Chapitre 1

Introduction

Chapitre 2

Le contexte du travail

2.1 Le cahier des charges ?

2.2 L'architecture de la plateforme cible

2.3 Le framework Angularjs ?

AngularJS [1] est un framework écrit en javascript par Google libre et open-source qui permet d'améliorer la syntaxe de javascript ainsi que la productivité du développeur. Il étend le HTML pour le rendre dynamique, et permet de développer ses propres balises et attributs HTML. C'est un framework qui se veut extensible et qui pousse vers un développement structuré, en couches, le but n'étant pas d'ajouter de simples animations au DOM, mais bien d'apporter un aspect applicatif au front-end. AngularJS est construit autour de concepts

2.3.1 Pourquoi angularJS

2.3.2 Concept

Angular est construit autour de concepts et de bonnes pratiques incontournables dans le monde du développement web.

—

1. url de angular

2.4 Les ressources REST

Un API REST est un système D'URIs, les ressources déterminent la structure des URIs par conséquent la manière dont une application trouve les données qu'elle manipule. REpresentational State Transfer (REST) est un style d'architecture pour les systèmes distribués bâti sur des APIs permettants de centraliser des services partagés. Il est donc évident que plusieurs applications technologiquement hétérogènes utilisent ces services via un réseau. Par exemple, on peut créer un compte Instagram [2] utilise avecles données de son compte facebook via un API de face

2.5 Le contrôle d'accès en utilisant des rôles (RBAC)

2. C'est quoi instagram

Chapitre 3

La Conception de notre générateur de module

3.1 Le concept général

3.1.1 Définition d'un module

Un module est une application métier [1] basée sur l'architecture client-serveur. Nous nous référons au terme module pour indiquer le caractère modulaire que peut prendre une application. Par exemple,la gestion des resources humaines est un assemblage de plusieurs sous-applications telles que la gestion des employés, la gestion des carrières, la gestion de la paie et la gestion des embaûches [2].

L'architecture client-serveur, implique qu'un module est composé d'une application *client* et d'une application *serveur*. L'application client est une interface graphique [3] permettant à aux utilisateur d'interagir avec un système d'information. L'application serveur est celle qui gère les données métiers dans un système d'information. La gestion des données est faite essentiellement via une base de données et les processus métiers.

3.1.2 Le portail applicatif des modules

C'est l'interface principale [4] permettant de présenter les différentes fonctionnalités des modules. Comme le montre la figure **??**, elle est divisée en six parties à savoir :

1. A expliquer
2. donner la liste complète si possible
3. spécifier pourquoi c'est une interface graphique
4. parler du PAE

— La barre d'entête (cf. le chiffre 1 sur la figure **??**). Cette partie indique le nom du module courant, la liste des tâches à réaliser et la liste des modules.

— Le bloc de menus (cf. le chiffre 2 sur la figure **??**). Il présente les différents éléments du menu d'un module.

— Le bloc des modules liées (cf. le chiffre 3 sur la figure **??**). Présente la liste des modules en lien avec le module courant.

— L'espace de travail (cf. le chiffre 4 sur la figure **??**).

— La barre d'actions (cf. le chiffre 5 sur la figure **??**).

L'nterface est basée sur la notion d'applications monopages[5]. De ce fait, les parties une, deux, trois et cinq sont statiques alors que la partie 4 est dynamique. Les éléments des parties statiques changent avec l'installation d'un nouveau module.

Actions versus menu. Les éléments du menu permettent de naviguer à travers les différentes vues d'un module alors que les actions permettent de faire des opérations sur les éléments de la vue courante. D'un point de vue technique, la différence se trouve au niveau des composants avec lesquels ils sont implmentés : les menus sont implémentés sous forme de liens et les actions boutons.

Liste des modules versus modules liés La liste des modules de la barre d'entête représente l'ensemble des modules installés sur le système tandisque la liste des modules liés représente les dépendances entre différents modules. Les modules de gestion de stocks et d'achats sont liés ou dépendent du module de la gestion des articles. Dans ce cas de figure, le module gestion des articles aura dans sa liste de modules liés les modules stocks et achats. L'intérêt principal des modules liés est d'éviter à l'utilisateur, d'aller chercher un module parmi une liste de module qui peut vite être conséquent.

L'installation d'un module. Elle consiste, pour un module donné, à fournir les éléments de parties statiques. ces dernières sont ensuite positionnées dans leur bloc respectif.

3.1.3 Le principe générale

L'idée du générateur est d'utiliser un environnement graphique permettant de spécifier les différents éléments nécessaires à l'implémentation des applications client et serveur, en vue d'augmenter la productivité générale[6] d'un développeur. En effet, une fois les éléments spécifiés, nous

5. Aussi nommée Single Page Application (SPA)
6. listing des productivités

passons à la génération automatique des codes des applications client serveur. Ceci permettra aux développeurs de se focaliser sur la conception de leurs modules.

Nous regroupons les éléments du module en deux catégories : ceux *côté client* et ceux *côté serveur*. Les éléments côté client sont ceux qui sont nécessaires pour la mise en place de l'interface principale. Ils permettent de définir les différentes pages ou vues html ainsi que les services qui permettent de recupérer les données du serveur. Les éléments côté serveur permettent de définir les bases de données, les processus métiers et ressources permettant d'interagir avec le client.

Productivité. Nous la définissons comme le temps nécesssaire à un dévelopeur pour implémenter, tester et pour déployer une application. Dans ce travail nous nous intéressons aux éléments qui font perdre du temps durant l'implémentation d'une application ; nous les avons regroupé comme suit :

- *les erreurs de syntaxe.* Beaucoup d'erreurs dans l'implémentation d'une application sont dues à des erreurs de syntaxe ou des erreurs de frappe dans le nom des fonctions ou des variables.
- *la recherche à l'aveugle.* Rechercher des informations telles que la valeur d'une variable ou la définition d'une fonction peut vite devenir fastidieux lorsqu'on travaille sur de grosses applications.
- *code ennuyeux.* Les développeurs perdent beaucoup du temps à chercher comment implémenter certain de leur concept dans des langages qu'ils n'apprécient pas ou qu'ils ne maîtrisent. Ils perdent aussi du temps à écrire plusieurs fois le même code.
- *distraction ?*

Nos solutions pour améliorer la productivé des développeur tournent autour de la réutilisation, l'isolation , la génération et la modularité de code. Dans le reste du document nous allons ... bla bla bla.

3.2 La présentation des éléments graphiques côté client

3.2.1 Overview

Présentation de la vue générale

La vue générale se concentre sur les éléments côté client d'un module. Elle est l'étape principale dans le processus de réalisation d'un module. Aussi simple qu'elle parait, elle rassemble tous les outils nécessaires pour concevoir chacun des éléments de la partie cliente d'un module. Elle se résume à l'interface de la figure suivante. Sous les onglets `components` et `treeview`, nous disposons respectivement de la palette de composants html et de leur organisation sur la vue en forme d'arborescence. sous les onglets `variables` et `functions`, on définit les variables et

les fonctions qui sont utilisées pour écrire le contrôleur de la vue. Sous les onglets `actions` et `configs` on définit respectivement les actions du module et les paramètres de configuration de la vue.

Les composants

Ils constituent un outil graphique permettant de manipuler les éléments qui entrent dans l'élaboration d'une interface web. Les manipulations sont basées sur un certains nombres de propriétés que nous leur avons associé afin de controller leur mise en forme, leurs attributs html ainsi que les données qu'ils manipulent.

Nous avons resencé les composants les plus couremment utilisés dans les applications web et nous les avons regroupé en trois grandes catégories :

- *Les composants simples* ; ce sont des éléments HTML dont la définition ne nécéssite pas l'imbrication d'un autre composant. exemple :
- *Les composants conteneurs* Ce sont des éléments qui peuvent contenir des composants de n'importe quelle catégorie, par exemple l'élément html `form` permet de regrouper les composants simples tels que les `input` et des `button` pour la réalisation d'un formulaire.
- Les composants widgets Ce sont des éléments non standards que nous avons entièrement écris, en proposant leur design et leur conception.

L'arbre des composants

Le concept de l'arbre des composants est d'augmenter la productivité des dévéloppeurs en limitant le perte de temps due aux changements récurents dans le développement des interfaces [7].

L'arbre des composants est un outil permettant de représenter les composants d'une interface. Ses composants sont organisés sous forme d'arbre afin de mieux schématiser la hiérarchie induite par les composants conteneurs. La figure **??** représente l'arbre des composants de l'interface de la figure **??**.

Notre concept se décline via l'interface de la figure **??** dans laquelle on constate qu'un certains nombres d'actions sont associées à chaque composant. En effet pour permettre la réutilisation de codes nous proposons les cinq actions suivantes :

- Monter - descendre : elles permettent les déplacements de composants vers le haut ou vers le bas, dans le même conteneur. Ceci permet de réorganiser les composants d'un même conteneur.

7. explication détaillée

— couper : elle permet les déplacements de composants d'un conteneur à un autre. Ceci permet la réorganisation globale des composants.

— copier : permet de cloner un composant, soit dans un même conteneur ou non. Ceci permet de générer facilement une interface avec des composants semblables.

— Supprimer : elle permet d'enlever un composant de l'arborescence.

Les fonctions

Le concept ici est d'améliorer la productivité du dévloppeur en :

— évitant les *erreurs d'écriture* dans les noms des fonctions dans les vues et dans les contrôleurs.

— proposant une *vue d'ensemble* du travail à réaliser (isolation ?).

Le concept se décline via l'inteface de la figure **??**

Erreur d'écriture Le lien « generate functions », génère les noms des fonctions, à partir de l'arbre des composants. Les composants tels que button, ... et ... font appel à des fonctions qu'ils définissent dans leurs propriétés. à partir des propriétés des composants cités plus haut, nous extrayons le nom des variables à déclarer.

Vue d'ensemble Nous pouvons dinstinguer parmi les variables à définir, celles qui sont dejà définies ou non. La définition d'une variable se fait en l'isolant des autres variables. Ceci permet au développeur de se focaliser sur le reste des variables et surtout de ne pas perdre de temps à chercher une information dans les codes qui peuvent rapideent devenir illisibles.

Les variables

Le concept ici est d'améliorer la productivité du dévloppeur en :

— évitant les *erreurs d'écriture* dans les noms des fonctions dans les vues et dans les contrôleurs.

— proposant une *vue d'ensemble* du travail à réaliser.

— simplifiant la déclaration des variables.

Le concept se décline via l'inteface de la figure **??**

Erreur d'écriture. Le lien « generate vars », génère les noms des variables, à partir de l'arbre des composants. Les composants manipulent des variables. à partir des propriétés, nous extrayons le nom des variables à implémenter.

Vue d'ensemble. Nous pouvons dinstinguer parmi les fonctions à implémenter, celles qui sont dejà définies ou non. La définition d'une fonction se fait en l'isolant [8] des autres fonctions. Ceci permet au développeur de se focaliser sur le reste des fonctions et surtout de ne pas perdre de temps à chercher une information dans les codes qui peuvent rapideent devenir illisibles.

Déclaration. Les variables peuvent être de type *resolve*, *async* et *cache*. les variables de type *resolve* sont celles dont les valeurs sont présentes avant le chargement de la page qui les utilise. les variables de type *async* sont celles dont les valeurs proviennent d'un serveur de manière asynchrone ; dans ce cas, le service permettant d'obtenir la donnée doit être préciser. Les variables de type *cache* sont celles dont les valeurs sont mises ou proviennent d'un cache lorsque l'attribut cache a pour valeur `to` ou `from`.

Les actions

Ils constituent un outil graphique permettant de spécifier les actions associées à une vue donnée.

représentent une partie des fonctionnalités du module faisables à partir de la vue encours d'édition. Pour ajouter une fonction, il faut :

- Préciser le nom de l'action ;
- Préciser le rôle de l'utilisateur qui peut y avoir accès ;
- Préciser l'icône de l'action, c'est sur cette dernière qu'il faut cliquer pour exécuter l'action ;
- Préciser les dépendances ; awesome file input
- Enfin écrire le corps de la fonction de l'action

La configuration

Le concept ici est d'améliorer la productivité du développeur en simplifiant la mise en place des menus, des droits et de la navigation. Ce concept se décline via l'interface de la figure suivante **??**.

Navigation. Elle consiste à construire l'état/route auquel(le) la vue est associée tout en précisant le nom de l'état, l'url associé et le chemin static ou dynamic du template. Ainsi, en parcourant la liste des vues, on peu les connaître les différents états du module.

Menus. La case à cocher « module menu »permet de spécifier si une vue a une entrée ou non dans la liste des menus d'un module. De ce fait, il est possible en parcourant la liste des vues

8. collapsible

de la figure **??**, d'obtenir les éléments du menu. On pourra non seulement identifier les icônes associées aux menus mais aussi changer l'ordre d'apparition des éléments dans le menu par un simple déplacement dans la liste des vues.

Droits d'accès

3.3 La présentation des éléments graphiques côté serveur

L'idée est de proposer un concept permettant le développement des APi restful de manière simple et conviviale. le développement des Api passe non seulement par la définition de chaque ressource mais aussi par la définition des fonctions ou méthodes associées aux différentes ressources.

Dans cette section nous présentons la définition des ressources ainsi que les concepts que nous avons défini autours des ressources. Dans un premier temps, nous avons introduit les concepts d'*héritage* et de *déclencheur* dans la définition des ressources afin de faciliter la structuration des grosses applications et de renforces de manière générale la modularité de code. Dans un second temps nous avons implémenté une ressource pour réaliser le chainage de ressources et une autre pour l'indexation de données ; ceci afin de limiter la perte de temps due à la procrastination et les code ennuyeux (cf. paragraphe 3.1.3)

La définition de ressources

Elle se fait via l'interface de la figure **??**. L'interface est composée de trois parties. la partie 1 permet d'avoir une liste hiérachisée des ressources déjà déclarées tandis que les parties 2 et 3 permettent, en isolant les ressources, de voir leur définition et de les compléter éventuellement.

L'héritage de ressources

L'idée est, d'une part, d'exploiter la définition et l'implémentation des ressources d'un module dans un autre module. D'autre part il nous permet de simplifier les droits d'accès aux ressources générique (à expliquer car pas trop clair).

Soient le module `personne` et le module `client` qui hérite de `personne`. Les tables **??** et **??** présentent respectivement les ressources de `personne` et ceux heritées de `client`. Un client étant une personne, nous pouvons alors utiliser l'implémentation des resources du module `personne` pour avoir la liste des clients. La particularité de l'héritage est que l'uri des ressources de `client` n'ont pas la même base que ceux des ressources `personne` ; ceci permet au module `client` de spécifier ses propres droits d'accès sur les ressources provenant de `personne`. Ce type de gestion des droits nous permet d'éviter l'attribution de droits trop complexe et pas toujours efficace.

En effet, si les modules `travailleur` et `client` héritent de `personne`, il sera impossibe pour un utilisateur ayant uniquement des droits sur `travailleur` d'accéder aux ressources `client`. Si nous mettons des droits sur les ressources de `personne`, tout utilisateur ayant les droits sur `travailleur` devra aussi avoir des droits sur `personne`, ce qui lui permettra par un effet de bord d'accéder aux ressources de clients sans y avoir les droits.

Déclencheur

C'est un mécanisme permettant de faire exécuter des tâches en arrière plan après une exécution réussie d'une ressource, sur le serveur. Par exemple, après une authentification réussie, il est possible de déclencher des tâches sur l'utilisateur en utilisant un déclencheur.

Ressources chaînées

La ressource d'indexation

Les opérations

Les opérations sont étroitement liées aux ressources

Les données Externes

Chapitre 4

Implémentation

4.1 Introduction

4.2 La structure interne d'un module

Nous utilisons un document json pour le stockage des différents éléments d'un module. Le document json est organisé autour des 5 attributs présentés dans la table 4.1. L'attribut `name` est une chaîne de caractères représentant le nom du module tandis que les attributs `models`, `ressources`, `opérations` et `dependencies` sont des tableaux représentant respectivement les vues, les ressources, les opérations et les dépendances. Comme le montre la figure **??**, seuls les éléments de l'attribut `dependencies` sont optionnels ; le tableau des dépendances peut donc rester vide.

Attributs	Types	Optionnel
name	string	–
models	array	–
ressources	array	–
operations	array	–
dependencies	array	oui

TABLE 4.1 – Les attributs du document json permettant de stocker un module.

4.2.1 Les dépendances

Le tableau `dependencies` représente la liste des modules dont dépend un module. Une dépendance est représentée par un objet doté de l'attribut `name`, représentant le nom d'un module et de l'attribut `type` représentant le type de dépendance. L'attribut `type` est optionnel et ne peut

Attributs	Types	Optionnel
num	integer	–
id	string	–
label	string	–
state	string	–
url	string	–
icon	string	–
children	array	oui
functions	array	oui
variables	array	oui
actions	array	oui

FIGURE 4.1 – Les attributs de l'objet json permettant de stocker une vue.

Attributs	Types	Optionnel
name	string	–
role	string	oui
icon	string	oui
action	string	oui

FIGURE 4.2 – Les attributs de l'objet json permettant de stocker une variable.

avoir que « inherited »comme valeur pour spécifier un héritage. Par défaut lorsque rien n'est spécifié, la dépendance est considérée comme une dépendance normale.

4.2.2 Les vues

Attributs	Types	Optionnel
name	string	–
cache	string	oui
service	string	oui
value	string	oui

TABLE 4.2 – Les attributs de l'objet json permettant de stocker une variable.

Le tableau `models` représente l'ensemble des vues d'un module. Une vue est caractérisée par un objet json dont les attributs sont décrits dans le tableau4.1. Les différents attributs de la vue peuvent être regroupés en trois catégories :

— *configuration.* Les attributs `state`,`label`, `url`, `icon` et `actions`, représentant respectivement l'état, le nom, l'url , l'icône et les actions associées à la vue, sont les éléments nécessaires lors de la génération de la configuration.

Attributs	Types	Optionnel
name	string	–
content	string	–

TABLE 4.3 – Les attributs de l'objet json permettant de stocker une variable.

— *model.* L'attribut `children`, représentant la liste des composants d'une vue, est l'élément nécessaire à la génération des pages html.

— *controller.* Les attributs `functions` et `variables`, représentant respectivement les fonctions et les variables associées à une vue, sont nécessaires à la génération des contrôleurs.

L' attribut `actions` est un tableau d'objets dont les attributs sont présentés dans la table 4.2. Les attributs `name` et `icon` représentent les éléments visibles de la barre des actions (cf. **??**). L'attribut `action` représente le code à exécuter tandis que `role` représente le droit qu'un utilisateur doit avoir pour faire cette action.

L' attribut `variables` est un tableau d'objets dont les attributs sont présentés dans la table 4.2. Les attributs`name` et `value` sont nécessaires à la déclaration de la variable. Si la valeur de l'attribut `resolve` est faux alors la déclaration est faite lors de la génération du contrôleur, sinon elle est faite lors de la génération de la config. Au cas où la déclaration est faite dans le contrôleur, si l'attribut `local` est vrai, alors la variable sera générée comme une variable locale. Si l'attribut `async` est vrai, alors il faut préciser le nom du service qui permet de récupérer la valeur de la variable avec l'attribut `service`.

L' attribut `functions` est un tableau d'objets dont les attributs sont présentés dans la table 4.3. pour générer une fonction, il faut deux attributs obligatoires que sont `name` et `content` représentant respectivement le nom et le contenu de la fonction.

L'attribut `children` est un tableau d'objet dont les attributs sont présentés dans la table **??**. l'attribut `type` représente le type du composant html, et les attributs `style` et `class` représentent l'habillage css du composant. L'attribut `children` représente le tableau de sous-composants d'un conteneur. Mis à part les attributs cités, chaque composant peut insérer dans l'objet des attributs qui lui sont propres.

4.2.3 Les ressources

Le tableau `resources` représente l'ensemble des ressources REST d'un module. Une ressource est caractérisée par un objet json dont les attributs sont décrits dans le tableau4.7. Les attributs `resources` et `parent` sont les éléments nécessaires pour la gestion de l'arborescence. En effet l'attribut `resources` est un tableau de mes objets que celui nous sommes en train de décrire. L'attribut `path` représente l'uri d'une ressource. tandis que l'attribut `methods` représente les méthodes associées à la dite ressource.

Attributs	Types	Optionnel
path	string	–
parents	string	oui
resources	array	oui
methods	array	oui

TABLE 4.4 – Les attributs de l'objet json permettant de stocker une variable.

Attributs	Types	Optionnel
name	string	–
id	string	–
requests	array	oui
responses	array	oui
role	string	oui
type	string	oui
trigger	array	oui

TABLE 4.5 – Les attributs de l'objet json permettant de stocker une variable.

L' attribut `methods` est un tableau d'objets dont les attributs sont présentés dans la table **??**. Les attributs `name`, `id`, `requests`, `responses` sont les éléments de définition d'une ressource classique. L'attribut `role` permet de définir un droit d'accès sur la ressource. L'attribut `type` représentant le type de ressource, est un élément nécessaire dans la génération du code (côté serveur). Il peut prendre les valeurs « request », « command », « reply », « empty », « inheritance »et « inherited ». L'attribut `trigger` représente une ressource à exécuter en tâche de fond.

L'attribut trigger est un objet caractérisé par l' attribut `name`, représentant le nom de l'opération à exécuter, et l'attribut `from`, représentant le nom du module de provenance.

4.2.4 Les opérations

Le tableau `operations` représente l'ensemble des opérations associées aux ressources d'un module. Une opération est caractérisée par un objet json dont les attributs sont décrits dans le tableau**??**.

4.3 Le principe de la génération

4.3.1 Les contrôleurs

La génération des contrôleurs consiste dans un premier temps, à parcourir le tableau des variables pour en extraire les dépendances à injecter dans l'entête du contrôleur. Nous déclarons ensuite

Attributs	Types	Optionnel
name	string	–
from	string	–

TABLE 4.6 – Les attributs de l'objet json permettant de stocker une variable.

Attributs	Types	Optionnel
name	string	–
type	string	–
reply	string	–
sql	object	oui
in	array	–

TABLE 4.7 – Les attributs de l'objet json permettant de stocker une variable.

les variables suivi de la définition des fonctions. Nous générons le tout dans un fichier dont le nom est la concaténation de l'attribut `label` et du mot « Ctrl ».

```
$scope.HTMLcomponents = HTMLcomponents;
$scope.model = globalVarFactory.getModule().models;
```

La figure **??** est un exemple de génération des variables présentées à la figure **??**. Comme on peut le constater, la génération se fait simplement en mettant les noms des variables comme attribut dans le scope et en leur attribuant comme valeur celle de l'attribut `value` ou service. Lorsque l'attribut `resolve` a pour valeur vrai, la valeur de la variable provient alors de la dépendance injectée dans l'entête ; c'est le cas de notre variable `HTMLcomponents`.

4.3.2 Les services

La génération des services consiste à créer un objet dont les attributs sont les méthodes associées aux ressources REST. l'idée est d'encapsuler l'appel d'une ressource REST dans un simple appel de fonction. Les services sont déclarés dans un fichier dont le nom est la concaténation de l'attribut `label` et du mot « Service ». Ainsi, les ressources de la figure**??** .... à completer.

4.3.3 Les vues

La génération de la vue consiste à générer le code html des composants du tableau children en fonction des propriétés positionnées par l'utilisateur. Outre les propriétés présentées à la section 4.2.2, les composant disposent des propriétés qui leur sont propres.

`input.` Sa génération se fait selon le modèle de la figure **??**.

```
<DIV [class="$class"] [style="$style"]>
    [<LABEL for="$id">$label</LABEL>]
     <INPUT [readonly] [placeholder="$label"]
     name="$id ||$name" id="$id" type="$type" ng-model="$model"/>
</DIV>
```

`hx.` Sa génération se fait selon la figure **??**. La construction de la balise est dynamique et dépend de la valeur de l'attribut `type`. Les attributs `model` et `label` permettent de définir soit un contenu dynamique soit un contenu statique.

```
<H$type>$label || ${model}</H$type>
```

`i.`

```
<i class="$class"  [style="$style"]></i>
```

`radio.` La génération des boutons radio se fait selon la figure **??**. Outre les attributs communs, ce composant introduit les attributs `separator` et `options`, cette dernière est une chaîne de caractère constituée des différentes options du bouton radio séparées par un séparateur spécifié par l'attribut `separator`. En effet pour générer ce composant, on parcourt la chaîne des options et on extrait chaque option délimiter par le séparateur, puis on génère un bouton pour l'option.

```
<DIV [class="$class"]>
[<P>$label</P>]
FOR item IN $options
      <INPUT id="$id" value="$value" ng-model="$model" type="radio" name="$name" [checked]>
      <LABEL for="$id">$item</LABEL>
ENDFOR
  </DIV>
```

`textarea.` La génération de ce composant se fait selon le code de la figure **??**. Il introduit l'attribut `codeEditor`, si ce dernier a pour valeur « vrai »alors le composant sera générer en tant qu'éditeur de code. Ce composant a été généré dans l'interface `vue` sous les onglets « variables »et « functions ».

```
<DIV [class="$class"] [style="$style"]>
    [<LABEL for="$id">$label</LABEL>]
     <TEXTAREA  [placeholder="$label"] class="form-control"
     name="$id ||$name" id="$id" [ui-codemirror="cmOption"] ng-model="$model"></TEXTAREA>
</DIV>
```

button. La génération des boutons se fait selon le code de la figure ??. Le bouton introduit les attributs `div` et `dynamic`. Ces attributs sont optionnels et spécifient respectivement si l'on veut générer le bouton avec une balise « div »au lieu de la balise « button »traditionnelle, et si l'on veut que le label du bouton soit dynamique.

```
<[DIV] || [BUTTON] [class="$class"] [style="$style"] id="$id" >  $label <<[DIV] || [BUTTON]>
```

select. La génération d'un `select` se fait selon le code de la figure ??. Il introduit les attributs `collection`, `rvalur` et `dvalue`. ces derniers sont obligatoires et spécifient respectivement le tableau de données qui alimente le composant, la valeur retournée par le composant et enfin la valeur affichée.

```
<DIV  class="form-group">
      [<label for="select297">$label</label>]
         <SELECT  [class="$class"] [style="$style"] name="select297" id="select297" ng-model="me
          <OPTION ng-repeat='item in $collection' value="{{item.$rvalue}}"> {{item.$dvalue}}   <
        </SELECT>
</DIV>
```

label. La génération d'un `label` se fait selon le code de la figure ??. Ce composant peut avoir un contenu statique ou dynamique selon que l'utilisateur spécifie une valeur pour l'attribut `label` ou spécifie la valeur de l'attribut `model`.

```
<LABEL [class="$class"] [style="$style"] id="$id">[$label] || [$model] </LABEL>
```

hr• La génération d'un `label` se fait selon le code de la figure ??. Ce composant n'a pas besoin de paramètres particuliers en dehors de l'habillage css.

```
<HR [class="$class"] [style="$style"] id="$id"/>
```

span• La génération d'un `span` se fait selon le code de la figure ??. Il n'a pas d'attributs particuliers et peut soit avoir un contenu statique si l'attribut `label` est renseigné ou un contenu dynamique si l'attribut`model` est renseigné.

```
<SPAN [class="$class"] [style="$style"] id="$id">[$label] || [$model]</SPAN >
```

img• La génération du composant `img` se fait selon le code de la figure ??. Il introduit l'attribut `src` qui permet à l'utilisateur de spécifier la source de l'image que le composant va afficher.

```
<IMG [class="$class"] [style="$style"] id="$id" src="$src"></IMG>
```

`group`• La génération du composant `group` se fait selon le code de la figure **??**. Ce composant introduit autant d'attributs que de paramètres nécessaires pour sa configuration. Il participe à la construction d'autres composants en fonction desquels ses attributs sont ou non optionnels. En effet nous avons :

- `position`. Cet attribut spécifie la position du composant sur l'écran, il possède 3 options que sont « center », « right »et « left ».
- `width`. C'est la largeur du composant, elle se définit en colonne et un `group` en avoir 12 au total.
- collapsible. Il spécifie si le composant sera pliable/ dépliable
- `uiview`. spécifie si oui ou non le composant affichera des bouts de pages html associées à des états donnés.
- `row`. Si cet attribut a pour valeur « vrai », le contenu du composant sera afficher en ligne.
- `src`. Spécifie le chemin vers une page html qui s'affichera dans le composant.
- `expression`. Spécifie les conditions dans lesquelles le composant sera affiché ou caché.

```
<DIV [class="$class"] [style="$style"]  id="$id"> </DIV>
```

`paragraph`• La génération du composant `group` se fait selon le code de la figure **??**.

```
<P  [class="$class"] [style="$style"]  id="$id"> </P>
```

`tab`• La génération du composant `tab` se fait selon le code de la figure **??**. Il est conçu à partir du composant `group`. En effet, il possède un attribut « children »qui est un tableau de `group`. Chaque groupe représente un onglet et ici, `group` introduit les attributs « index »et « heading »qui représentent respectivement l'index de l'onglet et le nom de l'onglet.

```
 <UIB-TABSET >
       FOR item IN children
          <UIB-TAB index="item.num"  heading="item.label">
              </UIB-TAB>
       ENDFOR
</UIB-TABSET>
```

4.3.4 Configuration des (routes) états

Nous générons la configuration des états dans un fichier config. Elle se fait à base de paramètres tels que les

4.4 La présentation du générateur sous forme de module

4.4.1 Les vues

Le générateur de vues

4.4.2 Les ressources

4.4.3 La base de données

4.4.4 La collaboration

Chapitre 5

conclusion