NickelRing

Adaptation des IHM

par Mesnier Maylanie, Forget Nicolas, Ding Feng, Saraïs Anthony

Introduction

Nous avons choisi d'implémenter un programme facilitant la lecture de notice de montage. Nous avons imaginé un utilisateur en situation de montage de meuble par exemple et nous nous sommes rendu compte que ce n'était pas toujours pratique de suivre une notice papier. En effet pour monter un meuble ou autre, en général nous avons besoin d'au moins une de nos mains libres (si ce n'est les deux) pour tenir la notice suivie, tourner des pages etc.

Le but de ce projet était de développer une application où un des paramètres du contexte d'usage variait. Nous avons alors décidé de faire varier l'environnement en rendant variable le nombre de main libre d'un utilisateur durant l'activité : Montage d'objet.

Nous avons ensuite décidé de 4 langages/framework différents pour la réalisation de ce projet afin de pouvoir comparer les avantages et inconvénients de chacun :

- Electron
- AngularJs
- Angular 2
- JavaFx

(se référer au rapport n°1 pour plus de détail).
L'ensemble des projets est récupérable sur gitHub .

Tutoriaux et techniques utilisées

ReactJS + Électron par Anthony

Adaptations intégrées : clic souris et motion detection

Installation des outils

NodeJS et npm étant obligatoire pour AngularJS, Angular 2 et React/Electron, veuillez vous reporter à l'annexe "Installer Node et Npm" avant de continuer.

À présent, il nous faut récupérer Électron, qui n'est autre qu'un module npm à installer globalement afin de pouvoir l'utiliser directement depuis un terminal. Pour cela, nous utiliserons la commande suivante :

```
$ sudo npm install -g electron
```

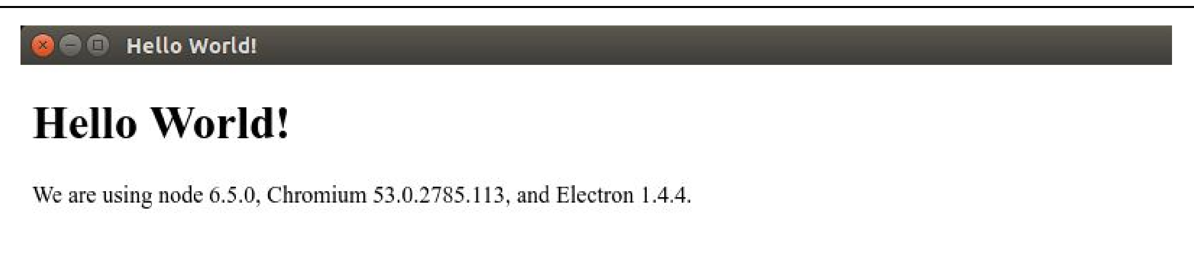
D'ailleurs, c'est quoi electron ?

Electron est un framework basé sur nodeJS permettant le développement d'applications desktop cross-os. Il utilise le moteur de rendu chromium et permet d'utiliser les technologies du web, nous pouvons citer Atom ou bien Visual Studio Code utilisant Electron, respectivement développés par Github et Microsoft.

Vérifions que nos outils sont bien installés grâce aux commandes suivantes qui lanceront un projet test :

```
$ git clone https://github.com/electron/electron-quick-start
$ cd electron-quick-start
$ npm install
$ npm start
```

Vous devriez obtenir quelque chose ressemblant à la figure suivante, les versions affichées seront dépendantes de votre installation.



Si ce n'est pas le cas, ou si vous obtenez une erreur lors de l'utilisation d'une commande, vérifiez que les étapes précédentes aient bien été appliquées.

Préparation de l'architecture

Les projets javascripts étant souvent laborieux à mettre en place (beaucoup de fichiers, de configuration, de dépendances ...), il n'est pas rare d'utiliser ce qu'on appelle : un boilerplate. Un boilerplate est en quelque sorte une normalisation d'architecture qui permet de garder les mêmes normes quels que soient le projet.

Nous avons choisi d'utiliser le boilerplate electron-react-boilerplate que nous pouvons trouver à l'adresse : https://github.com/chentsulin/electron-react-boilerplate.git.

Son installation est simple :

```
$ git clone https://github.com/chentsulin/electron-react-boilerplate.git
$ cd electron-react-boilerplate
$ npm install
```

Cette dernière commande permet d'installer toutes les dépendances définies par le fichier package.json à la racine du projet. Nous remarquons qu'il y a un grand nombre de dépendances, nous n'en utiliserons qu'une partie pour ce projet.

Dans notre cas, nous avons supprimé ce qui n'était pas nécessaire, cela amenant à des modifications de code. Afin d'éviter de provoquer des erreurs, nous allons, dans ce tutoriel, laisser le code tel quel, c'est à dire fonctionnel.

Parlons des dépendances importantes ici :

- Babel: c'est un transpiler (traducteur) qui permet d'utiliser JSX (du HTML dans du Javascript) et ES6 (qui est la dernière version de Javascript).

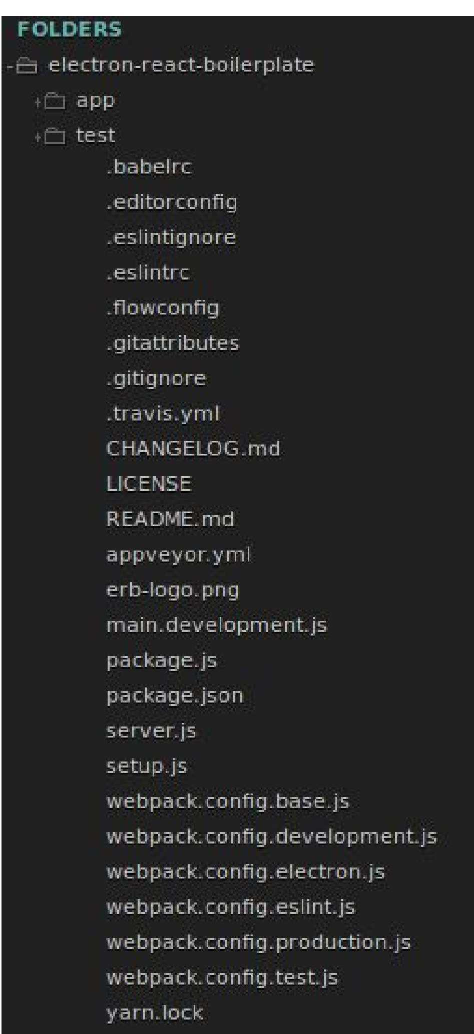
- Électron: il est nécessaire d'avoir électron en local pour certaines autres dépendances.

- Express: express est l'un des tout premier framework nodeJS, il fournit des fonctionnalités dans le domaine du routing essentiellement. Ici, il est nécessaire pour amorcer Électron.

- Webpack: l'une des révélation de l'année 2016 pour les front-end. Ce framework permet l'importation des modules dans le même format que nodeJS (cad: var maVariable = require("monModule"), ce qui évite de devoir charger toutes les dépendances dans le fichier html principal comme nous avions l'habitude de faire. Il apporte également un lot de fonctionnalité

d’automatisation, ce qui nous permet d’éviter l’utilisation de gulp (gestionnaire de tâches).

- React: logique.



Voyons de quoi est composée notre architecture.

Un dossier app qui va contenir toutes nos sources.

Un dossier test, utilisant le module karma, que nous ne verrons pas ici.

Des fichiers .* servant à configurer nos modules.

Un fichier server.js qui est la source de notre application, son travail est d’amorcer le back-end et l’interface.

Des fichiers webpack.config.* permettant de configurer webpack, les tâches que nous souhaitons mettre en place, les loaders, les presets, les plugins etc …

Notre premier composant react

Avant d’entrer dans le vif du sujet, supprimons les fichiers inutiles dans le dossier app. Vous pouvez tout supprimer sauf index.js et app.html.

A présent, créons un dossier ‘components’ dans ‘apps’ qui contiendra nos composant, puis à l’intérieur un dossier ‘Home’ dans lequel nous ajoutons un fichier ‘Home.js’. Afin de pouvoir utiliser, React, nous devons l’importer.

```
import React, { Component } from 'react';
```

Maintenant, créons notre composant

```
export default React.createClass({   })
```

Tout ce qui est situé à l'intérieur de ces accolades fait partie de notre composant, un composant react de base est composé de 3 fonctions (optionnelles):

```
getInitialState(){
},

componentDidMount(){
},

render(){
}
```

Qui fait quoi ?

React est basé sur un système d'état. Chaque composant aura le sien et la modification de celui ci engendrera un redraw du composant. Dans le `getInitialState()` nous retournerons l'état initial (c'est à dire l'état à la création) de notre composant. `componentDidMount()` est exécuté après la création du composant, la plupart du temps nous y injectons les parties de code faisant des appels au serveur afin d'initialiser nos variables. Pour finir, la méthode `render()`, plutôt explicite, qui renvoie le code HTML de notre composant.

Maintenant que nous avons vu comment fonctionnait un composant de base de react, remplissons le !

```
getInitialState(){
	return {
		currentPage: "Home"
	}
},

render(){
	return (
		<div className='home'>
			this page is the {this.state.currentPage}
		</div>
	)
}
```

Sans ouvrir une quelconque interface, il est facile de deviner que ce code affichera à l'écran : "This page is the Home"
D'ailleurs, nous ne pouvons toujours pas afficher ce composant. Pour cela nous devons l'appeler quelque part, c'est là qu'intervient le fichier index.js de tout à l'heure.

Supprimons son contenu et collons y les lignes suivantes

```
import React from 'react';
import { render } from 'react-dom';

import Home from "./components/Home/Home";

render(<Home/>, document.getElementById('root'));
```

Maintenant, si nous lançons la commande `$ npm run dev` nous devrions obtenir le résultat suivant:

Bravo ! Vous avez votre premier composant.
Voyons à présent comment implémenter une version basique (et simulée) du slider. Nous allons simuler l'existence des images et les remplacer par quelque chose ressemblant à ceci :

Ceci est l'image 1

Créons un composant Slider (Donc, un dossier Slider dans le dossier components, puis un fichier Slider.js), insérons le code suivant à l'intérieur:

```
import React, { Component } from 'react';


export default React.createClass({
  getInitialState(){
    return {
      currentSlide: 1
    }
  },

  _next(){
    console.log("next");
    this.setState({
      currentSlide: this.state.currentSlide + 1
    })
  },
```

```
  _prev(){
      currentSlide: this.state.currentSlide - 1
  },

  render(){
    return (
      <div className="slider">
              <span  onClick={this._prev}  style={{display:  "inline-block"}}>  prev
</span>
              <div  style={{border:  "solid  black  1px",  margin:  "50px",  display:
"inline-block"}}>Ceci est l'image {this.state.currentSlide}</div>
              <span  onClick={this._next}  style={{display:  "inline-block"}}>  next
</span>
      </div>
    )
  }
})
```

Regardons le `render()` de plus près, les nouvelles notions que nous pouvons voir sont :

- l'utilisation d'accolade pour délimiter une zone de script js à l'intérieur du html (`{this._prev}`)
- l'utilisation d'une balise style contenant un objet JSON (`style={{display: "inline-block"}}`)
- l'utilisation d'une propriété l'état de notre composant (`this.state.currentSlide`)

Qu'est ce qu'il se passe ?

Lorsque nous allons cliquer sur 'prev' ou 'next', cela va appeler la fonction associée qui va incrémenter ou décrémenter la variable currentSlide dans notre état. Or, quand l'état change, le composant est redessiné, par conséquent celui ci va se mettre à jour et afficher la bonne valeur.

Oui mais actuellement ça ne fonctionne pas !

En effet il faut maintenant appeler notre nouveau composant dans un autre qui est déjà rendu par React (cf: `render(<Home/>, document.getElementById('root'));`).
Nous allons donc dans le fichier Home.js, et y ajoutons cette ligne en haut :

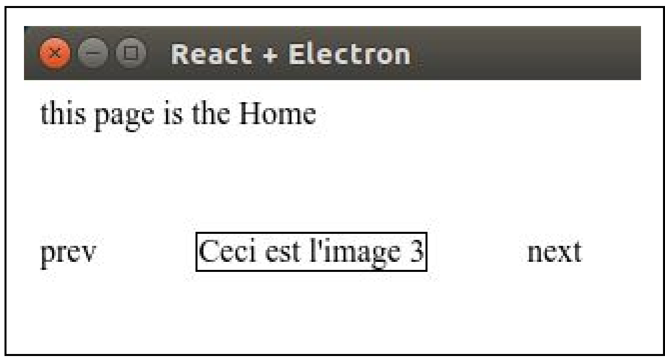
```
import Slider from "../Slider/Slider";
```

Puis dans le `render()`, appliquons notre nouveau composant :

```
<div className="home">
 this page is the {this.state.currentPage}

 <Slider/>
```

```
</div>
```

Une fois fait, si vous lancez le projet (`$ npm run dev`), vous devriez obtenir le résultat suivant :



Bon, d'accord c'est moche, pas ergonomique et pas intuitif, mais ça marche ! Lorsque nous cliquons sur prev et next, cela change bien notre page. Il ne reste plus qu'à imaginer que nous remplacions quelques lignes dans le `render()` du slider pour importer une image, par exemple :

```
<img src={require("../../assets/page" + this.state.currentPage + ".png")}
alt=""/>
```

Ce tuto est fini, la gestion des mouvements était tellement fastidieuse et compliquée que l'explication dans ce tutoriel prendrais 50 pages. Il suffit de savoir que notre Slider possède, dans sa fonction `componentDidMount()`, le code nécessaire au déclenchement d'événements lorsqu'il y a du mouvement devant la caméra, et qu'après traitement, ce code appelle la fonction `_next()` ou `_prev()`.

Petit récapitulatif

- Électron est basé sur nodeJS et Chromium et permet de développer des applications desktop avec les technologies du web
- React est basé sur des composants possédant un état. La modification de cet état entraîne un redraw du composant (avec une utilisation plus avancé, des fonctions permettent d'éviter ce redraw, notamment `shouldComponentUpdate()`)
- Les composants React sont composés (très souvent) d'au minimum 3 fonctions : `getInitialState()`, `componentDidMount()` et `render()`.

Avec quel outil de développement, de tests ?

Nous avons utilisé Webstorm 11.0.3 pour le développement.
Le projet a été testé sur Ubuntu 16.04 et Windows 10.
N'ayant pas de mac, nous n'avons pas pu tester le projet sur cette plateforme.
Au niveau des frameworks/modules, nodeJS 6.2.2 et npm 3.9.5

Comment déploie-t-on et exécute-t-on l'exemple ?

```
git clone https://github.com/maylme/NickelRing.git
cd NickelRing/Electron
npm install
npm start
```

Comment teste-t-on les capacités d'adaptation

Une fois que le projet est lancé, le slider apparaît et est par défaut sur la page 1. La fonction motion détection est désactivée par défaut, nous pouvons l'activer avec le bouton toggle sous le slider.

La détection est capricieuse, le mieux est de se placer en face de l'ordinateur et de faire un geste précis, avec la main de préférence. Le geste peut être rapide, du moment que la caméra arrive à suivre (la détection est calibrée à 30 images par seconde, mais la caméra pas forcément).

Il est également possible de switcher entre les slides grâce aux chevrons situés à côté.

Difficultés rencontrées

Un projet sans difficultés, ce n'est pas intéressant, alors voici une liste pas très exhaustive des problèmes rencontrés.

Commençons par ce qui ne fonctionne pas : **les commandes vocales.** Comme nous l'avons vu plus haut, Électron est basé sur le moteur Chromium (qui est développé par google). Or, le Big browser à découvert des failles de sécurités basé sur l'api speech web (qui permet donc d'utiliser son micro), cette API n'étant pas faite pour être utilisée en locale. La solution de Google à tout simplement été de supprimer la fonctionnalité comme nous pouvons le voir sur le magnifique screenshot ci-après:

UPDATE 23-JUN-2016 : Google speech recognition has been disabled for Electron, therefore this functionality is not available anymore. Read more about the issue here.

Par conséquent, nous perdons une fonctionnalité majeur de notre application. Malgré divers tests et tentatives de contournements, je n'ai pas réussi à intégrer la reconnaissance vocale.

De plus, foundation ne semble pas fonctionner avec Electron, ce qui peut être dû à la manière dont Webpack nous oblige à importer les fichier css (qui n'est pas pas "conventionnel" pour Electron)

Passons maintenant à ce qui fonctionne :

- Electron + React: Déjà, sujet qui fâche. Le problème principal ne vient pas forcément de ces 2 technologies utilisées ensemble, mais plutôt de l'écosystème derrière. Pour utiliser React dans de bonnes conditions, on utilise JSX, donc Babel, donc Gulp ou Webpack. De plus, Electron utilise NodeJS, puis Express, pour amorcer le code. Le mélange de tous ces frameworks, et surtout Webpack, a été un réel problème. De nombreux tricks doivent être mis en place pour contourner le problème puisqu'il n'existe pas de solution. Je n'ai d'ailleurs pas réussi à faire fonctionner SASS avec Webpack sur Électron (ce qui n'est pas un problème en version web)

- La navigation par geste: fonctionnelle uniquement dans de bonnes conditions. En effet la mise en place de la motion detection a été fastidieuse. Du point de vue algorithmique, il a fallu dans un premier temps trouver l'algorithme qui permet de détecter la direction du geste tout en ayant une marge d'erreur correct afin que la mains soit visible même en cas de luminosité atténuée. Ensuite, l'implémentation… C'est probablement une des pire fonctionnalités que j'ai eue à intégrer dans un projet javascript. Aucune librairie n'était installable via npm (ce qui est pourtant un standard). Il fallait obligatoirement télécharger les sources. Sachant que j'utilise webpack et ES6, les librairies créant des variables globales que l'on utilise dans la suite du code sont pas compatibles. J'ai donc dû modifier ces libs afin qu'elles le soient. Si on oublie que le code de la librairie était probablement développé par un allergique à la normalisation (pas de points virgules, pas de 'var' avant la déclaration

des variable, des duplications de code … et j'en passe), la librairie fonctionne très bien. Son intégration dans Electron fût plutôt facile une fois ces problèmes résolus.

Conclusion personnelle et améliorations

React est un très bon framework orienté composant, nous avons pu voir dans l'exemple qu'il est facile à mettre en place et n'est pas très codovore (de la famille des mangeurs de code). Les benchmarks plaident en sa faveur, sa fluidité et sa rapidité en font un framework de haut niveau.

Électron de son côté semble plus difficile à prendre en main, bien que ce ne soit pas insurmontable, il faut une multitude de packages pour un projet simple. De plus, de nombreux modules ne sont pas compatibles.

Pour ma part, je ne pense pas réutiliser Électron à l'avenir, il est bien trop compliqué à utiliser et mettre en place pour qu'il vaille le coup. Malgré ma réticence à utiliser Java, je préfèrerais utiliser JavaFX qui est plus stable.

Pour ce qui est des améliorations, nous avons plusieurs possibilités :

- Amélioration de l'interface graphique (actuellement, c'est très fade).
- Ajout d'un panel d'administration (sidenav ?) pour afin d'y insérer de nouveaux tutoriaux et d'afficher la liste des tutoriaux disponibles.
- Trouver un hack pour le responsive (JS ?) puisque les media query ne fonctionnent pas sous Electron

AngularJS x Material par Nicolas

Adaptations intégrées : clic souris et détection vocal

Architecture

Nous avons choisi de développer une application de visualisation de notice de montage (ou autre documents) dans différents langages et avec différentes interactions pour pouvoir faire varier l'utilisateur. Dans cette partie nous allons voir ensemble le développement du projet en AngularJs (à ne pas confondre avec le 2). Le projet est regroupé dans un dossier « app » et se structure de la manière suivante :

```
app
  images
    nikelRing
  scripts
    controllers
      mainCtrl.js
    app.js
  styles
    mainStyle.css
    style.css
  views
    mainView.html
  index.html
```

s

Le dossier images regroupe évidemment les pages de notre notices, mais peut aussi contenir d'autres images utilisées dans l'application.

Le dossier script contient les contrôleurs de notre application, car elle est développée selon le modèle MVC, ces derniers sont au format JavaScript. On note également la présence d'un fichier app.js qui permet d'effectuer les injections de dépendances des librairies dans le projet, mais nous verrons cela.

Les styles sont les fichiers CSS qui permettent d'appliquer des modifications de forme, de couleur, et autre sur les éléments HTML. Ici nous avons un style pour la vue principale de l'application et un style plus général qui s'applique à toute l'application avec des règles pour l'adaptation à la taille de l'écran, etc.

Les vues du dossier « views » sont des bouts de code HTML qui viendront se placer dans le contenu de l'index, ainsi il n'y a qu'une partie de la page qui est rechargée, c'est ce qu'on appelle une application « main page ».

Enfin le fichier index, il contient évidemment les balises connues tel que html, head, dans lequel on trouve le titre de l'application, des balises de lien vers les feuilles de style du projet, ou d'autres (Angular, Font Awesome, Materials…) et des balises de scripts qui elles aussi renvoient à des fichiers hébergés en ligne et à nos contrôleurs. Dans le body par contre on ne trouve qu'une balise de contenu dans laquelle il y a notre vue principale (le code html du fichier vue).

Ceci est la structure la plus minimaliste pour un projet AngularJs, qui reste quand même proprement découpée. Toutes les librairies utilisées dans ce projet sont importées via les balises scripts contenant des liens dits CDN et ne sont pas prisent en charge par des gestionnaires tel que bower ou npm. Le but ici est de voir si en plus des différences

potentielles entre Frameworks, entre librairies de style ou autre, il existe des différences notables entre plusieurs types d'importations de librairies.

Code

Le code est découpé en trois parties principale ; le HTML qui définit la structure des pages, le CSS qui définit le style et les contrôleurs JavaScript qui définissent le fonctionnement des pages par des variables et des fonctions. Dans l'image suivante on voit la structure de la page principale de l'application :

```html
<div class="mainView" layout="column" layout-align="space-around center">

    <md-content layout="row">
        <md-button class="navbtn md-primary md-raised" aria-label="Previous" ng-click="prevPage()" ng-disabled="currentPage==1">
            <i class="fa fa-chevron-left"/>
        </md-button>
        <img src="/images/nikelRing/page{{currentPage}}.png"/>
        <md-button class="navbtn md-primary md-raised" aria-label="Next" ng-click="nextPage()" ng-disabled="currentPage==maxImage">
            <i class="fa fa-chevron-right"/>
        </md-button>

    </md-content>

    <md-content layout="row">
        <div class="pagination">{{currentPage}} / {{maxImage}}</div>
    </md-content>

    <md-content layout="row" id="optionLayout" layout-align="space-between center">

        <md-content class="pagination">"Previous"</md-content>

        <md-content id="toggleBtn">
            <md-button class="md-fab md-primary" aria-label="Mic" ng-click="toggleMic()" ng-show="micBtn">
                <i class="fa fa-microphone" aria-hidden="true"></i>
            </md-button>
            <md-button class="md-fab md-warn" aria-label="Mic" ng-click="toggleMic()" ng-show="!micBtn">
                <i class="fa fa-microphone-slash" aria-hidden="true"></i>
            </md-button>

            <md-button class="md-fab md-primary" aria-label="Cam" ng-click="toggleCam()" ng-show="camBtn">
                <i class="fa fa-eye" aria-hidden="true"></i>
            </md-button>
            <md-button class="md-fab md-warn" aria-label="Cam" ng-click="toggleCam()" ng-show="!camBtn">
                <i class="fa fa-eye-slash" aria-hidden="true"></i>
            </md-button>
        </md-content>

        <md-content class="pagination">"Next"</md-content>

    </md-content>

</div>
```

La vue est découpé en trois parties regroupées dans une div principale, la première partie affiche l'image de la notice et les flèches de navigations, la seconde concerne uniquement la pagination et la dernière contient les boutons pour activer/désactiver le micro et la caméra, ainsi que les mots clé à utiliser pour naviguer avec la reconnaissance vocale. Dans la balise principale on utilise les attributs « layout » et « layout-align » d'Angular Material pour définir la position visuelle des éléments. Les bouton de navigation possèdent également des classes de style prédéfini et des directives pour l'action du clic et pour désactiver le bouton si on arrive à la première ou la dernière page. Plus bas on remarque également une directive pour afficher ou cacher entièrement un élément html, dans notre cas il sert à changer l'apparence d'un bouton. Une autre manière de faire cela aurait été de modifier dynamiquement seulement le nom d'une classe de style, pour cela on utilise des variables définies dans le contrôleur et accessibles depuis la vue grâce aux doubles accolades {{nomDeVariable}}. Cette variable est modifiable dans le contrôleur, comme celle de la page courante qui sera incrémentée et décrémentée en fonction des actions effectués par l'utilisateur. C'est variables sont interprétées autant en contenu de balise que dans une

chaine de caractère pour une classe ou pour une source. Il n'y a que dans les directives qui attendent simplement des expressions qu'il ne faut pas mettre d'accolades.

Certaines directives sont comprises dans Anuglar, pour en utiliser d'autres il faut ajouter les liens correspondant dans l'index.html et injecter la dépendance dans le fichier app.js comme suit pour Material par exemple :

```
1   window.app = angular.module("smartNotice", ["ngMaterial", "ngAnimate"]);
```

La partie contrôleur est structurée de la manière suivante :

```
app.controller("mainCtrl", ["$rootScope", "$scope", (mainCtrl)])
    .directive("mainView", (mainView));

function mainCtrl($rootScope, $scope){ ... }

function mainView(){
    return {
        templateUrl: "/views/mainView.html",
        controller: "mainCtrl"
    }
}
```

La première ligne injecte dans l'application le contrôleur et la vue en tant que directive pour pouvoir être utilisés depuis l'index.html ou d'autres fichiers. La vue est définie par un chemin vers le fichier HTML correspondant et le contrôleur associé, alors que la fonction du contrôleur contient toute les variables et fonctions du scope. C'est ce « $scope » qui permet d'accéder à ces dernière depuis la vue. Sur la page suivante vous trouverez le contenu détaillé de la fonction contrôleur. Il y a une valeur pour la page courante, une valeur pour le maximum de page d'une notice (à renseigner à la main pour l'instant) et deux booléens qui donnent l'état du micro et de la caméra. Annyang est une librairie importée dans l'index et utilisée ici pour faire de la reconnaissance vocale, il suffit de définir des commandes et d'activer ou désactiver le micro. Sources : https://github.com/TalAter/annyang .

```
$scope.currentPage = 1;
$scope.maxImage = 4;
$scope.camBtn = false;
$scope.micBtn = false;

$scope.prevPage = function(){
    console.log("previous");
    if($scope.currentPage > 1){
        $scope.currentPage--;
    }
}

$scope.nextPage = function(){
    console.log("next");
    if($scope.currentPage < $scope.maxImage){
        $scope.currentPage++;
    }
}

$scope.toggleMic = function(){
    $scope.micBtn = !$scope.micBtn;
    //window.alert("the audio interaction is not yet implemented");
    if (annyang) {
        if($scope.micBtn){
            var commands = {
                'back': function() { $scope.$apply($scope.prevPage()); },
                'next': function() { $scope.$apply($scope.nextPage()); },
                'help': function() { alert("say 'next' or 'previous' to navigate"); }
            };
            annyang.addCommands(commands);
            annyang.start();
        } else {
            annyang.removeCommands();
            annyang.abort();
        }

    }
}

$scope.toggleCam = function(){
    //$scope.camBtn = !$scope.camBtn;
    window.alert("the video interaction is not yet implemented");
}
```

Outils

Pour développer ce projet, j'ai choisi le logiciel Brackets, qui est un éditeur léger, autant par son fonctionnement rapide que par son design épuré. Grâce à un large choix de plug-in disponible directement depuis l'application dans le « Gestionnaire d'extension » chacun peu le personnaliser les raccourcis, l'indentation, et beaucoup d'autres fonctions bien plus poussées. Rassurez-vous tout de suite, dans le cadre de ce projet, aucune extension ne sera nécessaire, vous pouvez même importer le projet dans votre IDE favori et le lancer, mais dans la suite de ce document je vais vous expliquer comment procéder avec Brackets.

Voici donc à quoi ressemble l'interface principale de ce logiciel :

1. Menu contextuel
2. Fichier ouvert (onglets)
3. Arborescence global du projet
4. Zone principale d'édition du fichier
5. Informations sur le fichier courant (notamment erreur et avertissements)
6. Aperçu du projet et Gestionnaire d'extension

Pour importer le projet il suffit de faire Fichier > Ouvrir, puis sélectionner le dossier app du projet, là où vous l'avez sauvegardé. Vous pouvez aussi faire un clic droit sur le dossier dans votre explorateur de fichier et choisir dans le menu « Ouvrir en tant que projet Brackets » ce qui lancera directement le logiciel avec le projet. Une fois que le projet est

ouvert dans l'éditeur vous pouvez naviguer dans l'architecture présente à gauche et visualiser les différents fichiers de code.

Déploiement et Tests

Dans l'éditeur il suffit d'ouvrir l'index.html et de cliquer sur le petit éclair (6) pour lancer le projet dans la fenêtre d'un navigateur. Le projet est lancé sur un serveur émulé par l'IDE et s'exécute dans une fenêtre de Google Chrome, avec tous les outils de développement disponible. Dans le cadre d'une mise en production il suffirait de placer le dossier du projet dans le dossier d'un serveur Apach par exemple.

Une fois que le projet ne contient plus d'erreur de compilation et qu'il est déployé localement il devient possible de le tester. Pour cela il faut dans un premier temps regarder l'interface pour être sûr que tous les composants graphiques sont affichés correctement. Dans l'image ci-dessous tout semble correcte.

Ensuite on peut jeter un œil à la console du navigateur pour vérifier qu'il n'y a pas eu de problème d'importation de librairies, d'initialisation de variables ou autre. Dans l'exemple suivant le programme n'a pas été capable de trouver une ressource, pour corriger cela il suffit de l'ajouter dans le dossier correspondant, ici images.

Si tout semble correct on commence à effectuer des actions attendues (cliquer sur suivant, activer le micro et dire "previous") pour s'assurer que tout ce qu'on veut fonctionne. Puis il faut effectuer une série de test d'imprévu (cliquer sur un bouton grisé, parler lorsque le bouton du micro n'est pas activé, cliquer à répétition sur les boutons …) pour être certain qu'il n'y a pas de bug caché. Enfin à chaque ajout/modification de code il faut effectuer des tests de non régression pour confirmer par exemple que les flèches marchent toujours après l'ajout du micro, ou après un changement de style, etc. A la première activation du micro le navigateur va demander à l'utilisateur l'autorisation, il faut évidemment accepter pour pouvoir tester cette fonctionnalité, puis le navigateur affichera une point rouge dans l'onglet qui à accès au micro.

Toute les interactions sont directement testables individuellement en restant assis devant l'ordinateur étant donné qu'elles consistent à cliquer avec la souris, parler au micro ou faire des gestes devant la webcam. En revanche pour tester la capacité d'adaptation des interactions, c'est-à-dire la variation de la distance entre l'ordinateur et l'utilisateur, ou le nombre de mains disponibles de ce dernier il est préférable de se lever, de se reculer et d'effectuer les commandes à la voie et avec les gestes, que ce soit avec les mains libres ou occupé.

On se rend compte avec ce genre de test que l'interaction vocale peut-être rendu plus difficile compte tenu de la distance, et que la reconnaissance de mouvement pourrait être biaisée avec les mouvements de la personne en action. De plus avec notre interface actuelle l'utilisateur n'a aucun retour pour savoir s'il est bien placé face à la caméra ou si sa voie est bien perçue par le micro. Toutes ces remarques sont autant de pistes à améliorer pour rendre ce produit plus facilement utilisable.

Problèmes rencontrés

Le premier problème que j'ai rencontré a été de commencer le projet, en effet j'ai déjà effectué il y a plus d'un an un projet AngularJs en entreprise, mais depuis ce projet j'ai beaucoup travaillé avec d'autres Frameworks et il est très frustrant de revenir sur un environnement en ayant perdu l'habitude. J'ai donc ressorti mes documents d'entreprise et j'ai rapidement pu créer un premier projet minimum viable.

Une autre difficulté est d’avoir une application qui puisse s’adapter à différentes tailles d’écrans, mais comme nous n’avons pas choisi d’étudier la variation du support, j’ai choisi d’utiliser un fichier de style type qui permettra de s’adapter dans la plupart des cas et j’ai concentré mes efforts sur la fonctionnalité des interactions.

Pour ce qui est des interactions justement, l’intégration de annyang s’est très bien passée, avec deux copier/coller le « hello world » était fonctionnel. Il a fallu expérimenter avec de nouvelles commandes et la navigation en reconnaissance vocale était opérationnelle. À contrario je n’ai pas réussi aujourd’hui à intégrer de librairie capable d’effectuer la reconnaissance de mouvement directement ou avec des modifications moindres. Une possibilité serait de s’inspirer du code de mon collègue Anthony qui a choisi de développer le code de la détection lui-même. Exemple des librairies étudié et/ou testé :

https://github.com/mtschirs/js-objectdetect/
https://github.com/foo123/HAAR.js
https://github.com/jcmellado/js-handtracking

Angular2 x Bootstrap par Maylanie

Adaptations intégrées : clic souris/ touch pour la navigation directe ainsi que les commandes vocales

Rapide introduction à Angular2:

Age: moins d'un an.

Intérêts: Angular2 est orienté WebComponent. C'est à dire qu'une page web sera décomposée en composant autonomes capables d'être utilisés dans d'autre projets facilement.

Tutoriels suivis : n'ayant jamais fait d'Angular2 auparavant, j'ai commencé par le Quickstart fourni par Angular2 pour commencer un projet et j'ai enchainé sur le tutoriel Tour of Heroes fournit aussi par Angular2 qui permet d'acquérir les bases de développement d'une application Angular2.

Tutoriels futur? J'aimerai dans le futur suivre les autres tutoriels disponibles sur le site d'Angular2 afin d'approfondir les bases acquises. (Advanced)

Rapide introduction à Bootstrap:

Age: Première version sortie en 2011.

Intérêts: Bootstrap est un framework constitué de code CSS et Javascript permettant de construire des IHMs rapidement à moindre effort et faciliter la mise en place d'un responsive mobile par exemple.

Tutoriels suivis: Je n'ai suivi aucun tutoriel en particulier pour ce projet car je connaissais déjà Bootstrap. Néanmoins, quelques coups d'oeil à la documentation de Bootstrap a été utile.

Tutoriel de lancement du projet (pour Linux/Ubuntu):

Pour l'IDE, j'ai personnellement utilisé Sublime-text mais n'importe quel traitement de texte peut faire l'affaire. Les plus téméraires peuvent utiliser Vim.

Voici les étapes à suivre pour récupérer le projet Angular2 et l'exécuter sur votre machine.

Attention, pour ce tutoriel, vous devez avoir node, npm (et bower) d'installé sur votre machine. Si ce n'est pas le cas, veuillez vous référez à l'annexe de ce document à la partie "installer node et npm" et/ou "installer bower". De plus vous devez avoir des bases de manipulation console sous linux. La version node utilisé pour ce projet est la v4.6.0 et npm : 3.10.8

Commencez par cloner le répertoire gitHub du projet à l'endroit où vous souhaitez.

```
$ git clone https://github.com/maylme/NickelRing.git
```

Allez dans le répertoire du projet Angular2:

```
$ cd NickelRing/Angular2
```

Installez le projet:

```
$ npm install
```

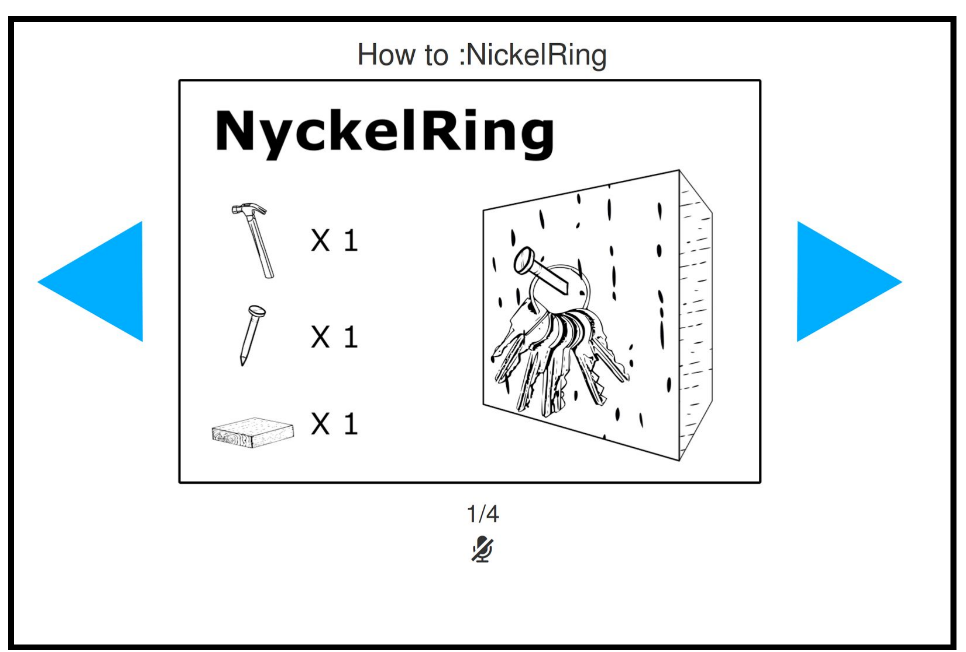
```
$ bower install
```

Une fois que tout est installé, vous pouvez lancer le projet:

```
$ npm start
```

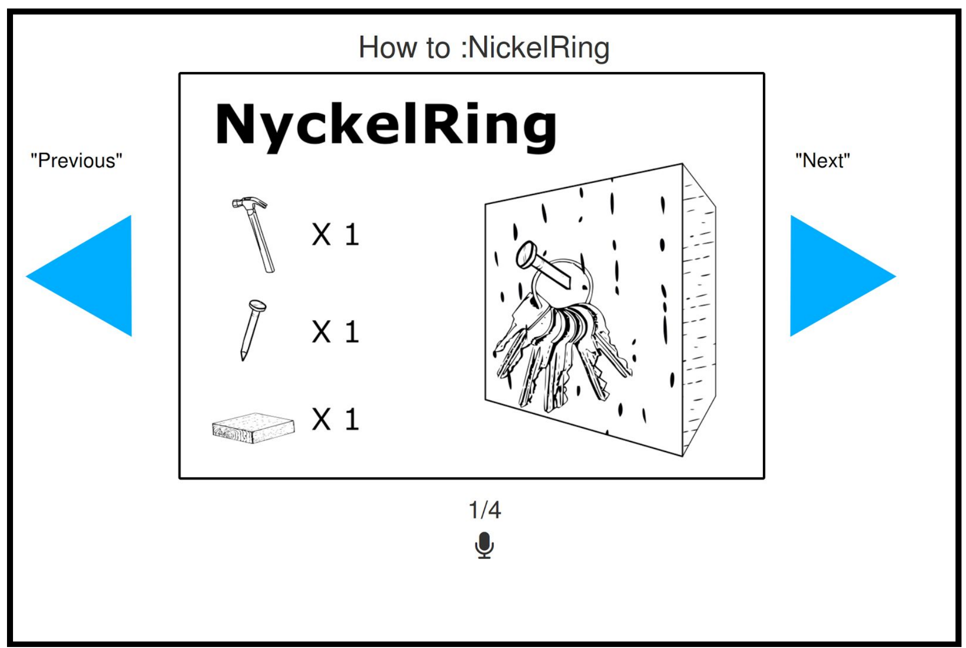
Node ouvre alors une fenêtre dans votre navigateur préféré et y lance la solution. Je recommande d'utiliser Chrome car chez moi, firefox sous ubuntu ne permet pas d'utiliser les commandes vocales.

On obtient alors l'interface suivante:



Pour passer les diapositives, il faut cliquer (ou toucher) les flèches bleues. En touchant la flèche de droite, on passe à la diapositive suivante. En touchant la flèche de gauche, on passe à la diapositive précédente.

Pour activer la reconnaissance vocale, on clique sur le micro. L’interface se modifie comme ceci:



Si le navigateur le demande, il faudra autoriser l’application à utiliser le micro. Ensuite pour passer à la diapositive précédente il faut dire “previous”. Pour passer à la diapositive suivante, “next”.

Explication de la structure:

Le projet se découpe comme le décrit le schéma suivant:

```
▼ Angular2
  ▼ app
      app.component.style.css
      app.component.template.html
      app.component.ts
      app.module.ts
      main.ts
      mock-NickelRing.ts
      slide.service.ts
      slide.ts
  ▼ assets
    ▶ NyckelRing
      after.svg
      before.svg
  ▶ bower_components
  ▶ node_modules
  ▶ typings
    index.html
    npm-debug.log
    package.json
    style.css
    systemjs.config.js
    tsconfig.json
    typings.json
```

- `app/` contient la solution du projet.
- `assets/` contient les images nécessaires à la réalisation de la solution et notamment les images de notices utilisées pour cet exemple.
- **bower_component/** et **node_modules/** contiennent les librairies utilisées pour faire tourner Angular2 et celles utilisées pour notre solution (comme anyang, bootstrap...)
- `typings/` contient les fichiers nécessaires à Typscript (utilisé par Angular2).
- **index.html** contient le squelette HTML de notre application. C'est le conteneur de l'application.
- **package.json** et **bower.json** contiennent les dépendances de la solution. C'est à dire les librairies à télécharger à l'installation.
- Les autres fichiers sont des fichiers de configuration de Angular2 et typscript.

Explication de la mécanique Angular2:

Puisque notre application est simple et ne consiste qu'en un "slider", tous les fichiers de la solution sont rangés dans un seul dossier pour plus de simplicité. Il sera préférable

dans une éventuelle version future, de reconfigurer Angular2 afin de pouvoir ranger les fichiers dans des dossiers plus appropriés en fonction de la complexité de l'application.

main.ts:

C'est le fichier qui amorce ("bootstrap") l'application. Il fait appelle au fichier app.module.ts qui contient l'application.

app.module.ts:

Décrit le module de l'application et les directives mises en jeux. Ici nous n'avons qu'un seul composant: AppComponent.

slide.ts:

Ce fichier décrit la forme d'un objet "Slide": comporte un index de type *int* et une addresse d'image de type *string*.

```
export class Slide{
    index: number;
    img: string;
}
```

mock-NickelRing.ts

Comporte la variable de notre notice de démo qui est constituée de plusieurs objets Slide (mis en référence dans "import")

```
import { Slide } from './slide';

export const NickelRing: Slide[] = [
  { index: 0, img: '/assets/NyckelRing/page1.png' },
  { index: 1, img: '/assets/NyckelRing/page2.png' },
  { index: 2, img: '/assets/NyckelRing/page3.png' },
  { index: 3, img: '/assets/NyckelRing/page4.png' },
];
```

app.component.style.css

C'est la feuille de style de notre composant.

slide.service.ts:

C'est le service utilisé pour récupérer le tutoriel de démo (depuis mock-NickelRing.ts). Il est appelé en tant que Provider de notre composant. La bonne pratique veut qu'il renvoie un promise comme expliqué ici.

app.component.ts:

C’est là que se trouve le coeur de notre composant.

Notre app.component est constitué de 3 parties:

La première partie se charge d’inclure les éléments nécessaires à son fonctionnement.

```
import { Component, ApplicationRef} from '@angular/core';
import { Slide } from './slide';
import { SlideService } from './slide.service';
import { OnInit } from '@angular/core';
declare var annyang: any; // for use of annyang in the component
declare var $ : any; //for use of jquery in the component
```

La deuxième partie décrit le composant utilisé:

```
@Component({
  selector: 'my-app',
  providers: [SlideService],
  templateUrl: "app/app.component.template.html",
  styleUrls: ["app/app.component.style.css"]
})
```

Il renseigne notamment dans l’ordre : le nom de la directive à utiliser pour appeler le composant, le provider du composant, le template/vue utilisé et le style utilisé.

Le composant est appelé à l’aide de la directive “my-app” depuis **`index.html`**, (“Loading… étant remplacé par le contenu du composant une fois celui-ci chargé):

```
<my-app>Loading...</my-app>
```

La troisième et dernière partie du composant est sa classe. Elle contient toutes les variables nécessaires à son affichage: titre (*title*), slide courante (*selectedSlide*), tutoriel utilisé (*howto*), fonctionnement ou non du micro (*mic_is_on*). Ainsi que les méthodes nécessaires à son fonctionnement.

```
14  export class AppComponent implements OnInit{
15    title = 'NickelRing';
16    selectedSlide: Slide;
17    howto: Slide[];
18    mic_is_on = false;
19    onSelect(slide: Slide): void {
20      this.selectedSlide = slide;
21    };
22    after(): void{-
30    };
31    before(): void{-
39    };
40    constructor(private slideService: SlideService, private ref: ApplicationRef) { };
41
42    getSlides(): void {-
48    };
49
50    ngOnInit(): void {
51      this.getSlides();
52    };
53    toogleMic(): void{-
75    }
76  }
```

Les méthodes implémentées ici sont:

- **after():** met à jour la slide courante "*selectedSlide*" en passant à la suivante selon le tutoriel "*howto*"
- **before():** met à jour la slide courante "*selectedSlide*" en passant à la précédente selon le tutoriel "*howto*"
- **toogleMic():** démarrer la détection de la voie en arrière plan avec Annyang à l'activation de la détection vocale ou arrête la détection de la voie si on la désactive.
- **constructor():** méthode propre à Angular2, on y met les objets privés à la classe. Ici le service permettant d'accéder au tutoriel de démo (aussi mis en provider) et l'objet *ApplicationRef* permettant par la suite de forcer le rafraichissement de la vue (problème décrit plus bas) à cause de Annyang par:

```
39      this.ref.tick();
```

- **getSlides():** fait appel au service *slideService* afin d'alimenter le tutoriel de démo "*howto*" et positionne la slide courante à la première du tutoriel:

```
42    getSlides(): void {
43      this.slideService.getSlides().then(slides => {
44        this.howto = slides;
45        this.selectedSlide = slides[0];
46
47      });
48    };
```

- **ngOnInit():** méthode appelée par Angular2 à l'initialisation du composant. Le composant doit avoir le label "*implement OnInit*". Cette méthode appelle la méthode **getSlides()** pour avoir le tutoriel.

En résumé: à sa création, le composant fait appel à son constructeur puis à sa méthode OnInit qui va alimenter les différentes variables manquantes grâce au service puis le template associé à ce composant se rafraichira de lui même avec les éléments qui ont été modifiés.

Pour chaque interaction qui change l'état du composant (changement de variable par exemple), la vue du composant se rafraichira d'elle même. (C'est la magie d'Angular2)

app.component.template.html:

C'est ici que se trouve le template utilisé par notre composant. Là où est tout l'intérêt d'Angular2 c'est que nous pouvons directement faire appel aux éléments (variables et fonctions) de notre composant et aussi faire des conditions sur certains éléments.

On peut afficher des éléments en les mettant entre doubles accolades directement dans le dom comme par exemple notre titre de composant:

```
<h1>How to :{{title}}</h1> <!-- title sera remplacé par “NickelRing” -->
```

On peut aussi faire appel à notre image de diapositive courante comme ceci (en utilisant l'attribut de Angular2 pour les images)

```
<img [src]="selectedSlide.img" />
```

On peut afficher quelque chose seulement si le composant l'alimente. Par exemple n'afficher l'image que si elle a fini d'être chargée avec "*ngIf"

```
<div class="img_container" *ngIf="selectedSlide">
      <img [src]="selectedSlide.img" />
</div>
```

On peut aussi faire apparaître certaines classes sous conditions. Par exemple je n'affiche le label "previous" seulement si le micro est activé (classe bootstrap)

```
<span [class.hidden]="mic_is_on === false">
    previous
</span>
```

On peut aussi faire des calculs directement dans le dom pour afficher par exemple l'index des pages et le nombre de page total :

```
<h2>{{selectedSlide.index+1}}/{{howto.length}}</h2>
```

Bootstrap m'a aussi beaucoup aidé à positionner rapidement les éléments grâce à son système de grille. J'ai pu par exemple diviser ma vue en laissant 2 colonnes pour l'affichage des flèches et 8 pour l'affichage de la diapositive (en vert). J'ai aussi pu cacher les flèches sur les cotés en affichage sur petit écran (en rose)

```
<a class="col-md-2 arrow hidden-sm hidden-xs before"
(click)="before()" >
    ...
</a>
<div class="col-md-8" >
    ...
</div>
<a class="col-md-2 arrow hidden-sm hidden-xs after"
(click)="after()" >
    ...
</a>
```

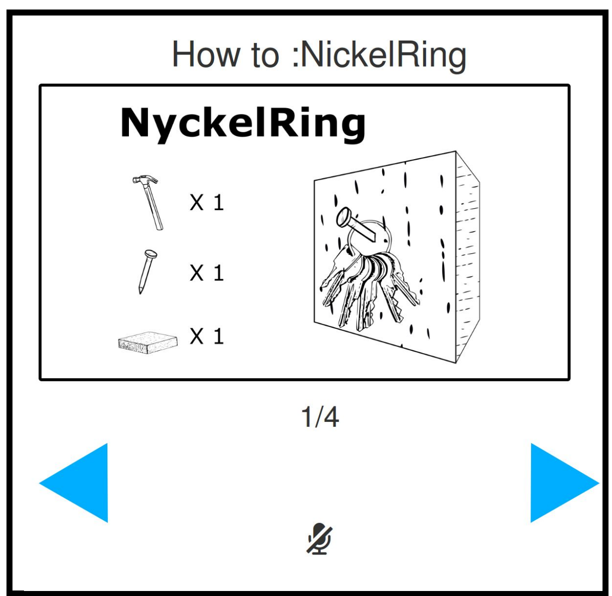
On remarquera aussi les appels au méthodes du composant au clique sur les liens des flèches (en orange).

Grâce à Bootstrap, j'affiche un autre jeux de flèches de navigation sur petit écran sur une ligne ("row") en dessous de la ligne contenant le diapo. Cette ligne n'est pas affichée sur grand écran (rose) :

```
<div class="row hidden-md hidden-lg">
  <a class="col-xs-2 arrow-xs text-left" (click)="before()">
     <img src="/assets/before.svg"/>
  </a>
     <a class="col-xs-2 arrow-xs col-xs-offset-8 text-right"
(click)="after()">
     <img src="/assets/after.svg"/>
  </a>
  </div>
```

J'aligne mes flèches à gauche et à droite grâce à la propriété en jaune.

Je les positionnes sur 2 colonnes (rouge) avec 8 colonnes vides entre elles (bleu) (voir rendu ci-dessous)



Description des étapes suivies et problèmes rencontrés.

Pour tester et débugger du Angular2 il faut dans un premier temps regarder si l'interface s'affiche comme prévu, puis si il y a un problème , voir dans la console du navigateur si des erreurs s'affichent (f12 sous firefox et chrome) ou même dans la console où est lancée Angular2.

Après avoir suivi les tutoriels listés plus haut, j'ai commencé par faire un **lecteur de diapositive basique**.
Cela a été assez rapide grâce à la technologie d'Angular2 qui met à jour la vue automatiquement après un changement.
L'utilisation de Bootstrap a aussi permis une mise en place rapide et propre de la vue.

Par contre l'implémentation de la **reconnaissance vocale** a été plus laborieuse. Je n'ai trouvé aucune librairie spécifique à Angular2 pour la reconnaissance vocale. En revanche nous avons trouvé une librairie JS appelée Annyang (utilisée aussi pour le projet AngularJS).
Installer et intégrer Annyang au container (à *index.html*) a été rapide (installation avec bower et intégration dans une balise script). Cependant, il n'est pas évident de l'intégrer au composant.

En effet, pour qu'un composant d'Angular2 puisse accéder à une variable globale issue d'une librairie comme Annyang, il faut la déclarer en tête de fichier. C'est pourquoi en haut de mon fichier **`app.component.ts`** on peut lire :

```
declare var annyang: any; // for use of annyang in the component
```

Une fois Annyang utilisable dans le composant, je ne parvenais pas à lui faire reconnaître des mots. La raison était que mon navigateur (firefox-linux) n'était pas

compatible: il ne détectait pas le micro de mon ordinateur. C'est pourquoi il est recommandé d'utiliser la solution avec google Chrome.

Une fois ce problème réglé, Annyang reconnaissait les ordres “next” et “previous” mais ne faisait pas passer les diapositives. Plus précisément, après avoir reconnu l'ordre “next” ou “previous”, Annyang appelait bien la fonction du composant after() ou before() pour changer de diapositive, la slide se mettait à jour au niveau du composant mais elle la vue ne changeait pas.

Un des intérêt d'Angular2 est que, quand un composant change, la vue se met automatiquement à jour sans à avoir à forcer le rafraichissement du composant (comme on pouvait le faire dans certains cas avec du angularJS avec un **`$scope.appply();`**).

Après de nombreuse recherches (et essais différents) j'ai appris que la vue pouvait ne pas se mettre à jour si une thread tournait hors du processus d'Angular2. Comme c'était le cas avec Annyang (qui tourne en arrière plan pour détecter un ordre), il m'a fallu avoir recours à un petite astuce pour forcer le rafraichissement du composant : ce qui va à l'encontre du principe d'Angular2.

N'étant pas experte en Angular2, je n'ai pas trouvé d'autre solution pour le moment. Cependant une bonne pratique serait de rendre la librairie Annyang parfaitement intégrée et compatible à Angular2.

Pour la **détection de geste**, je n'ai trouvé aucune librairie pour Angular2 permettant de réaliser de la détection de mouvement à la main. Je suis alors partie d'une librairie standar de reconnaissance faciale faite en JS : https://trackingjs.com/examples/face_camera.html.
Je voulais reproduire un travail fait en SI4 en Computer Vision où nous avions mis en place un tracking de la main à partir d'un algorithme de détection de visage (avec du machine learning en C++). Je n'ai malheureusement pas réussis à obtenir un résultat satisfaisant et ne maîtrisant pas Angular2 je n'ai pas non plus réussi à intégrer le moindre résultat.

J'ai voulu utiliser bootstrap-switch pour faire les switchs d'activation/désactivation des modes d'interaction. Mais je ne suis pas parvenue à les intégrer parfaitement à Angular2 surtout la partie “réaction au clic du switch” . Avec le recul, je pense que celà est possible avec le “hack” expliqué plus haut pour Annyang. Ou alors chercher un switch existant spécial Angualr2.

Les améliorations à apporter et conclusion personnelle.

Angular 2 est un très bon outils du moment qu'on sait l'utiliser sans avoir recours à des “hacks” qui brise ces principes. Selon moi ça vaut la peine de prendre le temps qu'il faut pour se former afin de ne pas prendre de mauvaise habitudes ou avoir recours à de mauvaises pratiques.

On appréciera aussi le fait qu'avec Angular2, le HTML se construit selon le script et que ce n'est pas le script qui construit le HTML comme on peut le voir dans des solutions Web classiques. Angular2 est donc plus intuitif pour ce point.

Les améliorations à apporter à ce projet sont les suivantes:

- Faire marcher la détection vocale sans "hack" (piste 1 et piste 2)
- Trouver/implémenter un système performant de détection de mouvement de main parfaitement intégré à Angualr2 (voir système trouvé pour Electron?)
- Mieux gérer la génération des fichiers (actuellement la génération des fichiers se fait dans le même répertoire que celui de développement pour le moment)
- améliorer le responsive: ce n'était pas le but de l'exercice, mais la vue n'est pas parfaitement adaptée à un smartphone en vue portrait.
- Ajouter une pop-up explicative pour l'activation du micro depuis le navigateur
- Trouver un switch car le bouton n'est pas évident en terme de compréhension d'action

JavaFX par Feng DING

Introduction JavaFx

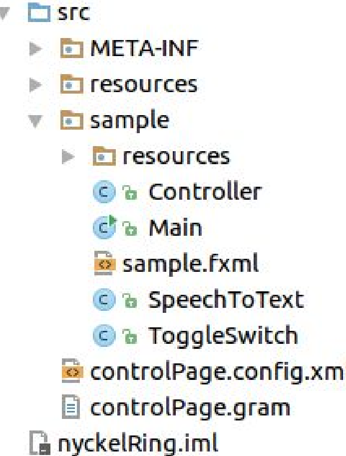
" JavaFX est une technologie créée par Sun Microsystems qui appartient désormais à Oracle.
Avec java 8 , JavaFX est devenue la bibliothèque de création d'interface graphique officielle du langage Java, pour toutes les sortes d'application (applications mobiles, applications sur poste de travail, applications Web), le développement de son prédécesseur Swing étant abandonné (sauf pour les corrections de bogues).
JavaFX contient des outils très divers, notamment pour les médias audio et vidéo, le graphisme 2D et 3D, la programmation Web, la programmation multi-fils etc.

Le SDK de JavaFX étant désormais intégré au JDK standard Java SE, il n'y a pas besoin de réaliser d'installation spécifique pour JavaFX. "

Source: Wikipédia

L'outil de développement

Pour la partie JavaFx, j'ai utilisé intellij pour créer un projet de l'application JavaFx. Après le projet est créé, j'ai trois fichers initiaux qui sont <Controller.java>, <sample.fxml> et <Main.java>. Le fichier <css> est optionnel. Le fichier <fxml> est utilisé pour faire la conception de l'interface d'utilisateur. Le fonctionnalité de chaque composant dans l'interface d'utilisatuer est défini dans le fichier <Controller.java>. <Main.java> est pour lancher l'application.



L'outil de tests

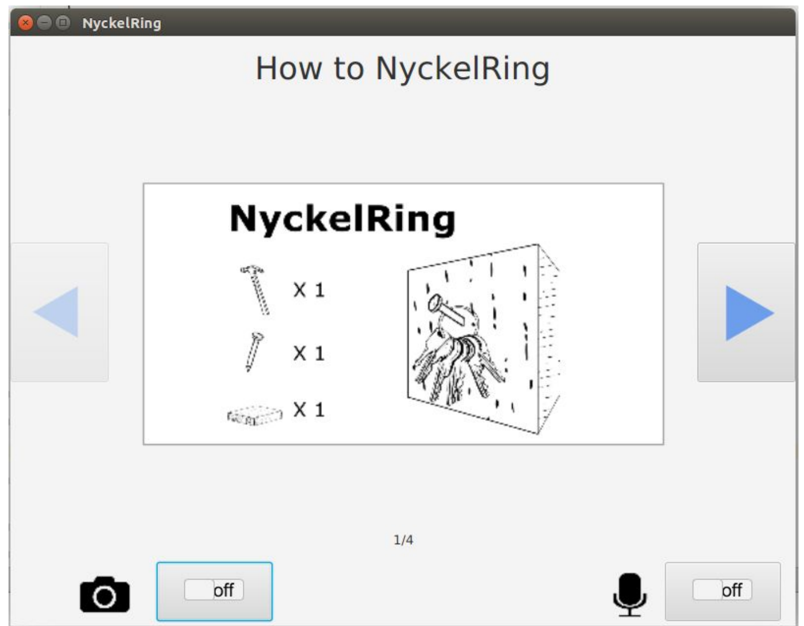
J'ai utilisé trois façons qui sont intellij, jar exécutable et Firefox+tomcat pour tester l'exemple.

Les étapes pour réaliser

Pour finaliser ce projet, je l'ai divisé en trois étapes. La difficulté s'augmente de la première étape à la troisième étape.

Contrôler les slides par deux boutons (suivant / précédent)

Dans cette version j'ai créé exactement le même interface que l'on a décidé dans notre première rendu.



Dans le fichier <fxml>, j'ai utilisé <BorderPane> comme une base de layout. Il est resemble comme le <BorderLayout> dans Java Swing, qui est composé par cinq parties. J'ai ajouté un attribut qui s'appelle <fx:controller> (fx:controller="Controller.java") pour le <BorderPane> pour faire une connection entre le controlleur et l'interface. Pour le principal, j'ai créé quatre boutons et un imageView. Le imageView simule le container des images de slide. Il y a deux boutons qui sont utilisés pour contrôler le slide en tournant suivant ou précédent. Et les autres deux boutons est pour contrôler l'opération d'ouverture et de fermeture de microphone et de caméra. Après les composants ont été bien définis, je défini leurs fonctionnalités dans le fichier <Controller.java>.

Dans <Controller.java>, il y a une méthode <initialize()> qui est appelée automatiquement quand le controlleur est chargé dans le Main. Alors, pour les initializations des objets, je les défini dans cette méthode. Pour localiser les composants dans l'interface, le nom de chaque objet doivent être pareils comme le valeur de l'attribut <fx:id> de chaque composant et il doit avoir une tête <@FXML>. Pour les actions de chaque bouton, il faut les définir dans fxml avec l'attribut <fx:onAction>. Le nom d'opération, c'est-à-dire, la valeur de <fx:onAction> est commencé par un dièse. Et dans le controlleur, je défini une méthode privée avec le même nom de la valeur de l'attribut <fx:onAction> et la tête <@FXML> pour définir l'action. Par exemple, pour le bouton <next>.

Définir le bouton dans le fichier fxml.

```
<Button fx:id="next" mnemonicParsing="false" onAction="#handleButtonNextonAction">
```

Définir l'action de ce bouton dans le controlleur.

```
@FXML
private Button next;
@FXML
private void handleButtonNextonAction(ActionEvent event) {
        System.out.println("next");
        nextPage();
}
```

Dans <Main.java>, la class hérite la class Application. Il faut recharger la méthode <start(Stage stage)> pour initialiser l'interface et le controlleur et pour définir la taille de la fenêtre. On peut aussi ajouter le fichier CSS si c'est nécessaire. Comme java projet, la méthode <main(String[] agrs)> est nécessaire pour lancer le programme.

```
@Override
public void start(Stage primaryStage) throws Exception{
    Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));
//scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm()); //Pour loader le fichier CSS si c'est nécessaire.
    primaryStage.setTitle("NyckelRing");
    primaryStage.setScene(new Scene(root, 800, 600));
    primaryStage.show();
}
public static void main(String[] args) {
    launch(args);
}
```

Contrôler les slides en reconnaissant la voix

Pour cette étape, j'ai utilisé une API qui s'appelle <Sphinx4> pour reconnaître le voix et pour changer le voix aux textes. <Sphinx4> est une bibliothèque JAVA pur de reconnaissance vocale. Il fournit une API rapide et facile de convertir les enregistrements vocaux en texte avec les modèles acoustiques. Il soutient en anglais et aussi de nombreuses autre langues.

```
▼ ant-contrib-0.6
  ▶ ant-contrib-0.6.jar library root
  ▶ batch.jar library root
  ▶ js.jar library root
  ▶ jsapi.jar library root
  ▶ jsapi-1.0-base.jar library root
  ▶ junit-4.8.1.jar library root
  ▶ opencv-2413.jar library root
  ▶ sphinx4.jar library root
  ▶ tags.jar library root
  ▶ TIDIGITS_8gau_13dCep_16k_40mel_130Hz_6800Hz.jar lib
  ▶ WSJ_8gau_13dCep_8kHz_31mel_200Hz_3500Hz.jar librar
  ▶ WSJ_8gau_13dCep_16k_40mel_130Hz_6800Hz.jar library
```

Premièrement, j'ai créé deux fichier qui sont <controlPage.config.xml> et <controlPage.gram>. <controlPage.gram> défini le grammaire de langage qu'il peut comprendre. Par exemple, dans le fichier

```
grammar hello;
public <pageControl> = (next | previous) ;
```

Je choisi deux mots <next> et <previous> et les met dans une liste pageControl. Alors, il peut juste comprendre les deux mots maintenant.

Le fichier <controlPage.config.xml> est pour faire la configuration de grammaire. Quand on prend la configuration fichier, il peut comprendre la grammaire que l'on a défini dans le <.gram> fichier.

Ensuite je créé une classe qui s'appelle SpeechToText.

```java
public class SpeechToText {

  private ConfigurationManager cm;
  private Recognizer recognizer;
  private Microphone microphone;

  public SpeechToText(){

        cm = new
ConfigurationManager(SpeechToText.class.getResource("../controlPage.config.xml"));

        recognizer = (Recognizer) cm.lookup("recognizer");
        recognizer.allocate();
        // start the microphone or exit if the programm if this is not possible
        microphone = (Microphone) cm.lookup("microphone");
        if (!microphone.startRecording()) {
            System.out.println("Cannot start microphone.");
            recognizer.deallocate();
            System.exit(1);
        }
```

```
            System.out.println("Microphone is ready");
    }
    public String recognization(){
            Result result = recognizer.recognize();
          String resultText = "";
            if (result != null) {
                resultText = result.getBestFinalResultNoFiller();
                System.out.println("You said: " + resultText + '\n');
            } else {
                System.out.println("I can't hear what you said.\n");
            }
        return resultText;
    }
}
```

Les classes <ConfigurationManager>, <Recognizer>, <Microphone> sont dans le bibliothèque de Sphinx4. <ConfigurationManager> est pour recharger la configuration de grammaire. <Recognizer> est utlisé pour reconnaître notre voix. Et <Microphone> est pour allumer et pour étaindre le microphone. La méthode <recognization()> est pour transformer le voix aux textes et retourner une chaîne de caractère de ces textes.

Quand on allume le micro avec le bouton, il va créer un Thread pour écouter le voix.

```
private class SpeechRecognize implements Runnable{
    private SpeechToText speechToText = new SpeechToText();
    private volatile boolean running = true;
    public void terminate() {
        running = false;
        System.out.println(running);
    }
    @Override
    public void run() {
        while(running){
            String text = speechToText.recognization();
            if("next".equals(text)){
                nextPage(); //aller au page suivant
            }else if("previous".equals(text)){
                previousPage(); //aller au page précédente
            }
        }
    }
}
@FXML
private void handleButtonMicroOnAction(ActionEvent event){
    if(microButton.isSelected()){
        System.out.println("microOn");
        microButtonImgView.setImage(microButtonOnImg);
        speechRecognize = new SpeechRecognize();
```

```
            speech = new Thread(speechRecognize);
            speech.start();
        }else{
            System.out.println("microOff");
            microButtonImgView.setImage(microButtonOffImg);
            if(speech != null){
                speechRecognize.terminate();
                speech.join();
            }
        }
    }
```

Quand on éteint le microphone, le Thread s’arrête en sécurité.

Contrôler les slides en reconnaissant le mouvement

Pour reconnaître le mouvement, je utilise la bibliothèque opencv. Pour bien présenter l’action, je choisi une couleur spéciale et la défini avec la façon HSV (Hue, Saturation, Value) dans le code. (HSV minRange = (53,74,160) et HSV maxRange = (90,147,255))

Ensuite, j’ai créé une méthode pour capturer l’objet avec la couleur correspondante. Les variable entrées pour cette méthode sont tous les matrices. <maskedImage> est la matrice de image mask. <frame> est la matrice de image qui dessine le contour d’objet. Les explications de code sont dans le commentaire.

```
private Mat findObject(Mat maskedImage, Mat frame){
  // initialization
  List<MatOfPoint> contours = new ArrayList<>();
  Mat hierarchy = new Mat();
   // On utilise la méthode findContours qui est dans la classe Imgproc de opencv pour capturer le contour d’objet qui a la couleur correspondante. Après il va sauvegarder les infos dans la liste de matrice <contours> et le matrice <Hierarchy>.
        Imgproc.findContours(maskedImage, contours, hierarchy, Imgproc.RETR_CCOMP, Imgproc.CHAIN_APPROX_SIMPLE);
//On crée un liste de <Moments> pour calculer et pour sauvegarder le moments d’objet avec les informations de son contour.
  List<Moments> mu = new ArrayList<Moments>(1);
//Soit l’objet existe.
  if(contours.size() != 0) {
//On utilise la méthode <moments> dans la classe de Imgproc pour calculer le moments.
      mu.add(0, Imgproc.moments(contours.get(0), false));
      Moments p = mu.get(0);
//x et y sont les coordonnées du moments de contour.
      int x = (int) (p.get_m10() / p.get_m00());
      int y = (int) (p.get_m01() / p.get_m00());
//Parce que l’on doit analyser la direction de mouvement d’objet. Alors, je défini deux int global CenterX et CenterXold. Chaque 20ms, on met à jour les deux valeur et les compare pour analyser le mouvement d’objet.
      centerX = x;
      if (first) {
         centerXold = x;
         first = false;
```

```
    }
  }
  return frame;
}
```

Après on allume la caméra, on crée un Thread pour capturer et analyser l'image. Il capture un image chaque 20 microseconds. La méthode <grabFrame()> est utilisé pour analyser la direction de mouvement d'objet. Elle retourne une chaîne de caractère, soit <left> soit <right>.

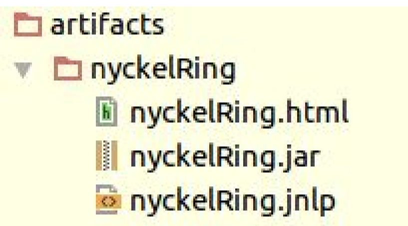
```
// grab a frame every 20 ms (30 frames/sec)
Runnable frameGrabber = new Runnable() {
  @Override
  public void run()
  {
    String direction = grabFrame();
    if("left".equals(direction)){
      nextPage();
    }else if("right".equals(direction)){
      previousPage();
    }
  }
};

this.timer = Executors.newSingleThreadScheduledExecutor();
this.timer.scheduleAtFixedRate(frameGrabber, 0, 20, TimeUnit.MILLISECONDS);
```

Pour analyser le mouvement, j'écris les codes suivant. Soit l'objet dans l'image nouvaux est 5 pixels le plus qu'il est dans l'ancien, je défini qu'il en train de bouger à droite. Et le conteur s'augmente. S'il continue à augmenter, la méthode retourne <right> pour dire que l'objet a bougé de gauche au droite.

```
private int count = 8;
if(centerX - centerXold > 5){
  count++;
}else if (centerXold - centerX > 5){
  count--;
}
System.out.println(count);
if(count==0){
  System.out.println("left");
  count = 8;
  direction =  "left";
}else if(count==16){
  System.out.println("right");
  count = 8;
  direction =  "right";
}
if(!first){
  centerXold = centerX;
}
```

Déployer et exécuter l’exemple

Après on generate l’application dans intellij, on a trois fichiers. <.html>, <.jar> et <.jnlp>.



Alors, on peut exécuter l’application directement dans le terminal.

java -jar nykelRing.jar

On peut aussi lancer l’application dans le navigateur à l’aide de fichier <.html> et de fichier <.jnlp>. Mais il faut installer la bibliothèque de JavaRunTime pour le navigateur.

Capacité d’adaptation

La capacité d’adaptation de JavaFX est forte, mais avec contrainte. Parce que l’on a toujours les fichiers <.html> et <.jnlp> après la génération du projet. Alors, on peut toujours la lancer dans un navigateur. C’est-à-dire, on peut lancer l’application sur le portable, le smartphone, la tablette ou le système embarqué. On peut aussi utiliser CSS pour définir le layout de l'application en RWD. Mais il faut que le navigateur peut installer la bibliothèque JavaRun ou le device support JavaMe.

Conclusion personnelle

Après, le développement de ce projet. Je trouve que JavaFX est un bon outil pour développer une application crossOS. Mais, il n’est pas facile pour configurer le device pour lancer l’application. Par exemple, ajouter un bibliothèque sur le navigateur. Et il y a aussi la fonctionnalité qu’il n’est pas supporté pour le navigateur, par exemple la reconnaissance vocale.
Pour faire un layout RWD, il y a de contrainte pour le faire par CSS. J'utilise la façon de changer le conteneur (Horizontal box(HBox) et Vertical box(Vbox) dans le JavaFX) pour le réaliser.

Conclusion générale:

Angular2 est plus difficile à prendre en main que AngularJS mais permet d'avoir des rendus simples plus rapidement. De plus la découpe en mode "component" facilite la récupération de composant pour une autre utilisation. Sur le rendu technique les deux Frameworks restent quand même très proche et même avec différentes librairies de style et différents manière de les importer, on ne note pas de grosses différences vraiment majeur.

Dans notre cas, l'utilisation de React et Électron n'est pas préconisé de par le fait que ce dernier n'est plus compatible avec la voice detection. Nous perdons donc une des fonctionnalités principale. Par contre, React fait très bien son travail qui est de découper le code en composants réutilisables comme le fait Angular2.

Nous pouvons également comparer AngularJS, Angular 2 et React sur plusieurs points:

- Difficulté d'apprentissage : React semble le plus facile car il n'inclut que très peu de nouvelles notions (seulement des états) alors que les 2 version d'angular apporte plusieurs points (directives, models, services ..)
- Code généré : AngularJS et React sont bien en dessous d'Angular 2 qui est très gourmand en matière de lignes de code
- Rigidité : React a l'avantage d'utiliser du javascript natif, les seuls limites sont le langage lui même. Les 2 Angulars possèdent leur propres règles (basé sur JS bien sûr) mais ont l'avantage d'avoir un code écrit plus lisible selon certains point de vue.
- Puissance (vitesse + efficacité) : Angular 2 semble être un poil devant React, AngularJS est loin derrière

Java étant un langage étant un langage compilé, nous ne pouvons pas le comparer aux autre sur les points précédent.

Les améliorations de l'application que nous pouvons apporter :

- faire une interface pour sélectionner une notice parmis plusieurs disponible à partir d'une base de donnée en ligne.
  - AngularJS: Pour cette fonctionnalité, il suffirait d'ajouter une nouvelle vue et un nouveau controller, il faudra également ajouter la directive ngRoute pour permettre a l'utilisateur de naviguer entre les deux pages.
  - Angular2: Cela revient à faire un composant en plus qui liste les tutoriels disponibles en faisant appel à un service qui les récupère sur une base de donnée et modifier le service existant pour le composant actuel. + faire un fichier de routing
  - Electron: Il faudrait ajouter un composant supplémentaire (une liste) qui contient une liste de composant (des items).
  - JavaFX: Il faudrait ajouter une fenêtre avec des notice.

- Pouvoir charger une notice présent sur l'appareil (type pdf):

- AngularJS: Prospecter au niveau des composant ou directives d'upload de fichiers.
- Angular2: piste: http://valor-software.com/ng2-file-upload/
- Electron: un <input type="file"/> avec une lecture du fichier grâce à l'api HTML5 Blob fait l'affaire
- JavaFX: Utiliser la classe File avec la path de fichier le récupérer.

- Ajouter du feedback utilisateur pour la détection du micro et de la caméra :
  - AngularJS: Ajouter une barre de détection de son et un petit retour de la caméra dans un coin de la page principale.
  - Angular2: piste: faire un composant de détection de micro/caméra et le positionner dans un coin.
  - Electron: Ajouter 2 composants, un détecteur de son (et volumétrie) et un affichage caméra (déjà fonctionnel, juste pas apparent pour concorder avec les autres interfaces).
  - JavaFX: Il faut ajouter les composants pour que l'utilisateur peut savoir si le micro ou le caméra fonctionne bien.

Annexe

Installer Node et Npm

La démarche pour installer Node et Npm est très simple

- Windows: L'installation la plus simple reste d'exécuter l'installateur téléchargeable sur ce lien : https://nodejs.org/en/

- Linux: ici, afin d'avoir nodejs, npm et certaines libs considérées comme essentielles, il est plus avisé d'utiliser les commandes suivantes

```
$ curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
$ sudo apt-get install -y nodejs
$ sudo apt-get install -y build-essential
```

Installer bower

Attention: Vous devez avoir installé node et npm !

Linux/Ubuntu

Ouvrez une console et tapez:

```
$ npm install -g bower
```

Il est possible que bower ne soit pas automatiquement ajouté à vos binaires , pour cela, rajouter le lien (chez moi node est installé dans /opt):

```
$ sudo ln -s /opt/node-4.6.0/lib/node_modules/bower/bin/bower /usr/local/bin
```