



Kévin Jean

Follow

Nov 23, 2017 · 5 min read

## A Progressive Web Application with Vue JS, Webpack & Material Design [Part 4]



A Progressive Web App  
with VueJS, Webpack &  
Material Design

Part IV  
Take a picture with  
VueJS

This article is part of a serie that aims to build a basic but complete Progressive Web Application with VueJs, Webpack & Material Design, step-by-step and from scratch. If you have not yet read it, you can find the initial part [here](#).

If you have not read them yet, you can find previous parts below:

- [Part 1] Create a Single Page Application with VueJS, Webpack and Material Design Lite (by Charles Bochet)
- [Part 2] Connect the App with a distant API with Vue-Resource and VueFire
- [Part 3] Offline experience with Service workers (by Charles Bochet)

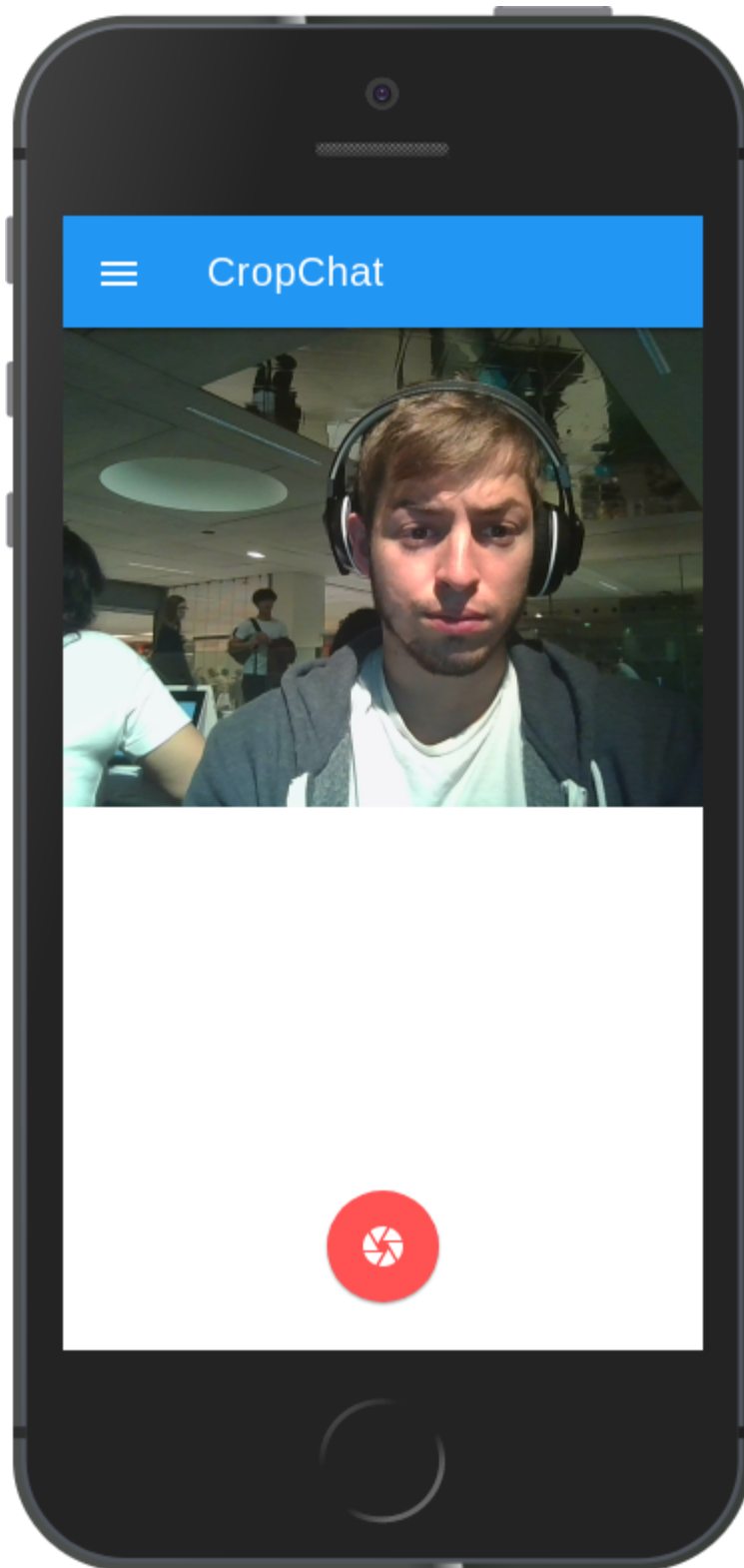
Code source is available on this [GitHub repository](#). And you can see the final result [here](#).

Here is where we stopped:



This fourth tutorial focuses on implementating a native functionality in a progressive web app. We are going to:

- Display the native camera with the [MediaDevices API](#);
- Upload pictures on firebase storage;
- Post pictures to Firebase.



Camera view (desktop)

Enjoy reading!

## [PART 4] Access device camera to take pictures

### Display the camera

Create a new empty view components/CameraView.vue:

```
1  <template>
2    <div></div>
3  </template>
4
5  <script>
6    export default {
7    }
8  </script>
```

And link the button action to this new view :

- Import the new view in the router config (router/index.js)

```
import CameraView from '@/components/CameraView'
```

Add the new view in the **routes** object (router/index.js)

```
{
  path: '/camera',
  name: 'camera',
  component: CameraView
}
```

- Add a new button with the link to the new CameraView (homeView.vue)

```
1 <template>
2   <div>
3     ...
4     <router-link class="take-picture-button mdl-button mdl-
5       <i class="material-icons">camera_alt</i>
6     </router-link>
7   </div>
8 </template>
9
10
11 <style scoped>
12   .take-picture-button {
13     position: fixed;
```

Now we can use the mediaDevices API

- First, we need an empty video tag in the cameraView that will contain the stream (CameraView.vue) :

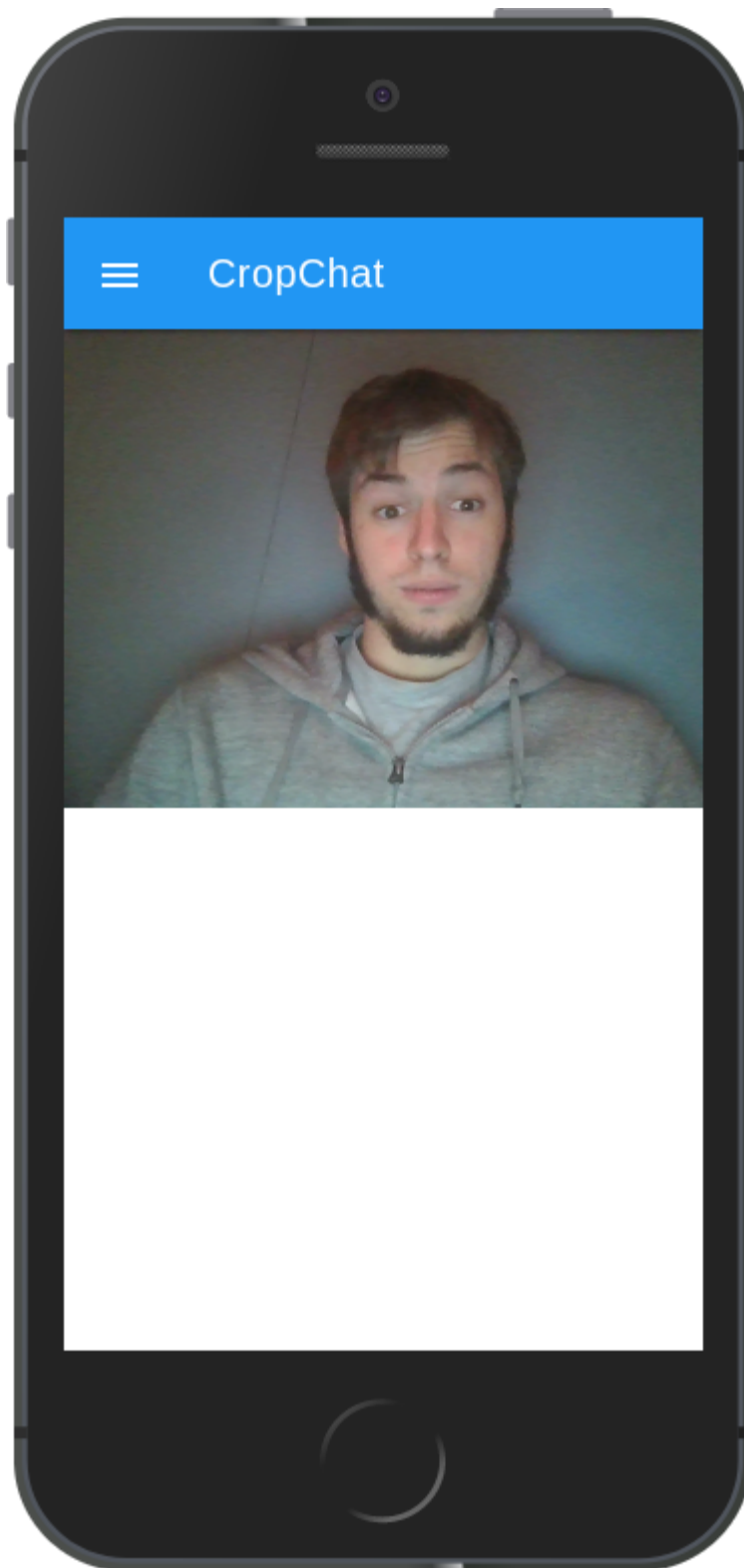
```
1 <template>
2   <div class="camera-modal">
3     <video ref="video" class="camera-stream"/>
4   </div>
5 </template>
6
7 <script>
8 </script>
9
10 <style scoped>
11   .camera-modal {
12     width: 100%;
13     height: 100%;
14     top: 0;
15     left: 0;
16     position: absolute;
```

We add a **ref** attribute to this tag to be able to dynamically attribute a video source to the video tag.

During the CameraView mount we will initilize the camera

```
1  <script>
2    export default {
3      mounted () {
4        navigator.mediaDevices.getUserMedia({ video: true })
5          .then(mediaStream => {
6            this.$refs.video.srcObject = mediaStream
7            this.$refs.video.play()
8          })
9        .catch(error => console.error('getUserMedia() error', error))
10      }
11    }
12  </script>
```

And tadaaa



A little explanation :

```
navigator.mediaDevices.getUserMedia({ video: true })
```

This method ask to the user through a prompt the permission to use his camera, then turns on a camera on the system and provides a MediaStream object containing a video/track. To more information see the API doc.

*NB : this API can be use to record audio too.*

Now we are ready to take a picture !

. . .

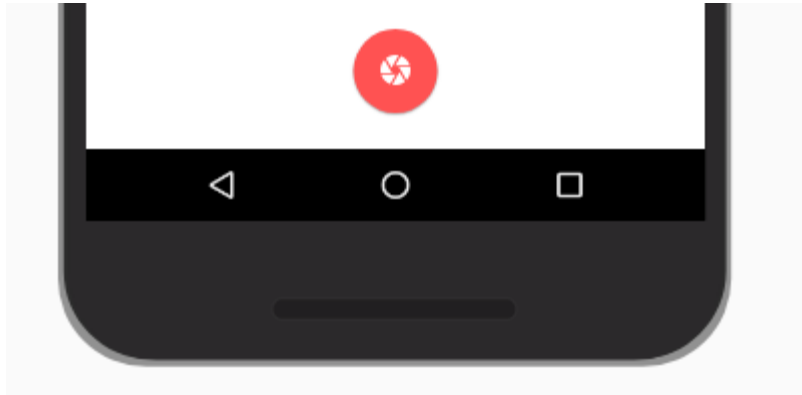
## Take a picture

Start to add a new button to take the picture , and create an empty function 'capture' binded to this button :

```
1  <template>
2    <div class="camera-modal">
3      <video ref="video" class="camera-stream"/>
4      <div class="camera-modal-container">
5        <span @click="capture" class="take-picture-but
6          <i class="material-icons">camera</i>
7        </span>
8      </div>
9    </div>
10 </template>
11
12 <script>
13 export default {
14   ...
15   methods: {
16     capture () {
17     }
18   }
19 }
20 </script>
21
22 <style scoped>
23   ...
```



Now you can see this :



It's time to capture an image in the `MediaStream` ! To make this happen, we will use one new HTML interface. The `ImageCapture` interface, which provides an interface to capture an image from a `MediaStreamTrack`. To get a `MediaStreamTrack` we must save the current `MediaStream` in our view data. This feature is only supported in Chrome 59 or above.

```
1  <script>
2    export default {
3      data () {
4        return {
5          mediaStream: null
6        }
7      },
8      mounted () {
9        navigator.mediaDevices.getUserMedia({ video: true })
10         .then((mediaStream) => {
11           this.mediaStream = mediaStream
12         })
13      }
14    }
15  </script>
```

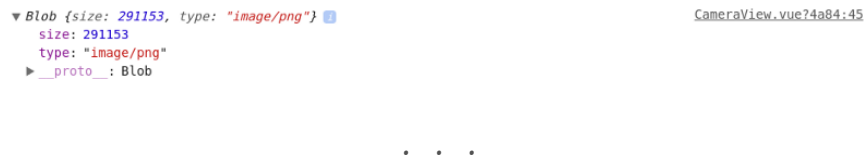
Now we can complete our capture function :

```

1  capture () {
2    const mediaStreamTrack = this.mediaStream.getVideoTracks()
3    const imageCapture = new window.ImageCapture(mediaStreamTr
4    return imageCapture.takePhoto().then(blob => {
5      console.log(blob)
6    })

```

Normally we will see a blob in your console



```

▼ Blob {size: 291153, type: "image/png"}
  size: 291153
  type: "image/png"
  __proto__: Blob
CameraView_vue74a84:45

```

## Close the camera

this part can seems to be useless, but it's primordial. First, the user don't want to keep his camera open. Second, the camera is a very greedy in battery, so we must use it with parsimony.

During the destroy lifecycle event, we get all open tracks in our stream and we stop them:

```

// CameraView.vue

export default {
  ...
  destroyed () {
    const tracks = this.mediaStream.getTracks()
    tracks.map(track => track.stop())
  }
  ...
}

```

## Upload the picture

To add the taken picture to the chat, we must store the picture on a server. In the part 2, we use Firebase as a database but Firebase also

offers a free storage service! To see the full storage documentation go [here](#).

Let's setup our firebase storage into the firebase console. Go to the firebase interface > storage tab > rules and update the rules to make the storage public. This manipulation is only good for prototyping, to secure the storage, use the [firebase auth](#).

```
service firebase.storage {  
  match /b/{bucket}/o {  
    match /{allPaths=**} {  
      allow read, write: if true;  
    }  
  }  
}
```

You remember the firebase service? We will add a new line to this service to export the storage service from firebase

```
1  firebase.initializeApp(config)  
2  const storage = firebase.storage()  
3  const database = firebase.database()  
4  
5  export {  
6    database,
```

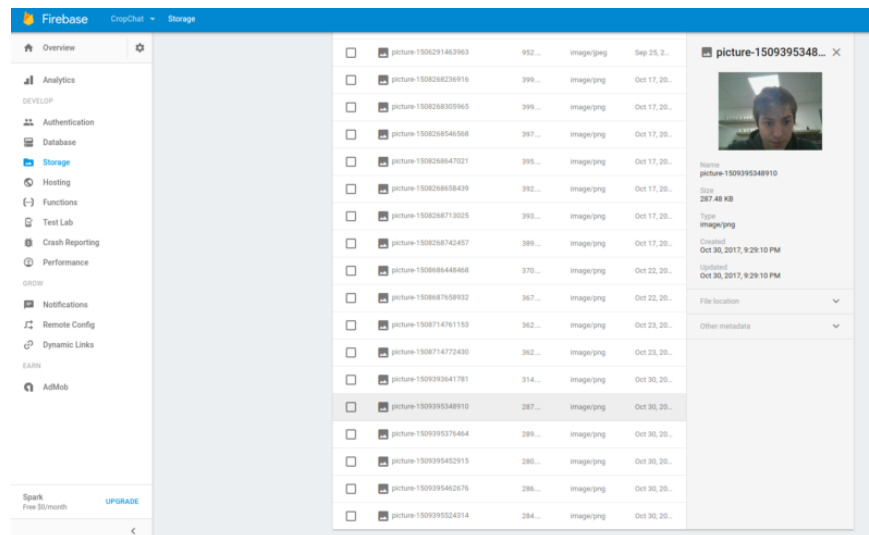
We can now import this service in our components to access our firebase storage.

```

1  <script>
2    ...
3    import { storage } from '../services/firebase'
4    methods: {
5      capture () {
6        ...
7        return imageCapture.takePhoto().then(blob => {
8          storage.ref().child(`images/picture-${new Date()}.
9            this.$router.go(-1)
10         })

```

Ok let's try to upload a picture, you should see your picture in your firebase console.



## Add the picture to the list

After the upload let's add the picture in the

Create a new mixin files to contains our postCat method written during the part 2 (mixins/postCat.js).

```
1  const postCat = {
2    methods: {
3      postCat (catUrl, title) {
4        this.$root.$firebaseRefs.cat.push(
5          {
6            'url': catUrl,
7            'comment': title,
8            'info': 'Posted by Charles on Tuesday',
9            'created_at': -1 * new Date().getTime()
10         }
11       ).then(
12         this.$router.go(-1)
```

We can update the postView with the mixins

```
1  <script>
2    import parse from 'xml-parser'
3    import postCat from '../mixins/postCat'
4
5    export default {
6      mixins: [postCat],
7      data () {
8        return {
9          'catUrl': null,
10         'title': ''
11       }
12     },
13     mounted () {
14       this.$http.get('https://thecatapi.com/api/images/get?')
15       const catUrl = parse(response.body).root.children['
```

And use the mixin in the camera view

```
1  <script>
2    import postCat from '../mixins/postCat'
3
4    export default {
5      mixins: [postCat],
6      ...
7      methods: {
8        capture () {
9          ...
10
11          return imageCapture.takePhoto().then(blob => {
12            storage.ref().child(`images/picture-${new Date()}.
13              .then(res => {
14                this.postCat(res.metadata.downloadURLs[0], 'Hel
```

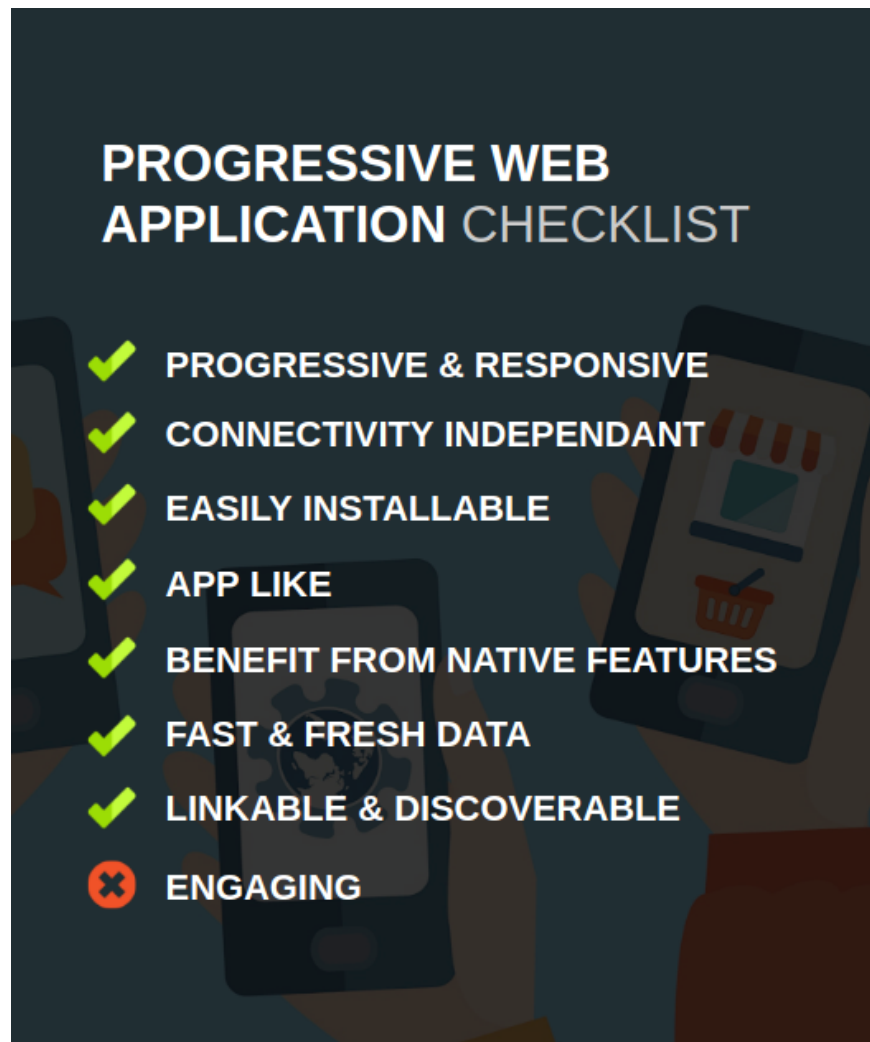
This is it ! Congratulation we can now taken and upload a picture in our list !

## Conclusions

I hope you have a better understanding of how to implement an native functionality with an HTML5 API. We learned:

- how to use `MediaStream` to access to the native camera ;
- how to store picture in Firebase with `VueFire` ;

Let's have a look to our Progressive Web App checklist:



One of our last requirements is not yet met. We will handle them in next parts. Part 5 will show you how to “use push notifications to engage your user”.





