

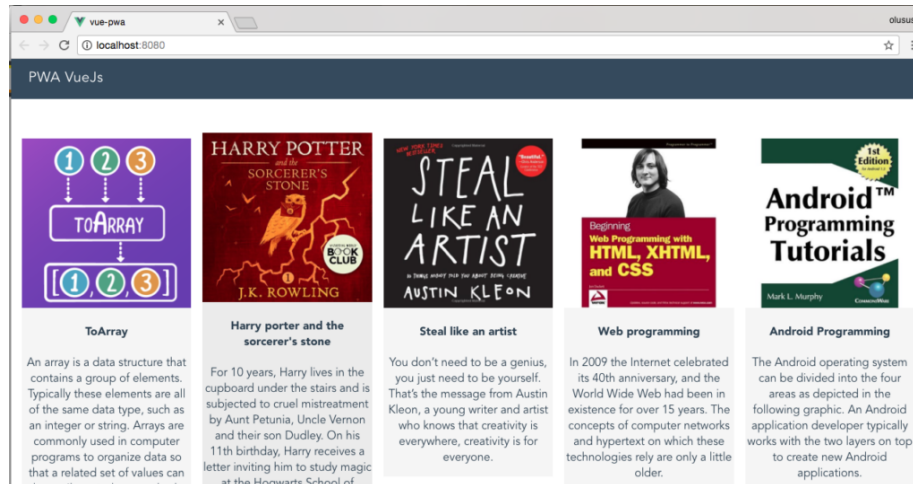
CHRIS NWAMBA

Getting started with PWA using Vue

DECEMBER 6, 2017

Imagine a web application that you can browse, even with a very poor network connection. An offline web application that will still give access to up-to-date data and keep engaging users. This is the promise of progressive web applications and, as evident from testimonies of increased conversion rate by leading companies such as Twitter, Forbes, AliExpress, and others, we should know by now that progressive web applications are the future.

This article shows and explains a hands-on experience on how to build a Progressive Web App using Vue from scratch. We will build a very simple book listing application with descriptions for each book. This UI will be built with Vue, a



Our application is a very simple one but suitable to get you started with building progressive web applications using Vue.

Progressive Web Apps

A Progressive Web Application (PWA) is a web application that offers an app-like user experience on the web. Modern web technological innovations such as service workers, Native APIs have contributed greatly to progressive web apps and help improve the quality of web applications. For more details about the background, history or principles of PWAs please visit the [Google developer page](#).

Setting Up Vue with PWA Features

Vue-CLI, a command line tool for generating Vue projects, comes with a few project templates. For the sake of this tutorial, we will choose `pwa-template`. With this, Vue-CLI is going to set up a dummy Vue application with Webpack, vue-loader and basic offline support powered by service workers.

A service worker is a background worker that runs independently in the browser. It doesn't make use of the main thread during execution. In fact, it's unaware of the

Setting up a service worker simplifies the process of making an app run offline. Even though setting it up is simple, things can go really bad when it's not done right. For this reason, a lot of community-driven utility tools exist to help scaffold a service worker with all the recommended configurations. Vue is not an exception.

Use the command below to install Vue-CLI if you don't have it installed already:

```
npm install -g vue-cli
```

Then initialize your application with :

```
vue init pwa vue-pwa
```

You will be required to answer a few questions, including the Vue build type, whether to install `vue-router`, whether to use ESLint and whether to set up unit tests. Choosing the default answers is perfect for our demo. Afterwards, the process will create a project folder with the following subfolders: `build`, `config`, `src`, `static`, `test`.

The major difference of this template is in the `build/webpack.prod.conf.js` file:

```
// service worker caching
new SWPrecacheWebpackPlugin({
  cacheId: 'my-vue-app',
  filename: 'service-worker.js',
  staticFileGlobs: ['dist/**/*.{js,html,css}'],
  minify: true,
  stripPrefix: 'dist/'
})
```

the files that match the glob expression in `staticFileGlobs`. As you can see, it is matching all the files in the `dist` folder. This folder is also generated after running the `build` command. We will see it in action later in this tutorial.

Create A Vue Component

Each of the books listed in our application will have an image for the book cover, a book title and a short description of the book.

Create a new component in `src/components/Book.vue` with the template below:

```
<template>
<div class="book">
  <div class="booksDetails">
    
    <h4>{{ list.title }}</h4>
    <p>{{ list.description }}</p>
  </div>
</div>
</template>
```

The template above expects a `list` property from whatever parent it will have in the near future.

To indicate that, add a Vue object with the `props` property:

```
<template>
.....
</template>

<script>
  export default {
    props: ['list'],
    name: 'book'
  }
```

To add basic styling to our Books.vue component:

```
<template>
  ....
</template>
<script>
  .....
</script>

<style>
.book {
  background: #F5F5F5;
  display: inline-block;
  margin: 0 0 1em;
  width: 100%;
  cursor: pointer;
  -webkit-perspective: 1000;
  -webkit-backface-visibility: hidden;
  transition: all 100ms ease-in-out;
}
.book:hover {
  transform: translateY(-0.5em);
  background: #EBEBEB;
}
img {
  display: block;
  width: 100%;
}
</style>
```

Since we are not really connecting to a backend server, lets create an array of fake data for our example application. Create `src/db.json`. You can find this [here](#) , alternatively, here is a truncated version of it:

```
[
  {
    "id": 1,
    "title": "The Great Gatsby",
    "author": "F. Scott Fitzgerald",
    "year": 1925,
    "genre": "Fiction",
    "rating": 4.5
  },
  {
    "id": 2,
    "title": "1984",
    "author": "George Orwell",
    "year": 1949,
    "genre": "Dystopian",
    "rating": 4.7
  },
  {
    "id": 3,
    "title": "The Catcher in the Rye",
    "author": "J.D. Salinger",
    "year": 1951,
    "genre": "Fiction",
    "rating": 4.2
  },
  {
    "id": 4,
    "title": "The Hobbit",
    "author": "J.R.R. Tolkien",
    "year": 1937,
    "genre": "Fantasy",
    "rating": 4.8
  },
  {
    "id": 5,
    "title": "The Lord of the Rings",
    "author": "J.R.R. Tolkien",
    "year": 1954,
    "genre": "Fantasy",
    "rating": 4.9
  },
  {
    "id": 6,
    "title": "The Silmarillion",
    "author": "J.R.R. Tolkien",
    "year": 1977,
    "genre": "Fantasy",
    "rating": 4.6
  },
  {
    "id": 7,
    "title": "The Hobbit and The Lord of the Rings",
    "author": "J.R.R. Tolkien",
    "year": 1954,
    "genre": "Fantasy",
    "rating": 4.9
  },
  {
    "id": 8,
    "title": "The Silmarillion",
    "author": "J.R.R. Tolkien",
    "year": 1977,
    "genre": "Fantasy",
    "rating": 4.6
  },
  {
    "id": 9,
    "title": "The Hobbit and The Lord of the Rings",
    "author": "J.R.R. Tolkien",
    "year": 1954,
    "genre": "Fantasy",
    "rating": 4.9
  },
  {
    "id": 10,
    "title": "The Silmarillion",
    "author": "J.R.R. Tolkien",
    "year": 1977,
    "genre": "Fantasy",
    "rating": 4.6
  }
]
```

```

    },
    {
      "imageUrl": "http://res.cloudinary.com/yemiwebby",
      "title": "Coding For Dummies",
      "description": "Coding For Dummies is the perfec
    },
    {
      "imageUrl": "http://res.cloudinary.com/yemiwebby",
      "title": "Harry porter and the sorcerer's stone",
      "description": "For 10 years, Harry lives in the
    },
    .....
  ]

```

Next action is to import the newly created `db.json` and consume it in `src/App.vue`:

```

import data from './db.json';

export default {
  name: 'app',
  data() {
    return {
      lists: []
    }
  },
  created() {
    this.lists = data;
  }
}

```

We added a property `list` which we set to the JSON data.

To successfully display the books in the browser, we will need to import the card component in the Vue object, then add it to the App template in `src/App`:

```
<template>
```

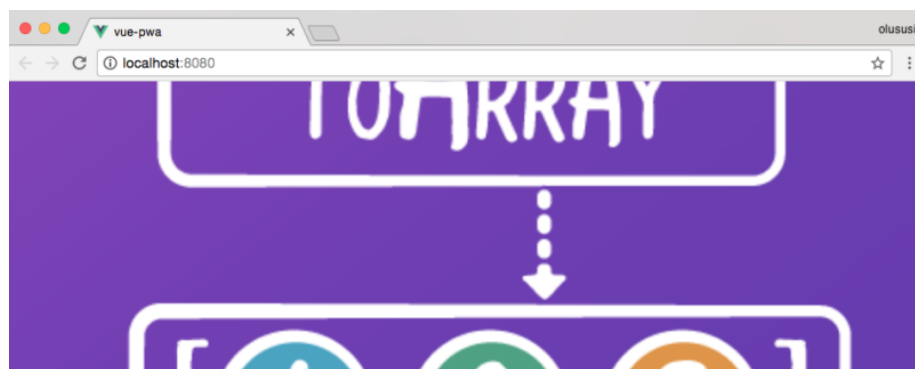
```
    <span>PWA VueJs</span>
  </header>
  <main>
    <div class="wrapper">
      <div class="books">
        <book v-for="list in lists" :key="list.in
      </div>
    </div>
  </main>
</div>
</template>

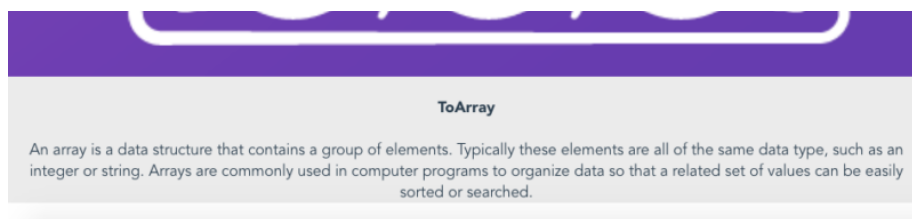
<script>
...
import Book from './components/Books';

export default {
  name: 'app',
  data() {
    ...
  },
  created() {
    ...
  },
  components: {
    Book
  }
}
</script>
```



We iterate over each card and list all the cards in the `.books` element:

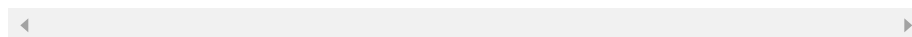




At this point, the app looks absurd and the images are too wide. To make our page look presentable and styled lets add a little bit of css to both books(parent) and book (child):

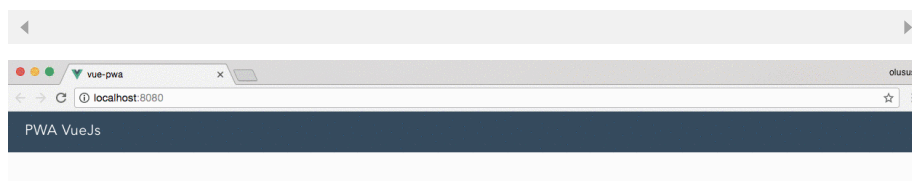
```
/* App.vue */
.books {
  column-count: 1;
  column-gap: 1em;
}

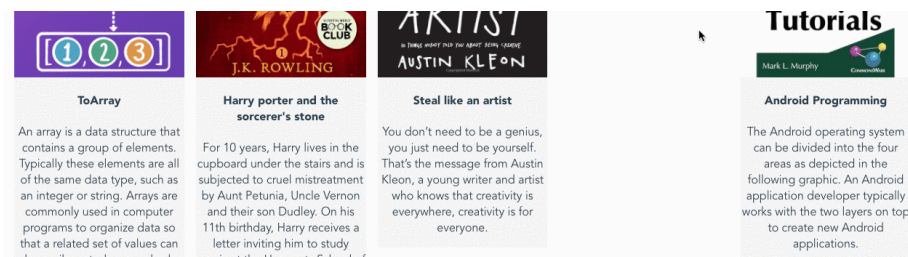
/* Book.vue */
.book {
  display: inline-block;
}
```



If you consider adding animations to the cards, be careful as you will experience flickers while using the transform property. Assuming you have this simple transition on .books:

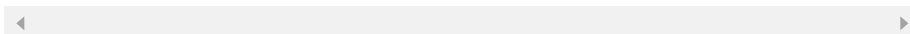
```
/* Book.vue */
.book {
  transition: all 100ms ease-in-out;
}
.book:hover {
  transform: translateY(-0.5em);
  background: #EBEBEB;
}
```





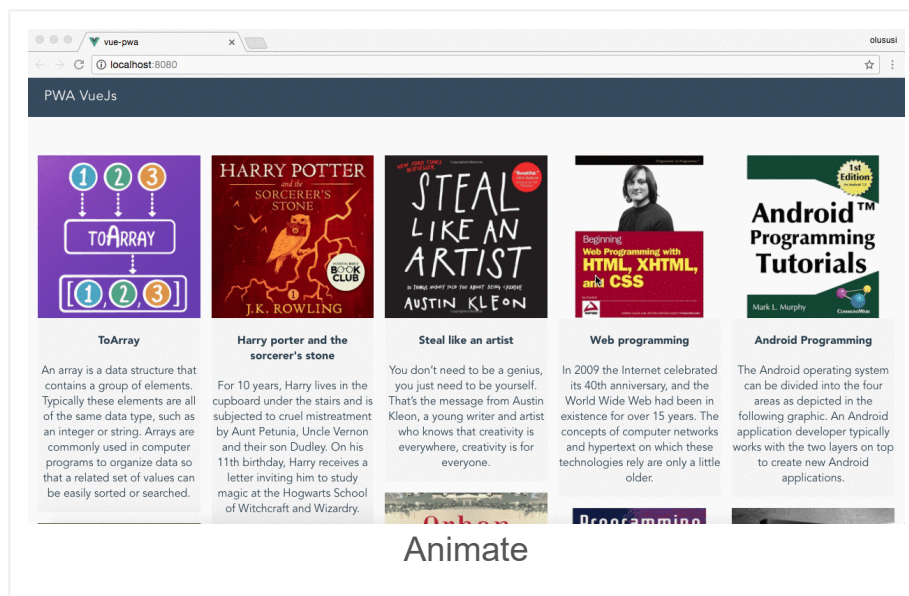
Setting perspective and backface-visibility to the element fixes that:

```
.card {
  -webkit-perspective: 1000;
  -webkit-backface-visibility: hidden;
  transition: all 100ms ease-in-out;
}
```



You can also account for screen sizes and make the grids responsive:

```
@media only screen and (min-width: 500px) {
  .books {
    column-count: 2;
  }
}
@media only screen and (min-width: 700px) {
  .books {
    column-count: 3;
  }
}
@media only screen and (min-width: 900px) {
  .books {
    column-count: 4;
  }
}
@media only screen and (min-width: 1100px) {
  .books {
    column-count: 5;
  }
}
```

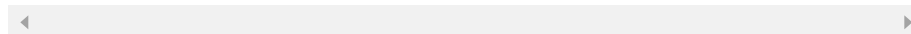


Going Offline

This is the most interesting aspect of this tutorial. Right now, if we were to deploy then go offline, we would get an error message. If you're using Chrome, you will see the popular dinosaur game.

Remember we already have a service worker configured. Now all we need to do is to generate the service worker file when we run the build command. To do so, run the following in your terminal:

```
npm run build
```

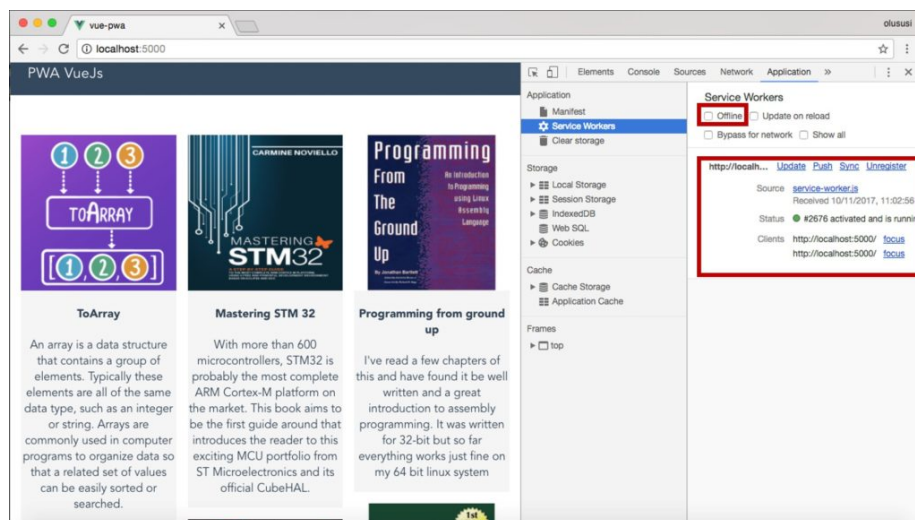


Next, serve the generated build file (found in the the dist folder). There are lots of options for serving files on localhost, but my favorite still remains serve:

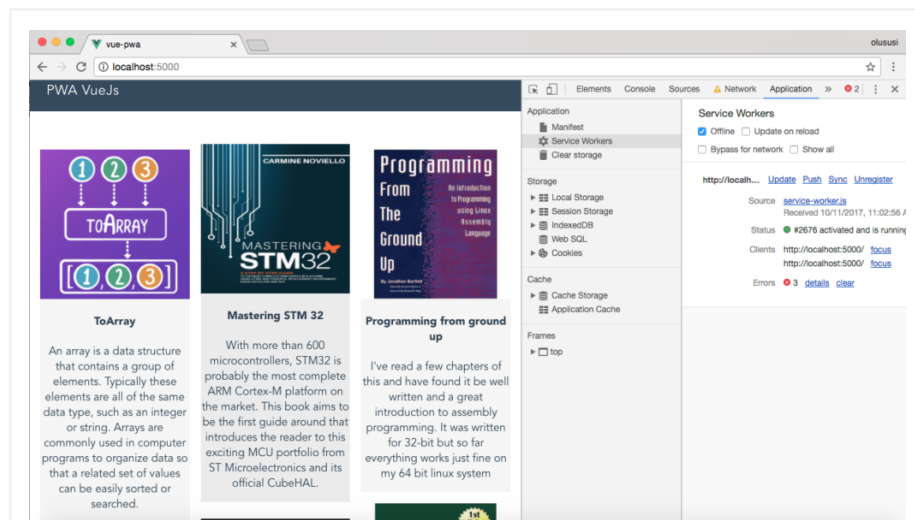
```
#install serve
npm install -g serve
```

```
# serve
serve dist
```

This will launch the app on localhost at port 5000. You would still see the page running as before. Open the developer tools, click the Application tab and select Service Workers. You should see a registered service worker:



As you can see, the status of the service worker shows it's active. Now let's attempt going offline by clicking the check box for offline. Reload the page and you should see our app runs offline:



Cached resources are served when offline

If our book cover images are gone, don't panic. That is because we haven't cached our remote images yet. So let's take a look at the service worker configuration again:

```
new SWPrecacheWebpackPlugin({
  cacheId: 'my-vue-app',
  filename: 'service-worker.js',
  staticFileGlobs: ['dist/**/*.{js,html,css}'],
  minify: true,
  stripPrefix: 'dist/'
})
```

staticFileGlobs property is an array of local files

Since our image is stored remotely, in order to cache stored assets and resources we need to make use of a different property called `runtimeCaching`. This array takes in an object that contains the URL pattern to be cached as well as the caching strategy. Update the `build/webpack.prod.config.js` to include the following block:

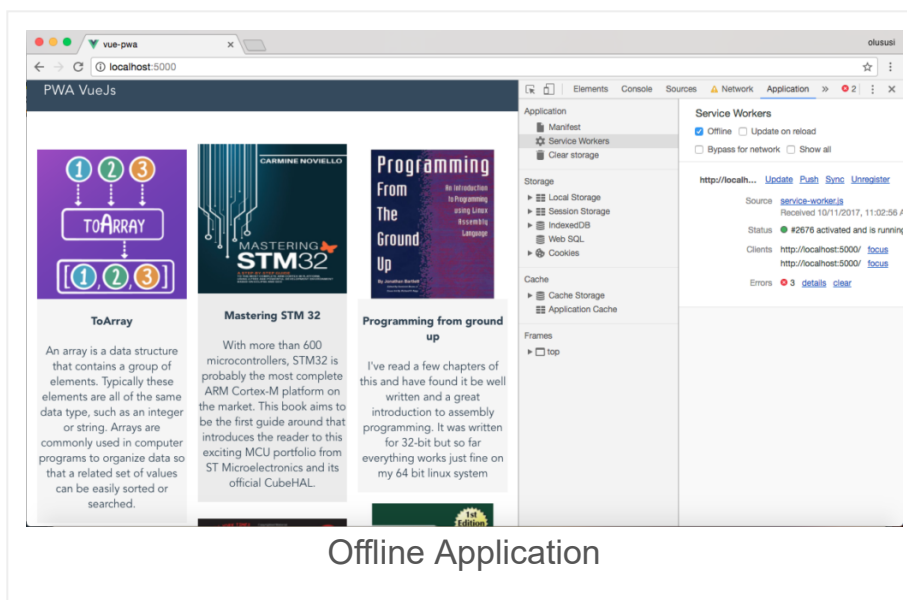
```
// service worker caching
new SWPrecacheWebpackPlugin({
  cacheId: 'my-vue-app',
  filename: 'service-worker.js',
  staticFileGlobs: ['dist/**/*.{js,html,css}'],
  runtimeCaching: [
    {
      urlPattern: /^http:\/\/res\.cloudinary\.co/,
      handler: 'cacheFirst'
    }
  ],
  minify: true,
  stripPrefix: 'dist/'
})
```

For the URL pattern, you will notice that we are using `http`. Service workers, for security reasons, only work with `HTTPS` -with an exception for `localhost`. And since we are using `localhost`, we can proceed, but don't forget to change the URL pattern to `HTTPS` if your assets and resources are served over `HTTPS`:

```
npm run build
```

```
# Serve
serve dist
```

Unregister the service worker from the developer tool, go offline, then reload. Now you have an offline app.



Offline Application

Conclusion

There is a possibility of retaining users on your platform by keeping them engaged at all times. This tutorial has introduced a very simple strategy on how to build a progressive web application using Vue with very little effort. We were able to create a dummy Vue project by using `pwa-template`, and we later went on to make our web application work offline by using a service worker to cache assets and resources of our web application.

I hope you have been able to learn how to cater for users that might experience poor connectivity or no internet access.

Getting started with Pusher is *FREE*

Join more than 250,000 happy developers.

Create a Free Account

PREVIOUS POST

← **How to set up your
Pusher and Librato
integration**

NEXT POST

**Getting started with
StencilJS** →

 COMMENTS


 AUTHOR

2 Comments

Pusher Blog

 Login ▾

 Recommend

 Share

Sort by Best ▾

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name



nice writeup! Especially the part with cached covers.

I'm a bit curious on your thoughts for handling other dynamic assets.

If one want to store other types of data (video, articles or other format), is this the go-to(service worker) way to handle it?

I'm wondering if service worker will cache it if no records have been found, and update it if things changed. If it's not the case, do I have to build a function that fetch data and compare the hash to see if it's changed?

Thx for the nice article!

1 ^ | v • Reply • Share ›



Harkirat Singh • 4 months ago

Hey, its a nice post but i ran through an issue and i can't find its solution

everything that you did works fine but when i add nested routes for example /home/dashboard and refresh the page on that url in my console i get -

Error during service worker registration: DOMException: Failed to register a ServiceWorker: The script has an unsupported MIME type ('text/html')

it is happening in nested routes only...
can you help me with this..

^ | v • Reply • Share ›

ALSO ON PUSHER BLOG

Build a REST API with Laravel API resources

5 comments • 2 months ago

Getting started with React Router v4

2 comments • 4 months ago