

Backend Development with Node.js

Lab 1

1	JAVASCRIPT AND EXPRESS.JS TERMINOLOGY	1
2	JAVASCRIPT BASICS	1
2.1	JAVASCRIPT ARROW FUNCTIONS.....	2
	<i>Example 1: Route Handlers</i>	<i>2</i>
	<i>Example 2: Middleware</i>	<i>2</i>
	<i>Example 3: Asynchronous Operations (e.g., database calls).....</i>	<i>2</i>
	<i>Example 4: Array operations within routes</i>	<i>3</i>
3	BASIC EXPRESS.JS PROJECT STRUCTURE	3
4	ROUTING	5
4.1	BASIC GET ROUTE	5
4.2	EXPLORING ROUTING	8
4.3	ADDING IN NODEMON FOR AUTOMATIC SERVER RESTARTS.....	10
4.4	ROUTE MATCHING FOR VARIOUS HTTP REQUEST METHODS	11
4.5	RETRIEVING ROUTE AND QUERY PARAMETERS	14
5	RETURNING DIFFERENT CONTENT TYPES.....	15
6	EXTRACTING CONTENT FROM REQUESTS	16
7	EXTRACTING CONTENT FROM REQUEST HEADERS	18
8	SETTING RESPONSE HEADERS AND STATUS CODE	19
9	SERVING STATIC FILES IN EXPRESS.....	20

1 JavaScript and Express.js terminology

A package is a collection of JavaScript code, functions or classes that are published as a distribution unit to the [NPM registry](#), which holds more than 2 million packages and is the largest software registry in the world

Dependencies are packages that are required a JavaScript application to run correctly. For e.g. a JavaScript web application would require Express package as a dependency.

npm is a package manager, which is a tool used for managing packages (e.g. downloading and installing packages). It is the default package manager that comes preinstalled with Node.js and is widely used in setting up. There are also other alternatives like Yarn and pnpm.

2 JavaScript basics

The source code files for this lab can be obtained from `labcode / basics` folder from the downloaded zip.

2.1 JavaScript arrow functions

Create a directory `firstapp`

[Arrow functions](#) are a concise way to write functions in JavaScript, introduced in ES6 (ECMAScript 2015). They are often used in modern Node.js and Express.js applications because they make code cleaner, especially for callbacks and asynchronous logic.

Place `demo-arrow.js` here and execute it with: `node demo-arrow.js`

Typical Use Cases in an Express.js App

Arrow functions are especially useful in Express.js for defining route handlers, middleware functions, and callbacks for asynchronous operations.

Example 1: Route Handlers

Traditional:

```
app.get('/hello', function(req, res) {  
  res.send('Hello World!');  
});
```

With arrow function:

```
app.get('/hello', (req, res) => {  
  res.send('Hello World!');  
});
```

This results in cleaner syntax

Example 2: Middleware

```
app.use((req, res, next) => {  
  console.log(`Request URL: ${req.url}`);  
  next();  
});
```

Here, the arrow function makes it concise and readable — especially for inline middleware.

Example 3: Asynchronous Operations (e.g., database calls)

```
app.get('/users', async (req, res) => {  
  try {  
    const users = await User.findAll(); // Assume Sequelize ORM  
    res.json(users);  
  }  
});
```

```

    } catch (err) {
      res.status(500).send('Error retrieving users');
    }
  });
}

```

Arrow functions are ideal here because:

- They integrate well with `async/await`.
- They preserve the lexical scope of `this`, avoiding unexpected context changes.

Example 4: Array operations within routes

```

app.get('/sum', (req, res) => {
  const numbers = [1, 2, 3, 4];
  const mult = numbers.map(x => x * 2);
  res.send(`Numbers multiplied by 2 is ${mult}`);
});

```

Arrow functions simplify inline callbacks in methods like `.map()`, `.filter()`, `.reduce()`.

Aspect	Traditional Function	Arrow Function
Syntax	<code>function() {}</code>	<code>() => {}</code>
<code>this</code> binding	Dynamic (depends on caller)	Lexical (inherits from outer scope)
arguments object	Available	Not available
Usable as constructor (<code>new</code>)	Yes	No
Express route handler use	Common	Cleaner & preferred

3 Basic Express.js project structure

Create a directory `firstapp` and open a DOS prompt in it. Type:

```
npm init
```

Press Enter to accept the default for all the questions except for the one below, for which you should enter a new value:

```
entry point (index.js): app.js
```

Press Enter to the final `Is this Ok` question and you will be returned to the prompt.

Open this folder in Visual Studio Code (or the particular IDE you are using).

You should now see a single file `package.json` in this folder. This was created by `npm init`, which is often the first command you run when starting an Express.js or any Node.js project. The `package.json` file is the main configuration file for any Node.js project (which can be Express.js or any other related JavaScript library / framework). It serves as the project manifest. Its purpose is to:

- a) Defines basic project metadata (name, version, description, author, license, etc.).
- b) Lists the dependencies and devDependencies that your project requires.
- c) Specifies scripts (like `npm start`, `npm test`) that automate tasks.
- d) Provides information to package managers (`npm`, `yarn`, `pnpm`) for installing the project (when the command `npm install` is run)
- e) Helps ensure portability and reproducibility — another developer can just run `npm install` and set up the environment.

The key fields within it are:

- **name / version** → Identifies the project.
- **description** → Optional, useful for documentation or publishing.
- **main** → Entry point file (e.g., `index.js` or `app.js`).
- **scripts** → Custom command shortcuts (`npm run <script>`).
- **dependencies** → Packages needed for runtime (Express, Mongoose, etc.).
- **devDependencies** → Packages needed only for development (Nodemon, testing tools, etc.).
- **keywords / author / license** → Metadata, helpful if publishing to npm.

Now, we will install Express.js in this directory by typing:

```
npm install express
```

These will download the required packages for the Express.js framework into a new subfolder called `node_modules` and also create an additional file called `package-lock.json`

The `node_modules` folder basically contains all the package dependencies listed in `package.json`. For e.g. `package.json` currently specifies a specific version of Express as its dependency. However, the Express framework itself consists of multiple packages, which in turn can depend on other packages (nested / transitive dependencies). All these package dependencies are stored as separate folders within `node_modules` folder

The purpose of `package-lock.json` is to lock the dependency tree to a specific version of each installed package, including nested (transitive) dependencies. This ensures that every developer (or CI/CD pipeline) installs exactly the same versions of packages. Its content includes:

- Exact version numbers of every installed package (not just top-level ones).
- The resolved URL of each package from the npm registry.
- Integrity hashes (SHA512) to verify package authenticity.
- Dependency relationships (which package depends on which).

4 Routing

The source code files for this lab can be obtained from `labcode / routing` folder from the downloaded zip.

4.1 Basic GET Route

Create a file `app.js` into this project folder (you had earlier specified this as the main entry point for this project in `package.json`).

Place it with the content from `app-v1.js`

In the command prompt in this directory, type the following to run the application which starts a server that will listen by default on port 3000.

```
node app.js
```

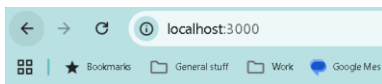
A log output message as follows should appear in the prompt:

Example `app` listening on port 3000

Then open a browser tab at this default port to send a HTTP request to this running server

<http://localhost:3000/>

You should be able to see the basic output returned from the app.



Hello World!

Congratulations ! You have run your first Express.js app.

A detailed explanation of the code:

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

- Loads the Express module using Node.js's CommonJS loader.
- `require('express')` returns the module's exported function (a factory for creating an Express application).
- `const express` binds that factory to a constant so you can call it later.

```
const app = express()
```

- Invokes the Express factory to create an **application instance**.
- `app` is your central object: you'll register routes (`app.get`, `app.post`, ...), middleware (`app.use`), settings (`app.set`), and start the HTTP server from it.

```
const port = 3000
```

- Declares the TCP port number the server will listen on.
- Using a constant keeps the value in central location so it is easier to locate and modify

```
app.get('/', (req, res) => {  
  res.send('Hello World!')  
})
```

- Registers a callback handler (**route handler**) for HTTP **GET** requests to the route path is the root URL /. The
- Signature:
 - req (IncomingMessage extended by Express): contains request data (headers, query, params, body, etc.).
 - res (ServerResponse extended by Express): methods to craft the response.
- Callback handler body:
 - res.send('Hello World!') sets a 200 OK status (by default), picks an appropriate Content-Type (here text/html; charset=utf-8 or text/plain depending on Express version), sends back the HTTP response to the originating client with the body containing the specified text.

```
app.listen(port, () => {  
  console.log(`Example app listening on port ${port}`)  
})
```

The `listen()` gets the app to start an underlying **HTTP server** and **bind** to `port`. Without a hostname, Node listens on all available interfaces (e.g., `0.0.0.0 / ::`)—meaning it's reachable from other machines on the network (firewall permitting).

- The **callback** runs once the server is successfully listening (non-blocking). It's a good place for startup logs, health checks, or to print the URL.

`console.log` uses template literal syntax. We typically use this syntax when working with long or complex strings that need to contain variable values. They provide features such as

- Multi-line strings: They can span multiple lines without needing escape characters like `\n`.
- String interpolation: They allow embedding expressions directly within the string using the `${expression}` syntax

NOTE: Express does not implement its own HTTP server, instead it relies on the built in `http` module from Node.js and provides a wrapper implementation around it so that the code is much easier to write.

Stop the server with Ctrl+C at the command prompt.

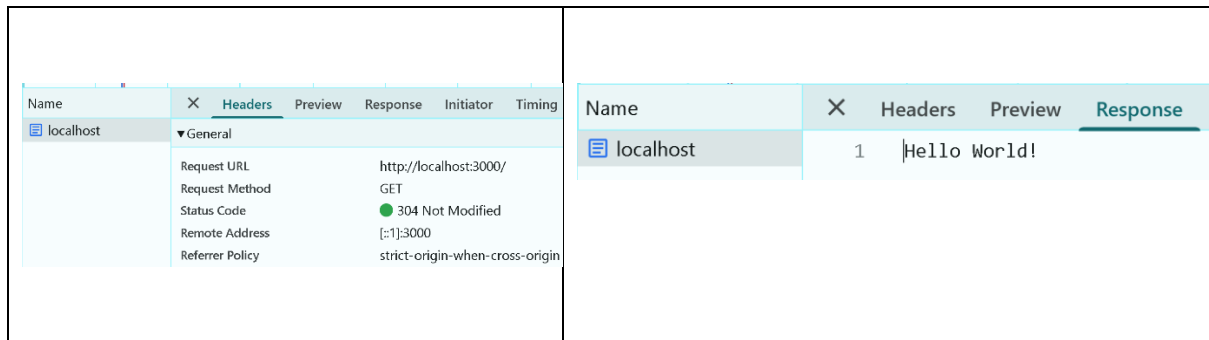
Now make some random changes in the string in the send function call, i.e.

```
res.send('Hello Western Digital!')
```

Restart the app again with `node server.js` (you can use the up arrow key to tab through previous commands) and verify that you see the latest updated message.

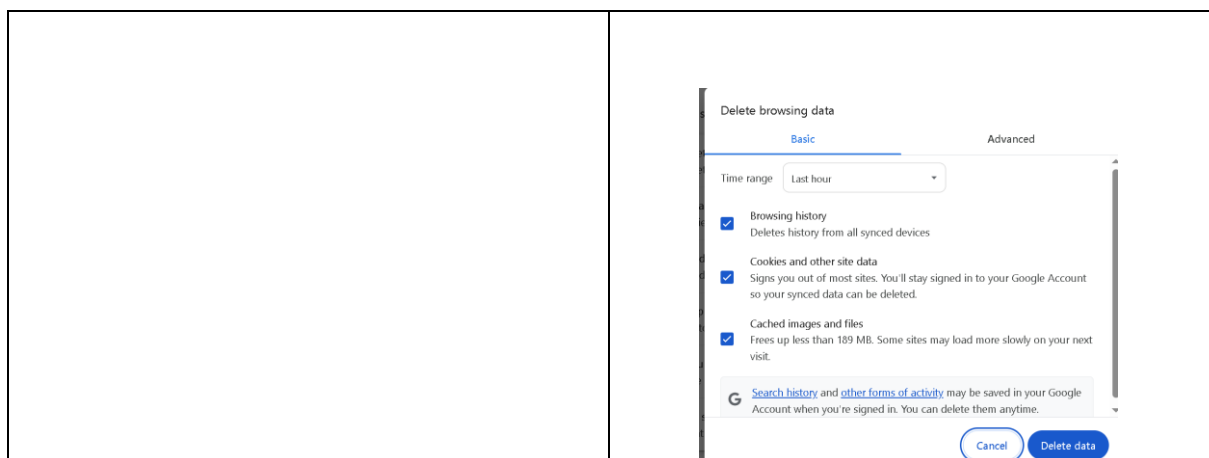
If you check in the Network tab of the Chrome Developer tools, you will be able to see the GET request being sent out to the `/` route, and the response being returned.

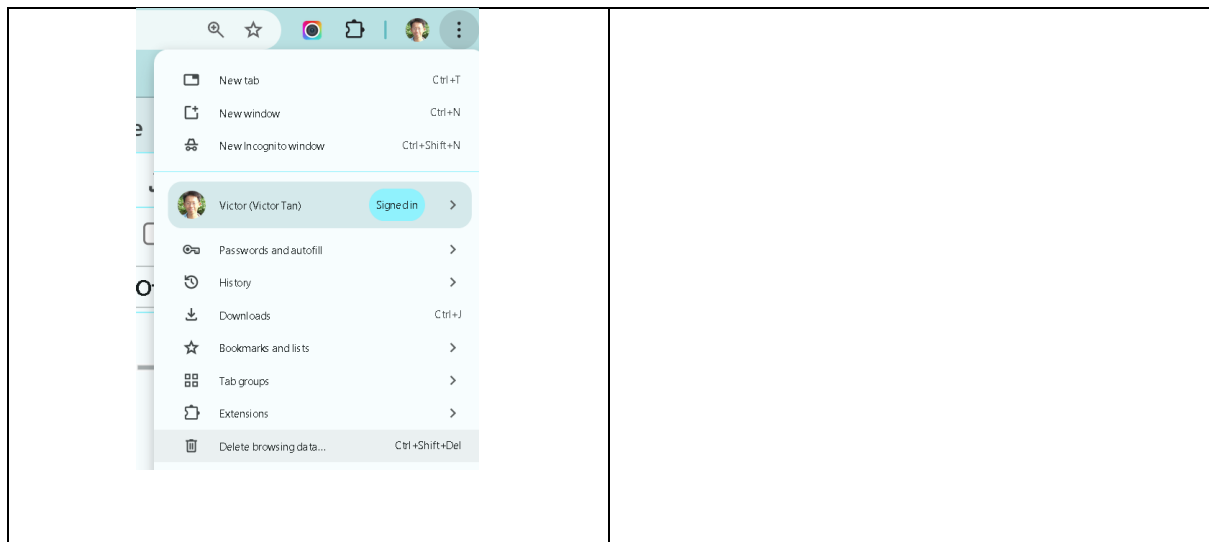
IMPORTANT NOTE: Make sure you [turn off all extensions](#) in Chrome (or whichever browser you are using) before you start inspecting network traffic via the Network tab as certain extensions may make additional calls to the running Express server in the background and provide confusing information.



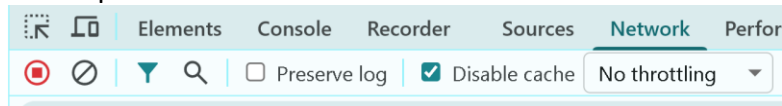
If you send out multiple requests to the app (simply by reloading the web page with F5) without changing the code to tailor the response returned, you will notice we are getting back a HTTP status of 304 Not modified, instead of the expected 200 OK. This is not an error; it is part of HTTP caching which Express.js uses in conjunction with conditional headers that it includes in its initial response to the client (in this case your browser) to instruct the client to try and cache responses so that identical responses will not be sent back in the future, thereby minimizing traffic.

If you wish to see a valid 200 OK response for the request for your browser, clear your browser cache first and send out the request again:

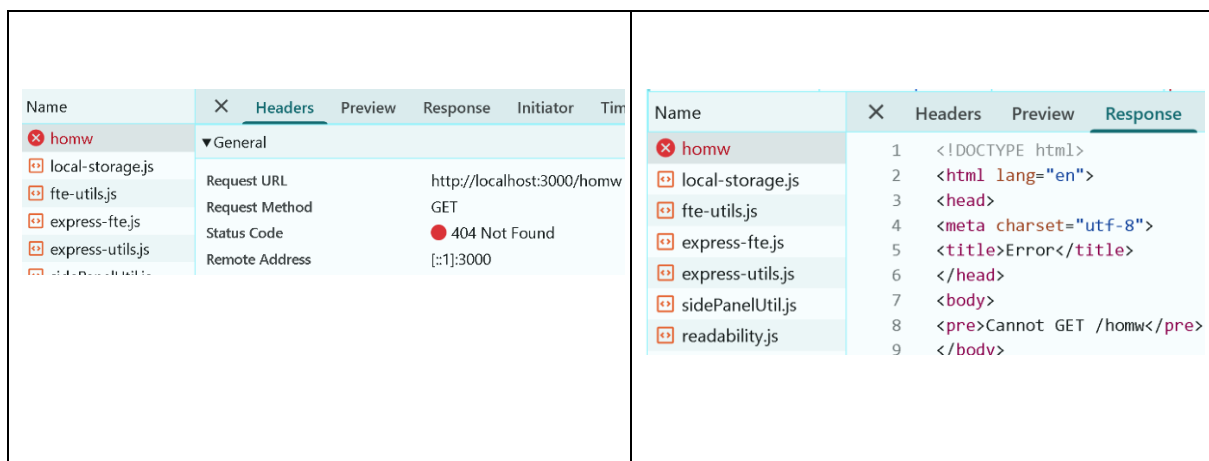




Alternatively, you can temporarily disable the cache in the Network view of Chrome Developer tools



Notice that if you attempt to send a request to any other route (for e.g. `localhost:3000/home`), you will get back an error response notifying you that the app cannot perform a GET request to that particular route. This will also be visible in the Network tab with a standard 404 error.



This is because there is no matching route specified in the app for `/home` yet, which we will be doing next.

4.2 Exploring routing

Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI / Path and a specific HTTP request method (GET, POST, and so on).

Each route can have one or more handler / callback functions, which are executed when the route is matched.

Route definition takes the following structure:

```
app.METHOD (PATH, HANDLER)
```

Where:

- `app` is an instance of `express`.
- `METHOD` is an HTTP request method, in lowercase.
- `PATH` is a path on the server.
- `HANDLER` is the callback function executed when the route is matched.

Update with content from `app-v2.js` and stop and restart the app

We now have defined several additional hardcoded routes (`/home`, `/order`, `/order/cars`) with a handler that just returns some text to indicate the matching of the route. The handlers use the `res.send` method to return a HTTP response back to the client. It is one of the most common ways to complete the request–response cycle.

When you call `res.send(...)`, this:

- a) Sets the response body (with whatever data you provide).
- b) Sets the Content-Type header automatically, based on the type of data you pass (we will see later the different types of content you can return)
- c) Ends the response (so you don't need to call `res.end()` manually).

For one of the routes (`/redirect`), we can call the `redirect` method to redirect back to one of the existing routes.

In addition, we now start using the middleware feature of Express via the method `app.use()`. This method is used to mount middleware or specify routes in an Express application. We will explore middleware use in greater detail in an upcoming lab.

Middleware functions are functions that have access to the following three objects: (`req`, `res`, `next`) where:

- `req` → the request object (HTTP request made by client)
- `res` → the response object (HTTP response to be sent to client)
- `next` → a callback function that passes control to the next middleware in the stack

Middleware functions sit between the incoming request and the outgoing response, processing or modifying them as needed. The method `app.use()` registers such functions and tells Express how/when to execute them.

Here, we will place it at the end of all route definitions.

```
app.use((req, res) => {
  res.status(404).send('Sorry, page not found!')
})
```

Express processes middleware and routes in the following order.

- If no earlier `app.get` etc. matches, the request "falls through" to this `app.use`.
- Since this `app.use` does not specify a path (defaults to `*`), it matches everything that comes after.
- By convention, you use this to return a 404 Not Found response.

Try out all the routes as well as a non-matching route (e.g. `/asdf`) and verify that you get back the expected responses.

4.3 Adding in nodemon for automatic server restarts

[Nodemon](#) is a very useful tool that helps in developer server-based Javascript application (such as our current Express.js application) by automatically restarting the application when any file changes in the project directory are detected. This removes the need for us to constantly restart the app every time we make changes to our source code or any other configuration file within the project directory.

Stop the currently running server and install Nodemon globally for now to speed up your development process.

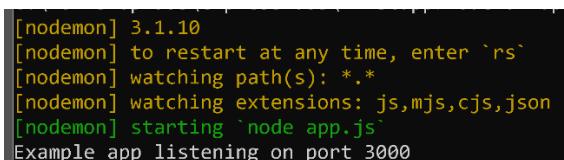
```
npm install -g nodemon
```

Since this is a global installation, you will be able to access it directly from the path variable in the DOS command prompt.

Now start your app again, but this time using `nodemon` instead of `node`:

```
nodemon app.js
```

A bunch of messages should appear indicating which file types (JavaScript, JSON, etc) that nodemon is monitoring for changes and in which directories (typically the project directory)



```
[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node app.js`
Example app listening on port 3000
```

Make some changes to `app.js` by introducing new routes in the source code and save the file.

You should see `nodemon` immediately pick up these changes and restart the server.

You can immediately try testing out accessing your newly added route through your browser in the usual manner.

As an alternative to starting nodemon directly from the command line, you can also start it as a script by modifying your `package.json` as follows:

```
...  
...  
  "main": "app.js",  
  "scripts": {  
    "start": "node app.js",  
    "dev" : "nodemon app.js"  
  },  
  "author": "",  
  "license": "ISC",  
...  
....
```

Stop nodemon by typing Ctrl-C.

Now you can start the app again using nodemon by using the configured script

```
npm run dev
```

Stop nodemon by typing Ctrl-C.

Alternatively, you can start the app using node by using this command:

```
npm start
```

Last, but not least, another alternative option was to initially have installed nodemon as a development dependency within the current project directory at the start with:

```
npm install --save-dev nodemon
```

This makes is available during development time but not during run time on a production server since we are unlikely keep restarting the server in that environment. You can skip doing this for now, and keep this in mind for future development work.

4.4 Route matching for various HTTP request methods

For REST APIs which frameworks like Express.js are used frequently to implement, there are a standard set of [frequently used HTTP request methods](#) that can be send to those REST API endpoints.

We will implement [matching routes](#) for all these methods.

Update with content from `app-v3.js` and stop and restart the app

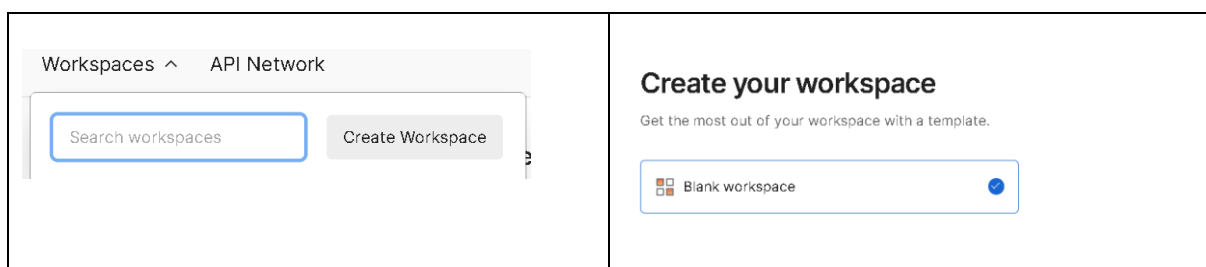
There is a limited number of HTTP request methods that a browser can issue directly on its own:

- a) GET: This is the most frequent method, used when a user navigates to a URL, clicks a link, or submits a form with the `method="get"` attribute.
- b) POST: This method is used when a user submits a form with the `method="post"` attribute, sending data to the server to create or update a resource.
- c) HEAD: Similar to GET, but it requests only the headers of a response, without the actual body. Browsers might use this to check resource existence or metadata without downloading the full content.
- d) OPTIONS: Browsers can automatically send an OPTIONS request as part of a preflight request in Cross-Origin Resource Sharing (CORS).

There are many GUI-based REST clients which provide access to the full range of REST HTTP methods, as well as ability to tailor the request headers and bodies as well as manipulate and store the results. The most popular and widely used of this at the moment is Postman

Start Postman (this process might take some time if there is an automatic update in the background during start up) and login using your Gmail account or existing account username or password.

We will create a new workspace for this lab. Select the Workspaces option from the main menu, and select Create New Workspace. Choose a Blank Workspace and click Next.



Providing the following details for the workspace for Only me (Personal)

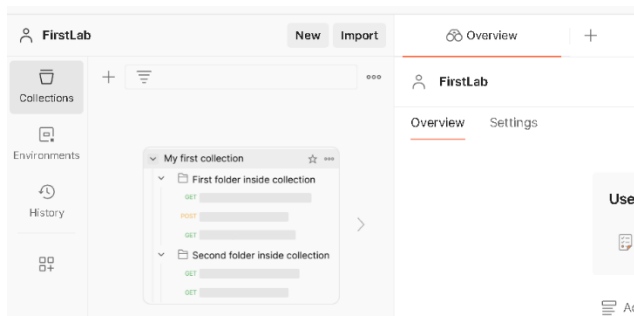
Name: FirstLab

Create your workspace

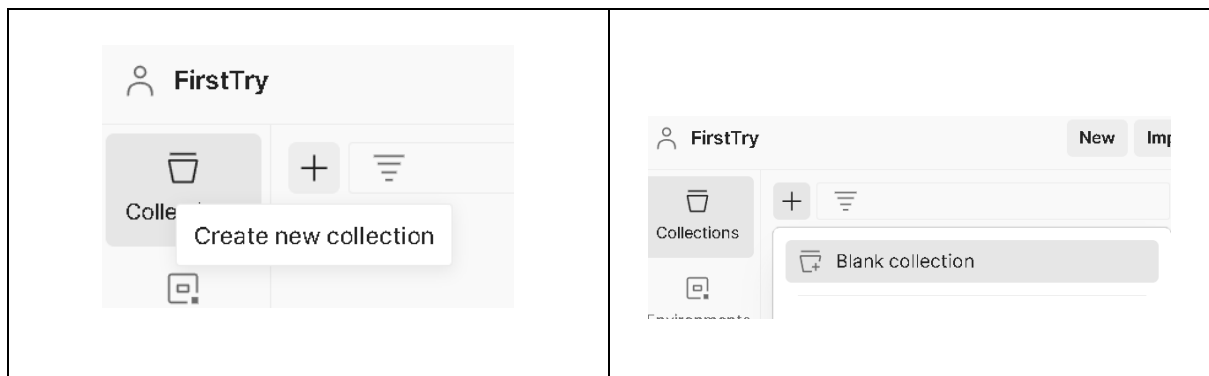
Name
FirstLab

Who can access your workspace?
Only me
Personal

Then select the Create button. Postman will spend a couple of minutes creating the workspace and navigate you into it.



A workspace consists of a series of collections, which in turn contain a group of one or more HTTP requests. We will first click the Create Collection button to create a collection, which we will name FirstCollection.



Then we will create a new HTTP request by any of these actions below:

- Clicking on New icon, selecting the HTTP icon
- Clicking on the + icon next to the most recent tab in the main view
- Click on the hamburger Main Menu icon in the upper left hand corner, select File -> New Tab

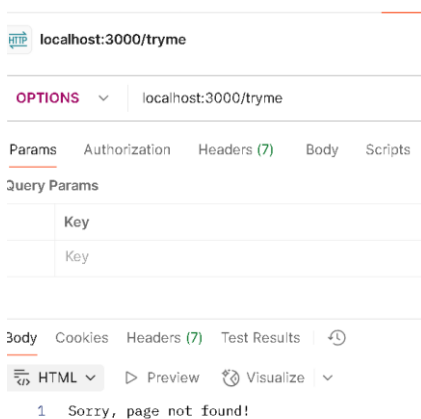
Type in the complete URL for the API endpoint (`http://localhost:3000/tryme`) that you want to send a request to and select the method you wish to use (by default this is GET). Click SEND and check the result you received back in return. You can repeat this for all the various HTTP methods by using the list of HTTP methods available from the drop down list.

--	--



Notice that you can implement different HTTP methods for the same route (`/tryme`) with different corresponding handlers: this is perfectly legal.

If you try a HTTP method that is not implemented (for e.g. `OPTIONS`) from Postman for this route, then the `catch-all` `app.use((req, res))` middleware, and the corresponding error message is returned.



Using the [HEAD](#) method however will not give this error message, since this is only used to retrieve HTTP header information for a resource without actually retrieving the resource (the error message).

4.5 Retrieving route and query parameters

Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the `req.params` object, with the name of the route parameter specified in the path as their respective keys.

Query parameters are key-value pairs that appear after the `?` in a URL. They are automatically parsed by Express and available in `req.query`.

Update with content from `app-v4.js`

You can test out GET requests to the routes with single and multiple path parameters directly from the browser for e.g.

`http://localhost:3000/books/4/2`

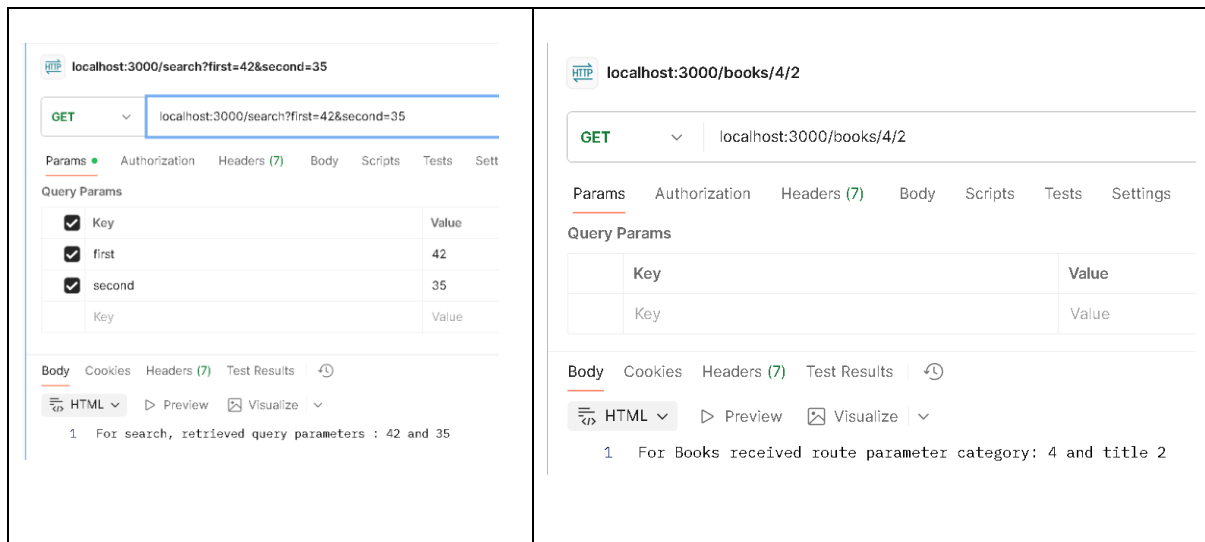
`http://localhost:3000/orders/3`

Test out the route with 2 query parameters:

`http://localhost:3000/search?first=42&second=35`

You can verify the parameters being passed from the Network view of the browser.

You can also send route and query parameters from Postman:



5 Returning different content types

So far by default, the content type returned from the handlers is text, this is represented as the [MIME type](#): `text/html; charset=utf-8`, which is visible in the Headers of the response from Postman.

Body	Cookies	Headers (7)	Test Results	🔄
Key	Value			
X-Powered-By	Express			
Content-Type	text/html; charset=utf-8			

We can also configure the route handlers to return different common MIME response types, in particular JSON (which is the de facto standard language for REST API interaction between web services).

The response headers will have appropriate values according to the response body content type (some of this can be verified in both the browser and Postman, some can only be seen in Postman).

We can also configure the route handlers to either render the requested resource (image, PDF) directly in the browser or prompt the browser to initiate a download of the resource instead.

Update with content from `app-v5.js`

Make requests to the various route paths and verify the correct content is returned. This demos most of the major response content types than an Express app will typically return:

- a) HTML (text/html)
- b) Plain text (text/plain)
- c) JSON (application/json)
- d) XML (application/xml)
- e) PDF (application/pdf)
- f) Static image files (image/png, image/jpeg)

For the text file `sample.txt` to be returned, you can just simply create this file yourself with sample content in the project folder with a text editor.

For the return and downloading of image file (`cat.jpg`) and PDF file (`words.pdf`), copy over those files from `labcode` into the project folder.

For resources to be downloaded, accessing the route path in the browser will cause the download dialog box to appear. For Postman this will be rendered directly in Postman itself. It may be easier to view the response headers in Postman.

Method	Purpose	Browser Behavior	Content-Disposition Header	Typical Use Case
<code>res.sendFile()</code>	Sends a file to be displayed inline (viewed in the browser if supported)	Displays file in browser (e.g. images, PDFs, text)	<code>inline</code>	Serve files like images, PDFs, or HTML
<code>res.download()</code>	Sends a file as an attachment for download	Triggers a "Save As..." dialog	<code>attachment; filename="..."</code>	Let users download files (reports, invoices, etc.)

6 Extracting content from requests

When building RESTful APIs, a number of different request methods (such as POST, PUT and PATCH) will included content in their bodies (typically JSON), which is subsequently extracted on the server side in the Express app

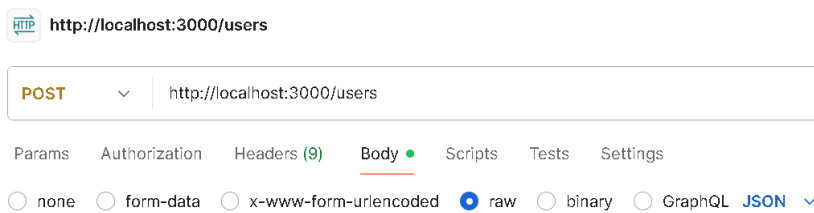
Method	Typical Use Case	Example JSON Usage
POST	Create a new resource	<code>{ "name": "Alice", "email": "alice@example.com" }</code>
PUT	Replace an existing resource entirely	<code>{ "id": 1, "name": "Updated Alice" }</code>
PATCH	Update part of an existing resource	<code>{ "email": "newalice@example.com" }</code>

Update with content from `app-v6.js`

To extract JSON content, we need to use built-in [express.json](#) middleware to parse JSON content in body of HTTP requests, which we declare right at the start of the app.

We can use Postman to create the JSON content to place in the various POST, PUT and PATCH requests. In a full stack deployment, these requests and their associated content will be sent from a front end app.

For the POST to `/users`, we can configure with the following parameters



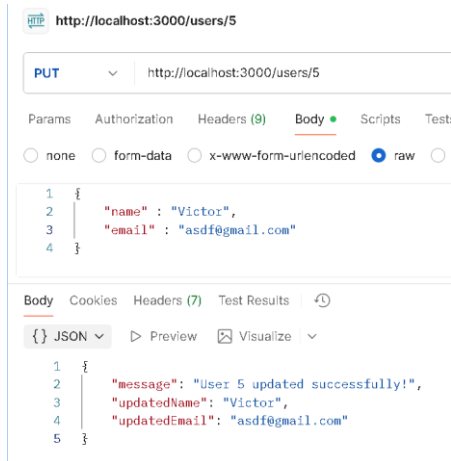
and use the JSON content similar to below (you can modify the values below but keep the key names as this will be what the Express app will look for in order to extract values).

```
{
  "name" : "Victor",
  "email" : "asdf@gmail.com"
}
```

Click Send and verify that the JSON content returned and console log output correctly reflects the content extracted from the POST body.



Repeat the same content for PUT to `/users/5`, with the same configuration parameters and also content.



The implementation and use of PATCH is nearly identical to PUT, so we skip it here. The key conceptual difference being that PATCH performs a partial modification of the content of a resource (which could be a database record or a JSON file), while PUT perform a complete modification.

7 Extracting content from request headers

When building RESTful APIs, we may also need to customize the values for specific [HTTP headers](#), which is subsequently extracted on the server side in the Express app

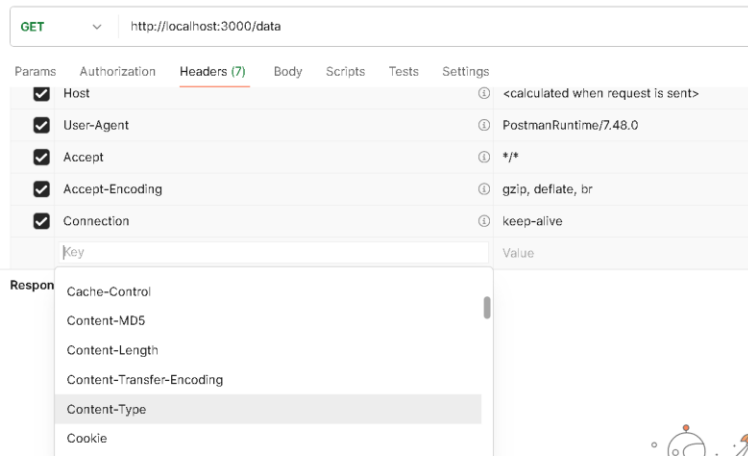
Header Name	Purpose
Content-Type	Specifies the media type of the request body (e.g., <code>application/json</code>)
Authorization	Used for passing credentials (e.g., bearer tokens or API keys)
User-Agent	Identifies the client application (browser, app, tool)
Accept	Indicates the media types the client is able to understand
X-Requested-With	Custom header often used to identify AJAX requests
X-Custom-Header	Example of a user-defined custom header
Referer	Indicates the origin of the request (where the user came from)
Origin	Indicates where the fetch request originated (for CORS)

Extracting HTTP request header info is a pretty straight forward process using the `req` object.

Here we again use middleware via `app.use()` which we register right at the top of the app, so it will intercept all incoming requests (regardless of the route path or the HTTP method) in order to extract the relevant HTTP headers.

Update with content from `app-v7.js`

To test via Postman, you can configure the header content from the drop down list of common headers or add in your own custom header:



We will set the following common HTTP headers: Content-Type, Authorization and Accept. Of course, we can extract any other header value we want in the same way in Express.js

<input checked="" type="checkbox"/>	Content-Type	application/json
<input checked="" type="checkbox"/>	Authorization	Basic secretpassword
<input checked="" type="checkbox"/>	Accept	application/json
	Key	Value

Click send and verify that these values are successfully extracted from the console log output at the server side.

8 Setting response headers and status code

Just as we can customize HTTP request headers for specific use cases, we can also customize HTTP response headers for responses returned from a RESTful API. Some of the more common HTTP response headers that we might wish to customize include:

Header Name	Purpose
Content-Type	Specifies the format of the response body (e.g., application/json)
Access-Control-Allow-Origin	For CORS: indicates which origins are allowed to access the resource
Authorization	Optional in some APIs to refresh tokens or confirm access
ETag	Entity tag for caching validation
Location	Used with 201 Created responses to specify the URL of the new resource
Set-Cookie	Used to set cookies (e.g., for session or token)

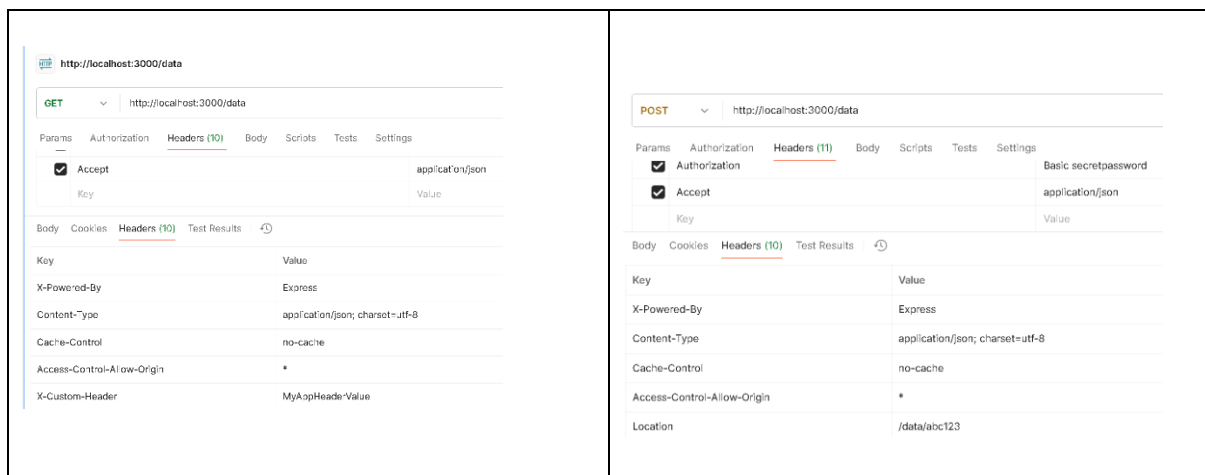
We can also customize [HTTP status code](#) for the response returned. By default, a successful (or normal) response will have a status code 200, and we can explicitly set this or set a different appropriate status code.

Update with content from `app-v8.js`

Populating HTTP response header info is a pretty straight forward process using the `res` object.

This can again be done using middleware via `app.use()` which we register right at the top of the app, so it will process all returned responses (regardless of the route path or the HTTP method) in order to populate the relevant HTTP headers. The population of HTTP response header info can also be done in individual route handler methods.

Send GET and POST requests via Postman to the `/data` path and verify that the appropriate HTTP response headers and status code are received.



9 Serving static files in Express

To serve static files such as images, CSS files, and JavaScript files from a dedicated subfolder within the main project folder, we can use the [express.static](#) built-in middleware function in Express. This is typically used in conjunction with HTML content that is returned from a route handler which in turn references all these resources. This allows us to construct a simple static website.

Update with content from `app-v9.js`

Place the `public` and `assets` folders into the current project folder.

Make a call to the root path route:

<http://localhost:3000/>

to view all the HTML content as well as the various files referenced from the HTML content

Check the Console view in Developer tools to verify that the `console.log` output from the various referenced Javascript files are executed, indicating the files were successfully downloaded.

You can also do a View Page Source on the webpage to show the actual HTML content downloaded.

We can also alternatively directly load these static files (independently of any HTML content that might reference them) through routes as follows:

<http://localhost:3000/images/cat.jpg>

<http://localhost:3000/css/styles.css>

<http://localhost:3000/js/script.js>

<http://localhost:3000/static/extra.js>