Rapport DSL 2

UXifier

01/03/2022

Par l'équipe C, composée de :

FARGEON Armand - BELKHIRI Abdelouhab - EL ADLANI Rachid - ROCCELLI Valentin - FERTALA Mohamed

Sommaire

Lien du code : https://github.com/armandfargeon/UXifier

1. Définitions

Notre sujet est le Dashboarding. Nous allons souvent utiliser le terme “Widget”.
Dans notre contexte, un “Widget” est un élément contenant les données que le designer utilisant notre DSL voudrait présenter. Ces “Widgets” peuvent prendre plusieurs formes, comme nous allons le détailler dans la suite de ce rapport. Le terme plateforme signifie les différentes tailles d’écran.

Une partie qui explique les différentes partie de notre extension .alc

Notre langage utilise l’extension .alc, fichier que créera notre designer.
Il contiendra le contenu de l’*app* et son identifiant.

Dans l’*app*, nous devons définir :

- le *theme*, suivi de son identifiant et de sa définition,
- le *header*, avec son identifiant, son niveau, sa couleur, le titre et le logo qui apparaîtra dans l’onglet du navigateur web,
- la définition des widgets qui n’apparaîtront pas directement et qui seront réutilisables à travers les pages, avec le mot clé *hide*,
- le *menu* de l’application avec son identifiant et sa définition.

Le *theme* permet de définir une liste de couleurs, utilisables ailleurs dans la page.
Une *couleur* est définie par son identifiant et son *code* en hexadécimal.

La section *menu* est le cœur de notre site. En effet, on va ici déclarer la liste des *page*s puis ensuite leur contenu.

Un *widgetWrapper* est défini par son identifiant, la largeur de l’écran qu’il occupe en pourcents et les widgets qu’il contient. C’est une enveloppe d’un ou plusieurs widgets, nous détaillerons notre choix pour ce composant de manière plus détaillé dans l’analyse critique

Chaque *page* est définie par son identifiant, son titre, et des *widgetWrapper*s.

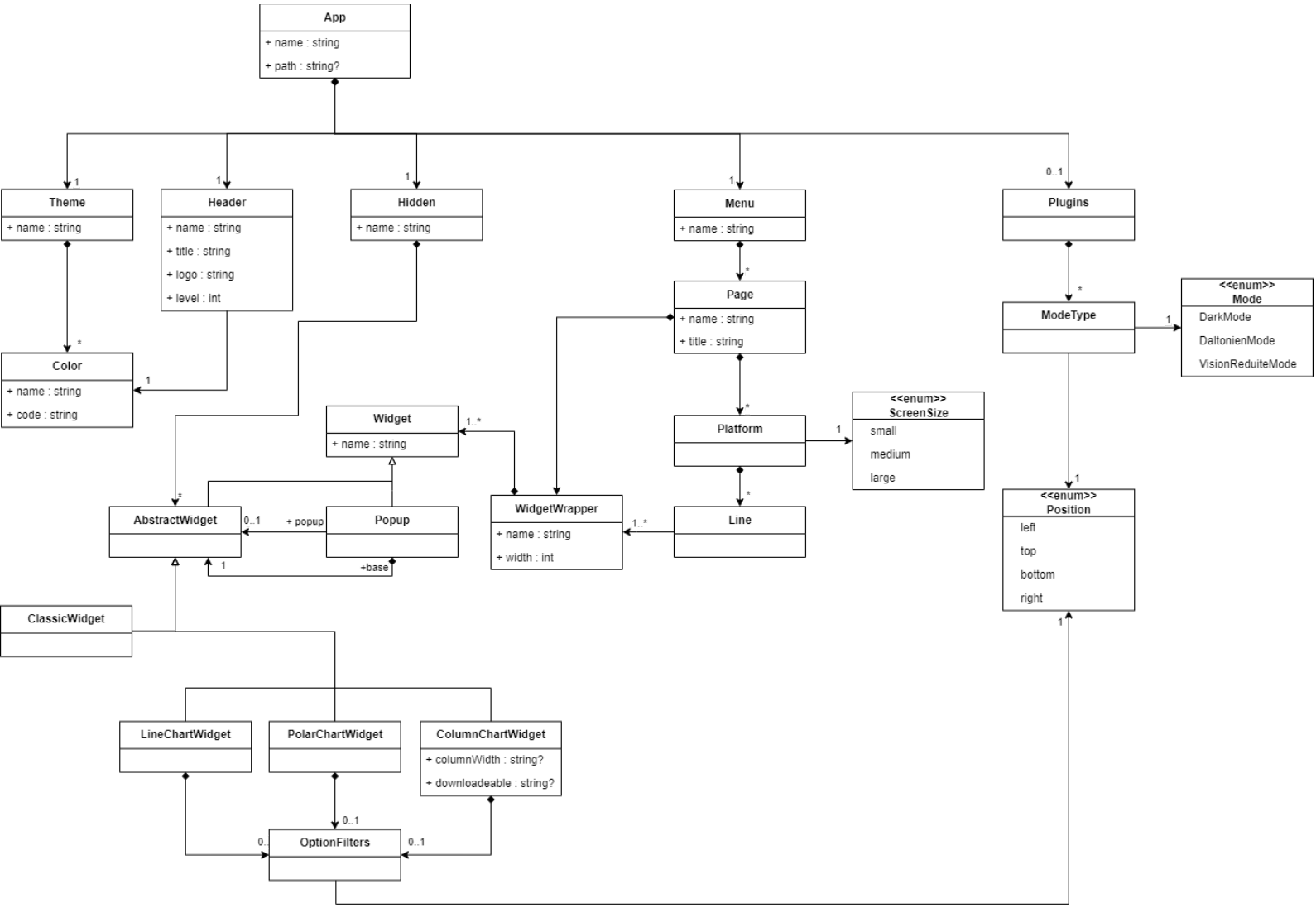
Ensuite, pour définir comment la page doit apparaître en fonction de la *plateform,* il est possible de définir, pour chaque taille d’écran (small, medium, large), quels *widgetsWrappers* (déjà définis ou non) doivent apparaître sur quelles *lines*. Si certaines plateformes n’ont pas été redéfinies, ils utiliserons la déclaration des widgetWrappers en début de page, sans bénéficier du système de ligne (et seront donc affichés dans l’ordre de la déclaration).

Un widget, caché ou non, est simplement défini par son type (classic, *columnchart*, *linechart* et *polarchart*) suivi des données qu'il doit afficher. Ces données sont représentées dans le DSL par un identifiant qui permet de les identifier parmi toutes les données que nous avons créées.

Certains widgets sont encore plus personnalisables.
En effet, après avoir spécifié leurs données, nous pouvons définir, pour *columnchart*, *linechart* et *polarchart*, où doivent se positionner leurs *filters (left / right / top / bottom)*.
En plus, *columnchart* a, comme option, la possibilité de définir la largeur des *column* et s'il est possible de le télécharger, avec le mot clé *downloadable*.

2. Description of the language proposed

Domain Model



Lien

Explication des relations de composition dans notre modèle domaine

App est composite de Theme, Header, Hidden, Menu et Plugins car c'est le seul qui les crée et se sert d'eux.

Theme est composite de Color car nous définissions les couleurs dans Theme, et Header utilise ensuite une référence sur la Color déjà créée.

Des ModeType ne sont instanciés et utilisés que par un Plugin, donc il est normal que ces premiers appartiennent à ces derniers.

- Les Page ne sont instanciées et utilisées que par un Menu (car il n'y a qu'un Menu).
- Les Platform ne sont définies que pour une Page car on peut définir l'affichage en fonction d'une plateforme de manière différente pour deux pages différentes.
- La découpe en Line n'est faite et utilisée uniquement pour l'implémentation de l'affichage en fonction des Platform.
- Un Widget ne peut pas être utilisé tel quel, il a besoin d'être contenu dans un WidgetWrapper.
- Les trois *ChartWidget sont composites de OptionsFilters car chacune des instances de ces widgets aura ses propres options pour ses filtres à lui.

donc la composition est une relation logique pour ces cas.

Un WidgetWrapper peut tout aussi bien être défini et utilisé dans une Page que dans une Line. Ainsi, c'est Page qui sera composite de WidgetWrapper car, même si un WidgetWrapper est défini dans une Line, le cycle de vie de ce WidgetWrapper dépend de la Page.

Dans la section Hidden, nous définissons des AbstractWidget qui peuvent être réutilisés à travers les pages et qui ne sont pas obligés d'apparaître. Ainsi, il est logique que l'objet Hidden soit composite des AbstractWidget qui sont créés en son sein.
NOTE : nous aurions pu avoir la composition entre l'App et AbstractWidget, mais nous avons préféré créer un objet Hidden qui, à notre sens, rendait le tout plus compréhensible.

Finalement, Popup contient une référence vers et est composite d'un AbstractWidget car la Popup n'apparaît qu'après avoir cliqué sur une AbstractWidget (et pas un Widget car on ne veut pas faire une Popup sur une Popup) → référence; et la Popup contient un AbstractWidget qu'il est le seul à contenir et dont il le seul responsable de son cycle de vie → composition.

3. Concrete syntax in BNF-like form

Définition du squelette de l'application :

```
App::= 'app' name=ID '{ theme=Theme header=Header hide=Hidden menu=Menu
plugins=Plugins? '}';

Theme::= 'theme' name=ID '{‘ 'colors''{‘ (colors+=Color)+ '}’ '}';

Header::= 'header' name=ID 'title' title=STRING 'logo' logo=STRING 'level'
level=INT color=[Color|FQN];

FQN::= {FQN} ID ('.' ID)*;

Color::= 'color' name=ID 'code' code=STRING;

Plugins::= 'plugins{' (modes+=ModeType)+ '}';

ModeType::= mode=Mode 'X' posX=Position;
```

Définition des widgets :

```
WidgetWrapper::= 'widgetWrapper' name=ID "width:" width=INT 'px'  '{'
widgets+=Widget (widgets+=Widget)* '}';

Popup::= 'popup' name=ID base=AbstractWidget "=>"
popup=[AbstractWidget|FQN]?;

Widget::= AbstractWidget | Popup;

AbstractWidget::= ClassicWidget | LineChartWidget | PolarChartWidget |
ColumnChartWidget;

ClassicWidget::= 'classic' name=ID;

ColumnChartWidget::= 'columnchart' name=ID ( "{"
(optionsFilters=OptionsFilters)? ("column" "width" "measures"
columnWidth=STRING)? ('downloadeable' downloadeable=STRING)? "}")?;

LineChartWidget::= 'linechart' name=ID ("{"
(optionsFilters=OptionsFilters)?  "}" )?;

PolarChartWidget: 'polarchart' name=ID ("{"(optionsFilters=OptionsFilters
)? "}" )?;

OptionsFilters ::= ("{" "filters" "on" "the" position=Position "}")?

Hidden : 'hide' name=ID '{' (widgets+=AbstractWidget)* '}';
```

Dans notre grammaire il y a une certaines spécificités qui ont été mises en place pour répondre à certaines problématiques.
Premièrement, nous avons dû définir notre propre **énumération** car **Langium** n'a pas encore ce modèle de données en natif:

```
Position returns string: 'left' | 'top' | 'bottom' | 'right';
Mode returns string: 'DarkMode' | 'DaltonienMode' | 'VisionReduiteMode';
```

Dans un second temps, nous ne pouvions pas faire de la **cross-référence** dans un objet qui était imbriqué dans un autre. Pour ce faire, nous avons utilisé la notion du **FQN** afin de permettre de faire du chaînage d'identifiant afin d'accéder à la référence de l'objet souhaité. L'ordre des éléments est important dans notre grammaire.

4. Implementation of language - how it was implemented

Contexte et objectifs de notre DSL

Nous avons créé un **DSL** pour un domaine spécifique : le *Dashboarding*. En effet, notre DSL se destine à tous ***UI/UX** designer* qui a pour objectif de produire des rapports dynamiques en forme de dashboard à partir de données. Si, par exemple, un employé administratif récolte des statistiques et doit les afficher aux grands publics, ce DSL pourra l'aider dans sa réalisation. En effet, il pourra réaliser l'affichage de différents widgets sous plusieurs formats : histogramme, graphique, simple donnée et description, camembert… Après avoir généré son interface utilisateur, il peuplera cette UI avec ses propres données identifiées par des noms.

Choix technologiques - DSL externe

Pour l'implémentation du DSL choisi, nous avions le choix entre le DSL **interne** avec **Groovy** et le DSL **externe** avec **Langium**. Nous avons décidé de partir sur le DSL externe avec Langium, car, malgré l'aspect novateur du projet, ce dernier fournit un ensemble de fonctionnalités de développement. A partir de la grammaire, Langium génère un **AST** nous permettant, à chaque ajout de nouvelles entités dans la grammaire, de générer automatiquement sa représentation dans un fichier ***ast.ts***. Ceci évite de passer par un kernel intermédiaire et de devoir y faire des modifications manuelles. Le service fourni par Langium permettant la gestion de l'AST et qui représente notre ***domain model*** est le service ***ASTReflection***. De plus, Langium est fourni avec un **LSP** que l'on utilise avec VScode qui nous permet d'avoir une analyse syntaxique que l'on peut faire évoluer pour ce projet. Il y a une multitude de services fournis par le LSP (***ReferenceFinder***, ***HoverProvider***, ***DocumentHighliter…***).

Groovy est un moyen simple pour créer un DSL interne car il va se reposer sur un langage hôte. Cependant, il peut s'avérer bloquant lorsqu'il s'agit d'un projet dans lequel on cherche à avoir une syntaxe appropriée, on se retrouve très vite limité.

Il y a, dans un premier temps, la partie grammaire. La grammaire définit comment utiliser notre DSL lors de l'écriture d'un script. Sur Langium, la grammaire se base majoritairement sur Xtext (à quelques exceptions près). Par exemple, Xtext implémente le "***until token***" qui n'a été pris en compte que récemment dans Langium. La grammaire utilisée dans Langium étend de la forme **BNF**, le **EBNF** (Extended Backus-Naur Form-like). Nous utilisons, dans notre DSL, des regex afin de définir des types. Celles-ci sont basées sur le dialecte *js regex*.

Choix technologiques - Framework UI

Nous sommes partis de l'idée que l'utilisation d'un framework afin de générer une interface utilisateur semblait quelque chose de cohérent. En effet, cela nous permettrait d'avoir une couche d'abstraction afin de nous faciliter le développement, de bénéficier d'un ensemble de fonctionnalités déjà implémentées et d'avoir un tronc commun afin de faciliter le travail cohérent en équipe. A partir de nos connaissances des différents frameworks et des recherches et tutoriels faits, nous avons choisi le framework **Grommet**.

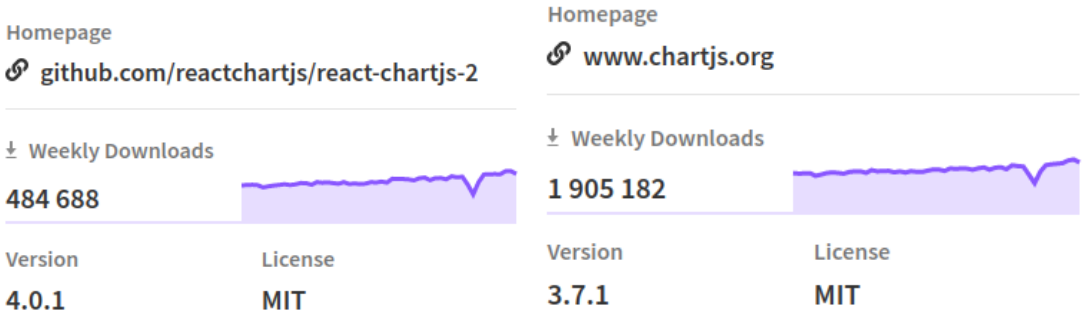
Grommet est une surcouche du framework React, elle donne accès à une bibliothèque de composants UI réutilisables. Partir sur un framework avec une architecture **MVC** serait beaucoup plus contraignant pour nous, il aurait fallu générer un nombre conséquent de fichiers et gérer le fait qu'ils communiquent ensemble avec du binding.

Environnement Grommet

Grommet est une application hiérarchisée composée de fichiers et dossiers nécessitant un serveur Web afin de permettre l'affichage dynamique d'une page web. Notre DSL permet la génération d'un certain nombre de fichiers directement dans la hiérarchie Grommet (qui sera détaillé par la suite), ces fichiers seront à destination d'un serveur web local mis en place pour notre projet.

Choix technologiques - Bibliothèques utilisées

Nous avons pris l'initiative de voir ce qui nous était proposé concernant les bibliothèques utilisées pour obtenir des graphiques qualitatifs et flexibles dans leurs configurations. Nous avons cherché ce qui s'intégrait le mieux avec **Grommet** et avons choisi ***chart.js*** ainsi que ***react-chartjs-2*** (qui est un component qui utilise *chart.js* avec React). Ce sont des bibliothèques qui sont fréquemment utilisées par les utilisateurs de **npm (gestionnaire de dépendances)**. Ceci assure donc une certaine maintenabilité. Nous avons aussi utilisé ***grommet-controls***, une bibliothèque qui s'appuie sur ces deux dernières et qui permet une meilleure intégration des graphiques dans notre application.



5. Implementation of the semantic model

Semantic model - Langium

Dans notre DSL, le target code est une application Grommet utilisant du HTML, du JavaScript et du CSS. Cela est tout de même facilité par ***Langium*** car Typescript est une surcouche du langage ***Javascript***. La génération du *target code* se fait avec le fichier **generator.ts**. Ceci permet d'analyser l'AST réconcilié, ce qui nous fournit toutes les données nécessaires pour générer le code selon notre logique métier. Afin de générer le code cible nous utilisons le **transformer generation**.

Dans notre cas de génération, nous étions contraints de faire plusieurs passes dans l'AST. Dans un premier temps, nous générions toutes les déclarations de nos widgets dans des fichiers séparés. Nous avons aussi opté pour une séparation des différents composants dans des fichiers distincts, regroupés dans un dossier commun aux composants réutilisables. Pour cela on génère le système de fichiers lors de la **deuxième passe** grâce au pattern **composite** qui nous permet de générer un arbre avec plusieurs nodes (fichiers components).

6. Set of relevant scenarios

Les scénarios présentés sont sous le dossier "scenarios", à la racine du repository.

Des widgets pour afficher des données

fichier : basic_scenario.alc

Dans ce scénario "basique", nous voulons simplement afficher tous les widgets disponibles. Pour ce faire, le designer utilisera notre DSL pour définir ces widgets afin de permettre l'élaboration d'un dashboard complet.

Popup

fichier : popup.alc

Dans ce scénario nous allons mettre en valeur l'utilité des popup en mettant deux widgets côte à côte. Le Widget de droite sera un widget classique et celui de gauche sera cliquable, permettant à une Popup d'apparaître. Cette dernière contiendra un Widget affichant plus d'informations à l'utilisateur.

Adaptation programmable aux différentes plateformes

fichier : responsive_scenario.alc

La fonctionnalité que nous voulons démontrer dans ce scénario est l'adaptabilité programmable aux différentes plateformes (PC/Tablette/Smartphone).

En effet, au-delà de l'adaptativité classique des sites webs, nous laissons à notre designer la possibilité de définir comment les widgets apparaîtront selon la plateforme de l'utilisateur final.

Pour présenter ceci, voici les pages que pourraient créer un designer :

- ➔ Page 1: une présentation différente selon la plateforme pour garantir le meilleur rendu, avec le même contenu. Il veut ainsi positionner deux widgets: une actualité sur la crypto ainsi qu'un montrant les différents cours de crypto-monnaies populaires.
  - ◆ Pour la version mobile: un widget par ligne.
  - ◆ Pour les écrans plus larges (tablette, PC): les deux widgets sur la même ligne
- ➔ Page 2: ici il va vouloir afficher un contenu différent selon la plateforme:
  - ◆ Pour la version mobile: un récapitulatif du nombre de cas contacts.
  - ◆ Pour les écrans plus larges: un graph plus détaillé.
- ➔ Page 3: cette page est commune à toutes les plateformes.
  - ◆ Sera affiché des informations de contact.

Ces différentes pages sont implémentées sous forme d'onglets au sein de la webapp. Ces onglets sont définis par le designer.

Dans le développement de cette application web, le designer va, pour chaque page, commencer par définir les widgets qu'il souhaite utiliser au sein de cette page.

Pour la première page et après avoir défini les deux widgets, il va, pour chaque plateforme, décrire le contenu de chaque ligne. Il définit pour les petits écrans, deux lignes. Pour les deux autres, il en définit une seule contenant les deux widgets.

Pour la seconde page, les actions sont similaires, sauf que cette fois, pour la plateforme mobile, notre designer choisit d'afficher seulement le récapitulatif des informations et le graphique détaillé pour les autres types d'écran.

Pour la dernière page, il déclare le widget contenant les informations de contact et ne déclare aucune plateforme. Toutes affichent alors ce seul widget, de la même façon.

En plus de pouvoir choisir la présentation ou le contenu selon les plateformes, il peut également définir la taille des widgets. Chaque widget doit être inclus dans un "WidgetWrapper", qui possède une taille définie en pourcentage. Ici, il aura choisi la taille de 100% pour les widgets qui tiennent sur une ligne et deux de 50% pour les lignes qui attendent deux widgets (PC, Tablette), qu'il aurait pu également regrouper dans le même WidgetWrapper avec une taille de 100%.

7. Critical analysis

Limitations de Langium

D'après ce que l'on a pu expérimenter sur Langium avec nos 2 projets, nous avons apprécié la facilité de prise en main de cette technologie. **Langium** donne accès à de nombreux services notamment par l'intermédiaire du LSP, ce qui facilite le développement et diminue le temps de "***mise en production".*** Cependant, l'aspect très novateur de la technologie nous amène à rencontrer certaines difficultés. L'insuffisance ou l'absence de documentation nous a empêchés de répondre à certaines problématiques concernant les spécificités du langage.

Flexibilité avec EBNF

Langium utilise **EBNF**, ce qui rend l'écriture de notre grammaire plus simple, avec la possibilité de spécifier des champs facultatifs, un ordre de spécification plus ou moins flexible, des références vers des types objets. Il y a aussi la fonctionnalité de préciser un **FQN** qui va faire de la cross-reference et qui nous permet de récupérer un objet imbriqué dans un autre.

Widget Wrapper

Les différents widgets d'une page sont tous inclus dans un composant que l'on a nommé "**WidgetWrapper**". Ce composant est une enveloppe permettant d'inclure un ou plusieurs widgets et dispose d'une certaine taille définie en pourcentage. La taille d'un WidgetWrapper indique la proportion qu'il prendra sur une ligne, et tous les widgets qu'il contient se partagent cette taille.

Nous avons choisi de ne pas mettre une taille directement au widget pour que le designer puisse regrouper plusieurs widgets dans un même composant. De plus, s'il souhaite que certains widgets prennent plus de place que d'autres, le designer peut créer un nouveau WidgetWrapper avec une taille plus importante. Cela permet aussi de réutiliser ces regroupements entre les différentes plateformes.

En effet, si une des lignes de sa page est similaire sur plusieurs plateformes, il n'a qu'à inclure tous les widgets voulus dans cette enveloppe et le déclarer sur une des lignes de chacune des plateformes.

Nous avons pensé à une amélioration intéressante pour ce composant qui serait de laisser le designer redéfinir la taille d'un WidgetWrapper au moment où il déclare une ligne sur une plateforme. Cela lui permettrait d'éviter de redéfinir certains widgets uniquement pour la présentation. Par exemple, il définit un WidgetWrapper qui contient deux widgets.

Actuellement, il doit définir sa taille qui sera la même pour toutes les plateformes. S'il veut ajouter sur la même ligne un troisième widget pour un écran plus grand, il doit définir un WidgetWrapper qui contient ces trois composants au lieu de réutiliser l'ancien, de lui mettre une taille de 2/3 et d'ajouter un nouveau WidgetWrapper qui contient le troisième widget.

Nous n’avons pas implémenté cette feature car nous avions, en parallèle, d’autres idées ajoutant de la valeur à notre DSL. Bien entendu, nous sommes bien conscients que c’est une amélioration intéressante en matière d’expérience utilisateur pour le designer qui conçoit l’application.

8. Responsibility of each members

Noms	Responsabilités
Toute l’équipe	- Définition de notre sujet, des fonctionnalités que nous allions implémenter dans notre DSL et des éléments dont nous allions avoir besoin. - Écriture de ce rapport.
FARGEON Armand	- Implémentation de l’adaptation des pages en fonction de la plateforme. - Implémentation des scénarios.
BELKHIRI Abdelouhab	- Implémentation différents types de widgets - Componentization - Génération de la base de code - Gestion snippets / validator / button action…
EL ADLANI Rachid	
ROCCELLI Valentin	- Design du “résultat final” en Grommet (les différents éléments et l’adaptation des pages en fonction de la plateforme)
FERTALA Mohamed	- Implémentation des popups. - Automatisation des builds avec le script.

9. Main usage and features

Comme défini dans notre contexte, notre DSL permet la création de dashboard avec un certain niveau de personnalisation.

Nous allons définir un certain nombre de services orientés ***designer*** mis en place dans notre DSL.

1. Feature - Validator

Langium intègre un système de **validator** permettant de faire de la vérification statique. Il va donc procéder à une vérification de certaines données, grâce au fichier **ast.ts** qui a été généré afin que le système puisse automatiquement prévenir l'utilisateur s'il effectue des actions qui mettent le système en état d'erreur.

Par exemple, lorsque l'utilisateur inscrit le type de widget qu'il souhaite utiliser, s'il fait une faute de frappe ou qu'il renseigne un type de widget qui n'est pas reconnu par le système, celui-ci lui indique où se trouve son erreur et lui propose les types existant dans notre grammaire. Nous proposons également une vérification similaire pour les couleurs qu'a définies l'utilisateur. Dans une version antérieure du DSL, l'utilisateur pouvait définir le chemin d'un répertoire sur sa machine permettant de générer l'application Grommet. Nous faisions une vérification de l'existence de ce path. Cependant nous avons, depuis, automatisé le processus de génération de l'application.

2. Feature - Bouton Action

Nous laissons la possibilité à l'utilisateur de lui simplifier **l'accès** à l'intégration via un seul bouton, mis à disposition via VS Code, ce qui va lui éviter de lancer un invite de commande. Cette feature va utiliser l'extension "**VsCode Action Buttons"** de ***Seun LanLege*****,** qui doit être installée dans le VS Code de l'utilisateur.

3. Feature - Intégration Automatisée

Pour permettre une meilleure expérience aux UI/UX designers qui vont utiliser notre solution, ils pourront, après avoir fini de définir leurs scénarios, exécuter le script *build.sh* suivi d'un paramètre *nomFichier.alc*.

Ce script va générer les composants nécessaires et le fichier *app.js*, avant de créer un nouveau projet ayant le nom du scénario. De plus, le script va mettre en place le *package.json*, les données utilisées par les widgets et les fichiers issus de notre *generator*. Finalement le script lancera l'application.

4. Feature - Snippets

L'utilisateur a aussi la possibilité d'utiliser des snippets fournis qui vont générer un **squelette** permettant à l'utilisateur de pouvoir naviguer dessus via la touche *tab* et de remplir ses champs un par un, sans pour autant les écrire et donc éviter les erreurs de syntaxe. Nous avons donc défini un snippet par défaut qui permet la génération du squelette de l'application entière avec les éléments qui sont obligatoires. En effet, proposer des squelettes de code dès lors que l'utilisateur renseigne un mot clé permet d'atténuer la cacophonie potentielles du DSL externe.

Nous avons également des snippets définissant une partie de notre application et générer le **DSL** script correspondant. Où voici une liste de certains de nos snippets définissant les éléments non obligatoires de notre app qui peuvent être ajoutés.

Voici une liste :

Mot clé	Élément généré
app	Squelette application
header ou head	Header
platform	Platform
ww	WidgetWrapper
theme	Theme avec des couleurs
page	Page (onglet)
plug	Plugin
colors	Color(s)
hide	Hidden Widget

Nous avons également implémenté plusieurs fonctionnalités orientées domaine.

5. Feature - Responsiveness

La responsiveness était, pour nous, un point clé pour rendre l'expérience utilisateur satisfaisante, que ce soit côté designer ou de côté de l'utilisateur final.

En effet, il est difficile de faire un site qui convient à toutes les plateformes qui peuvent l'utiliser. Quand on parle de plateforme, on parle en fait d'appareils ayant des tailles d'écran différentes.

Ainsi, il nous a semblé évident de développer cette feature pour que le designer puisse concevoir simplement des pages adaptées à différents appareils. Nous avons choisi pour cela, de laisser au designer la possibilité de définir, pour plusieurs plateformes, la présentation des widgets ligne par ligne.

Grâce à cela, il a le choix du nombre de widgets à afficher selon la taille de l'écran pour chaque ligne d'une page. De plus, le designer a également le possibilité d'afficher différents widgets plus adaptés à certaines plateformes plutôt qu'à d'autres.

6. Feature - Navigabilité Popup

La partie popup a été mise en place pour permettre au UI/UX designer de donner plus de précision sur un widget sans devoir tout mettre sur la page principale. En effet, il arrive qu'un simple widget puisse manquer d'informations et la possibilité d'ajouter une pop-up permet de préciser simplement les informations de base, sans alourdir la page.

7. Feature - Navigabilité Onglets

L'implémentation des onglets est surtout pour donner une meilleure répartition de nos informations, car, parmi les choses les plus importantes pour un UI/UX designer, est la simplicité et la compréhension de son site. Celà passe parfois par la séparation de l'information, pour que l'utilisateur final ne soit pas envahi de trop nombreuses informations.

8. Feature - Gestion des données statiques

Nous avons mis en place un système permettant de peupler l'UI lorsque le designer crée son application avec notre DSL. Ces données statiques sont découplées de notre DSL afin que l'utilisateur ne renseigne pas ses données lorsqu'il définit son UI au travers de notre script DSL (fichier d'extension `.alc`). Pour avoir une intégration plus complète nous avons lié les noms des types de données avec l'id des widgets permettant ainsi d'avoir un affichage dynamique des données.

9. Feature - Mode accessibilité

Nous avons mis en place, dans la grammaire de notre DSL, le fait d'avoir des modes d'accessibilités. À notre stade de développement, nous permettons à l'UX designer l'activation (ou non) du mode d'accessibilité **Light/Dark Mode** au sein de l'application.

Si ce mode est défini dans l'application, le designer pourra choisir l'emplacement d'un bouton permettant de basculer entre le thème lumineux et le thème sombre. Nous avons également permis d'étendre facilement notre DSL dans l'éventualité où nous devions ajouter d'autres modes d'accessibilité (par exemple agrandir le texte pour la vision réduite ou utiliser certaines couleurs et contrastes plus lisibles).