JavaScript

Custom PDF Rendering in JavaScript with Mozilla's PDF.Js

Tools & Libraries

Imran Latif
May 16, 2016

Share

> *This article was peer reviewed by **Jani Hartikainen**, **Florian Rappl**, **Jezen Thomas** and **Jeff Smith**. Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!*

When it comes to the Web, almost every modern browser supports viewing of PDF documents natively. But, that native component is outside of the developer's control. Imagine that because of some business rule in your web app, you wanted to disable the `Print` button, or display only few pages while others require paid membership. You can use browser's native PDF rendering capability by using the `embed` tag, but since you don't have programmatic access you can't control the rendering phase to suit your needs.

Luckily, there now exists such a tool, **PDF.js**, created by Mozilla Labs, which can render PDF documents in your browser. Most importantly, you as a developer have full control over rendering the PDF document's pages as per your requirements. Isn't this cool? Yes, it is!

Let's see what PDF.js actually is.

What Is PDF.js

PDF.js is Portable Document Format (PDF) built around HTML5-based technologies, which means it can be used in modern browsers without installing any third-party plugins.

PDF.js is already in use at many different places including some online file sharing services like **Dropbox**, **CloudUp**, and **Jumpshare** to let users view PDF documents online without relying on browser's native PDF rendering capability.

PDF.js is without any doubt an awesome and essential tool to have in your web app, but integrating it isn't as straightforward as it might seem. There is little to no documentation available on how to integrate certain features like rendering text-layers or annotations (external/internal links), and supporting password protected files.

In this article, we will be exploring PDF.js, and looking at how we can integrate different features. Some of the topics which we will cover are:

- Basic Integration
- Rendering Using SVG
- Rendering Text-Layers
- Zooming in/Out

Basic Integration

Downloading the Necessary Files

PDF.js, as it's name states is a JavaScript library which can be used in browser to render PDF documents. The first step is to fetch necessary JavaScript files required by

PDF.js to work properly. Following are two main files required by PDF.js:

- pdf.js
- pdf.worker.js

To fetch aforementioned files, if you are a Node.js user, you can follow **these steps** as mentioned on the GitHub repo. After you are done with the `gulp generic` command, you will have those necessary files.

If, like me, you don't feel comfortable with Node.js there is an easier way. You can use following URLs to download necessary files:

- **https://mozilla.github.io/pdf.js/build/pdf.js**
- **https://mozilla.github.io/pdf.js/build/pdf.worker.js**

The above mentioned URLs point to Mozilla's **live demo** of PDF.js. By downloading files this way, you will always have latest version of the library.

Web Workers and PDF.js

The two files you downloaded contain methods to fetch, parse and render a PDF document. `pdf.js` is the main library, which essentially has methods to fetch a PDF document from some URL. But parsing and rendering PDF is not a simple task. In fact, depending on the nature of the PDF, the parsing and rendering phases might take a bit longer which might result in the blocking of other JavaScript functions.

HTML5 introduced **Web Workers**, which are used to run code in a separate thread from that of browser's JavaScript thread. PDF.js relies heavily on Web Workers to provide a performance boost by moving CPU-heavy operations, like parsing and rendering, off of the main thread. Running processing expensive code in Web Workers is the default in PDF.js but can be turned off if necessary.

Promises in PDF.js

The JavaScript API of PDF.js is quite elegant and easy to use and is heavily based on **Promises**. Every call to the API returns a Promise, which allows asynchronous operations to be handled cleanly.

Hello World!

Let's integrate a simple 'Hello World!' PDF document. The document which we are using in this example can be found at http://mozilla.github.io/pdf.js/examples/learning/helloworld.pdf.

Create a project under your local web-server such that it can be accessed using **http://localhost/pdfjs_learning/index.html**. PDF.js makes Ajax calls to fetch documents in chunks, so in order to make the Ajax call work locally we need to place PDF.js files in a local web-server. After creating the `pdfjs_learning` folder on your local web-server, place the files (`pdf.js`, `pdf.worker.js`) in it that you downloaded above. Place the following code in `index.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>PDF.js Learning</title>
  </head>
  <body>
    <script type="text/javascript" src="pdf.js"></script>
  </body>
</html>
```

As you can see, we've included a link to the main library file, `pdf.js`. PDF.js automatically detects whether your browser supports Web Workers, and if it does, it will attempt to load `pdf.worker.js` from the same location as `pdf.js`. If the file is in

another location, you can configure it using `PDFJS.workerSrc` property right after including the main library:

```
<script type="text/javascript" src="pdf.js"></script>
<script type="text/javascript">
    PDFJS.workerSrc = "/path/to/pdf.worker.js";
</script>
```

If your browser doesn't support Web Workers there's no need to worry as `pdf.js` contains all the code necessary to parse and render PDF documents without using Web Workers, but depending on your PDF documents it might halt your main JavaScript execution thread.

Let's write some code to render the 'Hello World!' PDF document. Place the following code in a `script` tag, below the `pdf.js` tag.

```
// URL of PDF document
var url = "http://mozilla.github.io/pdf.js/examples/learning/helloworld.pdf";

// Asynchronous download PDF
PDFJS.getDocument(url)
  .then(function(pdf) {
    return pdf.getPage(1);
  })
  .then(function(page) {
    // Set scale (zoom) level
    var scale = 1.5;

    // Get viewport (dimensions)
    var viewport = page.getViewport(scale);

    // Get canvas#the-canvas
    var canvas = document.getElementById('the-canvas');

    // Fetch canvas' 2d context
    var context = canvas.getContext('2d');
```

```
    // Set dimensions to Canvas
    canvas.height = viewport.height;
    canvas.width = viewport.width;

    // Prepare object needed by render method
    var renderContext = {
      canvasContext: context,
      viewport: viewport
    };

    // Render PDF page
    page.render(renderContext);
  });
```

Now create a `<canvas>` element with an id `the-canvas` within `body` tag.

```
<canvas id="the-canvas"></canvas>
```

After creating the `<canvas>` element, refresh your browser and if you have placed everything at it's proper place, you should see **Hello, world!** printed in your browser. But that's not an ordinary **Hello, world!**. The **Hello, world!** you are seeing is basically an entire PDF document being rendered in your browser by using JavaScript code. Embrace the awesomeness!

Let's discuss different parts of aforementioned code which made PDF document rendering possible.

`PDFJS` is a global object which you get when you include `pdf.js` file in browser. This object is the base object and contains various methods.

`PDFJS.getDocument()` is the main entry point and all other operations are performed within it. It is used to fetch the PDF document asynchronously, sending multiple Ajax requests to download document in chunks, which is not only fast but efficient as well. There are different parameters which can be passed to this method but the most important one is the URL pointing to a PDF document.

`PDFJS.getDocument()` returns a Promise which can be used to place code which will be executed when PDF.js is done fetching document. The success callback of the Promise is passed an object which contains information about fetched PDF document. In our example, this argument is named `pdf`.

You might be wondering if, since the PDF document is fetched in chunks, for documents that are huge in size the success callback would only be called after a delay of quite few seconds (or even minutes). In fact, the callback will fire as soon as the necessary bytes for first page have been fetched.

`pdf.getPage()` is used to get individual pages in a PDF document. When you provide a valid page number, `getPage()` returns a promise which, when resolved, gives us a `page` object that represents the requested page. The `pdf` object also has a property, `numPages`, which can be used to get total number of pages in PDF document.

`scale` is the zoom-level we want PDF document's pages to render at.

`page.getViewport()` returns PDF document's page dimensions for the provided zoom-level.

`page.render()` requires an object with different key/value pairs to render PDF page onto the Canvas. In our example, we have passed Canvas element's `2d` context and `viewport` object which we get from `page.getViewport` method.

Rendering Using SVG

PDF.js supports two modes of rendering. It's default and popular mode of rendering is Canvas based. But it also allows you to render PDF documents using SVG. Let's render the Hello World! PDF document from previous example in SVG.

Update success callback of `pdf.getPage()` with the following code to see PDF.js' SVG rendering in action.

```
.then(function(page) {

  // Set scale (zoom) level
  var scale = 1.5;

  // Get viewport (dimensions)
  var viewport = page.getViewport(scale);

  // Get div#the-svg
  var container = document.getElementById('the-svg');

  // Set dimensions
  container.style.width = viewport.width + 'px';
  container.style.height = viewport.height + 'px';

  // SVG rendering by PDF.js
  page.getOperatorList()
    .then(function (opList) {
      var svgGfx = new PDFJS.SVGGraphics(page.commonObjs, page.objs);
      return svgGfx.getSVG(opList, viewport);
    })
    .then(function (svg) {
      container.appendChild(svg);
    });

});
```

Replace the `<canvas>` element in your body tag with `<div id="the-svg"></div>` and refresh your browser.

If you have placed the code correctly you will see **Hello, world!** being rendered, but this time it's using SVG instead of Canvas. Go ahead and check the HTML of the page and you will see that the entire rendering has been done using standard SVG components.

As you can see, PDF.js doesn't restrict you to a single rendering mechanism. You can either use Canvas or SVG rendering depending upon your requirements. For the rest of the article, we will be using Canvas-based rendering.

Rendering Text-Layers

PDF.js gives you the ability to render text layers atop PDF pages that have been rendered using Canvas. To do this, we need to fetch an additional JavaScript file from PDF.js GitHub's repo. Go ahead and download the **text_layer_builder.js plugin**. We also need to fetch its corresponding CSS file, **text_layer_builder.css**. Download both files and place them in the `pdfjs_learning` folder on your local server.

Before we get into actual text-layer rendering, let's get a PDF document with some more content than the 'Hello World!' example. The document which we are going to render is again taken from Mozilla's live demo, **here**.

Since this document contains multiple pages, we need to adjust our code a bit. First, remove the `<div>` tag we created in the last example, and replace it with this:

```
<div id="container"></div>
```

This container will be used to hold multiple pages of PDF document. The structure for placing pages rendered as `Canvas` elements is quite simple. Within `div#container`

each page of the PDF will have its own `<div>`. The `id` attribute of `<div>` will have the format `page-#{pdf_page_number}`. For example, the first page in a PDF document would have a `<div>` with `id` attribute set as `page-1` and 12th page would have `page-12`. Inside each of these `page-#{pdf_page_number}` divs, there will be a `Canvas` element.

Let's replace the success callback of `getDocument()` with the following code. Don't forget to update the `url` variable with `http://mozilla.github.io/pdf.js/web/compressed.tracemonkey-pldi-09.pdf` (or some other online PDF document of your choice).

```
PDFJS.getDocument(url)
  .then(function(pdf) {

    // Get div#container and cache it for later use
    var container = document.getElementById("container");

    // Loop from 1 to total_number_of_pages in PDF document
    for (var i = 1; i <= pdf.numPages; i++) {

        // Get desired page
        pdf.getPage(i).then(function(page) {

          var scale = 1.5;
          var viewport = page.getViewport(scale);
          var div = document.createElement("div");

          // Set id attribute with page-#{pdf_page_number} format
          div.setAttribute("id", "page-" + (page.pageIndex + 1));

          // This will keep positions of child elements as per our needs
          div.setAttribute("style", "position: relative");

          // Append div within div#container
          container.appendChild(div);

          // Create a new Canvas element
```

```
            var canvas = document.createElement("canvas");

            // Append Canvas within div#page-#{pdf_page_number}
            div.appendChild(canvas);

            var context = canvas.getContext('2d');
            canvas.height = viewport.height;
            canvas.width = viewport.width;

            var renderContext = {
              canvasContext: context,
              viewport: viewport
            };

            // Render PDF page
            page.render(renderContext);
          });
      }
  });
```

Refresh your browser and wait for few seconds (while the new PDF document is fetched in background) and as soon as the document has finished loading you should see beautifully rendered PDF pages in your browser. Now we've seen how to render multiple pages, let's discuss how to render the text-layers.

Add the following two lines to `index.html` to include the necessary files required for text-layer rendering:

```
<link type="text/css" href="text_layer_builder.css" rel="stylesheet">

<script type="text/javascript" src="text_layer_builder.js"></script>
```

PDF.js renders the text-layer above the Canvases within multiple `<div>` elements, so it's better to wrap all those `<div>` elements within a container element. Replace

`page.render(renderContext)` line with following code to see text-layers in action:

```
page.render(renderContext)
  .then(function() {
    // Get text-fragments
    return page.getTextContent();
  })
  .then(function(textContent) {
    // Create div which will hold text-fragments
    var textLayerDiv = document.createElement("div");

    // Set it's class to textLayer which have required CSS styles
    textLayerDiv.setAttribute("class", "textLayer");

    // Append newly created div in `div#page-#{pdf_page_number}`
    div.appendChild(textLayerDiv);

    // Create new instance of TextLayerBuilder class
    var textLayer = new TextLayerBuilder({
      textLayerDiv: textLayerDiv,
      pageIndex: page.pageIndex,
      viewport: viewport
    });

    // Set text-fragments
    textLayer.setTextContent(textContent);

    // Render text-fragments
    textLayer.render();
  });
```

Refresh your browser and this time you will not only see PDF pages being rendered but you can also select and copy text from them. PDF.js is so cool!

Let's discuss some important portions of above code snippet.

`page.render()`, as with any other method in PDF.js, returns a promise which is resolved when a PDF page has been successfully rendered onto the screen. We can use the success callback to render text-layers.

`page.getTextContent()` is a method which returns text fragments for that particular page. This returns a promise as well and in success callback of that promise text fragments representation is returned.

`TextLayerBuilder` is a class which requires some parameters we already have from `pdf.getPage()` for each page. The `textLayerDiv` parameter represents the `<div>` which will be used as a container for hosting multiple `<div>`s each representing some particular text fragment.

The newly created instance of `TextLayerBuilder` has two important methods: `setTextContent()`, which is used to set text fragments returned by `page.getTextContent()`, and `render()`, which is used to render text-layer.

As you can see we are assigning a CSS class `textLayer` to `textLayerDiv`. This class has styles which will make sure that the text fragments fit nicely atop the Canvas elements so that user can select/copy text in a natural way.

Zooming In/Out

With PDF.js you can also control the zooming of PDF document. In fact, zooming is quite straightforward and we just need to update the `scale` value. Increase or decrease `scale` with your desired factor to alter the zoom level. This is left as an exercise for the reader, but do try this out and let us know how you get on in the comments.

Conclusion

PDF.js is an awesome tool which provides us with a flexible alternative to the browsers' native PDF components using JavaScript. The API is simple, precise and elegant and can be used as you see fit. Let me know in comments about how you are intending to use PDF.js in your next project!

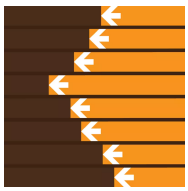
Share This Article

Imran Latif

Imran Latif is a Web Developer and Ruby on Rails and JavaScript enthusiast from Pakistan. He is a passionate programmer and always keeps on learning new tools and technologies. During his free time he watches tech videos and reads tech articles to increase his knowledge.

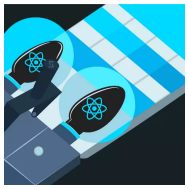
Mozilla **nilsonj** **pdf** **pdf.js**

Up Next



Writing Async Libraries – Let's Convert HTML to PDF

Christopher Pitt



Universal React Rendering: How We Rebuilt SitePoint

Brad Denver

Adventures in Aurelia: Creating a Custom PDF Viewer
Jedd Ahyoung

Filling out PDF Forms with PDFtk and PHP
Reza Lavarian

Creating Custom UI Components & Live Rendering
Joyce Echessa

JavaScript and jQuery PDF Viewer Plugins
Sam Deering