Publishing Web Pages

This lesson teaches how to publish and return web pages from a custom web server.

Finally, let's learn how to serve HTML content so that your web server can come into its own. For example, `GET` HTTP requests to the `"/hello"` route should show a basic web page. A naive way to do so would be to simply return an HTML string.

Output

HTML

```
// Return HTML content for requests to "/hello"
app.get("/hello", (request, response) => {
  const htmlContent = `<!doctype html>
    <html>
    <head>
      <meta charset="utf-8">
      <title>Hello web page</title>
    </head>
    <body>
      Hello!
    </body>
    </html>`;
  response.send(htmlContent);
});
```

However, things would quickly get out of hands as the complexity of the web page grows. A better solution is to define the HTML content in an external file stored in a dedicated subfolder, and return that file as a result of the request. For example, create a subfolder named `views` and a file named `hello.html` inside it. Give the HTML file the following content.

Output

HTML

```
<!doctype html>
<html>

<head>
    <meta charset="utf-8">
    <title>Hello web page</title>
</head>

<body>
    <h2>Hello web page</h2>
    <div id="content">Hello!</div>
</body>

</html>
```

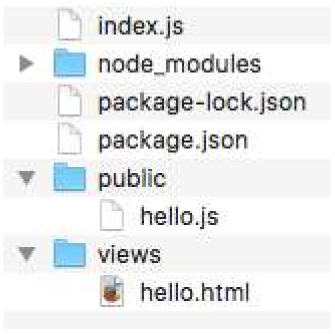
Then, update the callback for the `"/hello"` route to send the HTML file as the request response.

```
// Return a web page for requests to "/hello"
app.get("/hello", (request, response) => {
response.sendFile(`${__dirname}/views/hello.html`);
});
```

Pointing your browser to the `"/hello"` URL (http://localhost:3000/hello if your server runs locally) should now display the web page. Most web pages will need to load client-side resources such as images, CSS and JavaScript files. A common practice is to put these assets in a dedicated subfolder. For example, create a `public` subfolder and a `hello.js` JavaScript file inside it with the following content.

```
// Update the "content" DOM element
document.getElementById("content").textContent = "Hello from Java
Script!";
```

You should now have the following folder structure for your server.


```
index.js
node_modules
package-lock.json
package.json
public
    hello.js
views
    hello.html
```

Update the `hello.html` to load this JavaScript file.

> ```
> <script src="/hello.js"></script>
> ```

Lastly, you must tell Express that client assets are located in the `public` subfolder, so that the server can serve them directly. Add the following code towards the beginning of your main application file.

> // Serve content of the “public” subfolder directly
> app.use(express.static(“public”));

Accessing the `"/hello"` URL shows you a slightly different result. The `hello.js` file was loaded and executed by the browser, updating the web page content.

In this example, JavaScript was used both for back-end (server side) and front-end (client side) programming. This is one of its core strengths: knowing only one programming language empowers you to create complete web applications. How great is that?