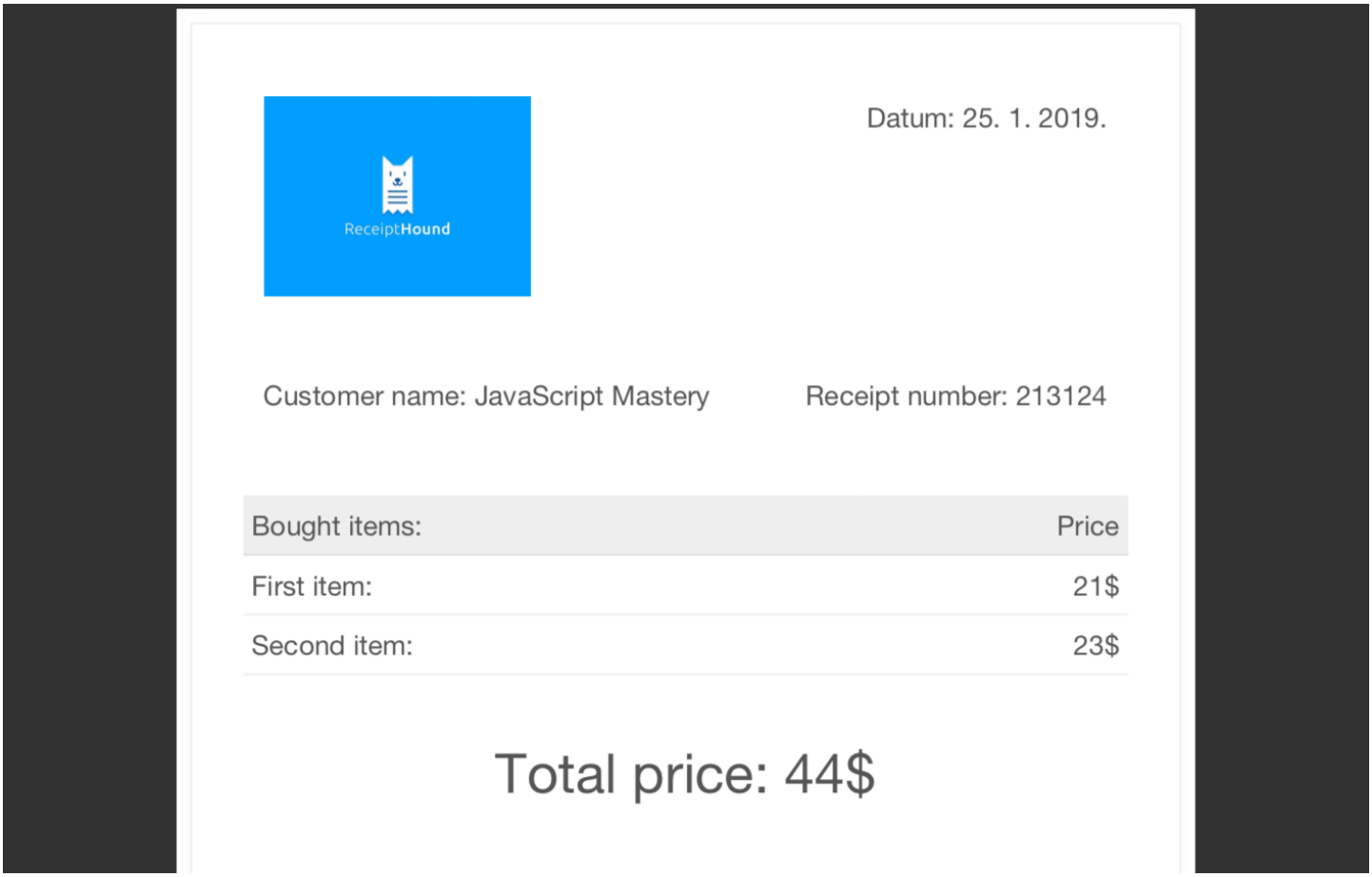
How to Generate Dynamic PDFs Using React and NodeJS

Adrian Hajdin
Feb 4, 2019 · 6 min read

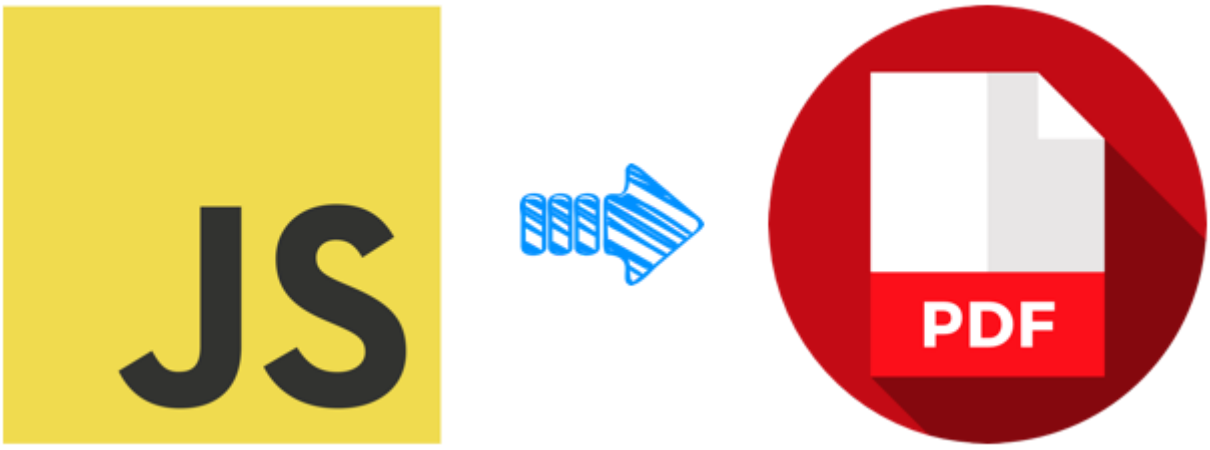


Finished Product

Introduction

In this article, you will learn how to generate dynamic PDFs using HTML code as a template.

For this particular example, we are going to create a simple receipt that will have dynamic data coming from React's state object.



Generate PDFs using Java

Table of Contents

1. Setting the project up with CRA and creating a simple Express server.
2. Sending data from the client to the server and returning generated PDF.

Just a note...

Along with writing this article, I have also created a YouTube video!
You can follow along and code while watching. I advise you to first glance over the article, code along with the video and then read the article.

Link of the video: Generate Dynamic PDFs Using React and Node JS

Project Setup

1. **Create a new directory**

   ```
   mkdir pdfGenerator && cd pdfGenerator
   ```

2. **Create a new React App** with

   ```
   create-react-app client
   ```

   and then move into newly created directory and install dependencies

   ```
   cd client && npm i -S axios file-saver
   ```

3. **Create an Express server** with

   ```
   mkdir server && cd server && touch index.js && npm init
   ```

   press enter a couple of times to initialize package.json and then run `npm i -S express body-parser cors html-pdf` to save all the necessary dependencies.

4. **Add proxy inside of client/package.json**, above the dependencies, simply add `"proxy": "http://localhost:5000/",` so you can call the localhost from the client.

5. **Open two different terminals:**
   First one: go into the client directory and run `npm start`
   Second one: go into the server directory and run `nodemon index.js`

Initial setup — Client Side

The client side of this project for generating dynamic PDFs is going to be straight forward.

First import the dependencies:

```
import axios from 'axios';
import { saveAs } from 'file-saver';
```

At the top, initialize the state:

```
state = {
   name: 'Adrian',
   receiptId: 0,
   price1: 0,
   price2: 0,
}
```

Delete the JSX that create-react-app generated for us and paste the following these few inputs with a button that will submit values to the state:

```
<div className="App">
   <input type="text" placeholder="Name" name="name" onChange
{this.handleChange}/>
   <input type="number" placeholder="Receipt ID" name="receiptId"
onChange={this.handleChange}/>
   <input type="number" placeholder="Price 1" name="price1"
onChange={this.handleChange}/>
   <input type="number" placeholder="Price 2" name="price2"
onChange={this.handleChange}/>
   <button onClick={this.createAndDownloadPdf}>Download PDF</button>
</div>
```

Create the `handleChange` method that is going to update the state of our input fields:

```
handleChange = ({ target: { value, name } }) => this.setState({
[name]: value });
```

Now that this is done we can finally move on to generating PDFs.

Inside of `createAndDownloadPdf` we will create a post request that will send the data to our server:

```
createAndDownloadPdf = () => {
   axios.post('/create-pdf', this.state)
}
```

Before we move any further with the client side, we need to set up the routes on the backend. We can provide them with the necessary information, generate PDFs and finally request generated PDFs back to the client.

Initial setup — Server Side

The client side of this project for generating dynamic PDFs is going to include only two routes. One for generating PDFs. One for sending them back.

First import all the necessary dependencies:

```
const express = require('express');
const bodyParser = require('body-parser');
const pdf = require('html-pdf');
const cors = require('cors');
```

Initialize app using express and set up the port:

```
const app = express();
const port = process.env.PORT || 5000;
```

Setup body parser so we can parse incoming request bodies available under the `req.body` property. Setup cors so we don't get blocked by cross-origin: `Cross-Origin Request Blocked`.

```
app.use(cors());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
```

Finally:

```
app.listen(port, () => console.log(`Listening on port ${port}`));
```

Now we can start creating the logic behind PDF creation.

Creating an HTML Template for our PDF

We need to create the HTML template for our PDF file. The possibilities are endless. Everything that you can create with pure HTML and CSS can be represented as PDF. So let's create a new directory with index.js inside of it:

```
mkdir documents && cd documents && touch index.js
```

Inside of index.js we will simply export one arrow function that is going to return all of the HTML code. In here we can use the parameters we will pass in when we actually call this function.

```
module.exports = ({ name, price1, price2, receiptId }) => { ... }
```

For this project, I will provide you with an example PDF template that you can simply copy. But remember, in here you can create anything that suits your needs: PDF Template Code

```
1  module.exports = ({ name, price1, price2, receiptId }) => {
2      const today = new Date();
3  return `
```

```
<!doctype html>
<html>
   <head>
      <meta charset="utf-8">
      <title>PDF Result Template</title>
      <style>
         .invoice-box {
         max-width: 800px;
         margin: auto;
         padding: 30px;
         border: 1px solid #eee;
         box-shadow: 0 0 10px rgba(0, 0, 0, .15);
         font-size: 16px;
         line-height: 24px;
         font-family: 'Helvetica Neue', 'Helvetica',
         color: #555;
         }
         .margin-top {
         margin-top: 50px;
         }
         .justify-center {
         text-align: center;
         }
         .invoice-box table {
         width: 100%;
         line-height: inherit;
         text-align: left;
         }
         .invoice-box table td {
         padding: 5px;
         vertical-align: top;
         }
         .invoice-box table tr td:nth-child(2) {
         text-align: right;
         }
         .invoice-box table tr.top table td {
         padding-bottom: 20px;
         }
         .invoice-box table tr.top table td.title {
         font-size: 45px;
         line-height: 45px;
         color: #333;
         }
         .invoice-box table tr.information table td {
         padding-bottom: 40px;
         }
         .invoice-box table tr.heading td {
         background: #eee;
```

```
            background: #eee;
            border-bottom: 1px solid #ddd;
            font-weight: bold;
            }
            .invoice-box table tr.details td {
            padding-bottom: 20px;
            }
            .invoice-box table tr.item td {
            border-bottom: 1px solid #eee;
            }
            .invoice-box table tr.item.last td {
            border-bottom: none;
            }
            .invoice-box table tr.total td:nth-child(2) {
            border-top: 2px solid #eee;
            font-weight: bold;
            }
            @media only screen and (max-width: 600px) {
            .invoice-box table tr.top table td {
            width: 100%;
            display: block;
            text-align: center;
            }
            .invoice-box table tr.information table td {
            width: 100%;
            display: block;
            text-align: center;
            }
            }
         </style>
      </head>
      <body>
         <div class="invoice-box">
            <table cellpadding="0" cellspacing="0">
               <tr class="top">
                  <td colspan="2">
                     <table>
                        <tr>
                           <td class="title"><img  src="https://i2.wp.com/cleverlogos.co/wp-c
                              style="width:100%; max-width:156px;"></td>
                           <td>
                              Datum: ${`${today.getDate()}. ${today.getMonth() + 1}. ${today.
                           </td>
                        </tr>
                     </table>
                  </td>
               </tr>
               <tr class="information">
```

```
                    <td colspan="2">
                        <table>
                            <tr>
                                <td>
                                    Customer name: ${name}
                                </td>
                                <td>
                                    Receipt number: ${receiptId}
                                </td>
                            </tr>
                        </table>
                    </td>
                </tr>
                <tr class="heading">
                    <td>Bought items:</td>
                    <td>Price</td>
                </tr>
                <tr class="item">
                    <td>First item:</td>
                    <td>${price1}$</td>
                </tr>
                <tr class="item">
                    <td>Second item:</td>
                    <td>${price2}$</td>
                </tr>
            </table>
            <br />
            <h1 class="justify-center">Total price: ${parseInt(price1) + parseInt(price2)}$</
        </div>
    </body>
```

Generating PDFs

We are going to have two routes:

`POST` route is going to fetch the data and generate a PDF.

`GET` route is going to send the generated PDF to the client.

`create-pdf` route:

Inside of this `/create-pdf` post route we are using to use `pdf.create()` that we imported from the module `html-pdf`.

Then, as the first parameter to the `pdf.create` method, we will pass the template along with all the information coming from the client inside of `req.body`.

Immediately after, we call another method, `toFile()` and pass the desired name of our pdf document, as well as a callback arrow function that will return `Promise.reject()` if it encounters an error, or `Promise.resolve()` if everything finishes smoothly.

```
app.post('/create-pdf', (req, res) => {
  pdf.create(pdfTemplate(req.body), {}).toFile('rezultati.pdf',
(err) => {
    if(err) {
        return console.log('error');
    }

res.send(Promise.resolve())
  });
})
```

`fetch-pdf` route:

While we're here, we can also create a get request that will be called after the previous one finishes. In here we will simply retrieve the document and send it to the client using `res.sendFile()`.

```
app.get('/fetch-pdf', (req, res) => {
  res.sendFile(`${__dirname}/rezultati.pdf`);
});
```

Finishing up

Finally, we can get back to the client and use axios to make those requests:

```
createAndDownloadPdf = () => {
   axios.post('/create-pdf', this.state)
}
```

After we make a post request, our PDF will be generated and we need to call get request that will send the file to the client.

We chain `.then()` onto our `axios.post()` request.
We can do that since we returned either resolved or rejected promise.

```
.then(() => axios.get('/fetch-pdf', { responseType: 'blob' }))
```

Above, we can see the use of `responseType` that is equal to `blob`, before we move any further, we need to explain what blobs are.

Blobs

One sentence explanation of what blobs are without any technical terms that I could come up with: **Blobs are immutable objects that represent raw data.**

They are often used for representing data that isn't necessarily in a JavaScript format.
A blob object represents a **chunk of bytes** that hold the data of a **file**.
Although it may seem like blob is a reference to an actual file, it is not.

Blobs allow you to construct file-like objects on the client that you can pass to API's that expect URLs instead of requiring the server to provide the files.

Now that we know what blobs are we can move on and actually create one, which will be a representation of our PDF file.

Know that we know what blobs are, we can chain one last `.then(res)` along with the response to the previous one like this:

```
.then(() => axios.get('/fetch-pdf', { responseType: 'blob' }))
.then((res) => {})
```

Finally, we can create a blob using Blob constructor:

```
.then((res) => {
  const pdfBlob = new Blob([res.data], { type: 'application/pdf' });
}
```

Here we simply create a `pdfBlob`, by passing `res.data` that is coming from our server and we set the type to `application/pdf`.

After we have created a blob, we can use `saveAs()`, that we imported from `file-saver` module to download the newly generated PDF.

`saveAs` takes in the newly created `pdfBlob` as the first parameter, and name of the file as the second one:

```
saveAs(pdfBlob, 'generatedDocument.pdf')
```

That's it!

You made it all the way until the end! If you get stuck along the way, feel free to ask and leave feedback in the comments down below. Most helpful would be the support on **YouTube** since I have just created a channel! **Click here**, there is a lot of interesting stuff coming soon! :)

You can also check out the article I did on:
Learning Async/Await on a Real World Project.

You can learn to code at microverse!

Nodejs React JavaScript Programming Software Development