# Sails Tutorial - Chapter 2

We have the application up and running. However, it currently does not have any routes or pages to visit! In this section, we will add a few routes to display content on the page.

Sails is a framework (not a library), and due to this, we need to follow the folder structure and naming conventions that Sails suggests.

In order to add routes, we first need to create a folder named `config`. Then, inside this folder, we need to create a file named `routes.js`. This file will store all the routes that Sails is going to serve.

```
$ mkdir config
$ touch config/routes.js
```

Open the `config/routes.js` file and write the following code:

```
module.exports.routes = {};
```

This code `exports` the `routes` object. When creating a specific file in the `config` folder, you're often exporting an object named after the file, such as the `routes` object from the `routes.js` file.

As this `exports` an object, define the routes in a key-value pair. The key should contain the HTTP verb and path, while the value should have the controller's name and the action that will be executed when the route is called or visited. Here's an example of how to define a route:

```
module.exports.routes = {
  'get /': 'PagesController.index',
};
```

When GET / is visited, Sails will run `PagesController`'s `index` action. Make sure you put a whitespace between HTTP verb and path. With this code, we have defined a route. Next, we need to define the corresponding controller action.

Controllers in Sails should be located in the `api/controllers` folder. Go ahead and create `api` folder first and then inside the `api` folder create `controllers` folder.

```
$ mkdir api
$ mkdir api/controllers
```

Now, we can create the controller we want in this `api/controllers` folder and Sails will auto execute it for us. Go ahead and create a file named `PagesController.js` inside the `api/controllers` folder:

```
$ touch api/controllers/PagesController.js
```

The file's name must match the controller name specified in the `routes` object. Since we mentioned `PagesController.index`, create a file named `PagesController.js`.

Open this file and write the following code:

```
module.exports = {};
```

You only need to `exports` the object from the controller's file.

Next, within this controller file, we need to define `index` action. An action is a fancy name for the method or function.

```
module.exports = {
  index: function () {},
};
```

Sails framework depends on Express, and because of that, the action should have the same signature as the function, allowing access to `req` and `res` parameters.

```
module.exports = {
  index: function (req, res) {},
};
```

We can easily send the response as *hello, world* using `res.send()` method.

```
module.exports = {
  index: function (req, res) {
    res.send('hello, world');
  },
};
```

Run the server if not running and execute the dev script.

```
$ npm run dev
```

Open the http://localhost:3000 and you should see *hello, world* text in the browser. If that is the case, you just wrote the hello, world application in/using Sails framework. Congratulations!

---

Let's create a few more routes to practice. We'll add two more routes: `/about` and `/contact`. Add these routes to the `config/routes.js` file:

```
module.exports.routes = {
  'get /': 'PagesController.index',
  'get /about': 'PagesController.about',
  'get /contact': 'PagesController.contact',
};
```

For these routes, we need to add `about()` and `contact()` methods in `PagesController`.

```
module.exports = {
  index: function (req, res) {
    res.send('hello, world');
  },
```

```
  about: function (req, res) {
    res.send('about');
  },
  contact: function (req, res) {
    res.send('contact');
  },
};
```

Code is ready! Check and confirm the `/about` and `/contact` routes in browser and you should see `about` and `contact` responses respectively.

---

In real-world applications, responses typically include HTML pages. Sails supports EJS views by default. We just need to follow there convention and we can have the views!

Go ahead and create `views` folder.

```
$ mkdir views
```

Within this folder, create `.ejs` files, and refer to them in the controller's actions using relative paths. For instance, if the file `index.ejs` is in the `views` folder, reference it as `index` to render, and Sails will resolve the path automatically.

For the purpose of the simplicity, Let's create a folder with name `pages` in `views` folder and this `pages` folder will contains all the pages.

```
$ mkdir views/pages
```

Let's create `index.ejs`, `about.ejs`, and `contact.ejs` files for `/`, `/about`, and `/contact` routes respectively.

```
$ touch views/pages/index.ejs
$ touch views/pages/about.ejs
$ touch views/pages/contact.ejs
```

For now, we're just going to write a single header in each of these created files. So, in `index.ejs` we can have the following code.

```
<h1>Home</h1>
```

In `about.ejs` file, we can have the following code.

```
<h1>About</h1>
```

Finally, in `contact.ejs` file, we can have the following code.

```
<h1>Contact</h1>
```

In `PagesController.js` file, we need to update the response from sending simple text words to render these views. These created views are within `pages` folder. So, we need to write `pages/index`, `pages/about`, and `pages/contact` as relative path to `views` folder.

```
module.exports = {
  index: function (req, res) {
    res.render('pages/index');
  },
  about: function (req, res) {
    res.render('pages/about');
  },
  contact: function (req, res) {
    res.render('pages/contact');
  },
};
```

Restart the server if not running and then visit the `/`, `/about`, and `/contact` routes in the browser. You should see the headers we just wrote in views. If that is the case, we just created Sails.js application with some routes and views. Also, we now know C (controller) and V (view) of MVC.