FREE eBook

LEARNING sails.js

Free unaffiliated eBook created from
Stack Overflow contributors.

#sails.js

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sails-js

It is an unofficial and free sails.js ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official sails.js.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with sails.js

Remarks

sails.js is an MVC (Model View Controller) web framework for node.js that emulates familiar MVC frameworks like Ruby on Rails. sails.js is based on Express and provides websocket support via socket.io.

sails.js provides a set of conventions and default configurations to quickly get a new website project started. It is highly configurable and allows you to easily override the default conventions.

sails.js comes with an ORM called Waterline which abstracts data access. Waterline allows you to use various datastores such as MySQL, PostgreSQL, MongoDB, Redis, etc. and have a clear API for working with your model data.

Versions

Version	Release notes	Changelog	Release Date
0.12.13	Release notes		2017-03-06
0.12.12	Release notes	Changelog	2017-03-03
0.12.11	Release notes	Changelog	2016-11-24
0.12.10	Release notes	Changelog	2016-11-17
0.12.9	Release notes	Changelog	2016-11-02
0.12.8	Release notes	Changelog	2016-10-22
0.12.7	Release notes	Changelog	2016-10-06
0.12.6	Release notes	Changelog	2016-09-28
0.12.5	Release notes	Changelog	2016-09-28
0.12.4	Release notes	Changelog	2016-08-01
0.12.3	Release notes	Changelog	2016-04-04
0.12.2	Release notes	Changelog	2016-04-02
0.12.1	Release notes	Changelog	2016-02-15
0.12.0	Release notes	Changelog	2016-02-06
0.11.5	Release notes	Changelog	2016-02-05

Version	Release notes	Changelog	Release Date
0.11.4	Release notes	Changelog	2016-01-06
0.11.3	Release notes	Changelog	2015-11-23
0.11.2	Release notes	Changelog	2015-09-23
0.11.0	Release notes	Changelog	2015-02-11
0.10.5	Release notes	Changelog	2014-08-30
0.10.4	Release notes		2014-08-13
0.10.3	Release notes		2014-08-07
0.10.2	Release notes		2014-08-06
0.10.1	Release notes		2014-08-02

Releases prior to `0.10.1` omitted from list. See earlier releases

Examples

Installation

Prerequisites

- nodejs

To install the latest stable release of sails with the command-line tool issue following command:

```
$ sudo npm install sails -g
```

Depending on your OS you might not need to use `sudo`.

Creating a new project

Once you have Sails installed, just type

```
$ sails new <project_name>
```

This will create a skeleton Sails project in a new folder called <project_name>.

You can also create a new project in an empty folder by typing

```
$ sails new
```

Launch app

Once your project has been created, you can launch the app by typing

```
$ sails lift
```

By default, you can access the app in the browser on port 1337. The URL with the port is shown in the terminal.

Another way to start the Sails app is with the `node` command:

```
$ node app.js
```

However, you lose some development features of the `lift` command like auto-reloading of the app when assets and view files are modified.

For development you can also use:

```
$ sails console
```

This allows you to execute command directly in command line. It's very useful for debugging Models.

Hello world

This example shows how to develop our first application step by step, assuming you already have Sails installed and a project created.

1. Create an empty controller file by typing

```
$ sails generate controller hello
```

2. Find the new controller file at `api/controllers/HelloControllers.js` and add the `hello` method to it.

```
module.exports = {

  hello : function (req, res) {
    var myName = 'Luis';
    return res.view('hello' , {name : myName});
    }
}
```

3. Create a new view file under the folder `views` named `hello.ejs` with the following HTML:

```
<html>
    <head></head>
    <body>
        <p>Hello {{}}.</p>
```

```
    </body>
</html>
```

4. Define a route in `config/routes.js` that calls the `hello` method in the `HelloController` controller.

```
'GET /' : 'HelloController.hello',
```

Now we have implemented all the code needed for this example. Let's try it:

1. Start the server

```
$ sails lift
```

2. Open the browser and type `http://localhost:1337`. If it's not coming up, check the URL in the `sails lift` output. The port may be different.

3. You should see the following output:

   Hello Luis

Generating sails project without frontend

If there is no need for frontend in your next project, you can run sails new with additional flag --no-frontend.

```
sails new NameOfProject --no-frontend
```

This will generate everything needed for backend and will omit view, assets and grunt files.

More about command line and sails-new: http://sailsjs.org/documentation/reference/command-line-interface/sails-new

Read Getting started with sails.js online: https://riptutorial.com/sails-js/topic/1205/getting-started-with-sails-js

Chapter 2: Blueprint API

Remarks

How does Blueprint API works?

When sails initially starts using `sails lift`, sails looks to see if you have any controller defined. In our example, we have one controller, the User controller. Sails then provides access to blueprint actions for this user controller as if we built them in the controller ourselves. Sails also automatically creates blueprint routes at the time of lifting the server. So even if no routes is defined in `/config/routes.js` and no action is defined in `/api/controllers/UserController.js` explicitly, after lifting the server all these routes and actions are available to use.

Examples

Blueprint Routes

When you run `sails lift` with blueprints enabled, the framework inspects your controllers, models, and configuration in order to bind certain routes automatically. These implicit blueprint routes allow your app to respond to certain requests without you having to bind those routes manually in your `config/routes.js` file. By default, the blueprint routes point to their corresponding blueprint *actions*, any of which can be overridden with custom code.

There are three types of blueprint routes in Sails:

- **RESTful routes**, where the path is always `/:model/:id?`. When the `User` model and controller is defined, blueprint binds RESTful routes implicitly in following way -

```
'GET /user/:id?': {
  controller: 'User',
  action: 'find'
},
'POST /user': {
  controller: 'User',
  action: 'create'
},
'PUT /user/:id?': {
  controller: 'User',
  action: 'update'
},
'DELETE /user/:id?': {
  controller: 'User',
  action: 'destroy'
}
```

  These routes use the HTTP verb to determine the action to take even if the route is same. So, a `POST` request to `/user` will create a new user, a `PUT` request to `/user/123` will update the user with primary key 123 and a `DELETE` request to `/user/123` will delete the user whose primary key is 123. In a production environment, RESTful routes should generally be

protected by policies to avoid unauthorized access.

- **Shortcut routes**, where the action to take is encoded in the path. For our User model and controller Sails binds following four shortcut routes implicitly.

```
'GET /user/find/:id?': {
  controller: 'User',
  action: 'find'
},
'GET /user/create/:id?': {
  controller: 'User',
  action: 'create'
},
'GET /user/update/:id?': {
  controller: 'User',
  action: 'update'
},
'GET /user/destroy/:id?': {
  controller: 'User',
  action: 'destroy'
}
```

  As example, the `/user/create?name=joe` shortcut creates a new user, while `/user/update/1?name=mike` updates the name field of user #1. Note that these routes only respond to `GET` requests. Shortcut routes are very handy for development, but generally should be disabled in a production environment. It's not designed to be used in production.

- **Action routes**, which automatically create routes for your custom controller actions. For example, let `query` be a custom action defined in User controller. Then following routes would be implicitly available to sails -

```
'GET /user/query/:id?': {
  controller: 'User',
  action: 'query'
},
'POST /user/query/:id?': {
  controller: 'User',
  action: 'query'
},
'PUT /user/query/:id?': {
  controller: 'User',
  action: 'query'
},
'DELETE /user/query/:id?': {
  controller: 'User',
  action: 'query'
}
```

  If request is made in `/user/query/:id?` route then independent on the HTTP verb the action would be same. Unlike RESTful and shortcut routes, action routes do *not* require that a controller has a corresponding model file. Which means, if you define a controller in `/api/controllers/FooController.js` file but no model in `/api/models/Foo.js` file, there would be no RESTful or shortcut route with `/foo` but there will still be action routes available to use.

Order of routes matching

When a request comes, sails will first match the route against explicitly defined routes. If it matches then no further matching is done and corresponding action is executed. But if it doesn't match then the route is matched firstly against blueprint action routes, if doesn't match then against rest routes and if it doesn't match either then shortcut routes. So if your `/config/routes.js` file has some entry like the following-

```
'/user/query/:id?': {
  controller: 'User',
  action: 'find'
}
```

Then you can't expect `query` action routes to work. Because the same route as `query` action route would be matched against the explicitly define routes and `find` action of User controller would be executed.

Blueprint Actions

Blueprint actions (not to be confused with blueprint action *routes*) are generic actions designed to work with any of your controllers that have a model of the same name (e.g. `ParrotController` would need a `Parrot` model). Think of them as the default behavior for your application. For instance, if you have a `User.js` model and an empty `UserController.js` controller, `find`, `create`, `update`, `destroy`, `populate`, `add` and `remove` actions exist implicitly, without you having to write them.

By default, the blueprint RESTful routes and shortcut routes are bound to their corresponding blueprint actions. However, any blueprint action can be overridden for a particular controller by creating a custom action in that controller file (e.g. `ParrotController.find`). Alternatively, you can override the blueprint action *everywhere in your app* by creating your own custom blueprint action.

Sails ships with the following blueprint actions:

- find
- findOne
- create
- update
- destroy
- populate
- add
- remove

Disabling Blueprint Routes

- **Global basis**: Blueprint API configuration is defined in `/config/blueprint.js` file. You can enable or disable all three types of blueprint routes for all controllers from there. As example, if you want to disable blueprint shortcut routes for all of your controllers but want to keep both action and rest routes enabled, then your `/config/blueprint.js` should be like this -

```
module.exports.blueprints = {
  action: true,
  rest: true,
  shortcut: false
}
```

- **On per-controller basis**: You may also override any of the settings from `/config/blueprints.js` on a per-controller basis by defining a '_config' key in your controller defintion, and assigning it a configuration object with overrides for the settings in this file. As example if you want to have shortcut routes enabled only for your user controller but not for any more controllers then with the above blueprint configuration you have to have following key value pair in user controller.

```
module.exports = {
  _config: {
    actions: true,
    shortcuts: true,
    rest: true
  }
}
```

Read Blueprint API online: https://riptutorial.com/sails-js/topic/3749/blueprint-api

Chapter 3: Configuring mysql with sails.js

Examples

How to configure mysql database connection in sails.js

To do this first locate config folder in your root. Then open `connections.js`

Locate

```
// someMysqlServer: {
//   adapter: 'sails-mysql',
//   host: 'YOUR_MYSQL_SERVER_HOSTNAME_OR_IP_ADDRESS',
//   user: 'YOUR_MYSQL_USER', //optional
//   password: 'YOUR_MYSQL_PASSWORD', //optional
//   database: 'YOUR_MYSQL_DB' //optional
// },
```

Uncomment these lines.

Give suitable name for the connector like this someMysqlServer to mysql_connection or any name as your wish

```
mysql_connection: {
  adapter: 'sails-mysql',
  host: '127.0.0.1', // can user localhost or mysql connection either
  user: 'root', // your mysql username
  password: 'xxxxxxxxx', // your mysql password
  database: 'your database name here' // database name
},
```

Save file

Go to your root folder and run following command :

```
$ npm install sails-mysql --save
```

Note: by running above command we are installing mysql driver package for sails.js.

Your are done.

Now you can use mysql_connection as connection name in your model config. When you lift your app using following command:

```
$ sails lift
```

you model schema will automatically get updated in MySQL database

Read Configuring mysql with sails.js online: https://riptutorial.com/sails-js/topic/5317/configuring-

mysql-with-sails-js

Chapter 4: Controllers

Remarks

Controllers (the **C** in **MVC**) are the principal objects in your Sails application that are responsible for responding to requests from a web browser, mobile application or any other system capable of communicating with a server. They often act as a middleman between your models and views. For many applications, the controllers will contain the bulk of your project's business logic.

Examples

ES2015 Syntax

```
'use strict';

// This is an example of a /api/controllers/HomeController.js
module.exports = {
  // This is the index action and the route is mapped via /config/routes.js
  index(req, res) {
    // Return a view without view model data
    // This typically will return the view defined at /views/home/index.<view engine
extension>
    return res.view();
  },
  foo(req, res) {
    // Return the 'foo' view with a view model that has a `bar` variable set to the query
string variable `foobar`
    return res.view({
      bar: req.param('foobar'),
    });
  },
};
```

Using ES2015 generators with co.js

```
'use strict';

const co = require('co');

module.exports = {
  // This is the index action and the route is mapped via /config/routes.js
  index(req, res) {
    co(function* index() {
      // Return a view without view model data
      // This typically will return the view defined at /views/home/index.<view engine
extension>
      return res.view();
    }).catch(res.negotiate); // Catch any thrown errors and pass the error to the `negotiate`
policy.
  },
  foo(req, res) {
    co(function* foo() {
```

```
      // Get an array of `FooBar` items from the database
      const items = yield FooBar.find();

      // Return the 'foo' view with a view model containing the array of `FooBar` items
      return res.view({
        items,
      });
    }).catch(res.negotiate); // Catch any thrown errors and pass the error to the `negotiate`
policy.
  },
};
```

Read Controllers online: https://riptutorial.com/sails-js/topic/3521/controllers

Chapter 5: JSON web token authentication with Sails

Examples

Configuration

Step one

We need to create a service called *jwToken*. Go to `api/services` directory and create `jwToken.js`.

```
'use strict';

const jwt = require('jsonwebtoken'),
      tokenSecret = "secretissecret";

module.exports = {
  // Generates a token from supplied payload
  issue(payload) {
    return jwt.sign(
      payload,
      tokenSecret, // Token Secret that we sign it with
      {
        expiresIn: "30 days" // Token Expire time
      });
  },

  // Verifies token on a request
  verify(token, callback) {
    return jwt.verify(
      token, // The token to be verified
      tokenSecret, // Same token we used to sign
      {}, // No Option, for more see https://github.com/auth0/node-
jsonwebtoken#jwtverifytoken-secretorpublickey-options-callback
      callback //Pass errors or decoded token to callback
    );
  }
};
```

Step two

Encrypt our password using `bcrypt`. Go to `api/models/User.js`.

```
'use strict';
const bcrypt = require('bcrypt');

module.exports = {

    attributes: {
```

```
        // your code...
    },

    // Here we encrypt password before creating a User
    beforeCreate(values, next) {
        bcrypt.genSalt(10, (err, salt) => {
            if (err) {
                sails.log.error(err);
                return next();
            }

            bcrypt.hash(values.password, salt, (err, hash) => {
                if (err) {
                    sails.log.error(err);
                    return next();
                }
                values.encryptedPassword = hash; // Here is our encrypted password
                return next();
            });
        });
    },

    comparePassword(password, encryptedPassword) {

        return new Promise(function(resolve, reject) {
            bcrypt.compare(password, encryptedPassword, (err, match) => {
                if (err) {
                    sails.log.error(err);
                    return reject("Something went wrong!");
                }
                if (match) return resolve();
                else return reject("Mismatch passwords");
            });
        });
    }
};
```

Step three

Create *isAuthorized* policy to check if a user has valid token in the request header. Go to `api/policies` and create `isAuthorized.js`.

```
'use strict';

module.exports = (req, res, next) => {
  let token;


  if (req.headers && req.headers.token) {
    token = req.headers.token;
    if (token.length <= 0) return res.json(401, {err: 'Format is Authorization: Bearer
[token]'});

  } else if (req.param('token')) {
    token = req.param('token');
    // We delete the token from param to not mess with blueprints
    delete req.query.token;
```

```
  } else {
    return res.json(401, {err: 'No Authorization header was found'});
  }

  jwToken.verify(token, function (err, token) {
    if (err) return res.json(401, {err: 'Invalid Token!'});
    req.token = token; // This is the decrypted token or the payload you provided
    next();
  });
};
```

Step four

We use config/policies.js to protect our controllers

```
module.exports.policies = {

  '*': ['isAuthorized'], // Everything resctricted here
  'UserController': { // Name of your controller
    'create': true // We dont need authorization here, allowing public access
  }
};
```

Step five

Let's test our implementation. Go to `api/controllers` and create `UserController.js`

```
'use strict';

module.exports = {
    create(req, res) {
        const data = req.body;
        if (data.password !== data.confirmPassword) return res.badRequest("Password not the
same");

        User.create({
                email: data.email,
                password: data.password,
                name: data.name
                    //etc...
            })
            .then((user) => {
                res.send({ token: jwToken.issue({ id: user.id }) }); // payload is { id:
user.id}
            })
            .catch((err) => {
                sails.log.error(err);
                return res.serverError("Something went wrong");
            });
    },

    login(req, res) {
        const data = req.body;
```

```
        if (!data.email || !data.password) return res.badRequest('Email and password
required');

        User.findOne({ email: email })
            .then((user) => {
                if (!user) return res.notFound();

                User.comparePassword(password, user.password)
                    .then(() => {
                        return res.send({ token: jwToken.issue({ id: user.id }) })
                    })
                    .catch((err) => {
                        return res.forbidden();
                    });
            })
            .catch((err) => {
                sails.log.error(err);
                return res.serverError();
            });
    }
};
```

Installation

We need two dependencies:

1. **bcrypt** for encryption `npm install bcrypt --save`
2. **JSON Web token** `npm install jsonwebtoken --save`

Read JSON web token authentication with Sails online: https://riptutorial.com/sails-js/topic/7050/json-web-token-authentication-with-sails

Chapter 6: Models

Remarks

Sails comes installed with a powerful ORM/ODM called Waterline, a datastore-agnostic tool that dramatically simplifies interaction with one or more databases. It provides an abstraction layer on top of the underlying database, allowing you to easily query and manipulate your data without writing vendor-specific integration code.

Examples

Basic Model

This example shows how to define a simple model in Sails.js

You can generate an empty model file by typing

```
sails generate model car
```

You'll find the new file `Car.js` in `api/models/`.

Next, you fill in some details.

```
modules.exports = {

  tableName : 'cars',
  connection : 'mongodb',

  attributes : {

    id : {
      type : 'integer',
      unique : true,
      primaryKey : true,
      autoIncrement : true
    },

    brand : {
      type : 'string',
      size : 25
    },

    description : {
      type: 'text',
      defaultsTo : ''
    },

    price : {
      type : 'float',
      required : true
    },
```

```
    seats : {
      type : 'integer'
    },

    sell_date : {
      type : 'datetime'
    },

    has_cooler : {
      type : 'boolean',
      columnName : 'cooler'
    },

    chassis_number : {
      unique : true,
      type : 'string'
    },

    color : {
      type : 'string',
      enum: ['white', 'red', 'black']
    }

  }

};
```

The example above uses nearly every possible model option, which are explained below.

1. tableName

This parameter defines the name of the table that will be created in the database. If not defined, the model name will be used (`car` in this example).

2. connection

This particular defines the database connection used for the model. The details of that connection are defined under the `mongodb` key inside `config/connections.js`. Here's the format of a connection:

```
mongodb : {

  // The driver that connect our models with the database
  adapter : '<adapter>',

  // The database parameters
  user : '<username>',
  port : <port>,
  host : '<host>',
  database : '<database>'

}
```

3. attributes

Each attribute references a column in the database table for the model. In this example, nine columns will be created. Each column can be configured with one or more of the following keys:

- **type** : The data type of the column. This page lists all the available types.
- **unique** : If true, an error will occur if you try to create an object that has the same value for this column as one already in the database.
- **primaryKey** : If true, the column will work as primary key.
- **autoIncrement** : A sequence will be associated to the column with an auto incrementable number starting at 0.
- **size** : The maximum length of the column.
- **required** : If true, it can't be null.
- **columnName** : This configures the column in the database, which defaults to the attribute name.
- **enum** : We can set an array of possible options for an attribute.

Read Models online: https://riptutorial.com/sails-js/topic/4646/models

Chapter 7: MongoDB Adapter for Sails

Examples

Configuration

You can configure the database settings in `config/connections.js`.

Example:

```
someMongoDb: {
  adapter: 'sails-mongo',
  host: 'localhost', // defaults to `localhost` if omitted
  port: 27017, // defaults to 27017 if omitted
  user: 'username_here', // or omit if not relevant
  password: 'password_here', // or omit if not relevant
  database: 'database_name_here' // or omit if not relevant
}
```

Alternatively, you can specify your Mongo configuration as a URL

```
someMongoDb: {
  adapter: 'sails-mongo',
  url: mongodb://username:password@hostname:port/database
}
```

Installation

Install from NPM.

```
npm install sails-mongo --save
```

Read MongoDB Adapter for Sails online: https://riptutorial.com/sails-js/topic/7047/mongodb-adapter-for-sails

Chapter 8: PostgreSQL Database Adapter for Sails

Examples

Install

You can install the postgreSQL adapter via NPM.

```
npm install sails-postgresql
```

Configuration

You can configure the database settings in `config/connections.js`.

Here's an example:

```
postgresql: {
  database: 'databaseName',
  host: 'localhost',
  user: 'root',
  password: '',
  port: 5432,
  poolSize: 10,
  ssl: false
};
```

Alternatively, you can supply the connection information in URL format:

```
postgresql: {
  url: 'postgres://username:password@hostname:port/database',
  ssl: false
};
```

Read PostgreSQL Database Adapter for Sails online: https://riptutorial.com/sails-js/topic/4676/postgresql-database-adapter-for-sails

Chapter 9: Routing

Remarks

Routes are rules that tell Sails what to do when faced with an incoming request.

Routes are defined in `config/routes.js`. The order of the routes is significant, as routes are matched top down. This means if you have a specific route that also could be matched by a wildcard route, the specific route should be defined above the wildcard route.

When a request enters your application sails.js grabs all the parameters that came with it and makes them available for you as `params` on the request object.

Properties in the route target object will be passed through to the route handler in the req.options object. The following are reserved properties that can affect the behavior of the route handler:

Property	Applicable Target Types	Data Type	Details
skipAssets	all	Boolean	Set to `true` if you *don't* want the route to match URLs with dots in them (e.g. myImage.jpg). This will keep your routes with wildcard notation from matching URLs of static assets. Useful when creating URL slugs.
skipRegex	all	Regexp	If skipping every URL containing a dot is too permissive, or you need a route's handler to be skipped based on different criteria entirely, you can use `skipRegex`. This option allows you to specify a regular expression or array of regular expressions to match the request URL against; if any of the matches are successful, the handler is skipped. Note that unlike the syntax for binding a handler with a regular expression, `skipRegex` expects _actual RegExp objects, not strings.
locals	controller, view, blueprint, response	Dictionary	Sets default local variables to pass to any view that is rendered while handling the request.
cors	all	Dictionary or Boolean or String	Specifies how to handle requests for this route from a different origin.

Property	Applicable Target Types	Data Type	Details
populate	blueprint	Boolean	Indicates whether the results in a "find" or "findOne" blueprint action should have associated model fields populated. Defaults to the value set in `config/blueprints.js`.
skip, **limit**, **sort**, **where**	blueprint	Dictionary	Set criteria for "find" blueprint.

Examples

Custom RESTful route definitions

```
module.exports.routes = {
    'GET /foo': 'FooController.index',
    'GET /foo/new': 'FooController.new',
    'POST /foo/create': 'FooController.create',
    'GET /foo/:id/edit': 'FooController.edit',
    'PUT /foo/:id/update': 'FooController.update',
    'GET /foo/:id': 'FooController.show',
    'DELETE /foo/:id': 'FooController.delete',
};
```

Redirect

```
module.exports.routes = {
  '/foo': '/bar',
  'GET /google': 'http://www.google.com'
};
```

Define custom variable for all views

```
module.exports.routes = {
  // This function will be executed for all http verbs on all urls
  'all /*', function (req, res, next) {
    // Expose the function `fooBar` to all views (via the locals object)
    res.locals.fooBar = function (arg1) {
      return 'foobar' + arg1;
    };
  },
};
```

Skip assets (urls with dots in them) from wildcard route

```
module.exports.routes = {
  'GET /foo/*': {
     fn: function(req, res) {
```

```
      res.send("FOO!");
    },
    skipAssets: true
  },
};
```

Routes with RegEx

```
module.exports.routes = {
    // sends matching regex (few patterns) as 'page' param
    'r|/foo/([0-9]+)|page': 'FooController.get',
    'r|/foo/(.*)|page': 'FooController.get',
    'r|/foo/(\\w+)|page': 'FooController.get'
};
```

URL slugs

```
module.exports.routes = {
  'GET /blog/:year/:month/:day/:posttitle/': 'BlogController.showPost',
  'GET /blog/:year/:month/:day/': 'BlogController.showDayArchive',
  'GET /blog/:year/:month/': 'BlogController.showMonthArchive',
  'GET /blog/:year/': 'BlogController.showYearArchive',
};
```

The parameters passed in the URL can then be accessed in the corresponding controller actions using `req.param('year')`, `req.param('month')` etc.

For example, a `GET` request to `/blog/2016/08/` triggers the `BlogController.showMonthArchive` controller action, with `req.param('year')` having the value `2016` and `req.param('month')` having the value `08`.

Read Routing online: https://riptutorial.com/sails-js/topic/4577/routing

Credits

S. No	Chapters	Contributors
1	Getting started with sails.js	Bonanza, Community, hlozancic, Jim Geurts, Lost in OWL, Luis González, mouche, Sebastialonso, Viktor, Yann Bertrand
2	Blueprint API	taufique
3	Configuring mysql with sails.js	vijeeshin, Yann Bertrand
4	Controllers	Jim Geurts
5	JSON web token authentication with Sails	SkyQ, Viktor
6	Models	Luis González, mouche, Viktor
7	MongoDB Adapter for Sails	SkyQ
8	PostgreSQL Database Adapter for Sails	Luis González, mouche
9	Routing	Bonanza, Jim Geurts, Viktor