Node.js Blog Tutorial

In this tutorial we are going to build a blog powered by Node.js on the back end, Bootstrap on the front end, and MongoDB as the data store. We are going to start from scratch and build out the project one step at a time. We'll see how to include all the packages we'll need into Node for building the blog. Some of these include Express, Bcrypt, nodemon, express-edge, mongoose, body-parser, express-fileupload, and express-session. We'll start with basic routes in the index.js file, and then gradually move the application into a Model View Controller architecture. Let's get started.

Setting Up The Blog Project

In our terminal, we can create our directory and run `npm init` to get started.

```
node $mkdir nodejs-blog-tutorial
node $cd nodejs-blog-tutorial
nodejs-blog-tutorial $npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.
```

Laravel Repository Pattern
VueJS Bootstrap Pagination Component
VueJs Parent Child Communication
29 Awesome JavaScript Learning Tutorials
Introduction To AngularJS
Laravel Collections Tutorial
The 27 Most Popular File Functions in PHP

```
See npm help json for definitive documentation on these fields
and exactly what they do.

Use npm install  afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (nodejs-blog-tutorial)
version: (1.0.0)
description: Create a blog using Node.js
entry point: (index.js)
test command:
git repository:
keywords: blog
author:
license: (ISC)
About to write to C:\node\nodejs-blog-tutorial\package.json:

{
  "name": "nodejs-blog-tutorial",
  "version": "1.0.0",
  "description": "Create a blog using Node.js",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "blog"
  ],
  "author": "",
  "license": "ISC"
}


Is this OK? (yes)
```

Excellent. Now, we will install a nice Bootstrap theme from the fine folks at **Start Bootstrap**.

PHP Variables and Strings Tutorial

PHP Cookies And Sessions

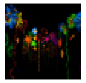
jQuery AJAX Tutorial

VueJS Dynamic Components

What Are Scalable Vector Graphics?

How To Favorite A Model

The Top 17 Most Popular PHP Array Functions


Linux Keyboard Tricks

Laravel Blade Master Template Configuration

How To Create A Custom Menu In WordPress

Upgrading VueJS

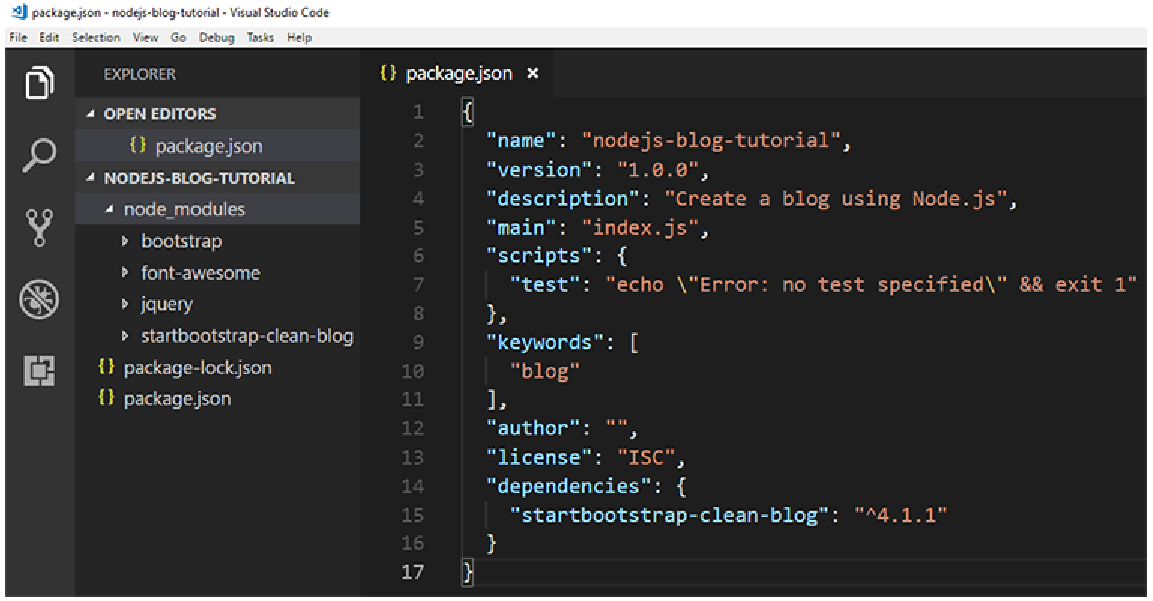
How To Highlight New Content For Returning Visitors

How To Filter Via Query Strings

Angular Parent Child Communication

```
nodejs-blog-tutorial $npm i startbootstrap-clean-blog
npm notice created a lockfile as package-lock.json. You should commit this fi
npm WARN nodejs-blog-tutorial@1.0.0 No repository field.

+ startbootstrap-clean-blog@4.1.1
added 4 packages from 10 contributors and audited 4 packages in 5.224s
found 0 vulnerabilities
```

Note that the package lives in the `node_modules` directory.



```
{
  "name": "nodejs-blog-tutorial",
  "version": "1.0.0",
  "description": "Create a blog using Node.js",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "blog"
  ],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "startbootstrap-clean-blog": "^4.1.1"
  }
}
```

We are also going to need **Express**, so let's go ahead and install it.

```
nodejs-blog-tutorial $npm i express
npm WARN bootstrap@4.1.1 requires a peer of popper.js@^1.14.3 but none is ins

+ express@4.16.3
added 50 packages from 47 contributors and audited 123 packages in 4.111s
found 0 vulnerabilities
```

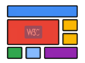
While we are at it, we can install nodemon for hot reloading of our JavaScript files. No need to constantly have to manually stop and re start your node application.

```
nodejs-blog-tutorial $npm i nodemon
```

Now let's add the entry point to our node.js blog project which is **index.js**.

Php Tutorials For Beginners

The Art of Creating a WordPress Post


CSS Grid Tutorial

JavaScript Operators and Expressions

PHP And MySQL Tutorial

Install Laravel on Windows

Create a Twitter Bootstrap Page

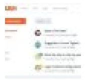
VegiThemes Twitter Bootstrap Themes

Drawing Scalable Vector Graphics With D3 JavaScript

Setting Up A Database With Seeding

Most Popular JavaScript Frameworks

Creating C# Class Properties And Tests


Flarum Forum Software

How To Add Flash Messages

```
nodejs-blog-tutorial $touch index.js
```

We will also need to have a **public** directory in the project.

```
nodejs-blog-tutorial $mkdir public
```

Let's get started by including express, setting up our public directory, and launching the server.

---

index.js

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

const app = new express();

app.use(express.static('public'));

app.listen(4000, () => {
    console.log('App listening on port 4000')
});
```

Building A Home Page

To get started building the home page, we can create a `pages` directory to store static files in. Within that directory we can start with an `index.html` file.

```
nodejs-blog-tutorial $mkdir pages
nodejs-blog-tutorial $cd pages
pages $touch index.html
```

Don't get too excited, I know this home page is epic.

---

index.html

```
<h1>Blog Home Page</h1>
```

Axios Powered VueJS Form Component

What are Functions in PHP

How To Use WordPress Excerpts

Building A Vue Front End For A Laravel API

27 Best Node.js Frameworks

Laravel belongsToMany Example

Reference Types And Value Types In C#

Introduction To The D3 JavaScript Library

How To Protect Specific Routes With Middleware

Introduction To VirtualBox and Vagrant

Twitter Bootstrap Grid Tutorial

How To Use VueJS With jQuery

How To Install WordPress on Laravel Homestead

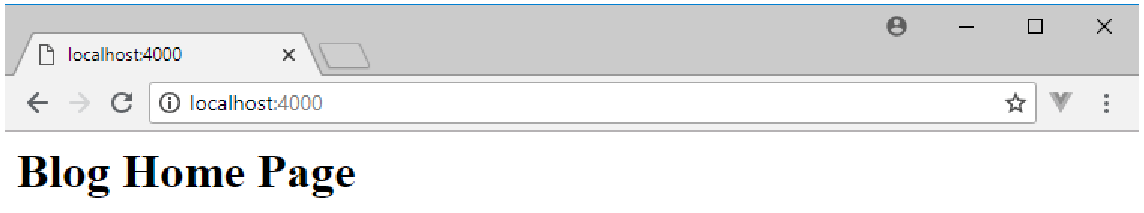
WordPress Toolbar Tutorial

Let's display that homepage in the browser now!

index.js

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

const express = require('express');

const app = new express();

app.use(express.static('public'));

app.get('/', (req, res) => {
    res.sendFile(path.resolve(__dirname, 'pages/index.html'));
});

app.listen(4000, () => {
    console.log('App listening on port 4000')
});
```

It's Working!



Blog Home Page

To make things look a lot better, we can copy the `startbootstrap-clean-blog` directory to the `theme` directory. The **command line** makes this fast and easy.

```
nodejs-blog-tutorial $cp -r node_modules/startbootstrap-clean-blog theme
```

Note the `theme` directory has everything we need to make this blog look great.

The Ultimate PHP String Functions List

Creating Monitoring and Killing Processes in Linux

Check Authorization With Policies Before Delete Function

25 Popular Applications Built With Laravel

How to Create Laravel Artisan Console Commands

Laravel Vue Component Example

How To Get Started With Eloquent In Laravel

What is a View Composer in Laravel?

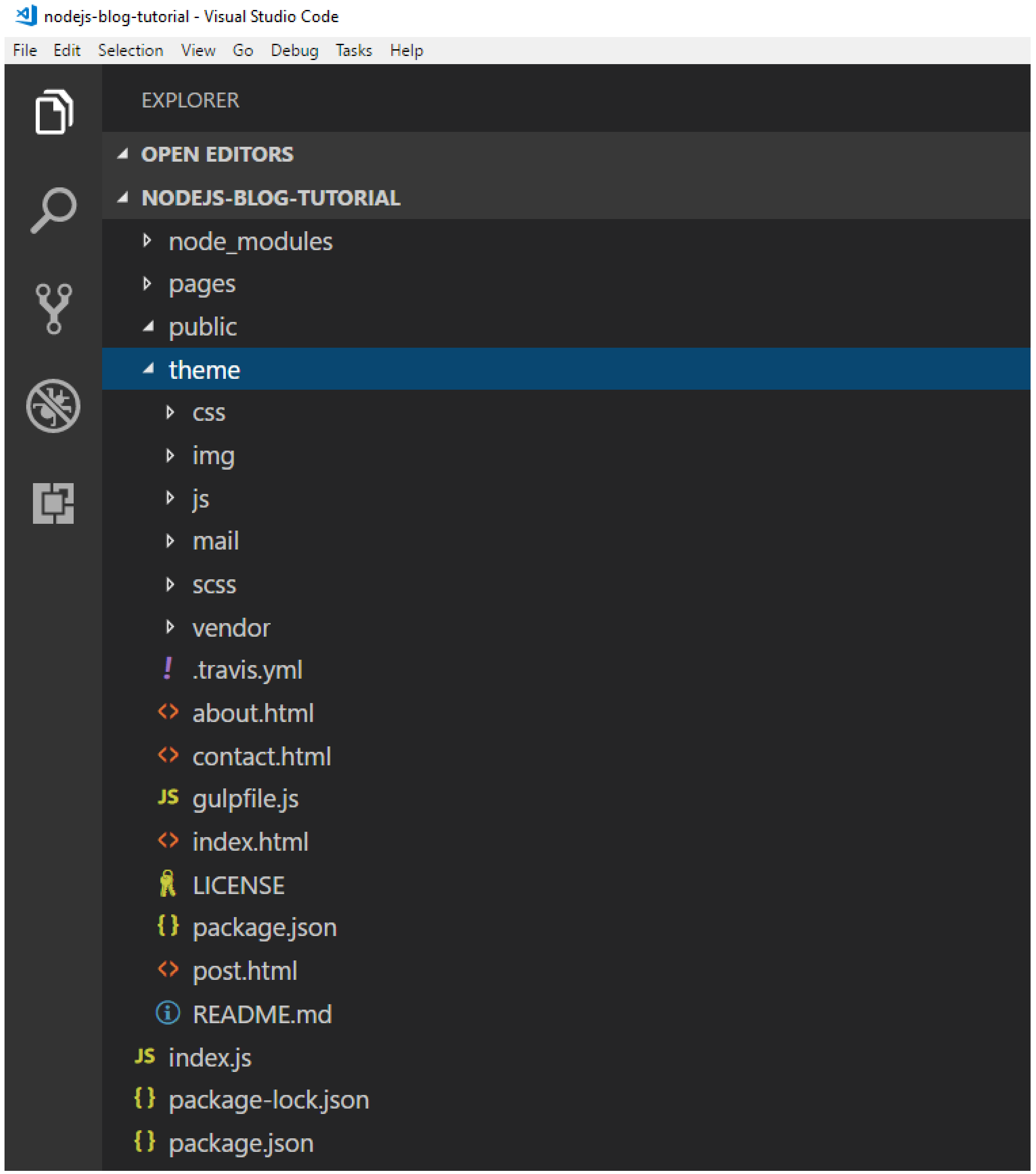
What Is Middleware In Laravel 5?

JavaScript Module Pattern

Angular Root Component Tutorial

Feature Test vs Unit Test And Adding Replies To Threads

How To Use The Laravel Query Builder


nodejs-blog-tutorial - Visual Studio Code

File Edit Selection View Go Debug Tasks Help

```
EXPLORER
◢ OPEN EDITORS
◢ NODEJS-BLOG-TUTORIAL
  ▷ node_modules
  ▷ pages
  ◢ public
  ◢ theme
    ▷ css
    ▷ img
    ▷ js
    ▷ mail
    ▷ scss
    ▷ vendor
    .travis.yml
    about.html
    contact.html
    gulpfile.js
    index.html
    LICENSE
    package.json
    post.html
    README.md
  index.js
  package-lock.json
  package.json
```

Since we have set the `public` directory as where we will serve assets, we need to copy the `vendor` `css` `img` and `js` directories over. We also copy the theme `index.html` over to the `pages` directory.

Configure Form and Button For A Post Request

An Example of JavaScript Closure Using The Date Object

VueJS Subnet Calculator

Even More Awesome WordPress Fundamentals

Traverse the DOM With jQuery

Rendering A List With React

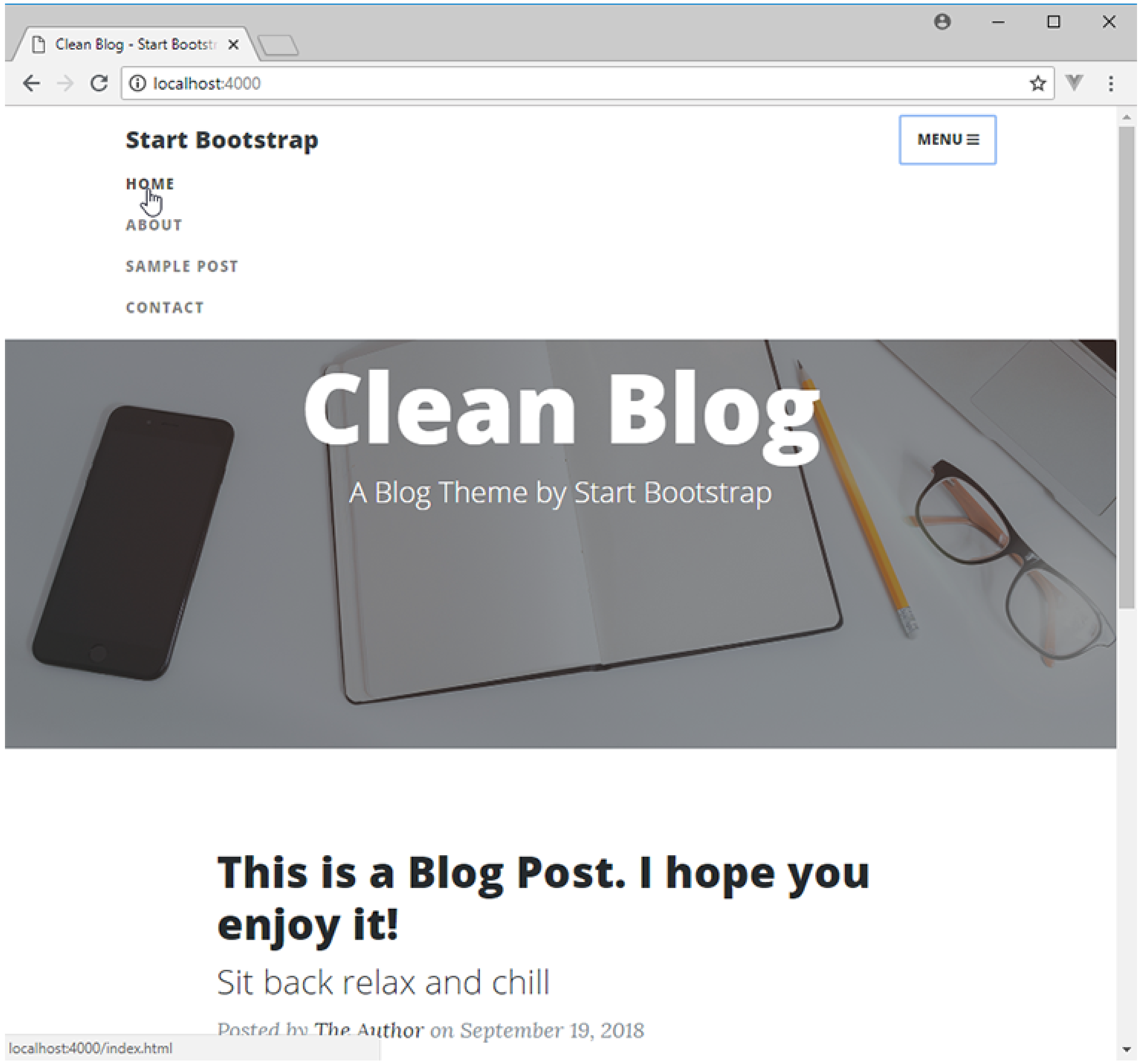
How To Toggle Exception Handling

Process Returned MySQL Query Results In PHP

PHP Include Vs Require

```
nodejs-blog-tutorial $cp -r theme/vendor public/vendor
nodejs-blog-tutorial $cp -r theme/css public/css
nodejs-blog-tutorial $cp -r theme/img public/img
nodejs-blog-tutorial $cp -r theme/js public/js
nodejs-blog-tutorial $cp -r theme/index.html pages/index.html
nodejs-blog-tutorial $node index.js
App listening on port 4000
```

Once we launch the server, and load the home page – Wow! It looks pretty good!


Clean Blog - Start Bootstr
localhost:4000
Start Bootstrap
MENU ≡
HOME
ABOUT
SAMPLE POST
CONTACT
Clean Blog
A Blog Theme by Start Bootstrap
This is a Blog Post. I hope you enjoy it!
Sit back relax and chill
Posted by The Author on September 19, 2018
localhost:4000/index.html

Adding About, Contact, and Post Pages

First we can copy **about.html** to our `pages` directory.

```
nodejs-blog-tutorial $cp -r theme/about.html pages/about.html
```

Now we can add a route handler to serve the requests to `/about`.

---

index.js

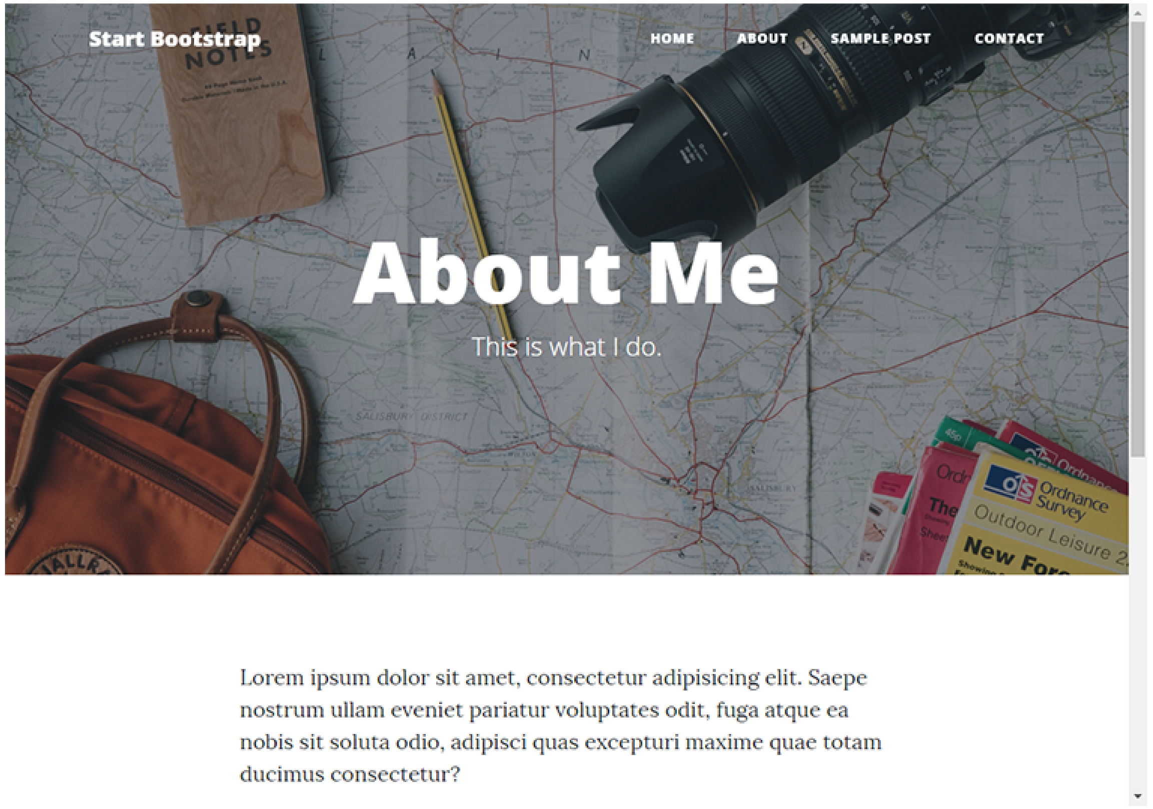
```javascript
const path = require('path');

const express = require('express');

const app = new express();

app.use(express.static('public'));

app.get('/', (req, res) => {
    res.sendFile(path.resolve(__dirname, 'pages/index.html'));
});

app.get('/about', (req, res) => {
    res.sendFile(path.resolve(__dirname, 'pages/about.html'));
});

app.listen(4000, () => {
    console.log('App listening on port 4000')
});
```

Go ahead and launch **nodemon** so the server will restart any time we change our files.

```
nodejs-blog-tutorial $nodemon index.js
[nodemon] 1.17.5[nodemon] to restart at any time, enter rs[nodemon] watching:
```

```
[nodemon] starting node index.js
App listening on port 4000
```

There we go! A nice about page.



Lorem ipsum dolor sit amet, consectetur adipisicing elit. Saepe nostrum ullam eveniet pariatur voluptates odit, fuga atque ea nobis sit soluta odio, adipisci quas excepturi maxime quae totam ducimus consectetur?

We can do the same for contact and post pages.

```
nodejs-blog-tutorial $cp -r theme/contact.html pages/contact.html
nodejs-blog-tutorial $cp -r theme/post.html pages/post.html
```

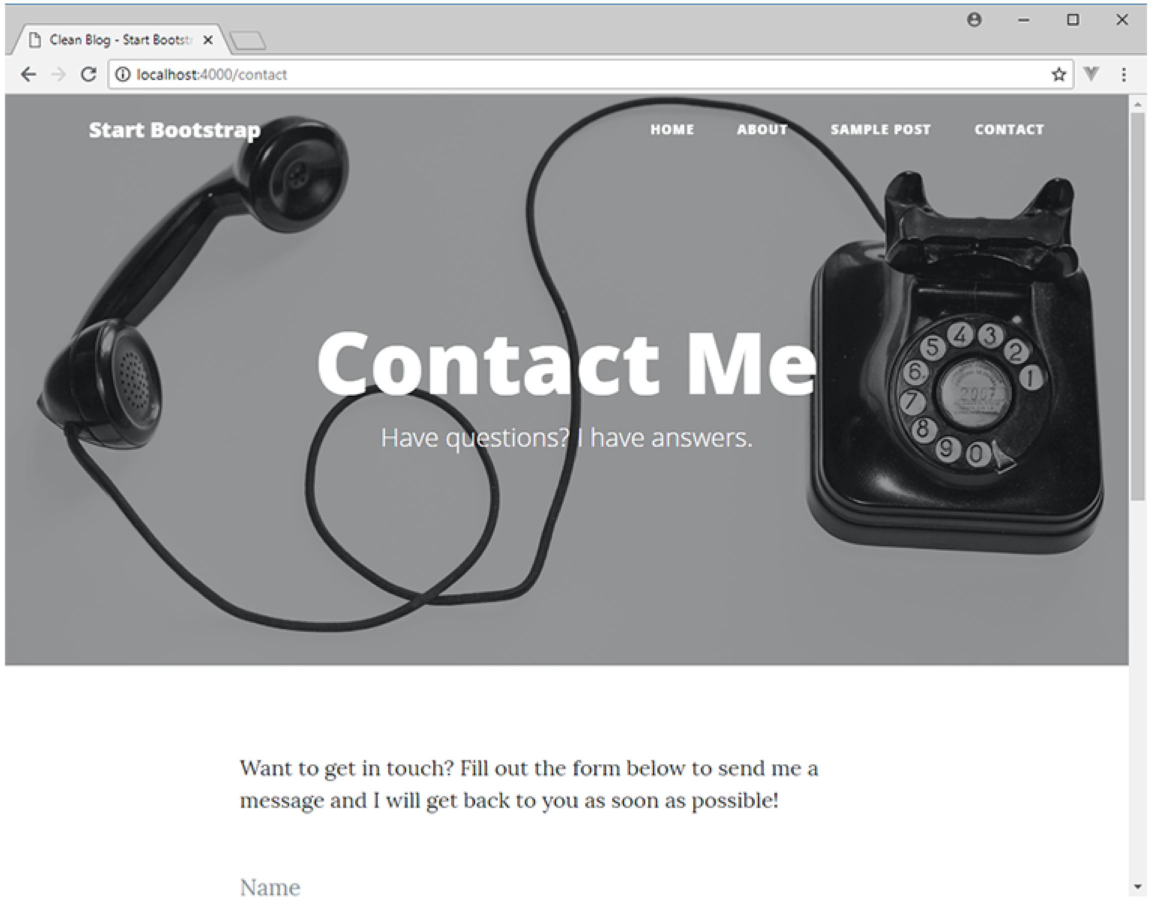
Add the new route handlers for `/contact` and `/post` like so.

---

index.js

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

const express = require('express');

const app = new express();

app.use(express.static('public'));

app.get('/', (req, res) => {
    res.sendFile(path.resolve(__dirname, 'pages/index.html'));
});

app.get('/about', (req, res) => {
    res.sendFile(path.resolve(__dirname, 'pages/about.html'));
});

app.get('/contact', (req, res) => {
    res.sendFile(path.resolve(__dirname, 'pages/contact.html'));
});

app.get('/post', (req, res) => {
    res.sendFile(path.resolve(__dirname, 'pages/post.html'));
});

app.listen(4000, () => {
    console.log('App listening on port 4000')
});
```

Bingo!



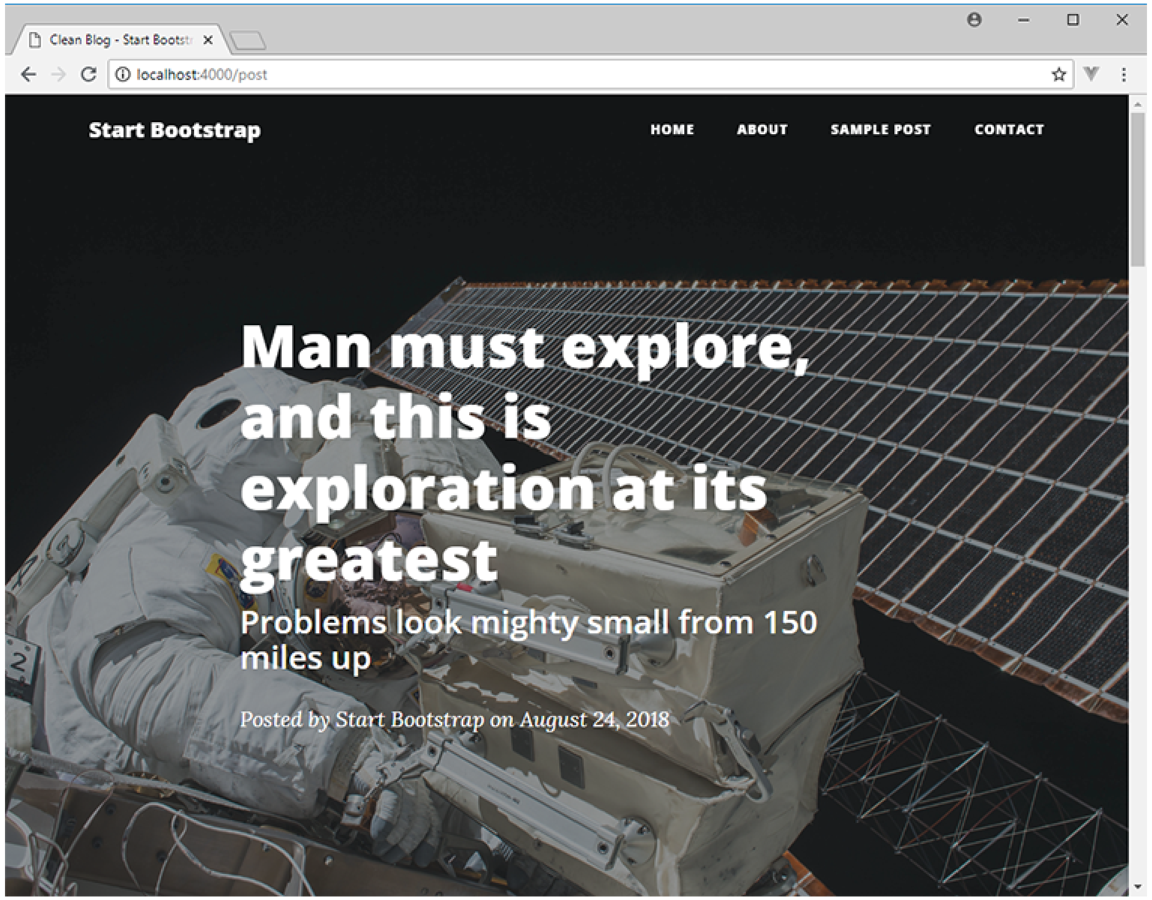
Contact Me

Have questions? I have answers.

Want to get in touch? Fill out the form below to send me a message and I will get back to you as soon as possible!

Name

Excellent!



Man must explore, and this is exploration at its greatest

Problems look mighty small from 150 miles up

Posted by Start Bootstrap on August 24, 2018

Edge Template Engine With Express

The pages so far are static. We want more of a dynamic situation. For this we can use the **Edge Template Engine** for use with Express.

```
nodejs-blog-tutorial $npm install express-edge --save
```

Here is how we can specify we are now using the Edge template engine in **index.js**.

index.js

```
const path = require('path');
const expressEdge = require('express-edge');
const express = require('express');

const app = new express();

app.use(express.static('public'));
app.use(expressEdge);
app.set('views', __dirname + '/views');
```

Now let’s create the `views` directory which will hold our `.edge` files.

```
nodejs-blog-tutorial $mkdir views
nodejs-blog-tutorial $touch views/index.edge
```

In **index.js**, go ahead and remove this code.

```
app.get('/', (req, res) => {
    res.sendFile(path.resolve(__dirname, 'pages/index.html'));
});
```

Once the snippet above is removed, you can put this code in it’s place. This tells our application that we are now going to render an edge template named `index` from the `views` folder instead of our original static file.

```
app.get('/', (req, res) => {
    res.render('index');
});
```

Layouts With Edge

Just like all the other popular templating engines, we can set up *layout* files. Here we add a `layouts` directory within `views`, and add an `app.edge` file.

```
nodejs-blog-tutorial $mkdir views/layouts
nodejs-blog-tutorial $touch views/layouts/app.edge
```

In `app.edge`, we can add the common markup all pages will share. The “middle” of the page is removed and replaced with `@!section('content')`.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrinl
  <meta name="description" content="">
  <meta name="author" content="">

  <title>Clean Blog - Start Bootstrap Theme</title>

  <!-- Bootstrap core CSS -->
  <link href="vendor/bootstrap/css/bootstrap.min.css" rel="stylesheet">

  <!-- Custom fonts for this template -->
  <link href="vendor/font-awesome/css/font-awesome.min.css" rel="styleshee
  <link href='https://fonts.googleapis.com/css?family=Lora:400,700,400italic,7
  <link href='https://fonts.googleapis.com/css?family=Open+Sans:300italic,40
    rel='stylesheet' type='text/css'>

  <!-- Custom styles for this template -->
  <link href="css/clean-blog.min.css" rel="stylesheet">
</head>

<body>
  <!-- Navigation -->
  <nav class="navbar navbar-expand-lg navbar-light fixed-top" id="mainNav">
    <div class="container">
      <a class="navbar-brand" href="index.html">Start Bootstrap</a>
      <button class="navbar-toggler navbar-toggler-right" type="button" data-t
```

```
        aria-controls="navbarResponsive" aria-expanded="false" aria-label="Togg
        Menu
        <i class="fa fa-bars"></i>
      </button>
      <div class="collapse navbar-collapse" id="navbarResponsive">
        <ul class="navbar-nav ml-auto">
          <li class="nav-item">
            <a class="nav-link" href="index.html">Home</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="about.html">About</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="post.html">Sample Post</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="contact.html">Contact</a>
          </li>
        </ul>
      </div>
    </div>
  </nav>

  @!section('content')

  <!-- Footer -->
  <footer>
    <div class="container">
      <div class="row">
        <div class="col-lg-8 col-md-10 mx-auto">
          <ul class="list-inline text-center">
            <li class="list-inline-item">
              <a href="#">
```

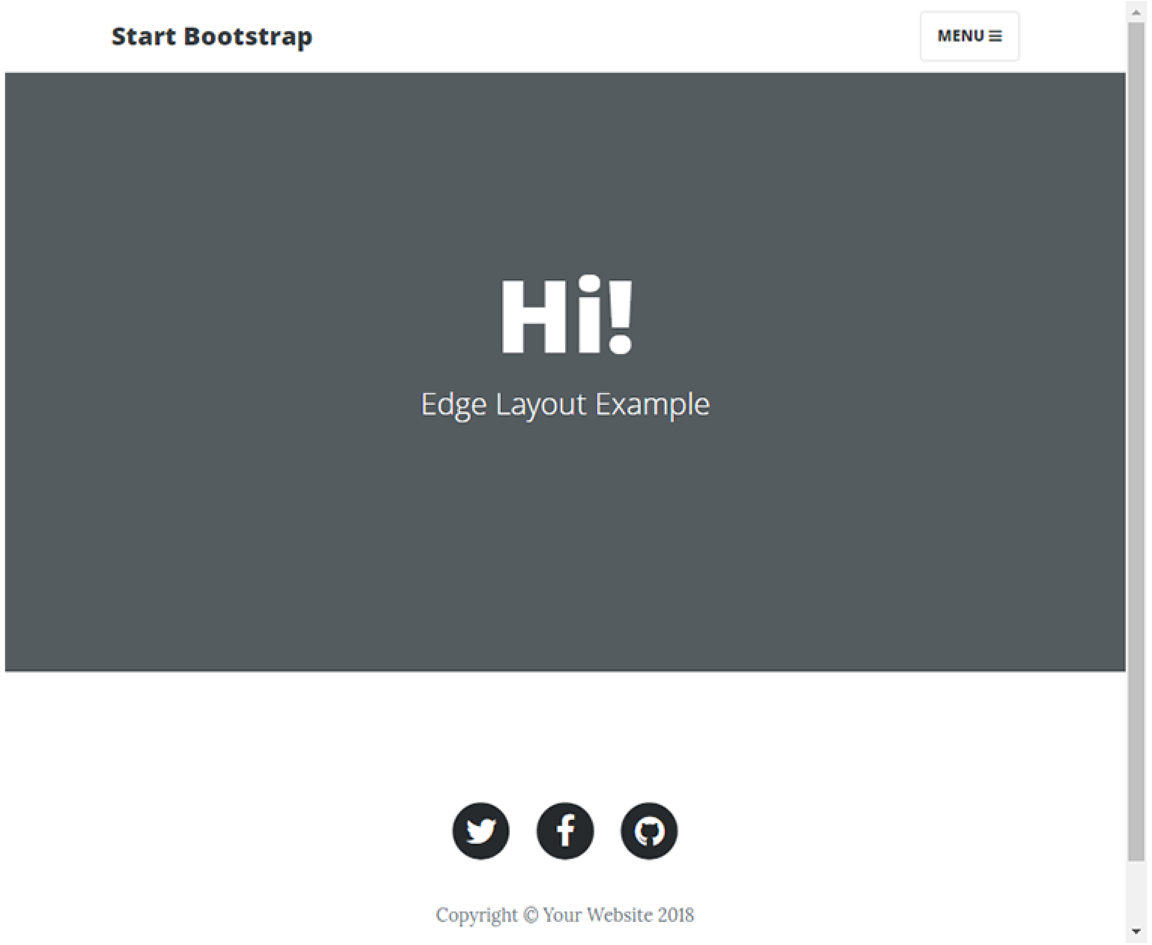
```html
            <span class="fa-stack fa-lg">
              <i class="fa fa-circle fa-stack-2x"></i>
              <i class="fa fa-twitter fa-stack-1x fa-inverse"></i>
            </span>
          </a>
        </li>
        <li class="list-inline-item">
          <a href="#">
            <span class="fa-stack fa-lg">
              <i class="fa fa-circle fa-stack-2x"></i>
              <i class="fa fa-facebook fa-stack-1x fa-inverse"></i>
            </span>
          </a>
        </li>
        <li class="list-inline-item">
          <a href="#">
            <span class="fa-stack fa-lg">
              <i class="fa fa-circle fa-stack-2x"></i>
              <i class="fa fa-github fa-stack-1x fa-inverse"></i>
            </span>
          </a>
        </li>
      </ul>
      <p class="copyright text-muted">Copyright &copy; Your Website 2018<
    </div>
  </div>
</div>
</footer>

<!-- Bootstrap core JavaScript -->
<script src="vendor/jquery/jquery.min.js"></script>
<script src="vendor/bootstrap/js/bootstrap.bundle.min.js"></script>

```

```
  <!-- Custom scripts for this template -->
  <script src="js/clean-blog.min.js"></script>
</body>

</html>
```

Now we can really simplify any other files that extend the **layout.app** file. For example, now we can add this markup to **index.edge**.

```
@layout('layouts.app')

@section('content')

 <!-- Page Header -->
 <header class="masthead">
   <div class="overlay"></div>
   <div class="container">
     <div class="row">
       <div class="col-lg-8 col-md-10 mx-auto">
         <div class="site-heading">
           <h1>Hi!</h1>
           <span class="subheading">Edge Layout Example</span>
         </div>
       </div>
     </div>
   </div>
 </header>

@endsection
```

Here is the result in the browser. Looks pretty good!



Dynamic Data With MongoDB

Our Node js blog system is going to use MongoDB to store blog posts. The goal is to be able to store new blog posts into MongoDB, and also to retrieve blog posts from MongoDB and send that data to our .edge template files which will display the dynamic data. We can get started by installing Mongoose.

```
nodejs-blog-tutorial $npm i mongoose --save
```

Now we can require mongoose in our **index.js** file and connect to the database like we see here. Don't forget the **useNewUrlParser** property.

```
const path = require('path');
const expressEdge = require('express-edge');
const express = require('express');
const mongoose = require('mongoose');

const app = new express();

mongoose.connect('mongodb://localhost:27017/node-blog', { useNewUrlParser:
    .then(() => 'You are now connected to Mongo!')
    .catch(err => console.error('Something went wrong', err))
```

Mongoose Models

We can create a dedicated directory to store our Mongoose Models and we'll start with a Post.js file.

```
nodejs-blog-tutorial $mkdir database
nodejs-blog-tutorial $cd database
database $mkdir models
database $touch models/Post.js
```

Post.js will contain this code.

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

const PostSchema = new mongoose.Schema({
    title: String,
    description: String,
    content: String
});

const Post = mongoose.model('Post', PostSchema);

module.exports = Post;
```

New Blog Post Form

Let's add a page to the blog where a form is presented to the user so they can submit a new blog post.

```
nodejs-blog-tutorial $touch views/create.edge
```

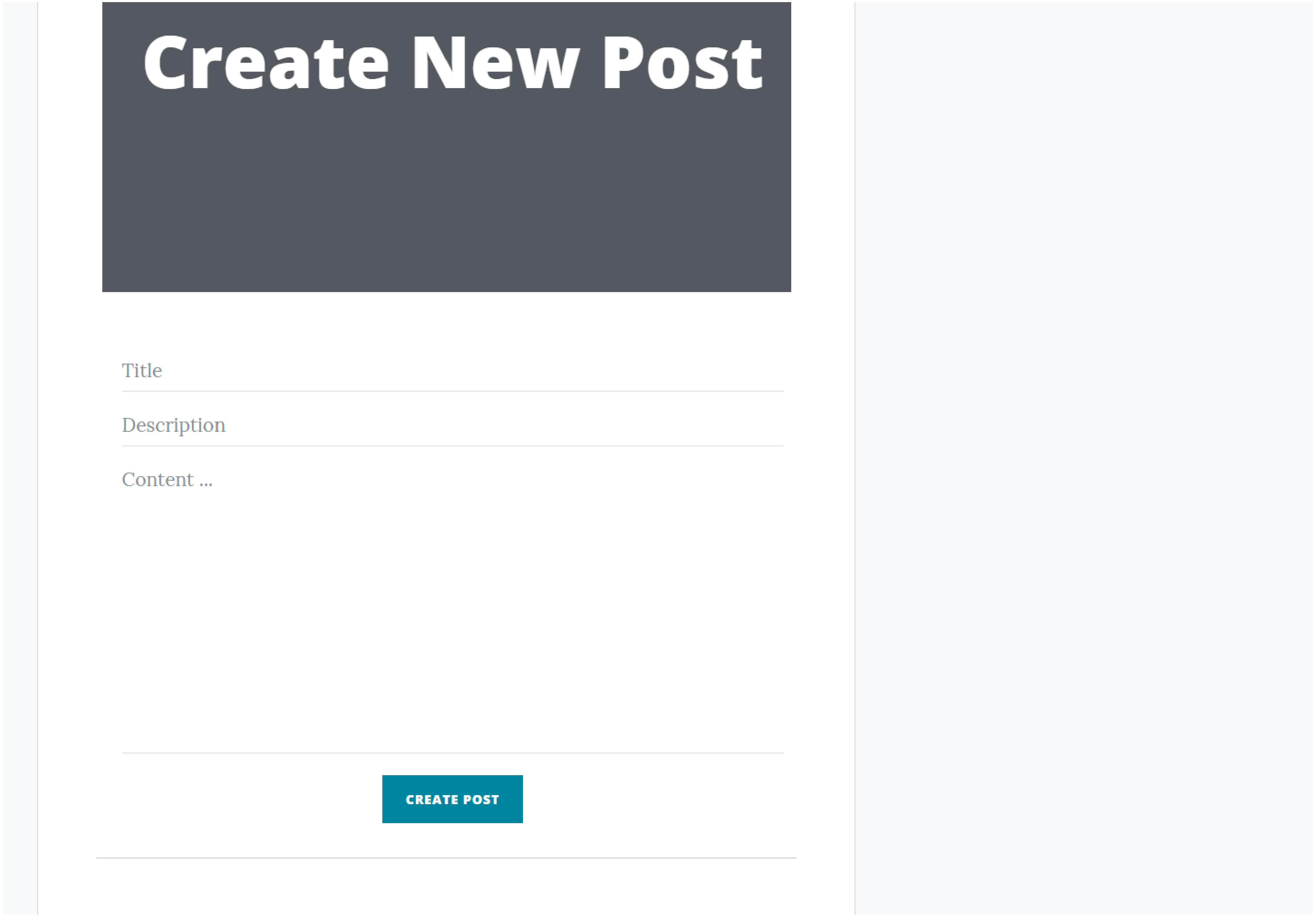
The following markup can be put in the **create.edge** file.

```
@layout('layouts.app')

@section('content')
<!-- Page Header -->
<header class="masthead">
  <div class="overlay"></div>
  <div class="container">
    <div class="row">
      <div class="col-lg-8 col-md-10 mx-auto">
        <div class="page-heading">
          <h1>Create New Post</h1>
        </div>
      </div>
    </div>
  </div>
</header>

<div class="container">
  <div class="row">
    <div class="col-md-8 offset-md-2">
      <form action="/posts/store" method="POST">
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Title</label>
            <input type="text" name="title" placeholder="Title" class="form-contro
          </div>
        </div>
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Description</label>
            <input type="text" name="description" placeholder="Description" class
          </div>
        </div>
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Content</label>
            <textarea name="content" placeholder="Content ..." cols="30" rows="
          </div>
        </div>
        <div class="form-group my-4 text-center">
          <button class="btn btn-primary">Create Post</button>
        </div>
      </form>
    </div>
  </div>
</div>
@endsection
```

Now we need a route in **index.js** to present the form.

```
app.get('/posts/new', (req, res) => {
    res.render('create')
});
```

Now we can visit http://localhost:4000/posts/new and we have a form for creating a new blog post.


Create New Post
Title
Description
Content ...
CREATE POST

Setting Up POST Requests in Express

Our form for a new blog post is going to send a POST request and we need to set up the code to handle that. We will also need the **body-parser** package to read the data that gets sent in the POST request. Let's add **body-parser** first.

```
nodejs-blog-tutorial $npm install body-parser
```

Now we need to use body-parser in **index.js**. The relevant snippets are highlighted.

```
const path = require('path');
const expressEdge = require('express-edge');
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');

const app = new express();

mongoose.connect('mongodb://localhost:27017/node-blog', {
        useNewUrlParser: true
    })
    .then(() => 'You are now connected to Mongo!')
    .catch(err => console.error('Something went wrong', err))

app.use(express.static('public'));
app.use(expressEdge);
app.set('views', __dirname + '/views');
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({
    extended: true
}));

app.get('/', (req, res) => {
    res.render('index');
});

app.get('/posts/new', (req, res) => {
   res.render('create')
});

app.post('/posts/store', (req, res) => {
   console.log(req.body)
   res.redirect('/')
});
```

Now if you enter some data into the form, you'll notice we can inspect it in the console if we submit the form. Nice!

```
[nodemon] starting node index.js
App listening on port 4000
{ title: 'This is the title field',
  description: 'Here is the description field',
  content: 'The content for the post is here' }
```

Store New Blog Post To MongoDB

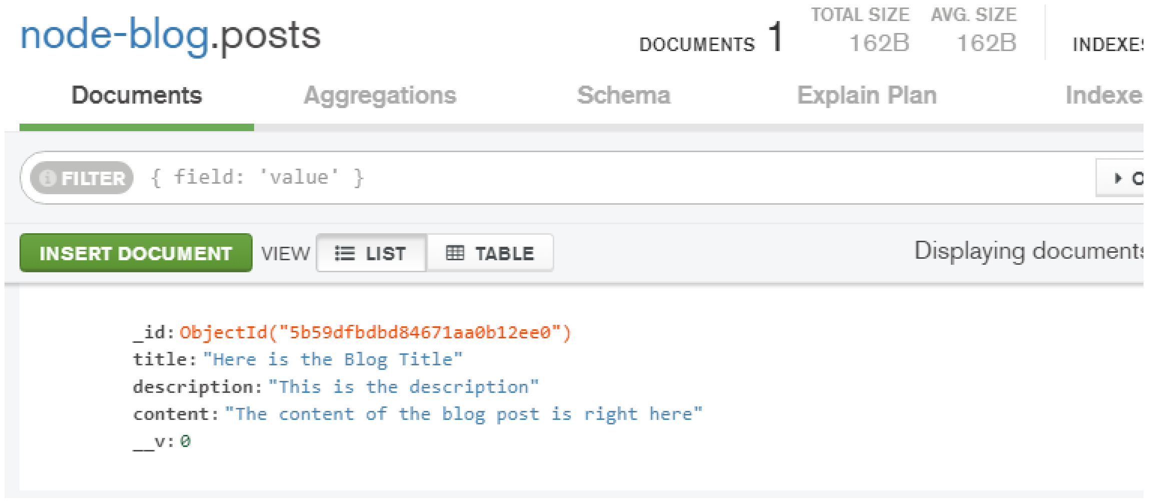
Now that we are getting the data from the form we have to do something with it. We want to store it in the database. We will make use of the **Post.js** model we had created earlier.

```
const Post = require('./database/models/Post');
```

Now we can update the `/posts/store` like so.

```
app.post('/posts/store', (req, res) => {
   Post.create(req.body, (error, post) => {
      res.redirect('/')
   })
});
```

That should be enough to store a new post into the database. Go ahead and fill out the new blog post form, then click Submit. Then we can use **Compass** to check it out and our data is there.



```
{ field: 'value' }
```

```
_id: ObjectId("5b59dfbdbd84671aa0b12ee0")
title: "Here is the Blog Title"
description: "This is the description"
content: "The content of the blog post is right here"
__v: 0
```

Displaying Blog Posts From The Database

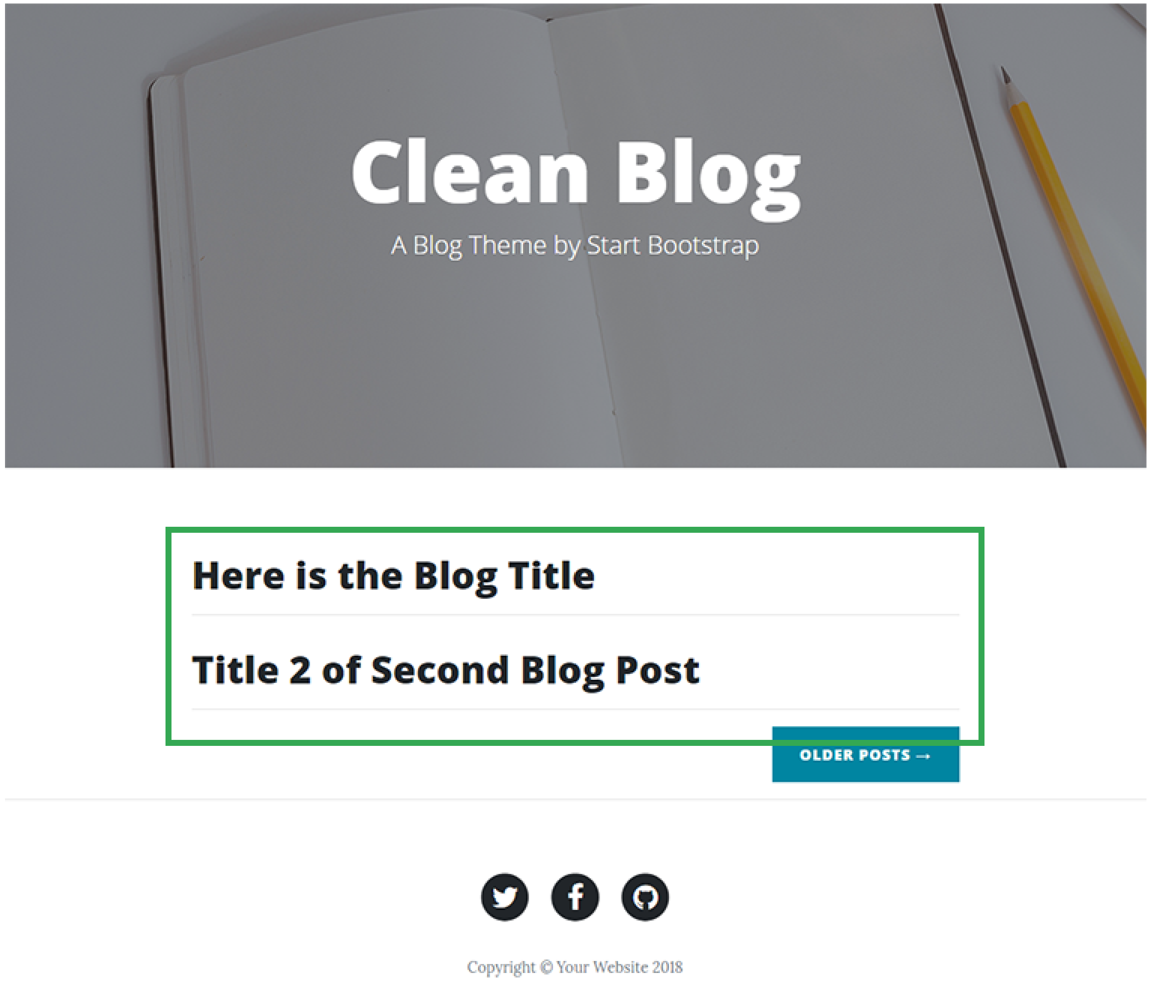
Now we can remove the “static” pages that represented blog posts from before and use dynamic data from MongoDB to display blog posts. We can modify the `/` route to now fetch data from MongoDB. Then, we will pass that data to the index.edge file.

```
app.get('/', async (req, res) => {
   const posts = await Post.find({})
   res.render('index', {
      posts
   })
});
```

The **index.edge** file must now be fixed to account for dynamic data instead of the static markup we had before. We use the `@each` directive in Edge to do this. This allows us to loop over multiple blog posts stored in the database and display them.

```
@layout('layouts.app')

@section('content')
<!-- Page Header -->
<header class="masthead" style="background-image: url('img/home-bg.jpg')">
  <div class="overlay"></div>
  <div class="container">
    <div class="row">
      <div class="col-lg-8 col-md-10 mx-auto">
        <div class="site-heading">
          <h1>Clean Blog</h1>
          <span class="subheading">A Blog Theme by Start Bootstrap</span>
        </div>
      </div>
    </div>
  </div>
</header>

<!-- Main Content -->
<div class="container">
  <div class="row">
    <div class="col-lg-8 col-md-10 mx-auto">
      @each(post in posts)
        <div class="post-preview">
          <a href="post.html">
            <h2 class="post-title">
              {{ post.title }}
            </h2>
          </a>
        </div>
        <hr>
      @endeach
      <!-- Pager -->
      <div class="clearfix">
        <a class="btn btn-primary float-right" href="#">Older Posts &rarr;</a>
      </div>
    </div>
  </div>
</div>

<hr>
@endsection
```

We added a second blog post to the database, now let’s view the home page. Nice! We can see the title of two blog posts now thanks to our `@each` loop.


Clean Blog
A Blog Theme by Start Bootstrap
Here is the Blog Title
Title 2 of Second Blog Post
OLDER POSTS →
Copyright © Your Website 2018

Displaying a Single Blog Post

Now that we have the home page sorted which can display all posts, let's set up the ability to click on a single post and view it's contents. First off we need to modify the `/post` route to the following.

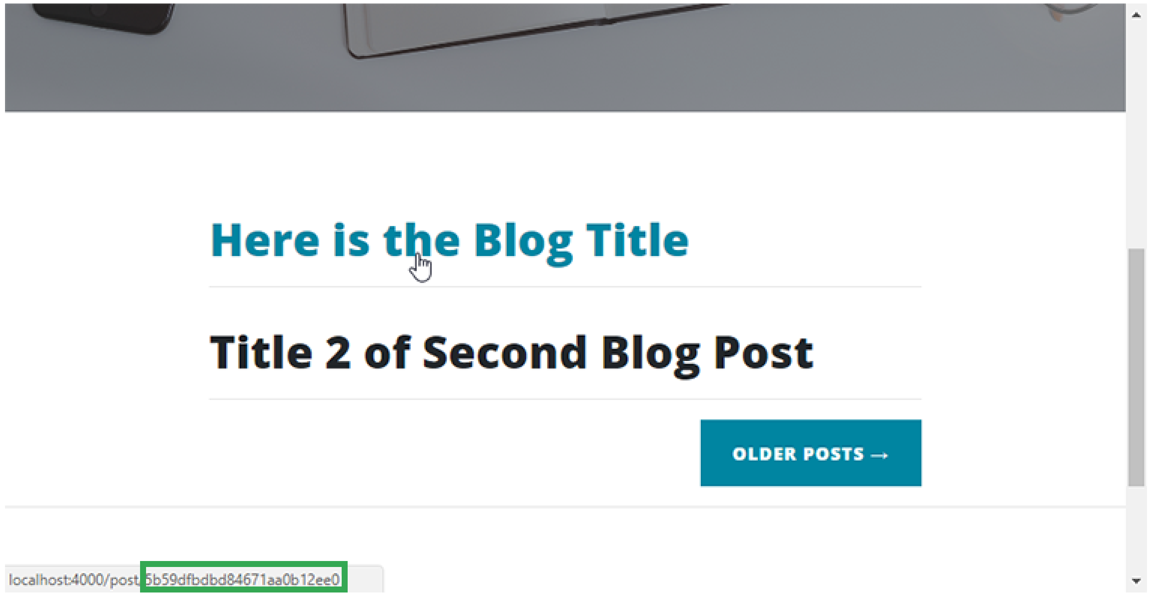
```
app.get('/post/:id', async (req, res) => {
  const post = await Post.findById(req.params.id)
  res.render('post', {
    post
  })
});
```

Now we must make the links clickable in the **index.edge** file so that we can click a title, and be brought to the single post view. Note the addition of the anchor tag which links to the unique post id of each blog post.

```
@layout('layouts.app')

@section('content')
<!-- Page Header -->
<header class="masthead" style="background-image: url('img/home-bg.jpg')">
  <div class="overlay"></div>
  <div class="container">
    <div class="row">
      <div class="col-lg-8 col-md-10 mx-auto">
        <div class="site-heading">
          <h1>Clean Blog</h1>
          <span class="subheading">A Blog Theme by Start Bootstrap</span>
        </div>
      </div>
    </div>
  </div>
</header>

<!-- Main Content -->
<div class="container">
  <div class="row">
    <div class="col-lg-8 col-md-10 mx-auto">
      @each(post in posts)
        <div class="post-preview">
          <a href="/post/{{ post._id }}">
            <h2 class="post-title">
```

```
       {{ post.title }}
      </h2>
     </a>
    </div>
    <hr>
   @endeach
   <!-- Pager -->
   <div class="clearfix">
    <a class="btn btn-primary float-right" href="#">Older Posts &rarr;</a>
   </div>
  </div>
 </div>
</div>

<hr>
@endsection
```

Now when we hover over the link on the main page, notice how the browser shows us that it is linking to the objectid of that blog post.



Here is the Blog Title

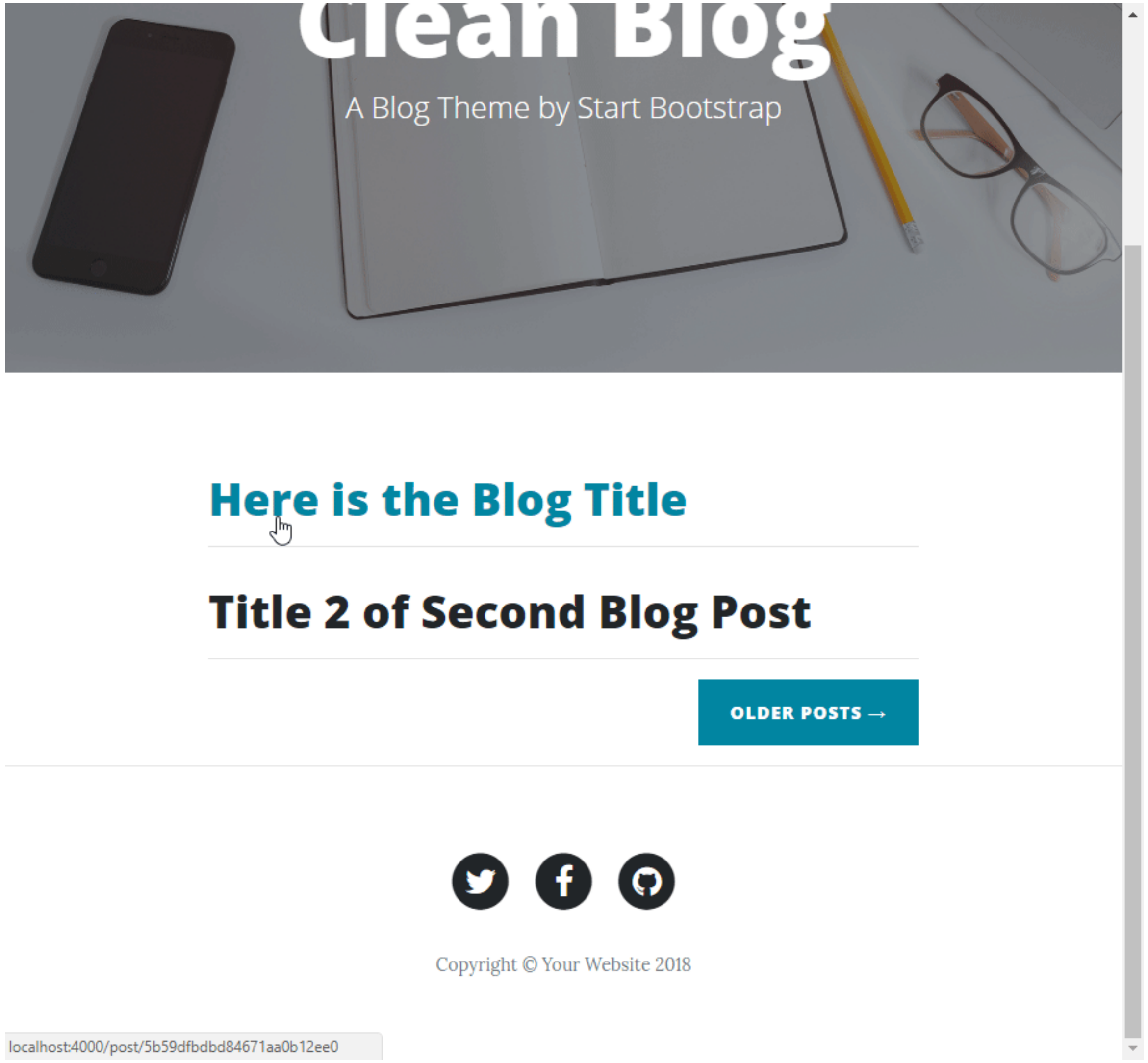
Title 2 of Second Blog Post

localhost:4000/post/5b59dfbdbd84671aa0b12ee0

Now let’s update the **post.edge** file to display the blog post dynamically.

```
@layout('layouts.app')

@section('content')
  <!-- Page Header -->
  <header class="masthead">
    <div class="overlay"></div>
    <div class="container">
      <div class="row">
        <div class="col-lg-8 col-md-10 mx-auto">
          <div class="post-heading">
            <h1>{{ post.title }}</h1>
          </div>
        </div>
      </div>
    </div>
  </header>

  <!-- Post Content -->
  <article>
    <div class="container">
      <div class="row">
        <div class="col-lg-8 col-md-10 mx-auto">
          {{ post.content }}
        </div>
      </div>
    </div>
  </article>

  <hr>
@endsection
```

Now we can easily click on any blog post title, and we are taken to that particular post to view it.


Clean Blog
A Blog Theme by Start Bootstrap
Here is the Blog Title
Title 2 of Second Blog Post
OLDER POSTS →
Copyright © Your Website 2018
localhost:4000/post/5b59dfbdbd84671aa0b12ee0

Adding User Name and Created At To Blog Posts

Blog posts should have a user name associated with them, as well as a date of creating. We can update the **Post.js** model to reflect this.

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

const PostSchema = new mongoose.Schema({
    title: String,
    description: String,
    content: String,
    username: String,
    createdAt: {
        type: Date,
        default: new Date()
    }
});

const Post = mongoose.model('Post', PostSchema);

module.exports = Post;
```

We'll need to adjust our **create.edge** file to accommodate for the user name.

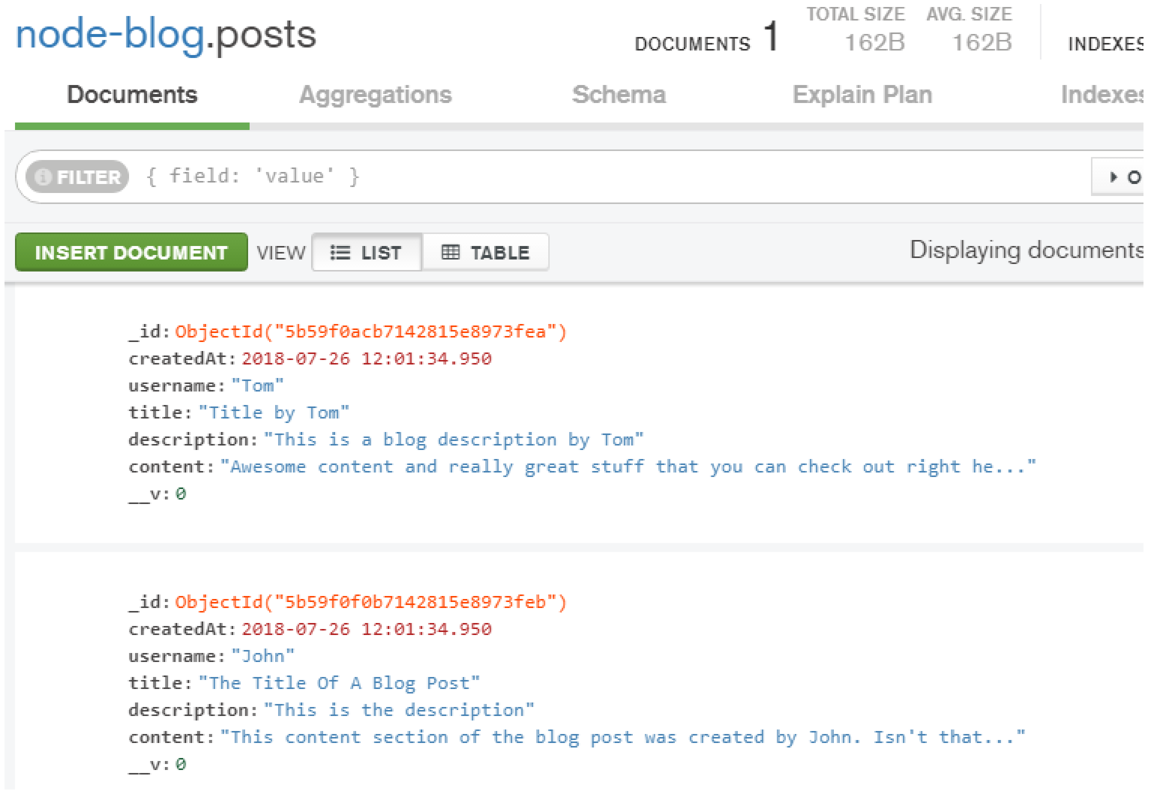
```
@layout('layouts.app')

@section('content')
<!-- Page Header -->
<header class="masthead">
  <div class="overlay"></div>
  <div class="container">
    <div class="row">
      <div class="col-lg-8 col-md-10 mx-auto">
        <div class="page-heading">
          <h1>Create New Post</h1>
        </div>
      </div>
    </div>
  </div>
</header>
```

```html

<div class="container">
  <div class="row">
    <div class="col-md-8 offset-md-2">
      <form action="/posts/store" method="POST">
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Username</label>
            <input type="text" name="username" placeholder="Username" class='
          </div>
        </div>
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Title</label>
            <input type="text" name="title" placeholder="Title" class="form-contro
          </div>
        </div>
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Description</label>
            <input type="text" name="description" placeholder="Description" class
          </div>
        </div>
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Content</label>
            <textarea name="content" placeholder="Content ..." cols="30" rows="
          </div>
        </div>
        <div class="form-group my-4 text-center">
          <button class="btn btn-primary">Create Post</button>
        </div>
      </form>
    </div>
  </div>
</div>
@endsection
```

Before we add any new blog posts, go ahead and drop the database from Mongo using the command line shell.

```
> mongo
> use node-blog
switched to db node-blog
> db.dropDatabase()
{ "dropped" : "node-blog", "ok" : 1 }
>
```

Now we can add some new blog posts to a fresh database by visiting http://localhost:4000/posts/new and if we check in Compass we can see the new properties in our database. Notice the `createdAt` and `username` properties.

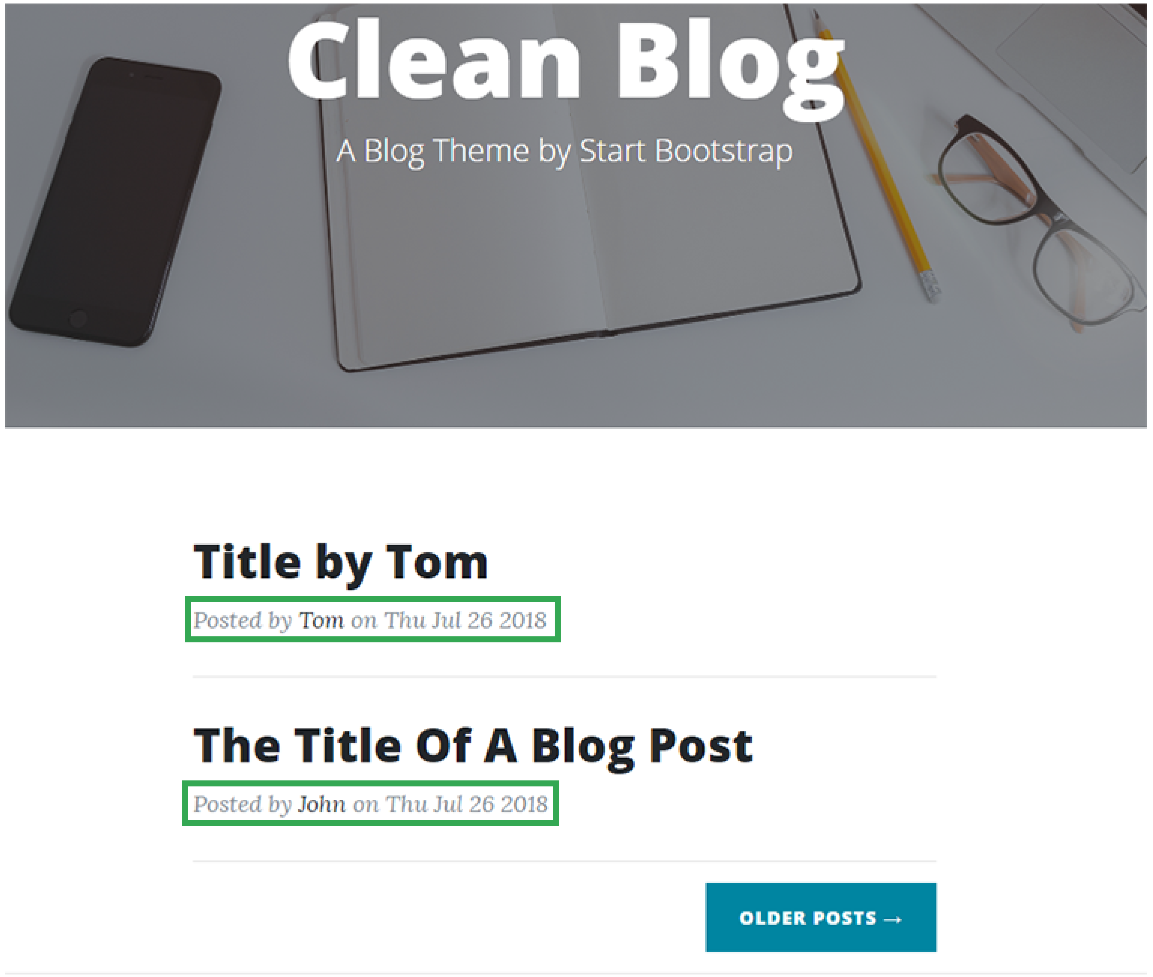


```
_id: ObjectId("5b59f0acb7142815e8973fea")
createdAt: 2018-07-26 12:01:34.950
username: "Tom"
title: "Title by Tom"
description: "This is a blog description by Tom"
content: "Awesome content and really great stuff that you can check out right he..."
__v: 0

_id: ObjectId("5b59f0f0b7142815e8973feb")
createdAt: 2018-07-26 12:01:34.950
username: "John"
title: "The Title Of A Blog Post"
description: "This is the description"
content: "This content section of the blog post was created by John. Isn't that..."
__v: 0
```

We can update **index.edge** so that we can display this new data now.

```
@layout('layouts.app')

@section('content')
<!-- Page Header -->
<header class="masthead" style="background-image: url('img/home-bg.jpg')">
  <div class="overlay"></div>
  <div class="container">
    <div class="row">
      <div class="col-lg-8 col-md-10 mx-auto">
        <div class="site-heading">
          <h1>Clean Blog</h1>
          <span class="subheading">A Blog Theme by Start Bootstrap</span>
        </div>
      </div>
    </div>
  </div>
</header>

<!-- Main Content -->
<div class="container">
  <div class="row">
    <div class="col-lg-8 col-md-10 mx-auto">
      @each(post in posts)
        <div class="post-preview">
          <a href="/post/{{ post._id }}">
            <h2 class="post-title">
              {{ post.title }}
            </h2>
          </a>
          <p class="post-meta">Posted by
            <a href="#">{{ post.username }}</a>
            on {{ post.createdAt.toDateString() }}
          </p>
        </div>
        <hr>
      @endeach
      <!-- Pager -->
      <div class="clearfix">
        <a class="btn btn-primary float-right" href="#">Older Posts &rarr;</a>
      </div>
    </div>
  </div>
</div>

<hr>
@endsection
```

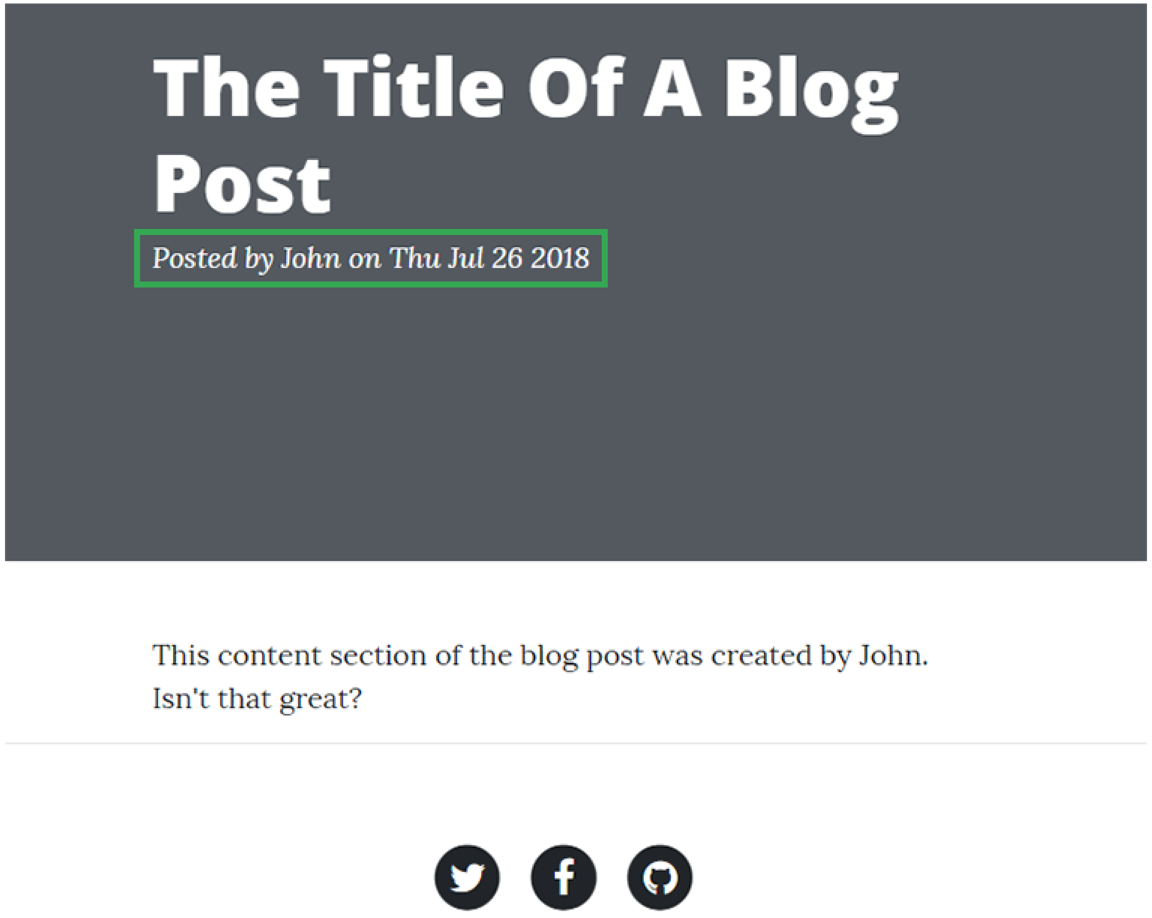
Visiting the home page of the blog shows that these new properties are working great!


Clean Blog
A Blog Theme by Start Bootstrap
Title by Tom
Posted by Tom on Thu Jul 26 2018
The Title Of A Blog Post
Posted by John on Thu Jul 26 2018
OLDER POSTS →

Let’s ensure **post.edge** is also now taking advantage of the post-meta information.

```
@layout('layouts.app')

@section('content')
  <!-- Page Header -->
  <header class="masthead">
    <div class="overlay"></div>
    <div class="container">
      <div class="row">
        <div class="col-lg-8 col-md-10 mx-auto">
          <div class="post-heading">
            <h1>{{ post.title }}</h1>
            <span class="meta">Posted by
              <a href="#">{{ post.username }}</a>
              on {{ post.createdAt.toDateString() }}
            </span>
          </div>
        </div>
      </div>
    </div>
  </header>

  <!-- Post Content -->
  <article>
    <div class="container">
      <div class="row">
        <div class="col-lg-8 col-md-10 mx-auto">
          {{ post.content }}
        </div>
      </div>
    </div>
  </article>

  <hr>
@endsection
```

Looks Good!


The Title Of A Blog Post

Posted by John on Thu Jul 26 2018

This content section of the blog post was created by John. Isn't that great?

How To Upload Images

Let’s add the ability to upload images when creating a new blog post. We can use the **express-fileupload** package to help us.

```
nodejs-blog-tutorial $npm install --save express-fileupload
```

We can add a field to add a new image when creating a post in **create.edge**.

```
@layout('layouts.app')

@section('content')
<!-- Page Header -->
```

```html
<header class="masthead">
  <div class="overlay"></div>
  <div class="container">
    <div class="row">
      <div class="col-lg-8 col-md-10 mx-auto">
        <div class="page-heading">
          <h1>Create New Post</h1>
        </div>
      </div>
    </div>
  </div>
</header>

<div class="container">
  <div class="row">
    <div class="col-md-8 offset-md-2">
      <form action="/posts/store" method="POST" encType="multipart/form-data
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Username</label>
            <input type="text" name="username" placeholder="Username" class='
          </div>
        </div>
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Title</label>
            <input type="text" name="title" placeholder="Title" class="form-contro
          </div>
        </div>
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Description</label>
            <input type="text" name="description" placeholder="Description" class
          </div>
        </div>
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Content</label>
            <textarea name="content" placeholder="Content ..." cols="30" rows="
          </div>
        </div>
        <div class="form-group mt-3">
          <input type="file" name="image" class="form-control-file">
        </div>
        <div class="form-group my-4 text-center">
          <button class="btn btn-primary">Create Post</button>
        </div>
      </form>
    </div>
  </div>
</div>
@endsection
```

We need to update the **Post.js** model to allow for our new image.

```
const mongoose = require('mongoose')

const PostSchema = new mongoose.Schema({
  title: String,
  description: String,
  content: String,
  username: String,
  image: String,
  createdAt: {
    type: Date,
    default: new Date()
  }
})

const Post = mongoose.model('Post', PostSchema)

module.exports = Post
```

We need to make a few updates to **index.js** as well to handle image uploads.

```
const path = require('path');
const expressEdge = require('express-edge');
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
const fileUpload = require("express-fileupload");

const Post = require('./database/models/Post');

const app = new express();

mongoose.connect('mongodb://localhost:27017/node-blog', { useNewUrlParser:
    .then(() => 'You are now connected to Mongo!')
    .catch(err => console.error('Something went wrong', err))

app.use(fileUpload());
app.use(express.static('public'));
app.use(expressEdge);
app.set('views', __dirname + '/views');
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({
    extended: true
}));

```

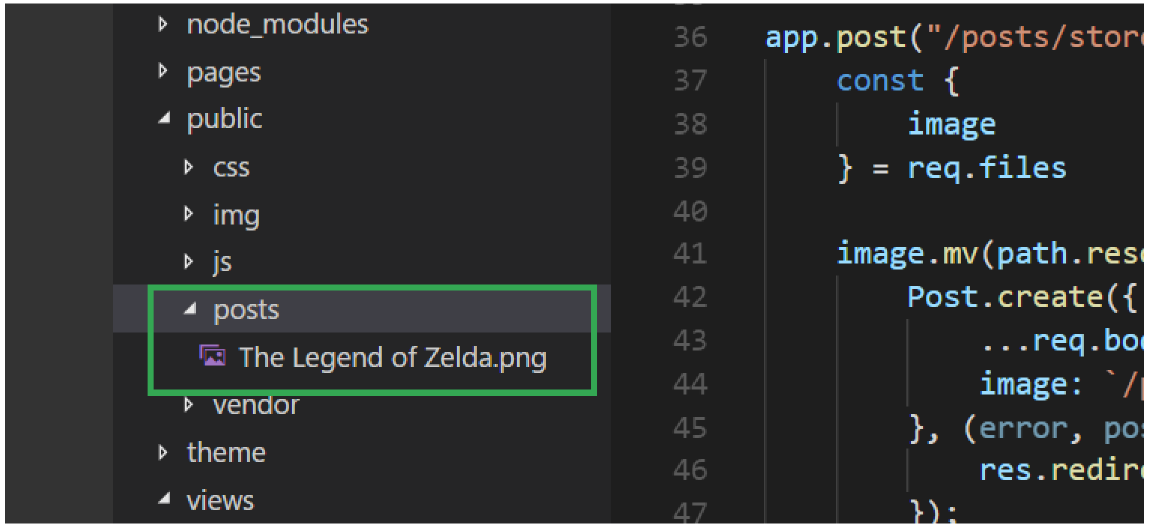
```
app.get('/', async (req, res) => {
    const posts = await Post.find({})
    res.render('index', {
        posts
    })
});

app.get('/posts/new', (req, res) => {
    res.render('create')
});

app.post("/posts/store", (req, res) => {
    const {
        image
    } = req.files

    image.mv(path.resolve(__dirname, 'public/posts', image.name), (error) => {
        Post.create({
            ...req.body,
            image: `/posts/${image.name}`
        }, (error, post) => {
            res.redirect('/');
        });
    })
});

app.get('/about', (req, res) => {
    res.render('about');
});

app.get('/contact', (req, res) => {
    res.render('contact');
});
```

```

app.get('/post/:id', async (req, res) => {
    const post = await Post.findById(req.params.id)
    res.render('post', {
        post
    })
});

app.listen(4000, () => {
    console.log('App listening on port 4000')
});
```

We also need a directory to hold the images in.

```
nodejs-blog-tutorial $mkdir public/posts
```

Now if we create a new post and include an image, it gets stored in this directory.



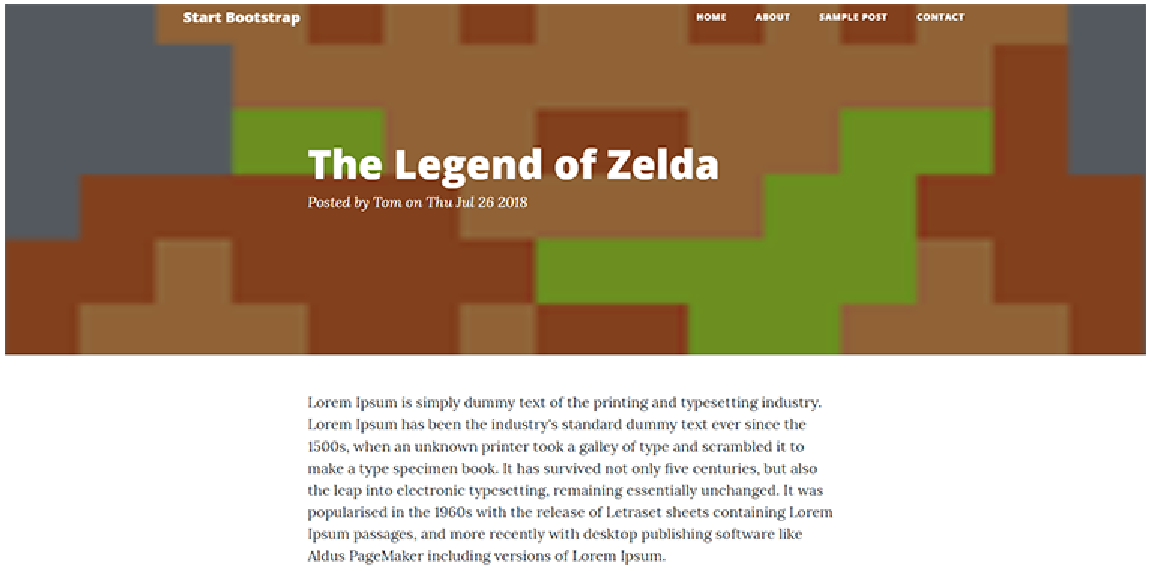
```
app.post("/posts/stor
    const {
        image
    } = req.files

    image.mv(path.res
        Post.create({
            ...req.bo
            image: `/
        }, (error, po
            res.redir
        });
```

Now we can display that image in each post we create by updating **post.edge**.

```
@layout('layouts.app')

@section('content')
```

```
  <!-- Page Header -->
  <header class="masthead" style="background-image: url('{{ post.image }}'
    <div class="overlay"></div>
    <div class="container">
      <div class="row">
        <div class="col-lg-8 col-md-10 mx-auto">
          <div class="post-heading">
            <h1>{{ post.title }}</h1>
            <span class="meta">Posted by
              <a href="#">{{ post.username }}</a>
              on {{ post.createdAt.toDateString() }}
            </span>
          </div>
        </div>
      </div>
    </div>
  </header>

  <!-- Post Content -->
  <article>
    <div class="container">
      <div class="row">
        <div class="col-lg-8 col-md-10 mx-auto">
          {{ post.content }}
        </div>
      </div>
    </div>
  </article>

  <hr>
@endsection
```

When viewing this post now, it looks pretty cool with the background image set dynamically!



The Legend of Zelda

Posted by Tom on Thu Jul 26 2018

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Adding Simple Validation To Blog Post Creation

We do not yet have any way to validate data before we try to submit a post. Let's add some middleware to help with this.

```
nodejs-blog-tutorial $mkdir middleware
nodejs-blog-tutorial $touch middleware/storePost.js
```

In the **storePost.js** file, we can add the most basic of validation. Basically, we just want all fields to be required. If any are missing, we just redirect back to the form.

```
module.exports = (req, res, next) => {
  if (!req.files.image || !req.body.username || !req.body.title || !req.body.descri
    return res.redirect('/posts/new')
  }

  next()
}
```

Now, let’s use that middleware in **index.js**.

```
const path = require('path');
const expressEdge = require('express-edge');
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
const fileUpload = require("express-fileupload");

const Post = require('./database/models/Post');

const app = new express();

mongoose.connect('mongodb://localhost:27017/node-blog', { useNewUrlParser:
    .then(() => 'You are now connected to Mongo!')
    .catch(err => console.error('Something went wrong', err))

app.use(fileUpload());
app.use(express.static('public'));
app.use(expressEdge);
app.set('views', __dirname + '/views');
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({
    extended: true
}));

const storePost = require('./middleware/storePost')
app.use('/posts/store', storePost)

app.get('/', async (req, res) => {
    const posts = await Post.find({})
    res.render('index', {
        posts
    })
});

app.get('/posts/new', (req, res) => {
    res.render('create')
});

app.post("/posts/store", (req, res) => {
    const {
        image
    } = req.files

    image.mv(path.resolve(__dirname, 'public/posts', image.name), (error) => {
        Post.create({
            ...req.body,
            image: `/posts/${image.name}`
        }, (error, post) => {
            res.redirect('/');
        });
    })
});

app.get('/about', (req, res) => {
    res.render('about');
});

app.get('/contact', (req, res) => {
    res.render('contact');
});

app.get('/post/:id', async (req, res) => {
    const post = await Post.findById(req.params.id)
    res.render('post', {
        post
    })
});

app.listen(4000, () => {
    console.log('App listening on port 4000')
});
```

Great! Simple validation is now working.

Adding Controllers for a Model View Controller Application

Have you noticed that the index.js file is starting to become very bloated. This is not really ideal. We can fix this by restructuring the application to an MVC architecture. We can create the controllers directory and the controller files we need now.

```
nodejs-blog-tutorial $mkdir controllers
nodejs-blog-tutorial $touch controllers/createPost.js
nodejs-blog-tutorial $touch controllers/getPost.js
nodejs-blog-tutorial $touch controllers/homePage.js
nodejs-blog-tutorial $touch controllers/storePost.js
```

In those files will be this code.
createPost.js

```
module.exports = (req, res) => {
    res.render("create");
};
```

getPost.js

```
const Post = require('../database/models/Post')

```

```
module.exports = async (req, res) => {
    const post = await Post.findById(req.params.id);
    res.render("post", {
        post
    });
}
```

homePage.js

```
const Post = require('../database/models/Post')

module.exports = async (req, res) => {
    const posts = await Post.find({});

    res.render("index", {
        posts
    });
}
```

storePost.js

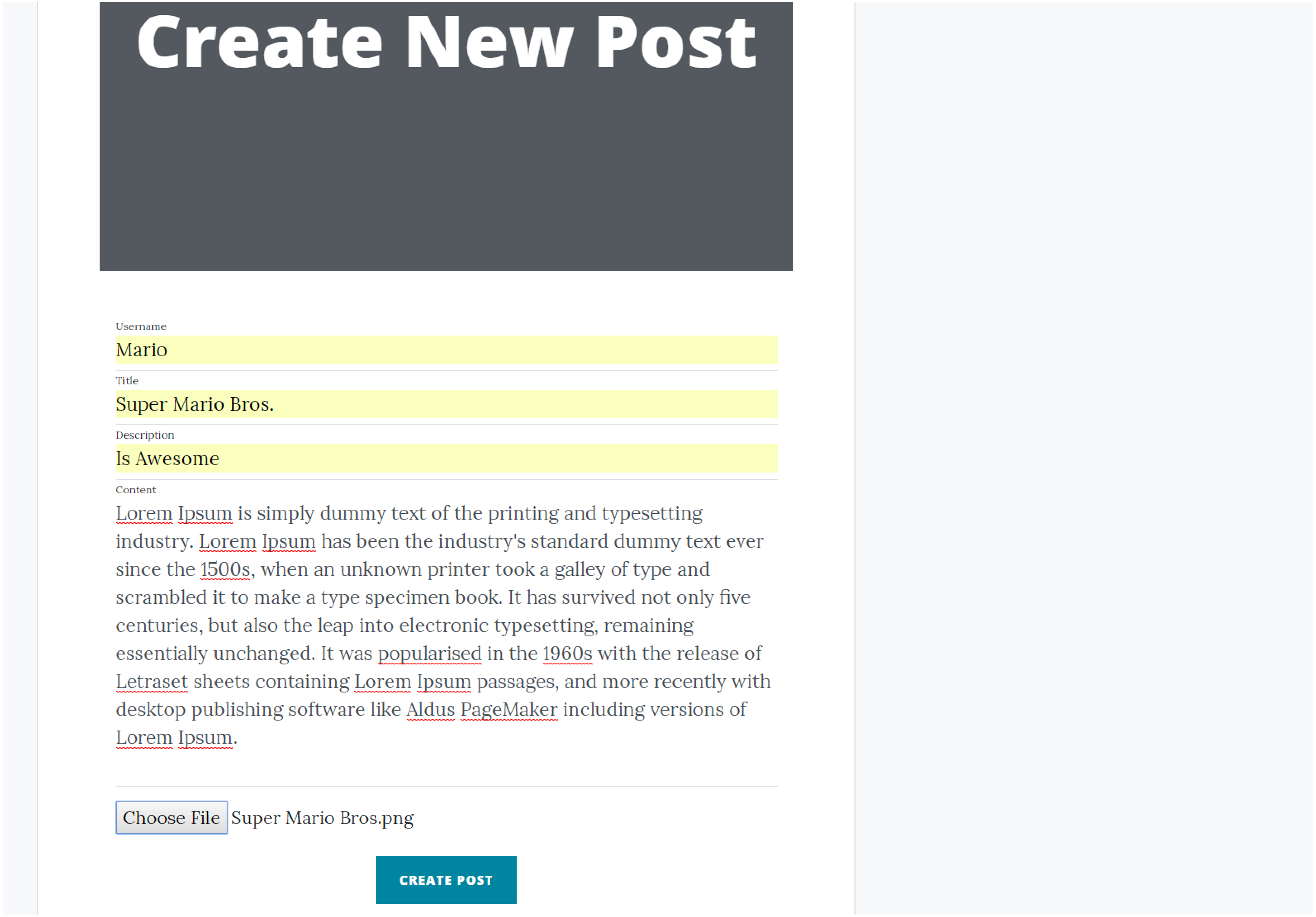
```
const path = require('path')
const Post = require('../database/models/Post')

module.exports = (req, res) => {
    const {
        image
    } = req.files

    image.mv(path.resolve(__dirname, '..', 'public/posts', image.name), (error) =
        Post.create({
            ...req.body,
            image: `/posts/${image.name}`
        }, (error, post) => {
            res.redirect("/");
        });
    })
}
```

With our controllers in place we can really simplify **index.js** now:

```
const expressEdge = require("express-edge");
const express = require("express");
const mongoose = require("mongoose");
const bodyParser = require("body-parser");
const fileUpload = require("express-fileupload");

const createPostController = require('./controllers/createPost')
const homePageController = require('./controllers/homePage')
const storePostController = require('./controllers/storePost')
const getPostController = require('./controllers/getPost')

const app = new express();

mongoose.connect('mongodb://localhost:27017/node-blog', { useNewUrlParser:
    .then(() => 'You are now connected to Mongo!')
    .catch(err => console.error('Something went wrong', err))

app.use(fileUpload());
app.use(express.static("public"));
app.use(expressEdge);
app.set('views', __dirname + '/views');

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

const storePost = require('./middleware/storePost')
```

```

app.use('/posts/store', storePost)

app.get("/", homePageController);
app.get("/post/:id", getPostController);
app.get("/posts/new", createPostController);
app.post("/posts/store", storePostController);

app.listen(4000, () => {
  console.log("App listening on port 4000");
});
```

Ok! With **MVC** now in place, does the application still work? Let’s create a new blog post about Super Mario Bros.



Create New Post

Username
Mario

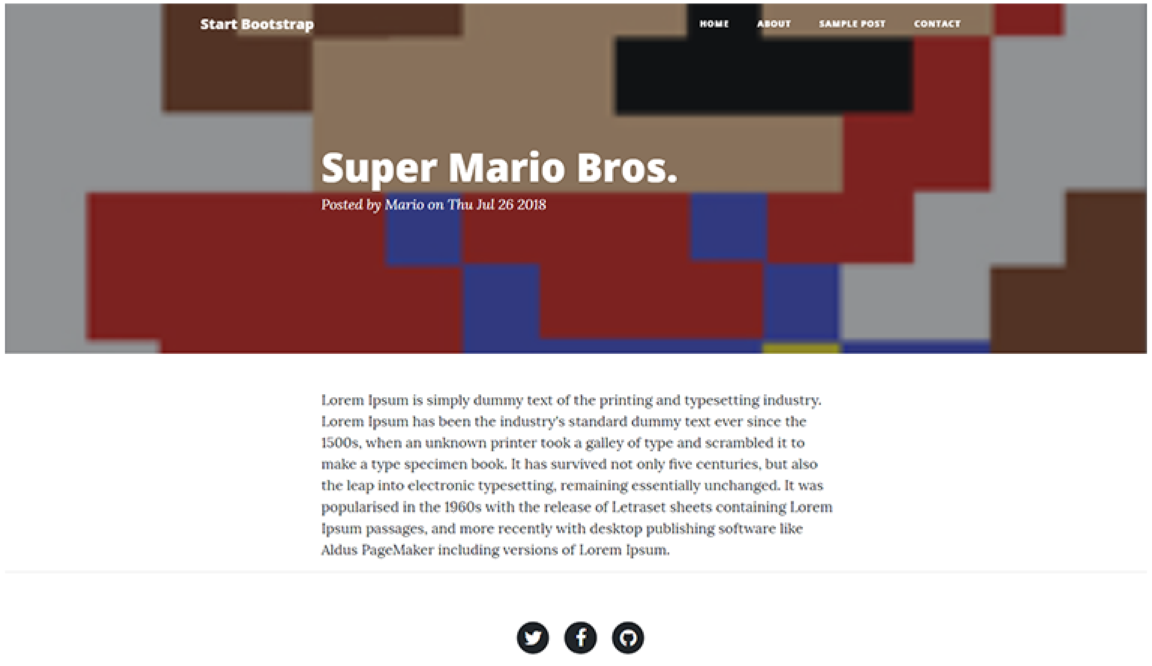
Title
Super Mario Bros.

Description
Is Awesome

Content
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Super Mario Bros.png

Navigating to that new blog post shows us that all is still working great!



Super Mario Bros.

Posted by Mario on Thu Jul 26 2018

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Adding User Registration

Let’s add the ability for a user to register to the site so they can post blogs. First we can create a new view file for that.

```
nodejs-blog-tutorial $touch views/register.edge
```

In **register.edge** we can add this markup.

```
@layout('layouts.app')

@section('content')
<!-- Page Header -->
<header class="masthead">
  <div class="overlay"></div>
  <div class="container">
    <div class="row">
```

```
        <div class="col-lg-8 col-md-10 mx-auto">
          <div class="page-heading">
            <h1>Create A New Account</h1>
          </div>
        </div>
      </div>
    </div>
  </header>

  <div class="container">
    <div class="row">
      <div class="col-md-8 offset-md-2">
        <form action="/users/register" method="POST" encType="multipart/form-d
          <div class="control-group">
            <div class="form-group floating-label-form-group controls">
              <label>Username</label>
              <input type="text" name="username" placeholder="Username" class='
            </div>
          </div>
          <div class="control-group">
            <div class="form-group floating-label-form-group controls">
              <label>Email</label>
              <input type="email" name="email" placeholder="Email" class="form-c
            </div>
          </div>
          <div class="control-group">
            <div class="form-group floating-label-form-group controls">
              <label>Password</label>
              <input type="password" name="password" placeholder="Password" cla
            </div>
          </div>

          <div class="form-group my-4 text-center">
```

```
     <button class="btn btn-primary">Register now</button>
    </div>
   </form>
  </div>
 </div>
</div>
@endsection
```

Now let’s add a **createUser.js** controller and add the code we need.

```
nodejs-blog-tutorial $touch controllers/createUser.js
```

```
module.exports = (req, res) => {
  res.render('register')
}
```

Finally, we can update **index.js** like so.

```
const expressEdge = require("express-edge");
const express = require("express");
const mongoose = require("mongoose");
const bodyParser = require("body-parser");
const fileUpload = require("express-fileupload");

const createPostController = require('./controllers/createPost');
const homePageController = require('./controllers/homePage');
const storePostController = require('./controllers/storePost');
const getPostController = require('./controllers/getPost');
const createUserController = require("./controllers/createUser");

const app = new express();

mongoose.connect('mongodb://localhost:27017/node-blog', { useNewUrlParser:
  .then(() => 'You are now connected to Mongo!')
  .catch(err => console.error('Something went wrong', err))

app.use(fileUpload());
app.use(express.static("public"));
app.use(expressEdge);
app.set('views', __dirname + '/views');

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

const storePost = require('./middleware/storePost')

app.use('/posts/store', storePost)

app.get("/", homePageController);
app.get("/post/:id", getPostController);
app.get("/posts/new", createPostController);
app.post("/posts/store", storePostController);
app.get("/auth/register", createUserController);

app.listen(4000, () => {
  console.log("App listening on port 4000");
});
```

Now we need a **User.js** model to handle users.

```
nodejs-blog-tutorial $touch database/models/User.js
```

```
const bcrypt = require('bcrypt')
const mongoose = require('mongoose')

const UserSchema = new mongoose.Schema({
    username: {
        type: String,
        required: true
    },
    email: {
        type: String,
        required: true,
        unique: true
    },
    password: {
        type: String,
        required: true
    }
})

UserSchema.pre('save', function (next) {
    const user = this

    bcrypt.hash(user.password, 10, function (error, encrypted) {
        user.password = encrypted
        next()
    })
})

module.exports = mongoose.model('User', UserSchema)
```

We also need a **storeUser.js** controller.

```
nodejs-blog-tutorial $touch controllers/storeUser.js
```

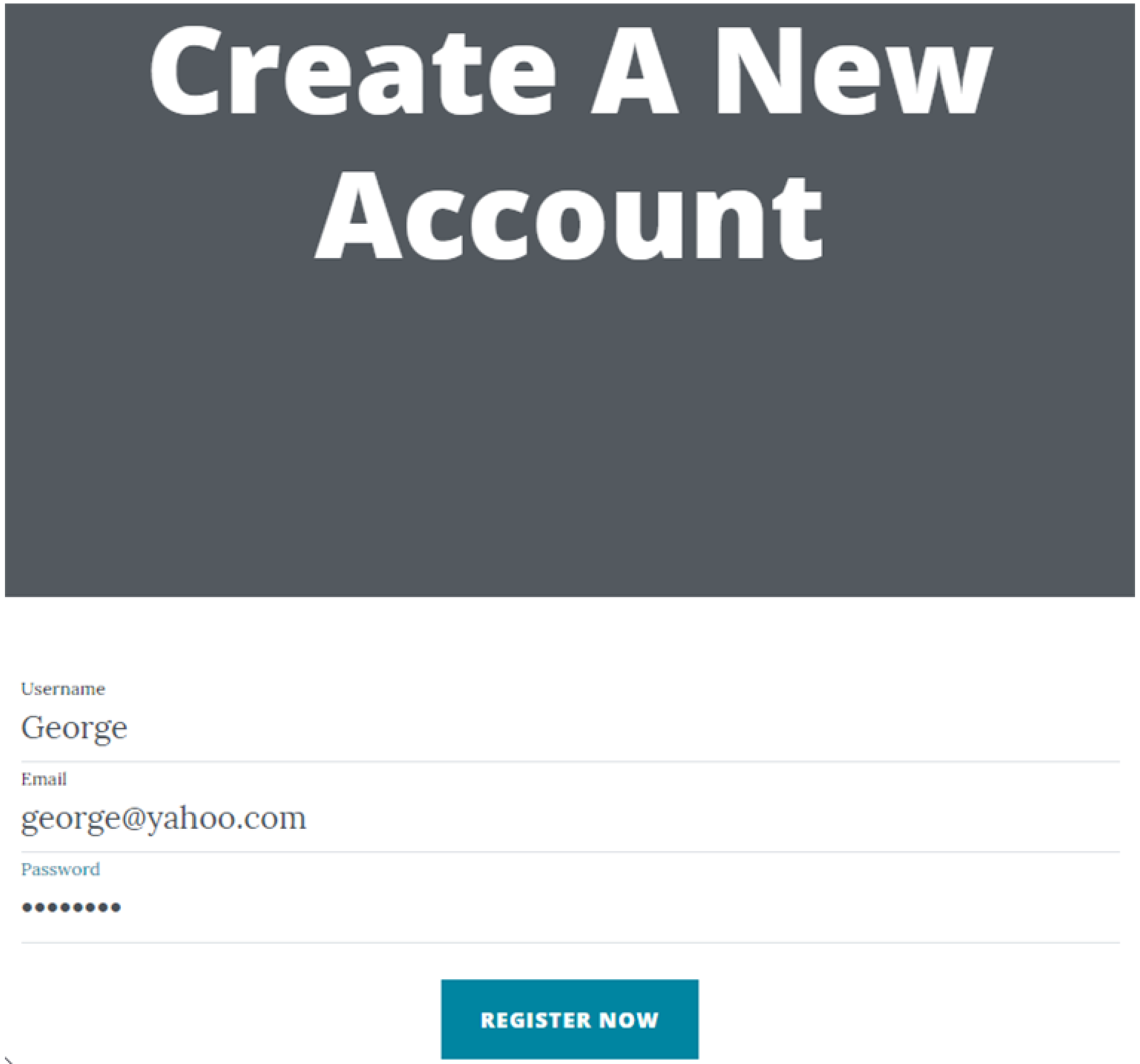
```
const User = require('../database/models/User')

module.exports = (req, res) => {
    User.create(req.body, (error, user) => {
        if (error) {
            return res.redirect('/auth/register')
        }
        res.redirect('/')
    })
}
```

Now we can reference this new controller in **index.js**.

```
const expressEdge = require("express-edge");
const express = require("express");
const mongoose = require("mongoose");
const bodyParser = require("body-parser");
const fileUpload = require("express-fileupload");

const createPostController = require('./controllers/createPost');
const homePageController = require('./controllers/homePage');
const storePostController = require('./controllers/storePost');
const getPostController = require('./controllers/getPost');
const createUserController = require("./controllers/createUser");
const storeUserController = require('./controllers/storeUser');

const app = new express();

mongoose.connect('mongodb://localhost:27017/node-blog', { useNewUrlParser:
    .then(() => 'You are now connected to Mongo!')
    .catch(err => console.error('Something went wrong', err))

app.use(fileUpload());
app.use(express.static("public"));
app.use(expressEdge);
app.set('views', __dirname + '/views');

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

```

const storePost = require('./middleware/storePost')

app.use('/posts/store', storePost)

app.get("/", homePageController);
app.get("/post/:id", getPostController);
app.get("/posts/new", createPostController);
app.post("/posts/store", storePostController);
app.get("/auth/register", createUserController);
app.post("/users/register", storeUserController);

app.listen(4000, () => {
  console.log("App listening on port 4000");
});
```

Ok with everything in place, let’s try to create a new user.

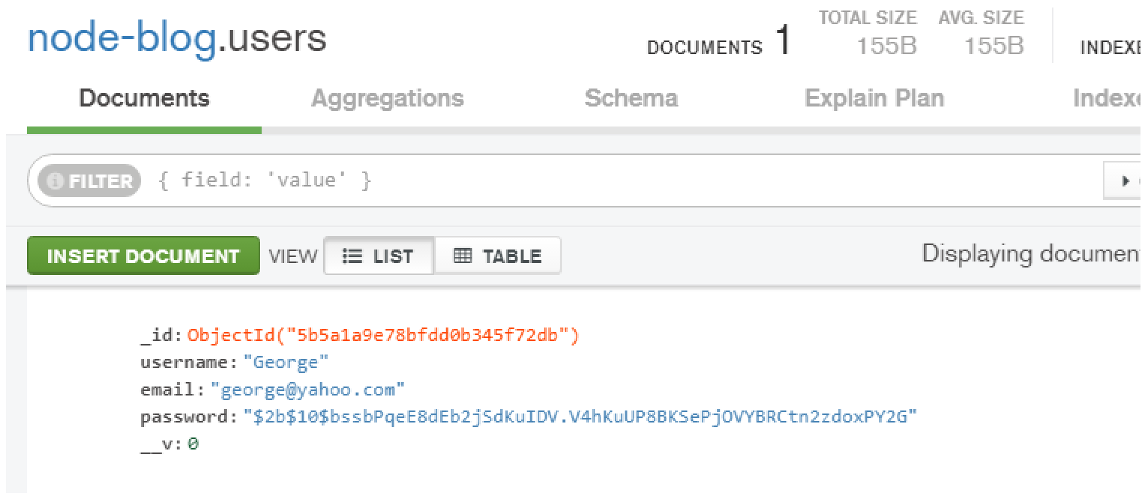

Create A New Account

Username
George

Email
george@yahoo.com

Password
●●●●●●●●●

Now we can use Mongo Compass to check and see if our new user is there. In fact, the user is there, so it looks like it is working great.



```
_id: ObjectId("5b5a1a9e78bfdd0b345f72db")
username: "George"
email: "george@yahoo.com"
password: "$2b$10$bssbPqeE8dEb2jSdKuIDV.V4hKuUP8BKSePjOVYBRCtn2zdoxPY2G"
__v: 0
```

Setting Up User Login

Since users are now able to register, we also need to give them the ability to log in. Let's make a **login.edge** file first.

```
nodejs-blog-tutorial $touch views/login.edge
```

We can add this markup to login.edge.

```
@layout('layouts.app')

@section('content')
<!-- Page Header -->
<header class="masthead">
  <div class="overlay"></div>
  <div class="container">
    <div class="row">
      <div class="col-lg-8 col-md-10 mx-auto">
        <div class="page-heading">
          <h1>Login</h1>
        </div>
      </div>
    </div>
  </div>
</header>

<div class="container">
  <div class="row">
    <div class="col-md-8 offset-md-2">
      <form action="/users/login" method="POST" encType="multipart/form-data
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Email</label>
            <input type="email" name="email" placeholder="Email" class="form-co
          </div>
        </div>
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Password</label>
            <input type="password" name="password" placeholder="Password" cla
          </div>
        </div>

        <div class="form-group my-4 text-center">
          <button class="btn btn-primary">Login</button>
        </div>
      </form>
    </div>
  </div>
</div>
@endsection
```

Now we can create the **login.js** controller file.

```
nodejs-blog-tutorial $touch controllers/login.js
```

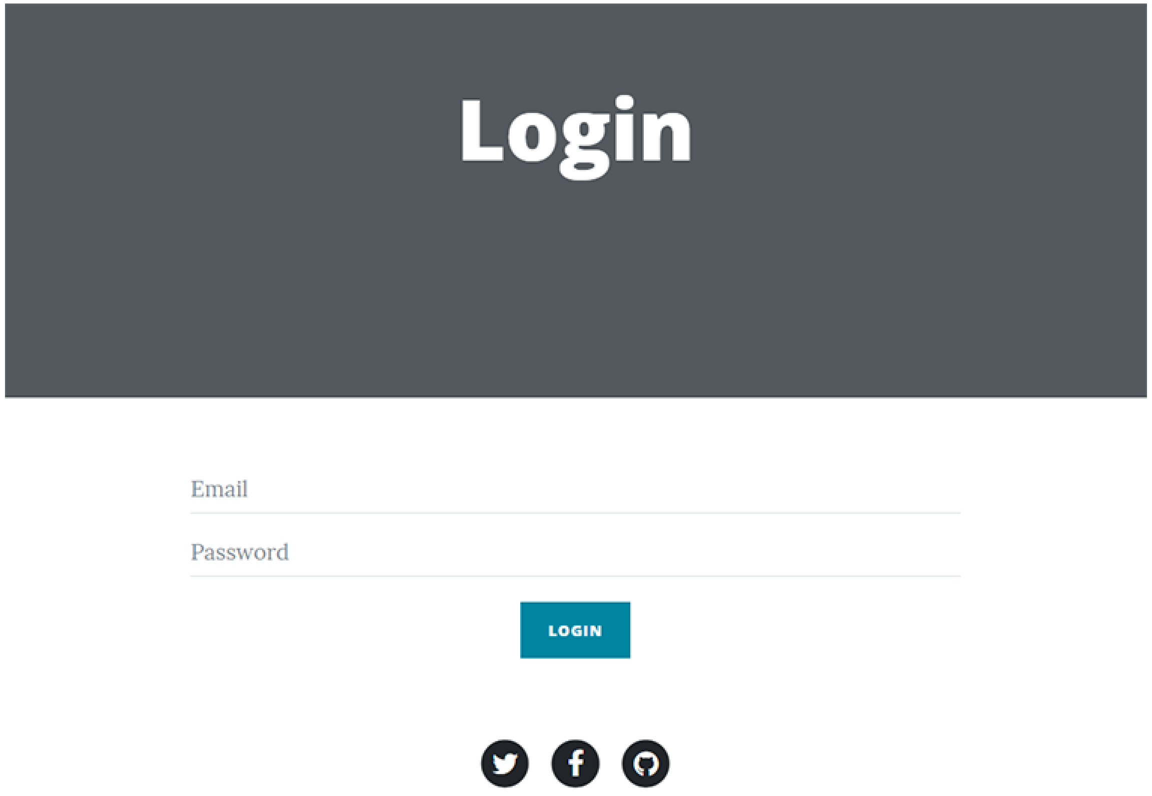
Here is the code for **login.js**.

```
module.exports = (req, res) => {
    res.render('login')
}
```

Let’s make use of that new loginController in **index.js**

```
const expressEdge = require("express-edge");
const express = require("express");
const mongoose = require("mongoose");
const bodyParser = require("body-parser");
const fileUpload = require("express-fileupload");

const createPostController = require('./controllers/createPost');
const homePageController = require('./controllers/homePage');
const storePostController = require('./controllers/storePost');
const getPostController = require('./controllers/getPost');
const createUserController = require("./controllers/createUser");
const storeUserController = require('./controllers/storeUser');
const loginController = require("./controllers/login");

const app = new express();

mongoose.connect('mongodb://localhost:27017/node-blog', { useNewUrlParser:
    .then(() => 'You are now connected to Mongo!')
    .catch(err => console.error('Something went wrong', err))

app.use(fileUpload());
app.use(express.static("public"));
app.use(expressEdge);
app.set('views', __dirname + '/views');

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

const storePost = require('./middleware/storePost')

app.use('/posts/store', storePost)

app.get("/", homePageController);
app.get("/post/:id", getPostController);
app.get("/posts/new", createPostController);
app.post("/posts/store", storePostController);
app.get('/auth/login', loginController);
app.get("/auth/register", createUserController);
app.post("/users/register", storeUserController);

app.listen(4000, () => {
  console.log("App listening on port 4000");
});
```

Just like that we have a nice login page now!



Handling the login POST request

The section above allows us to present a form to the user so they can type in their credentials. Now when they click the button to log in, we need to have the code in place to handle that POST request. First, we'll create a **loginUser.js** controller and add some code.

```
nodejs-blog-tutorial $touch controllers/loginUser.js
```

```
const bcrypt = require('bcrypt')
const User = require('../database/models/User')

module.exports = (req, res) => {
    const {
        email,
        password
    } = req.body;
    // try to find the user
    User.findOne({
        email
    }, (error, user) => {
        if (user) {
            // compare passwords.
            bcrypt.compare(password, user.password, (error, same) => {
                if (same) {
                    // store user session.
                    res.redirect('/')
                } else {
                    res.redirect('/auth/login')
                }
            })
        } else {
            return res.redirect('/auth/login')
        }
    })
}
```

Once again, we need to use this controller in **index.js**

```
const expressEdge = require("express-edge");
const express = require("express");
const mongoose = require("mongoose");
const bodyParser = require("body-parser");
const fileUpload = require("express-fileupload");

const createPostController = require('./controllers/createPost');
const homePageController = require('./controllers/homePage');
const storePostController = require('./controllers/storePost');
const getPostController = require('./controllers/getPost');
const createUserController = require("./controllers/createUser");
const storeUserController = require('./controllers/storeUser');
const loginController = require("./controllers/login");
const loginUserController = require('./controllers/loginUser');

const app = new express();

mongoose.connect('mongodb://localhost:27017/node-blog', { useNewUrlParser:
    .then(() => 'You are now connected to Mongo!')
    .catch(err => console.error('Something went wrong', err))

app.use(fileUpload());
app.use(express.static("public"));
app.use(expressEdge);
app.set('views', __dirname + '/views');

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

const storePost = require('./middleware/storePost')

app.use('/posts/store', storePost)

app.get("/", homePageController);
app.get("/post/:id", getPostController);
app.get("/posts/new", createPostController);
app.post("/posts/store", storePostController);
app.get('/auth/login', loginController);
app.post('/users/login', loginUserController);
app.get("/auth/register", createUserController);
app.post("/users/register", storeUserController);

app.listen(4000, () => {
  console.log("App listening on port 4000");
});
```

Persisting Logins Using Sessions

All of the logic above works great, but the user is not persisted to a session just yet, let’s fix that now by installing **express-session**.

```
nodejs-blog-tutorial $npm install express-session
```

Next, add the `express-session` package to **index.js**.

```
const expressEdge = require("express-edge");
const express = require("express");
const mongoose = require("mongoose");
const bodyParser = require("body-parser");
const fileUpload = require("express-fileupload");
const expressSession = require('express-session');
```

We also want to use that package like so.

```
const expressEdge = require("express-edge");
const express = require("express");
const mongoose = require("mongoose");
const bodyParser = require("body-parser");
const fileUpload = require("express-fileupload");
const expressSession = require('express-session');

const createPostController = require('./controllers/createPost');
const homePageController = require('./controllers/homePage');
const storePostController = require('./controllers/storePost');
const getPostController = require('./controllers/getPost');
const createUserController = require("./controllers/createUser");
const storeUserController = require('./controllers/storeUser');
const loginController = require("./controllers/login");
const loginUserController = require('./controllers/loginUser');

const app = new express();

app.use(expressSession({
    secret: 'secret'
}));
```

We need to adjust the **loginUser.js** controller to store the user's session.

```
const bcrypt = require('bcrypt')
const User = require('../database/models/User')

module.exports = (req, res) => {
    const {
        email,
        password
    } = req.body;
    // try to find the user
    User.findOne({
        email
    }, (error, user) => {
        if (user) {
            // compare passwords.
            bcrypt.compare(password, user.password, (error, same) => {
                if (same) {
                    req.session.userId = user._id
                    res.redirect('/')
                } else {
                    res.redirect('/auth/login')
                }
        })
      } else {
        return res.redirect('/auth/login')
      }
    })
}
```

With our session now in place, we can put a check in our **createPost.js** controller. If the user is not logged in, we can redirect to the login page. We will only show the “Create new Post” page to a logged in user.

```
module.exports = (req, res) => {
    if (req.session.userId) {
        return res.render("create");
    }

    res.redirect('/auth/login')
};
```

Storing Sessions In MongoDB

We can use the **connect-mongo** package to enable the ability to store sessions in the database. Let’s do that now.

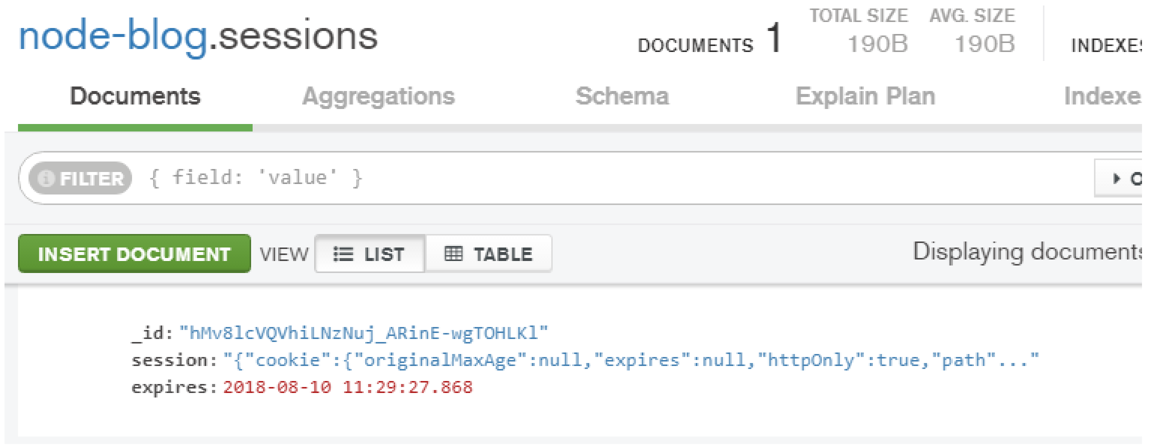
```
nodejs-blog-tutorial $npm i connect-mongo
```

Once installed, we can update index.js like so.

```
const expressEdge = require("express-edge");
const express = require("express");
const mongoose = require("mongoose");
const bodyParser = require("body-parser");
const fileUpload = require("express-fileupload");
const expressSession = require('express-session');
const connectMongo = require('connect-mongo');

const createPostController = require('./controllers/createPost');
const homePageController = require('./controllers/homePage');
const storePostController = require('./controllers/storePost');
const getPostController = require('./controllers/getPost');
const createUserController = require("./controllers/createUser");
const storeUserController = require('./controllers/storeUser');
const loginController = require("./controllers/login");
const loginUserController = require('./controllers/loginUser');

const app = new express();

mongoose.connect('mongodb://localhost:27017/node-blog', { useNewUrlParser:
    .then(() => 'You are now connected to Mongo!')
    .catch(err => console.error('Something went wrong', err));

const mongoStore = connectMongo(expressSession);

app.use(expressSession({
    secret: 'secret',
    store: new mongoStore({
        mongooseConnection: mongoose.connection
    })
}));

app.use(fileUpload());
app.use(express.static("public"));
app.use(expressEdge);
app.set('views', __dirname + '/views');

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

```

const storePost = require('./middleware/storePost')

app.use('/posts/store', storePost)

app.get("/", homePageController);
app.get("/post/:id", getPostController);
app.get("/posts/new", createPostController);
app.post("/posts/store", storePostController);
app.get('/auth/login', loginController);
app.post('/users/login', loginUserController);
app.get("/auth/register", createUserController);
app.post("/users/register", storeUserController);

app.listen(4000, () => {
  console.log("App listening on port 4000");
});
```

Now when we log in, the session information is stored in MongoDB. In fact when we check our database in MongoDB using Compass, we can see that new sessions collection.



```
_id: "hMv8lcVQVhiLNzNuj_ARinE-wgTOHLKl"
session: "{"cookie":{"originalMaxAge":null,"expires":null,"httpOnly":true,"path"..."
expires: 2018-08-10 11:29:27.868
```

Set Up Authentication Middleware

We can protedct various pages or routes using middleware. Let's create **auth.js** and add the needed code.

```
nodejs-blog-tutorial $touch middleware/auth.js
```

```
const User = require('../database/models/User')

module.exports = (req, res, next) => {
    User.findById(req.session.userId, (error, user) => {
        if (error || !user) {
            return res.redirect('/')
        }

        next()
    })
}
```

Now you can import that module into **index.js** and pass it as the second parameter to app.get("/posts/new", createPostController).

```
const auth = require("./middleware/auth");

app.use('/posts/store', storePost)

app.get("/", homePageController);
app.get("/post/:id", getPostController);
app.get("/posts/new", auth, createPostController);
app.post("/posts/store", storePostController);
app.get('/auth/login', loginController);
app.post('/users/login', loginUserController);
app.get("/auth/register", createUserController);
app.post("/users/register", storeUserController);
```

Flash Messages with connect-flash

Let’s see if we can set up flash messages for when a user tries to submit a form with errors. We’ll use the **connect-flash** package to do this. We an install it first.

```
nodejs-blog-tutorial $npm i connect-flash
```

Import into **index.js** using:

```
const connectFlash = require("connect-flash");
```

Then register it in **index.js** like this:

```
app.use(connectFlash());
```

With `connect-flash` in place, we can now use it in the **storeUser.js** controller.

```
const User = require('../database/models/User')

module.exports = (req, res) => {
    User.create(req.body, (error, user) => {
        if (error) {
            const registrationErrors = Object.keys(error.errors).map(key => error.

            req.flash('registrationErrors', registrationErrors)
            return res.redirect('/auth/register')
        }
        res.redirect('/')
    })
}
```

The **createUser.js** controller will need to be updated as well.

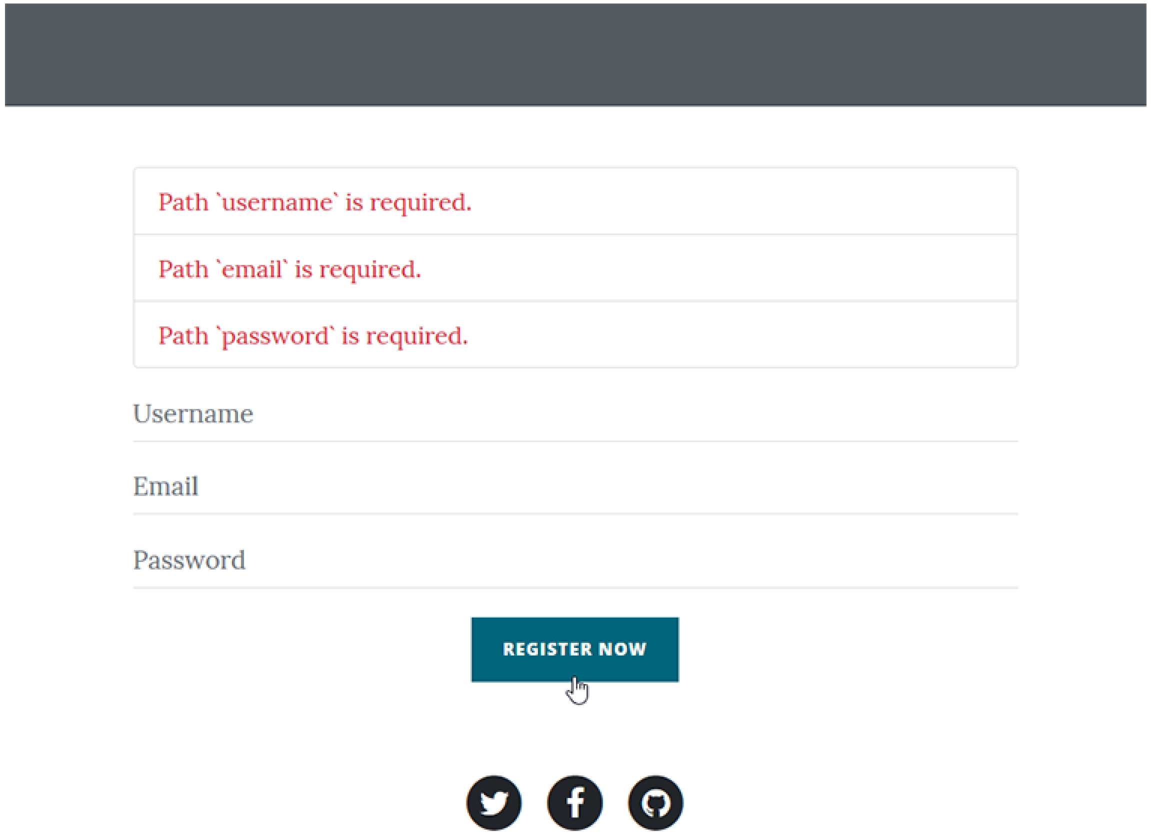
```
module.exports = (req, res) => {
    res.render('register', {
        errors: req.flash('registrationErrors')
```

```
  })
}
```

Lastly, in order to actually display these flash messages, we can update **register.edge** by adding this snippet just above the form.

```
@if(errors.length > 0)
  <ul class="list-group">
    @each(error in errors)
      <li class="list-group-item text-danger">{{ error }}</li>
    @endeach
  </ul>
@endif
```

Now, if a user makes a mistake during the registration process, the errors will be displayed just one time.


Path `username` is required.

Path `email` is required.

Path `password` is required.

Preventing Authenticated Users From Visiting Register or Login Pages

Once a user is logged in, they no longer need to see the Register or Login pages. We can apply a middleware to enforce this.

```
nodejs-blog-tutorial $touch middleware/redirectIfAuthenticated.js
```

```
const User = require('../database/models/User')

module.exports = (req, res, next) => {
  if (req.session.userId) {
    return res.redirect('/')
  }

  next()
}
```

Now, all of the middleware in **index.js** can be updated like so.

```
const storePost = require('./middleware/storePost');
const auth = require("./middleware/auth");
const redirectIfAuthenticated = require('./middleware/redirectIfAuthenticated')

app.get("/", homePageController);
app.get("/post/:id", getPostController);
app.get("/posts/new", auth, createPostController);
app.post("/posts/store", auth, storePost, storePostController);
app.get("/auth/login", redirectIfAuthenticated, loginController);
app.post("/users/login", redirectIfAuthenticated, loginUserController);
app.get("/auth/register", redirectIfAuthenticated, createUserController);
app.post("/users/register", redirectIfAuthenticated, storeUserController);
```

Only Display Login and Register links to guests

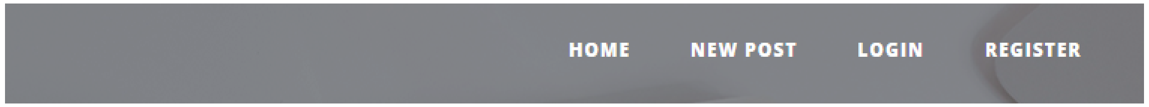
We also need a way to conditionally show the login and register links based on if the user is logged in or not. First, update **index.js** like so.

```
const expressEdge = require("express-edge");
const express = require("express");
const edge = require("edge.js");
const mongoose = require("mongoose");
const bodyParser = require("body-parser");
const fileUpload = require("express-fileupload");
const expressSession = require('express-session');
const connectMongo = require('connect-mongo');
const connectFlash = require("connect-flash");
const createPostController = require('./controllers/createPost');
const homePageController = require('./controllers/homePage');
const storePostController = require('./controllers/storePost');
const getPostController = require('./controllers/getPost');
const createUserController = require("./controllers/createUser");
const storeUserController = require('./controllers/storeUser');
const loginController = require("./controllers/login");
const loginUserController = require('./controllers/loginUser');

const app = new express();

mongoose.connect('mongodb://localhost:27017/node-blog', { useNewUrlParser:
    .then(() => 'You are now connected to Mongo!')
    .catch(err => console.error('Something went wrong', err));

app.use(connectFlash());

const mongoStore = connectMongo(expressSession);

app.use(expressSession({
    secret: 'secret',
    store: new mongoStore({
        mongooseConnection: mongoose.connection
    })
}));

app.use(fileUpload());
app.use(express.static("public"));
app.use(expressEdge);
app.set('views', __dirname + '/views');

app.use('*', (req, res, next) => {
    edge.global('auth', req.session.userId)
    next()
});
```

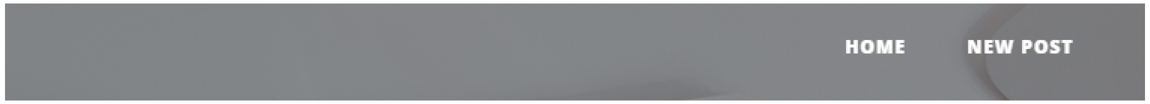
In **app.edge**, we only show the login and register links if the user is not logged in.

```
@if(!auth)
  <li class="nav-item">
    <a class="nav-link" href="/auth/login">Login</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="/auth/register">Register</a>
  </li>
@endif
```

Now, a guest will see the register and login links.



The logged in user does not see those links.



Logging Out

First off, we can update the Nav area in **app.edge** like so.

```
<ul class="navbar-nav ml-auto">
  <li class="nav-item">
    <a class="nav-link" href="/">Home</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="/posts/new">New Post</a>
  </li>
  @if(!auth)
    <li class="nav-item">
      <a class="nav-link" href="/auth/login">Login</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="/auth/register">Register</a>
    </li>
  @else
    <li class="nav-item">
      <a class="nav-link" href="/auth/logout">Logout</a>
    </li>
  @endif
</ul>
```

Add this route to **index.js**.

```
app.get("/auth/logout", redirectIfAuthenticated, logoutController);
```

Create the **logout.js** controller and add the needed code.

`nodejs-blog-tutorial $touch controllers/logout.js`

```
module.exports = (req, res) => {
    req.session.destroy(() => {
        res.redirect('/')
    })
}
```

Make sure to import the logout controller like so.

```
const logoutController = require("./controllers/logout");
```

This will allow the user to easily logout of the application.

Node.js Blog Tutorial Summary

That about sums it up for this tutorial on how to build a blog from scratch using Node.js and various packages from the NPM ecosystem. That’s for checking it out!

#javascript #nodejs

← Node MongoDB Native | Vue.js Express Tutorial →