Creating Express Applications

One thing on my long term to-learn cue is creating node applications using express. I've been playing with the idea of also working with GraphQL and this combination gives me the perfect excuse to start with this project.

I'll look at the basics of creating a Node application using Express, learn about Middleware and write routes for our application. We'll look at creating routes for our application API

As of right now this is the software that I want to use:

- Node
- Express and middleware
- MongoDB

Getting started

Before we get started we'll build our application using the Express Application Generator. This provides a good starting point and sensible defaults for our application. We'll configure the application with the following parameters

- Handlebars template engine
- SASS support

```
express --hbs --css sass express-demo
```

After the installation process go to the application directory and install the project's dependencies:

```
cd express-demo
npm install
```

The last step in this section is to test that everything is working properly. Run the following command to start the server:

```
npm run start
```

Point your browser to http://localhost:3000

Making things easier as a developer

The start script, as configured, requires you to stop and start the server every time you make a change. We can create a new way to start the server that will automatically restart it when we make changes.

To do this install the nodemon package to take care to monitor for changes and restart your application if needed.

Install nodemon globally:

```
npm install -g nodemon
```

Then place a comma on the previous line add the following line to the `scripts` section of your `package.json` file.

```
"devstart": "nodemon bin/www"
```

The full script section should now look like this:

```
  "scripts": {
    "start": "node ./bin/www",
    "node ./bin/www""devstart": "nodemon bin/www"
  },
```

When you start your server using the devstart script, `npm run devstart`, the server will start as normal but will also watch for changes and automatically reboot as needed.

There are situations when this will not work and you will have to manually.

Understanding The Default Installation

When running the Express generator we get a structure like the one below where `express-demo` is the root of the application and `package-lock.json` is a new feature of NPM 5.0.

```
express-demo/
├── app.js
├── bin
├── node_modules
├── package-lock.json
├── package.json
├── public
├── routes
└── views
```

`app.js` is the main entry point for the application.

`bin/` is the directory where we hold binary or executable files. In this case it holds the `www` script that will start the application.

`node_modules` is where all the plugins and their dependencies are installed.

`package.json` includes metadata about the application and its dependencies.

`package-lock.json` is a new feature of NPM 5.0 and is similar in function to `yarn.lock` generated when running the Yarn package manager.

`public` holds all our static assets such as images, scripts and style sheets.

`routes` holds one or more routes for the application. Routes tell express what to do when a request matches a specific route. We'll discuss routes in a separate chapter as they are a key part of an Express applications.

`views` hold one or more views for our resources. If a resource matches a route the view iswhat do we show the user? This will also be discussed in more detail in its own chapter.

The default application

Now that we've seen the structure of the application let's look at `app.js`, the main entry point for it.

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var routes = require('./routes/index');
var users = require('./routes/users');

var app = express();
```

As with all Node applications we have to setup the required plugins for the application using the `require` syntax. We also require the routes for the application.

As the last step in this section is to define our Express application by assigning it to a variable. We'll use the app through the reminder of the script to reference the Express application.

```
// view engine setup
app.engine('handlebars', exphbs({defaultLayout: 'main'}));
app.set('view engine', 'handlebars');


app.use('/', routes);
app.use('/users', users);
```

We next use set to (pun intended) set parameters, in this case the views and what template engine we want to use (Handlebars in this case).

```
// uncomment after placing your favicon in /public
//app.use(favicon(__dirname + '/public/favicon.ico'));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

We continue working with the `use` command to mount additional pieices of middleware for the application. We only have to do this once for the whote application so it makes sense to do it here.

The items we use are:

- A logger using Morgan
- A JSON Parser (using bodyParser)
- A URL-Encoded Parser (using bodyParser)
- A cookie parser
- The location of the application's static assets folder

The last block is our error handlers.

The first error handler that we use is a page not found error by returning a new error and a status code of 404.

For any other error we create two handlers. The first one will trigger when the application is in development mode (env === development) where it will show the full stack trace for the error.

When we're in production we don't want stack traces to appear to the users so we disable it. We may want to add a log in that case so that we know something broke.

Lastly we export the application by using `module.exports = app;`.

```
// catch 404 and forward to error handler
app.use(function(req, res, next) {
    var err = new Error('Not Found');
    err.status = 404;
```

```
    n'Not Found');

// error handlers
// development error handle will print stacktrace
if (app.get('env') === 'development') {
  app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
      message: err.message,
      error: err
    });
  });
}

'env'// production error handler; no stacktraces leaked to user
app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
        message: err.message,
        error: {}
    });
});


module.exports = app;
```

Now we are really done. Before jumping into routes and views we'll look at the startup script for the application provided by the generator.

The Startup Script

The startup script, located in `bin/www`, does a few things to get the application ready to go:

It sets the port as the value of the `port` environment variable or a default of 3000 and it starts the server listening on the port we specified. In addition, if the server is configured in debug mode, it prints a message to console indicating what port the server is listening in.

```
#!/usr/bin/env node
var debug = require('debug')('express-demo');
var app = require('../app');

app.set('port', process.env.PORT || 3000);

var server = app.listen(app.get('port'), function() {
  debug('Express server listening on port ' +
        server.address().port);
});
```

Routes

Routes tell express what to do when the application receives a path. In the table below are some examples of what would happen if I use a given HTTP verb with the exception of patch which is a more difficult version of put in my opinion.

Sample routes, verbs and expected results

Route	Verb	Result
/	get	Returns the content of the default page
/about	get	Returns content for the about page
/files/:name	get	gets a specific static file
/projects	get	Returns a list of all projects
/projects	post	Creates a new project
/projects/:project_id	get	Gets a single project
/projects/:project_id	put	Updates a single project
/projects/:project_id	delete	Deletes a project

For most of the routes outside our projects we want to render a view that has already been built with Handlebars and HTML. app.render will take care of this for us. It takes 3 parameters:

- A view to render
- An optional object with local parameters for the view to use
- A callback function

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res) {
  res.render('index', { title: 'Express' });
});
```

```
module.exp'/'s = router;
```

We can also use router.route to specify multiple HTTP verbs that apply to the same path. In the example below the `/projects/:project_id` use `get` and `put` verbs; Rather than create individual routes we create one route using `router.route` and put the different verbs under it.

Note that these routes will change when we look at how to add content from a database to the routes.

```
router.route('/projects/:project_id')
  // get the project with that id
  // accessed at GET http://localhost:8080/api/projects/:project_id
  .get((req, res) => {
      Project.findById(req.params.project_id, (err, project) => {
          if (err)
              res.send(err);
          res.json(project);
      });
  })
  // update the project with this id
  // accessed at PUT http://localhost:8080/api/projects/:project_id
  .put((req, res) => {
    project.findById(req.params.project_id, (err, project) => {
      if (err)
          res.send(err);
      project.name = req.body.name;
      project.save(err => {
          if (err)
              res.send(err);
          res.json({ message: 'project updated!' });
      });
    });
  });
'project updated!'
```

Another static route to look at is one that will send static files, PDF versions of a specific page for example, when so requested.

```
app.get('/file/:name', (req, res, next) => {
  var options = {
    root: __dirname + '/public/files',
    dotfiles: 'deny',
    headers: {
        'x-timestamp': Date.now(),
        'x-sent': true
    }
  };
  var fileName = req.params.name;
  res.sendFile(fileName, options, (err) => {
    if (err) {
      next(err);
    } else {
      console.log('Sent:', fileName);
    }
  });
});
```

We'll revisit routes when we look at using MongoDB and the Mongoose ORM to build database enabled routes.

Views

Views are how we present the content to the user. We can use full HTML files of we can use templates that combine HTML with special tags that will perform different tasks.

When working with Express' routes one of the first things I discovered is that it expects all Express related files in one directory. I've gotten used to working with a more modular structure where layouts and partials are directories containing handlebar layouts and partials respectively:

```
views
├── error.hbs
├── index.hbs
├── layouts
└── partials
```

Instead of working in the single directory structure I've installed `express-handlebars` as a replacement for the standard Handlebars view engine.

Install it as you would any other package:

```
npm install express-handlebars
```

The configuration looks different too. We specify additional configuration for our default layout, partials and layouts. I'm still experimenting with YAML Front Matter to see if I can specify page specific content

```
// view engine setup
app.engine('.hbs', exphbs({
    extname: '.hbs',
    defaultLayout: 'main',
    partialsDir: path.join(__dirname, 'views/partials'),
    layoutsDir: path.join(__dirname, 'views/layouts')
  }));
app.set('view engine', '.hbs');
```

```
app.set('views',path.join(__dirname,'views'))
```

The templates you render can be as simple as this HTML template that creates a 2 column layout with an image on the left and text on the right

```
---
title: Rivendellweb Labs
author: Carlos Araya
layout: default
---

<div class="index-wrapper">
<div class="image"></div>

<div class="text">

  <h1>{{title}}</h1>
  <h2 class="lead">These are the things I've been playing with in the last


  <article>

    <dl>
      <dt><a href="data-vis/">Data Visualization</a></dt>
      <dd>How to make data easier to understand</dd>

      <dt><a href="css-vis">CSS Property Visualization</a></dt>
      <dd>I create a visualization of CSS properties</dd>

      <dt><a href="layouts">Layout experiments</a></dt>
      <dd>Inspired by technology, magazines (Wired, Kinfolk)
      and people (Jen Simmons, Rachel Andrew)</dd>

      <dt><a href="pages-media">Paged media</a></dt>
      <dd>Using tools like PrinceXML or Antenna House
      Formatter you can create amazing layouts for
      printed material using only web technologies.</dd>
```

```
      <dt><a href="webgl">AR and VR tests</a></dt>
      <dd>Content created with WebGL, Three.js and A-Frame

      <dt><a href="essays">Essays</a></dt>
      <dd>Longer articles that are either a compilation of
      multiple blog posts or unrelated to my technical blog.</dd>
    </dl>
  </article>
</div>
</div>
```

To a template constructed from JSON data from a `projects.json` file. It uses more advanced Handlebars features and, along with the corresponding CSS and Javascript, produces a fairly rich user experience.

```
---
title: Project Archive
author: Carlos Araya
layout: default
---
<html>
  <head>
    {{#>head-content}}{{/head-content}}
  </head>
  <body>
    {{#>title-block}}{{/title-block}}

    <div id="card-container">
      {{#each projects}}
        {{> project-single}}
      {{/each}}
    </div>

    {{#>footer-block}}{{/footer-block}}
    {{#>botttom-scripts}}{{/botttom-scripts}}
  </body>
```

```
</html>
```

Database Layer: MongoDB and Mongoose

To add databases to our application we need to redefine our routes with database information. This should make the routes easier to read but it introduces additional dependencies.

```
// grab the things we need
var mongoose = require('mongoose');
var Schema = 'mongoose'chema;
var autoIncrement = require("mongodb-autoincrement");
autoIncrement.setDefaults({
    field: project_id,       "mongodb-autoincrement"// auto increment fie
    step: 1                  // auto increment step
});

// create a schema
var projectSchema = new Schema({
  name: {
    type: String,
    required: true,
    unique: true
  }
  project_id: {
    type: Number,
    required: true,
    unique, true
  },
  description: String,
  notes: String,
  stage: String,
  type: String,
  url: {
    code: String,
    other: String,
    writeup: String
```

```
  },
  created_at: {
    type: Date,
    default: Date.now
  },
  updated_at: {
    type: Date,
    default: Date.now
  }
});

schema.plugin(autoIncrement.mongoosePlugin);

// the schema is useless so far
// we need to create a model using it.model('Project', projectSchema);

// make this available 'Project'// make this available to our users in our
module.exports = User;
```

Static Long Form Content

GraphQL Instead of REST?

In a future iteration I may consider experimenting with GraphQL and look at what GraphQL is, how it works and how to create routes that work with GrapQL using GrapQL-js.

- http://danialk.github.io/blog/2014/12/05/express-4-tutorial-simple-server/