How I build Node.js Applications

1 APR 2015 NODEJS NODE ARCHITECTURE WORKFLOW

"Keep it simple, keep it modular."

Today I would like to share with you how I build Node.js applications with the hope that someone else will find it useful. This article is structured in a sequence of steps that I use in my workflow and will attempt to be as detailed as possible.

Development Process

My development process usually begins with a wireframe illustrating the project requirements. It is very important to plan how you will build your application before writing any code. Here is a series of steps I like to follow for each project:

1. Analyse project wireframe and understand domain requirements
2. Create README file with project documentation
3. Based on the wireframe, document all routes and API end points
4. Create a domain model
5. Go shopping - Select software stack and dependencies
6. Set up project repo and do necessary installations
7. Write database schema, and create database
8. Start Coding, Create Models and Collections
9. Write unit tests
10. Create Controllers and Library modules
11. Create Routes
12. Write integration tests
13. Create API
14. Review code and make adjustments where necessary

Architecture

My applications are heavily influenced by the MVC architecture which has served me well in my short Software Development career. The MVC architecture is well suited for Node.js development because it promotes modular code and separation of concerns.

Below is my typical directory structure.

```
/
 api/
 bin/
 collections/
 config/
 controllers/
 env/
 lib/
 models/
 public/
 routes/
 test/
 views/
 .gitignore
 .jshintrc
 app.js
 package.json
 README.md
```

I will now zoom-in and look at each first level directory/file in the architecture, describing its role.

Documentation (./README.md)

The `README.md` is one of the most important files in my projects. I like to practice Readme Driven Development because I find it very effective for clarity and project direction.

My `README.md` file usually contains the following information:

1. Project title and description
2. Software requirement
3. Dependencies
4. Getting started instructions
5. Required configuration
6. Tasks commands
7. Style Guide
8. Application Architecture
9. Routes/API End Points
10. License information

The API and routes are the entry points to the application, they give us a rough idea about its size. Below is a simple example of how I document my routes/api:

```
/**
 *  Routes
**/

GET  /items     - get a collection of items
GET  /items/:id - get one item
POST /items     - save an item
```

./models

Now that we are armed with some valuable information about all the requests our beloved users will be making to our application, let us create `models` to store and retrieve information. A `model` represents a set of information describing a particular database entity. In a software application, a model usually represents a single record in a database table.

Please Note: All examples in this post will use Bookshelf for database interaction.

./models/mymodel.js

```
// get config
var config = require('../config');

// connect to the database
var Bookshelf = require('../lib/dbconnect')(config);

// define model
var myModel = Bookshelf.Model.extend({
  tableName: 'items'
});

// export collection module
module.exports = myModel;
```

./collections

We also need to be able to handle batch requests for a particular set of data. This is the job for `collections`. If you are familiar with backbone.js then the concept of `collections` should be easy to follow. Collections represent a group of the same `model`, they come with special methods to easily query and fetch data. A `collection` usually represents a complete database table.

./collections/mycollection.js

```
//require the model for this collection
var myModel = require('../models/mymodel');

// define collection
var myCollection = Bookshelf.Collection.extend({
  model: myModel
});

// export collection module
module.exports = myCollection;
```

./controllers

Controllers, like in any other typical MVC set up, are responsible for the business logic of the application. Our controllers process data passed by routes and then query the database using `models` and `collections`.

./controllers/items.js

```
var myModel = require('../models/mymodel');
var myCollection = require('../collections/mycollection');

module.exports = {

  // GET /items/:id
  getItem: function(req, res, next) {
    var id = req.params.id;

    myModel.forge({id: id})
    .fetch()
    .then(function (model) {
      res.json(model.toJSON());
    })
    .otherwise(function (error) {
      res.status(500).json({msg: error.message});
    });
  },


  // GET /items
  getItems: function(req, res, next) {
    var id = req.params.id;

    myCollection.forge()
    .fetch()
    .then(function (collection) {
      res.json(collection.toJSON());
    })
    .otherwise(function (error) {
      res.status(500).json({msg: error.message});
    });
  },


  // POST /items
  // (Don't forget to validate and sanitize all user input)
  saveItem: function(req, res, next) {
    myModel.forge(req.body)
    .save()
    .then(function (model) {
      res.json(model.toJSON());
    })
    .otherwise(function (error) {
      res.status(500).json({msg: error.message});
    });
  }
};
```

./routes

Routes are responsible for handling traffic and connecting it to the appropriate controllers, for example, if a user requests for one item, the job of a router would be to direct the request to the `getItem` method of the `itemsController`.

./routes/items.js

```
var express = require('express');
var itemsController = require('../controllers/items');

module.exports = function () {
  var router = express.Router();

  router.get('/items', itemsController.getItems);
  router.get('/items/:id', itemsController.getItem);
  router.post('/items', itemsController.saveItem);

  return router;
};
```

./config

Earlier, we required the `config` module when we were creating our model. The sole purpose of the `config` directory is to check the environment mode and load the appropriate config file from the `env` directory. The `config` directory contains a single file, `index.js`.

```
module.exports = (function (env) {
  var config = {};

  switch (env) {
    case 'production':
      config = require('../env/production');
      break;

    case 'development':
      config = require('../env/development');
      break;

    case 'testing':
      config = require('../env/testing');
      break;

    case 'staging':
      config = require('../env/staging');
      break;

    default:
      console.error('NODE_ENV environment variable not set');
      process.exit(1);
  }

  return config;
})(process.env.NODE_ENV);
```

./env

The `env` directory contains files representing different environment modes: `development.js`, `production.js`, `test.js`, and `staging.js`.

Here is an example of one file:

```
module.exports = {
  pg: {
    host: '127.0.0.1',
    database: 'test',
    user: 'test',
    password: 'test',
    charset: 'utf8'
  },
  mongodb: {
    url: 'mongodb://localhost:27017/test'
  },
  sessionSecret: 'ninja_cat'
};
```

Heads up: Do not include any sensitive data in your config file, instead use environment variables.

./api

The `api` directory contains the application API files. I create `api` files exactly the same way as I do `controllers`, the only difference being that controllers sometimes load a view template.

./lib

The `lib` directory is very common in most Node modules. If your application uses any special algorithms or helpers then the lib directory is the perfect destination for them. In most cases a controller will require a `lib` file to perform a certain special task.

./bin

The `bin` directory contains all my command-line scripts. Below is an example:

```
#!/usr/bin/env node
console.log('I am an executable file');
```

./public

The `public` directory contains asset files like images, css, front-end JavaScript, fonts, e.t.c.

./views

All my application templates go into the views `directory`.

./test

The `test` directory contain all test cases.

./.gitignore

The `.gitignore` is used to to inform GIT about which files to ignore and not track. Some of the file and directories that I don't track with git include `node_modules`, PSDs, and temp files.

```
*.zip
*.psd
*~
node_modules/
bower_components/
build/
temp/
```

./.jshintrc

The `.jshintrc` is a config file for jshint, it sets the rules for JavaScript quality validation.

```
{
  "curly": false,
  "eqeqeq": true,
  "immed": true,
  "latedef": false,
  "newcap": true,
  "noarg": true,
  "sub": true,
  "undef": true,
  "boss": true,
  "eqnull": true,
  "node": true,
  "browser": true,
  "globals": {
    "jQuery": true,
    "define": true,
    "requirejs":true,
    "require": true,
    "describe": true,
    "it": true,
    "beforeEach": true,
    "before": true
  }
}
```

./package.json

`package.json` is a standard npm file for listing app dependencies and metadata. While Grunt and Gulf serve their purpose, in most cases I find `package.json` scripts adequate for doing command-line tasks.

Example:

```
{
    ...

    "scripts": {
      "start": "node app.js",
      "dev": "nodemon app",
      "jshint": "jshint api collections config controllers env lib models public/javascripts routes test ap
p.js",
      "test": "npm run jshint && mocha test",
      "precommit": "npm test",
      "prepush": "npm shrinkwrap && npm test",
      "postmerge": "npm install"
    }

    ...
}
```

Some of my most used modules

- Express - App frameworks
- Bookshelf - Postgres and MySQL ORM
- lodash - utility library
- passport - authentication
- mongoose - MongoDB ODM
- when.js - promises library
- moment - parsing, validating, manipulating, and formatting dates

Conclusion

I have tried to cram in as much information as I can in one post but it is by no means exhaustive. Building Node.js application is constantly evolving and it is important to always be on the lookout for new and better solutions from the community.

I have more posts lined up for this month - subscribe to my RSS feed to stay up-to-date.

PREV NEXT