Learn Git and GitHub without any code!

Using the Hello World guide, you'll start a branch, write comments, and open a pull request.

Read the guide

SEI-CC / **SEIR-11-08-21**

Code | Issues | Pull requests 1 | Projects | Wiki | Insights

main

SEIR-11-08-21 / work / w04 / d5 / **01-02-express-routers-controllers.md**

cogilvy Add W4D5 — History

0 contributors

Raw | Blame

`Executable File  692 lines (455 sloc)  20.2 KB`

Express Routers & Controllers

Learning Objectives

Students Will Be Able To:
Use the Express Generator to Scaffold a Skeleton App
Implement Best Practice Routing
Organize App Logic Into Controller Modules

Road Map

- Setup
- Express Generator
- MVC Code Organization
- Best Practice Routing
- To-Do Refactor
- Controller Modules
- MVC Organization Revisited
- URL/Route Parameters
- Adding "Show a To-Do" Functionality

Setup

To get ready for this lesson, simply `cd ~/code`.

Express Generator

Okay, so we've had big fun getting an Express app up and running from scratch.

We defined some basic routes and rendered a couple of views using the EJS view engine.

The first thing we'll take a look at in this lesson is a popular tool: `express-generator`.

`express-generator` creates a "skeleton" Express app that:

- Separates the HTTP server code from our web app's logic.
- Has best practice routing implemented.
- Has key middleware configured and mounted by default.
- Is configured to serve static assets from a `public` folder.
- Will configure the EJS view engine (if we specify the `-e` option)
- Has error handling configured.

Install `express-generator`

We install `express-generator` globally using NPM:

```
npm install -g express-generator
```

`express-generator` is a CLI that can be run from anywhere, that's why we install it using the global `-g` flag.

If `express-generator` Won't Install or Run

If your system won't install or run express-generator, it can be used with `npx` instead as described in a bit below.

Available Options

Let's take a look at the options available to us

```
express -h
```

```
Usage: express [options] [dir]

Options:

      --version        output the version number
  -e, --ejs            add ejs engine support
      --pug            add pug engine support
      --hbs            add handlebars engine support
  -H, --hogan          add hogan.js engine support
  -v, --view <engine>  add view <engine> support (dust|ejs|hbs|hjs|jade|pug|twi
      --no-view        use static html instead of view engine
  -c, --css <engine>   add stylesheet <engine> support (less|stylus|compass|sas
      --git            add .gitignore
  -f, --force          force on non-empty directory
  -h, --help           output usage information
```

Specify the View Engine

We'll definitely want to use the EJS view engine each time we scaffold a new Express project.

The help above identifies the following option flags to make EJS the view engine:

- `-e`
- `-ejs`
- `--view=ejs`
- `-v=ejs`

All of the above option flags do the exact same thing, so take your pick.

Scaffold Our `express-todos` App

Let's learn more about Express by building a simple To-Do app.

We scaffold an app using the `express` command:

```
express -e express-todos
```

> Note that when a name is specified, `express-generator` creates a folder with that name and creates the app within it.

If `express-generator` did not install or if the above `express` command did not work, use `npx` instead as follows:

```
npx express -e express-todos
```

We now need to `cd` into the new folder and open in our text editor:

```
cd express-todos
code .
```

Install the Node Modules

Express generator has created a `package.json` that lists the necessary Node modules, however, those modules are not installed.

Let's open a terminal session (`option + backtick`) and install them:

```
npm i
```

Folder Structure

Let's briefly review the scaffolded contents in VS Code:

```
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.js
    └── index.js
```

Renaming `app.js` to `server.js`

MERN/MEAN Stack apps often have a client-side file named `app.js` and this could get confusing having two `app.js` files, thus many developers name their main Express file `server.js`.

So let's rename it:

1. Rename `app.js` to `server.js`.
2. Inside of `bin/www`, we need to update line 7 to require `server` instead of `app`:

   ```
   var app = require('../app');
   ```

   to:

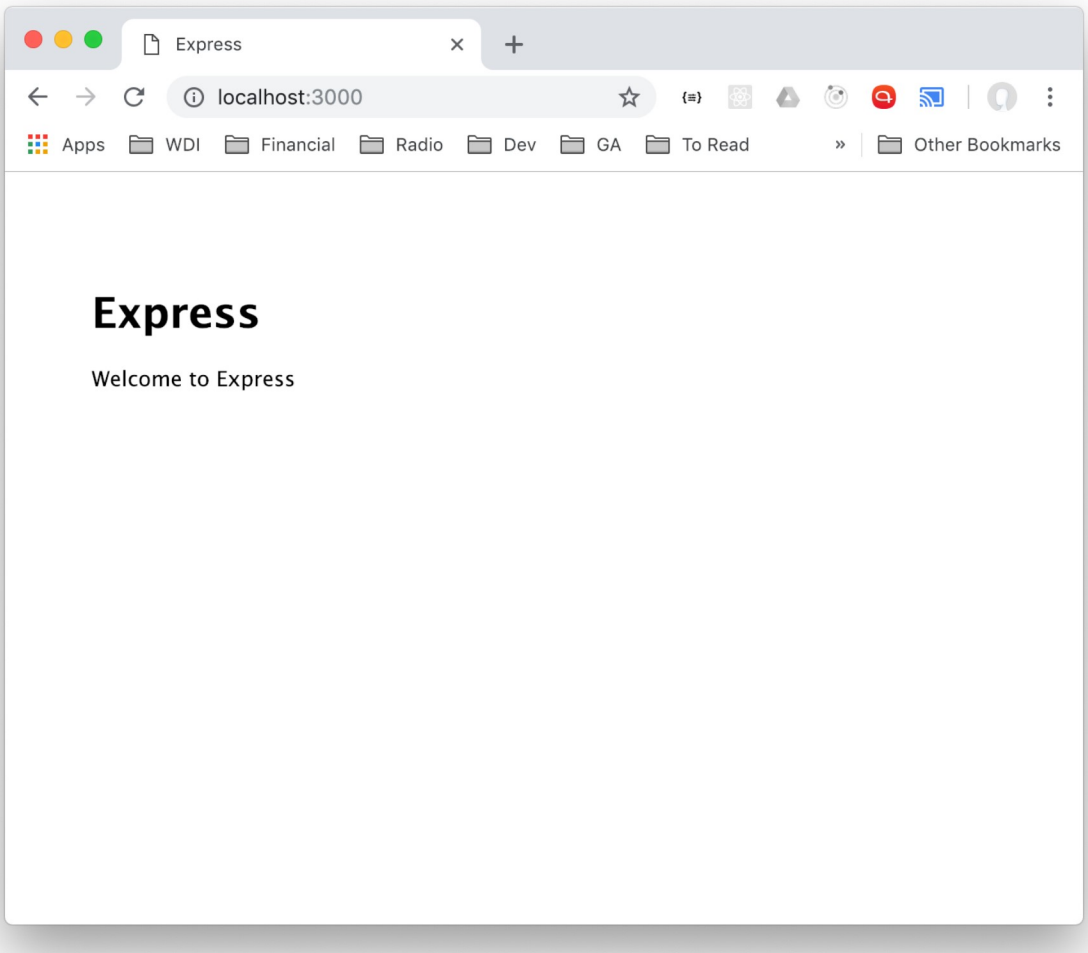
   ```
   var app = require('../server');
   ```

Starting the Application

One option to start the server is to type `npm start`. This will execute the start script specified in *package.json*. However, it doesn't restart the app when there's changes...

`nodemon` is still our best option and we can now just type `nodemon` which will use that same `start` script.

Browsing to `localhost:3000` greets us with:



MVC Code Organization

MVC (Model-View-Controller) has been a proven approach for organizing code for decades where the:

- **Model**: Is the data concern
- **View**: Is the presentation concern
- **Controller**: Is the "processing" concern that processes HTTP requests coming from the browser, CRUDs (creates/reads/updates/deletes) data using models, and ultimately responds to the browser's request by rendering views and returning the resulting HTML or by issuing a redirect.

MVC in Express

Express, as it states on its landing page, is *unopinionated*. This means we are free to structure and organize our Express apps anyway we please.

In fact, many web frameworks such as Ruby on Rails, ASP.net, Spring MVC (Java), and others implement the MVC architectural pattern.

Accordingly, most Express developers use MVC to organize their Express applications as well...

Adding a `models & controllers` Folder

Express generator has already organized the view templates into a `views` folder.

So we just need to make folders to hold our model and controller modules:

```
mkdir models controllers
```

With the `models`, `views` and `controllers` folders set up, we're ready to discuss best practice routing...

Best Practice Routing

In our `first-express` app, we used the `app.get` method to define routes and although it worked, the better practice is to:

- Use Express `router` objects to define routes for a particular purpose or dedicated to a certain data resource such as `todos`.
- Create each `router` in its own module from which it is exported.
- Inside of **server.js** `require` and mount the `router` object in the request pipeline.

> `route` objects are also middleware functions

Best Practice Routing Set Up Express Generator

As an example of using this better approach to routing, let's look at how `express-generator` sets up routing...

First, there's a `routes` folder containing two router modules:

- **index.js**: Great for defining general purpose routes, e.g., the root route.
- **users.js**: An example of a router dedicated to a *data resource*, in this case, *users*.

Note how routes are defined on those two `router` objects using `router.get()` method call just like we did previously with `app.get()`

Each `router` object has one route defined - compare those two routes, notice the *method* and the *paths*? They're the same - isn't that a problem? Nope, they're not actually the same because of the way the routers are mounted in **server.js**...

Router Objects in the Scaffolded App

The two route modules are required on lines 7 & 8 of `server.js` .

Then those routers are mounted in the middleware pipeline with the `app.use` method on lines 22 & 23:

```
app.use('/', indexRouter);
app.use('/users', usersRouter);
```

> **IMPORTANT KEY POINT:** The path specified in `app.use` is **prepended** to the paths specified in the router object forming the actual path of the defined route. Think of the above paths as a "starts with path".

Determining the Actual Path of a Route Defined in a Router Object

Let's say you have a `router` object that defines a route like this:

```
// routes/todos.js

var express = require('express');
var router = express.Router();

router.get('/', function(req, res) {...
```

and is mounted like this:

```
const todosRouter = require('./routes/todos');

// All routes defined in todosRouter will start with /todos
app.use('/todos', todosRouter);
```

❓ What is the actual path of the route?

Another example, let's say you have a `router` object that defines a route like this:

```
// routes/calendar.js
```

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

router.get('/today', function(req, res) {...
```

and is mounted like this:

```
const calendarRouter = require('./routes/calendar');

app.use('/calendar', calendarRouter);
```

❓ **What is the actual path of that route?**

To-Do Refactor

We're going to refactor the To-Do code from yesterday to follow best practices by:

- Copying over the **index.ejs** view and put the todos "database" into the `models` folder.
- Implementing best-practice routing.
- Organizing the route handling code into a controller functions (AKA controller actions).

To-Do Refactor - `index.ejs`

- Create **todos/index.ejs**:

    ```
    mkdir views/todos
    touch views/todos/index.ejs
    ```

- Add the HTML boilerplate.
- Update the title to: `<title>Express To-Do</title>`
- Here's the EJS from yesterday to copy/paste (replacing the existing `<body>` element):

    ```
    <body>
    ```

```
    <h1>Todos</h1>
    <ul>
      <% todos.forEach(function(t) { %>
        <li>
          <%= t.todo %>
            -
          <%= t.done ? 'done' : 'not done' %>
        </li>
      <% }); %>
    </ul>
  </body>
```

To-Do Refactor - `models/todo.js`

- Now let's create and copy over our model.
- Create **models/todo.js**:

```
touch models/todo.js
```

> Note that modules for models should be named singularly.

- Here's the slightly refactored code from yesterday, copy it into **models/todo.js**:

```
const todos = [
  {id: 125223, todo: 'Feed Dogs', done: true},
  {id: 127904, todo: 'Learn Express', done: false},
  {id: 139608, todo: 'Buy Milk', done: false}
];

module.exports = {
  getAll
};

function getAll() {
  return todos;
}
```

Look it over - any questions?

To-Do Refactor - Routing

Since we need a router for our **todos** resource and don't need the **routes/users.js** router module that Express Generator created, we'll modify it instead of having it lay around unused.

- First, rename the **routes/users.js** route module to a name that's more appropriate for our resource - **routes/todos.js**.
- The renaming of **routes/users.js** to **routes/todos.js** requires a couple of changes in **server.js**; both when the router module is being *required*:

```
// server.js

// around line 8
var todosRouter = require('./routes/todos');
```

  and when it's being *mounted*:

```
// around line 23
app.use('/todos', todosRouter);
```

- In **routes/todos.js**, let's remove the following code:

```
// DELETE the following...

/* GET users listing. */
router.get('/', function(req, res, next) {
  res.send('respond with a resource');
});
```

  and add the following comment to remind us of how the router was mounted:

```
// All actual paths start with "/todos"
```

- We now want to define the route for the To-Dos **index** functionality (show all To-Dos). However, we are not going to write an anonymous inline function for the route handler. Instead, we are going to follow a best practice of putting the function in a controller module that can export any number of controller actions (functions).
- Here's the route that uses a controller action that we'll code in a bit:

```
// routes/todos.js

var express = require('express');
var router = express.Router();

// Require the controller that exports To-Do CRUD functions
var todosCtrl = require('../controllers/todos');

// All actual paths begin with "/todos"

// GET /todos
router.get('/', todosCtrl.index);
```

> Is that route definition tidy or what?!?!

- Note that the Express app is unhappy as expected because the controller and the `index` function does not yet exist. That's next...

Controller Modules

In a web application that follows the MVC architectural pattern, **controllers**:

- Handle the request coming from the client (browser).
- Use Models to perform CRUD (create, retrieve, update & delete) data operations.
- Implement any additional application logic, often relying on other services and utility modules; and
- Pass data to Views to be rendered then return the resulting markup to the browser.
- Controllers are functions no different than the inline functions that we've already written!
- We just want to separate our concerns, i.e., we need to separate the **route definitions** from their respective **controller functions**.

Code the controllers/todos.js `index` Action

- Let's start by creating a controller module for the *todos* resource:

```
touch controllers/todos.js
```

> Yes, only modules for **models** are named singularly.

- Here's the function we used yesterday, just refactored into a function declaration named `index`:

```
// controllers/todos.js

function index(req, res) {
  res.render('todos/index', {
    todos: Todo.getAll()
  });
});
```

- Let's export the `index` controller method (also know as a controller action):

```
// controllers/todos.js

module.exports = {
  index
};

function index(req, res) {
  res.render('todos/index', {
    todos: Todo.getAll()
  });
}
```

> A common approach is to export an object near the top because you don't have to scroll to the bottom of the module to see what functionality is being exported.

- Yup, the controller is going to need to require that Todo model:

```
// controllers/todos.js

// Should name the model in uppercase and singular
const Todo = require('../models/todo');

module.exports = {
  index
};
```

Test the Refactor!

With the refactor complete, browsing to `localhost:3000/todos` should render the to-dos just like yesterday!

Hey, let's add a link on **views/index.ejs** so that we can click it to see the To-Dos instead of navigating via the address bar...

- In **views/index.ejs**:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <a href="/todos">To-Do List</a>
  </body>
</html>
```

- For styling, let's copy that `<link>` over to **todos/index.ejs** and...
- In **routes/index.js**, fix the value of the `title` property being passed to the view:

```
res.render('index', { title: 'Express To-Do' });
```

  That's better!

MVC Organization Revisited

Notice how we now have the following for the **todos data resource**:

- **models/todo.js**
- **views/todos** (directory)
- **controllers/todos.js**
- **routes/todos.js**

> Again, everything is named plurally except the model.

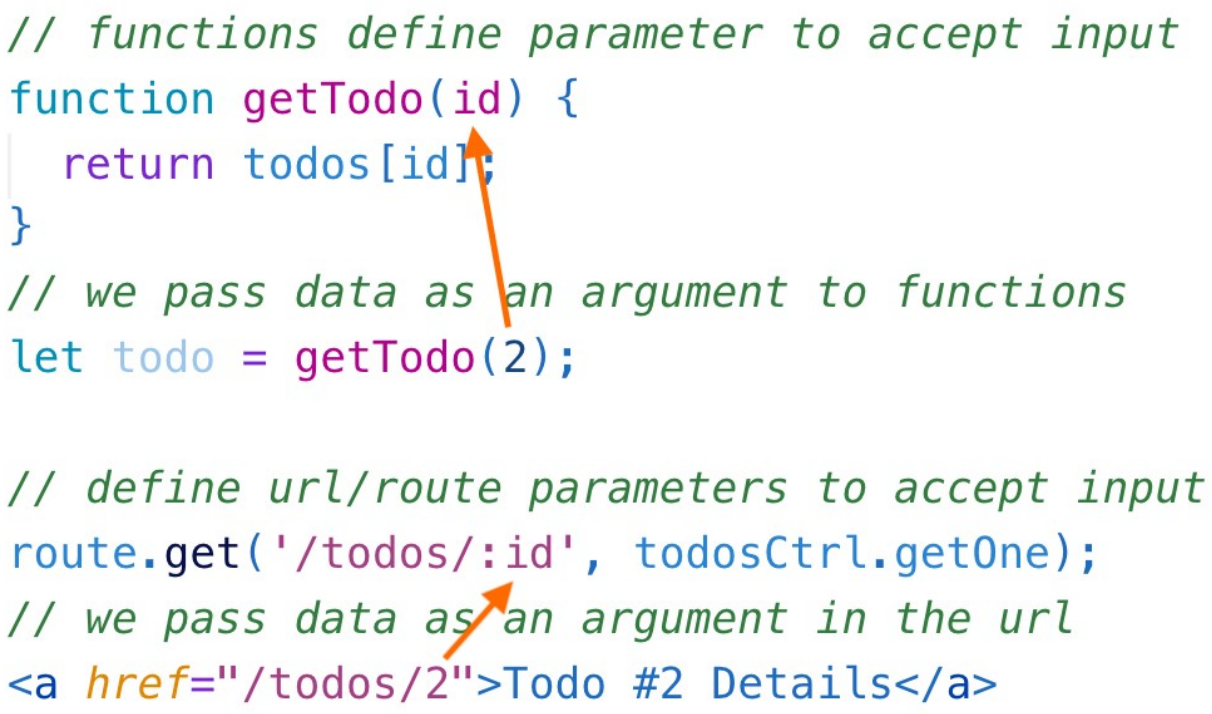
Each data *resource* should receive the same treatment. For example, if you had a **cats** data resource, another set of the above modules would be created and dedicated to CRUDing cats.

URL/Route Parameters

In our web apps, we will often need to pass information, such as an identifier for a certain data resource, in the **path** of the HTTP request.

URL Parameters, also known as **Route Parameters**, just like parameters in functions, provide a way for data to be passed in to the router & controller via the URL of the request.

Let's look at this analogy...



In Express, we define route parameters in the path string using a colon, followed by the parameter name.

Let's say we want to view a details page for a resource. Just like how we use an **index** route/action to list all of a resource, we will use a **show** route/action when displaying the details of a single resource.

Let's add the functionality to view a single To Do...

Adding Show To-Do Functionality

When adding functionality to your apps, start by identifying what route makes sense - this is usually based on RESTful/Resourceful Routing conventions.

We'll definitely be reviewing RESTful/Resourceful Routing later, in fact we just might quiz you on it one day - it's that important 😊

Identify the Proper Route

According to REST, the "proper" route to display a single To-Do would be:

```
GET /todos/:id
```

With the proper route identified, the next step is to create some UI that will send a request that matches that route!

Update the View to Add a Link

Let's refactor **todos/index.ejs** as follows:

```
<% todos.forEach(function(t) { %>
        <li>
        <a href="/todos/<%= t.id %>"><%= t.todo %></a>
```

Refresh the page and hover over the links. Looking at the bottom-left of the window will verify the paths look correct!

❓ Links always send an HTTP request using what HTTP method?

The UI is set to send the proper HTTP requests to the server.

However, clicking one of those links will display a
Not Found 404
error - this means that there is no route on the server that matches the HTTP request.

Let's add one...

Define the Route

Add the **show** route below the **index** route as follows:

```
// GET /todos
router.get('/', todosCtrl.index);
// GET /todos/:id
router.get('/:id', todosCtrl.show);
```

Saving will crash the app because there is no `todosCtrl.show` being exported from the controller...

Code and Export the `show` Controller Action

Add the `show` action inside of **controllers/todos.js** and don't forget to export it!

```
function show(req, res) {
  res.render('todos/show', {
    todo: Todo.getOne(req.params.id),
  });
}
```

> **KEY POINT:** Express's `req.params` object will have a property for each **route parameter** defined, for example...

A route defined like this:

```
router.get('/category/:catName/page/:pageNo', ...);
```

and a link like this:

```
<a href="/category/socks/page/2">Next Page</a>
```

would have a `req.params` available in the controller of:

```
console.log(req.params.catName) //=> "socks"
console.log(req.params.pageNo) //=> "2"
```

> Note that all route param values are strings.

Another refresh informs us that the `show` action in the controller is calling a `Todo.getOne` method that doesn't exist.

Add the `getOne` Method to the `Todo` Model

Let's fix that error! In **models/todo.js**:

```
module.exports = {
  getAll,
  getOne
};

function getOne(id) {
  // URL params are strings - convert to a number
  id = parseInt(id);
  return todos.find(todo => todo.id === id);
}
```

Refresh and of course there's an error because we haven't created the **views/todos/show.ejs** that we're trying to render.

Code the `show.ejs` View

Touch the **views/todos/show.ejs**.

Copy the boilerplate from **views/todos/index.ejs** and then add this:

```
<body>
  <h3><%= todo.todo %></h3>
  <h3>Complete: <%= todo.done ? 'Yes' : 'No' %></h3>
</body>
```

Refresh - BAM!

❓ Routing Quiz

Use the Routing Guide as necessary...

Assume a data resource of `cats` when answering the following:

1. What will the name of the router module be? (include its parent directory)
2. Write the line of code within **server.js** that would require the above router and assign it to a variable named `catsRouter`.

3. Write the line of code within **server.js** that would mount the above router object prefixing the proper path.

Using the `router` object within `routes/cats.js` and assuming a cats controller assigned to a variable named `catsCtrl`:

1. Write the line of code that defines the proper route that would read/display all cats (cats **index** route).
2. Write the line of code that defines the proper route that would read/display a single cat (cats **show** route).
3. Write the line of code that defines the proper route that would display a view that includes a form for submitting a new cat (cats **new** route).
4. Write the line of code that defines the proper route that would handle the cat form being submitted and creates a new cat (cats **create** route).

Congrats!

References

Note: When searching for info on the Express framework, be sure that you search for the info for version 4 only - there were significant changes made from earlier versions. Also note that version 5 is currently in alpha although all of the code we've written should be compatible.