Project Idea: Creating your own blog platform

Sooner or later everyone creates a content publishing system, right? :)

The idea is to put together a lot of the things I've learned over the past couple years and build a Node-based platform to publish content online.

It is also a way to learn server-side best practices and a lot of the things I read as far as performance and good web citizenship.

Finally, this project is an exercise on front-end development. Put your money where your mouth is and use the newer web technologies.

Content And Structure

I'm looking to replace multiple pieces that used to exists and, some, still exists spread around my domain. The idea is to create a simple way to manage these pieces in one app.

Long form writing is the one that I'm currently finding more and more intriguing. Combined with a working front-end (discussed later) it gives me the flexibility to do pretty much anything I may want to.

A short list of the content areas is below:

- Projects
- Space to serve static files
    - WebGL
    - Docs
- Long Form Writing
    - Markdown Authoring
    - Syntax Highlighting (Prism.js)

Backend Requirements

This is where a lot of the work will happen. I know how to work with most of the CRUD functions in a Express/MongoDB/Mongoose/Handlebars stack but now I

want to get the full functionality in the back-end before jumping to the front end and the decisions that need to happen there.

I'm also experimenting with Google Cloud as the hosting platform. Google App. Engine Standard Environment has recently included Node 8 as part of the offering. There are several outstanding questions regarding the product and whether Google offers a MongoDB equivalent database but having everything in one environment sounds interesting.

An incomplete list of the stack:

- Node
    - Express
- Database
    - MongoDB
        - Mongoose
- Templating Engine
    - Handlebars
- Deployment
    - Google Cloud

Front-End Requirements

In the front-end I'm working on two primary questions:

Do I want to use a framework or design system library? I've been looking at the new Material Design system and it looks promising.

On the other hand, creating my own design system library would make it easier to incorporate other ideas, like testing if we can use a single variable font for an entire site or application.

Jason Pamental has done a lot of work exploring this area; the ones I like the best are his FF Meta Variable Font Demo and the Outdoor Adventure Variable Font/Art Direction Demo Codepen demos where he uses variable fonts (and in the case of the FF Meta Variable Font Demo a single font) to build an application with all the typographical requirements.

This would also allow me to create a design system that works for me. Perhaps building a smaller set of elements that I can compose as I need using Handlebar's layouts and partials system may be a better idea than adopting a system

wholesale.

Building the site

Building the back-end will be an exercise on learning how to use Express and Handlebars. We'll start with configuring the site.

Getting Started

Since this is a Node application we have to create the default `package.json` file. The following command will generate the package file with default values.

```
npm init -y
```

The next step is to install Express globally. This will not install any packages in our project, it'll just build it.

```
npm i -g expess
```

Once Express is installed globally move to the parent directory of where you want to install the project and run the command below:

```
express --view hbs --css sass --git my-project
```

This command will create an Express application with the following configuration:

- Handlebars for the views
- SASS (SCSS) for the CSS pre-processing
- A default `.gitignore` file

When it's done installing follow the instructions and run the following commands:

```
cd my-project
npm i
```

This will download and add a long list of packages to `package.json` and get the

project ready to actually begin writing code. To make sure that everything works, run this command:

```
node DEBUG=publisher-demo:* ./bin/www
```

Running your application like this requires that you stop the server and restart it when you make any changes. That's tedious and I don't really want to do that.

To make things better, we'll use nodemon to monitor our files and automatically restart the application when we make changes. This is good for development but ***it is not good for production***. Please keep that in mind

```
npm i -D nodemon
```

And modify your start up command to look like this:

```
DEBUG=publisher-demo:* nodemon ./bin/www
```

Basic Layout and functionality

The basic layout of an Express application looks like this. In the rest of this section we'll dissect the code and see what it does and how it affects the application.

```
.
├── app.js
├── bin
|   └── www
├── package.json
├── static
├── public
|   ├── images
|   ├── javascripts
|   └── stylesheets
├── routes
|   ├── index.js
|   └── users.js
```

```
└── views
    ├── error.hbs
    ├── index.hbs
    └── layout.hbs
```

uEven though it's not the first file on the list, I'll explain `bin/www` first. It is a Node script that will configure our server and make sure that there are no errors in the server configuration before the server runs.

`app.js` is the server script. You may see examples that call it `server.js` or `index.js`. This is where we configure the server and its behavior.

`package.json` holds Node and NPM related configuration and the list of packages installed for the app.

`static` is the place for all the sta

The `public` directory hold static assets for the application such as images, scripts, stylesheets, fonts and anything else that we load rather than generate.

The files in the `routes` directory are the controllers for different areas of the application. Rather than have one monolithical router that loads everything from one central place I've followed the example of the app and broke the routes into their intended purpose.

`views` holds all the Handlebar templates that will render your content on the users' browsers.

Configuring the database

To make sure we keep our database connection from being shown to users or people who don't need it, I created a `config` directory. In this directory I created a `database.js` file to hold all my database-related configuration.

```
module.exports = {
  // For local development only
  'url': 'mongodb://127.0.0.1:27017/publisher',
};
```

```
'url'
```

Next we need to make sure that Mongoose is installed. We can write the configuration without having it installed, but I forgot to install it and couldn't figure out why it wasn't working. The command is a simple NPM install command:

```
npm i mongoose
```

Finally we write the database connection into our main entry point, app.js.

```
/ Database
const connection = require('./config/database.js');
const mongoose = require('mongoose');

'mongoose'// DB Connection Stuffconnect(connection.url);
mongoose.Promise = global.Promise;
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', () => {
  console.log(`Database connection to ${connection'error'as successful :)
});
`Database connection to ${connection.url} was successful :)`
```

Now, whenever we fire up the application, assuming that the database is up and running and that there is a database named publisher, we'll get a message in the console that the connection to the database was successful.

Making Changes to The Index Route

The Index Route is the basic route that users will hit when they access the URL for your application. In theory we could get away with using one route in this router, something like the code below:

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

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

m'/'le.exports = router;
```

This will render the index template when it receives a `get` request for `/` which is the root of the application.

I've added a second route for an about page. It's almost identical to the homepage route except for the path we follow, the template we render and the name of the page we are rendering. The route now looks like this:

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

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

r'/'er.get('/about', function(req, res, next) {
  res.render('about', {
    title: 'About',
  });
});

module.exports = router;
```

We can add routes for additional functionality that will work the root of the application. For everything else I'm creating separate routes.

Projects

Projects the first route where we will interact with the database using Mongoose. This requires a model and the corresponding route.

We'll look at the model first.

Building The Model

The model describes the shape of the data we want to work with in the database. It also describe constraints for each item in the model; for example, the project name is required and must be unique in the collection.

When the definition is complete we export the model as `Project`, an instance of `mongoose.Schema`.

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

const projectSchema = new Schema({
  name: {
    type: String,
    required: true,
    unique: true,
  },
  slug: {
    type: String,
    default: '',
  },
  author: {
    type: Stri'',
    default: '',
  },
  created: {
    type: Date,
    default: Date.now,
  },
  updated: {
    type: Date,
```

```
    default: Date.now,
  },
  summary: {
    type: String,
    required: true,
    default: 'Default Content, please change',
  },
  photo: {
    type: String,
''  lowercase: true,
    trim: true,
    default: '',
  },
  codeURL: {
    type: String,
    lowercase: true,
    trim: true,
    default: '',
  },
  otherURL: {
    type: String,
    lowercase: true,
    trim: true,
    default: '',
  },
  writingURL: {
    type: String,
    lowercase: true,
    trim: true,
    default: '',
  },
});

''// Export the Mongoose model
module.exports = mongoose.model('Project', projectSchema);
```

The Route

The route for projects is where the pedal hits the metal. It makes use of everything we've worked with so far to build the projects CRUD (Create, Read, Update, Delete) functionality.

We use slugify to create slugs from the title of the page. so we need to import it.

The default route will query the database for all projects and then use that JSON data to render the projects template.

`/new` will render a form to add new projects to the database. The form uses the `post` HTTP verb and the `/projects` URL to add a project to the database.

To retrieve a single project I `get /:slug`. This route will get the single project matching the slug we choose.

Using `post` in a route to projects will actually create the resource in the database. It will then redirect to the default route and display all the projects.

There are two additional routes that need to be created.

The `put /:slug` project will edit an individual project and update the content of the project in the database. It is aso possible to use `patch` instead of put.

Finally `delete /:slug` will delete the refernced project from the database.

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

// Get the model
const Project = require('../models/project');
'../models/project'// To create slugs for entriesquire('@sindresorhus/slug

router.get('/', function(req, res, next) {
  Project.find({}, function(err, projects) {
    if (err) {
      res.error(500).send(err);
    } else {
```

```
      r'@sindresorhus/slugify'
        title: 'current projects',
        projects: projects,
      });
    }
  });
});

router.get('/new', function(req, res, next) {
  res.render('form', {
    title: 'Add New Project',
  });
});

router.get('/:slug', function(req, res, next) {
  Project.find({slug: req.params.slug}, function(err, project) {
    if (err) {
      res.error(500).send(err);
    } else {
      res.render('project', {
     'current projects't,
      });
    }
  });
});

router.post('/', function(req, res, next) {
  // Create a new ins'/'// Create a new instance of the Project model
  let project = new Project();

  project.name = req.body.name;
  project.slug = slugify(req.body.name);
  project.author = req.body.author;
  project.type = req.body.type;
  project.summary = req.body.summary;
  project.photo = req.body.photo;
  project.codeURL = req.body.codeURL;
  project.otherURL = req.body.otherURL;
```

```
  project.writingURL = req.body.writingURL;

  project.save(function(err) {
    if (err) {
      res.send(err);
    }

    res.redirect('/projects');
  });
});

router.put('/:slug', function(req, res, next) {
  res.send('Still working on put /:slug ');
});

router.delete('/:slug', function(req, res, next) {
  res.send('still working on delete /:slug');
})

module.exports = router;
```

The views