Chapter 9

Template Engines

What is in This Chapter ?

This chapter begins by discussing the need to be able to generate **dynamic HTML** and gives an example of how to do it by combining **HTML** and **JavaScript** manually. It then discusses **Template Engines** and the embedded **JavaScript** template engine called **EJS**. The manually generated **HTML** website example is then modified to use **EJS.** Code connected to this chapter is also available to examine how the **FutureTech** site can be converted to use **EJS** to allow dynamic content. We end the chapter with a discussion of the **PUG** template engine and apply it to our **FutureTech** website.

9.1 Generating Dynamic HTML with JavaScript

At this point in the course, we can make dynamic webpages on the client side, we can make basic **HTTP** servers and we can serve static **HTML**. But sometimes we need to customize a page's contents because websites often display content that changes based on user interactions, preferences, or data from a server (e.g., user profiles, product lists, blog posts, search results, etc.) We cannot realistically write static **HTML** for every possible scenario, so we need to generate **HTML** content dynamically.

Dynamically generated **HTML** makes it possible for websites to show different content to different users or change what's on the page based on live data. For example, a social media website can show your feed, messages, and notifications, while someone else sees theirs. Instead of creating hundreds of separate pages by hand, developers use templates that fill in the blanks with the right information at the right time. This keeps websites flexible, easier to manage, and more interactive.

One way to create dynamic **HTML** is to manually write out **HTML** code as plain text in **JavaScript** and add variables:

```
const user = {
    name: "Anastasia",
    age: 23,
    bio: "Loves hiking, coffee, pizza and coding."
};

let html = "";
html += "<div class='profile'>\n";          // \n is not needed but allows string to be displayed nicely
html += "  <h2>" + user.name + "</h2>\n "; // spaces at start are not needed either
html += "  <p>Age: " + user.age + "</p>\n ";
html += "  <p>Bio: " + user.bio + "</p>\n ";
html += "</div>";
```

This will produce the **HTML** code as follows:

```
<div class='profile'>
  <h2>Anastasia</h2>
  <p>Age: 23</p>
  <p>Bio: Loves hiking, coffee, pizza and coding.</p>
</div>
```

This approach works for simple cases but there are downsides:

- **Error-prone** - it's easy to forget a closing tag or misplace a variable.
- **Hard to maintain** - mixing logic and **HTML** makes code messy and difficult to read.
- **Unsafe** - can create security problems (e.g., hackers can run harmful scripts on our site)

Recall that back in chapter 4.4, we discussed the use of ***template literals*** by using the backtick character and we compared this with string concatenation using the **+** operator.

As we can see from the table below, the code is very similar, but the template literal is more readable, easier to maintain and less prone to syntax error (i.e., we are more likely to miss " or + when doing string concatenation).

With + Concatenation

```
let user = {
  name: "Anastasia",
  age: 23,
  bio: "Loves hiking, coffee, pizza & coding."
};

let html =
  "<div class=\"profile\">" +
    "<h2>" + user.name + "</h2>" +
    "<p>Age: " + user.age + "</p>" +
    "<p>Bio: " + user.bio + "</p>" +
  "</div>";
```

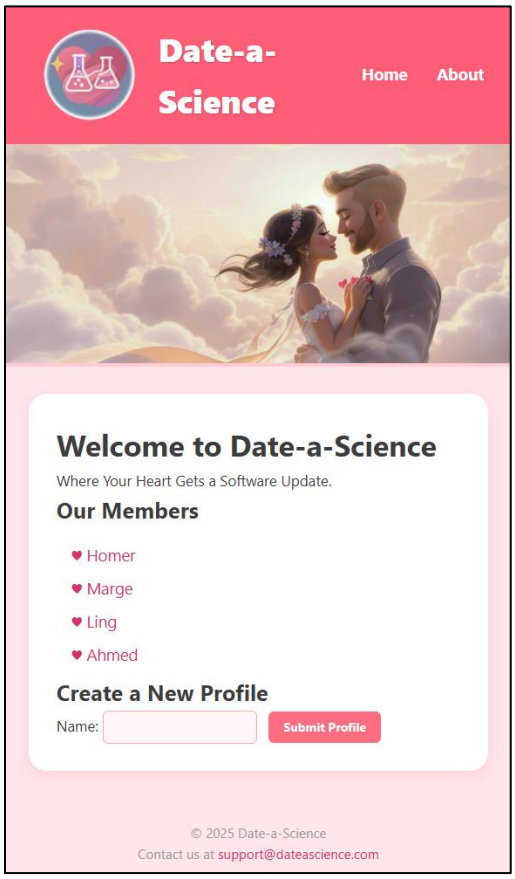
With Template Literals

```
let user = {
  name: "Anastasia",
  age: 23,
  bio: "Loves hiking, coffee, pizza & coding."
};

let html = `
  <div class="profile">
    <h2>${user.name}</h2>
    <p>Age: ${user.age}</p>
    <p>Bio: ${user.bio}</p>
  </div>
`;
```

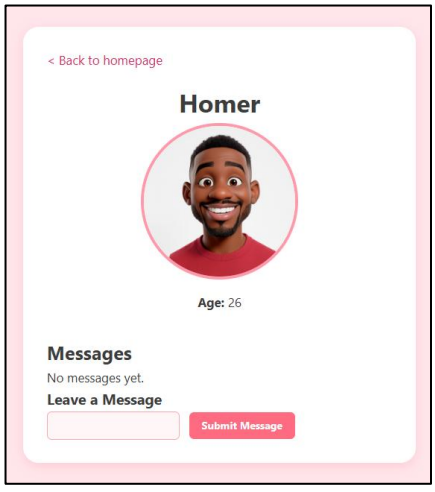
With template literals, we can actually do a lot. We could add functions to read files and then replace specific sections of those files with provided variable values.

Example

Let's look at an example of using template literals for a fake dating site called "Date-a-Science" that has a main homepage that looks like what is shown here on the right.



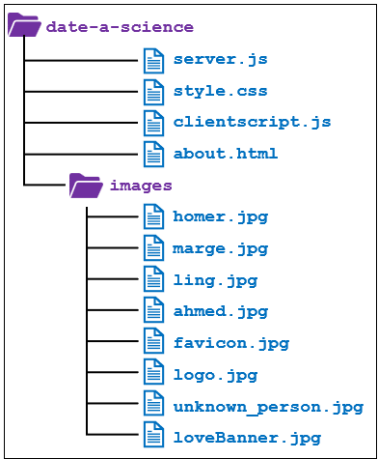
When the user clicks on a member name, it goes to their profile page, and the **about** goes to another page (see below):



We can also add a member to the list by creating a new profile by entering a name and pressing the **Submit Profile** button. And then from within a member's profile page, we can also enter a message and press the **Submit Message** button to leave a message for that person.

The **styles.css** file will not be described in our example, but it is mostly specifying spacing, padding, alignment, coloring, rounding corners, over effects, etc..

We will set up our server's folder structure as shown here →
(Assumes we begin with **4** profile images: homer, marge, ling, ahmed).



Below is a basic version of the main server (called **server.js**). We begin with a fixed array of **4** members, where each member has a name, id, age, photo and array of messages sent to them. Of course, this example can be expanded upon by adding more fields (e.g., friends, posts, etc.). Also, the messages are currently just strings, but the array could be an array of objects. We can then have more state for each message (e.g., creator, likes, views, match history, mutual interests, etc.).

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

let contents = "";
let nextID = 101;
let people = [{name: "Homer", id:nextID++, age: 26, photo:"/images/homer.jpg", messages:[]},
              {name: "Marge", id:nextID++, age: 25, photo:"/images/marge.jpg", messages:[]},
              {name: "Ling", id:nextID++, age: 27, photo:"/images/ling.jpg", messages:[]},
              {name: "Ahmed", id:nextID++, age: 24, photo:"/images/ahmed.jpg", messages:[]}];
let anonImg = "/images/unknown_person.jpg";

// Helper functions for sending specific messages
function respondMainPage(res) {…}
function respondAboutPage(res) {…}
function respondWithProfile(res, id) {…}
function serveStaticFile(filename, res) {…}
function send404(response) {…}

// 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

    // Handle all GET requests
    if (req.method === "GET") {
        if (pathname === "/" || pathname === "/home") {// Return the main page
            respondMainPage(res);
        } else if (pathname.startsWith("/profile")) {  // Return a specific profile
            respondWithProfile(res, parsed.query.id);
        } else if (pathname.startsWith("/about")) {    // Return the about page
            serveStaticFile(("./about.html", res);
        } else if (pathname === "/clientscript.js") {  // Return the client-side javascript
            serveStaticFile("./clientscript.js", res);
        } else if (pathname.startsWith("/images/")) {  // Return an image for a page
            serveStaticFile("." + pathname, res);
        } else if (pathname === "/style.css") {        // Return the style file
            serveStaticFile("./style.css", res);
        } else { // If anything else ... respond with 404 error
            send404(res);
        }
    } else { // If not a GET request, send 405
        res.writeHead(405);
        res.end("Unsupported method");
    }
}
```

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

Notice that the server currently handles just **GET** requests for when the browser asks for the main page (i.e., `"/"`), the about page (i.e., `"/about"`), a particular profile (e.g., `"/profile?101"`), the client-side **JavaScript** file (i.e., `"/clientscript.js"`), the style file (i.e., `"/style.css"`) or an image file (e.g., `"./images/homer.jpg"`). For any other request, it responds with a **404** message. The code is straight forward, but what goes into the helper functions?

The simplest one is the `send404()` because it is static:

```
// Helper function for sending 404 message
function send404(response) {
    response.writeHead(404, {"Content-Type": "text/plain"});
    response.write("Error 404: Resource not found.");
    response.end();
}
```

The `serveStaticFile()` is also straight forward because it also is static:

```
// Send back the specific file requested (html, JavaScript, css, jpg, or plain text)
function serveStaticFile(filename, res) {
    fs.readFile(filename, function(err, data) {
        if (err) {
            res.statusCode = 404;
            res.end("File not found");
        } else {
            res.writeHead(200, {"content-type": lookupMIMEType(filename)});
            res.end(data);
        }
    });
}
```

… which makes use of another helper function to get the **MIME** type:

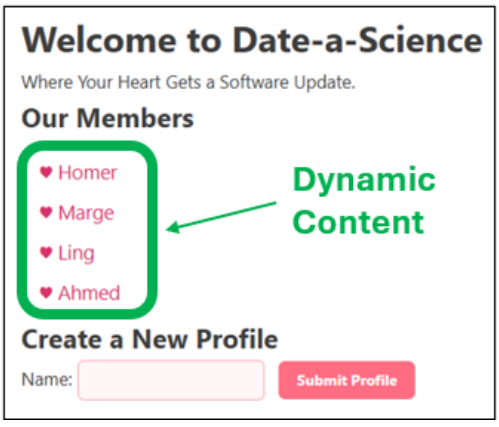
```
// Return the MIME type given the filename
function lookupMIMEType(filename) {
    if (filename.endsWith(".js")) return "application/javascript";
    if (filename.endsWith(".css")) return "text/css";
    if (filename.endsWith(".jpg") || filename.endsWith(".jpeg")) return "image/jpeg";
    if (filename.endsWith(".png")) return "image/png";
    if (filename.endsWith(".html")) return "text/html";
    return "text/plain";
}
```

There is nothing new here so far because this is the static content.

The **about.html** page has no dynamic content, so we can serve it back as a static file:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Date-a-Science</title>
        <link rel="stylesheet" href="/style.css">
        <link rel="icon" href="/images/favicon.jpg" type="image/jpeg">
    </head>
    <body>
        <nav class="navbar">
            <div class="nav-left">
                <img src="/images/logo.jpg" alt="Logo" class="logo">
                <span class="site-title">Date-a-Science</span>
            </div>
             <ul class="nav-menu">
                <li><a href="/">Home</a></li>
                <li><a href="/about">About</a></li>
            </ul>
        </nav>
        <div class="container">
            <h1>About Date-a-Science</h1>
            <p>Date-a-Science is a playful dating platform connecting scientifically curious minds
who believe in love through logic, chemistry, and a dash of nerdy fun. Whether you are into
molecules or algorithms, we've got a match for you.</p>
        </div>
        <footer>
            <p>&copy; 2025 Date-a-Science</p>
            <p>Contact us at <a
href="mailto:support@dateascience.com">support@dateascience.com</a></p>
         </footer>
    </body>
</html>
```

The **home** page, however, will have non-static content because the members list will grow over time. Also, the **profile** page is a template that we want to fill in based on the particular member. So, these two pages will have some dynamic content.



Let's look now at the **home** page. We need to consider what parts of the page are static and what parts are dynamic. The dynamic content is the unordered list of member names, which will depend on whatever is in the `people` array variable →

Initially, there will be **4** names, but more names will be added by the user. Here is the `respondMainPage()` function that will build this dynamic page for us:

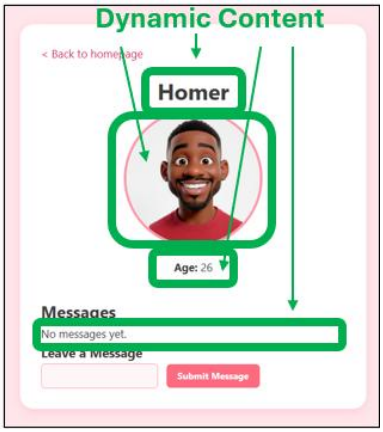
```
// Helper function for sending back the main homepage
function respondMainPage(res) {
  contents = `
    <!DOCTYPE html>
    <html>
      <head>
        <title>Date-a-Science</title>
        <link rel="stylesheet" href="./style.css">
        <link rel="icon" href="./images/favicon.jpg" type="image/jpeg">
```

```
      </head>
      <body>
        <nav class="navbar">
            <div class="nav-left">
                <img src="./images/logo.jpg" alt="Logo" class="logo">
                <span class="site-title">Date-a-Science</span>
            </div>
             <ul class="nav-menu">
                <li><a href="/">Home</a></li>
                <li><a href="/about">About</a></li>
            </ul>
        </nav>
        <div class="banner">
            <img src="./images/loveBanner.jpg" alt="Love Banner">
        </div>
        <div class="container">
          <h1>Welcome to Date-a-Science</h1>
          <p>Where Your Heart Gets a Software Update.</p>

          <h2>Our Members</h2>
          <ul class="profile-list">
  `;
  people.forEach(person => {
    contents += `<li><a href="./profile?id=${person.id}">&#x2665; ${person.name}</a></li>`;
  });
  contents += `
          </ul>

          <h2>Create a New Profile</h2>
          <label for="name">Name:</label>
          <input type="textbox" id="name">
          <button type="button" onclick="submit()">Submit Profile</button>
          <script src="/clientscript.js"></script>
        </div>

        <footer>
            <p>&copy; 2025 Date-a-Science</p>
            <p>Contact us at <a href="mailto:support@dateascience.com">support@dateascience.com</a></p>
         </footer>
      </body>
    </html>
  `;

  res.writeHead(200, { "Content-Type": "text/html" });
  res.end(contents);
}
```

Thanks to VSCode's color scheme, it is easy to identify the static content (as orange), while the dynamic content stands out in different colors. Notice that we use the template literal approach (i.e., the backtick) to specify the **HTML** `contents` for the top of the page (i.e., before the list). Then we append to the `contents`, our dynamic list. Then we append more static `contents` and send it back in the response. Notice the yellow code. This will insert the **clientscript.js** code into this location in the code. We will discuss this soon.

The interesting part of the static content is that we are making anchors for each member and then just showing the person's name. The anchor is the link that will specify the specific profile page that we want to request from the server when we click on the link:

```
http://localhost:3000/profile?id=101
```



Otherwise, the code should be quite straight forward and familiar to you.

Now… what about the **profile** page? What is dynamic on it? Well, look at the list item anchors from the code we just wrote:

```
<li><a href="./profile?id=${person.id}">&#x2665; ${person.name}</a></li>
```

We can see that we will be requesting the same exact profile page regardless of the person, but the `id` number (passed as a query) will differ each time. So, we just need to dynamically fill in the **name**, **picture**, **age** and **messages** for that person based on that `id`.

Below is the `respondWithProfile()` function that will build this dynamic page for us. We can quickly identify the dynamic content by the non-orange color:

```
// Helper function for sending back the profile page
function respondWithProfile(res, id) {
    // This is not an efficient way of finding people. You may want to use a HashMap.
    let person = null;
    for (let i=0; i<people.length; i++) {
        if (people[i].id == id) {
            person = people[i];
            break;
        }
    }
    // If the person is not found, send back an error
    if (person === null) {
        send404(res);
        return;
    }

    const photo = person.photo || anonImg;
    contents = `
        <!DOCTYPE html>
        <html>
            <head>
                <title>${person.name} - Date-a-Science</title>
                <link rel="stylesheet" href="/style.css">
                <link rel="icon" href="/images/favicon.jpg" type="image/jpeg">
            </head>
            <body>
                <div class="container">
                    <a href="/" class="back-link">< Back to homepage</a>

                    <div class="profile-card">
                        <h1>${person.name}</h1>
                        <img src="${photo}" width="200" alt="${person.name}'s photo">
                        <p><strong>Age:</strong> ${person.age}</p>
                    </div>

                    <div class="messages">
                        <h2>Messages</h2>`;
                        if (person.messages.length !== 0) {
                            person.messages.forEach(msg => {
                                contents += `<div class="message">${msg}</div>`;
```

```
                        });
                    } else {
                        contents += `<p>No messages yet.</p>`;
                    }
                    contents += `<h3>Leave a Message</h3>
                    <input type="textbox" id="msg">
                    <button type="button" onclick="submitMessage(${person.id})">Submit
Message</button>
                </div>

                <script src="/clientscript.js"></script>
            </div>
        </body>
    </html>
`;
    res.writeHead(200, { "Content-Type": "text/html" });
    res.end(contents);
}
```

To start, we look for the person in the array based on the `id`. We did something simple here, but in a real system, we would likely want to use something like a HashMap to find the person faster. Of course, if the person is not found (i.e., the `id` was invalid), then we send back the **404** message. Otherwise, we simply insert the person's **name**, **photo**, **age** and **messages** into the template literal and write out the response.

Again, notice the same yellow code to insert the **clientscript.js** code. This **JavaScript** code will be the code that must run on the client browser. It will have two functions … one to handle the **Submit Profile** button (from the **main** home page) and the other to handle the **Submit Message** button on this **profile** page.

We will need to send this **clientscript.js** file from the server to the browser as a static file. The browser will request it because we included **<script src="/clientscript.js"></script>** in the dynamic **HTML** code that will get executed on the client browser.

For the **Submit Profile** button, we need to start off **clientscript.js** with the following code:

```
// Function to handle the Submit Profile button on the main page
function submit() {
    let name = document.getElementById("name");
    if (name.value.length == 0) {
        alert("You must enter a name");
    } else {
        req = new XMLHttpRequest();
        req.onreadystatechange = function() {
            if (this.readyState == 4) {         // 4 means the response has come back
                if (this.status == 200) {
                    alert("Profile Created!");
                    name.value = "";
                    window.location.href = "/"; // Tell browser to load main page again
                } else {
                    alert("Error while trying to create profile. Try again.");
                }
            }
```

```
        }
        let person = {name: name.value};
        req.open("PUT", "/newprofile");
        req.send(JSON.stringify(person));
    }
}
```

It is standard **AJAX** request code that ensures there is non-empty name before sending, and then it sends a **PUT** request to this site: **`http://localhost:3000/newprofile`**. We will need to go to the server and update the **`handleRequest()`** function to handle this **PUT** request.

We need to add code before the last **send404() else** statement to handle this case. The grayed out code below is what was there already and is show for context of where to insert the new code:

```
// Listener for incoming client requests
function handleRequest(req, res) {
    // Handle all GET requests
    if (req.method === "GET") {
        if (req.url === "/"){ //... code not shown, to save space
        } else if (req.url.startsWith("/profile?")) { //... code not shown, to save space
        } else if (req.url === "/about") { //... code not shown, to save space
        } else if (req.url === "/clientscript.js") { //... code not shown, to save space
        } else if (req.url.startsWith("/images/")) {//... code not shown, to save space
        } else if (req.url === "/style.css") {//... code not shown, to save space
        } else { // If anything else ... respond with 404 error
            send404(res);
        }
    } else if (req.method === "PUT" && req.url === "/newprofile") {
            readRequestBody(req, function(body) {
                newPerson = JSON.parse(body);
                newPerson.id = nextID++;
                newPerson.messages = [];
                people.push(newPerson); // add profile to the array
                res.writeHead(200, { "content-type": "text/plain" });
                res.end();
            });
    } else {
        res.writeHead(405);
        res.end("Unsupported method");
    }
}
```

The code simply checks if the request's method is **PUT** and that the **URL** is **"/newprofile"** and then reads the request body … which is a **JSON** object with only the **`name`** defined. It then sets the **`id`** to be the next available one from the **`nextID`** counter and sets the **`messages`** array to an empty one. It finally sends back a successful **200** response.

Of course, we need the function to read the request body as it comes in piece-by-piece (although there is likely only one small piece in our case):

```
// Helper function to read the request body
function readRequestBody(req, callback) {
    let body = ""
    req.on("data", (chunk) => {
        body += chunk;
    })
```

```
    //Once the entire body is ready, process the request
    req.on("end", () => {
        callback(body);
    });
}
```

Now, let's handle the **Submit Profile** button on the profile page. The client request code is similar, but now we will use a **POST**, along with the message, instead of a **PUT**:

```
// Function to handle the Submit Message button on the profile page
function submitMessage(id) {
    let msg = document.getElementById("msg");
    if (msg.value.length == 0) {
        alert("You must enter a message");
    } else {
        req = new XMLHttpRequest();
        req.onreadystatechange = function() {
            if (this.readyState == 4) {       // 4 means the response has come back
                if (this.status == 200) {
                    alert("Message Sent!");
                    msg.value = "";
                    window.location.reload(); // Tell browser to load this page again
                } else {
                    alert("Error while sending message. Try again.");
                }
            }
        }
        let messageContent = { message: msg.value, id: id };
        req.open("POST", "/message");
        req.send(JSON.stringify(messageContent));
    }
}
```

Again, we see standard **AJAX** request code that ensures there is non-empty message before sending, and then it sends a **POST** request to this site: **`http://localhost:3000/message`**. We will need to go to the server and update the **`handleRequest()`** function again to handle this **POST** request.

We need to add code again, before the last **send404() else** statement. The grayed out code below is what was there already and is show for context of where to insert the new code:

```
// Listener for incoming client requests
function handleRequest(req, res) {
    // Handle all GET requests
    if (req.method === "GET") {
       if (req.url === "/"){ //... code not shown, to save space
        } else if (req.url.startsWith("/profile?")) { //... code not shown, to save space
        } else if (req.url === "/about") { //... code not shown, to save space
        } else if (req.url === "/clientscript.js") { //... code not shown, to save space
        } else if (req.url.startsWith("/images/")) {//... code not shown, to save space
        } else if (req.url === "/style.css") {//... code not shown, to save space
        } else { // If anything else ... respond with 404 error
            send404(res);
        }
    }
    // Combination of PUT and /newprofile is interpreted as a request to create a profile
    else if (req.method === "PUT" && req.url === "/newprofile") {
        readRequestBody(req, function(err, body) {
            newPerson = JSON.parse(body);
```

```
            newPerson.id = nextID++;
            newPerson.messages = [];
            people.push(newPerson); // add profile to the array
            res.writeHead(200, { "content-type": "text/plain" });
            res.end();
        });
    // Combination of POST and /message is interpreted as a request to add a message
    } else if (req.method === "POST" && req.url === "/message") {
        readRequestBody(req, function(body) {
            console.log("Adding message: " + body);
            let message = JSON.parse(body);
            if (addMessage(message)) {
                res.writeHead(200, { "content-type": "text/plain" });
                res.end();
            } else {
                send404(res);
            }
        });
    //Any other combination of method/URL is not supported
    } else {
        res.writeHead(405);
        res.end("Unsupported method");
    }
}
```

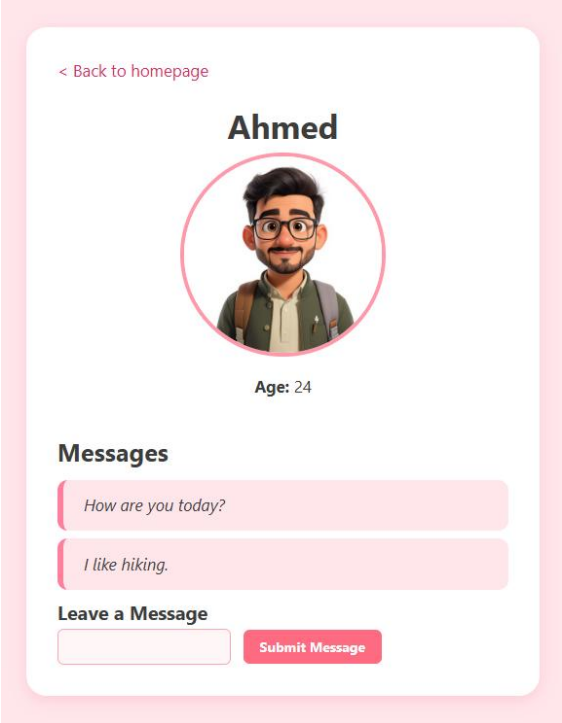
The code checks for a **POST** message type and to ensure that the **URL** is **/message** and then reads the request body … which is a **JSON** object which contains the `message` (defined as a string) as well as the `id` of the person to send it to.

It then calls `addMessage()` to add it to the profile's `messages` array. The function returns `true` if it was successful:

```
// Helper function to add a message to a person's profile
function addMessage(msg) {
    for (let i=0; i<people.length; i++) {
        if (people[i].id === msg.id) {
            people[i].messages.push(msg.message);
            return true;
        }
    }
    return false;
}
```

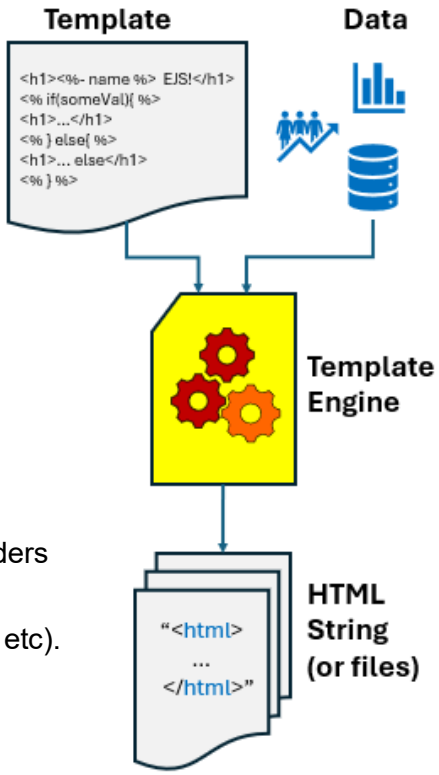
That's it! Messages will appear one after another in a list after we add submit them →

So, we now understand that we can write **JavaScript** functions for generating **HTML**. But as mentioned earlier, there are downsides to doing things this way.



9.2 Embedded JavaScript Templates (EJS)

A ***template engine*** is a tool that combines **HTML** with dynamic data to automatically generate web pages in a clean and organized way. Template engines solve the problems mentioned above by providing a cleaner, more structured way to separate application logic from presentation (**HTML**).



Template engines allow us to:

- ✓ Write **HTML** templates with placeholders for dynamic data.
- ✓ Keep **HTML** readable and maintainable.
- ✓ Prevent common security risks by escaping output by default.
- ✓ Speed up development with built-in utilities (e.g., looping or conditional rendering).

At a high level, a template engine does three things:

1. Takes a **Template** file (e.g., **HTML** with special syntax for placeholders representing document structure & generalized contents).
2. Combines it with **Data** from our application (e.g., users, products, etc).
3. Produces a final **HTML output** string to send to the browser.

The template engine combines the data and templates in a process called ***rendering*** … producing the ***view*** … which is the visual part of the application that users interact with in their browser.

We will talk soon about ***Express***, which is a fast, widely-used, minimal web framework for **Node.js** that makes it easy to build web servers and **API**s. Several template engines work with **Express** such as **Pug**, **Mustache** and **EJS**.

Template engines are useful when we want to rapidly build web applications that are split into different components … especially when we want to:

- **reuse** common **HTML** structures across multiple pages (e.g., headers, footers, navigation bars, sidebars, etc.).
- **insert** dynamic content into **HTML** (e.g., usernames, product lists, messages).
- **quickly** render views in server-side web applications, where the **HTML** changes based on user requests or data from a database.
- **separate** our visual specification from data and server logic.

They're a great choice when we want to build pages that share a consistent layout but display different content. In short: they help us write clean **HTML** layouts that automatically get filled with the right content when needed.

EJS (Embedded JavaScript) is one of the most popular and lightweight template engines for **Node.js**. It lets us embed **JavaScript** code directly in our **HTML** to dynamically produce portions of the page. To use **EJS**, we need to ensure that it is first installed using **NPM**: `npm install ejs` and then use require in our server code: `const ejs = require("ejs");`

EJS supports the idea of ***partials*** … which is a piece of an entire "page" that can be re-used (e.g., headers, footers, menus, etc.)

For example, the **home** page and **about** page of our **Date-a-Science** site both contain the same exact code to start the page, so they have the same header:

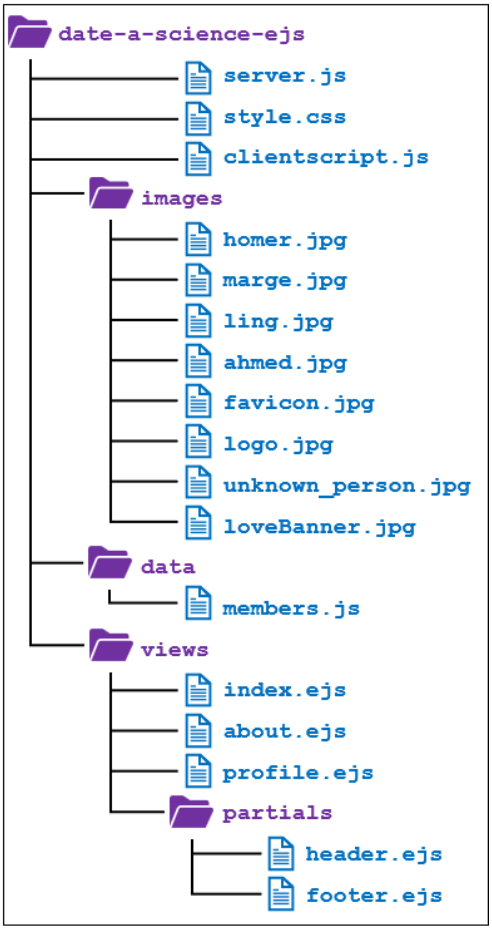
```
<!DOCTYPE html>
<html>
    <head>
        <title>Date-a-Science</title>
        <link rel="stylesheet" href="/style.css">
        <link rel="icon" href="/images/favicon.jpg" type="image/jpeg">
    </head>
    <body>
        <nav class="navbar">
            <div class="nav-left">
                <img src="/images/logo.jpg" alt="Logo" class="logo">
                <span class="site-title">Date-a-Science</span>
            </div>
             <ul class="nav-menu">
                <li><a href="/">Home</a></li>
                <li><a href="/about">About</a></li>
            </ul>
        </nav>
```

We could consider this a ***partial*** and place this code into a file called `header.ejs`. At the moment this is just an **HTML** file, but we will adjust it to have a small bit of dynamic content in a moment, so we will use the `.ejs` file extension. Notice that the file is not a completed **HTML** file because there are no closing tags for the **<body>** and **<html>** tags. That is because we will include this code as "partial" code that will be appended to other code, which will have the proper closing of those tags as needed.

In addition, the **home** and **about** pages both share the same footer information. So, we can make a `footer.ejs` file as well:

```
        <footer>
            <p>&copy; 2025 Date-a-Science</p>
            <p>Contact us at <a href="mailto:support@dateasci.com">support@dateasci.com</a></p>
        </footer>
    </body>
</html>
```

Let's convert our code to make use of the **EJS** template engine. We will adjust our folder hierarchy to be as shown here on the right. The top portion is the same, but we added a `data` folder to hold our data, a `views` folder to hold our dynamic pages, and a `partials` folder to hold our `header.ejs` and `footer.ejs` partials.



We may notice that we have three `.ejs` files in the `views` folder. These are mostly made up of **HTML** but they will have some dynamic **EJS** content as well. Let's begin our conversion to **EJS** by taking a look at the `about.html` file from before:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Date-a-Science</title>
        <link rel="stylesheet" href="/style.css">
        <link rel="icon" href="/images/favicon.jpg" type="image/jpeg">
    </head>
    <body>
        <nav class="navbar">
            <div class="nav-left">
                <img src="/images/logo.jpg" alt="Logo" class="logo">
                <span class="site-title">Date-a-Science</span>
            </div>
             <ul class="nav-menu">
                <li><a href="/">Home</a></li>
                <li><a href="/about">About</a></li>
            </ul>
        </nav>
        <div class="container">
            <h1>About Date-a-Science</h1>
            <p>Date-a-Science is a playful dating platform connecting
scientifically curious minds who believe in love through logic, chemistry, and
a dash of nerdy fun. Whether you are into molecules or algorithms, we've got a
match for you.</p>
        </div>
        <footer>
            <p>&copy; 2025 Date-a-Science</p>
            <p>Contact us at <a
href="mailto:support@dateascience.com">support@dateascience.com</a></p>
         </footer>
    </body>
</html>
```

The first thing that we will notice is that all of the yellow code is code that is in our header and footer.

So, for our `about.ejs` file, we will simply include the `header.ejs` and `footer.ejs` files instead of duplicating the code.

To include an **EJS** partial in another **EJS** file, we simply use **EJS**'s **include()** function which has this format:

```
<%- include("filename") %>
```

Here, the *`filename`* is the name of the `.ejs` file that we want to include, but we leave off the file extension. If the file is in the same folder as the one that we write this include statement in, then we do not need to specify a path (e.g., *`"header"`*). However, if it is in subfolder or other folder, we need to specify the path relative to where we are (e.g., *`"partials/header"`*).

So, we can write our `about.ejs` file (which will replace our previous `about.html` file) as follows:

```
<%- include('partials/header') %>

<div class="container">
  <h1>About Date-a-Science</h1>
  <p>Date-a-Science is a playful dating platform connecting scientifically curious minds who
believe in love through logic, chemistry, and a dash of nerdy fun. Whether you are into molecules
or algorithms, we've got a match for you.</p>
</div>

<%- include("partials/footer") %>
```

Notice that the main content of the page looks *short & sweet* now, since we do not clutter it up with the header and footer code.

The `<%-` at the beginning of the include call indicates that we want to output …

Unescaped HTML - **HTML** code that is rendered directly in the browser without converting special characters (e.g., **<**, **>**, **&**, etc..), meaning it produces actual **HTML** elements (e.g., headings, images, or links) instead of displaying the raw code as plain text.

We use this if we trust the content being rendered (i.e., we wrote it). We would not use it if we had content that may contain **user input** (e.g., messages, comments) because it could lead to security risks. In those cases, we would use `<%=` instead, which will output …

Escaped HTML - **HTML** code in which special characters (e.g., **<**, **>**, **&**, etc..) are turned into safe text so that they don't act like real **HTML** in the browser. It's like putting a disguise on **HTML** so the browser doesn't treat it as code and just shows it as plain text.

Escaping protects our site from *code injection* and keeps user-submitted content safe and harmless.

A nice feature of the **include()** function is that it allows us to pass in data by supplying a second parameter. So, if we want to pass along some data into our **EJS** file, we could do it by specifying a second parameter which must be a **JavaScript** object containing one or more ***key***: ***value*** pairs, separated by commas:

```
<%- include(file, { key1: val1, key2: val2, key3: val3, etc.. }) %>
```

Let's try this by passing in the page title to our `header.ejs` file. This will be useful because the header partial contains the **<head>** tag which has the page **<title>** in it … which will be shown as the title of the page on the browser tab. By passing in a title to the header, each page that shares the header will be able to have their own unique page title (e.g., "**Date-a-Science – Home**", "**About – Date-a-Science**", "**Homer - Date-a-Science**", "**Marge - Date-a-Science**", etc..)

So, we will adjust the first line of our `about.ejs` page to be as follows:

```
<%- include("partials/header", { title: "About - Date-a-Science" }) %>
```

Then, we can make use of this in our **`header.ejs`** file by using this **title** variable.

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel="stylesheet" href="/style.css">
    <link rel="icon" href="/images/favicon.jpg" type="image/jpeg">

etc.
```

This tells **EJS** to get the **`title`** variable's value (because there are no quotes around it, it assumes that it is a variable defined). It also tells it to render the value as plain text (i.e., **<%=** instead of **<%-**). As we can see, it is quite easy to supply some data when including **EJS** files inside of others.

Now, what about our main homepage? Well, recall that we had a function in our server for producing the **HTML** code that did this (highlighted header/footer code in yellow):

```
contents = `
    <!DOCTYPE html>
    <html>
      <head>
        <title>Date-a-Science</title>
        <link rel="stylesheet" href="/style.css">
        <link rel="icon" href="/images/favicon.jpg" type="image/jpeg">
      </head>
      <body>
        <nav class="navbar">
            <div class="nav-left">
                <img src="/images/logo.jpg" alt="Logo" class="logo">
                <span class="site-title">Date-a-Science</span>
            </div>
             <ul class="nav-menu">
                <li><a href="/">Home</a></li>
                <li><a href="/about">About</a></li>
            </ul>
        </nav>
        <div class="banner">
            <img src="/images/loveBanner.jpg" alt="Love Banner">
        </div>
        <div class="container">
          <h1>Welcome to Date-a-Science</h1>
          <p>Where Your Heart Gets a Software Update.</p>

          <h2>Our Members</h2>
          <ul class="profile-list">
  `;

  people.forEach(person => {
    contents += `<li><a href="./profile?id=${person.id}">${person.name}</a></li>`;
  });

  contents += `
          </ul>
```

```
          <h2>Create a New Profile</h2>
          <label for="name">Name:</label>
          <input type="textbox" id="name">
          <button type="button" onclick="submit()">Submit Profile</button>

          <script src="/clientscript.js"></script>
        </div>

        <footer>
            <p>&copy; 2025 Date-a-Science</p>
            <p>Contact us at <a
href="mailto:support@dateascience.com">support@dateascience.com</a></p>
         </footer>

      </body>
    </html>
  `;
```

Instead of putting the homepage in a server function that will produce the **HTML** code, we will extract it to an **`index.ejs`** file and place it in our **`views`** folder. Since the header and footer are the same as what is in our partial files, we will begin by replacing those with appropriate include calls and get rid of the backticks and **`contents`** variable (which was something specific to our server code):

```
<%- include("partials/header", { title: "Date-a-Science - Home" }) %>

        <div class="banner">
            <img src="/images/loveBanner.jpg" alt="Love Banner">
        </div>
        <div class="container">
          <h1>Welcome to Date-a-Science</h1>
          <p>Where Your Heart Gets a Software Update.</p>

          <h2>Our Members</h2>
          <ul class="profile-list">

  people.forEach(person => {
    <li><a href="./profile?id=${person.id}">${person.name}</a></li>
  });

          </ul>

          <h2>Create a New Profile</h2>
          <label for="name">Name:</label>
          <input type="textbox" id="name">
          <button type="button" onclick="submit()">Submit Profile</button>

          <script src="/clientscript.js"></script>
        </div>

<%- include("partials/footer") %>
```

Now what about the dynamic code in the middle? We have to change it slightly because this is JavaScript code, not **EJS** code.

Notice that it makes use of a `people` variable. We do not have this available to us here at the moment. However, we can pass the array of members in as some **EJS** data to this file (i.e., when the **EJS** engine renders this page) and then we will have access to what we need. So, really, all we need to do is to change the syntax from **JavaScript** to **EJS**.

For any code that we want to evaluate, we enclose it within `<%  %>` to indicate that we want to insert a block of code. It does not output anything to the rendered **HTML**. Instead, it is used for control flow, variable declarations, or any logic we want to run.

So, we replace this:

```
people.forEach(person => {
  <li><a href="./profile?id=${person.id}">${person.name}</a></li>
});
```

With this:

```
<% people.forEach(person => { %>
  <li><a href="./profile?id=<%= person.id %>"><%= person.name %></a></li>
<% }) %>
```

Which means …. for our **4** initial sample members … the code that will be rendered is as follows:

```
  <li><a href="./profile?id=1">Homer</a></li>
  <li><a href="./profile?id=2">Marge</a></li>
  <li><a href="./profile?id=3">Ling</a></li>
  <li><a href="./profile?id=4">Ahmed</a></li>
```

So, here is our completed `index.ejs` file which has included partials and supports dynamic content:

```
<%- include("partials/header", { title: "Date-a-Science - Home" }) %>
        <div class="banner">
            <img src="/images/loveBanner.jpg" alt="Love Banner">
        </div>
        <div class="container">
            <h1>Welcome to Date-a-Science</h1>
            <p>Where Your Heart Gets a Software Update.</p>

            <h2>Our Members</h2>

            <ul class="profile-list">
                <% people.forEach(person => { %>
                <li><a href="/profile?id=<%= person.id %>">&#x2665; <%= person.name %></a></li>
                <% }) %>
            </ul>

            <h2>Create a New Profile</h2>
            <label for="name">Name:</label>
            <input type="textbox" id="name">
            <button type="button" onclick="submit()">Submit Profile</button>

            <script src="/clientscript.js"></script>
        </div>
<%- include("partials/footer") %>
```

For this to work, we need to ensure that we pass in the **people** data. That brings up an important point. Currently our data is merged with our server code. This is not good because we should really de-couple our content from our logic. So, let's extract the member-specific data and put it in its own **JavaScript** file. Here is the current data-specific code from the server file:

```
let nextID = 101;
let people = [{name: "Homer", id:nextID++, age: 26, photo:"/images/homer.jpg", messages:[]},
              {name: "Marge", id:nextID++, age: 25, photo:"/images/marge.jpg", messages:[]},
              {name: "Ling", id:nextID++, age: 27, photo:"/images/ling.jpg", messages:[]},
              {name: "Ahmed", id:nextID++, age: 24, photo:"/images/ahmed.jpg", messages:[]}];
let anonImg = "/images/unknown_person.jpg";
```

We will place this in a **members.js** file under the **data** folder in our hierarchy. While we are at it, we will adjust the **nextID** variable to be somewhat private by making a public function that gets us the next id for us so that we can avoid misusing the variable. Here is the completed **members.js** file:

```
let nextID = 101;

// Gets the next profile ID available (its just a counter)
export function getNextID() {
  return nextID++;
}

// These are sample people for the purposes of demonstration
export let people = [
  { name: "Homer", id: getNextID(), age: 26, photo: "/images/homer.jpg", messages: [] },
  { name: "Marge", id: getNextID(), age: 25, photo: "/images/marge.jpg", messages: [] },
  { name: "Ling",  id: getNextID(), age: 27, photo: "/images/ling.jpg",  messages: [] },
  { name: "Ahmed", id: getNextID(), age: 24, photo: "/images/ahmed.jpg", messages: [] }
];

// This is the image to show for new profiles that have no uploaded picture
export const anonImg = "/images/unknown_person.jpg";
```

Notice the use of **export** to ensure that these are visible and usable to others. We can then just include this file by using **require()** to ask for this module in our server code as follows:

```
const { people, anonImg, getNextID } = require("./data/members");
```

This will give us access to the array, the anonymous image and the function to get the id. But before we adjust our server code, let's look at how to convert the profile rendering code from our server into a **profile.ejs** file to place in our **views** folder. Here is the code that we have in our server:

```
    contents = `
        <!DOCTYPE html>
        <html>
            <head>
                <title>${person.name} - Date-a-Science</title>
                <link rel="stylesheet" href="/style.css">
                <link rel="icon" href="/images/favicon.jpg" type="image/jpeg">
```

```
            </head>
            <body>
                <div class="container">
                    <a href="/" class="back-link">< Back to homepage</a>

                    <div class="profile-card">
                        <h1>${person.name}</h1>
                        <img src="${photo}" width="200" alt="${person.name}'s photo">
                        <p><strong>Age:</strong> ${person.age}</p>
                    </div>

                    <div class="messages">
                        <h2>Messages</h2>`;
                        if (person.messages.length !== 0) {
                            person.messages.forEach(msg => {
                                contents += `<div class="message">${msg}</div>`;
                            });
                        } else {
                            contents += `<p>No messages yet.</p>`;
                        }
                        contents += `<h3>Leave a Message</h3>
                        <input type="textbox" id="msg">
                        <button type="button" onclick="submitMessage(${person.id})">Submit
Message</button>
                    </div>

                    <script src="/clientscript.js"></script>
                </div>
            </body>
        </html>
    `;
```

We will notice that it does NOT have the same header as our other files and it does NOT even have a footer. We will make an adjustment so that we have consistency across our pages by including the same header and footer as the other pages. Notice that the code makes use of a `person` variable. We will need to ensure that this is passed in as data to this page when we render it. So, we can do what we did for the other **EJS** pages … replace the yellow highlighted code by including the header and footer in place of them, and then by altering the **JavaScript** syntax to enclose the dynamic code within `<%  %>`. Here is the resulting `profile.ejs` file:

```
<%- include("partials/header", { title: person.name + " - Date-a-Science" }) %>

<div class="container">
  <a href="/" class="back-link">&lt; Back to homepage</a>

  <div class="profile-card">
    <h1><%= person.name %></h1>
    <img src="<%= person.photo || anonImg %>" width="200" alt="<%= person.name %>'s photo">
    <p><strong>Age:</strong> <%= person.age %></p>
  </div>

  <div class="messages">
    <h2>Messages</h2>
    <% if (person.messages.length !== 0) { %>
      <% person.messages.forEach(msg => { %>
```

```
        <div class="message"><%= msg %></div>
      <% }) %>
    <% } else { %>
      <p>No messages yet.</p>
    <% } %>

    <h3>Leave a Message</h3>
    <input type="textbox" id="msg">
    <button type="button" onclick="submitMessage(<%= person.id %>)">Submit Message</button>
  </div>

  <script src="/clientscript.js"></script>
</div>

<%- include("partials/footer") %>
```

As we can see, **EJS** files are quite readable, even though they mix code with **HTML**. I am sure that we will find this easy as we do it more and more. Of course, this code only works if we pass in the **person** variable data when we render the page.

OK, so we are now ready to adjust our server code. It will be shorter now that we don't have the **HTML** rendered code in it since we extracted the code for rendering the main home page and profile page and placed them into **EJS** files.

These functions will remain the same: `send404()`, `getMimeType()`, `serveStaticFile()`, `readRequestBody()` and `addMessage()`. And in the code to handle incoming requests, we only need to update the code in three locations:

```
    // Handle all GET requests
    if (req.method === "GET") {
        if (pathname === "/" || pathname === "/home") {// Return the main page
            renderPage("./views/index.ejs", res, { people });
        } else if (pathname.startsWith("/profile")) { // Return a specific profile
            respondWithProfile(res, parsed.query.id);
        } else if (pathname.startsWith("/about")) {    // Return the about page
            renderPage("./views/about.ejs", res);
        ...
    }

    // Combination of PUT and /newprofile is interpreted as a request to create a profile
    else if (req.method === "PUT" && pathname === "/newprofile") {
        readRequestBody(req, function(body) {
            const newPerson = JSON.parse(body);
            newPerson.id = getNextID();                    // This used to be nextID++
etc..
```

Notice that we call a `renderPage()` function now to do the rendering for the main page and about page. We pass it the particular page to render as well as the response variable and any needed data.

Here is the `renderPage()` function:

```
// Render an EJS page
function renderPage(templatePath, res, data = {}) {
    ejs.renderFile(templatePath, data, function(err, html) {
        if (err) {
            console.error(err);
            res.writeHead(500);
            res.end("Internal Server Error");
        } else {
            res.writeHead(200, { "Content-Type": "text/html" });
            res.end(html);
        }
    });
}
```

This function calls the `renderFile()` function in the imported `ejs` module … which does all the work for us. The function takes three parameters:

- **templatePath** - string path to the **EJS** template file.
- **data** - an object containing variables to pass into the template.
- **callback** - a function to call with the rendered result (or error).

Notice that a **500** error is returned if the page cannot be rendered from the server. Otherwise, it returns a **200** along with the rendered `html` string … which is equivalent to the resulting `contents` variable that we kept appending to in the non-**EJS** server version of our code.

The `respondWithProfile()` function is as follows:

```
// Handle individual profile
function respondWithProfile(res, id) {
    // This is not an efficient way of finding people. You may want to use a HashMap.
    let person = null;
    for (let i=0; i<people.length; i++) {
        if (people[i].id == id) {
            person = people[i];
            break;
        }
    }
    // If the person is not found, send back an error
    if (person === null) {
        send404(res);
        return;
    }
    renderPage("./views/profile.ejs", res, { person, anonImg });
}
```

The code does the same as before to find the person, but now instead of all that code to write out the **HTML** code, it is all rendered from the `profile.ejs` file. Notice that we pass in the `person` and `anonImg` data so that these can be used within the `profile.ejs` file's code.

Of course, we need to include these at the top of our `server.js` file as well now:

```
const  ejs = require("ejs");
const { people, anonImg, getNextID } = require("./data/members");
```

but we can delete all the data that was at the top of the file because we moved it into the **`members.ejs`** file.

If you get a moment, take a look at the **FutureTech** website code which has been altered to use the **EJS** Template Engine. Of particular note is the **`product-data.js`** file that contains all our data now, just like the **`members.js`** contained data for our **Date-a-Science** site:

```
export const products = [
    {
        name: "SkyCore JetPack",
        id: 1,
        image: "jetPack.jpg",
        imageSmall: "jetPack_small.jpg",
        desc: "Take to the skies delivering powerful, stable flight that turns gravity into your playground.",
        imageAlt: "SkyCore JetPack",
        par1: "Experience the future of personal flight with the <strong>SkyCore JetPack</strong> — engineered with next-gen quantum propulsion technology for unparalleled speed, agility, and eco-friendly performance. Designed for thrill-seekers and urban adventurers alike, the AeroX Quantum redefines freedom, allowing you to soar effortlessly above the city skyline or explore rugged terrains with ease.",
        par2: "Built with lightweight carbon-titanium alloys and powered by a zero-emission quantum core, this jet pack is not only fast but also sustainable. Integrated AI navigation and safety systems ensure a smooth and secure flight experience, whether you're a beginner or a seasoned pilot.",
        features: [
            {feature: "Quantum Propulsion Engine", desc:"Silent, ultra-efficient engine offering up to 45 minutes of continuous flight at speeds reaching 120 mph."},
            {feature: "AI-Assisted Navigation", desc:"Real-time obstacle detection, automatic stabilization, and route optimization via onboard AI."},

            etc..
        ],
        specs: [
            {key: "Flight Time", value: "Up to 45 minutes" },
            {key: "Maximum Speed", value: "120 mph (193 km/h)"},
            {key: "Weight", value: "15 kg (33 lbs)"},
            etc..
        ],
        pricing: [
            {option: "Standard Package", price: 19999.00, includes: "Jet pack, charger, basic user manual"},
            {option: "Advanced Package", price: 24999.00, includes: "Standard + AI flight coach app + extended warranty (3 years)"},
            {option: "Premium Package", price: 29999.00, includes: "Advanced + custom design panels + on-site training"}
        ],
        accessories: [
            {option: "Solar Charging Pad", price: 499.00},
            {option: "Extra Modular Thrusters", price: 799.00},
            {option: "Protective Flight Helmet with HUD", price: 1199.00},
            {option: "Portable Flight Case", price: 299.00}
        ]
    },
```

Thus, we were able to get rid of our individual product pages (i.e., **`product_01.html`**, **`product_02.html`**, etc..) and make one **`product.ejs`** file that accommodates all our products.

Closing remarks:

Recall that we used this code to look up the person:

```
let person = null;
for (let i=0; i<people.length; i++) {
    if (people[i].id == id) {
        person = people[i];
        break;
    }
}
```

This is a linear search, which could take a long time. A faster approach would be to use a **Hashmap**. We could set this up ahead of time:

```
let peopleById = new Map();
people.forEach(p => peopleById.set(p.id, p));
```

Then we can replace our search code with this line (remember to convert the string `id` to a number):

```
let person = peopleById.get(Number(id));
```

Alternatively, we could have just used a **Hashmap** from the start by replacing this:

```
// These are sample people for the purposes of demonstration
export let people = [
    { name: "Homer", id: getNextID(), age: 26, photo: "/images/homer.jpg", messages: [] },
    { name: "Marge", id: getNextID(), age: 25, photo: "/images/marge.jpg", messages: [] },
    { name: "Ling",  id: getNextID(), age: 27, photo: "/images/ling.jpg",  messages: [] },
    { name: "Ahmed", id: getNextID(), age: 24, photo: "/images/ahmed.jpg", messages: [] }
];
```

With this:

```
export let people = new Map();
let nid;
people.set(nid = getNextID(),
           { name: "Homer", id: nid, age: 26, photo: "/images/homer.jpg", messages: [] });
people.set(nid = getNextID(),
           { name: "Marge", id: nid, age: 25, photo: "/images/marge.jpg", messages: [] });
people.set(nid = getNextID(),
           { name: "Ling",  id: nid, age: 27, photo: "/images/ling.jpg",  messages: [] });
people.set(nid = getNextID(),
           { name: "Ahmed", id: nid, age: 24, photo: "/images/ahmed.jpg", messages: [] });
```

We now have a good grasp of writing dynamic server-side code. However, currently, the data is hard-coded into our server. That means we would need to restart the server every time that we want to change the data. Later, we will find out how to get this data from a database and as well be able to modify the data through some other server logic. At that point, we would not need to restart our server when we need to alter the data.

9.3 PUG

Another nice template engine is ***PUG*** (it is not an acronym … just a name). **Pug** is similar to **EJS** in that both are template engines for **Node.js** that let us generate dynamic **HTML** using **JavaScript** data. They allow us to insert variables, use conditionals (if statements), and loop through data to build **HTML** pages dynamically. Both ultimately produce standard **HTML** that is sent to the browser.

The key idea behind both engines is that of separating our layout (i.e., **HTML** structure) from our logic (i.e., **JavaScript** code), which makes our web applications more organized and maintainable. While their syntax is different, the main purpose and functionality are essentially the same.

When we think of our **EJS** code, it has a mix of **HTML** and **JavaScript**. In some ways this is good, because we are familiar with it and it is easy to identify the **JavaScript** tags/code (especially in **VSCode**). The downside of **EJS**, however, is that it can get messy and hard to manage, especially as our templates grow. It's easy to lose track of where the logic ends and the **HTML** begins. This can make the file harder to read, especially when we are nesting conditionals or loops. Over time, our template can start to look cluttered, with a lot of angle brackets, opening and closing tags, and **JavaScript** syntax all jammed together. This makes debugging and maintaining our code more difficult, particularly for larger teams or beginners who are still learning how templating works.

In contrast, **PUG** offers a clean, minimalist syntax that makes templates easier to read and write. By removing the need for closing tags and angle brackets, **PUG** reduces clutter and helps us focus on the structure of our content. Its indentation-based layout clearly shows nesting, which makes our **HTML** structure more obvious and less error-prone. Here is a quick side-by-side comparison:

```
<%- include("partials/header", { title: "Date-a-Science -
Home" }) %>

  <div class="banner">
    <img src="/images/loveBanner.jpg" alt="Love Banner">
  </div>

  <div class="container">
    <h1>Welcome to Date-a-Science</h1>
    <p>Where Your Heart Gets a Software Update.</p>

    <h2>Our Members</h2>
    <ul class="profile-list">
      <% people.forEach(person => { %>
        <li><a href="/profile?id=<%= person.id %>">&#x2665;
                    <%= person.name %></a></li>
      <% }) %>
    </ul>

    <h2>Create a New Profile</h2>
    <label for="name">Name:</label>
    <input type="textbox" id="name">
    <button type="button" onclick="submit()">
            Submit Profile</button>

    <script src="/clientscript.js"></script>
  </div>

<%- include("partials/footer") %>
```

```
include partials/header.pug
  - var title = "Date-a-Science - Home"

.banner
  img(src="/images/loveBanner.jpg" alt="Love Banner")


.container
  h1 Welcome to Date-a-Science
  p Where Your Heart Gets a Software Update.

  h2 Our Members
  ul.profile-list
    each person in people
      li
        a(href=`/profile?id=${person.id}`) &#x2665;
#{person.name}


  h2 Create a New Profile
  label(for="name") Name:
  input#name(type="textbox")
  button(type="button" onclick="submit()") Submit Profile


  script(src="/clientscript.js")


include partials/footer.pug
```

PUG Syntax:

PUG uses indentation and whitespace, much like Python, to define structure instead of relying on opening and closing tags. Here is a comparison of **HTML** vs. **PUG**:

```html
<!-- HTML -->

<h1>Welcome</h1>
<p>This is a paragraph.</p>
```

```
// PUG

h1 Welcome
p This is a paragraph.
```

Each line represents an **HTML** element, with the tag name at the beginning:

```html
<img src="/logo.png" alt="Logo">
```

```
img(src="/logo.png" alt="Logo")
```

The indentation shows the nesting of elements with child elements are indented under their parents:

```html
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
```

```
ul
  li Item 1
  li Item 2
```

There are no closing tags in **PUG**. They are handled automatically based on indentation:

```html
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
    </tr>
  </thead>
  <tbody>
    <% people.forEach(person => { %>
      <tr>
        <td><%= person.name %></td>
        <td><%= person.age %></td>
      </tr>
    <% }) %>
  </tbody>
</table>
```

```
table
  thead
    tr
      th Name
      th Age
  tbody
    each person in people
      tr
        td= person.name
        td= person.age


  // Nice ... isn't it?
```

Notice the **=** beside the table data (i.e., **`td=`**). The **=** tells **PUG** to "evaluate the following **JavaScript** expression and output the result." That is needed every time we have a variable value to insert. Without the **=**, **PUG** would simply write out **`person.name`** and **`person.age`** as plain text instead of substituting the variable values. Notice as well the difference in the looping constructs. We use **`each`**/**`in`** to iterate.

We can use three particular operators in **PUG**:

`=` Outputs the result of an expression with **HTML** escaping (i.e., renders as plain text).

Example **PUG** code:

```
p= "Welcome to Date-a-Science!"
p= 3 + 4
p= "Your score is: " + (10 * 2)
p= loggedIn ? "Yes" : "No"
p= "<b>Not bold</b>"
p= "<script>alert("Saved")</script>"
```

Resulting **HTML** output:

```
<p>Welcome to Date-a-Science!</p>
<p>7</p>
<p>Your score is: 20</p>
<p>Yes</p>
<p>&lt;b&gt;Not bold&lt;/b&gt;</p>
<p>&lt;script&gt;alert(&quot;Daved&quot;)&lt;/script&gt;</p>
```

`-` Runs **JavaScript** but it outputs nothing directly. Use it for logic, variables, loops, etc..

Example **PUG** code:

```
- var greeting = "Hello, world!"
- var isLoggedIn = true
- var items = ["Apples", "Bananas", "Cherries"]
- var color = "red"

p= greeting

if isLoggedIn
  p Welcome back!
else
  p Please log in.

ul
  each item in items
    li= item

p(style="color:" + color)= "This text is red"
```

Resulting **HTML** output:

```
<p>Hello, world!</p>

<p>Welcome back!</p>

<ul>
  <li>Apples</li>
  <li>Bananas</li>
  <li>Cherries</li>
</ul>

<p style="color:red">This text is red</p>
```

`!=` Outputs the result of an expression without **HTML** escaping (i.e., renders raw **HTML**).

Example **PUG** code:

```
- var description = "<strong>Astrophysicist</strong> from MIT"
- var bio = "Loves stars<br>Enjoys dark matter theories"
- var video = "<iframe src="https://www.youtube.com/embed/19N1rUBA-Qk"></iframe>"
- var highlight = "<span class="highlight">Quantum Entanglement</span>"

p!= description
p!= bio
div.video!= video
p!= "Topic: " + highlight
```

Resulting **HTML** output:

```
<p><strong>Astrophysicist</strong> from MIT</p>
<p>Loves stars<br>Enjoys dark matter theories</p>
<div class="video">
  <iframe src="https://www.youtube.com/embed/19N1rUBA-Qk"></iframe>
</div>
<p>Topic: <span class="highlight">Quantum Entanglement</span></p>
```

We can also include variables by using:

> ***Interpolation*** **- inserting a variable or expression inside text or markup, so its evaluated result appears in the final output.**

This is often used for displaying user names, counts, prices, dates, etc., directly in sentences. We use the `#{}` notation to indicate that we want the variable's value inserted there as follows:

Example **PUG** code:

```
- let username = "Unknown"
p= "Username is " + username
p Username is #{username}
```

Resulting **HTML** output:

```
<p>Username is Unknown</p>
<p>Username is Unknown</p>
```

Notice that the first paragraph uses quotations since the **=** indicates that we are using **JavaScript** code. The second paragraph allows us to use plain text and have the variable inserted using interpolation. **HTML** element attributes are specified in parentheses beside the element:

```
<!-- HTML -->

<a href="/profile?id=123" class="profile-link">View Profile</a>

<div id="main" data-role="page"></div>

<form action="/submit" method="post">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name">
  <label for="age">Age:</label>
  <input type="number" id="age" name="age">

  <button type="submit">Submit</button>
</form>

<script src="/scripts/app.js" defer></script>

<img src="/images/avatar.jpg" alt="User Avatar" width="100" height="100">
```

```
// PUG

a(href="/profile?id=123" class="profile-link") View Profile

div(id="main" data-role="page")

form(action="/submit" method="post")
  label(for="name") Name:
  input(type="text" id="name" name="name")
  label(for="age") Age:
  input(type="number" id="age" name="age")

  button(type="submit") Submit

script(src="/scripts/app.js" defer)

img(src="/images/avatar.jpg" alt="User Avatar" width="100" height="100")
```

Classes and IDs can be expressed as attributes or by using a **.className** and **#idName** notation. Consider this **HTML** code that has many element attributes (i.e., light blue):

```
<!-- HTML -->

<div id="header" class="site-header" data-theme="dark">
  <h1 class="title">Welcome</h1>
  <p class="subtitle" id="intro-text">Thanks for visiting our site.</p>
</div>
<div class="card featured dark-theme"></div>
  <a href="/about" class="nav-link" id="about-link" target="_blank">About Us</a>
  <img src="/images/logo.png" alt="Site Logo" class="logo-img" id="main-logo" width="120">
</div>
<script src="/scripts/main.js" async></script>
```

Here they are expressed as attributes in **PUG**:

```
// PUG as attributes only (notice 2nd div has multiple classes)

div(id="header" class="site-header" data-theme="dark")
  h1(class="title") Welcome
  p(class="subtitle" id="intro-text") Thanks for visiting our site.

div(class="card featured dark-theme")
  a(href="/about" class="nav-link" id="about-link" target="_blank") About Us
  img(src="/images/logo.png" alt="Site Logo" class="logo-img" id="main-logo" width="120")

script(src="/scripts/main.js" async)
```

And here they are expressed using **.** and **#** notation, which goes after the element tag name:

```
// PUG using . and # instead (notice 2nd div has multiple classes hence multiple dots are used)

div#header.site-header(data-theme="dark")
  h1.title Welcome
  p.subtitle#intro-text Thanks for visiting our site.

div.card.featured.dark-theme
  a.nav-link#about-link(href="/about" target="_blank") About Us
  img.logo-img#main-logo(src="/images/logo.png" alt="Site Logo" width="120")

script(src="/scripts/main.js" async)
```

The | (pipe) operator can be used to start a line with plain text. It allows us to include multiple lines of text inside an **HTML** tag, while keeping the code readable and properly indented. It can make long text more readable.

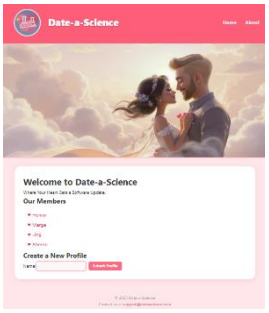
```
<!-- HTML -->
<p>
  This is the first sentence of the paragraph.
  It continues on a second line using the pipe operator.
  This helps maintain indentation and readability.
</p>
```

```
// PUG
p
  | This is the first sentence of the paragraph.
  | It continues on a second line using the pipe operator.
  | This helps maintain indentation and readability.
```

Without the pipe operator, we get errors:


```
// PUG – ERROR … thinks This, It and This are tags
p
   This is the first sentence of the paragraph.
   It continues on a second line using the pipe operator.
   This helps maintain indentation and readability.
```

Example:



Let's look at our **EJS** version of our **Date-a-Science** site and convert it to **PUG** code. One thing that I have noticed is that due to indentation rules in **PUG**, it gets complicated (with unexpected behavior) if we include the `html` and `body` tags in the header because it makes it difficult for nesting. Therefore, we will separate the **head** tag portion from the **nav** tag portion to make two separate partials. Here will be the `header.pug` file:

```
head
    title= title
    link(rel="stylesheet" href="/style.css")
    link(rel="icon" href="/images/favicon.jpg" type="image/jpeg")
```

… which is the portion that covered this part of the `header.ejs` file:

```
<head>
    <title><%= title %></title>
    <link rel="stylesheet" href="/style.css">
    <link rel="icon" href="/images/favicon.jpg" type="image/jpeg">
</head>
```

Notice that the `title` variable is being used, which we will need to pass in when we render the file.

Here will be the `nav.pug` file:

```
nav.navbar
    div.nav-left
        img.logo(src="/images/logo.jpg" alt="Logo")
        span.site-title Date-a-Science
    ul.nav-menu
        li
            a(href="/") Home
        li
            a(href="/about") About
```

… which is the portion that covered this part of the `header.ejs` file:

```
<nav class="navbar">
    <div class="nav-left">
        <img src="/images/logo.jpg" alt="Logo" class="logo">
        <span class="site-title">Date-a-Science</span>
    </div>
    <ul class="nav-menu">
        <li><a href="/">Home</a></li>
        <li><a href="/about">About</a></li>
    </ul>
</nav>
```

Here is the **`footer.pug`** partial:

```
footer
    p &copy; 2025 Date-a-Science
    p
        | Contact us at
        a(href="mailto:support@dateascience.com") support@dateascience.com
```

So, now we have our partials. Next, it is important to understand how **PUG** handles indentation when we use **include** statements.

In **PUG**, an **`include`** literally injects the included file's content at that exact indentation level where the **`include`** statement appears. The included file's content is treated as if it were written inline at that spot, respecting the indentation of the **`include`**. So, indentation matters because the included content will be nested inside whatever block or tag the **`include`** is inside.

Here is the converted **`index.pug`** file for the main homepage. We may notice that we took the **`doctype`**, **`html`** and **`body`** tags out from the **`header.ejs`** file and put them here. Notice how we included the three partials, which are highlighted:

```
doctype html
html
    include partials/header.pug
    body
        include partials/nav.pug
        div.banner
            img(src="/images/loveBanner.jpg" alt="Love Banner")

        div.container
            h1 Welcome to Date-a-Science
            p Where Your Heart Gets a Software Update.

            h2 Our Members
            ul.profile-list
                each person in people
                    li
                        a(href="/profile?id=" + person.id) &#x2665; #{person.name}

            h2 Create a New Profile
            label(for="name") Name:
            input#name(type="textbox")
            button(type="button" onclick="submit()") Submit Profile
```

```
            script(src="/clientscript.js")

        include partials/footer.pug
```

Notice that the first **include** is indented exactly once … which is where the **head** tag would be placed. By using **include**, it is as if we pasted the partial file right here at this level of indentation. Similarly, the **nav.pug** partial is placed within the **body** tag … which is where the navigation stuff would be placed. Lastly, the **footer.pug** partial is placed inside of the **body** and at the end of it. The indentation for these **include** statements is crucial for **PUG** to properly render the required **HTML** nesting.

It is also important that the **each person in people** code is indented within the **ul** tag and that the **li** is within that loop and that the **a** anchor tags are within the **li** tags.

Now look at the unordered **profile-list**. We are using **person.id** directly to get its value but then we use **person.name** within **#{ }** notation. Why the difference for these variables?

In the case of **person.id**, **PUG** treats the whole attribute value as **JavaScript**, so that variable gets evaluated directly. In the case of **person.name**, we are inside the text content of the tag, not inside an attribute. The **#{ }** tells **PUG** to evaluate the variable and insert it as escaped text at that position. So, the difference is that inside the attributes, it is treated as **JavaScript**, but outside it is treated as text unless we use **#{ }**.

It is also important to understand that variables behave differently in **PUG** than in **EJS**, especially when it comes to **includes**. In **EJS**, when we include the header partial like this:

```
<%- include("partials/header", { title: "Date-a-Science - Home" }) %>
```

We are passing the variable **title** just to that partial. If we don't pass it, the included file has no idea what **title** is. We have to send it manually each time. This is analogous to passing a parameter to a function.

However, in **PUG**, when we simply do this:

```
include partials/header.pug
```

No variables are specified because **PUG** automatically shares the current template's variables with the included file. So, as long as we pass the **title** to the **index.pug** file when we render it, then partials that we include (such as **header.pug**) automatically have access to the title variable. There is no need to pass it again. So, while things are simpler in **PUG**, we still need to be careful because since all variables are shared, naming conflicts are more likely to occur if we are not organized.

Here is the **about.pug** file converted from **about.ejs**:

```
doctype html
html
    include partials/header.pug
    body
        include partials/nav.pug
        div.container
            h1 About Date-a-Science
            p
                | Date-a-Science is a playful dating platform connecting
                | scientifically curious minds who believe in love through
                | logic, chemistry, and a dash of nerdy fun. Whether you are
                | into molecules or algorithms, we've got a match for you.

        include partials/footer.pug
```

We can see the same three included partials. Notice the use of `|` as well. This allows our text to look nice in the editor without wraparound. The rest is straight forward.

Finally, …here is the **`profile.pug`** file converted from **`profile.ejs`**:

```
doctype html
html
    include partials/header.pug
    body
        include partials/nav.pug
        div.container
            a.back-link(href="/") &lt; Back to homepage

            div.profile-card
                h1= person.name
                img(src=person.photo || anonImg, width="200", alt=person.name + "'s photo")
                p
                    strong Age:
                    = person.age

            div.messages
                h2 Messages
                if person.messages.length !== 0
                    each msg in person.messages
                        div.message= msg
                else
                    p No messages yet.


                h3 Leave a Message
                input#msg(type="textbox")
                button(type="button" onclick="submitMessage(" + person.id + ")") Submit Message

        script(src="/clientscript.js")

        include partials/footer.pug
```

Notice that the **IF** statement has the same indentation as the **`h2`** header. If we had accidentally indented it further, it would have been made part of the header, as well as all the messages under it. This would mess up our styles that we already created.

Notice in the `img` tag that we separated the attributes by commas, yet in the `button` tag we did not. Why the difference? Well, for the `button` tag, the values are complete strings, so **PUG** has no problem parsing it. But for the `img` tag, there is confusion due to the `||` code, so **PUG** would get confused without the commas. In fact, we could put the commas in all the time, if we would like. Sometimes, it is even clearer to put the items on separate lines (with commas) as follows:

```
            img(
                src=person.photo || anonImg,
                width="200",
                alt=person.name + "'s photo"
            )
```

PUG in the Server:

To use **PUG**, we need to ensure that it is first installed using **NPM**: `npm install pug` and then use require in our server code: `const pug = require("pug");`

Since **Pug** and **EJS** are both template engines, they serve the same purpose: converting data into **HTML** pages. However, the way they render templates differs slightly, especially in terms of flexibility and performance.

- **EJS** allows us to pass a callback function when rendering a template. This means rendering can be handled asynchronously, and we can even offload processing (like in a separate thread or worker if needed).
- **Pug**, on the other hand, doesn't natively support passing a callback for rendering. It renders synchronously (i.e., directly and immediately) which can be simpler … but limits threading options and might be less ideal for performance in high-load scenarios.

So, we have to do things a little differently. The callback function in **EJS** allowed us to be informed if there was an error during rendering (for whatever reason). Since **PUG** does not have the callback function, we need to take the code from **EJS**'s the callback function and put it in a `try/catch` block to handle the potential error. Here, for example, is a comparison of the `renderPage()` function from our **EJS** server code, and the **PUG** version alongside it:

```
// Render an EJS page

function renderPage(templatePath, res, data = {}) {
  ejs.renderFile(templatePath, data, (err, html) => {
    if (err) {
        console.error(err);
        res.writeHead(500);
        res.end("Internal Server Error");
    } else {
        res.writeHead(200, { "Content-Type":
                             "text/html" });
        res.end(html);
    }
  });
}
```

```
// Render a Pug page

function renderPage(templatePath, res, data = {}) {
  try {
    let html = pug.renderFile(templatePath, data);
    res.writeHead(200, { "Content-Type":
                         "text/html" });
    res.end(html);
  } catch (err) {
    console.error(err);
    res.writeHead(500);
    res.end("Internal Server Error");
  }
}
```

So, the callback's `err` parameter becomes the `catch` block parameter and its `html` parameter becomes a local variable. The rest of the code is the same. Other than this function, the remaining server code is the same, with the exception that we alter the file names to have the `.pug` extension instead of `.ejs`.

Instead of rendering the **PUG** template from scratch every time a request comes in, we can compile it once using `pug.compileFile()`. What does it mean to "compile it"? Well, the function will process the template to determine which **HTML** parts are static and which parts are dynamic (i.e., will change with incoming data). That way, when the function is called, it will not re-render the static parts of the **HTML** … just the dynamic parts. This means that we will save time rendering if the function needs to be called more than once but with different data (e.g., when we have many users viewing the same page layout but with personalized content).

Here, for example, is a side-by-side comparison showing how a **PUG** file would be compiled:

```
// code from views/profile.pug

doctype html
html
  include partials/header.pug

  body
    include partials/nav.pug
    div.container
      a.back-link(href="/") &lt; Back to homepage

      div.profile-card
        h1= person.name
        img(
            src=person.photo || anonImg,
            width="200",
            alt=person.name + "'s photo"
        )
        p
          strong Age:
          = person.age

       div.messages
         h2 Messages
         if person.messages.length !== 0
           each msg in person.messages
             div.message= msg
         else
           p No messages yet.

         h3 Leave a Message
         input#msg(type="textbox")
         button(
                type="button"
                onclick="submitMessage(" + person.id + ")"
         ) Submit Message

        script(src="/clientscript.js")

        include partials/footer.pug
```

Assume that we did this:

```
const f = pug.compileFile("views/profile.pug");
```

The **HTML** string on the right would be "hard-

```
<!DOCTYPE html>
<html>
  <head>
    <title>Date-a-Science</title>
    <link rel="stylesheet" href="/style.css">
    <link rel="icon" href="/images/favicon.jpg"
          type="image/jpeg">
  </head>
  <body>
    <nav class="navbar">
      <div class="nav-left">
        <img class="logo" src="/images/logo.jpg" alt="Logo">
        <span class="site-title">Date-a-Science</span>
        <ul class="nav-menu">
          <li><a href="/">Home</a></li>
          <li><a href="/about">About</a></li>
        </ul>
      </div>
    </nav>

    <div class="container">
      <a class="back-link" href="/">&lt;
                      Back to homepage</a>

      <div class="profile-card">
        <h1>person.name</h1>
        <img
          src="person.photo || anonImg"
          width="200"
          alt="person.name + ' photo'">
        <p>
          <strong>Age:</strong> person.age
        </p>
      </div>

      <div class="messages">
        <h2>Messages</h2>

        if person.messages.length !== 0
          each msg in person.messages
            <div class="message">msg</div>
        else
          <p>No messages yet.</p>

        <h3>Leave a Message</h3>
        <input id="msg" type="textbox">
        <button type="button"
            onclick="submitMessage(person.id)">
            Submit Message</button>
      </div>
```

coded" into the function `f`, with the exception of the yellow highlighted code, which would be rendered dynamically when we call `f` with data. The data being a single **JavaScript** object.

```
    </div>

    <script src="/clientscript.js"></script>

    <footer>
      <p>&copy; 2025 Date-a-Science</p>
    </footer>
  </body>
</html>
```

As we can see, most of the produced code is static and unchanging, so it need not be rendered each time the function is called.

We can do this pre-computation upon startup by adding this code to our **`server.js`**:

```
// Precompiled templates - done at startup
const compiledTemplateFunctions = {};

["./views/index.pug", "./views/about.pug", "./views/profile.pug"].forEach(name => {
  compiledTemplateFunctions[name] = pug.compileFile(`${name}`);
});
```

Then we could re-write our **`renderPage()`** function as follows:

```
// Render function that selects the right compiled template
function renderPage(templatePath, res, data = {}) {
  try {
    const renderFn = compiledTemplateFunctions[templatePath];
    if (!renderFn) {  // Make sure that things pre-compiled ok
      res.writeHead(404);
      res.end("Template Not Found");
      return;
    }
    const html = renderFn(data); // Call the precompiled function

    res.writeHead(200, { "Content-Type": "text/html" });
    res.end(html);
  } catch (err) {
    console.error(err);
    res.writeHead(500);
    res.end("Internal Server Error");
  }
}
```

That's all for now. To really get the hang of this, we need to practice more.

Here are some more links if you want to learn more about **EJS** or **PUG** syntax:

https://ejs.co/

https://pugjs.org/api/getting-started.html

https://codepen.io/mimoduo/post/learn-pug-js-with-pugs

Here are some ideas for expanding our **Date-a-Science** site. Feel free to practice adding some of these features to ensure that you understand how to do them.

1. **Add timestamps on messages**
   - Just change the message from a string to an object. E.g., `{ text, time }`
   - Then modify the display logic in `profile.ejs` or `profile.pug`.

2. **Emoji or reaction picker**
   - Add simple emoji buttons next to messages (e.g., ❤️, 😂, 👍, 🔥, 😍).
   - No backend logic is needed if these are stored on the client-side or in memory.

3. **Add a "Like" button (with counter) on profile pages**
   - Add a button (e.g., ❤️ or 👍) and increment a like counter.
   - Display the number of likes somewhere (e.g., "❤️ 5 Likes")
   - Make sure to send the “like” to the server for storage

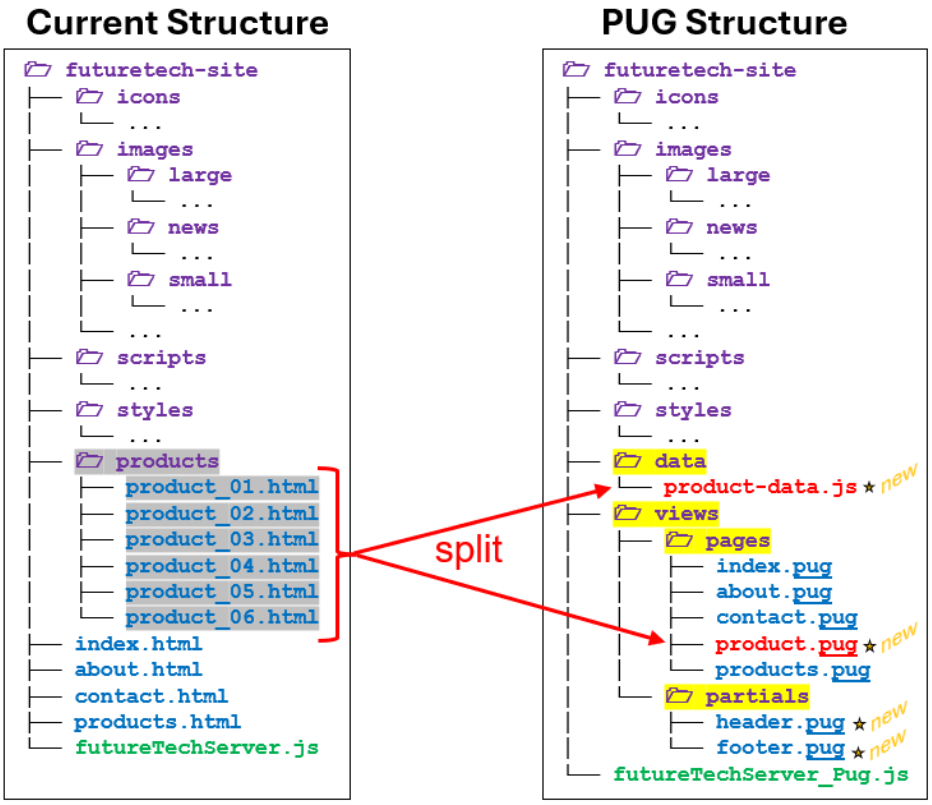
4. **Create a "Daily Featured Profile"**
   - Randomly pick one person either upon server start or daily.
   - Display that person at the top of `index.ejs` or `index.pug`.

5. **Keep track of current user** (e.g., pretending a member logged in)
   - Let the user choose who they are from a dropdown list on the homepage: "Select Your Profile".
   - Maintain track of who the user is by storing the selected ID in a query parameter (e.g., ?userId=3) passed onto all pages

6. **Compute Match Percentage** (assume current user is selected from above)
   - Add a few profile questions (e.g., "Do you like hiking?", “Quiet Evenings?”, etc..).
   - Store each person’s answers as an array: (e.g., ["yes", "no", "yes", "yes"]).
   - Compare the answers of two profiles, count how many match, return a percentage.
   - Add calculated "match score" to each profile view against the current user.
   - Show top few match scores

9.4 Updating the FutureTech Corp. Site

Let's adjust our **FutureTech Corp.** site to use **PUG**. The first step is to decide how we will organize our files. Below, on the left is our current folder structure and on the right is how we will restructure things:



Since we will no longer have static product pages, we will remove those html files and replace them with a single **product.pug** page. At the same time, we will extract the product data from the static pages and encode it into an array in the **product-data.js** file. We will be changing our **.html** page files to have **.pug** extensions. We will also extract the common header and footer stuff and place them in their own files. Lastly, we will adjust the folder structure to be more organized with the use of the **data**, **views**, **view/pages** and **views/partials** folders.

Let's start converting by looking at the shared header on each page. We take this:

```
<header>
  <img src="images/logo.jpg" height=200 alt="FutureTech Corp. Logo">
  <nav>
    <ul>
      <li><a href="index.html" class="active">Home</a></li>
      <li><a href="about.html">About Us</a></li>
      <li><a href="products.html">Products</a></li>
      <li><a href="contact.html">Contact Us</a></li>
    </ul>
  </nav>
</header>
```

And convert to this, saving it to **/partials/header.pug**:

```
header
  img(src="/images/logo.jpg" height=200 alt="FutureTech Corp. Logo")
  nav
    ul
      li
        a(href="/" class="#{currentPage === 'home' ? 'active' : ''}") Home
      li
        a(href="/about" class="#{currentPage === 'about' ? 'active' : ''}") About Us
      li
        a(href="/products" class="#{currentPage === 'products' ? 'active' : ''}") Products
      li
        a(href="/contact" class="#{currentPage === 'contact' ? 'active' : ''}") Contact Us
```

The main difference is that we will now need a variable (i.e., `currentPage`) to decide if the navigation link should be highlighted (i.e., `'active'`). Why? Well, in our static page we hard-coded `class='active'` for the specific link on each page. But now all pages will share this header, so we need a variable to indicate the page we are on, so we can highlight for that page only.

The common footer on each page is easy to adjust as well. We take this:

```
<footer>
  <p>&copy; 2025 FutureTech. All rights reserved.</p>
  <div class="status-bar">
    <div id="clock">Loading time...</div>
    <button id="darkModeToggle">Switch to Dark Mode</button>
  </div>
</footer>

<script src="scripts/time-clock.js"></script>
<script src="scripts/dark-mode.js"></script>
```

and then convert it to this, saving it to **/partials/footer.pug**:

```
footer
  p &copy; 2025 FutureTech. All rights reserved.
  div.status-bar
    div#clock Loading time...
    button#darkModeToggle Switch to Dark Mode

script(src="/scripts/time-clock.js")
script(src="/scripts/dark-mode.js")
```

Nothing new here, except to point out that we only added the two scripts that were specific to the footer, to the **/partials/footer.pug** file. Now we can do the main **/views/pages/index.pug** page:

```
DOCTYPE=html
html(lang="en")

head
  meta(charset="UTF-8")
```

```
  title FutureTech - The Future is Now
  link(rel="icon" href="/icons/logo-icon.png")
  link(rel="stylesheet" href="/styles/image-slider.css")
  link(rel="stylesheet" href="/styles/general-body.css")
  link(rel="stylesheet" href="/styles/header-footer.css")


body
  include ../partials/header
  br
  div#quote(style="text-align:center; font-style:italic; padding: 10px;")
  div#quoter(style="text-align:center; font-style:italic;")

  section.slider-container
    div.slider#imageSlider
      div.slide
        img(src="/images/news/scientists.jpg" alt="Molecular Stabilizer Lab")
        div.caption Molecular Stabilizer Lab
      div.slide
        img(src="/images/news/robotics.jpg" alt="Design and Manufacturing Lab")
        div.caption Design and Manufacturing Lab
      div.slide
        img(src="/images/news/portals.jpg" alt="Portal Experimentation Lab")
        div.caption Portal Experimentation Lab
      div.slide
        img(src="/images/news/android.jpg" alt="Android Development Lab")
        div.caption Android Development Lab

    div.dots#dotsContainer
      span.dot.active(data-slide="0")
      span.dot(data-slide="1")
      span.dot(data-slide="2")
      span.dot(data-slide="3")

  include ../partials/footer

  // All the JavaScripts to run
  script(src="/scripts/quotes.js")
  script(src="/scripts/slider-script.js")
  script(src="/scripts/captions.js")
```

When including the header & footer, we must indicate where to find it by backing up a directory first … otherwise the server won't find them because they are not in the **/view/pages** folder with this file.

The **about.html** is similarly converted to **/views/pages/about.pug** … nothing exciting here:

```
DOCTYPE=html
html(lang="en")
  head
    meta(charset="UTF-8")
    title About Us
    link(rel="icon" href="/icons/logo-icon.png")
    link(rel="stylesheet" href="/styles/general-body.css")
    link(rel="stylesheet" href="/styles/header-footer.css")

  body.about-page
```

```
  include ../partials/header

  main
    section
      h2 About Us
      div(style="text-align: center;")
        img(src="/images/headquarters.jpg" alt="FutureTech Corp. Headquarters" width=600)

      div.about
        p
          | At FutureTech Corp., we don't just imagine the future, we invent it.
          | ...

        p
          | Over the past ten years, FutureTech Corp. has combined breakthrough
          | ...

        p
          | As a global leader in high-tech development, we remain committed to
          | ...

  include ../partials/footer
```

Also, the **contact.html** file is easily converted to **/views/pages/contact.pug** …

```
DOCTYPE=html
html(lang="en")
  head
    meta(charset="UTF-8")
    title Contact Us
    link(rel="icon" href="/icons/logo-icon.png")
    link(rel="stylesheet" href="/styles/general-body.css")
    link(rel="stylesheet" href="/styles/header-footer.css")

  body
    include ../partials/header

    main
      section
        h2 Contact Us

        table.contact-table
          tr
            td
              h3 Our Office
              p
                strong FutureTech Corp.
                br
                | 4567 Hyperion Avenue
                br
                | Sector 9, NeoCity, CA 94018
                br
                | United States
              p
                strong Phone:
                | (555) 013-2048
```

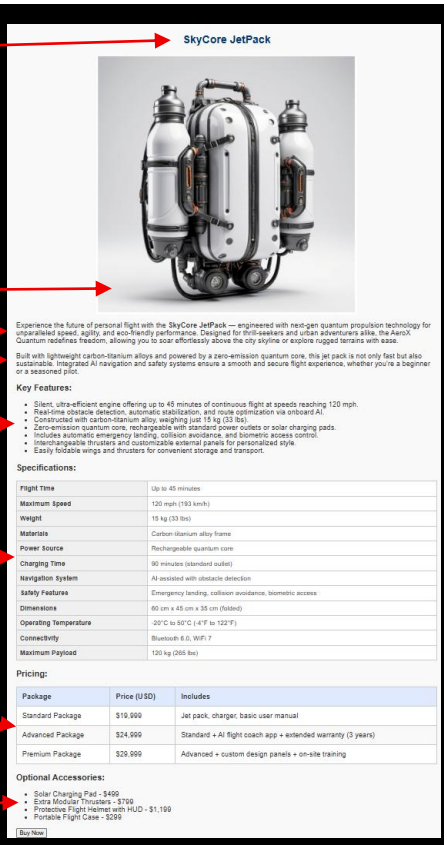
```
            br
            strong Email:
            a(href="mailto:support@futuretechcorp.com") support@futuretechcorp.com
        td
          img(src="/images/topview.jpg" width=500 alt="FutureTech Corp. Headquarters")
      tr
        td.contact-info
          div.map-container
            h3 Find Us
            iframe(
              src="https://maps.google.com/maps?q=37.7749,-122.4194&z=14&output=embed",
              width="100%",
              height="300",
              style="border:0;",
              allowfullscreen="",
              loading="lazy",
              title="FutureTech Corp Map")

        td.contact-form-box
          p
            | Feel free to get in touch using the form on the right
            | or through the contact information below.

          form(action="/contact/message" method="POST" target="")
            label(for="name") Name:
            br
            input#name(type="text" name="name" required)
            br
            br
            label(for="email") Email:
            br
            input#email(type="email" name="email" required)
            br
            br
            label(for="message") Message:
            br
            textarea#message(name="message" rows="5" required)
            br
            br
            button(type="submit") Send Message

include ../partials/footer
```

Notice … we set the action to `"contact/message"` so that it will now contact our server and we will be able to handle it (more on this later). We also set `target` to `""` to stay on the same page after.

The conversion of **products.html** to **/views/pages/products.pug** will now require some additional coding because it used to simply contain hard-coded links to individual static product pages but now we need to go to just one common page (i.e., **/views/pages/product.pug**). Also, we need to make sure all the products are shown on this page in a dynamic way. That means, this page will now require an incoming variable (we will call it **products**) representing the array of products. We will also need to give each product an **id** so that when we select that product to view its page, we will be able to tell the server which product we want it to render on that page.

Let's first discuss the **/data/product-data.js** file which will simply contain an array with all the product data (note: in the future, we will replace this with some database logic). We will need to make an object to represent the product. If we were to examine the product pages that we created, we would be able to identify the following common information for each product:

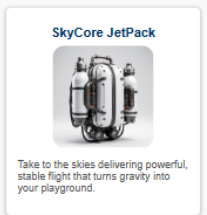
```
{
   name: "",
   id: 1,
   imageSmall: "?.jpg",
   desc: "...",
   imageAlt: "...",

   image: "?.jpg",
   par1: "...",
   par2: "...",
   features: [
      {feature: "...", desc:"..."},
      {feature: "...", desc:"..."},
      ... ],
   specs: [
      {key: "...", value: "..." },
      {key: "...", value: "..."},
      ... ],
   pricing: [
      {option: "...", price: 99.00, includes: "..."},
      {option: "...", price: 99.00, includes: "..."},
      ... ],
   accessories: [
      {option: "...", price: 99.00},
      {option: "...", price: 99.00},
      ... ]
}
```



So, we just need an array of the above product objects in our **/data/product-data.js** file that we export as follows:

```
export const products = [
    { /* product 1 object */ },
    { /* product 2 object */ },
    { /* product 3 object */ },
    { /* product 4 object */ },
    { /* product 5 object */ },
    { /* product 6 object */ }
];
```

Now we can look at the **/views/pages/products.pug** page. We will assume that the `products` variable will be passed into this page when we render it. This page will now need to take the `products` array and loop through it to create an **HTML** ** ** for each one. Within the ** **, we will need to have a header, an image and a short description so that we can make it look as shown here on the right.



Assume that we loop through each product in `products`. Then for each product `p`, we can access the attributes that we need to render this page (i.e., `p.name`, `p.imageSmall`, `p.imageAlt` and `p.desc`). We will also need the `p.id` so that we can place a unique `ID` on each rendered product so that when the user selects the product, we can send that `ID` to the **/views/pages/product.pug** page to render the correct product for that page. Here is the code for the **/views/pages/products.pug** page:

```
DOCTYPE=html
html(lang="en")
  head
    meta(charset="UTF-8")
    title Our Products
    link(rel="icon" href="/icons/logo-icon.png")
    link(rel="stylesheet" href="/styles/general-body.css")
    link(rel="stylesheet" href="/styles/products-style.css")
    link(rel="stylesheet" href="/styles/header-footer.css")

  body
    header
      include ../partials/header

    main
      section
        h2 Our Products
        div.product-list
          each p in products
            div.product(id="product#{p.id}")
              h3 #{p.name}
              a(href="product?id=" + p.id)
                img(src="/images/small/" + p.imageSmall alt=p.imageAlt width="150")
              p #{p.desc}

    include ../partials/footer
```

Notice how we use **PUG** interpolation to insert the product name into the header and description into the paragraph. We also use it to insert the product id number so that the ** ** id becomes `"product1"`, `"product2"`, etc.. Lastly, we use inline **JavaScript** to produce the correct image file path.

Finally, we need to make the **/views/pages/product.pug** file to represent the page that will render a specific product. To render this page we will assume that a particular product has been passed in as a variable called `prod`. Here is the code … I highlighted the important things. We will notice that there are **4** loops now, since we need to loop through product features, specs, pricing and accessories.

```
DOCTYPE=html
html(lang="en")
  head
    meta(charset="UTF-8")
    title Product - FutureTech Corp.
    link(rel="icon" href="/icons/logo-icon.png")
    link(rel="stylesheet" href="/styles/header-footer.css")
    link(rel="stylesheet" href="/styles/general-body.css")
```

```
    link(rel="stylesheet" href="/styles/details.css")
  body
    include ../partials/header
    main
    section.product-detail
      h2 #{prod.name}
      div(style="text-align: center;")
        img(src="/images/large/" + prod.image alt=prod.imageAlt)
      p!= prod.par1  ← != needed to render <strong> tag in the data
      p!= prod.par2

      h3 Key Features:
      ul
        each f in prod.features
          li
            f.feature= f.desc

      h3 Specifications:
      table.specs-table
        tbody
          each s in prod.specs
            tr
              th= s.key
              td= s.value

      h3 Pricing:
      table.pricing-table
        thead
          tr
            th Package
            th Price (USD)
            th Includes
        tbody
          each pr in prod.pricing
            tr
              td= pr.option
              td= "$" + pr.price.toLocaleString("en-US", { minimumFractionDigits: 0,
maximumFractionDigits: 0 })
              td= pr.includes

      h3 Optional Accessories:
      ul
        each a in prod.accessories
          li= a.option + " - $" + a.price.toLocaleString("en-US", { minimumFractionDigits: 0,
maximumFractionDigits: 0 })

      button.buy-button Buy Now

    include ../partials/footer
```

The use of `toLocaleString` when displaying money has a two-fold purpose. It will allow commas to be displayed (e.g., $499,000) and it will also allow us to specify the number of decimal places (in our case we just want integers shown).

The pages are done! Now we need to adjust the server so that it renders the pages and so that we pass in the needed variables when doing so.

Currently, our server accepts a request for any file (which were all static) and sends it back by using the following code:

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

  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);
  });
}
```

Now that we are using dynamic pages, we will need to handle each page. Below is the top portion of our new `requestListener`. It no longer looks for absolute paths by using `__dirname`, but instead will look for each page relative to where the server started. Since we will render each page, we will need to extract the pathname from the request **URL**. We use `url.parse()` to extract (i.e., separate) the `pathName` from the `query` string.

```
// Handle incoming requests
function requestListener(req, res) {
    let parsedUrl = url.parse(req.url, true); // true parses query string
    let pathName = parsedUrl.pathname;
    let query = parsedUrl.query;

    // Serve static files if the URL matches typical file folders
    if (pathName.startsWith("/styles/") || pathName.startsWith("/images/") ||
        pathName.startsWith("/icons/") || pathName.startsWith("/scripts/")) {
        serveStaticFile(req, res);
        return;
    }
    ... more to come ...
}
```

Notice that we first look for the static files by examining where the files are located in the path. If it is in one of the **styles**, **images**, **icons** or **scripts** folders, then we know that it will be a static file and we can serve that file easily.

To serve the static file, we call the function shown in the chunk of code below:

```
function send404(response){
    response.statusCode = 404;
    response.write("Not Found");
    response.end();
}
```

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

// Serve static files as requested
function serveStaticFile(req, res) {
    const filePath = path.join(__dirname, req.url);
    const ext = path.extname(filePath);
    const contentType = mimeTypes[ext] || "application/octet-stream";

    fs.readFile(filePath, (err, data) => {
        if (err) {
            send404(res);
        } else {
            res.writeHead(200, { "Content-Type": contentType });
            res.end(data);
        }
    });
}
```

As we did previously, it just looks at the file extension so that it can create the correct **Content-Type** header attribute. Then it reads the file and sends it back as before, or sends back a **404** error (we made a helper function for this). There is nothing really new here, since there is no dynamic content.

Let's continue with the remainder of the `requestListener` function. It will need to look at the `pathName` and decide which page to render:

```
// Handle incoming requests
function requestListener(req, res) {
    ...
    // Render PUG pages
    switch (pathName) {
        case "/":
        case "/home":
        case "/index":
            renderPage("./views/pages/index.pug", res, "home");
            break;
        case "/about":
            renderPage("./views/pages/about.pug", res, "about");
            break;
        case "/contact":
            renderPage("./views/pages/contact.pug", res, "contact");
            break;
        case "/products":
            renderPage("./views/pages/products.pug", res, "products");
            break;
        case "/product":
            if (query.id)
                renderProductPage("./views/pages/product.pug", res, query.id);
```

```
            else
                send404(res);
            break;
        default:
            send404(res);
    }
}
```

Notice a couple of things. First, all pages are rendered with the same function, which we will discuss in a moment, except for the one for rendering a particular product (which has a different function that will require the id of the product to be rendered, which we get from the extracted `query.id`).

Notice as well that the pages to be rendered are specified relative to the current directory (i.e., `"./"`). Now … let's look at the functions to render the pages. We will pre-compile the pages as we did with our **Date-a-Science** site:

```
// Precompiled templates - done at startup
const compiledTemplateFunctions = {};
["./views/pages/index.pug", "./views/pages/about.pug", "./views/pages/contact.pug",
 "./views/pages/product.pug", "./views/pages/products.pug"].forEach(name => {
  compiledTemplateFunctions[name] = pug.compileFile(`${name}`);
});
```

We will also render the pages the same way as our **Date-a-Science** site:

```
// Render a PUG file as requested
function renderPage(fileName, res, pageName) {
    try {
        const renderFn = compiledTemplateFunctions[fileName];
        if (!renderFn) { // Make sure that things pre-compiled ok
            res.writeHead(404);
            res.end("Template Not Found");
            return;
        }
        const html = renderFn({products: products, currentPage: pageName});
        res.writeHead(200, { "Content-Type": "text/html" });
        res.end(html);
    } catch (err) {
        console.error(err);
        res.writeHead(500);
        res.end("Internal Server Error");
    }
}
```

Notice in the pre-compiled render function (i.e., `renderFn`) we pass in an object that contains `products` and `currentPage` attributes. The products will be from our `products` array that we exported from our **product-data.js** file, so we will need this at the top of our server code:

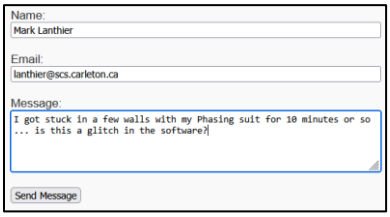
```
const { products } = require("./data/product-data.js");
```

The `currentPage` is set to the `pageName` that was passed in as a parameter from our routing code in the `requestListener` function (e.g., `"./views/pages/about.pug"`).

The call to this `renderFn()` will do the **PUG** rendering for us by producing the dynamic pages that we need, based on the passed-in parameters 😊. In a similar way, we can write the function that will render an individual product page:

```
// Render a product PUG file as requested
function renderProductPage(fileName, res, id) {
    try {
        const renderFn = compiledTemplateFunctions[fileName];
        if (!renderFn) { // Make sure that things pre-compiled ok
            res.writeHead(404);
            res.end("Template Not Found");
            return;
        }
        const html = renderFn({prod: products[id-1], currentPage: "products"});
        res.writeHead(200, { "Content-Type": "text/html" });
        res.end(html);
    } catch (err) {
        console.error(err);
        res.writeHead(500);
        res.end("Internal Server Error");
    }
}
```

Notice that the `id` is passed in as a parameter. It was obtained from the query id value (i.e., **`http://localhost:3000/product?id=2`**). Otherwise the code should be easily understood at this point. So, now we have our converted **PUG** server code for our site.



At this point, though, our **Contact Us** page sends our form to

`http://localhost:3000/contact/message`

At the moment, our server doesn't handle the post, so let's add that now. What do we do with the data? We will store the data in an array, and assume that someone will process them later. Ultimately, a database would be used to store these. Let's add this to our **product-data.js** file:

```
// Store all incoming messages like this:
// messages = [
//      {name: "Mark", email: "mark@somewhere.com", message: "This is my message."}
//      {name: "Jen", email: "jen@somewhere.com", message: "My message is better."}
//      ...
// ];
export const messages = [  ];
export   let messageCount = 0;
```

Then we will add this to the top of our server code:

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

Now we have a location for the incoming messages from our **Contact Us** form. We just need to handle the **POST** requests coming in from "contact/message" (which is the `action` that we set for our form).

We can add this **`case`** to our **`switch`** statement at the end of our **`requestListener()`** function:

```
  case "/contact/message":
     handlePost(req, res);
  break;
```

and add this function:

```
function handlePost(req, res) {
    console.log("got it");
}
```

We can do a quick test to see if it worked by going to **`http://localhost:3000/contact/message`**, filling out the form, and then submitting it. We should see the “got it” message appear, but the browser will hang because we are not sending anything back yet.

Now, let’s store the data and send back an **OK** by adjusting the **`handlePost()`** function. We will need to read the body, so we will use the same helper code as our **Date-a-Science** site to do this:

```
// Helper function to read the request body
function readRequestBody(req, callback) {
    let body = ""
    req.on("data", (chunk) => {
        body += chunk;
    })
    //Once the entire body is ready, process the request
    req.on("end", () => {
        callback(body);
    });
}
```

Now let’s adjust the **`handlePost()`** to store the incoming message. With a **POST**, the data is sent in the request’s body. It will be a **URL**-encoded string like this, for example:

> "name=Mark%20Lanthier&email=lanthier%40scs.carleton.ca&message=I%20got%20stuck%20in%20a%20few%20walls%20with%20my%20Phasing%20suit%20for%2010%20minutes%20or%20so%20%E2%80%A6%20is%20this%20a%20glitch%20in%20the%20software%3F"

We will parse this into an object by making use of the **Node.js** **`querystring`** module, so we add this to the top of our code:

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

Then we can convert this string into a message object that we can store in our array by using the `parse` function. Once we have the object, we can add it to our messages, then send back an acknowledgement page:

```
// Handle the message post
function handlePost(req, res) {
    readRequestBody(req, function(body) {
        const mesgObj = querystring.parse(body);
        messages.push(mesgObj);
        messageCount++;
        renderPage("./views/pages/messageReceived.pug", res, "contact/message")
    });
}
```

The message being returned is stored in **/views/pages/messageReceived.pug**. We can pre-compile it as we did with the others by adding it to our initializing code:

```
const compiledTemplateFunctions = {};
["./views/pages/index.pug", "./views/pages/about.pug", "./views/pages/contact.pug",
 "./views/pages/product.pug", "./views/pages/products.pug",
"./views/pages/messageReceived.pug"].forEach(name => {
  compiledTemplateFunctions[name] = pug.compileFile(`${name}`);
});
```

The page itself is simple in that it just acknowledges that the message was received at the server:

```
DOCTYPE=html
html(lang="en")
  head
    meta(charset="UTF-8")
    title Thank You
    link(rel="icon" href="/icons/logo-icon.png")
    link(rel="stylesheet" href="/styles/general-body.css")
    link(rel="stylesheet" href="/styles/header-footer.css")

  body.received-page
    include ../partials/header

    main
      section
        div.mrecv
          h1 Your message has been received
          p Please wait up to 5 business days for a reply.

    include ../partials/footer
```

Although we will add this to the **general-body.css** to indent the text a little:

```
/* For the "message received" page */
.mrecv {
  margin-left: 40px;  /* push it right from the left */
}
```