 res.send("Get all blog posts! - from router");  
  
 })  
 .post(**function** (req, res) {  
 res.send("Create a new post - from router! ");  
 });  
  
router.put("/:postId", **function** (req, res) {  
 res.send("Update post with postId");  
});

Then, in our main app we can do:

**var** posts = require("./posts");

app.get("/", **function** (req, res) {  
 res.send("Welcome to my blog!");  
});  
app.use("/posts", posts);

The obvious benefit of this is realised in large applications where one might have hundreds of routes. Keeping all the handlers to these routes would become a maintenance nightmare.

What this also means is we can plug this router in any other express application if we need to duplicate this functionality! We just have to *require* it in.

# Middleware

A middleware function is one that can *assist* in the processing of a request by performing a specific function without necessarily responding to the request – although it may. For example, we can have middleware to authenticate a user on every request. Or, we could have middleware to log every request that comes onto our site.

A middleware function may or may not respond to a request. In the case that it does not respond to the request, it will pass control over to the next middleware. If one middleware responds to a request then the processing pipeline ends and any middleware that was yet to be invoked is ignored.

To add middleware to a route is easy. We simply add it to the app:

**function logger**(req, res, next) {  
 console.log("%s %s - %s ", **new** Date().toISOString(), req.method, req.originalUrl);  
 next();  
}

app.use(logger);

Note the extra **next** parameter passed to our middleware function. This parameter refers to the next function in the middleware chain. In the above snippet, this middleware will log the date, method and url of the request and then pass control over to the next middleware, where actual processing of the request can occur.

**var** express = require("express");  
**var** posts = require("./posts");  
**var** users = require("./users");  
**var** logger = require("../middleware/logger");  
  
**var** app = express();  
  
app.use(logger);  
app.get("/", **function** (req, res) {  
 res.send("Welcome to my blog!");  
});  
app.use("/posts", posts);  
app.use("/users", users);  
  
  
app.listen(1339, **function** () {  
 console.log('Server running on port 1339');  
});

In the above example, if we hit */users* express will first run our request through our *logger* middleware and then – since the logger calls next() – the handler for */users* will take over.

**NB**: The order of the middleware matters. Because we placed the logger before any of the route handlers, logging will happen before every request.

## Path specific middleware

We can also mount middleware on specific routes. Let’s add logging middleware exclusively to our *posts* route.

router.use("/", **function** (req, res, next) {  
 console.log("%s - Logging on the posts page", **new** Date());  
 next();  
});  
  
  
router.get("/", **function** (req, res) {  
 res.send("Get all blog posts! - from router");  
  
});  
router.post("/", **function** (req, res) {  
 res.send("Create a new post - from router! ");  
});  
router.get("/:postId", **function** (req, res) {  
 res.send("get post with id " + req.params.postId);  
});

**NB**: Middleware added to the *app* instance is known as **Application level middleware** and middleware attached to the *Router* instance is called **Router level middleware.**

Middleware can also be mounted as a series of functions at the route handler level.

router.get("/lots-of-pre-processing", **function** (req, res, next) {  
  
 req.responseMessage = "First middleware here - ";  
 next();  
 }, **function** (req, res, next) {  
 req.responseMessage += "Second middleware here";  
 next();  
 },  
 **function** (req, res) {  
 res.send("3rd middleware returning the result - " + req.responseMessage);  
 });

# Error handling middleware

Error handling middleware in express is characterised by a four parameter signature. The first parameter is the error object and the last 3 are req, res and next, in that order. If any middleware calls next(someError) then only error handling middleware will be invoked. The rest of the middleware in the pipeline will be ignored.

module.exports = **function** (err, req, res, next) {  
 console.log("Some logging in our error handler");  
 res.status(err.status || 500).send(err.message);  
};

# Morgan logger

There exists a plethora of middleware on npm and for most common tasks you will almost certainly find one. Our primitive loggers are pretty weak and if we want to get proper, rich logging we are better off using third party request logging middleware – Enter Morgan logger. Morgan (<https://github.com/expressjs/morgan>) is a highly customizable request logging middleware.

Add a simple logger to the console for every request:

**var** morgan = require("morgan");

app.use(morgan('combined'));

And Voila!

There are many options to configure Morgan (Maybe discuss some of them). Like skip and stream.

# Body Parser

<https://github.com/expressjs/body-parser>

Express is a very minimalistic framework and uses mostly 3rd party middleware for most functionality. Express does not provide a way to access the body of a request out of the box. For this we use another middleware called body-parser.

**var** bodyParser = require('body-parser');

In order to parse a json body we use,

app.use(bodyParser.json());

body-parser will pass the body and attach the json object to our the req object We can then access this body using req.body. The parser can parse other types of bodies

* Raw – parses body as buffer
* Text – parses as string
* Urlencoded – URL encoded body

# Cors

<https://www.npmjs.com/package/cors>

Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources (e.g. fonts) on a web page to be requested from another domain outside the domain from which the resource originated (Wikipedia). Ajax request from one domain to a different domain will ordinarily be blocked. CORS allows us to calibrate our server and whitelist certain domains so that they are able to access data from our site.

**var** cors = require("cors");  
app.use(cors()); //enables cors for all domains and for all requests

app.use(cors({origin: 'http://example.com'})); //whiltelists only example.com

If we want to enable CORS on certain routes we can call cors() on the route’s handler pipeline.

router.get("/", cors() , **function** (req, res) { //A  
 res.send("Get all blog posts! - from router");  
  
});

We can also use cors with a dynamic origin. Where origin is a function that returns true or false to indicate whether or not a domain is whiltelisted.

Eg on link.