Introduction to Computer Science I
COMP 2406A – Winter 2020

Express Part 2

Dave McKenney
david.mckenney@carleton.ca

Learning Outcomes

by the End of this Lecture, Students that have Completed the Reading Assignment and Review Questions should be Able to:

Perform advanced routing using Express parameters

Use regular expressions in your Express routes

Support multiple content types in your apps

Use Express to modularize/organize your code

Create RESTful CRUD APIs using Express

Express So Far

Last lecture we discussed Express – a Node.js framework for creating web apps

We set up some simple route handlers (e.g., GET /something, POST /something, etc.)

Express So Far

Now we will look at some more Express functionality

We will build up a store database web app using Express and other tools we have covered

Express So Far

To start, we will look at the completed code

We can view products, users, reviews. We can add new products, users, reviews.

Express So Far

What does the data look like in this app?

We have products, users, and reviews

Each has specific information associated with it

Store API

A user:

- **Has an ID, name, address**
- **May have 0+ reviews they have made**
- **May have 0+ products they have purchases**

Store API

A product:

- **Has an ID, name, price**
- **May have 0+ reviews made about them**
- **May have been bought by 0+ users**

Store API

A review:

Has an ID, reviewer (a user), product, rating, summary, and review text

Store API

How will the data be stored?

We could read all products/users/reviews into RAM when the server starts

This is not very scalable…

Store API

For now, we will keep the information in files

Next week, we will discuss databases and simplify our data storage in many ways

Store API

What routes does our server need to support?

Do these routes respond with HTML? JSON? Both?

Store API

A client may:

GET /products – searches all products, responds with JSON/HTML

POST /products – accepts a JSON body with name and price

GET /products/*someID* – gets a specific product

PUT /products/*someID* – update product using JSON body

Store API

A client may:

GET /users – searches all users , responds with JSON/HTML

POST /users – accepts a JSON body user information

GET /users/*someID* – gets a specific user

PUT /users/*someID* – update user using JSON body

Store API

A client may:

GET /reviews – searches all reviews , responds with JSON/HTML

POST /reviews – accepts a JSON body with reviewer and product URLs, creates a new random review

GET /reviews/*someID* – gets a specific review

PUT /reviews/*someID* – update review using JSON body

Adding Template Engine

The first thing we will do is add a template engine to our Express app

Express supports adding a template engine to your app with the command:

```
app.set("view engine", "pug"); //or ejs, etc.
```

You can set the base views directory with:

```
app.set("views", "./some/directory");
```

Adding Template Engine

You can then use the response object's render method to render a template:

res.render("/path/to/template", {*dataObject*});

Express fills in file extensions automatically based on view engine value (modularity!)

Adding Template Engine

Add a template engine to the store app and provide a way to render the home page

Add route handling for /products, /users, /reviews

Each should list some of the details with links to specific pages

Express and Routers

If we add all of our route handlers to this one file, things will get messy/confusing fast

Express provides the idea of 'routers' to further divide the server's functionality

Leads to cleaner code, modularity, maintainability...

Express and Routers

A router is like a mini-app, which can handle requests for a specific subsection of the API

Allows you to organize your code into various components (e.g., /users router, /products router)

Routers can themselves create additional routers

Allows you to define routers of various depth within the API to handle specific sections, sub-sections, etc.

Can add middleware to a router to perform intermediary processing (e.g., load a user's profile)

These ideas are best explained through an example

See the code in express-router-example.js and its referenced modules

Another common use of routers in Express is to support multiple API versions easily

If you create an API and it becomes popular, you can update the API by adding new versions without breaking old versions

Express and Routers

```
const express = require('express');
const app = express();

//Import the modules
const v1 = require("./v1/router.js");
const v2 = require("./v2/router.js");
const v3 = require("./v3/router.js");

//Mount the routers
app.use("/v1", v1);
app.use("/v2", v2);
app.use("/v3", v3);

app.listen(3000); //Start server
```

Routers and the Store

Refactor the existing code to provide a router to handle /products, /users, and /reviews

We will continue to add to these routers as we progress through the development process

RESTful Design

In our last Express lecture, we defined a route for each resource/method combination (e.g., GET /something)

Now we are considering MANY resources (e.g., /products/0, /products/1, /products/2, etc.)

RESTful Design

It wouldn't make sense to define a route for each specific resource

Imagine waiting to use your new account while developers add code and test the system…

We will now see Express gives us ways to easily handle large amounts of resources

Parameterized Routing

Express allows you to add parameters into route definitions

Allows for a general specification of routes and a way to extract the required information from a request (req.params)

Parameterized Routing

To add a parameter to a route, use ‘:paramName’ to represent the parameterized value in the route

For example, if you have many user profiles represented by /products/someUniqueID

You can: app.get(“/products/:profileID”, handleFunc)

Parameterized Routing

app.get("/profiles/:profileID", handleFunc)

Inside of handleFunc(req, res, next)…

If URL is /profiles/davemckenney
req.params.profileID is "davemckenney"

If URL is /profiles/AJGH-391-ASDGJOIE-3112
req.params.profileID is "AJGH-391-ASDGJOIE-3112"

Parameterized Routing

Inside handleFunc, you can write code to read information about the specified resource, if it exists (e.g., read the profile from a file/database)

Or send a 404 error if the resource does not exist

Parameterized Routing

Multiple parameters can be specified inside of a single route

Combining multiple parameters with unique IDs allows you to build a large API relatively easy...

Parameterized Routing

/profiles is all profiles

/profiles/:uid is the profile of user with ID=uid

/profiles/:uid/purchases is the purchases of user with ID=uid

Parameterized Routing

/profiles/:uid/purchases/:pid is a single purchase with ID=pid made by the single user with ID=uid

/profiles/:uid/purchases/:pid/prods is the set of products involved in the purchase with ID=pid made by the user with ID=uid

/profiles/:uid/purchases/:pid/prods/:p/price is the price of the product with ID=p that is part of the purchase with ID=pid made by the user with ID=uid

Parameterized Routing

Note, our URLs often don't get this long/deep

Instead, we can capitalize on interlinkings

Example:

/profiles/:uid/purchases/ could be an array of purchase IDs

You can then access specific purchases using their ID

Parameterized Routing

Add a parameterized route to handle GET requests for specific products

That is, URLs of the form: /products/*someID*

Add similar for specific users and reviews

Parameterized Routing

Using this approach, it is possible for us to get values we don't want

For example, we may have a constraint that all user IDs be made up of digits (i.e., 0-9)

When we have app.get("/profiles/:pid"), this is still matched by /profiles/willferrell

Parameterized Routing

We can write code for this validation ourselves

For example, we can verify that the parameter is made up of digits

Express also gives us a way to do this automatically in the route definition: regular expressions

Regular expressions are a way of specifying a search pattern

They are often used for finding matching strings

For a guide on regular expressions, see:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

A Note on Regular Expressions

For many, regular expressions are intimidating at first glance

/^([0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]{3}-[89ab][0-9a-f]{3}-[0-9a-f]{12})$/i;

/^\/users\/(\d+)-(\d+)$/

A Note on Regular Expressions

Use a reference sheet, break them down, test them out

For our purposes here, basic regular expressions will likely be sufficient

Note: you don't need to remember all of the rules of regular expressions (at least, not for this course)

Routing with Regular Expressions

So if our profile ID must be made up of digits (i.e., the values 0-9), we can do this in Express:

```
app.get(/^\/users\/(\d+)$/, (req, res, next) => {
    let userID = parseInt(req.params[0], 10);
});
```

Note: the route is not a string (“ “), it is a regular expression (/ /)

Again, this looks confusing at first, but we can break it down into the parts it is representing:

/^\/users\/(\d+)$/

Routing with Regular Expressions

Again, this looks confusing at first, but we can break it down into the parts it is representing:

/^\/users\/(\d+)$/

/ / surrounds the regular expression (like “ ”)

Routing with Regular Expressions

Again, this looks confusing at first, but we can break it down into the parts it is representing:

/^\/users\/(\d+)$/

^ represents the start of a string

$ represents the end of a string

So our URL must match the expression exactly

(i.e., nothing before/after what we specify)

Again, this looks confusing at first, but we can break it down into the parts it is representing:

/^\/users\/(\d+)$/

\ escapes 'special' characters

So \/ represents "/"

\d represents 'a digit' (0-9)

Routing with Regular Expressions

Again, this looks confusing at first, but we can break it down into the parts it is representing:

/^\/users\/(\d+)$/

+ represents ‘1 or more’ of the preceding character

In this case, one or more digits (\d)

Routing with Regular Expressions

Again, this looks confusing at first, but we can break it down into the parts it is representing:

/^\/users\/(\d+)$/

So this regular expression matches:

/users/*ONE_OR_MORE_DIGITS*

e.g., /users/292, /users/391102, etc.

And does not match: /users/dave, /users/83A, etc.

Routing with Regular Expressions

If you have a specific pattern for your unique IDs, you can filter requests so you only handle valid ones

For example, if you are using version 4 UUID, it follows the pattern:

xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx

Where x is any hex digit (0-9, A-F), y is 8, 9, A, or B.

Routing with Regular Expressions

So you can handle request URLs for a valid UUIDv4 with:

```
app.get(/^([0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]{3}-[89ab][0-9a-f]{3}-[0-9a-f]{12})$/i, someFunc)
```

i is for ‘ignore case’

Routing with Regular Expressions

For readability sake, it might be easier to save a complex regular expression into a variable first

```
let validUUID = /^([0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]{3}-[89ab][0-9a-f]{3}-[0-9a-f]{12})$/i;

app.get(validUUID, someFunc);
```

Update the parameterized product route to require product IDs to be digits…

Multiple Response Types

APIs often support multiple content types

For example, we may want to return HTML for /products (as we already are)

OR

We may want to provide JSON for /products (i.e., an array of products, IDs, etc.)

Multiple Response Types

Express provides a convenient way to implement support for multiple content types: res.format(…)

Takes an object as input with:

Keys = MIME types to match

Values = function to handle that MIME type

Multiple Response Types

So for example, we could have something like:

```
res.format(
  {
    "application/json" : sendJSONData,
    "text/html" : sendHTMLData
  }
)
```

Where sendJSONData and sendHTMLData are functions

Multiple Response Types

Modify the products router so that it can return JSON or HTML, depending on the value of the Accept header in the request

Test this with the Postman tool

Next, we want to add support for adding new products (or users, reviews, etc.)

We will accept POST requests to /products

Expect JSON data containing product information

Sharing Data Across App

One issue – our products need unique IDs

Our products router does not know what the existing IDs are

The next ID is stored in a config file – but we need to access this data all over our app

Sharing Data Across App

Express app supports the idea of local variables:

app.locals is an object you can add properties too

Can be accessed in middleware functions with:

req.app.locals

Sharing Data Across App

Add code in the main app file to load the config data and set up app.locals

Add code to the products router to handle POST requests to /products

Test functionality with Postman

Updating a Product

One last this we need to do to complete our product-related functionality:

Allow updating/changing a product

Updating a Product

The general workflow will be:

1. **Client requests a specific product JSON**
2. **Client modifies that JSON somehow**
3. **Client makes PUT request to product URL with new representation**
4. **Server updates specific product**

Updating a Product

We could achieve this workflow using HTML

We can also achieve it using JSON

Updating a Product

One more useful Express feature – you can define middleware to be executed when a route parameter exists

e.g., when :pid is present, we want do execute X

X will execute for any request where :pid parameter is present, BEFORE any route handler

Updating a Product

To add a route parameter middleware:

```
app.param(":paraName",function(req,res,next,para){
        //Argument para has value of :paraName
}
```

Example use – any time a product ID parameter is included, load that product's information first

Updating a Product

Add support for PUT requests to a parameterized product route

Test with Postman

Query Parameters

Currently, we only return the first 20 products, all the time

What about allowing a user to specify what subset of products they want?

e.g., name contains X, price > Y, etc.

Query Parameters

How can we add support for this functionality?

Query Parameters

Express's request object contains a query object

So req.query allows you to access specific query parmater values

Query Parameters

Add support for various query parameters to the /products route

Parameters to consider: name, minprice, maxprice

We can also consider pagination…

Next Steps

With these methods, you can make an Express-based API to create/retrieve/update/delete resources

This is the essence of Assignment #3, which will support HTML and JSON manipulation of restaurant data

Next Steps

Next, we will look at adding a back-end database to our web apps

This will allow us to more easily store/update/retrieve resource representations (i.e., no more manually reading files)