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 / w05 / d1 / 01-04-express-middleware / **express-middleware.md**

cogilvy-ga Update express-middleware.md — History

1 contributor

Raw | Blame

Executable File 491 lines (317 sloc) 15.3 KB

Express Middleware

Learning Objectives

Students Will Be Able To:
Describe the Use Case of Middleware
Use `urlencoded` Middlware and HTML Forms to Create Data on the Server
Use `method-override` Middleware and HTML Forms to Update & Delete Data on the Server
Use Query Strings to Provide Additional Information to the Server

Road Map

- Setup
- What is Middleware?
- Our First Middleware
- Creating To-Dos
- `method-override` Middleware
- Delete a To-Do
- Exercise: Update a To-Do

Setup

This lesson builds upon the `express-todos` project created in the previous lesson.

What is Middleware?

In the Intro to Express lesson, we identified the three fundamental capabilities provided by web application frameworks:

1. The ability to define routes
2. The ability to process HTTP requests using middleware
3. The ability to use a view engine to render dynamic templates

We've already defined routes and rendered dynamic templates.

In this lesson we complete the trifecta by processing requests using middleware.

Middleware are Functions

A middleware is simply a function with the following signature:

```
function(req, res, next) {

}
```

As you can see, middleware functions have access to Express's *request* (`req`) and *response* (`res`) objects.

These objects contain useful properties and methods.

Because they are just objects, middleware may modify them in anyway they see fit.

The `next` parameter is a function provided by Express used to pass control to the next middleware in the pipeline.

Mounting Middleware

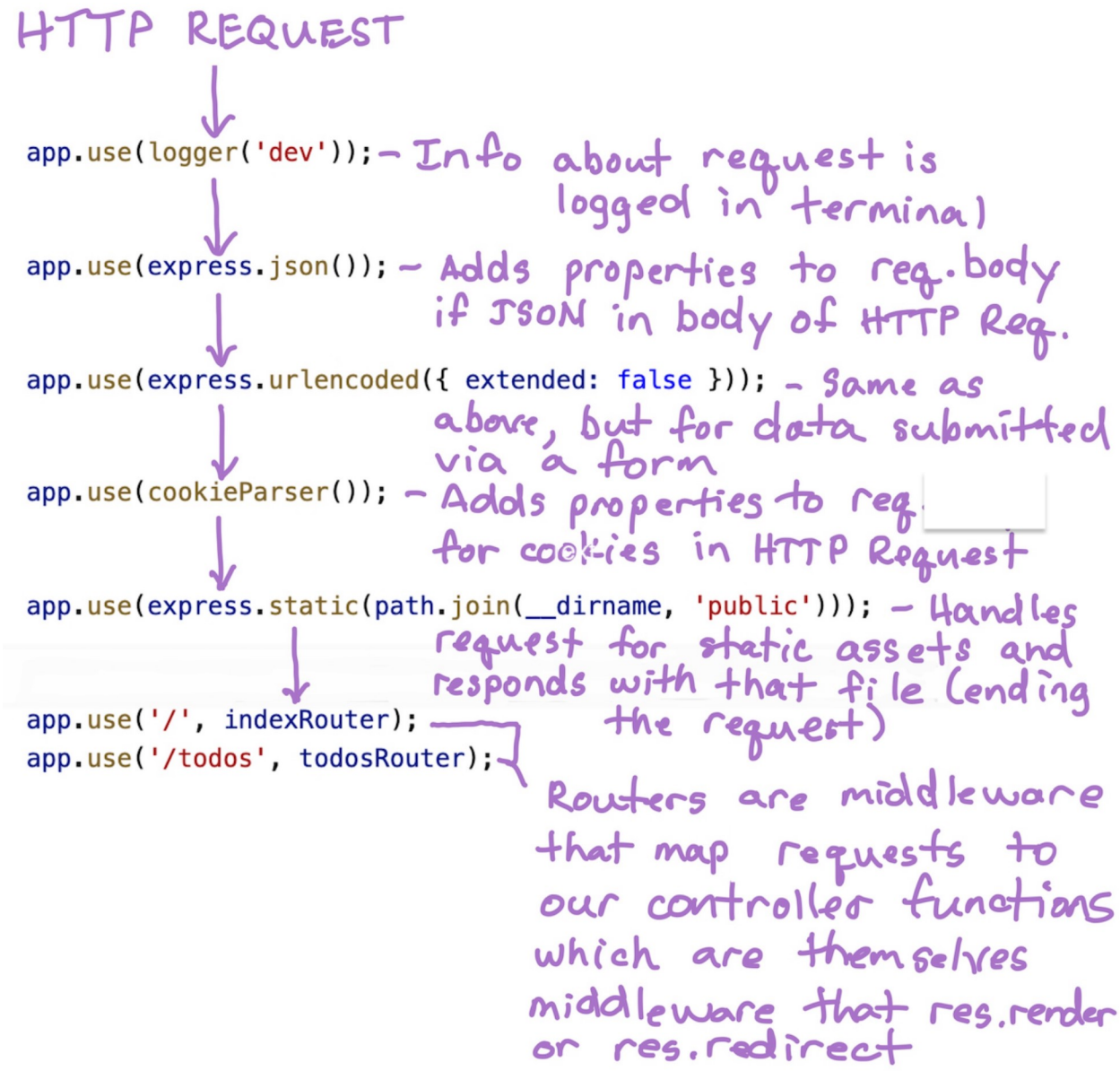
Express' `app.use()` method is used to mount middleware into its **middleware pipeline**.

It's called a **pipeline** because the HTTP request flows through it.

Purpose of Middleware

Middleware can be used to perform functionality such authentication and processing the request in multitude of ways.

Let's review the purpose of each middleware mounted in our Express generated app:



Controller Actions (Route Handlers) Are Middleware

Yes, you have already written middleware - the controller actions, `todosCtrl.index` & `todosCtrl.show`, are technically middleware!

The controller middleware functions didn't need to define the `next` parameter because they were at the **end of the middleware pipeline**. That is, they ended the request/response cycle by calling a method on the `res` object, e.g., `res.render`.

Our First Middleware

There's no better way to understand middleware than to see one in action.

Open **server.js** and add this "do nothing" middleware:

```
app.set('view engine', 'ejs');

// add middleware below the above line of code
app.use(function(req, res, next) {
  console.log('Hello SEI!');
  next();  // Pass the request to the next middleware
});
```

Type `nodemon` to start the server, browse to `localhost:3000`, and check terminal.

Let's add a line of code that modifies the `req` object by adding the current time to Express's request object that then can be accessed by any subsequent middleware:

```
app.use(function(req, res, next) {
  console.log('Hello SEI!');
  // Add a time property to the res.locals object
  // The time property will then be accessible when rendering a view
  res.locals.time = new Date().toLocaleTimeString();
  next();
});
```

> The res.locals object can be used to provide data to a view rendered during that request. In fact, the object provided as the second arg to `res.render` is merged with `res.locals`.

Now we can render the time in **todos/index.ejs** by updating the `<h1>` as follows:

```
<h1>Todos as of <%= time %></h1>
```

Refresh!

The Order That Middleware is Mounted Matters

We call it the **middleware pipeline** for a reason - the request flows through the middleware in the order they are mounted using `app.use`.

In **server.js**, let's move our custom middleware below where the routers are being mounted:

```
app.use('/', indexRouter);
app.use('/todos', todosRouter);

app.use(function(req, res, next) {
  console.log('Hello SEI!');
  res.locals.time = new Date().toLocaleTimeString();
  next();
});
```

Refresh shows that it no longer works because the router middleware are ending the request/response cycle before our "first middleware" is reached.

Move it back above the routes - yup, order of middleware matters.

Creating To-Dos

Time to add some additional functionality to our app - adding to-dos!

What exact functionality do we want?

Do we want to show a form on the `index` view, or do we want a separate page dedicated to adding a To Do?

Typically, for adding to-dos, you'd want have the form on the same page, however, today we'll demo the dedicated page approach so that we can see how **creating data is often a two-request task**:

1. Browser sends an initial request to see a page that includes a form to input the data, and...
2. The second request will happen when the form is submitted to the server so that it may create the new data, in this case a to-do, and respond to the client with a "redirect" (status code 302), i.e., tell the browser to make a new GET request.

Suggested Workflow to Add Functionality to a Web App

Here's a great flow to follow when you want to add functionality to your web app:

1. Identify the "proper" Route (HTTP Method + Path)

2. Create the UI (`<a>` or `<form>`) that will send an HTTP request that matches that route.
3. Define the route on the server and map it to a controller action.
4. Code and export the controller action.
5. `res.render` a view in the case of a `GET` request, or `res.redirect` if data was changed. Write the view template if it does not already exist.

Okay, since we are going for the two-request approach, let's get started implementing the functionality to show a page with a form to enter a new to-do...

Step 1 - Identify the "proper" Route

Checking the Resourceful Routing for CRUD Operations in Web Applications Chart, we find that the proper route is:

```
GET /todos/new
```

Step 2 - Create the UI that issues the request

Next step is to add a link in **views/partials/header.ejs** that will invoke this route:

```
...
  <a class="nav-link" href="/todos/new">Add To-Do</a>
```

Step 3 - Define the route on the server

Add the `new` route in **routes/todos.js** as follows:

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

I'm going to post this question in Slack for you to REPLY to:

Do you see why the `new` route must be defined before the `show` route?

Step 4 - Code and export the controller action

We need to code the `todosCtrl.new` action we just mapped to the `new` route...

In **controllers/todos.js**:

```
module.exports = {
  index,
  show,
  new: newTodo
};

function newTodo(req, res) {
  res.render('todos/new');
}
```

> Note that you cannot name a function using a JS *reserved* word, however, there's no problem with object properties.

Step 5 - Render the view & write it if necessary

We already called `res.render`, we just need to write the **new.ejs** template.

Create **views/todos/new.ejs**, include your partials (header & footer), then put this good stuff in there:

```
<h1>New Todo</h1>
<form action="/todos" method="POST" autocomplete="off">
  <input name="todo">
  <button type="submit" class="btn btn-primary">Save Todo</button>
</form>
```

Just a basic HTML form is being used to send data to the server when the form is submitted.

When the form is submitted, its `method` and `action` attributes determine what method and path the HTTP request will have:

- The `method` attribute holds the HTTP method/verb. It will usually be set to "POST", but may be a "GET" when performing searches.
- The `action` attribute holds the path. We'll see why we set `action="/todos"` in a bit.

> Note: The `autocomplete="off"` attribute will prevent the often annoying autocomplete feature of inputs.

Verify that clicking the **Add To-Do** link displays the page with the form - bravo!

Implementing the second-request functionality

Again, creating (or updating) data can take two separate requests - it depends upon the design of the app.

Let's get started implementing that second-request responsible for creating the new to-do on the backend...

Step 1 - Identify the "proper" Route

Check the Routing Chart and slack the proper route (HTTP Method & Endpoint) for creating data on the server

Step 2 - Create the UI that issues the request

The `<form>` is the UI and it's ready for business!

> Check this out if you want to learn more about HTML Forms.

✅ Step 1 - Determine proper route

✅ Step 2 - Create UI

Step 3 - Define the Route

In **routes/todos.js**:

```
router.get('/:id', todosCtrl.show);
router.post('/', todosCtrl.create);  // add this route
```

Yay - our first non- `GET` route!

Step 4 - Code and export the controller action

In **controllers/todos.js**:

```
  ...
  create
};

function create(req, res) {
  console.log(req.body);
  // The model is responsible for creating data
  // Todo.create(req.body);
```

```
    // Do a redirect anytime data is changed
    res.redirect('/todos');
  }
```

Note which properties on the `req.body` object gets logged out.

`req.body` is courtesy of this middleware in **server.js**:

```
  app.use(express.urlencoded({ extended: false }));
```

The properties on `req.body` will always match the values of the `<input>`'s `name` attributes:

```
  <input type="text" name="todo">
```

Okay, let's uncomment `Todo.create(req.body);` and go code it!

All we need is a `create` function in **models/todo.js**:

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

  function create(todo) {
    // Add the id
    todo.id = Math.floor(Math.random() * 1000000);
    // New todos wouldn't be done :)
    todo.done = false;
    todos.push(todo);
  }
```

Test it out!

> Note that when `nodemon` restarts the server, added to-dos will be lost.

`method-override` Middleware

As shown on the Resourceful Routing for CRUD Operations in Web Applications Chart, performing full-CRUD data operations requires that the browser send `DELETE` & `PUT` requests instead of `GET` s.

Using JavaScript (AJAX), the browser can send HTTP requests with any method, however, HTML can only send `GET` & `POST` methods. So what do we do if we want to delete a To-Do?

method-override middleware to the rescue!

Using `method-override` allows the request to be sent as a `POST` from the browser, but be changed on the server to a `DELETE` , `PUT` , etc.

Because `method-override` is not built into Express, we need to install it:

```
$ npm i method-override
```

Require it below `logger` in **server.js**:

```
var logger = require('morgan');
var methodOverride = require('method-override');
```

Now let's add `method-override` to the middleware pipeline:

```
app.use(express.static(path.join(__dirname, 'public')));
app.use(methodOverride('_method'));  // add this
```

> We are using the Query String approach for `method-override` as documented here.

Delete a To-Do

With `method-override` ready to go, let's add the functionality to delete to-dos!

The user story reads:
As a User, I want to delete a To Do from the list

Same process...

Step 1. Determine the proper route

▶ What's the RESTful route?

Cool, on to step 2...

Step 2 - Create the UI

By default, `method-override` only listens for `POST` requests, therefore we use a `<form>` to send requests that need to be treated as `PUT` or `DELETE` on the server.

Therefore, we'll use a `<form>` for the UI in **views/todos/index.ejs**:

```
...
  <td><a href="/todos/<%= t.id %>"><%= t.todo %></a></td>
  <td><%= t.done ? 'done' : 'not done' %></td>
  <td>
    <form action="/todos/<%= t.id %>?_method=DELETE" method="POST">
      <button type="submit" class="btn btn-danger">X</button>
    </form>
  </td>
```

The `?_method=DELETE` is the query string that `method-override` looks for. If it finds it, it changes the HTTP method to whatever is specified - always us all caps, e.g., `DELETE`.

Refresh and use DevTools to ensure the links look correct.

Step 3 - Define the route on the server

I bet you could have done this one on your own!

In **routes/todos.js**:

```
router.post('/', todosCtrl.create);
// new route below
router.delete('/:id', todosCtrl.delete);
```

Step 4 - Code and export the controller action - next...

Similar to `newTodo`, we can't name a function `delete`, so...

```
  create,
  delete: deleteTodo
};
```

```
function deleteTodo(req, res) {
  Todo.deleteOne(req.params.id);
  res.redirect('/todos');
}
```

The separation of concerns of the MVC design pattern means it's the Todo model's job to perform the delete.

Any questions?

Add the `deleteOne` method to the `Todo` model

All that's left is to add the `deleteOne` method to the `Todo` model:

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

function deleteOne(id) {
  // Find the index based on the id of the todo object
  const idx = todos.findIndex(todo => todo.id === parseInt(id));
  todos.splice(idx, 1);
}
```

Does it work? Of course it does!

💪 Exercises: Update a To-Do

Updating a To-Do is very similar to creating one because it also is a two-request process:

1. One request to display a form used to edit the To-Do.
2. Another request to submitted the form to the server so that it can update the To-Do.

Exercise #1:

As a User, when viewing the show page for a To-Do, I want to be able to click a link to edit the text of the To-Do

Hints:

- Follow the same steps we followed multiple times for adding functionality!
- Be sure to reference the Routing Chart to determine the proper routes!
- Your users will expect to have the data they want to edit pre-filled in the form's inputs. Use EJS squids to assign the current value of the todo to the input's `value` attribute. Don't forget the double-quotes though outside of the squids.
- In order to pull off the above hint, the controller action will first have to get the To-Do being edited (using the `Todo` model's `getOne` method) and provide it to the view.

Exercise #2:

As a User, when editing a To-Do, I want to be able to toggle whether or not it's done

Hints:

- Use an `<input type="checkbox" ...>`
- Checkboxes are checked when a `checked` attribute exists (no value is assigned).
- Use squids and a ternary expression to emit into the input a `checked` attribute, or an empty string.
- If the checkbox is checked when submitted, `req.body.done` will have the value of `"on"`, otherwise there won't even be a `req.body.done` property thanks to the crazy way HTML checkboxes were designed to work back in the day.

Have fun and enjoy the challenge!

References

When searching for info on the Express framework, be sure that the results are for version 4 - there were significant changes made from earlier versions.