Express Middleware

In this lab, you will use several Express middleware components in the creation of a simple login form.

# Objectives

In this lab, you will learn to

* use several built in Express middleware components together, and
* implement custom middleware.

# Set things up

You have been provided a minimal Express application to start the lab but first we need to install the required modules.

## Install dependencies

1. Run npm install in the lab directory.

This will install the dependencies for our application from the package.json file. Our dependencies are connect, cookie-parser, cookies, and express-session.

While not required, you may want to install nodemon (npm –g nodemon). nodemon automatically restarts your application when a file system change is detected and will save time so you don’t have to manually restart Node to view your edits.

# Add a logger

Open app.js in a text editor and you can see that currently our application doesn't do anything aside from listen to port 3000. Let's begin with one of the most common middleware components you will encounter – logging.

## Add logger middleware

1. Add the logger middleware to the application with an app.use() statement.

Use the provided LOGGER variable to format the log messages by passing it as the argument to morgan(). It contains the default logging format plus a couple of additional arguments that we will use later to examine cookies.

## Test the logger middleware is working

1. Navigate to <http://localhost:3000> and verify that a logging statement appears in the console. The browser will contain an error because we have not yet defined any working routes so let's do that next!

# Create a login form

Creating a login form means that our application will need to handle POST form data so next we need to use the body-parser middleware.

## Add bodyParser middleware

1. Add the body-parser middleware after the logger. This will ensure that our application and correctly interpret application/x-www-form-urlencoded data used in the login form.

## Create login middleware

1. Add a custom middleware function called login after the bodyParser middleware declaration.

The first thing this middleware should do is check the req.method and if it is not GET, return next(). We will be creating middleware next to handle POST data and this middleware should only handle GET requests.

1. Create a login form by using res.setHeader to set the Content-Type to text/html and then use res.write to generate a HTML form that takes a user and password as inputs and then a submit button. Make sure the input types are text for the user and password for the password. End the response with res.end().

## Create middleware to handle POST data

1. Add a custom middleware function called checkAuth and make sure it is declared after the login form. The first thing this middleware should do is check the req.method and if it is not POST, return next().

We will create a middleware soon to handle DELETE data and this middleware should only handle POST requests. The bodyParser middleware allows POST values to be available on req.body. The user and password values should be available as req.body.user and req.body.password. Create a simple check by validating that req.body.user === 'admin' and req.body.password === 'password'. If true, use res.end to indicate they are logged in and otherwise display an invalid credentials message.

## Test the login form

1. Navigate to <http://localhost:3000> and make sure that your login form appears. You should be able to see a successful login message with the correct credentials or an error message. This login form isn't very interesting without a session so let's add that next.

# Add a session store

Connect supports backing session stores with MongoDB, Redis, and others. In this lab we will just be using the default memory store.

## Add session middleware

1. After the bodyParser middleware, add the session() middleware. The session middleware requires a secret to be passed in the format app.use(session({ resave: false, saveUninitialized: false, secret: 'super secret string' })).

## Add session regeneration in checkAuth function

1. In the checkAuth function, add req.session.regenerate(function createSession(err) {…}) if there is a successful login. Move the successful login message inside this callback. Add the user name to the session (req.session.user = req.body.user;)

A normal web application would redirect to some other location after a successful login but redirect functionality is not built into Connect. Express, a higher-level framework built on top of Connect, provides this functionality for more traditional web development.

# Add logout middleware

At this point, we can login successfully with the application and create a session but we don’t have a way yet to logout and destroy the session.

## Add methodOverride middleware

1. Add the methodOverride middleware to the application after the bodyParser. This middleware will allow us to send a form with a DELETE method and Connect will populate the req.method object.

## Add a logout form to the login middleware

1. Modify the login middleware by checking to see if req.session.user exists. If so, present a form to logout. Inside the form element use the action attribute to pass \_method=DELETE as a query parameter. If req.session.user does not exist, present the login form as before.

## Add logout middleware

1. Create a custom middleware and place it last in the chain of middleware functions you have defined so far. Check that req.method is equal to DELETE and then use req.session.destroy(function logout(err) {…}) to destroy the session. Inside this callback, use res.end with a message indicating the user is logged out.

## Test the application

1. Navigate to <http://localhost:3000> and verify that the ability to log in and out of the session works properly.

Once everything is working properly, this lab is now complete!