

• 🕒 2 hours

HTTP Full-Stack Project

In this project, you are going to use Node.js to build a data-driven Web site. This project already includes the Sequelize models and migrations for you. You will create a Node.js HTTP server and use it to handle incoming requests from a browser. Then, you will generate HTML to respond to the request.

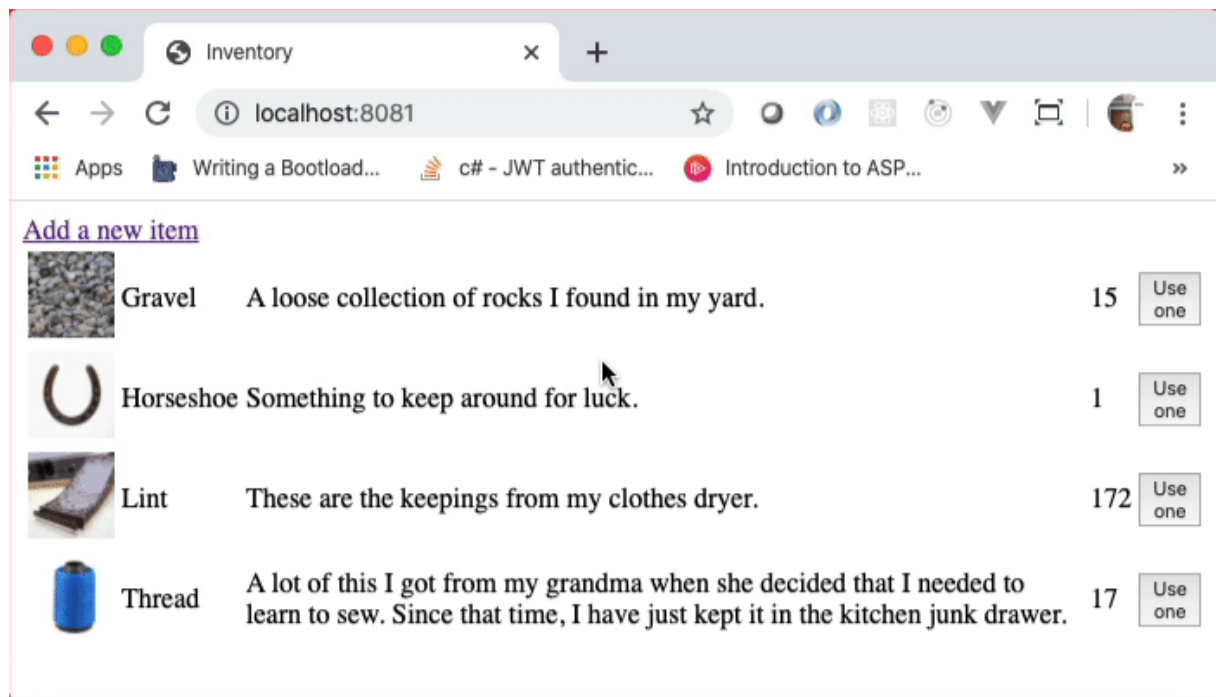
Today's project does not address the aesthetics of the visual appearance of the Web pages. You will have an opportunity later this week to do that. Today is about *functionality*.

Project overview

You will build a simple inventory tracking system for managing the amount of stuff that you have. The Sequelize data model is already created for you because you now know how to do that pretty well. You'll get to flex those muscles later this week, too.

You will build the server that accepts incoming HTTP requests using *only* functionality built into Node.js. You will process the incoming request, determine what needs to be done, and generate HTML to send back to the client.

This project shows you the underpinnings of how Node.js-based Web applications work. Then, when you use a framework like Express.js or Koa.js, you will know what they're doing.



The data model

To focus on the server portion of this, the data model is very simple. It consists of one entity, the Item. The Item has the following properties.

Property name	Data type	Constraints
name	string	not null, unique
description	text	not null
imageName	string	
amount	integer	not null, default 0

The functionality

You will create two HTML pages, one static and one dynamic. The static HTML page will consist of a form that allows you to add new items that you want to track. The dynamic HTML page will list the each item and its details and give you a way to reduce the amount on hand.

Get started

- Clone the starter repository from <https://github.com/appacademy-starters/node-web-app-starter>. But, this time, use an extended version of the Git `clone` command to put it in a specific directory. You will use the same starter project in the next project, too.

```
git clone https://github.com/appacademy-starters/node-web-app-starter native-node-app
```

Instead of creating a directory named after the repository, "node-web-app-starter", this will create a directory named "native-node-app" and put the cloned repository into there.

- Change the working directory into "native-node-app"
- Install the npm dependencies
- Create a database user named "native_node_app" with the password "oMbE4FNk3db2LwFT" and the CREATEDB privilege which will look like

```
CREATE USER ... WITH CREATEDB PASSWORD ...
```

You add the CREATEDB in there so you can do the next step and not be bothered with creating the database yourself

- Run the Sequelize CLI with the `db:create` command to create the database
- Run the Sequelize CLI to migrate the database
- Run the Sequelize CLI to seed the database

Phase 1: Installing one tool

You will use a development tool to restart the server each time you make a change to a JavaScript file. This prevents you from having to hit CTRL+C each time you want to stop and start your server.

The tool is named **nodemon** and is the standard for this type of server restarting. It is a *development* tool, so you will install it as a special kind of dependency, a *development dependency*. You can do that with

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

When you deploy your application to production, npm will ignore the development dependencies because they're not needed when you run your application for other people to use. Hopefully by that point, your Web application *doesn't restart!*

Phase 2: Getting the server started

Open the **package.json** file. It specifies that the "main" file for this project is **server/index.js**. Create a **server** directory and an **index.js** file in there.

Now, in **package.json**, find the "scripts" section. Add a new entry in there named "dev" with the value "nodemon server/index.js". It should look like this.

```
"scripts": {  
  "dev": "nodemon server/index.js",  
  "test": "echo \"Error: no test specified\" && exit 1"  
},
```

That sets up a way to conveniently run the "nodemon" command by typing the command `npm run dev`. You can run that right now. Because you have an empty **server/index.js** file, it should report something like this:

```
[nodemon] starting `node server/index.js server/index.js`  
[nodemon] clean exit - waiting for changes before restart
```

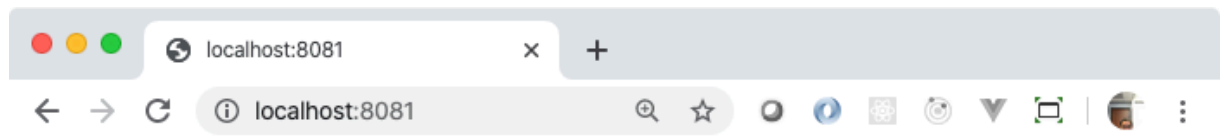
So, it's just waiting for you to add some code!

To get an HTTP server up and running, you will add code to do the following in the **server/index.js** file.

- Import the built-in "http" module
- Create a server using the "http" module that returns "I have items" to every request
- Tell that server to start listening on port 8081
- Print a message when the server is ready to accept incoming messages

Please look at the sample on the [About Node.js®](#) page. It has all of the code that you need to get the above done. You'll want to change the port number from what it uses to 8081. You'll also want to change the text it sends to the browser from what it reads to "I have items".

See if you can figure that out on your own. You'll know you're done when you open up your browser to <http://localhost:8081/> (or refresh it because it's already there) and see the following.



I have items

Phase 3: Understand the code

Hopefully, your code looks similar to the following code.

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 8081;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('I have items');
});

server.listen(port, hostname, () => {
  console.log(`Server running at
http://${hostname}:${port}/`);
});
```

Just a reminder: the first three variable declarations and the last call to `listen` are boilerplate code. Every time you write a Node.js server, you would write the same code over and over. The real meat of the application is in the callback function that you pass to `createServer`.

```
(req, res) => {
  // The code here is what matters. This is the stuff
```

```
// that handles requests from the browser and sends
// content back to it.
}
```

The first parameter is the "request" object and is of type `http.IncomingMessage` ([link](#)). The second parameter is the "response" object and is of type `http.ServerResponse` ([link](#)).

In the code that you wrote, you set the status code of the response to 200 which means "OK", if you recall. Then, you set the content type of the content of the response to "text/plain" which means the browser should just show the content as plain text. Finally, you use the `end` method to send some content *and* end the response.

That last part is *very* important. If you don't end the response, the browser will just hang, waiting, expecting more from your server.

In this project, you will use more methods and properties of the `IncomingMessage` and `ServerResponse` objects to get your application working.

Phase 4: Showing images

In the **assets/images** directory of this project are four images that your server should be able to show. (And more, if you add more.)

A normal thing to do is to translate a URL to a path relative to your application's root directory. For example, say you typed the following URL into your browser.

```
http://localhost:8081/images/thread.jpeg
```

It would make sense to have the server send back the content of **assets/images/thread.jpeg** so the browser can show it. That's what you will do in this step, but for any of the images.

You'll need a way to read the contents of each file. The modern way to do this is to use the Promises-based portion of the file system library. At the top of your **index.js**, import the `readFile` function from the "promises" property of "fs" library.

```
const { readFile } = require('fs').promises;
```

You will use the `await` keyword with that function, so you need to change the signature of the callback method that you pass to the `createServer` method. Note the addition of the `async` keyword before the parameter list.

```
const server = http.createServer(async (req, res) => {
```

Again, you will map requests for images to the corresponding file in the **images** directory. It looks like this.

```
http://localhost:8081/images/filename.ext
```



```
./assets/images/filename.ext
```

If the image exists, you'll send the contents of the image to the browser. If it does not, then you will tell the browser that it does not exist by sending a 404 NOT FOUND status code.

Phase 4a: The happy path

To determine if the path is one that you want, at the top of your `async` callback, put an `if` statement that tests if the `req.url` property (which is a string) starts with `"/images/"`. Replace the comment below to do that.

```
const server = http.createServer((req, res) => {  
  if (/* req.url start with "/images" */) {  
  
  }  
  
  res.statusCode = 200;  
  res.setHeader('Content-Type', 'text/plain');  
  res.end('I have items');  
});
```

If the test passes, that means that `req.url` will contain a string like `"/images/thread.jpeg"`. That means that you will want to load the file from `./assets/images/thread.jpeg` which is the concatenation of the string `./assets` and the value of `req.url`. This code goes inside the `if` block.

```
const imagePath = './assets' + req.url;  
const imageFileContents = await readFile(imageFilePath);
```

Notice that you **did not** specify `'utf-8'` as part of the `readFile` call. That's because the content of an image file is **not** UTF-8 encoded text. Instead, it's binary. This way without the encoding just returns the raw data that is then sent to the browser.

After that, you need to should set the status code of the response to 200 to indicate everything is OK. Then, you need to set the content type which takes a little bit of figuring, so you can delay that for just a moment. Assume that the browser has requested an image in the JPEG format. Finally, you end the response by sending the data of the file that you read.

Add this code inside the `if` block after reading the file's contents.

```
res.statusCode = 200;  
res.setHeader('Content-Type', 'image/jpeg');  
res.end(imageFileContents);  
return;
```

The `return` at the end prevents any other code after it to run, that code at the bottom that sends back plain text.

You should now be able to see any of the following in your browser!

- <http://localhost:8081/images/thread.jpeg>
- <http://localhost:8081/images/horseshoe.jpeg>
- <http://localhost:8081/images/lint.jpeg>

Most likely, you can also see the following image, too.

- <http://localhost:8081/images/gravel.png>

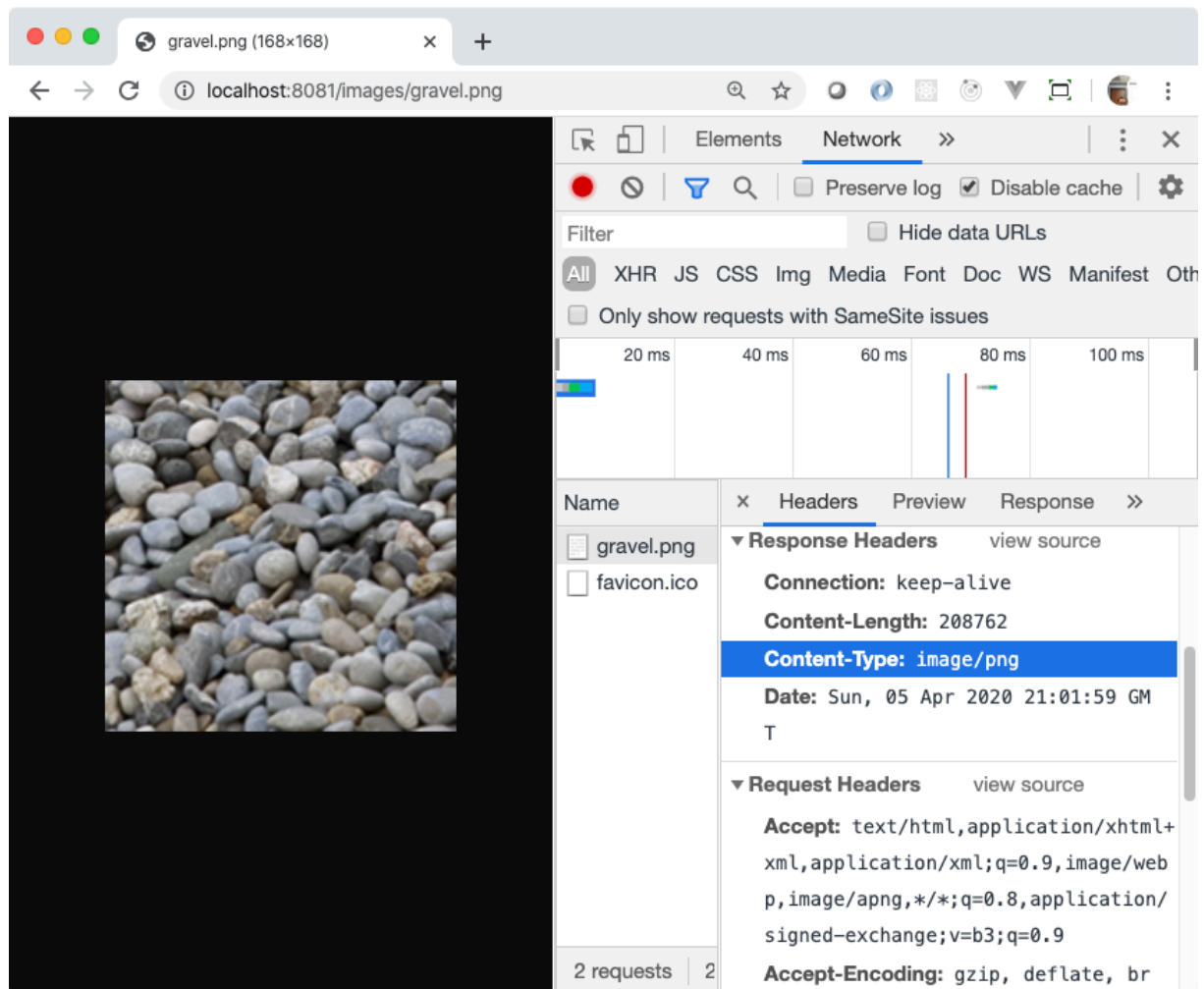
That's because browsers are really for giving. Even though you tell the browser that you are sending JPEG data with the content type "image/jpeg", the browser inspects the data and figures out it's an image in the PNG format. But, you should not rely on the forgiveness of the browser. Instead, you should determine the type of image format the file contains from the file extension, either ".jpeg" or ".png". Then, you send back "image/jpeg" or "image/png" based on the file extension.

You can use the built-in "path" library to determine the file extension. Then, you can use that information to send back the correct image format type in the `setHeader` method. At the top of the `index.js` file, import the "path" library.

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

Here's a link to the "path" library: <https://nodejs.org/api/path.html>. Find the method that will extract the file extension from a path. Then, use that in your code to send back the correct image type.

```
const fileExtension = /* Use the path library to get the file extension */;  
const imageType = 'image/' + fileExtension.substring(1);  
res.statusCode = 200;  
res.setHeader('Content-Type', imageType); // Use the image type
```

Make sure you still see "I have items" when you go to <http://localhost:8081>.

Phase 4b: No image found

Try accessing this URL: <http://localhost:8081/images/unknown.png>. You will see an error message in your console about an unhandled promise rejection not being able to open './assets/images/unknown.png'. Worse yet, the browser is just hanging. That's because this line of code:

```
const imageFileContents = await readFile(imageFilePath);
```

threw an error, it wasn't handled, and the `end` method never gets called on the response object. That means the browser just waits and waits and waits.

If you get a request for an image that does not exist, you can just catch this error and send back a 404 and no content. Replace that single line of code above with this block of code.

Wrap that line of code above in a `try/catch` block. In the `catch` block, set the status code of the response to 404. Then, just call the `end` method of the response with no parameters. The last statement of the `catch` block should be a `return;` statement to prevent other code from running after you handle this error. You'll have to fix the declaration of the `imageFileContents` variable so that it works.

Refresh the browser. You should now get a 404 page when you try to access an image that does not exist. You should see the images that do exist when you go to their corresponding URLs.

Make sure you still see "I have items" when you go to <http://localhost:8081>.

Phase 5: Showing a static HTML page

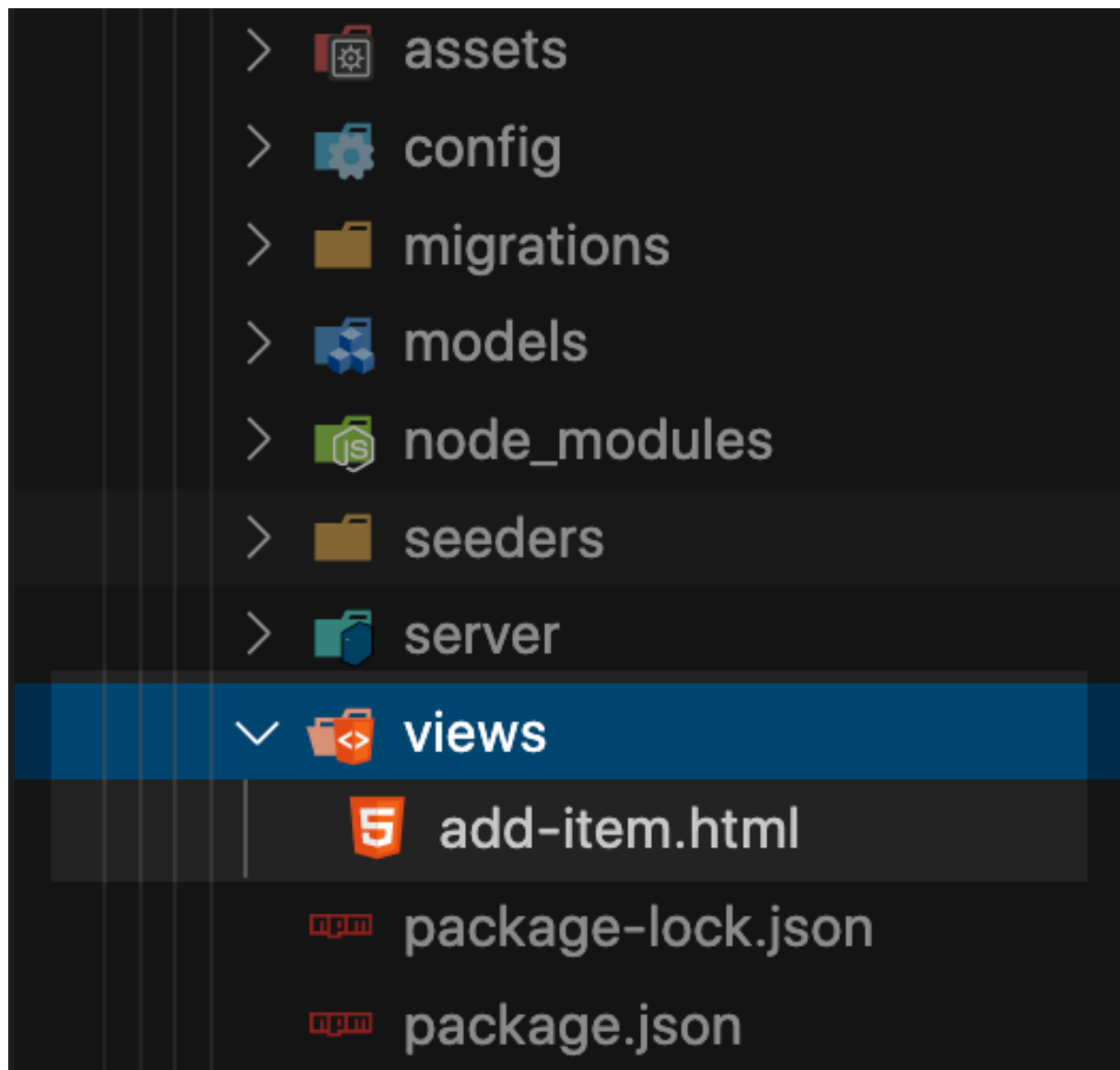
Here's some HTML that shows a form that you will use to add new items to the database. You will serve this statically. That means you won't change any of its contents. Instead, you'll just read the file from disk and send it to the browser.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Add an item</title>
</head>
<body>
  <header>
    <a href="/">Back to the main page</a>
  </header>
  <main>
    <form method="post" action="/items">
      <div>
        <label for="name">Name</label>
        <input type="text" name="name" id="name" required>
      </div>
      <div>
        <label for="description">Description</label>
        <textarea name="description" id="description"
required></textarea>
      </div>
      <div>
        <label for="amount">Starting amount</label>
        <input type="number" name="amount" id="amount"
required>
      </div>
    </form>
  </main>
</body>
</html>
```

```
<div>
  <button type="submit">Create a new item</button>
</div>
</form>
</main>
</body>
</html>
```

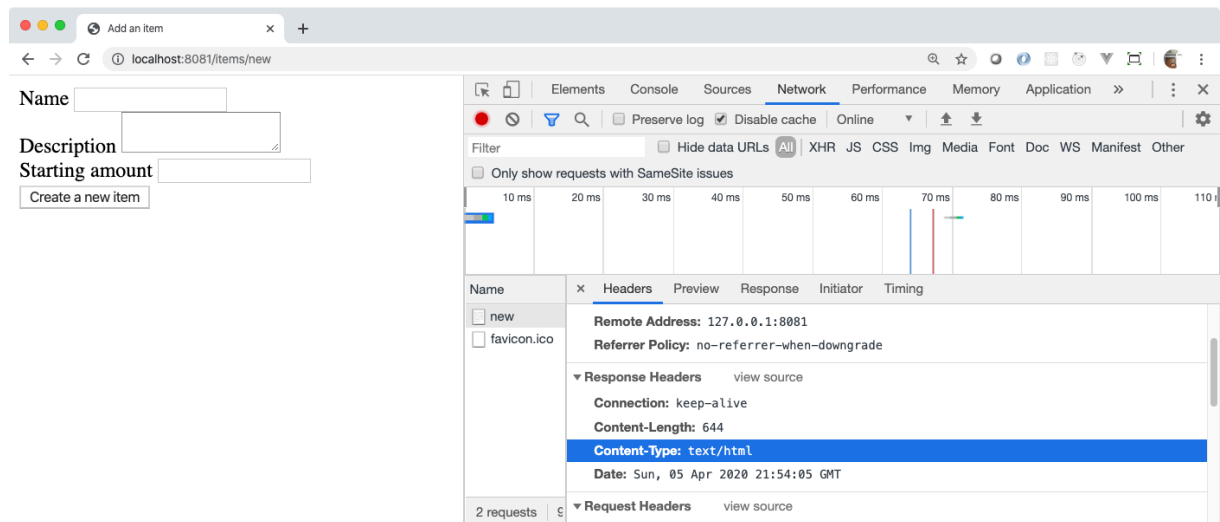
You'll learn a lot more about forms, this week. There are three things to note about this form.

- The "method" attribute of the `form` element is "post" which means the value of `req.method` in our request handler will be "POST". (It is always uppercase when read from the `req.method` property.)
 - The "action" attribute of the `form` element is `/items`. That will be the value of `req.url` that you will need to check when you want to handle the form submission.
 - The "name" attribute of the `input` and `textarea` (and all form elements) are the keys that we will use to get the values that a person supplies by typing into the form.
- Create a **views** directory in the root of your project. Save the HTML into a file there named **add-item.html**.



To serve this HTML, create a new `if` block that checks to see if the value of the `req.url` property is equal to `"/items/new"`. If it is, then do what you did with the images. The path to the HTML file should be `"/views/add-item.html"`. Read the file's contents. Set the status code to 200. Set the content type to `"text/html"`. Send the content of the file to the browser and end the response. Use a `return;` statement to make sure no other code runs.

When you get that working, you should be able to navigate to <http://localhost:8081/items/new> and see this.



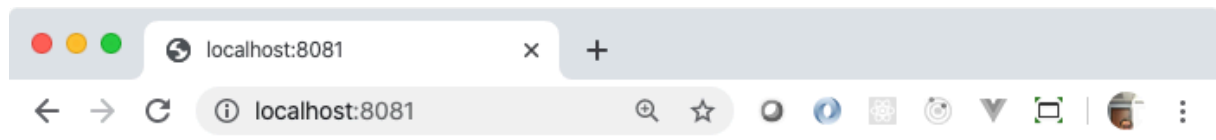
Phase 6: First step in dynamic content

Navigate your browser back to <http://localhost:8081> where you see "I have items". (If that's not working, figure out how it broke and fix it.) Now, you will query the database for the number of items in it and report it. Instead of seeing "I have items", it should report something like "I have 4 items".

This is primarily Sequelize code. At the top of **index.js**, import the Item model.

```
const { Item } = require('../models');
```

Down at the bottom of your callback after your `if` blocks and before the line that reads `res.statusCode = 200;`, use the `findAll` method of the Item model to get all of the items in the database. That should return an array of the objects. Use the `length` of that array to show the current number of items in the database by changing `res.end('I have items');` to include the number of items.

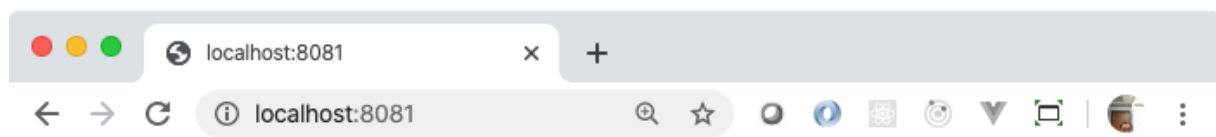


I have 4 items

It may surprise you to learn that this is really what most Web applications do. Read some data from a database. Use that data to generate some content. Send the content to the browser. That's the simple recipe.

Do something real quick before the next phase. Instead of serving plain text, here, change that content type to serve HTML. Then, in whatever string you're passing to the `res.end` method, add this HTML snippet at the beginning of it so you can easily get to the "add a new item" form.

```
<div><a href="/items/new">Add a new item</a></div>
```



[Add a new item](#)

I have 4 items

Test the link by clicking on it. It should take you to the form.

Phase 7: Handling the adding of an item

Now's the time to handle the adding of an item from that form! Click the link to get to the add the form or navigate to <http://localhost:8081/items/new> in your browser. If you fill out the form and click the button, it just takes you back to the main page and doesn't do anything. It's time to change that.

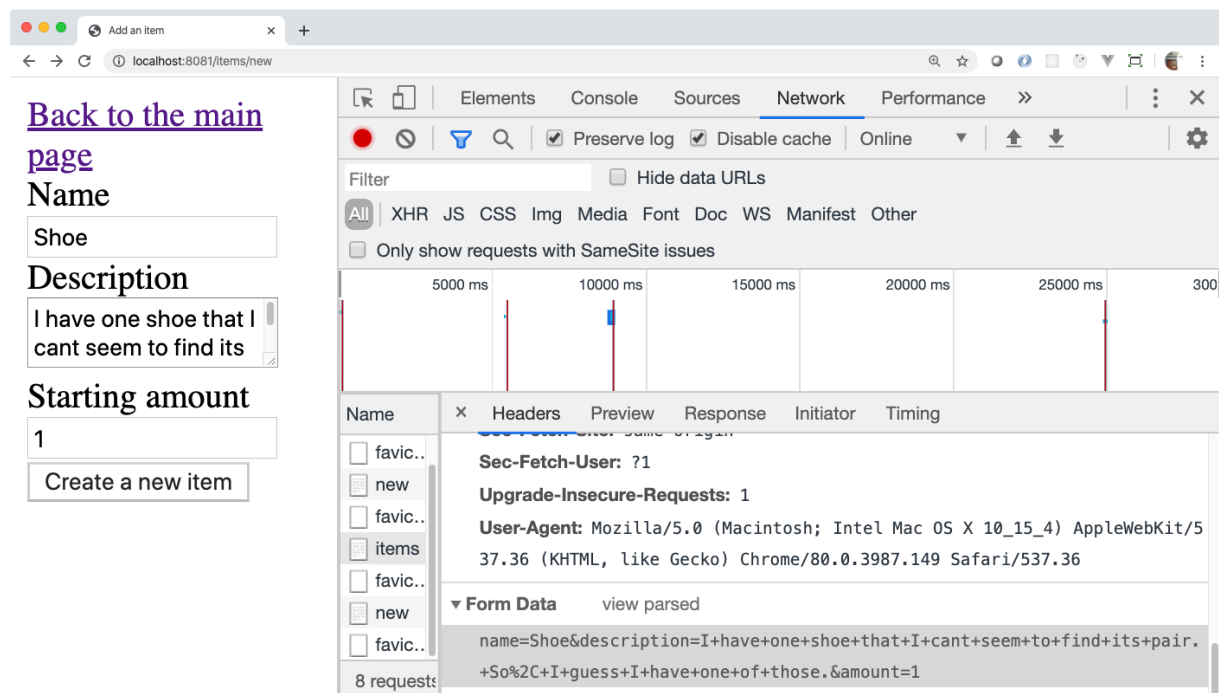
Add a new `if` block that checks that *both* of these conditions are true:

- The value of `req.url` is equal to `"/items"`
- The value of `req.method` is equal to `"POST"`

Inside that `if` block is where you will handle the data that someone sends to the server through the form. You'll use it to create a new `Item` and save it to the database. Then, you'll redirect back to the main page.

Phase 7a: Getting the submitted data

Open up your developer tools. On the Network tab, click the "Preserve log" checkbox above the timeline. Then, fill out the form and click the "Create a new item" button. That will make the request. Select the "items" entry in the list of network requests below the timeline. You should see a section entitled **Form Data**. Click the "view source" link.



What you see is likely like this, but with whatever values you put in the fields.

```
name=Shoe&description=I+have+one+shoe+that+I+cant+seem+to+find+its+p  
air.+So%2C+I+guess+I+have+one+of+those.&amount=1
```

That's the content that is sent with the HTTP request to your server. The full HTTP request looks something like

```
POST /items HTTP/1.1  
Host: localhost:8081  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 116  
... more headers ...  
  
name=Shoe&description=I+have+one+shoe+that+I+cant+seem+to+find+its+p  
air.+So%2C+I+guess+I+have+one+of+those.&amount=1
```

That's the "URL encoded" format that you read about in the Five Parts Of A URL reading. You'll parse that in the next step. What you have to do, now, is get it from the `IncomingMessage` object. That object is a readable stream, so you will read the bytes from the stream and turn them into a string to use in the next step.

When your callback is invoked by the server object, it has *only* read the headers portion of the HTTP request. The body of the HTTP request (if there is one) could still be traveling over the airwaves and wires from your computer to the server. This way, your Web application can look at the values in the headers and determine whether or not it wants to even respond. Maybe the content length is 400Gb. You don't want your server spending however long it takes to read all of that data, so you can just end it.

To do this easily, you will use a variety of the `for` loop that works with asynchronous iterable values as well as normal one. It is the `for await...of` loop. Like the `for of` loop, it loops over values rather than indexes. But, the value after the `of` can return Promises which the `for` loop will wait on for them to resolve before invoking the block of code. That's a lot of words. Here's what it looks like. Put this in your `if` block that handles "POST /items".

```
let body = '';  
for await (let chunk of req) {  
  body += chunk;  
}  
// body now contains all of the data  
// from the request
```

This works because `req` is an `IncomingMessage` message object which inherits from `ReadableStream` which implements the `asynchronous iterator` property.

Phase 7b: Parsing the submitted data

Now that you have all of the data in the `body` variable, it's time to split it up into the data that you want. From the form, it will look like this as a raw string:


```
name=value1&description=value2&amount=value3
```

Use string manipulation to break that into its separate pieces so that you can access each of key value pairs.

- Split the string on ampersands, first.
- Split each value from the previous step on equal signs.

To handle the encoded values on the right side of the equal sign, it is a two-step process:

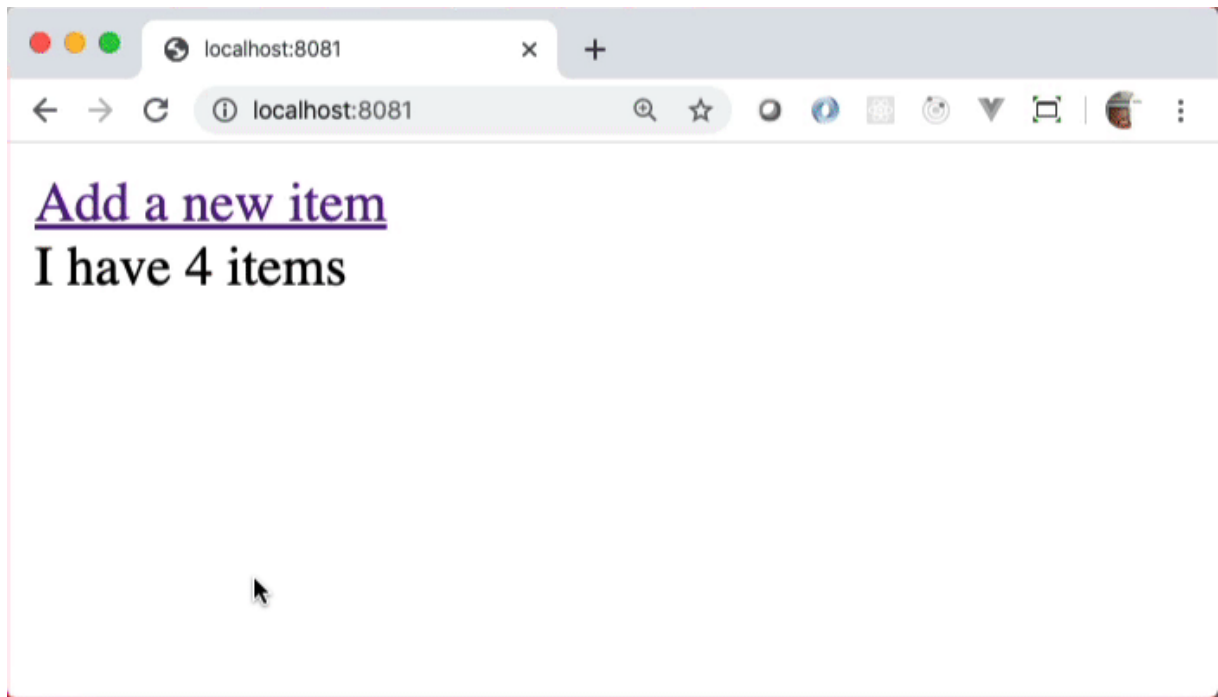
- Replace each of the "+" characters with a space. You have to use a global regular expression to do this with the `replace` method because JavaScript will only replace the first occurrence in the string without a global regular expression. If the value is in a variable named `s`, you would call `s.replace(/\+/g, ' ')` to replace all of the "+" characters in a string with spaces.
- After replacing the plusses, take the value and pass it to the `decodeURIComponent` function which will go about translating the percent-sign encoded values into single characters.

Phase 7c: Create an Item

You should have the data broken into pieces that you can now access. Use your Item model to build and save (or create) a new item.

Phase 7d: Redirect the browser

Redirecting the browser to go to another URL is a two-step process, too. You send back status code 302. You also set the header "Location" to the URL that you want it to navigate to. For this project, set the "Location" to `/`. Then end the response.



Make sure that you use a `return` statement or something to prevent the default code at the bottom of your request handler from running.

Phase 8: Generate dynamic content

At the bottom of your handler, you've already queried the items in your `Item` objects from the database. Now, it is time to show the items rather than just displaying how many are in the database.

You can use the `write` method of the `ServerResponse` object in the `res` to write your HTML to the browser as you're generating it. Your code may look something like this.

```
const items = await Item.findAll();
res.statusCode = 200;
res.setHeader('Content-Type', 'text/html');
res.end(`
  <div><a href="/items/new">Add a new item</a></div>
  I have ${items.length} items
`);
```

Take a look at this code which just expands on the previous block. It writes the proper beginning of an HTML document, then writes the dynamic content, then ends it with the proper end of an HTML document.

```
const items = await Item.findAll();
```

```

res.statusCode = 200;
res.setHeader('Content-Type', 'text/html');
res.write(`
  <!DOCTYPE html>
  <html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Inventory</title>
  </head>
  <body>
    <header>
      <div><a href="/items/new">Add a new item</a></div>
    </header>
    <main>`);

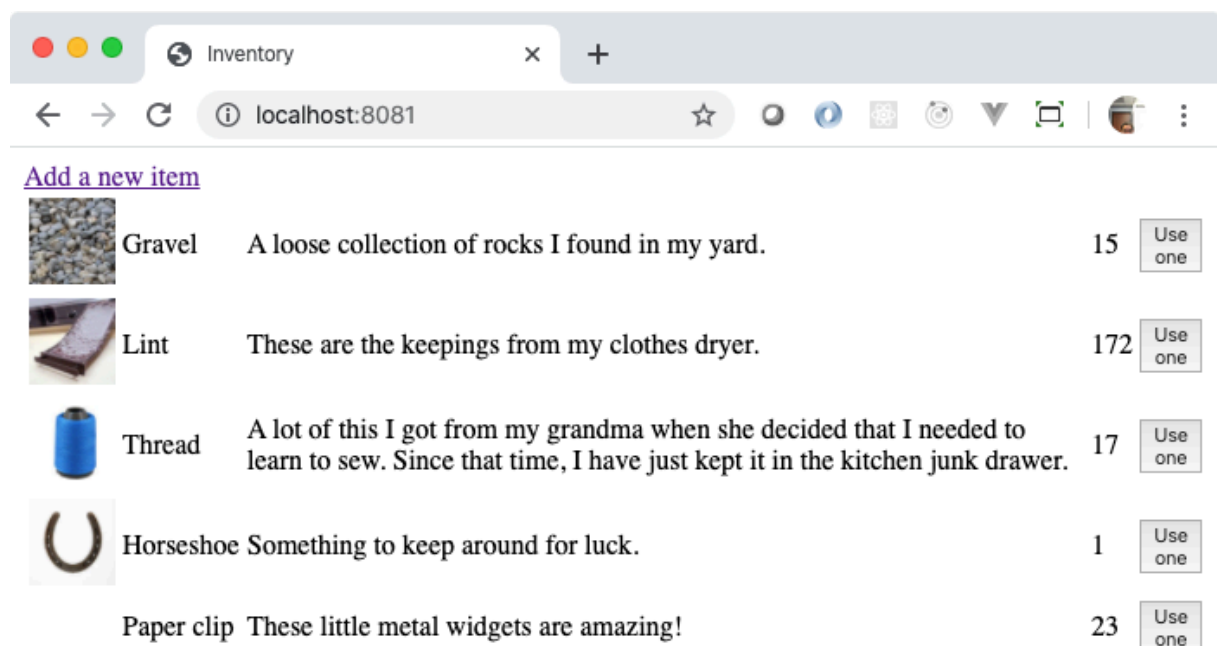
res.write(`I have ${items.length} items`);

res.end(`
  </main>
</body>
</html>`);





```

In the place where there's only the one line of dynamic content, change it to have something else, something that shows the name of the item, the amount of them you have, and the associated image, if `imageName` is not `null` or `undefined`.

The following screenshot shows where an open `table` tag has been added to the end of the string for the first write, a close `table` tag has been added to the beginning of the string of the `res.end` call, and looping is used to create a new table row (`tr`) with table data (`td`) for each of the properties of the `Item`.



[Add a new item](#)

	Gravel	A loose collection of rocks I found in my yard.	15	<button>Use one</button>
	Lint	These are the keepings from my clothes dryer.	172	<button>Use one</button>
	Thread	A lot of this I got from my grandma when she decided that I needed to learn to sew. Since that time, I have just kept it in the kitchen junk drawer.	17	<button>Use one</button>
	Horseshoe	Something to keep around for luck.	1	<button>Use one</button>
	Paper clip	These little metal widgets are amazing!	23	<button>Use one</button>

It looks, in part, like this.

```
for (let item of items) {
  res.write(`
    <tr></td>
  `);

  // Only write an IMG tag if there is a value
  // in imageName

  res.write(`
    </td>

    <!-- Write more TDs here with the details of the item -->

    <td>`);
  if (item.amount > 0) {
    res.write(`
      <form method="post" action="/items/${item.id}/used">
        <button type="submit">Use one</button>
      </form>
    `);
  }
  res.write(`</td>
</tr>`);
}
```

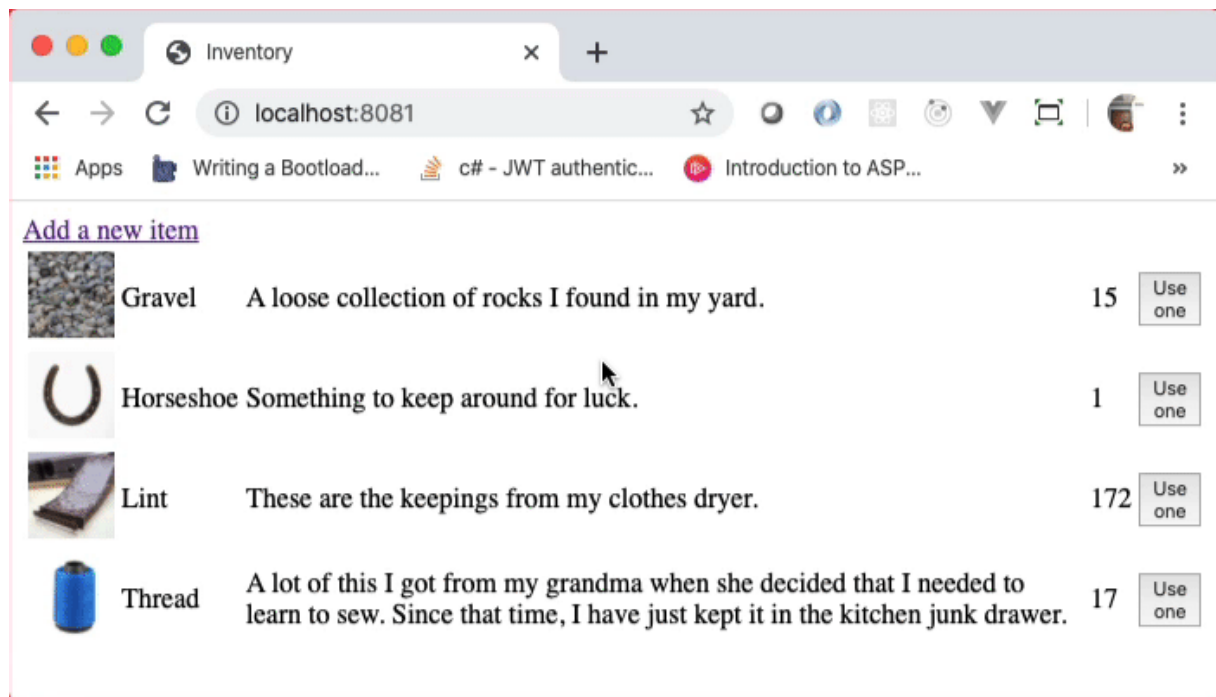
As seen above, for each item, you should also create a form that has the following content for items with an amount greater than 0.

```
<form method="post" action="/items/«item id»/used">
  <button type="submit">Use one</button>
</form>
```

That's the last handler that you'll write to complete the project!

Phase 9

When there is a POST request to the path `"/items/«item id»/used"`, you want to reduce the amount by 1 of the item specified by the `«item id»` in the path. Write another `if` block that handles that HTTP request. Parse the id from the path. Use that id to get the Item from the database. Reduce the amount by 1. Save the Item back to the database. Redirect back to `"/"`.



Complete!

That was quite a ride! You created a full-stack Web application! You pulled data from a database to generate HTML. You sent the HTML to the browser. You handled requests, both GET and POST, from the browser to interact with and modify the data in the database. This is literally what Web developers do every single day.

Except with better tools. Tools like you learn about tomorrow.

Did you find this lesson helpful?

No
Yes