Chapter 10

Express

What is in This Chapter ?

This chapter introduces us to the **Express** module which will simplify the code on our servers. We discuss the **Express** ***request/response cycle*** and then get into the details about **routing** and **middleware**. We discuss various **types of middleware** and show how static pages can be served easily. We then discus details about request/response **bodies**, **queries** and request **router objects**, response **types** and **shared data**. We will then discuss the conversion of our **FutureTech Corp.** website to **Express**. Finally, we will conclude the chapter with a presentation of an example store site called **The Random Rack** which has randomly generated data and we will see how to use the **postman.com** site to test out our server.

10.1 Express Intro

At this point in the course, we can build useful web applications using **Node.js**. It allows us to create a server that listens for incoming requests, examines the request **URL** and method (e.g., **GET**/**POST**), and sends back the appropriate response. However, as our site grows (i.e., more pages, form data, files, headers, etc..) the code can quickly become large and difficult to manage. We have to manually parse requests, use **IF** or **SWITCH** statements to route them, and handle responses by setting headers and converting data to **JSON** by hand. We also need to write our own logic for parsing form data and request bodies. All of this becomes repetitive and tedious. Thankfully, there is a useful **Node.js** module that can simplify our lives:

> ***Express** = a minimal and flexible Node.js framework module that simplifies building web servers by handling routing, requests, and responses with less code and better structure.*

It doesn't give us abilities beyond what we can already do, but it saves us time, reduces repetition, and helps us organize code better. Some key features are:

- **Routing:** Handles different **URL**s and **HTTP** methods easily.
- **Request/Response Helpers:** Simplifies sending responses and accessing request data.
- **Middleware:** Lets us add reusable functions to process requests.
- **Static File Serving:** Easily serves **CSS**, images, and **JavaScript** files.
- **Modular Structure:** Helps organize large projects into smaller parts.

To use **Express**, we install it using **NPM**: `npm install express`
Then we use `require()` in our server code: `const express = require("express");`
Finally, we create a new **Express** application: `const app = express();`

This `app` object represents our web server. It is the main tool to build and control our website. Let's compare the simplest server using **express** vs. (**http** as we did before):

Using **Express**:

```
const express = require("express");
const app = express();
const port = 3000;



// Handles GET request to main page
app.get("/", function(req, res) {
   res.send("I received your request!")
});



app.listen(port);
console.log(`Server listening at http://localhost:${port}`);
```

Using **HTTP**:

```
const http = require("http");
const port = 3000;

// Handles GET request to main page
const server = http.createServer(function (req, res) {
   if (req.method === "GET" && req.url === "/") {
      res.writeHead(200, { "Content-Type": "text/plain" });
      res.end("I received your request!");
   } else {
      res.writeHead(404);
      res.end("Not Found");
   }
});

server.listen(port);
console.log(`Server listening at http://localhost:${port}`);
```

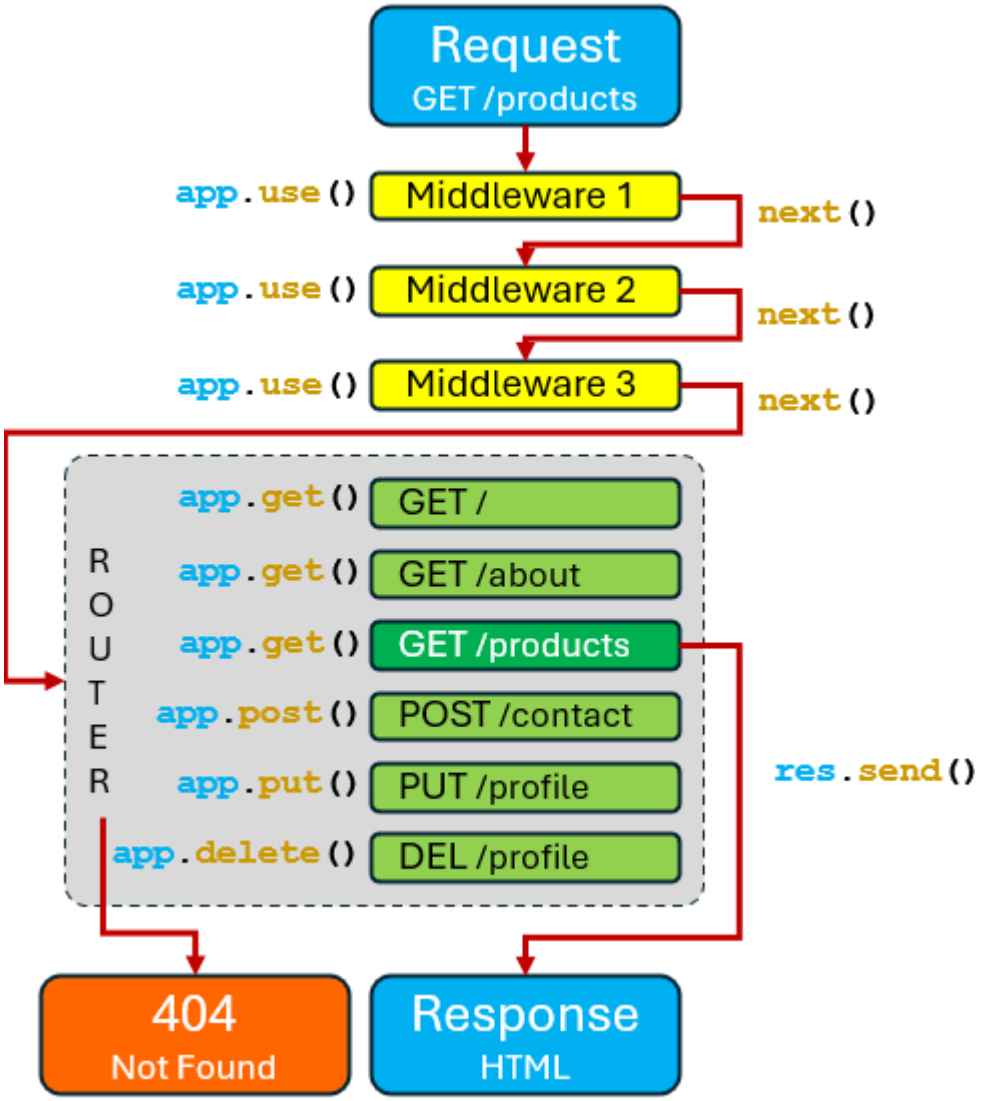
Notice that we don't need to call `http.createServer()` since the `app` is our created server. Notice the `"GET"` request is handled by its own function now. We can easily see that **Express** gets rid of quite a few of those tedious details.

Express follows what is called a …

> ***Request-Response Cycle** = the full sequence of steps that occur from the moment a client (e.g., browser) makes a request to the moment the server processes it and sends back a response.*

This cycle includes these main steps:

1. A request comes in from the browser.
2. (optional) Middleware functions are evaluated (e.g., parsing, logging, auth).
3. A router calls the function that matches the path and method of the request. If none found, **404 Not Found** is sent back.
4. The matched route handler function is evaluated.
5. The route handler sends a response (with `res.send()`, `res.render()`, etc.)



We will first look at how the routing works and then discuss the optional middleware. We will also discuss how the function calls work (i.e., the ones we see in the above diagram).

The ROUTER in the diagram above, is an internal **Express** mechanism for …

> ***Routing** = the process of matching an incoming request to a specific piece of code that handles it.*

A route is a specific URL path and an HTTP request method to which a callback function is assigned.

When someone visits our website and types a **URL** like this: **http://futuretech/products** ... the server uses routing to figure to decide which code should run and what response to send back. A route's callback function will have request and response parameters, as well as a next parameter.

The `app` server object has a unique function for each of the request methods (i.e., `get()`, `post()`, `put()`, `delete()`). We will write a bunch of these types of function calls to inform **Express** how we would like to deal with particular incoming requests (see inside ROUTER in the image above). Here is the typical format for these function call specifications:

```
app.<method>(<path>, function(req, res) {
    ...
});
```

Here, `<method>` is either `get`, `post`, `put` or `delete` (although there are other options available that we will not discuss). This method name will be used during routing as part of the matching process. **Express** will look at the incoming request's method and see which function matches. However, when doing the matching, it will also look at the `<path>` parameter as part of the matching process. The 2nd parameter lets us specify the function that will get called when **Express** finds an incoming request that matches the `<method>` and `<path>`. This callback function that we supply, should take at least a `req` (i.e., incoming request) and `res` (i.e., outgoing response) parameter.

Let's refer back to our **FutureTech Corp.** website. Here is how we might set up the **routes** (using either function or arrow notation):

```
app.get("/", function (req, res) {
    /* send back the Home page */
});
app.get("/about", function (req, res) {
    /* send back the About page */
});
app.get("/contact", function (req, res) {
    /* send back the Contact Us page */
});
app.get("/products", function (req, res) {
    /* send back the Products page */
});
app.get("/product", function (req, res) {
    /* send back the Product page */
});
```

```
app.get("/", (req, res) => {
    /* send back the Home page */
});
app.get("/about", (req, res) => {
    /* send back the About page */
});
app.get("/contact", (req, res) => {
    /* send back the Contact Us page */
});
app.get("/products", (req, res) => {
    /* send back the Products page */
});
app.get("/product", (req, res) => {
    /* send back the Product page */
});
```

What goes into the callback functions? Well, to send back static content such as plain text, **JSON**, or pre-made **HTML** strings, we would use the `send()` function as follows:

```
res.send("FutureTech is a cool company.");  // Just sending a simple string
```

If we have some pre-made static/fixed **HTML** files we use `sendFile()` instead:

```
res.sendFile(__dirname + "/views/about.html");
```

`sendFile()` requires the absolute path to the file we want to send and in the code above, we are getting the file relative to the server's file location `__dirname`. However, the above code does not use OS-independent file naming. So, this is a bad approach. Instead, as we did previously, we would want to use the `path` module and ruse `join()` to piece together the absolute path based on the proper file separator for that OS:

```
const path = require("path");
res.sendFile(path.join(__dirname, "views", "about.html"));
```

The `join()` function in the `path` module will take the root directory of the server and then append the `views` folder and then the filename.

Both the `app.send()` and `app.sendFile()` send back a `200 OK` status by default.

As we discussed before, when it comes to the **POST**, **PUT** & **DELETE** methods, they make use of a **body**, which contains additional information that comes along as part of the request. This information is often a **JSON** object and sometimes **<form>** data. In the case of `app.post()`, the **body** usually contains information needed to create something (e.g., a new product or user). For an `app.put()`, the **body** usually contains information needed to update a record or a resource and an `app.delete()` has a body that may include an **id** or metadata.The point is, there is some work to do before sending back a response.

Here is what it might look like for a simple **POST** or **PUT** request:

```
app.post("/profile", (req, res) => {
    const newMember = req.body;
    members.push(newMember);              // assumes members array exists
    res.status(201).send("Member added"); // 201 Created sent back
});

app.put("/profile/:id", (req, res) => {   // :id indicates a route parameter
    members[req.params.id] = req.body;
    res.send("Member updated");           // 200 OK sent back by default
});
```

So, if we had only static pages with some client-side **JavaScript**, we could write our server like this:

```
const path = require("path");

const express = require("express");
const app = express();
const PORT = 3000;


app.get(["/", "/index.html"], (req, res) => {
    res.sendFile(path.join(__dirname, "index.html"));
});

app.get("/about.html", (req, res) => {
    res.sendFile(path.join(__dirname, "about.html"));
});
```

```
app.get("/contact.html", (req, res) => {
    res.sendFile(path.join(__dirname, "contact.html"));
});

app.get("/products.html", (req, res) => {
  res.sendFile(path.join(__dirname, "products.html"));
});

// Explicit routes for individual product pages
app.get("/products/product_01.html", (req, res) => {
  res.sendFile(path.join(__dirname, "products", "product_01.html"));
});

app.get("/products/product_02.html", (req, res) => {
  res.sendFile(path.join(__dirname, "products", "product_02.html"));
});

app.get("/products/product_03.html", (req, res) => {
  res.sendFile(path.join(__dirname, "products", "product_03.html"));
});

app.get("/products/product_04.html", (req, res) => {
  res.sendFile(path.join(__dirname, "products", "product_04.html"));
});

app.get("/products/product_05.html", (req, res) => {
  res.sendFile(path.join(__dirname, "products", "product_05.html"));
});

app.get("/products/product_06.html", (req, res) => {
  res.sendFile(path.join(__dirname, "products", "product_06.html"));
});

// Start server
app.listen(PORT);
console.log(`Server is listening at http://localhost:${PORT}`);
```

```
futuretech-site
├── icons
│   └── logo-icon.png
├── images
│   ├── large
│   │   ├── cloakingSuit.jpg
│   │   ├── jetpack.jpg
│   │   ├── neuralBooster.jpg
│   │   ├── phasingSuit.jpg
│   │   ├── portalTransporter.jpg
│   │   └── timeMachine.jpg
│   ├── news
│   │   ├── android.jpg
│   │   ├── portals.jpg
│   │   ├── robotics.jpg
│   │   └── scientists.jpg
│   ├── small
│   │   ├── cloakingSuit_small.jpg
│   │   ├── jetpack_small.jpg
│   │   ├── neuralBooster_small.jpg
│   │   ├── phasingSuit_small.jpg
│   │   ├── portalTransporter_small.jpg
│   │   └── timeMachine_small.jpg
│   ├── headquarters.jpg
│   ├── logo.jpg
│   ├── logo_short.jpg
│   └── topview.jpg
├── products
│   ├── product_01.html
│   ├── product_02.html
│   ├── product_03.html
│   ├── product_04.html
│   ├── product_05.html
│   └── product_06.html
├── scripts
│   ├── captions.js
│   ├── dark-mode.js
│   ├── quotes.js
│   ├── slider-script.js
│   └── time-clock.js
├── styles
│   ├── details.css
│   ├── general-body.css
│   ├── header-footer.css
│   ├── image-slider.css
│   └── products-style.css
├── index.html
├── about.html
├── contact.html
├── products.html
└── server.js
```

This would send all the **HTML** files, but it would not send any of the images, icons, **CSS** stylesheets nor **JavaScript** files. We could make an `app.get()` call for each of the **41** files we see above, but that seems excessive!

As it turns out, **Express** has a useful function for sending only static pages. Here is a working **Express** server that handles all files in one single line:

```
const express = require("express");
const app = express();
const PORT = 3000;

app.use(express.static(__dirname));  // Wow! One line handles it all!

// Start server
app.listen(PORT);
console.log(`Server is listening at http://localhost:${PORT}`);
```

The `app.use()` function is used to tell **Express** that we want to use some …

> ***Middleware* = *a function that intercepts, processes, or handles a request before sending a response (or passing control forward).***

As it turns out, `express.static()` is a built-in middleware function in **Express** that serves any static files (e.g., **html**, **css**, **js**, **jpg**, **png**, etc..). So, this one line of code tells **Express** that when a request comes in, we want it to check the specified folder (and its subfolders) for a matching file and serve it. Since we passed in the absolute path to current folder (i.e., `__dirname`), it will serve all our files. It really cannot get any easier than this.

This is so much shorter than this code that we were using before:

```
const http = require("http");
const fs = require("fs");
const path = require("path");

const PORT = 3000;
const mimeTypes = {
  ".html": "text/html",
  ".css":  "text/css",
  ".js":   "application/javascript",
  ".png":  "image/png",
  ".jpg":  "image/jpeg",
  ".ico":  "image/x-icon"
};

function requestListener(req, res) {
  let filePath = req.url === "/" ? "/index.html" : req.url; // add index.html
  filePath = path.join(__dirname, filePath);                // get the absolute path to the file

  let ext = path.extname(filePath);                         // get the file extension
  let contentType = mimeTypes[ext]                          // lookup content type based on ext
                        ||  "application/octet-stream";     // if not there, treat as download

  fs.readFile(filePath, (err, data) => {                    // read file, set data to contents
    if (err) {                                              // if error, return 404 Not Found
      res.writeHead(404, { "Content-Type": "text/plain" });
      return res.end("404 Not Found");
    }
    res.writeHead(200, { "Content-Type": contentType });    // otherwise send 200 OK and the data
    res.end(data);
  });
}

http.createServer(requestListener).listen(PORT);
console.log(`Server running at http://localhost:${PORT}`);
```

The order of our `app` calls and other routing or middleware methods in an **Express** server is important because **Express** processes incoming requests in the exact order that the routes and middleware are defined. That is, when a request comes in, **Express** starts at the top of our server file and checks each route (or middleware) in order. As soon as it finds a match, it runs the corresponding callback. If nothing matches, it moves to the next one.

Middleware functions (e.g., body parsers, static file servers, custom loggers, or authentication checks) need to be defined before the routes that use them. So, if we call `app.get(route)` before `app.use(middleware)`, that `route` won't have access to the `middleware`. It is like it never existed for that route.

Route matching in **Express** follows a **top-down** or **"first match wins"** approach. That means **Express** checks each route in the order we defined it, and once it finds a match, it stops looking.

Consider this example:

```
app.get("/products", (req, res) => {
    res.send("All products");
});

app.get("/products/:id", (req, res) => {
    res.send(`Product ID: ${req.params.id}`);
});
```

A request to `/products` matches the first route, because it is an exact match. A request to `/products/6` does not match the first route (because ethe first one does not expect an id), so it continues and matches the second route, where **id** becomes **6**. In this case, the **order doesn't matter** because the two paths are clearly different. Now consider this example:

```
app.get("/products/:id", (req, res) => {
    res.send(`Product ID: ${req.params.id}`);
});

app.get("/products/:name", (req, res) => {
    res.send(`Product Name: ${req.params.name}`);
});
```

Here, both routes match any `/products/xxx` pattern … whether it's a number or a name. So, a request to `/products` matches neither because there is no value for `id` or `name`. A request to `/products/6` matches the first route, which has an **id** as expected. But a request to `/products/bob` also matches the first route because **Express** doesn't know the difference between an **id** and a **name** just from the **URL**. This means the second route will never be reached, no matter what value is passed, because the first one already catches all cases. It is best to avoid defining multiple conflicting dynamic segments at the same level like this.

It is important use more specific routes first (e.g., `/products/details/` before `/products/:id`) because **Express** matches routes based on the number and position of path segments, not the parameter names … and it uses the first matching route it finds.

So, there is a problem here:

```
app.get("/products/:id", (req, res) => {
    res.send(`Product ID: ${req.params.id}`);
});

app.get("/products/details/", (req, res) => {
    res.send("Product Info");
});
```

A request to **`/products/6`** correctly matches the first route. However, a request to **`/products/details`** also matches the first route, because **Express** sees **details** as a value for **id** since the number of path segments is the same. As a result, the second route is never reached. To fix this, the more specific route (**`/products/details`**) should be placed before the general route (**`/products/:id`**).

We can attach multiple callback functions to a route so that they run in sequence when the route is matched:

```
app.<method>(<path>,
      function1(req, res, next){},
      function2(req, res, next){},
      function3(req, res, next){},
      …
});
```

Each callback behaves like middleware and can perform tasks such as logging, validation, authentication, etc. Every function receives three arguments: **`req`**, **`res`**, and **`next`**. Calling **`next()`** passes control to the next function in the chain.

One last point … we can leave out the **`<path>`** parameter in **`app.get()`**, **`app.post()`** etc.:

```
app.<method>(function(req, res, next){});

app.<method>(
      function(req, res, next){},
      function(req, res, next){},
      function(req, res, next){},
      …
});
```

When we do this, the route will match **any** path as long as the **HTTP** method matches (**GET**, **POST**, etc.). It acts almost like middleware that runs for all **URL**s of that method.

10.2 More About Middleware

We saw in the previous section, that `app.use()` lets us tell **Express** to apply some middleware after it receives a request but before it returns a response. In **Express**, when a request comes into the server, it doesn't just go straight to a route handler. Instead, it can pass through a chain of middleware functions first. Middleware functions are like steps in an assembly line. Each one can do something useful with the request (like logging, authentication, or parsing data) and then pass it along to the next step.

When we call `app.use()`, we are merely telling **Express** to register this middleware function to be executed in the request-response cycle. It tells **Express** to run a given middleware function every time a request comes in (unless it is limited to certain paths). The middleware functions are executed in the order that they are listed from top-to-bottom in the server code.

Middleware functions have access to the **request** object, the **response** object, and the **next** middleware function in the application's request-response cycle. This means they can read or change the request, prepare a response, end the request-response cycle or pass control to the next function in the chain … ultimately leading to a final response being sent.

If the current middleware function does not end the request-response cycle (i.e., by sending back a response to the browser), it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging … which means that the browser will not get a reply from the server and will sit their waiting … eventually timing out.

Just as with the route functions (e.g., `app.get()`, `app.post()`), a call to `app.use()` can have a variety of parameters:

```
// Register middleware function for all paths
app.use(midFunc)

// Register multiple middleware functions for all paths
app.use(midFunc1, midFunc2, midFunc3, … )

// Register middleware function for a specific path (and its subpaths)
app.use(<path>, midFunc)

// Register multiple middleware functions for a specific path (and its subpaths)
app.use(<path>, midFunc1, midFunc2, midFunc3, …)
```

There are three categories of middleware:

1. **Built-in** - included with **Express**
   - `express.static()` - serves static files (e.g., **.css**, **.jpg**, **.js**)
   - `express.json()` - parses incoming **JSON** objects (e.g., `{ "name": "Bob", "age": 30 }`)
   - `express.urlencoded()` - parses **URL**-encoded data (e.g., form submissions)

2. **Third-Party** - hundreds available, installed from **NPM**
   - **morgan** - logs **HTTP** request information
   - **cookie-parser** - parses the cookie header in an **HTTP** request
   - **cors** - handles Cross-Origin Resource Sharing
   - **helmet** - sets security-related **HTTP** headers
   - **express-session** - manages user sessions data on the server
   - **body-parser** - does legacy body parsing (now built into **Express**)
   - **errorHandler** - helps with debugging during development
   - **csurf** - protects against cross-site request forgery (CSRF)
   - **compression** - compresses response bodies

3. **Custom** - functions that we write ourselves to …
   - add properties to `req` or `res`
   - check authentication
   - log requests
   - block certain IPs, etc.

Let's talk a little about writing our own **custom** middleware. Looking back at our static version of the **FutureTech Corp.** site, we had a simple server that served the static files with one line. Let's add some middleware to do three things:

- log some information for the incoming requests
- log the date and time
- log the number of requests handled so far

Later, we will see how to do other interesting things other than log information, but these simple examples should be sufficient as proof-of-concept.

Here is the code that we will add to our **FutureTech Corp.** **expressServer.js**:

```
const express = require("express");
const app = express();
const PORT = 3000;

// Log some information for any requests
app.use(function(req, res, next) {
    console.log(req.method);
    console.log(req.url);
    console.log(req.query);
    console.log("Body:  ", req.body); // req.body is a JavaScript object
    next();  // go to the next registered handler/middleware
});

// Log the date and time
app.use(function(req, res, next) {
    const now = new Date();
    console.log(`[${now.toLocaleString()}] ${req.method} ${req.url}`);
    next(); // go to the next registered handler/middleware
});
```

```
// Log the number of requests handled so far
let requestCount = 0;
app.use(function(req, res, next) {
    requestCount++;
    console.log("Request count = " + requestCount);
    next(); // go to the next registered handler/middleware
});

// Serve all files
app.use(express.static(__dirname));

// Start server
app.listen(PORT);
console.log(`Server is listening at http://localhost:${PORT}`);
```

Notice that we are making a call to `next()` at the end of each of our middleware functions that we created. This will tell **Express** to evaluate all three of these middleware functions in sequence, followed by the final built-in middleware function to serve the static files. Can you answer these questions:

1. What would happen if we forgot one of the calls to `next()`?
2. What would happen if I moved the middleware that logs the date & time to appear AFTER the middleware that serves static files?
3. Can we combine our three middleware functions with one call to `use()` as shown below?
4. Can we also combine the static-serving middleware function in there as well?

```
let requestCount = 0;
app.use(
    function(req, res, next) { // Log some information for any requests
        console.log(req.method);
        console.log(req.url);
        console.log(req.query);
        console.log("Body:  ", req.body); // req.body is a JavaScript object
        next();  // go to the next registered handler/middleware
    },
    function(req, res, next) { // Log the date and time
        const now = new Date();
        console.log(`[${now.toLocaleString()}] ${req.method} ${req.url}`);
        next(); // go to the next registered handler/middleware
    },

    function(req, res, next) { // Log the number of requests handled so far
        requestCount++;
        console.log("Request count = " + requestCount);
     next(); // go to the next registered handler/middleware
    }
);
```

5. Why not combine all of these middleware functions into one function as shown below?

```
let requestCount = 0;
app.use(
    function(req, res, next) { // Log some information for any requests
        console.log(req.method);
        console.log(req.url);
        console.log(req.query);
        console.log("Body:  ", req.body); // req.body is a JavaScript object

        const now = new Date();
        console.log(`[${now.toLocaleString()}] ${req.method} ${req.url}`);

        requestCount++;
        console.log("Request count = " + requestCount);
    }
);
```

As mentioned earlier, we are doing some basic things in our middleware by just logging some information. As it turn out, somebody already realized that this kind of middleware can be useful for development. The third-party middleware known as **morgan**, can do this for us, so that we do not have to do this on our own.

To use **morgan**, we install it using **NPM**: `npm install morgan`
Then we use require in our server code: `const morgan = require("morgan");`
Then we just do this to use it:

```
// Log some information for any requests
app.use(morgan("dev"));
```

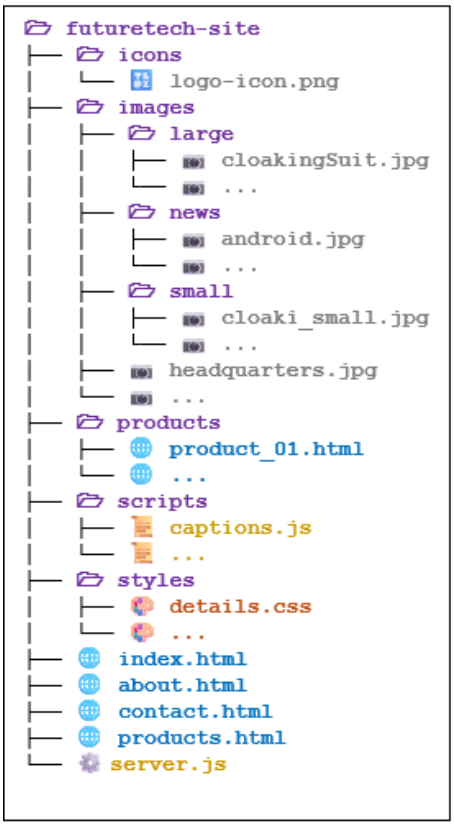
On our **FutureTech Corp** site we would see this logged with this `"dev"` option

```
GET / 304 3.951 ms - -
GET /styles/image-slider.css 304 0.671 ms - -
GET /styles/general-body.css 304 0.521 ms - -
GET /styles/header-footer.css 304 1.306 ms - -
GET /images/logo.jpg 304 1.796 ms - -
GET /images/news/scientists.jpg 304 2.225 ms - -
```
etc..

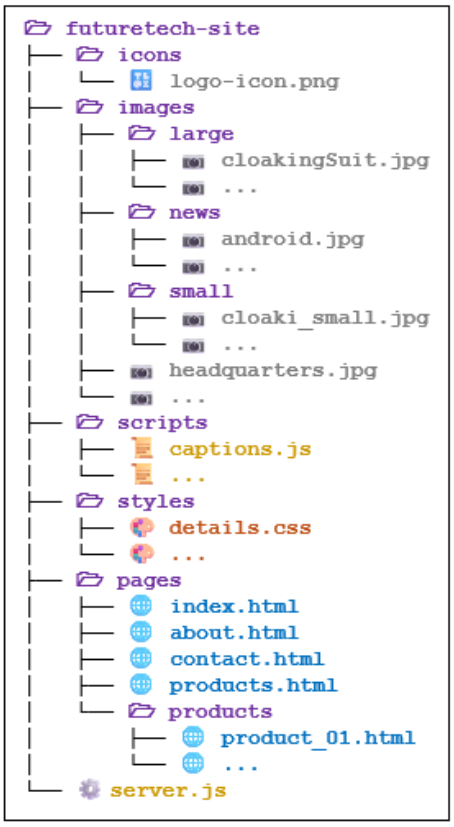
The `"dev"` parameter just specifies one format. There are a few options available:

- `"dev"` - concise colored output (great for dev)
- `"tiny"` - minimal output
- `"combined"` - standard Apache combined log format
- `"common"` - standard Apache common log format
- we can define our own format

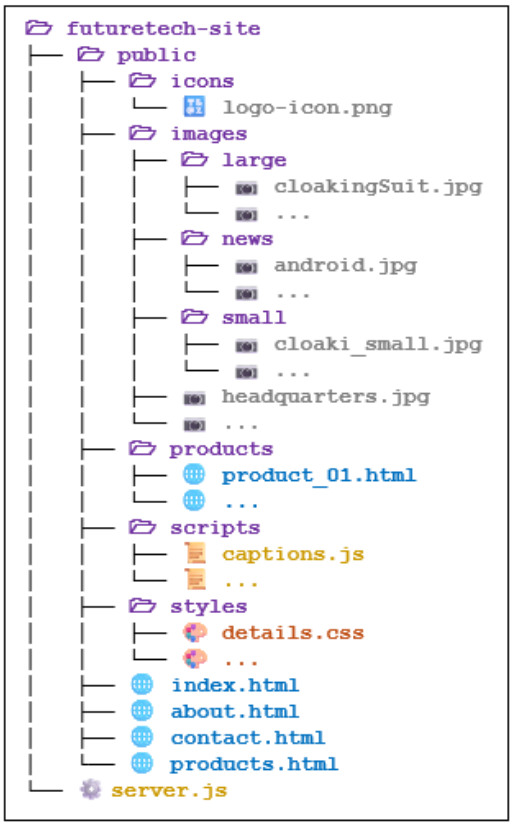
Now, let's talk a little more about directory structure and serving static files. Consider these three directory structures for our static **FutureTech Corp.** website:



(A)



(B)



(C)

As it turns out, we have to make some slight changes in our code to serve the files so that the path structure matches. We saw originally, that for structure **(A)**, we used this code:

```
app.use(express.static(__dirname));
```

This worked when we went to **http://localhost:3000** because the **index.html** file that we want to serve is in the same folder as the server. But for structure (B) there is no **index.html** file in the folder that the server is in. So, we would have to change the web address to **http://localhost:3000/pages** because the **index.html** file is in the **pages** subfolder.

Alternatively, we could add a separate middleware function to handle the **HTML** pages by specifying that they are in the **pages** subfolder:

```
// Serve all static files from the server's current folder and inward
app.use(express.static(__dirname));

// Serve static pages from the pages subfolder
app.use(express.static(path.join(__dirname, "pages")));
```

Structure **(C)** is the easiest of all. Since we put all the webpage files in a public folder at the same level as the server code, we can just do this … without needing to know the **path** nor current working directory:

```
app.use(express.static("public"));
```

Here, public is assumed to be a subfolder of the folder in which we launched the server using **node.js**.

As our site grows, we will often need to serve static files (e.g., images, stylesheets, scripts). These files are commonly accessed directly from folders such as **`/images`**, **`/styles`**, **`/scripts`** etc.. When we serve such static files directly from folders named **`/images`** or **`/styles`**, those paths become part of our website's **URL** structure (e.g., an image might be accessible at **`/images/logo.png`** and a stylesheet at **`/styles/main.css`**).

While this works fine for small sites, as our application grows and we add many dynamic routes (e.g., **`/about`**, **`/contact`**, **`/products`**, etc..), having many static asset paths directly under the root can make the **URL** space crowded and harder to manage. This clutter can cause confusion or conflicts if a future dynamic route uses the same path segment (e.g., if we later want a route called **`/images`** to show an image gallery page, it will clash with our static **`/images`** folder).

To avoid this, **Express** lets us define a ***virtual path prefix*** … which is a custom **URL** prefix that doesn't exist on our file system but helps organize how static files are accessed.

For example, using a virtual path (such as **`/static`**) allows us to load our **CSS** file from **`/static/styles/header-footer.css`** instead of **`/styles/header-footer.css`**, keeping our **URL**s clean, predictable, and separate from our main routes.

If we want the browser to specify a virtual path prefix (i.e., one that does not really exist in the file system), we can do this:

```
app.use("/static", express.static("public"));
```

Then, to access the pages, we would use **http://localhost:3000/static** as our starting point so that we access pages like this:

http://localhost:3000/static/index.html
http://localhost:3000/static/about.html
http://localhost:3000/static/products.html
http://localhost:3000/static/products/product_02.html
http://localhost:3000/static/contact.html

10.3 Bodies and Query Strings

Looking back at our **Date-a-Science** server, this is how we handled requests before:

```
// Listener for incoming client requests
function handleRequest(req, res) {
    const parsed = url.parse(req.url, true); // get the query params (for profile pages)
    const pathname = parsed.pathname;        // get path without query

    if (req.method === "GET") {
        if (pathname === "/" || pathname === "/home") {// Return the main page
            ...
        } else if (pathname.startsWith("/profile")) {  // Return a specific profile
            ...
        } else if (pathname.startsWith("/about")) {    // Return the about page
            ...
        } else if (pathname === "/clientscript.js") {  // Return the client-side javascript
            ...
        } else if (pathname.startsWith("/images/")) {  // Return an image for a page
            ...
        } else if (pathname === "/style.css") {        // Return the style file
            ...
        } else { // If anything else ... respond with 404 error
            ...
        }

    } else if (req.method === "PUT" && pathname === "/newprofile") {
        ...
    } else if (req.method === "POST" && pathname === "/message") {
        ...
    } else {
        res.writeHead(405);
        res.end("Unsupported method");
    }
}

http.createServer(handleRequest).listen(3000);
console.log("Server running on http://localhost:3000");
```

When we convert this to **Express**, all the `"GET"`, `"PUT"` and `"POST"` handling will NOT be merged in a `handleRequest()` function since that function will no longer exist. Instead, they will each appear one after the other, with the middleware set up to serve the static files as follows:

```
app.use(express.static(__dirname));

app.get(["/", "/home"], (req, res) => {...});
app.get("/about", (req, res) => {...});
app.get("/profile", (req, res) => {...});

app.put("/newprofile", (req, res) => {...});

app.post("/message", (req, res) => {...});

app.use((req, res) => {...}); // Catch others as 404 Not Found
```

So, already, we can see the benefits of the **Express** server in the way it gets rid of clutter.

There are some simplifications as well in the code for handling the `"GET"`, `"PUT"` and `"POST"` requests. Typically, we would write code like this to send responses:

```
// Sending "Error 404: Resource not found." response
response.writeHead(404, {"Content-Type": "text/plain"});
response.end("Error 404: Resource not found.");

// Sending "200 OK" response with static file in body
res.writeHead(200, { "Content-Type": getMimeType(filePath) });
res.end(data);

// Sending "200 OK" response with html page in body
res.writeHead(200, { "Content-Type": "text/html" });
res.end(html);
```

What's nice about **Express**, is that it does a few things for us by allowing default assumptions on content type and response texts. There are different options for sending customized error responses:

```
// Sending customized "Error 404: Resource not found." response
res.status(404).send("Error 404: Resource not found.");

// Sending default "404 Not Found" response
res.sendStatus(404);

// Sends status "404" but no body ... so browser will timeout
res.status(404);   ❌
```

When it comes to sending back **200** (i.e., "ok") responses, we have a few ways to do it. If we are just sending back a response to a `"PUT"`, `"POST"` or `"DEL"` to let the browser know that all was ok, we usually send back a simple text body:

```
// sends only status, no body ... browser will timeout
res.status(200);   ❌

// sends customized text
res.status(200);
res.send("200 Got it!");

// more compact version
res.status(200).send("200 Got it!");

// easiest ... sends default "OK" text
res.sendStatus(200);
```

For requests that require a body of data to be sent back (e.g., `"GET"` requests that need an **HTML** or particular static file sent back), we typically do this in **Express** to send back the content stored in a `body` variable:

```
res.status(200).send(body);
```

Express will automatically set the **Content-Type** accordingly. Here is what it is set to based on the type of the `body` variable:

- **string** = `Content-Type: text/html`
- **buffer** = `Content-Type: application/octet-stream`
- **object** = `Content-Type: application/json`
- **array** = `Content-Type: application/json` ←same as **object**

If we want a different **Content-Type**, we can set it by calling `set()` before calling `send()`:

```
res.status(200);
res.set("Content-Type", "text/plain");
res.send(body);
```

However, if we know that we want to send a **JSON** object, then we should use the `json()` function instead of `send()` as follows:

```
res.status(200).json(body);
```

Express also makes things a little easier when accessing queries. Recall that if we specified an address something like this:

http://localhost:3000/index.html?year=2022&month=october

… then we would write code to process it like this within the appropriate **GET** route:

```
const parsed = url.parse(req.url, true); // get the query params
let month = parsed.query.month; // get the month from the query
let year = parsed.query.year;   // get the year from the query
```

But in **Express**, the query is parsed by default. So, we just need to ask the **request** for the parameter that we want:

```
let month = req.query.month; // get the month from the query
let year = req.query.year;   // get the year from the query
```

Now, lets talk about **POST** bodies. Recall this **HTML** form from our **FutureTech Corp.** site:

```
<form action="/contact/message" method="POST" target="">
   <label for="name">Name:</label><br>
      <input type="text" id="name" name="name" required><br><br>

   <label for="email">Email:</label><br>
      <input type="email" id="email" name="email" required><br><br>

   <label for="message">Message:</label><br>
      <textarea id="message" name="message" rows="5" required></textarea><br><br>

   <button type="submit">Send Message</button>
</form>
```

When this form is submitted, its data is sent in the request body as a **URL**-encoded string. **Express** provides convenient middleware that automatically parses request bodies when the `Content-Type` header is set to `application/x-www-form-urlencoded`. To use it, we include the following near the top of our server code:

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

As long as this is included, then we can simply use `req.body` to access the body as a **JavaScript** object. The code is much simpler now:

```
// Route: contact/message post
app.post("/contact/message", (req, res) => {
    messages.push(req.body);
    messageCount++;
    res.status(200).send("Your message has been received.");
});
```

So, whether the action is **GET**, **POST**, **PUT**, or **DELETE**, our code will be much simpler in **Express**.

For situations requiring a variety of actions (i.e., not just **GET** and **POST**), **Express** offers a way to combine multiple handlers into a single route definition, keeping related logic together and making the code more organized. This approach simplifies route management by grouping all **HTTP** methods for a resource in one place, making our code easier to read and maintain.

For example, suppose we have a web **API** that manages books at a bookstore. We might define actions like this:

- **GET** - show me a book (available to all clients)
- **POST** - add a book (available to all clients)
- **PUT** - replace/update this book (for authorized clients only)
- **DELETE** - remove a book (for authorized clients only)

We could use the `route` function as shown on the left below (on the right is a comparison of what we would do without the `route` function). It is better because we don't need to re-write the `"/book"` route each time and we save writing out `app` and the use of a few semicolons.

```
app.route("/book")
    .get((req, res) => {
        // Retrieve & return a book
    })
    .post((req, res) => {
        // Add a new book
    })
    .put((req, res) => {
        // Update an existing book
    })
    .delete((req, res) => {
        // Delete a book
    });
```

```
app.get("/book", (req, res) => {
    // Retrieve and return a book
});
app.post("/book", (req, res) => {
    // Add a new book
});
app.put("/book", (req, res) => {
    // Update an existing book
});
app.delete("/book", (req, res) => {
    // Delete a book
});
```

Express Router Objects

When building a web server with **Express**, as our project grows, we will likely have many different routes to handle (e.g., routes for users, products, or contact forms). Instead of putting all our route code in one big file, **Express** lets us create ***Router*** objects. These routers act like mini-servers that group related routes together in their own files. Using routers helps keep our code organized, easier to read, and simpler to maintain, especially as our app gets bigger and more complex.

Suppose we have a server that manages information about both users and products. For example, an online store may want to show lists of products and user profiles, and also provide details about each individual product or user. At a very basic level, the server may look like this:

```
const express = require("express");
const app = express();
const PORT = 3000;


// Users routes
app.get("/users", (req, res) => {
    res.send("List of users");
});

app.get("/users/:id", (req, res) => {
    res.send(`User details for ID: ${req.params.id}`);
});


// Products routes
app.get("/products", (req, res) => {
    res.send("List of products");
});

app.get("/products/:id/size/:sz?", (req, res) => {
    const size = req.params.sz || "default size";
    res.send(`Product ${req.params.id}, Size: ${size}`);
});


// Root route
app.get("/", (req, res) => {
    res.send("Welcome to the home page!");
});


app.listen(PORT);
console.log("Server listening at http://localhost:" + PORT);
```

':id' is a placeholder (i.e. parameter) that we can access by doing this: `req.params.id`

So, for a routed request like **users/367**, `id` would be **367**

Here we use two placeholders in the route (i.e., **':id'** and **':sz?'**). So, for a routed request like **products/23789/size/large**, `id` would be **23789** and `sz` would be **'large'**. By using the **?** on the **:sz** parameter, we indicate that the size is optional. So, a routed request like **products/23789**, will match but the size will be undefined.

We could create a router object for the user-specific routes and one for the product-specific routes and place them both in a **routes/** directory.

Here is a **routes/users.js** file:

```
const express = require("express");
const router = express.Router(); // creates a router object

router.get("/", (req, res) => { // the paths are relative to where the router is mounted
    res.send("List of users");
});

router.get("/:id", (req, res) => {
    res.send(`User details for ID: ${req.params.id}`);
});

module.exports = router; // export it
```

And here is a **routes/products.js** file:

```
const express = require("express");
const router = express.Router(); // creates a router object

router.get("/", (req, res) => { // the paths are relative to where the router is mounted
    res.send("List of products");
});

router.get("/:id/size/:sz?", (req, res) => {
    const size = req.params.sz || "default size";
    res.send(`Product ${req.params.id}, Size: ${size}`);
});

module.exports = router; // export it
```

Then we can use these routers in a simplified server file as follows:

```
const express = require("express");
const app = express();
const PORT = 3000;

const userRouter = require("./routes/users");       // import users router
const productRouter = require("./routes/products"); // import products router

// Mount routers on specific URL prefixes
app.use("/users", userRouter);        // route everything in the /users path
app.use("/products", productRouter);  // route everything in the /products path

// Root route
app.get("/", (req, res) => {
    res.send("Welcome to the home page!");
});

app.listen(PORT);
console.log(`Server is listening at http://localhost:${PORT}`);
```

When someone visits **/users** or **/users/123**, they are handled by the `userRouter`, while visits to **/products**, **/products/43423** or **/products/43423/size/small** are handled by the `productRouter`. The routers keep all user-related routes in one file and product-related routes in another … making our app nice and organized.

Express Response Types

When creating a website, the same route might receive requests from different types of clients, each expecting to receive a different type of response:

- A web browser expects an **HTML** response that it can display.
- A mobile app might want **JSON** it can parse into native UI.
- A command-line script might want plain text for logging.

Instead of creating three separate routes (i.e., **/product/html**, **/product/json** and **/product/text**), **Express** has a function called `format()` that lets us serve all versions from the same route, automatically picking the right one based on the client's **Accept** header. It's like a waiter who says: "Would you like your meal in a plate, a takeout box, or a cup?"

We use the `format()` function within a route listener as shown below. We simply specify the **MIME** format, followed by a colon **:** and then an anonymous function that returns the specific reply … which could be **HTML**, a **JSON** object or a plain text, for example.

```
app.get("/products/:id/size/:sz", (req, res) => {
    const product = {
        id: req.params.id,
        name: "Neural Booster",
        size: req.params.sz
    };
    res.format({
        "text/html": () => {        // for a browser
            res.send(`
                <h1>${product.name}</h1>
                <p>ID: ${product.id}</p>
                <p>Size: ${product.size}</p>
            `);
        },
        "application/json": () => { // for a mobile app
            res.json(product);
        },
        "text/plain": () => {       // for a command-line-interface
            res.send(`${product.name} (ID: ${product.id}) - Size: ${product.size}`);
        },
        default: () => { res.status(406).send("Not Acceptable"); }
    });
});
```

With the above code, a visit to **/products/43423/size/large** we would get a different reply (as shown below) depending on the device and what is in the **Accept** header of the **HTTP** request:

Browser	Mobile App	Terminal Window
`Accept: text/html`	`Accept: application/json`	`Accept: text/plain`

Neural Booster
ID: **43423**
Size: **Large**
	`{`
	`  "id": "43423",`
	`  "name": "Neural Booster",`
	`  "size": "Large"`
	`}`
		`Neural Booster (ID: 43423) - Size: Large`

Express Shared Data

In many web applications and websites, it's important to assign unique identifiers (IDs) to new users, products, orders, or other items as they are created. For example, the first user might receive ID 1, the next user ID 2, and so on. The same goes for products or orders. However, if our app restarts or is used by many visitors over time, it must remember the last ID it assigned so that it can continue incrementing from that number without accidentally reusing an ID. If it doesn't, the app risks assigning duplicate IDs to different users or products, which can lead to bugs, data corruption, and confusion.

Where do we keep this information? If we just keep it in a variable inside our code, it resets every time we restart the server … so that doesn't work. Ideally, we can keep this in a database … but that can be complicated to set up. A simple and common strategy for small or mid-sized apps is to keep this info in a **JSON** file, which is commonly called **config.json**.

Let's consider creating a **config.json** file that contains only this **JSON** information:

```
{
    "nextUserID":72,
    "nextProductID":1002213
}
```

We could then load this file each time that we need to access either of these values and then re-write it again once we increment them as follows:

```
const fs = require("fs");
const configPath = "./config.json";

// Load up the config values and return them
function loadConfig() {
    const data = fs.readFileSync(configPath, "utf-8");
    return JSON.parse(data);
}

// Re-write the config values to the file
function saveConfig(config) {
    fs.writeFileSync(configPath, JSON.stringify(config, null, 2));
}

function createProduct(req, res, next) {
    try {
        const config = loadConfig();

        let p = {
            id: config.nextProductID,
            name: req.body.name,
            size: req.body.size,
            price: req.body.price
        };
```

```
        config.nextProductID++;
        saveConfig(config);

        res.status(201).send(p);
    } catch (err) {
        next(err);
    }
}
```

We use `readFileSync()` and `writeFileSync()` so that there are no race conditions (i.e., other threads trying to read or change these values while we are as well). There are extra parameters on the `stringify()` function. The **null** indicates that we don't want to filter or modify any keys. The **2** indicates spacing indentation, which allows for a nicer, "spaced out" printing in the file.

10.4 A FutureTech Corp. Express PUG Server

How do we make an express version of our **FutureTech Corp. PUG** server? Below is what we have at the top and bottom of our current **PUG** server. The red highlighted lines are no longer needed because express handles a lot of that for us:

```
const http = require("http");
const fs = require("fs");
const url = require("url");
const pug = require("pug");
const querystring = require("querystring");

const path = require("path");

const { products } = require("./data/product-data.js");
let { messages, messageCount } = require("./data/product-data.js");

const PORT = 3000;

// Handle incoming requests
function requestListener(req, res) {
    ... routing stuff is here ...
}

http.createServer(requestListener).listen(PORT);
console.log(`Server running at http://localhost:${PORT}`);
```

But we do have to add a couple of lines to get the **Express** `app` object. Then we need to tell **Express** that we will be using **PUG** as the view engine and we also need to tell it where to find the views.

We do this by setting **Express**'s `"view engine"` and `"views"` attributes:

```
const path = require("path");

const { products } = require("./data/product-data.js");
let { messages, messageCount } = require("./data/product-data.js");

const PORT = 3000;

// Get the Express object
const express = require("express");
const app = express();                          // Get the Express object
app.set("view engine", "pug");                  // Set up PUG as the view engine
app.set("views", path.join(__dirname, "views")); // Tell Express where the views are

... routing stuff is here ...

// Start server
app.listen(PORT);
console.log(`Server is listening at http://localhost:${PORT}`);
```

Notice again, that we are NOT creating the server … that was done when we called `express()`.

Now, what about the routing stuff? It is similar to what we did with the **Date-a-Science** site. We don't have the `requestListener()` function, so we just list all the routes one after another and render each page accordingly:

```
// Handle URL-encoded request queries (needed for the contact/message post requests)
app.use(express.urlencoded({ extended: false }));

// Serve static files
app.use(express.static(__dirname));

// Route: Home
app.get(["/", "/home", "/index"], (req, res) => {
    res.render("pages/index", { products, currentPage: "home" });
});

// Route: About
app.get("/about", (req, res) => {
    res.render("pages/about", { products, currentPage: "about" });
});

// Route: Contact
app.get("/contact", (req, res) => {
    res.render("pages/contact", { products, currentPage: "contact" });
});

// Route: Products
app.get("/products", (req, res) => {
    res.render("pages/products", { products, currentPage: "products" });
});

// Route: Individual Product (via query string ?id=1)
app.get("/product", (req, res) => {
    const id = parseInt(req.query.id);
    const prod = products[id - 1];

    if (!prod) {
        return res.status(404).send("Product not found");
    }

    res.render("pages/product", { prod, currentPage: "products" });
});

// Route: contact/message post
app.post("/contact/message", (req, res) => {
    messages.push(req.body);
    messageCount++;
    res.render("pages/messageReceived");
});

// 404 fallback
app.use((req, res) => {
    res.status(404).send("Not Found");
});
```

The code seems greatly simplified. We simply handle the static files, then our individual routes and we add one more middleware at the end that will get called when no routes match.

10.5 Sample Store Site and the Postman Tool

In this section we will describe an express server site that has been set up for a fake store called **The Random Rack**. Try it out, examine the code and test the **API** interface. It may give you ideas for future assignments and projects. We will also discus briefly a useful tool from http://postman.com that can help us test out our web **API**s without needing to make a whole bunch of client pages.

The **postman.com** tool is useful for directly interacting with an **API** or server without needing a front-end interface (i.e., client-side browser pages). For **Express** development, it allows us to send various types of **HTTP** requests (i.e., **GET**, **POST**, **PUT**, **DELETE**, etc.) to our server endpoints, include query parameters or request bodies, and then view the exact responses returned. This makes it easier to verify that routes are functioning correctly, test different input scenarios, and debug issues before integrating with a client-side application.

We will see how to use it in a moment, but let's describe our **The Random Rack** store site first.

The site structure is set up as shown here on the right →

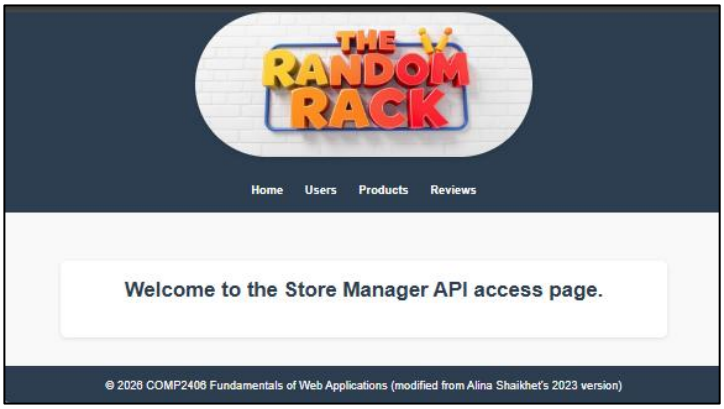
The Random Rack store

```
random-rack-store
├── images
│   └── ...
├── node_modules
│   └── ...
├── products
│   └── ...
├── reviews
│   └── ...
├── routers
│   ├── products-router.js
│   ├── reviews-router.js
│   └── user-router.js
├── styles
│   └── ...
├── users
│   └── ...
├── views
│   ├── pages
│   │   ├── index.pug
│   │   ├── product.pug
│   │   ├── products.pug
│   │   ├── review-profile.pug
│   │   ├── reviews.pug
│   │   ├── user-profile.pug
│   │   └── users.pug
│   └── partials
│       ├── footer.pug
│       └── header.pug
├── config.json
├── package.json
├── package-lock.json
├── data-generator.js
└── random-rack-express-server.js
```

The server file is **random-rack-express-server.js** which we can run with **node.js**. There is a **data-generator.js** file that needs to be run first, which generates random fake data by using an **NPM** module called ***faker***. In the given files, some random data has already been generated and is sitting in the **products**, **reviews** and **users** folders … so we should not have to run the generator.

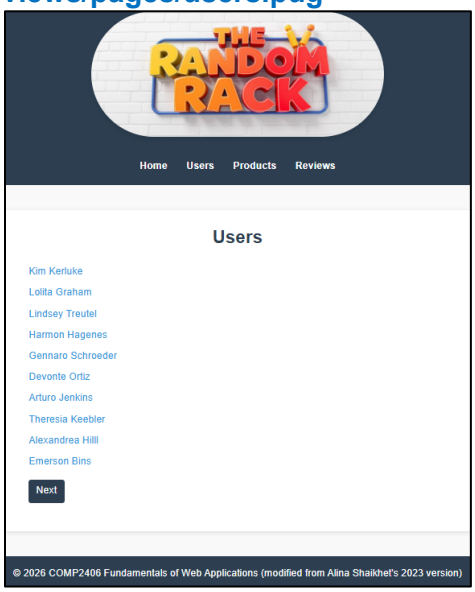
There are various also **PUG** page files as well as some **router** files.

When we run the server, the home page looks as shown below (rendered from the **views/pages/index.pug** file):

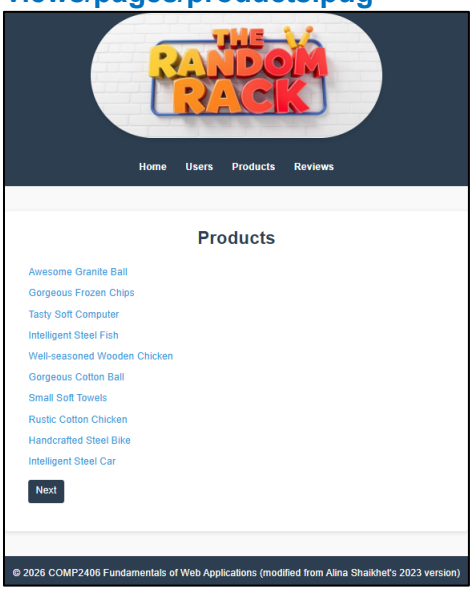


We can browse around at the data by using the navigation buttons, which will produce the pages shown on the next page. Functionality is limited, however, from these browser pages. However, through the **postman.com** site, we can test the following additional features of the site's **API**: adding a user, a product or a review … and changing a user, a product and a review. Here are the pages:

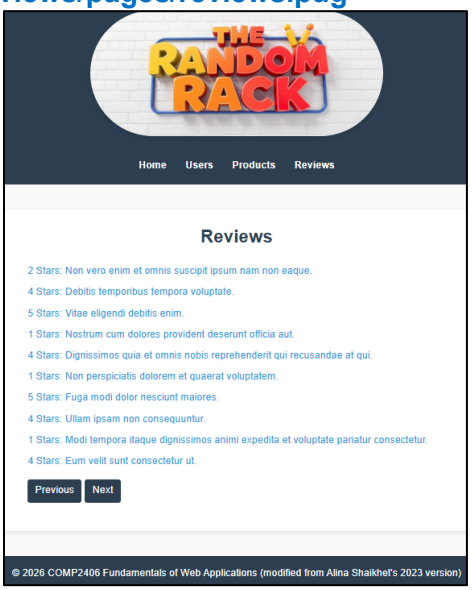
views/pages/users.pug



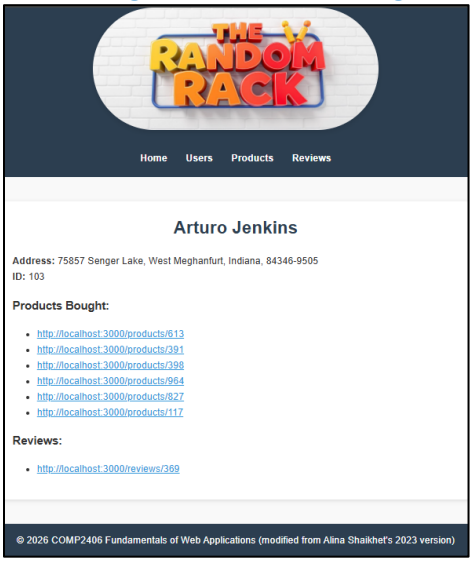
views/pages/products.pug



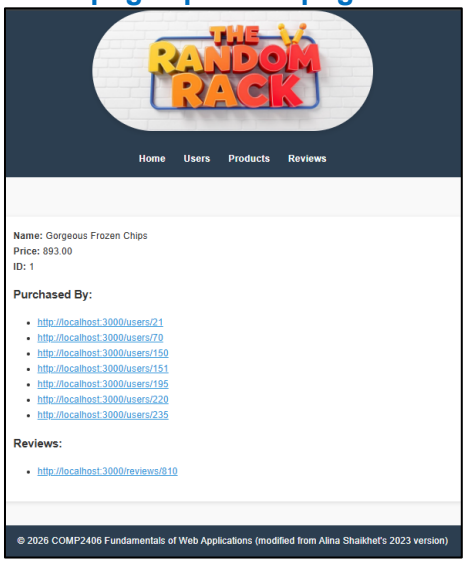
views/pages/reviews.pug



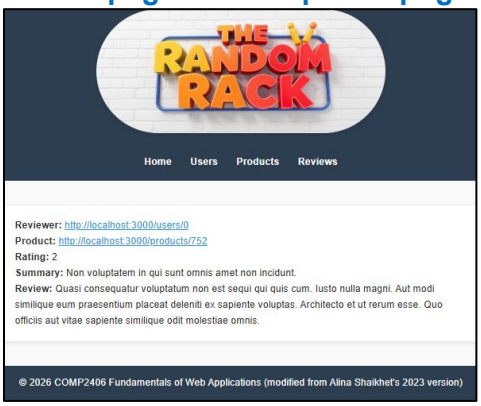
views/pages/user-profile.pug



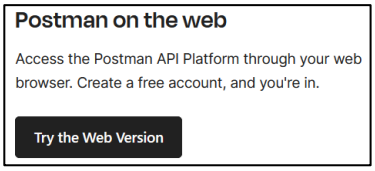
views/pages/product.pug



views/pages/review-profile.pug

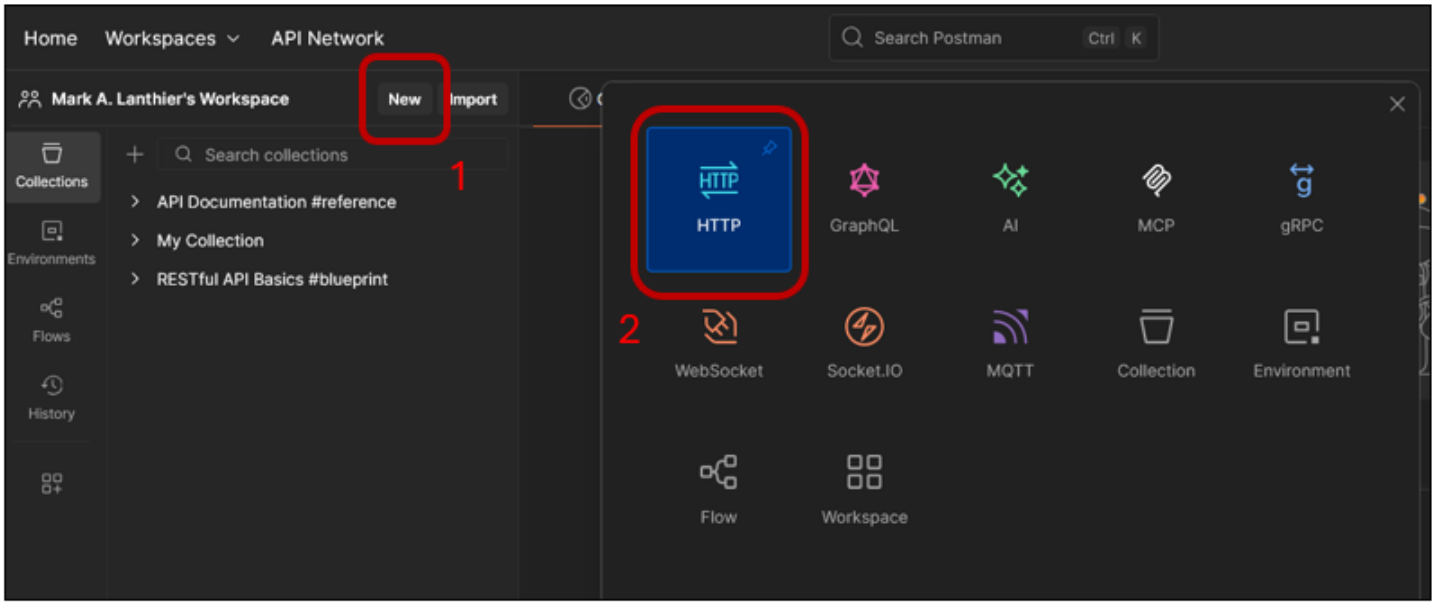


To use the postman, download it from http://postman.com . There is a simple free version. We can also try it from the web by selecting the appropriate link from the download page. Even though we use it from the web, since we need to access our site locally, we are also required to download the postman agent to our laptop/pc.

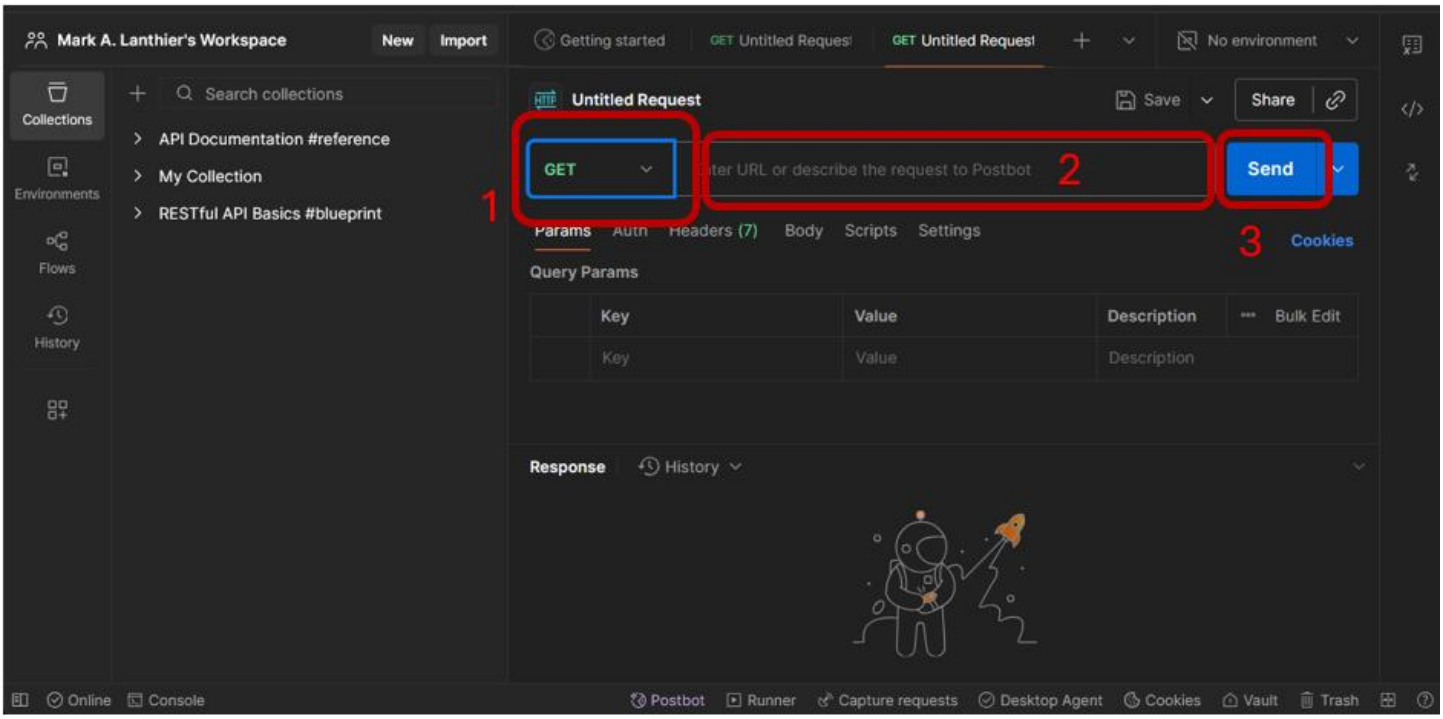


Just follow the instructions. You will need to create an account. I just did a “Sign Up with Google” to get started (strangely, it is the same link name after you sign up).

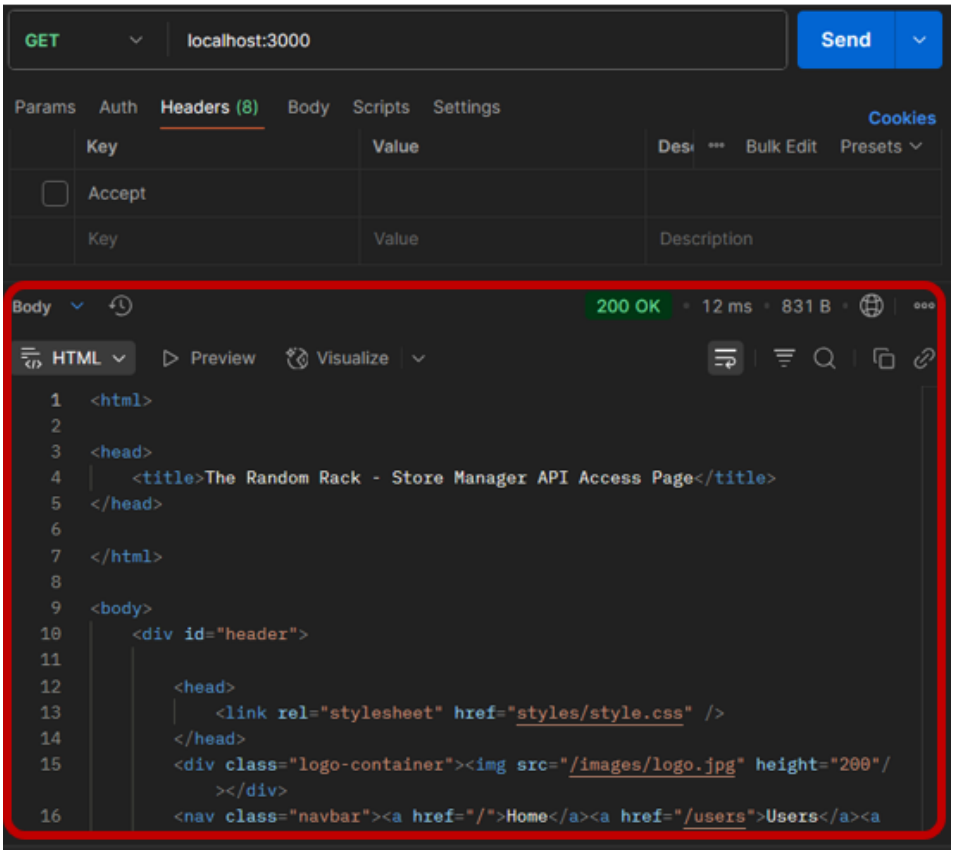
Once you log in, there is a lot of stuff everywhere. Just select **Workspaces** to your workspace (you can create one if you don't have one) … then press the **New** button and select **HTTP** as shown here:



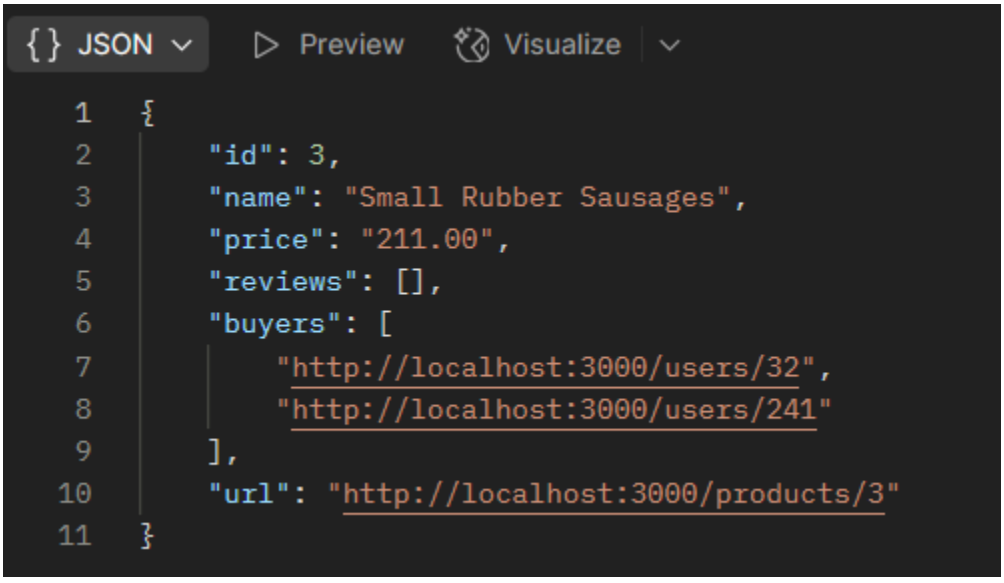
To start sending requests, first, we should make sure that our server is running already on our local host. Then select the **request type** from the dropdown box (i.e., **GET**, **POST**, **PUT**, **DELETE**), enter the **URL** in the text box and press the blue **Send** button:



For example, if we do a **GET** to **localhost:3000**, we should get the main page back:



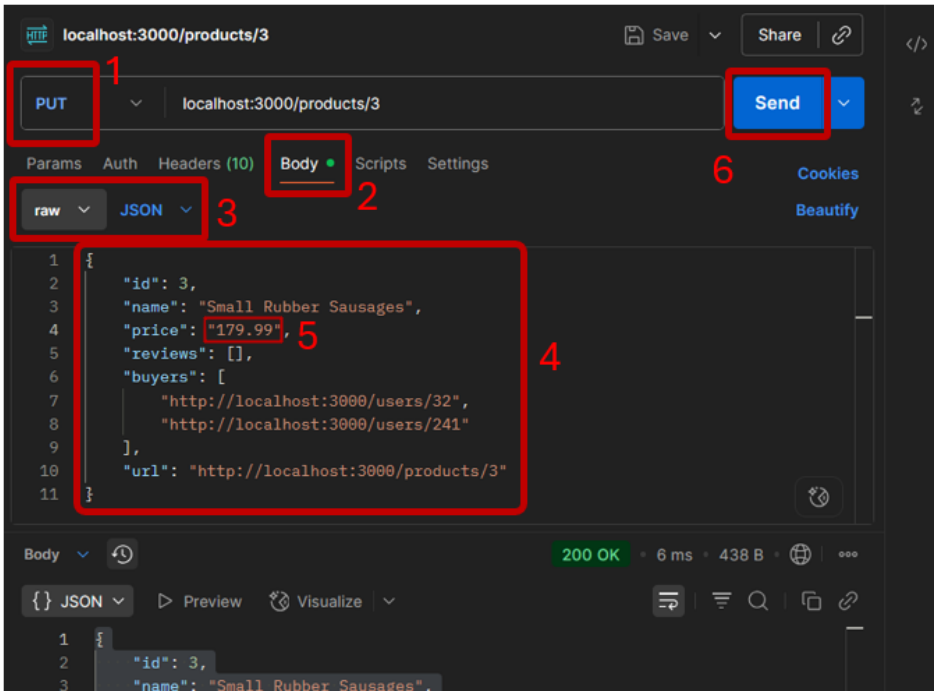
If we do a **GET** with this **URL**: **localhost:3000/products/3** we will see something like this (although our products were generated randomly, so what you see will be different):



It is a **JSON** object. We can test the changing of a product by editing this data and sending it back with a **PUT** request as follows:

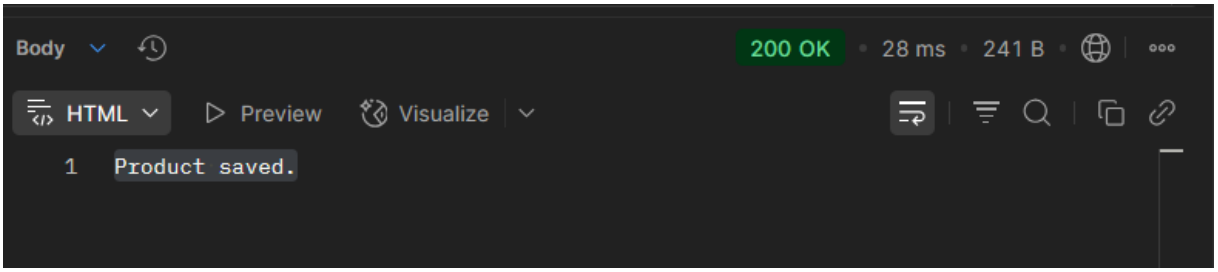
1. Set the request type to **PUT**, keep the same **URL**
2. Select the **Body** tab from the request header

3. Select **raw** from the body type
4. Copy the **JSON** object from the response body (i.e., copy from the response body we just received) into the request body.
5. Change the price (for example).
6. Press **Send**.



You should see **Product saved.** in the response:

How do we know if it worked … that we really changed the price?



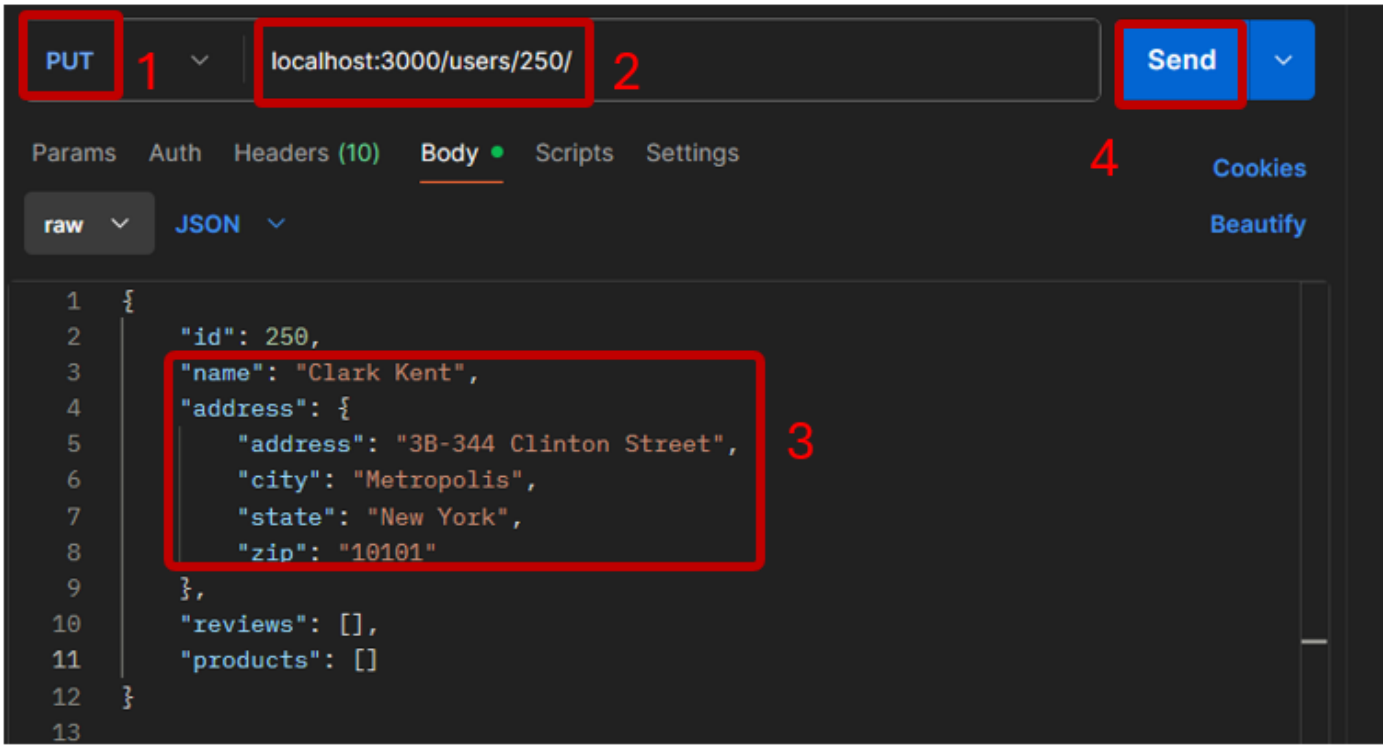
Well, do the same **GET** with **URL**: **localhost:3000/products/3** that we did before and see if the price is what we changed it to (i.e., **179.99**). We should see that it indeed worked.

Let's try adding a new user. We just need to do a **POST** to **localhost:3000/users**. Our server code will generate a new user with a new unique ID:

With the **JSON** response body data in this format (although yours may differ):

```
{
    "id": 250,
    "name": "Samara Hackett",
    "address": {
        "address": "85340 Eduardo Pines",
        "city": "Homestead",
        "state": "Iowa",
        "zip": "06462-9196"
    },
    "reviews": [],
    "products": []
}
```

Of course, we can copy this data into a **PUT** request body and alter the name and address however we would like. The steps below should result in a **User Saved.** response:



The postman can be a useful tool for testing out proper server functionality without requiring us to produce client pages/forms.