# Approche générale

Le projet est en Angular 4, Typescript et géré par angular-cli (génération, build, tests).

La partie graphique est basée sur du Bootstrap 3 avec la librairie ngx-bootstrap.

La documentation est générée par compodoc qu’on lancera ainsi :

./nodes\_modules/.bin/compodoc -t ./tsconfig.json

Il y a pas mal de programmation fonctionnelle, il faut être à l’aise avec l’immuabilité, le spreading de tableaux et d’objets. En revanche, le pattern architectural est du classique MV\*, avec autant que possible un respect du Single Responsability Principle (SRP). On aura donc des composants de type « légers », faisant appel à des services injectés pour tout ce qui est traitement interne et externe (requêtes) et des pipes pour la gestion des affichages programmatiques.

# Structure

C’est du très classique généré initialement par angular-cli. Le point d’entrée est donc app.module.ts. Les modules de premier niveau sont :

* Core : l’application une fois que l’utilisateur est connecté
* Forbidden : la page pour indiquer un refus de connexion
* Idp-authorized : gestion du retour d’OpenId, c’est ce qui va déclencher l’ouverture d’une session
* Not-Found : la page 404
* Shared : tout ce qui est commun à l’ensemble de l’application

Le seul module de premier niveau à disposer de sous-modules significatifs est CoreModule. Il est décomposé selon les sections de l’application : Dashboard, Bim-Assistant et Admin (désactivé pour l’heure).

# Bootstrap

La première chose qui est faite est la récupération via HTTP du fichier config.json. Ce fichier doit être lu avant l’initialisation même de l’application puisque la configuration de cette dernière en dépend. Typiquement on trouvera dans le fichier config.json (sous la forme d’un simple objet JSON) les urls d’API, d’authentification, des paramètres pour debugger plus facilement, etc. Pour savoir comment sont utilisés ces attributs, faire une recherche dans les fichiers sur « config.get ».

Le processus de lecture démarre dans app.module.ts. Il passe par l’utilisation d’un token d’injection interne à Angular : APP\_INITIALIZER. Evidemment, celui-ci est multi et on peut y greffer le chargement du config.json. L’emploi de useFactory est nécessaire afin de pouvoir injecter en cascade (deps) le ConfigService dont le rôle est le chargement en lui-même, et la récupération des valeurs (méthode get). A noter : les valeurs de config sont hard codées dans app.constants.ts pour tout ce qui concerne les environnement de production Renault.

Une fois le chargement de config.json effectué, le composant global de l’application (AppComponent) prend le relais. Il s’occupe d’enclencher le processus d’OpenId via AuthService. Au bout, il y a redirection vers le module/composant Idp-Authorized qui fera à son tour une redirection vers la page de départ de l’application, après ouverture d’une session (SessionService).

# Usage de la librairie D3

D3 fournit deux types de fonctionnalités :

* Des helpers pour calculer les éléments nécessaires à la génération de graphiques.
* Des helpers pour générer le DOM à partir de ces éléments.

Comme nous sommes en Angular, il est « interdit » de gérer le DOM (création et mise à jour) autrement que par le framework lui-même. Toutefois, dans le cadre d’un usage restreint cette interdiction est moindre (ça n’en reste pas moins une TRES mauvaise pratique).

Sur RRF il y a deux types de graphiques : pie (camemberts dans le dashboard) et treemap (dans le Bim-Assistant). Le premier type est géré par pie.component.ts et respecte la contrainte Angular. Pour être précis, D3 a pour seul rôle de calculer les paramètres du graphique et de générer la string SVG qui sera ensuite évaluée par Angular.

Le treemap en revanche se base intégralement sur D3. Il y a donc ajout, update et suppression de nœuds DOM directement par D3, sans aucun contrôle par Angular. Si vous voyez des comportement étrange sur les treemaps, ça sera un point à investiguer.

Comme dans l’ensemble de l’application, la détection de changement se fait par immutabilité. Ainsi, dans le cas du pie, lorsque les données changent, c’est tout le graphique qui est redessiné. En revanche, pour le treemap, comme D3 gère le DOM, il s’occupe d’ajouter ou retirer les nœuds en fonction d’une fonction d’identification. En l’occurrence il s’agit simplement de

d => d.id

dans

this.rootNode.selectAll("g").data(nodes, d => d.id)