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
* res.send
* and more

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

# Routing

Routing is one of the most important functions and abstractions provided by express. Routing is matching of a route to a handler in our app.

Let us revisit our Hello World example:

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

Our express app is contained within the app variable. And any routes we want to handle must be attached to the app and assigned a handler.

Routes in express carry the following signature:

app.METHOD(ROUTE, HANDLER);

* app is an instance of our express app.
* METHOD refers to any of the HTTP request methods (get, put, post, patch etc.).
* ROUTE is a path in our app.
* HANDLER is the function executed when the requested route matches ROUTE .

Handling other HTTP Methods

app.post('/posts', **function** (req, res) {  
 res.send('Create a new post here!);  
});  
  
app.put('/', **function** (req, res) {  
 res.send('Got a Put request');  
});

Routes can get quite sophisticated and we can even match string patterns or regular expressions.

app.get('/special\*', **function** (req, res, next) {  
 next('You asked for the special path ' + req.path.replace("/", ""));  
});  
  
app.get('/[0-9]', **function** (req, res, next) {  
 next('You asked for the numbers path ' + req.path.replace("/", ""));  
});

In the above examples only paths starting with special and those containing a single numerical digit will be handled.

## Route parameters

Routes with route parameters can also be easily matched. The syntax for a route parameter is:

'/path/:routeParamName’

If a route is defined as above then any route with the signature ‘/path/ routeParamName’ will be matched and handled by the handler assigned to this route. To access any route parameter we can query the **req.params** object. So in the above example we can get the value of *routeParamName* at req.params.routeParamName

Say we wanted to get a specific post identified by postId. We can match the post by its id by defining the route:

app.get("/posts/:postId", **function** (req, res, next) {  
 res.send("get post with id " + req.params.postId);  
});

Now if we hit /posts/5 for example, we will get the response :

"get post with id 5"

Talk about dangers of putting routes with params before those without.

## 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 in one router (like a mini app), Any routes related to blog post resources will be delegated to this router.

That router might look something like this:

**var** express = require("express");  
**var** router = express.Router();

router.get("/", **function** (req, res) {  
 res.send("Get all blog posts! - from router");  
  
});  
router.post("/", **function** (req, res) {  
  
 console.log(JSON.stringify(req.body, **null**, 4));  
 res.status(201).json({postYouTriedToCreate: req.body});  
 //res.send("Create a new post - from router!");  
  
  
 //Use this as the post data  
 //{  
 // "user":"sello",  
 // "title":"a post",  
 // "content":"Its very interesting"  
 //}  
  
});

router.get("/:postId", **function** (req, res) {  
 res.send("get post with id " + req.params.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 or route handler. If one middleware responds to a request then the processing pipeline can be ended and any middleware that was yet to be invoked ignored.

Adding 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. Had we place it at the end then the middleware would get run at the end of the chain – provided a response is not sent before then.

## Route specific middleware

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

//index.js

app.use("/posts", posts);

//posts.js  
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.**

We can also attach a series of callback functions that act as middleware to our route handler. This way we isolate them to that very specific route.

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.

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

Typically Error handling middleware is the last middleware the stack because we want to catch any errors that are thrown by any other middleware.

We can have multiple error to handle different error cases. For example we could define an error handler that only handles 404 errors.

First we call next(err) from one of our functions in our route/middleware stack.

router.get("/:postId", **function** (req, res, next) {  
 **if** (req.params.postId == 100) {  
 **return** next({status: 404, message: "No post with Id " + req.params.postId});  
 }  
 res.send("get post with id " + req.params.postId);  
});

And then we use our custom error handler to return catch that error:

router.use(**function** (err, req, res, next) {  
 **if** (err.status === 404) {  
 return res.status(404).json({message: err.message });  
 }  
 next(err);  
});

In this case there is an explicit check to see if the status attached to the error object is 404. If it is then the handler will handle the error and return a response. Otherwise it will pass the error on to the next middleware

Express also has built in Error handling middleware and this is the very end of the middleware stack. If any error is not handled, it will be handled by the default express error handler.

# 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 parse the body and attach a *body*  json object to our 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

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

# 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). Because of the same-origin policy – which is a security measure implemented by browsers - , 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 resources from our site.

Access to resources on our site can be granted to other domains by specifying the

Access-Control-Allow-Origin: <origin>

Where <origin> is protocol://host:port from where the request is coming from. For example is a request comes from a script obtained at <http://ww.requsting-script.com:80>, then the origin is indeed <http://ww.requsting-script.com:80>.

If we would like grant access to this domain, then we set:

Access-Control-Allow-Origin: <http://ww.requsting-script.com:80>

We can also choose to grant access to all domains by setting:

Access-Control-Allow-Origin: \*

The npm package *cors* allows us to easily manage and configure cors for our server.

**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.

Cors has many other interesting options – check it out at <https://www.npmjs.com/package/cors#usage>

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS> - The CORS standard

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

# JSON Schema

## The need for JSON Schema

Javascript is inherently dynamically typed, this means that the type of a variable is not determined at compile time (JS does not even have a compiler). It is therefore easy to get silly bugs in your code where a certain type is passed in when u were expecting something different. To avoid such bugs, the JSON schema was developed. The JSON schema is an option to add “types” to objects so that data that is in the wrong format does not lead to unexpected behaviour in code.

As an example, consider the hypothetical *Post* object:

{  
 "\_id": 1036,  
 "postName": "The good life",  
 "datePosted": "2016-03-03:T12:23:00Z",  
 "author": "Sello Mkantjwa",  
 "views": 25,  
 "tags": [  
 "home",  
 "green"  
 ]  
}

Without types, one could provide a number the value “xyyz” for the views value, or a “Sello” For the *datePosted* field. Without JSON schema (Or any other type of validation) these values could be persisted to the database.

JSON schema defines a contract for what a valid object like. For the example above, we can force our *post* object to look the way we desire. A possible for the object might look like:

{  
 "$schema": "http://json-schema.org/draft-04/schema#",  
 "title": "Post",  
 "description": "A post - yes a post",  
 "type": "object",  
 "properties": {  
 "\_id": {  
 "type": "ObjectId",  
 "title": "id"  
 },  
 "title": {  
 "title": "Post Title",  
 "type": "string"  
 },  
 "datePosted": {  
 "title": "Date Posted",  
 "type": "string",  
 "format": "date-time"  
 },  
 "author": {  
 "title": "Author",  
 "type": "Object",  
 "properties": {  
 "firstName": {  
 "title": "First Name",  
 "type": "string"  
 },  
 "surname": {  
 "title": "Surname",  
 "type": "string"  
 }  
 },  
 "required": [  
 "firstName"  
 ]  
 },  
 "views": {  
 "title": "Views",  
 "type": "number"  
 },  
 "tags": {  
 "title": "Tags",  
 "items": {  
 "type": "string"  
 },  
 "minItems": 1,  
 "type": "array"  
 }  
 },  
 "required": [  
 "id",  
 "title",  
 "datePosted"  
 ]  
}

The labels are self-explanatory. We have a *post* object and it can have all the properties listed in the nested *properties* object. Additionally, a post MUST have the id", "title" and "datePosted” fields. This is indicated by the *required* field of the schema.

Also note that we can have nested schemas. For example the *author* property within the *post* object has its own schema.

Once we have a JSON schema defined we can validate incoming data against it. For this we use a package called jsonschema.

**var** schema = require("./post-schema.json");  
**var** post = require("./example-post.json");  
**var** Validator = require('jsonschema').Validator;  
**var** v = **new** Validator();  
console.log(v.validate(post, schema));

The validator returns a validation result, and if the data does not match the schema the errors are populated in an *errors* array*;*

{  
 instance:  
 { id: 1036,  
 title: 'The good life',  
 datePosted: '2016-03-03T12:23:00Z',  
 author: { firstName: 'Sello' },  
 views: 25,  
 tags: [ 'home', 'green' ] },  
 schema:  
 { '$schema': 'http://json-schema.org/draft-04/schema#',  
 title: 'Post',  
 description: 'A post - yes a post',  
 type: 'object',  
 properties:  
 { \_id: [Object],  
 title: [Object],  
 datePosted: [Object],  
 author: [Object],  
 views: [Object],  
 tags: [Object] },  
 required: [ 'id', 'title', 'datePosted' ] },  
 propertyPath: 'instance',  
 errors: [],  
 throwError: undefined,  
 disableFormat: **false** }

The JSON schema is actually quite extensive and one can do a lot of advanced stuff with it. Check it out at <http://json-schema.org/>.

# Templating

Templates allow us to build pages that are similar in structure but differ in content. For example, a personalised email for the different members on the system. Or a profile page that includes a user’s unique activity on the system – the layout is the same but the content (which is determined by our model) is different for different users. Templates can be produced through the use of a template engines. We will be using Handlebars.

## Handlebars

Handlebars looks largely like regular html with values for our model surrounded by double curly braces. In the case of handlebars the model is referred to as the *context.* As an example,

<p>  
 Hello <b>{{**firstName**}} {{**surname**}}</b>  
</p>

If firstName = *Sello* and surname is *Mkantjwa*, we end up with a template that looks like this:

<p>  
 Hello <b>Sello Mkantjwa</b>  
</p>

The syntax is really simple. Let us look at a more in depth example of a page that shows the details of a blog post:

Say our post looks like this :

{  
 "id": 1036,  
 "title": "The good life",  
 "datePosted": "2016-03-03T12:23:00Z",  
 "author": {  
 "firstName": "Sello",  
 "surname": "Mkantjwa"  
 },  
 "views": 25,  
 "tags": [  
 "home",  
 "green"  
 ]  
}

We can define a template *post.hbs* that looks like so:

<p>  
 id: {{**id**}}<br/>  
 title: {{**title**}}<br/>  
 datePosted: {{**datePosted**}}<br/>  
 author:<br/>  
 {{#**each author**}}  
 &nbsp;&nbsp;&nbsp;- {{**@key**}}: {{**.**}}  
 <br/>  
 {{/**each**}}  
 views: {{**views**}}<br/>  
 Tags  
<ul>  
 {{#**tags**}}  
 <li>{{**.**}}</li>  
 {{/**tags**}}  
</ul>  
</p>

To combine the model/context and the template we first compile the template. Where *postTemplate* is a valid HTML string.

**var** handlebars = require("handlebars");  
**var** fs = require("fs");  
**var** postTemplate = fs.readFileSync("./post.hbs", "utf-8");  
**var** compiled = handlebars.compile(postTemplate);

*compiled* is now a function that we can call to render our template. It takes in our context and some options.

**var** post = require("../json schema/example-post.json");  
**var** rendered = compiled(post);

The result:

<p>  
 id: 1036<br/>  
 title: The good life<br/>  
 datePosted: 2016-03-03T12:23:00Z<br/>  
 author:<br/>  
 &nbsp;&nbsp;&nbsp;- firstName: Sello  
 <br/>  
 &nbsp;&nbsp;&nbsp;- surname: Mkantjwa  
 <br/>  
 views: 25<br/>  
 Tags  
<ul>  
 <li>home</li>  
 <li>green</li>  
</ul>  
</p>

### Helpers

Note that *tags* is an array in our *post* object. In order to list all the items in the tags array we used the **each block helper.**

<ul>  
 {{#**each tags**}}  
 <li>{{**this**}}</li>  
 {{/**each**}}  
</ul>

Once we are in a **#each** block, context switches to each of the elements in the array. Therefore the above snippet lists every element in the tags array.

We also made use of the **#with** helper. This switched context to the author object allowing us to access properties of the nested author object without using the *dot* notation.

{{#**with author**}}  
 {{**firstName**}} {{**surname**}}  
 <br/>  
{{/**with**}}

The full list of helpers can be found at <http://handlebarsjs.com/builtin_helpers.html>. You can also write your own helpers but that is outside the scope of this presentation.

# References

http://expressjs.com/