

[Open in app](#)

John Martinez

[Follow](#)

42 Followers

[About](#)

Full CRUD with JavaScript

[John Martinez](#) Jan 16, 2019 · 17 min read


JavaScript is sometimes difficult for me to grasp. I can't beat myself up too much about it, after all I've only been practicing JS for a little more than two weeks at this point. But there are still some concepts that I struggle with, and some issues that I'm having with the language. What's the best way for me to overcome these problems?

It's a question my fencing students ask me all the time. How do I beat this stance, or how do I fight against a left-handed fencer? It got me thinking about how other instructors approach the topic, the advice they give and how receptive some students are to that advice. More often than not, simple concepts are met with complicated solutions. Newer students are inundated with intermediate(or higher)-level jargon, far-off theories, and complex minutiae when they're still struggling to hold their sword properly.

I like having things explained to me in very basic language. I learned that from my time at the Byakkokan Dojo, an incredible Battodo school in Chelsea. The instructors there have a way of teaching that greatly appeals to me, and I think a part of their success is how they manage to take complex movements and break them down into simple, fundamental movements. Swinging a sword effectively is harder than people think. There's a lot of tiny motions and adjustments that your body makes without you even knowing it, which is why practice is so essential. But, in order to get good practice in, you need to be doing each step effectively, and if you can't see what the individual pieces are, it can often be frustrating or detrimental to your learning experience.

That being said, when I searched online for help or documentation regarding my issues with JavaScript, I was more often than not hitting a dead-end. So I decided to write this article, detailing the creation of a simple, full CRUD app with JavaScript from start to finish. Hopefully with language and descriptions of certain concepts that people who are learning JS might find appealing.

The Book Lister App



A quick and simple mock-up of the app.

So the app we'll be writing is a very basic book listing app. It will allow a user to add books to a collection, including a title, author, a cover image, and a description. They can then scroll through the books they've added. They will also be able to edit books, and delete books.

This will not be a pretty app, since instead I want to focus entirely on getting the CRUD (that is, the CREATE, READ, UPDATE and DELETE) actions in place. I will be using the Atom Text Editor, Google Chrome, and iTerm to make this app.

Step 1: Making the files

We're going to start by creating the files we need to get this project up and running. Using our terminal, and making sure we're in the proper directory (I have a `dev` directory that I use for all my coding), we're going to type

```
mkdir book-project
```

to make a `book-project` directory. We will then `cd` into that directory and use the `touch` command to make the only three files we'll need for this app:

1. `touch index.html` — which we'll use to hold the HTML of our app,
2. `touch index.js` — which we'll use to hold the JavaScript of our app,
3. `touch books.json` — which we'll use to hold all of our book JSON data.

For those wondering, JSON is basically a way of formatting written data in a way that's human readable. It's also incredibly nifty for storing and sorting through data! Speaking of which, we'll need to download and run our JSON server. From your terminal, type `npm install -g json-server` and once that's done installing, type `json-server books.json` to run your JSON server.



```
1. johnmartinez@Johns-MacBook-Air (node)
// ♥ atom .
[21:17:36] book-project
// ♥ npm install -g json-server
/Users/johnmartinez/.nvm/versions/node/v10.13.0/bin/json-server -> /
rtinez/.nvm/versions/node/v10.13.0/lib/node_modules/json-server/lib/
+ json-server@0.14.2
updated 1 package in 12.183s
[21:29:59] book-project
// ♥ run json-server books.json
-bash: run: command not found
[21:30:14] book-project
// ♥ json-server books.json

\{^_^}/ hi!

Loading books.json
Done

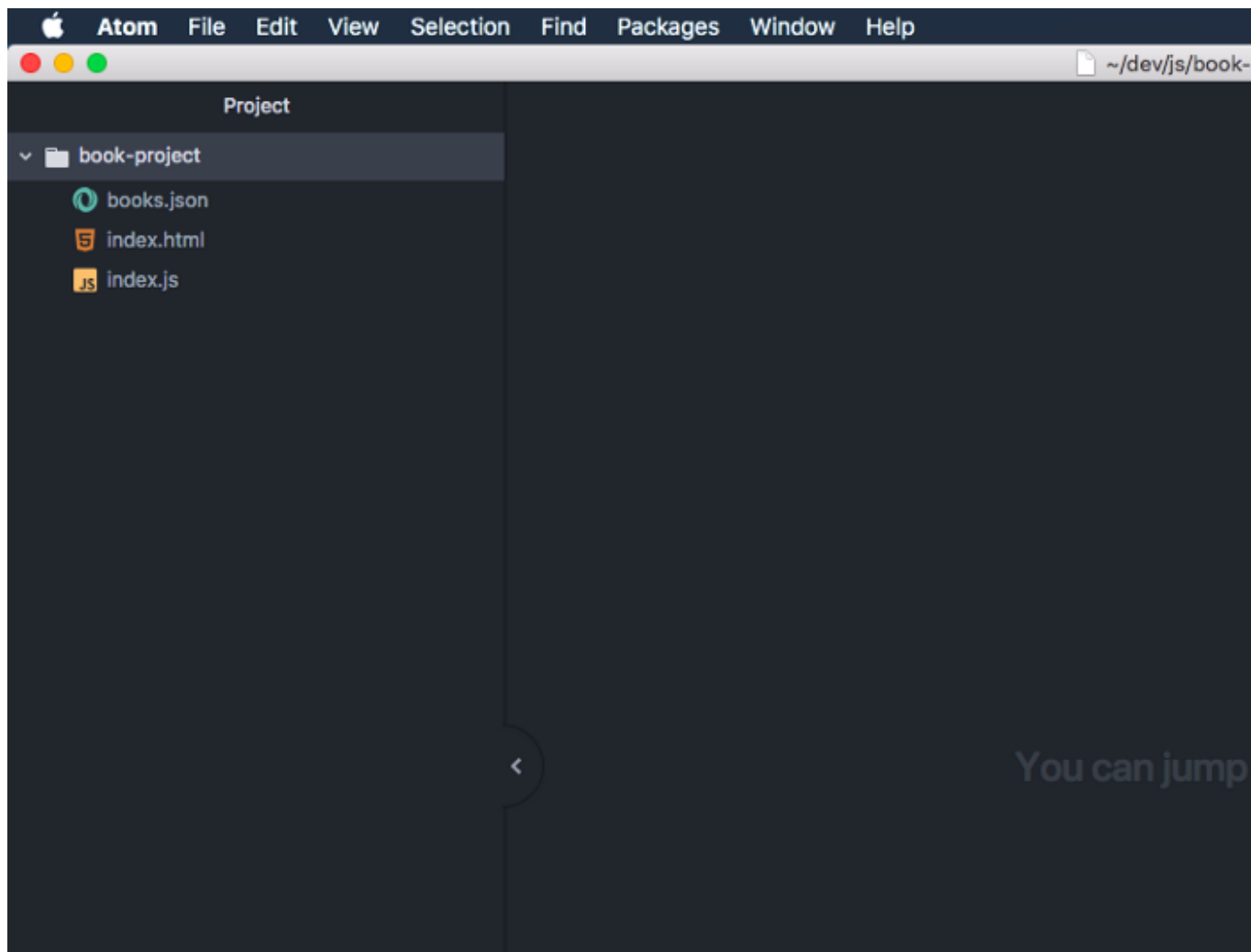
Resources
```

```
Home
http://localhost:3000

Type s + enter at any time to create a snapshot of the database
```

Our JSON server is up and running

After all of this is done, we can open our directory in our text editor of choice. If you're using Atom like I am, the command is `atom .` in the terminal.



It should look something like this

and we can type <http://localhost:3000> into our browser (or just click it!).



JSON Server

Congrats!

You're successfully running JSON Server

💎*.۹('۞`*)و💎*۔

Resources

No resources found

To access and modify resources, you can use any HTTP method

GET POST PUT PATCH DELETE OPTIONS

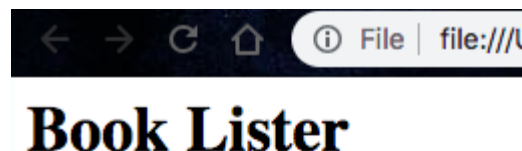
Documentation

View [README](#)

We're up and running!

Step 2: Basic HTML and a Test Entry for Our Data

First, let's make sure our HTML is working. If we type `open index.html` into our terminal, we should open a blank page in our browser (if your terminal is pre-occupied with the JSON server, just press `ctrl + c` to close it. You can always run it again afterwards!). In your `index.html` file, type `<h1>Book Lister</h1>` and save. Then hit refresh on the `index.html` tab in your browser and you should see this



Now we know the index.html file is working!

Next we're going to add some additional HTML as a foundation for our app. This will contain things that we'll need in the future, like a place for our books to populate, a form with inputs we can use to create new books, and a script that will allow the `index.html` file to read our `index.js` file. Copy and paste the following code into your `index.html` file:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Book Lister</title>
  </head>
  <body>
    <h1>Books</h1>
    <div id='create-book'>
      <form id="book-form">
        <input id="title" placeholder="title...">
        <input id="author" placeholder="author...">
        <input id="coverImage" placeholder="cover-image...">
        <input id="description" placeholder="description...">
        <input type="submit" value="Add Book">
      </div>
    <div id='book-container'></div>

    <script src='index.js'></script>
  </body>
</html>
```

If we save and refresh the `index.html` page in our browser, we should see this:



Books

title...	author...	cover-image...	description...	Add Book
----------	-----------	----------------	----------------	----------

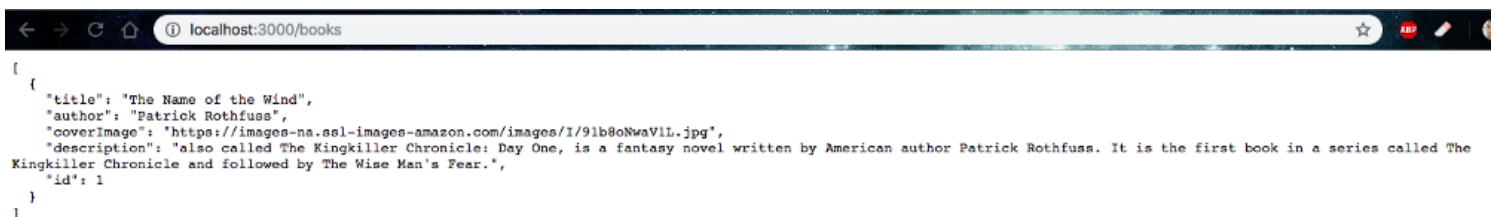
Looks good already!

But we don't see any books yet! Let's fix that. To get a better feel for the JSON format, we'll make our first entry in the `books.json` file. Feel free to copy and paste the following code into the `books.json` file:

```
{
  "books": [
    {
      "title": "The Name of the Wind",
      "author": "Patrick Rothfuss",
      "coverImage": "https://images-na.ssl-images-
amazon.com/images/I/91b8oNwaV1L.jpg",
      "description": "also called The Kingkiller Chronicle: Day One,
is a fantasy novel written by American author Patrick Rothfuss. It
is the first book in a series called The Kingkiller Chronicle and
followed by The Wise Man's Fear.",
      "id": 1
    }
  ]
}
```

What this is basically doing is creating a list of book objects in `books.json`. Each book has a key of title, author, coverImage, description, and id. Later on we can use those keys to help show existing data in the JSON file, and create new data as well.

If you reset your JSON server and navigate to <http://localhost:3000/books>, you should see that the data in your `books.json` file has successfully been loaded to the page!



Step 3: Loading the DOM and the 'R' of CRUD

For those interested, [these are some great docs on what the DOM is](#), but for our purposes the DOM is basically the medium on which we are writing our app.

In order to get started, we need to make sure that the DOM has fully loaded before anything else occurs. To do this, we are going to add an event listener to the DOM. In `index.js` you can write the following:

```
document.addEventListener('DOMContentLoaded', function() {  
  // everything else we type will go inside this!!  
  
})
```

What this is doing is taking the entire `document`, which is the whole HTML page we are working on, and listens for the event of `'DOMContentLoaded'` which is basically when the entire DOM has finished loading in. Once that event is heard, it then runs whatever code we put inside the curly brackets `{}` which I've commented in for you.

Now we can start writing the first part of our CRUD functionality. I'm skipping 'C' for now and jumping right to "R" because we need to be able to see our data before we can add stuff to our database!

First let's create a constant for where our books are going to show up. In `index.html` you'll notice `<div id="book-container"></div>`. This is where we'll make our books appear! In `index.js` inside the event listener we just made, add the following code:

```
const bookContainer = document.querySelector('#book-container')
```

What this container is doing is creating a nice landing-pad for the HTML we'll create using JavaScript code. We'll be adding the code to `bookContainer` and when the page is refreshed our newly-added code should appear on the screen!

While we're at it, let's make another constant for the URL of our books database:

```
const bookURL = `http://localhost:3000/books`
```

This way, later on when we're using the URL to fetch data, we don't have to type in the whole thing!

Now we need to fetch the data from `bookURL` and add that data mixed with HTML to our `bookContainer`. It may sound complicated but it's really quite simple. Write the following code into `index.js`:

```
fetch(`${bookURL}`)
  .then( response => response.json() )
  .then( bookData => bookData.forEach(function(book) {
    bookContainer.innerHTML += `
      <div id=${book.id}>
        <h2>${book.title}</h2>
        <h4>Author: ${book.author}</h4>
        
        <p>${book.description}</p>
        <button data-id="${book.id}" id="edit-${book.id}" data-
action="edit">Edit</button>
        <button data-id="${book.id}" id="delete-${book.id}" data-
action="delete">Delete</button>
      </div>`
  }))) // end of book fetch
```

There's a lot to unpack here, so let's start with the top. We're calling `fetch` to pull data from `bookURL`. Then we're taking the `response` from `bookURL` (all of the data from the database) and running `.json()` on it. This will take the data and make it human-readable. Then we're calling that now human-readable response and calling it `bookData`. We then run a `forEach` on `bookData`, which will run a function on each individual book in `bookData`.

For each book in `bookContainer` we are adding new HTML to `bookContainer.innerHTML`, which is the actual HTML code of the `bookContainer` we made. This additional HTML code will have that book's title, author, `coverImage`, and description, as well as an edit and delete button that we can use later.

index.js should look something like this:

```
1 document.addEventListener('DOMContentLoaded', function() {
2
3   const bookContainer = document.querySelector('#book-container')
4   const bookURL = `http://localhost:3000/books`
5
6   fetch(`${bookURL}`)
7     .then( response => response.json() )
8     .then( bookData => bookData.forEach(function(book) {
9       bookContainer.innerHTML += `
10       <div id='singleBook'>
11         <h2>${book.title}</h2>
12         <h4>Author: ${book.author}</h4>
13         
14         <p>${book.description}</p>
15         <button data-id="${book.id}" data-action="edit">Edit</button>
16         <button data-id="${book.id}" data-action="delete">Delete</button>
17       </div>`
18     })) // end of book fetch
19
20 })
21
```

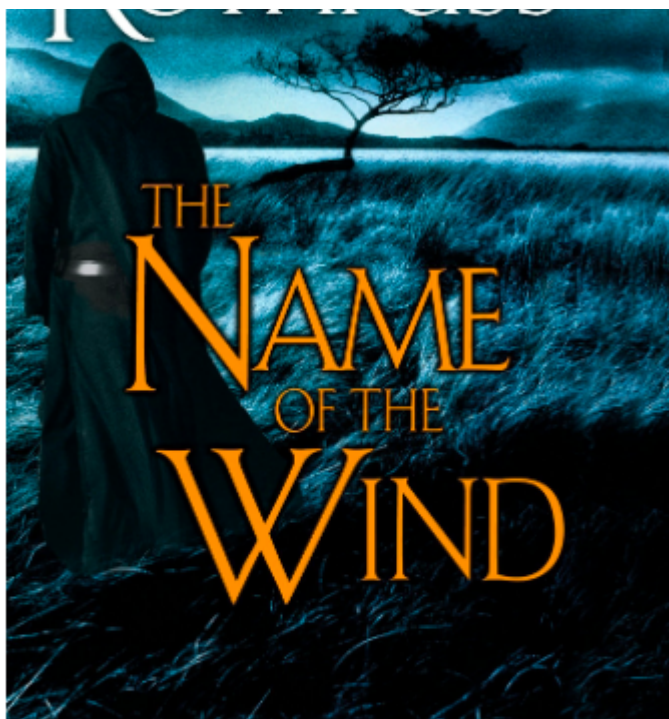
and if you refresh your Book Lister app, it should look something like this:

Books

The Name of the Wind

Author: Patrick Rothfuss





also called *The Kingkiller Chronicle: Day One*, is a fantasy novel written by American author Patrick Rothfuss.

[Edit](#)[Delete](#)

Now we can see our books!

Step 4: The 'C' of CRUD

Now that we can `READ` the data on our app, let's add the ability to `CREATE` new books! We have four input fields and an "Add Book" button at the top of our page, just below **Books**, but they don't do anything. You can fill them in and click the button, but nothing happens. This is all intentional, since they don't have any functionality yet!

First, just like for the `bookContainer`, let's create a constant for the book-form, which is what we'll be manipulating in order to add our input to the database:

```
const bookForm = document.querySelector('#book-form')
```

If you're wondering what the `#` is in `('book-form')` it means we're selecting the HTML object with an **ID** of "book-form". If you look at our HTML, you'll notice the code `<form`

`id="book-form">` . This is the HTML object that our form is located in. If, instead of ID, it was `<form class="book-form">` , we would use a `.` instead of `#` — i.e. `('.book-form')` .

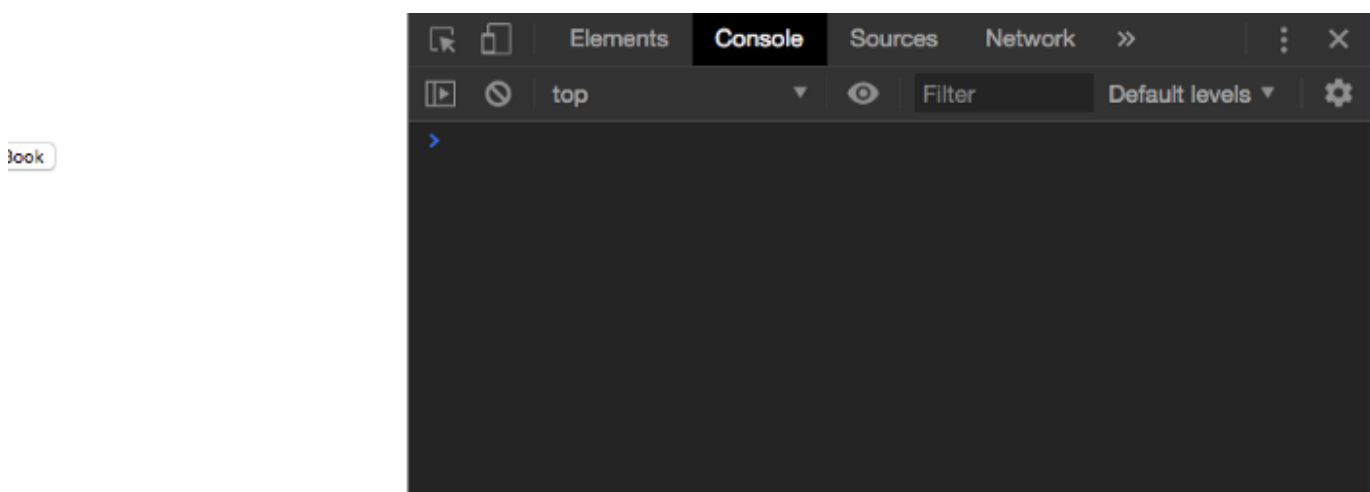
Now that we have `bookForm` we can add an `EventListener` to it:

```
bookForm.addEventListener('submit', (e) => {  
  e.preventDefault()  
  // additional functionality goes down here!!  
})
```

This is listening for an event called `'submit'` on `bookForm` . Since we have an input with the type `"submit"` (`<input type="submit" value="Add Book">`) if that input is submitted then it will trigger the functions in this event listener.

`e.preventDefault()` prevents this button from automatically refreshing the page, so you see the data uploaded to the database appear automatically instead of on a page refresh. Also, `e` is the name we're giving to this specific event in this method. The event has attributes inside it that will be useful later on for determining not only where we clicked, but what we clicked! For more information on `EventListeners` and `event` itself, check out the [MDN docs on the subject](#).

We haven't done much `console.logging`, so let's use this as an opportunity to try it out. Under `e.preventDefault()` type `console.log(e.target)` , then hit refresh on your Book Lister page. Press `cmd+option+j` to open up the Chrome developer tools. It should look something like this:

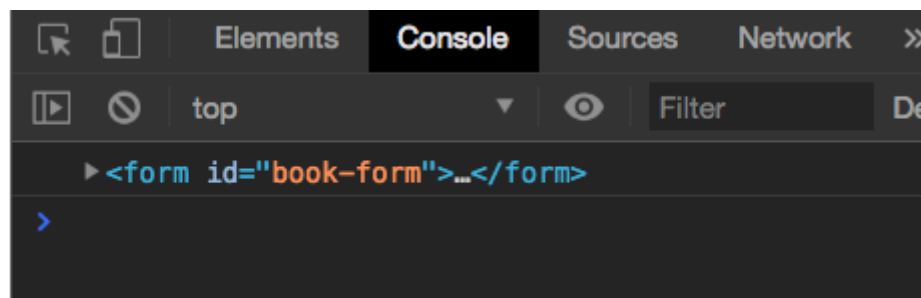


thor Patrick Rothfuss. It is the first

Console

Mind you, I have dark-theme activated.

Then, click the “Add Book” button. You should see the following:



Hitting the triangle on the left will show more info

This is what `e.target` is returning to us, and we are using `console.log` to print that information to the console in our developer tools! Try messing around with `console.log`. You can try `console.log(e)` to see what `e` is returning by itself, or you can type `console.log('whatever text you like')` to return any text you want to the console. This is a great tool that will prove to be very useful in the future.

Next let's create some additional constants, one for the input of each attribute we're adding to the new book:

```
const titleInput = bookForm.querySelector('#title').value
const authorInput = bookForm.querySelector('#author').value
const coverImageInput = bookForm.querySelector('#coverImage').value
const descInput = bookForm.querySelector('#description').value
```

These constants are selecting their respective keys from `bookForm`, and returning their value with `.value`. Try console logging these constants, and put some text in the input fields to see what you get!

Now that we have the inputs, we can add a new `fetch` request that will `POST` to the database with our new data:

```
fetch(`${bookURL}`, {
  method: 'POST',
  body: JSON.stringify({
    title: titleInput,
    author: authorInput,
    coverImage: coverImageInput,
    description: descInput
  }),
  headers: {
    'Content-Type': 'application/json'
  }
})
```

Similarly to the previous `fetch`, this is opening `bookURL` and using `method: 'POST'` to post the data to the database. Then, `body: JSON.stringify()` takes the data we feed into it and turns it into stringified data, which is difficult for humans to read, but easier for JSON to read. It takes in an object with the same keys as our original database, but with our newly inputted data.

```
{
  title: titleInput,
  author: authorInput,
  coverImage: coverImageInput,
  description: descInput
}
```

Consider, but feel free to ignore the following code:

```
headers: {  
  'Content-Type': 'application/json'  
}
```

It's necessary, but currently just something you need to make the `fetch` work. They're something I'm actively learning more about, and if you're interested in reading about them you can find a link to their documentation [here](#).

If you go to your Book Lister page and hit refresh, you can try typing in a title, author name, a URL for your book cover, and a description. When you're ready, hit "Add Book" and...

...nothing happens. The new data is in your database, but the HTML hasn't updated along with it! Hit refresh and you should see your new book added beneath "The Name of the Wind".

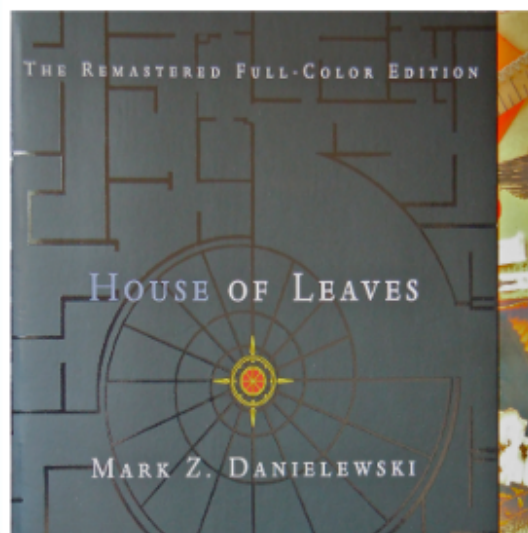


also called The Kingkiller Chronicle: Day One, is a fantasy novel written by American author Patrick Rothfuss. It is the first book in a

Edit Delete

House of Leaves

Author: Mark Z. Danielewski



Having to refresh to see what you've posted is a bit annoying and not exactly user-friendly. Let's make it so when you hit "Add Book" your new data is automatically rendered on the page! Under `fetch` write the following code:

```
.then( response => response.json())
.then( book => {
  bookContainer.innerHTML += `
    <div id=${book.id}>
      <h2>${book.title}</h2>
      <h4>Author: ${book.author}</h4>
      
      <p>${book.description}</p>
      <button data-id="${book.id}" id="edit-${book.id}" data-
action="edit">Edit</button>
      <button data-id="${book.id}" id="delete-${book.id}" data-
action="delete">Delete</button>
    </div>`
})
```

Similarly to the READ action, this will add new HTML to the `bookContainer` with your newly created book's information! By changing the form *after* updating the database, we are performing what's known as "*pessimistic rendering*". It's useful because if you don't update the data in the database (say, because you've lost connection to the internet), your data will not update on the page.

The opposite of this is "optimistic rendering", which renders the page with your updated information *before* you update the database. Both of these concepts have their pros and cons, but for this app we'll focus on pessimistic rendering. For more information on pessimistic rendering, I've linked the documentation [here](#).

Up to this point, our code should look like this:

```
1 document.addEventListener('DOMContentLoaded', function() {
2
3   const bookContainer = document.querySelector('#book-container')
4   const bookURL = 'http://localhost:3000/books'
5   const bookForm = document.querySelector('#book-form')
6
7   fetch(`${bookURL}`)
8     .then( response => response.json() )
9     .then( bookData => bookData.forEach(function(book) {
10       bookContainer.innerHTML += `
11         <div id=${book.id}>
12           <h2>${book.title}</h2>
13           <h4>Author: ${book.author}</h4>
14           
15           <p>${book.description}</p>
```



```

6         <button data-id="${book.id}" data-action="edit">Edit</button>
7         <button data-id="${book.id}" data-action="delete">Delete</button>
8     </div>`
9     ))) // end of book fetch
10
11
12     bookForm.addEventListener('submit', (e) => {
13         event.preventDefault();
14
15         console.log(e.target)
16
17         const titleInput = bookForm.querySelector('#title').value
18         const authorInput = bookForm.querySelector('#author').value
19         const coverImageInput = bookForm.querySelector('#coverImage').value
20         const descInput = bookForm.querySelector('#description').value
21
22
23         fetch(`${bookURL}`, {
24             method: 'POST',
25             body: JSON.stringify({
26                 title: titleInput,
27                 author: authorInput,
28                 coverImage: coverImageInput,
29                 description: descInput
30             }),
31             headers: {
32                 'Content-Type': 'application/json'
33             }
34         }).then( response => response.json())
35         .then( book => {
36             bookContainer.innerHTML += `
37             <div id=${book.id}>
38                 <h2>${book.title}</h2>
39                 <h4>Author: ${book.author}</h4>
40                 
41                 <p>${book.description}</p>
42                 <button data-id="${book.id}" data-action="edit">Edit</button>
43                 <button data-id="${book.id}" data-action="delete">Delete</button>
44             </div>`
45             ))
46         }) // end of eventListener for adding a book
47     })

```

Step 5: The “U” of CRUD

Now that we can CREATE and READ, we should be able to UPDATE our information. Let's say we accidentally misspelled an author's name, or we found a better looking cover for our entry. We should be able to edit our books with our new information!

To get this started, let's add an eventListener to bookContainer. We'll end up using the same eventListener for DELETE later, so keep that in mind:

```

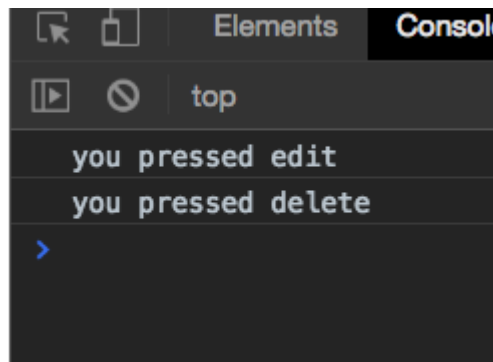
bookContainer.addEventListener('click', (e) => {
    if (e.target.dataset.action === 'edit') {

```

```
        console.log('you pressed edit')
    } else if (e.target.dataset.action === 'delete') {
        console.log('you pressed delete')
    }
}) // end of eventListener for editing and deleting a book
```

What this is saying is that we are listening for an event of a “click” on `bookContainer`. We then check what was actually clicked on. If what we clicked on (`e.target.dataset.action`) is equal to “edit”, then we’ll console log “you pressed edit”. If it’s equal to “delete”, we’ll console log “you pressed delete”.

After adding this code, refresh Book Lister, and click on the edit and delete buttons. You should see these console logs appear in the console of your developer tools.



Now that we can isolate the button we’re pressing, we’ll start adding some functionality. When a user presses the “edit” button, a menu similar to the one we use to create new books should appear under the book they’re trying to edit, with that book’s information already inside it. Then, the user just has to make their changes, hit the newly created “Edit Book” button, and the new changes should appear instantly, and the edit-form should disappear.

First we gotta make sure that the edit button of the book we’re clicking on is giving us the information we want, namely the information of the book we want to edit! Let’s start by creating a new `let` at the top of our page, just below `bookForm`:

```
let allBooks = []
```

We will then take this empty array and, in our first `fetch`, we'll add the following bolded code:

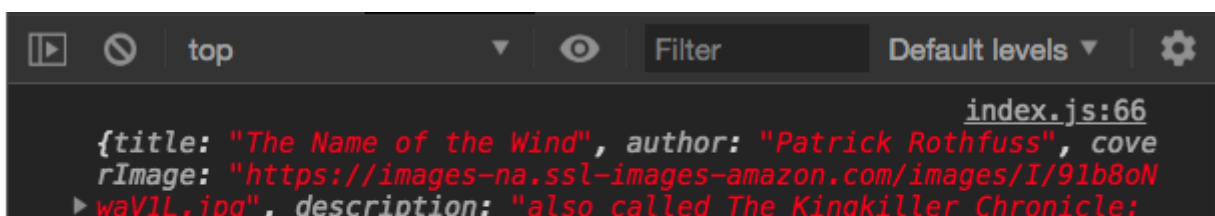
```
fetch(`${bookURL}`)
  .then( response => response.json() )
  .then( bookData => bookData.forEach(function(book) {
    allBooks = bookData
    bookContainer.innerHTML += `
      <div id=book-${book.id}>
    `
  }))) // end of book fetch
```

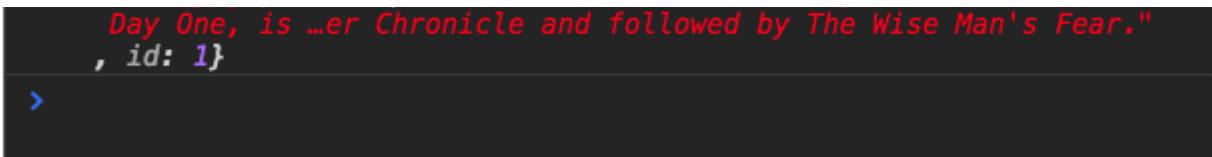
This way whenever we add a new book to our database, `allBooks` will automatically update with that new book. It's fairly inefficient (after all, this will change `allBooks` once for every time our books are individually fetched from the database), but it gets the job done for now. We can always refactor later on!

Now that we have all of our books stored in `allBooks`, we can use it in our new `eventListener`:

```
bookContainer.addEventListener('click', (e) => {
  if (e.target.dataset.action === 'edit') {
    const bookData = allBooks.find((book) => {
      return book.id == e.target.dataset.id
    })
    console.log(bookData)
  }) // end of eventListener for editing and deleting a book
```

This will find the book we're looking for in `allBooks`, whose ID is equal to our current target's ID. It will then return that book and hold it in the `bookData` constant. With `console.log(bookData)`, if you refresh Book Lister and click an edit button, you should see that book's information appear in the console:





We're almost there!!

Now that we have the book we're looking for, we can change the inner HTML of the targeted book to show the edit form:

```
bookContainer.addEventListener('click', (e) => {
  if (e.target.dataset.action === 'edit') {

    const editButton =
    document.querySelector(`#edit-${e.target.dataset.id}`)
    editButton.disabled = true

    const bookData = allBooks.find((book) => {
      return book.id === e.target.dataset.id
    })
    e.target.parentElement.innerHTML += `
    <div id='edit-book'>
      <form id="book-form">
        <input required id="edit-title"
placeholder="${bookData.title}">
        <input required id="edit-author"
placeholder="${bookData.author}">
        <input required id="edit-coverImage"
placeholder="${bookData.coverImage}">
        <input required id="edit-description"
placeholder="${bookData.description}">
        <input type="submit" value="Edit Book">
      </div>`

  } else if (e.target.dataset.action === 'delete') {
    console.log('you pressed delete')
  }
}) // end of eventListener for editing and deleting a book
```

I've also added a new `const editButton` , which finds the edit button we're targeting, and then we can `disable` that button so that we can't keep clicking it. This way we can't add more than one edit field at a time.

also called *The Kingkiller Chronicle: Day One*, is a fantasy novel written by American author Patrick Rothfuss. It is the first book in a series called *The Kingkiller Chronicle* and followed by *The Wise Man's Fear*.



The 'edit' button is greyed out. We can't click it again until the page is refreshed!

Next we'll add an eventListener for when a user hits the "Edit Book" button:

```

editForm.addEventListener("submit", (e) => {
    event.preventDefault()

    const titleInput = document.querySelector("#edit-title").value
    const authorInput = document.querySelector("#edit-author").value
    const coverImageInput = document.querySelector("#edit-coverImage").value
    const descInput = document.querySelector("#edit-description").value
    const editedBook =
document.querySelector(`#book-${bookData.id}`)

    fetch(`${bookURL}/${bookData.id}`, {
        method: 'PATCH',
        body: JSON.stringify({
            title: titleInput,
            author: authorInput,
            coverImage: coverImageInput,
            description: descInput
        }),
        headers: {
            'Content-Type': 'application/json'
        }
    }).then( response => response.json() )
    .then( book => {
        editedBook.innerHTML = `
<div id=book-${book.id}>
  <h2>${book.title}</h2>
  <h4>Author: ${book.author}</h4>
  
  <p>${book.description}</p>
  <button data-id=${book.id} id="edit-${book.id}" data-
action="edit">Edit</button>
  <button data-id=${book.id} id="delete-${book.id}"
data-action="delete">Delete</button>
</div>
<div id=edit-book-${book.id}>
</div>`
        editedForm.innerHTML = ""
    })
})
// end of this event Listener for edit submit

```

This will do several things: first it will create constants containing the data in the input fields of the `editForm`. This way, when a user hits “Edit Button”, the constants will populate with whatever the user typed into the input fields. It will also create a constant `editedBook` to represent the exact book we are editing, for use later on.

Next, it will start a `fetch` on that book’s *specific* URL. Similar to the `POST` request we made before, this will `PATCH` the previously existing data with our newly generated inputs.

Then it displays the newly edited book’s information in real time by changing `editedBook`’s `innerHTML`.

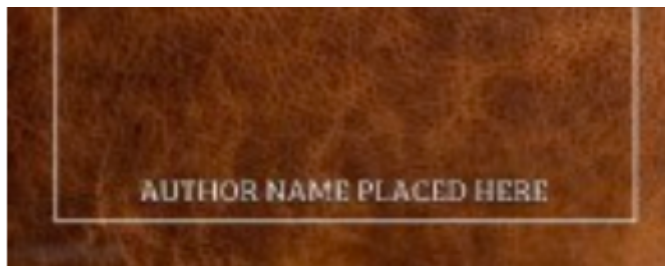
Finally, it changes the `editForm`’s `innerHTML` to an empty string, hiding the `editForm` from view.

We can now make changes to our books by clicking “Edit Book”, and having those changes appear in real-time *and* persist on the database!

a fake name

Author: a fake author





a fake book description

Edit

Delete

Perdido Street Station

Edited "The Name of the Wind"

Step 6: The "D" of CRUD

The final step! Thankfully, `DELETE` is much easier than previous steps. We've already made that nifty `else if (e.target.dataset.action === "delete")`. Using that, we can `querySelector` for the book we're currently deleting, and `remove()` it:

```
document.querySelector(`#book-${e.target.dataset.id}`).remove()
```

Next, we'll create a new `fetch` on that book's specific URL, and call the method `DELETE` on it.

```
} else if (e.target.dataset.action === 'delete') {
  document.querySelector(`#book-${e.target.dataset.id}`).remove()
  fetch(`${bookURL}/${e.target.dataset.id}`, {
    method: 'DELETE',
    headers: {
      'Content-Type': 'application/json'
    }
  }).then( response => response.json())
}
```


And that's it! Now if you click the “delete” button on the fake book we edited, you should see it disappear from the DOM as well as the database!

Books

<input type="text" value="title..."/>	<input type="text" value="author..."/>	<input type="text" value="cover-image..."/>	<input type="text" value="description..."/>	<input type="button" value="Add Book"/>
---------------------------------------	--	---	---	---

Perdido Street Station

Author: China Mieville



Goodbye, fake book

And that's basically everything!

Obviously there are many ways to do full CRUD in JavaScript, in more or less efficient ways than what I've described here, but this is one way that I hope illustrates some of the concepts and reasoning behind why things are done in particular ways. This isn't the most efficient app, and it certainly has its share of flaws, but it's a quick work in progress that I plan to improve in the future with cleaner, more efficient code.

Resources

[jmartinez729/book-lister-app-full-crud-js](#)

Contribute to [jmartinez729/book-lister-app-full-crud-js](#) development

by creating an account on GitHub.

github.com

JavaScript HTML DOM

Well organized and easy to understand Web building tutorials with lots of examples of how to use HTML, CSS, JavaScript...

www.w3schools.com

UI rendering: optimistic vs pessimistic

The popularity of AJAX has grown since its inception in 2005 because of the rich user experience that it provides...

medium.com

EventTarget.addEventListener()

The EventTarget method addEventListener() sets up a function that will be called whenever the specified event is...

developer.mozilla.org

Using Fetch

This kind of functionality was previously achieved using XMLHttpRequest. Fetch provides a better alternative that can...

developer.mozilla.org

[About](#) [Help](#) [Legal](#)

Get the Medium app

