EMU-Interface Documentation

Pierre-Antoyne Fontaine

December 18, 2017

Introduction

EMU WebApp is a web application for visualizing and correcting speech and derivated speech data. However, its GUI and backend are strongly correlated, meaning that it is complicated to use the GUI for an other tool like a TTS system. The objective of EMU Interface is to extract the GUI logic, and to implent a proof of concept showing how to use this package.

1 EMU webApp extraction

EMU WebApp is a web application running on a browser. It is written in JavaScript, using a framework called AngularJS. The application is linked with a database containing files to display an audio signal, its spectrogram, its annotations for a speech signal, etc. You can also add your own file to display its audio signal and spectrogram.

In a regular AngularJS application, the program is mainly divided into *Services*, *Directives* and *Views*. The *Services* contain datas and useful function, *Directives* add specified behaviors in the *Views*. The *Views* display data.

In the EMU WebApp, *Services* are used to manage data and providing useful functions for drawing the signals, annotations, etc. on canvas (sort of drawing area in HTML). *Directives* add views in the main area, called by the *Emuwebapp directive*. Then, most of the *Directives* implement the `$watch()` method to update the views whenever datas are updated. This method is native to AngularJS, you can find more information on the official website of AngularJS (`https://angularjs.org/`). There is also a *worker* called *SpectroDrawingWorkerClass*, which allows the application to calculate the values for the spectrogram.

The goal of the project is to extract all the *Directives* and *Services* useful for drawing the signal, its spectrogram and annotations linked with the speech data.

2 Package

My package is also divided into *Services* and *Directives*. Once you get an *audioBuffer* in the application (using the `setAudioBuffer()` method of the *bufferService*), the *Directives* update their views and display the audio signal.

The first version of the package extract the interface for displaying the audio signal and its spectrogram. For that, we needed to keep differents *Services* :

- *drawhelperservice* : provides functions for drawing in a canvas
- *mathhelperservice* : provides some maths functions
- *Wavhelperservice* : provides some functions to extract .wav files

I also kept the *SpectroDrawingWorkerClass* to help drawing the spectrogram.

Then, I added two new *Services* called *BufferService* and *PlayService*. The *BufferService* is mainly extracted from the *FileService* that existed in the EMU WebApp. However, in our case, we only need the getters and setters for the audio signal, converted into an *audioBuffer* (object containing all informations about the signal).

The *PlayService* is used to create an *audioContext* in the browser to allow it to play the audio.

I also added new *Directives*, called *Osci* and *Spectro*. Those *Directives* directly comes from the EMU WebApp, however I remove a lot of "useless" functions and parameters which were not useful for our package.

After implementing all of those components, I manage to have an application working as the *Figure 1* indicates.

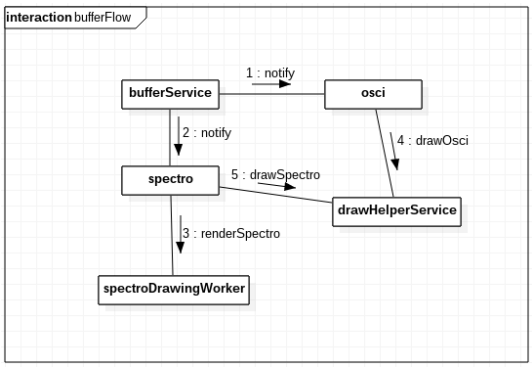

Figure 1: Audio buffer flow

This is how the application is working. First, the *Osci* and *Spectro Directives* listen to the *BufferService*. It means the when you put an audio buffer in

the application, it sends a notification to the *Osci* and *Spectro*. Then, those *directives* use the *drawHelperService* (and the *SpectroDrawingWorker* for the *Spectro*) to display the signal and the spectrogram in the canvas.

I also add a *Controls Directive* to zoom on the signal, but also to play it. It uses a *appStateService* which contains the start and the end of the signal to be displayed. It also means that the *Osci* and *Spectro Directives* listen to this *Service* too. On *Figure 2*, you can figure what *Services* are used.

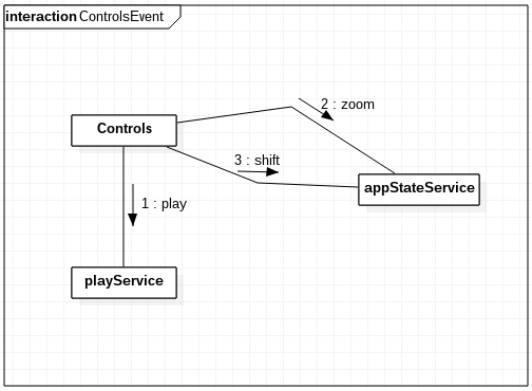

Figure 2: Events from *Controls directive*

3 Proof of concept

To install the package, you have two solutions. The first solution is using the package as a regular AngularJS application. You need to have the `npm` and `bower` package installated. Then you need to run the following commands in a terminal :

1. `npm install grunt`
2. `bower install`
3. `npm install`

Then you can run the application using `grunt serve`. Normally a page should be open on the browser. If no default browser is configured open one brower at the following url: `http://localhost:9000/`.

The second solution is using the minified code. You need to add the JS scripts in an HTML file. Then you need to add the `ng-app="EMUInterface"` to the Body tag. Then you need to add a `<div ng-view=""></div>` to add the views directly in the HTML file.

To use the package, either using minified code or not, you will need to add a new javascript script. You will then need to call the functions in the *bufferService*, using the following javascript code `angular.element(document.body).injector().get('bufferService')`, which gives you an instance of *bufferService*.

I implement the package in a small application which takes a .WAV file at the entrance of the application. The script is the following :

```
1  var startDraw = function(){
2          var bufferService = angular.element(document.body).injector()
3          .get('bufferService');
4          var browserDetector = angular.element(document.body).injector()
5          .get('browserDetector');
6          var file = document.getElementById('fileEntry').files[0];
7          var reader = new FileReader();
8          var res;
9
10         reader.readAsArrayBuffer(file);
11
12         reader.onloadend = function (evt) {
13                 if (evt.target.readyState === FileReader.DONE) {
14                         if (browserDetector.isBrowser.Firefox()) {
15                                 res = evt.target.result;
16                         } else {
17                                 res = evt.currentTarget.result;
18                         }
19                         bufferService.setAudioBufferFromArray(res);
20                 }
21         };
22 };
```

I add a file input in the HTML to call the `startDraw()` function when the user add a file. This function loads the file and add it in the *BufferService* using the `setAudioBufferFromArray()` method.

Conclusion

The next step to improve this project is to add the interface about annotations of a given speech to the package. This should work as the first part, using different functions to render the annotation on a canvas.