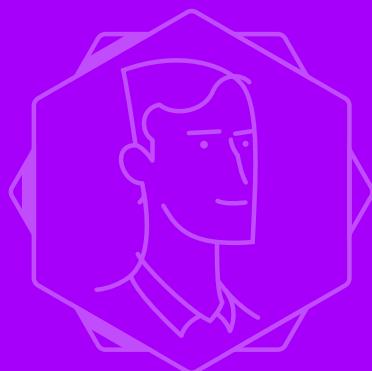


BENJAMIN GABOURY
formation Développeur web et mobile



RAPPORT DE STAGE

ACCENTURE TECHNOLOGY SOLUTIONS / Nantes
10 février au 5 juin 2020

Le Liberty - Liquid Studio
68, Boulevard Marcel Paul
Saint-Herblain
44800 Nantes

SOMMAIRE

PARTIE 1	Introduction	4
1.	Profil du stagiaire	4
2.	L'environnement du stage	4
PARTIE 2	L'entreprise	5
1.	ACCENTURE dans le monde	5
2.	ACCENTURE en France et à Nantes	6
3.	Le Liquid Studio de Nantes	6
PARTIE 3	Le contexte	7
1.	Présentation du projet	7
2.	Les problématiques	10
3.	Les besoins	10
4.	Les spécifications techniques	11
PARTIE 4	Le projet	14
1.	Documentation	14
2.	Lancement du projet	16
3.	Phase de développement	21
4.	Développement d'une librairie Angular	31
5.	Les vulnérabilités de sécurité	36
PARTIE 5	Conclusion	38
1.	Analyse du stage	38
2.	Compétences acquises	38



Remerciements

Je tiens tout d'abord à remercier la WILD CODE SCHOOL :

- les campus managers, **Cécile Ménard** et **Lucille Guillet**, qui m'ont donné l'opportunité de suivre la formation de *Développeur web*,
- ainsi que les formateurs, **Simon Philouze**, **Julien Duhem** et **François Doussin**, qui ont assuré un suivi sans faille et se sont toujours montrés disponibles pour des conseils avisés.

Tout au long des 5 mois de formation, ils ont su nous apporter la motivation et le bagage nécessaire pour nos premiers pas en entreprise en tant que développeur.



Au sein d'ACCENTURE, je remercie chaleureusement toutes les personnes qui m'ont apportées leur aide précieuse pendant mon stage :

- **Olivier Demarez**, mon tuteur de stage, qui a assuré un suivi régulier, même pendant la période de télétravail et qui m'a permis de ne jamais être seul face à mes lignes de code,
- **Tanguy de Lignières** et **Maxime Bureau** pour leurs retours et encouragements, notamment après chaque démo,
- **Raphaël Hascoët**, lead sur le projet, pour ses précieux conseils et son aide tout au long du stage,
- **Yohann Perraud** pour sa patience et sa disponibilité lors de nos échanges Teams,
- **Nathanaëlle Guy** qui m'a précédée sur le projet en tant que stagiaire et a assuré le suivi en me donnant de précieuses indications,
- **tous les membres du Liquid Studio** pour leur disponibilité et leur envie de partager leurs connaissances et leurs expériences sur des domaines variés.

Je tiens également à remercier **Anne-Frédérique Chagon** et **Gaëtan Guillot** qui m'ont proposé ce stage suite à notre entretien lors d'un job dating au centre de formation. Plus généralement, je remercie ACCENTURE TECHNOLOGY pour son accueil et pour m'avoir permis d'effectuer ce stage.



INTRODUCTION

1. Profil du stagiaire

Motion designer et graphiste 3D, plusieurs projets sur lesquels j'ai travaillé, m'ont montré l'importance du métier de développeur, notamment pour l'**interactivité**.

Pour PublicisActiv, j'ai notamment animé des illustrations pour un site web, qui se déclenchaient au passage de la souris. Cette opération s'est faite grâce à du JavaScript. Mais n'ayant pas alors les connaissances nécessaires, cette partie a été confiée à un développeur. J'ai donc décidé d'enrichir mes compétences en apprenant la programmation.

Après avoir suivi plusieurs cours en e-learning (Coursera, Udacity, SoloLearn), je me suis vite rendu compte de l'importance d'avoir des formateurs compétents pour acquérir des connaissances solides.

De septembre 2019 à février 2020, j'ai donc suivi la formation de **Développeur web & mobile** à la Wild Code School de Nantes dans la promo «JavaScript / React». L'apprentissage s'est basé sur le développement de sites internet qui utilisent HTML, CSS, JavaScript (React / Node.js), ainsi que des outils indispensables comme Visual Studio Code, Git, Github et les méthodes agiles.



Motion design pour le 3^e projet à la Wild Code School

2. L'environnement du stage

A l'issu d'un entretien avec **Anne-Frédérique Chagnon** (responsable RH) et **Gaëtan Guillot** lors d'un job dating au centre de formation, j'ai été recontacté pour un 2^e entretien avec **Olivier Demarez**, responsable du Liquid Studio. Il m'a présenté plus en détails les types de projets qui y étaient développés ainsi que le Liberty, le bâtiment d'ACCENTURE TECHNOLOGY à Nantes.

Après ces différentes rencontres, Anne-Frédérique Chagnon m'a proposé un stage dans l'équipe du **Liquid Studio** que j'ai accepté pour plusieurs raisons :

- ACCENTURE est un groupe à dimension internationale où la pluralité des projets et des profils permettent un apprentissage d'une grande richesse
- Les technologies utilisées sur le projet étaient nouvelles pour moi (Angular / Typescript / JHipster) et m'ont permis de parfaire mon apprentissage
- L'opportunité de travailler au Liquid Studio où sont développées des applications en AR, VR, MR et XR



L'ENTREPRISE

1. ACCENTURE dans le monde

ACCENTURE est une entreprise internationale de conseil et de technologies. Elle tient ses origines d'un département informatique d'Arthur Andersen, créé en 1913 sous le nom Andersen, DeLany & Co. ACCENTURE opère dans 120 pays et est constituée d'environ 500 000 collaborateurs, ce qui en fait la plus grosse entreprise de services du numérique.

Dans le monde aujourd'hui, elle est constituée de 6 grandes marques qui conseillent et accompagnent les plus grandes entreprises mondiales :

- **Strategy** - définition de nouveaux axes de croissance
- **Consulting** - accompagnement et analyse pendant la transformation des clients
- **Digital** - exploitation du potentiel du digital en créant de nouveaux leviers
- **Operations** - gestion des processus métier
- **Security** - services de cybersécurité
- **Technology** - maîtrise d'œuvre (centre de Nantes)



Enfin, les clients d'Accenture sont classés parmi 5 grandes classifications d'industries :

- **Ressources** - Energie
- **Communication** - Média, Télécom
- **Financial Services** - Banques et assurances
- **Health Public & services** - Hôpitaux, Services publics
- **Products** - Produits



2. ACCENTURE en France et à Nantes

ACCENTURE compte 7 000 employés en France répartis dans 5 centres : Paris, Nantes, Lyon, Toulouse et Sophia Antipolis.

Le centre de Nantes est constitué de 750 employés, ce qui en fait le 2e bureau après celui de Paris. Il a été fondé en 2003 et le Liberty à Saint Herblain a été aménagé en 2014. Gaël Garandeau en est actuellement le Directeur exécutif.

Une collaboration étroite existe avec les centres de Casablanca et de l'île Maurice que l'on nomme le **French Cluster**. Elle est principalement liée à la facilité de communiquer dans la même langue et au faible décalage horaire entre les pays. Le centre de Nantes est divisé en **4 intelligent platforms** pour la répartition des projets, les studios intervenant sur toutes ces plateformes :

- IPS SAP
- IPS SalesForce
- ICI (infrastructure)
- IES (Intelligent Engineering Services) Développement

On peut y ajouter les plateformes secondaires IMS de gestion de projet et PCF pour le support. A date, 70 projets sont en cours dans le centre, gérés par 2 à 120 personnes.



Le showroom du Liquid Studio à Nantes

3. Le Liquid Studio de Nantes

Avec les Liquid Studios, ACCENTURE propose un développement rapide pour permettre aux entreprises de se construire en adoptant technologies numériques, compétences et méthodes de travail. **Des logiciels et applications sont prototypés** par les Liquid Studios pour les aider à atteindre leurs clients et à pénétrer de nouveaux marchés.

Le Liquid Studio de Nantes est une force de proposition auprès des clients sur des technologies émergentes, comme par exemple la réalité augmentée qui est très présente dans le studio. Quand un projet est lancé, c'est généralement pour proposer un MVP (most valuable product) qui servira de base pour une mise en production future. Les projets sont donc très variés, tant en terme de technologies utilisées que de finalité, et sont la plupart du temps prévus sur des durées de développement relativement courtes.

Le Liquid Studio de Nantes est constitué d'un grand espace de travail, ainsi que d'un showroom et de différentes salles de meeting permettant aux clients et aux équipes d'ACCENTURE de se retrouver autour de sujets sur l'innovation. Le manager et responsable du Liquid Studio est **Olivier Demarez**. Il est y accompagné d'une dizaine de collaborateurs, comprenant des managers, des chefs de projets et des développeurs.



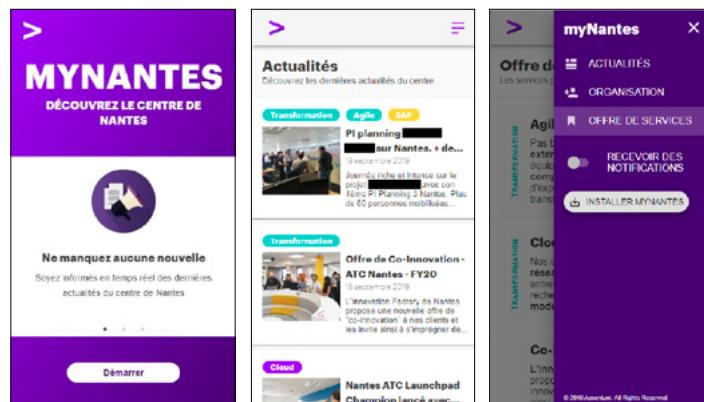
LE CONTEXTE

1. Présentation du projet

Le Liquid Studio de Nantes a développé une **PWA** qui fonctionne en mode pilote, *My Nantes*.

Cette application interne à ACCENTURE est un outil de communication utilisé par des managers pour présenter les dernières nouveautés de leur site à leurs collègues. Elle a 3 grandes fonctionnalités :

- le post d'actualités,
- la présentation de l'organisation du centre,
- et la présentation des offres et services du centre



Ecrans de MyNantes



Une **PWA** (Progressive Web App) combine les fonctionnalités offertes par les navigateurs avec les avantages des applications mobiles. Une PWA se consulte comme une page web via un URL sécurisé et offre le même type d'expérience utilisateur que les applications mobiles, mais sans les inconvénients majeurs des App Stores, tels que l'utilisation importante de mémoire, le développement spécifique en fonction des plateformes iOS et Android.



LE CONTEXTE

Le front-end de *MyNantes* est développé avec *React*, et le back-end et l'hébergement par *Strapi*. Le CMS *Strapi* a pour particularité d'être headless et open source. Il ne génère aucune mise en forme de l'application finale comme le ferait par exemple *Wordpress*. Il fournit simplement une interface permettant de gérer la partie base de données. Le CRUD (Create Read Update Delete) est généré automatiquement, et les données sont stockées et protégées sur les serveurs de *Strapi*.

La volonté sera d'étendre l'application *My Nantes* en la proposant dans un premier temps à 2 autres sites ACCENTURE, Casablanca et Maurice. En l'état, le CMS utilisé ne permet pas de gestion multi sites.

L'équipe du Liquid Studio a donc cherché un outil existant permettant de palier à ce problème, mais aucune solution satisfaisante n'étant trouvée, il a été décidé de développer un CMS propre. L'objectif était de créer un **CMS custom** en faisant abstraction de *MyNantes*, pour en permettre une utilisation généralisée et en adaptant le concept pour des clients aspirant à ce besoin de gestion.

Plusieurs axes étaient à prendre en considération, le CMS devant s'articuler autour de 3 piliers :

- 1. HEADLESS :** ne permet pas de développer un site web mais d'administrer les données
- 2. GESTION MULTI SITES :** plusieurs sites peuvent exploiter la même API
- 3. « PHOENIXING » en cas de crash :** il doit être possible de recréer l'application avec les données



Un **CMS** (Content Management System ou Système de Gestion de Contenu) regroupe une catégorie de logiciels qui permettent de concevoir, gérer et mettre à jour des sites Web ou des applications mobiles de manière dynamique. Pour pouvoir être définis comme CMS, les logiciels doivent pouvoir être utilisés simultanément par plusieurs individus, proposer une chaîne de publication de contenu et permettre de gérer séparément la forme et le contenu.

exemples : *Wordpress*, *Drupal*...

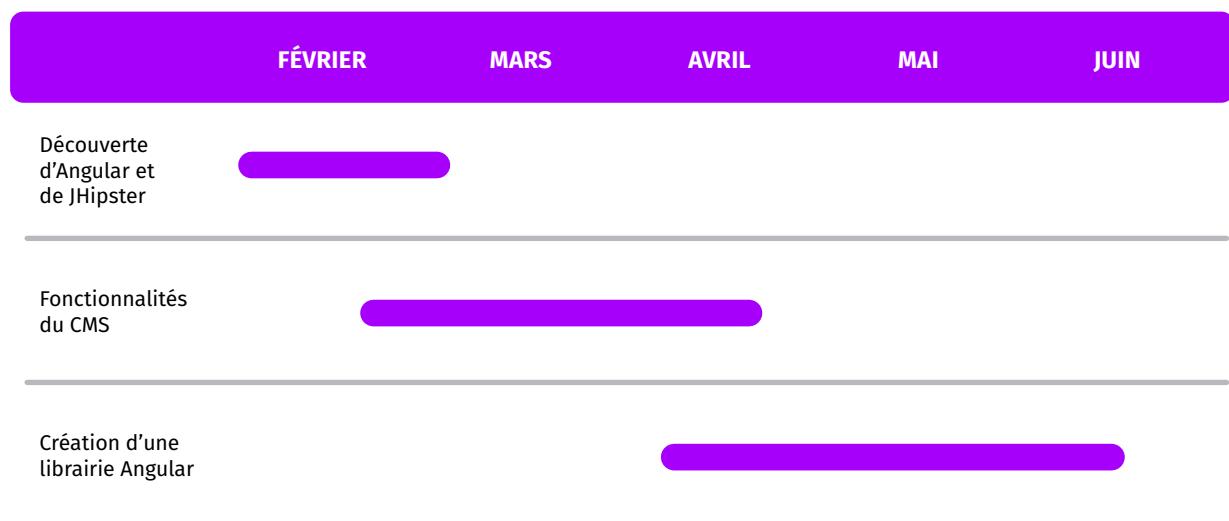


LE CONTEXTE

Pendant mon stage chez ACCENTURE TECHNOLOGY, j'ai travaillé sur le développement du CMS dont la phase de création a été initiée par **Raphaël Hascoët** et **Nathanaëlle Guy**.

Dans un premier temps, j'ai commencé par la découverte de *JHipster* et de son environnement Java, ainsi que d'*Angular* que j'ai pris en main petit à petit. Puis, j'ai travaillé sur les fonctionnalités inhérentes à la page d'édition de schémas.

La dernière partie de mon stage a été consacrée à la création d'une librairie *Angular*, très utile pour éviter la duplication de code et minimiser le temps de production. Elle permet de créer des composants réutilisables dans plusieurs projets ainsi que la procédure de publication et de mises à jour NPM.



2. Les problématiques

Dans une volonté de faire évoluer *MyNantes*, cette application sera proposée dans un premier temps aux deux autres centres du **French Cluster**, Casablanca et Maurice, puis par la suite, à tous les centres Accenture France.

Au commencement du projet, l'utilisation de *Strapi* a été envisagée car ce CMS permet de créer facilement, de mettre en ligne et de gérer des API.

À mi-chemin entre un framework *Node.js* et un headless CMS, il permet d'économiser des semaines de développement. Grâce à son système extensible de plugin, il propose de nombreuses fonctionnalités : panel d'administration, authentification et gestion des permissions, gestion de contenu, générateur d'API...

Cependant, n'ayant aucun moyen de gérer le multi-sites via *Strapi*, il faut dupliquer l'application et avoir donc une API par centre. Outre la multiplication en terme d'hébergement, cela empêche, par exemple, de poster directement une news destinée à plusieurs centres, autrement dit, pas de moyen simple de partager les données. Cette option a donc été écartée.



Strapi, headless CMS

3. Les besoins

L'objectif est de créer un **CMS custom** mais en faisant abstraction de *MyNantes* pour ouvrir l'horizon à une utilisation généralisée. L'idée est d'adapter le concept pour des clients ACCENTURE pouvant aspirer au même besoin de gestion.

Ce CMS devra s'appuyer sur 3 grands piliers :

1. HEADLESS : Garder l'approche de *Strapi* car notre CMS ne générera pas de mise en forme. Il devra permettre un accès Administrateur pour l'administration globale, et un accès Utilisateur pour l'ajout et la modification de données

2. GESTION MULTI SITES : Une seule API pour toute la gestion d'un groupe de sites. L'administrateur doit pouvoir ajouter un site, un utilisateur qui a les bonnes autorisations doit pouvoir choisir le site sur lequel il opère un ajout ou une modification

3. PHOENIXING (terme « made in » Accenture) : En cas de crash, la structure et la configuration doivent pouvoir se remettre en place facilement. Il faut donc gérer le système de sauvegarde des données



4. Les spécifications techniques

Afin d'évoluer sur des technologies éprouvées en interne et fiables pour la pérennité du projet, l'application a été lancée de la façon suivante :

BACK-END



p. 10



BASE DE DONNÉES



PostgreSQL



mongoDB

FRONT-END



p. 10

TypeScript



ENVIRONNEMENT



p. 11



p. 11



Azure DevOps

p. 11





JHipster

JHipster est un générateur d'applications à l'origine créé par un français, Julien Dubois. Il permet d'avoir côté serveur, une pile Java (à l'aide de *Spring Boot*) et côté client un frontal web adaptatif (avec *Angular* et *Bootstrap*).

Les avantages :

- Son installation en lignes de commandes est simple et rapide
- Il permet de gagner un temps considérable sur le développement d'une application avec la génération des entités, des contraintes, ou encore des relations entre entités
- Il fournit un système d'authentification sécurisé sous forme de « bearer token » (un JSON web token dont le rôle est d'indiquer que l'utilisateur qui accède aux ressources est bien authentifié, par envoi du token au serveur qui le valide et ce, à chaque requête)
- Il gère la création automatique des ID pour tous les éléments créés en base de données
- Il offre une interface graphique assez pratique pendant la phase de développement
- Enfin, et surtout, il offre une API avec des requêtes HTTP générées automatiquement, un CRUD complet donc, et la possibilité de créer des requêtes personnalisées. C'est donc un outil très puissant qui présente néanmoins quelques inconvénients
- Une prise en main complexe, il est nécessaire de comprendre et d'adapter du code généré
- Spécialement conçu pour les développeurs, son interface graphique est efficace pour un développeur, beaucoup moins pour un utilisateur classique
- Nous voudrions nous en abstraire en cas de besoin ou d'envie de changer d'API on ne veut pas être dépendant de *JHipster*



Angular

Pour les besoins du CMS, l'interface graphique est développé avec *Angular*, afin de ne pas être dépendant de *JHipster* et de pouvoir créer des écrans, pratiques pour un administrateur et aussi pour les utilisateurs.

Angular est la technologie front-end la plus utilisée chez Accenture, ce qui permettra au projet de perdurer dans le temps, même si les développeurs changent régulièrement de technologie. Géré par Google, son suivi est donc solide.

Le langage *TypeScript* permet un développement plus facile et plus stable, et offre des fonctionnalités utiles et robustes (le typage strict, les interfaces,...). Ce framework permet de développer une application dynamique et responsive





Visual Studio Code

Cet **éditeur de code** a pour avantage d'être open source. Il supporte de nombreux langages, dont l'ensemble de ceux utilisés pour le développement de notre CMS. Il permet également l'installation d'extensions facilitant le travail du développeur. Voici les 4 qui ont été le plus utiles sur le projet :

- **Prettier** formate le code automatiquement à chaque sauvegarde et permet ainsi une meilleure lisibilité et un meilleur travail collaboratif,
- **Git Graph** et **GitLens** sont des outils de comparaison de versions,
- **TypeScriptHero** est un outil d'aide au code pour TypeScript,
- **Visual Studio IntelliCode** propose des suites logiques de code à l'autom complétion.



Git

Git est un **logiciel de gestion de versions** décentralisé. Il permet de travailler sur des branches indépendantes, de partager ses fichiers, de conserver les versions antérieures de ses fichiers, pour pouvoir à tout moment revenir en arrière ou encore sauvegarder des versions sans les partager. C'est un outil extrêmement pratique pour le travail en équipe.



Azure DevOps

Azure DevOps Il s'agit d'une plateforme web de **gestion de projets et développement de logiciels**.

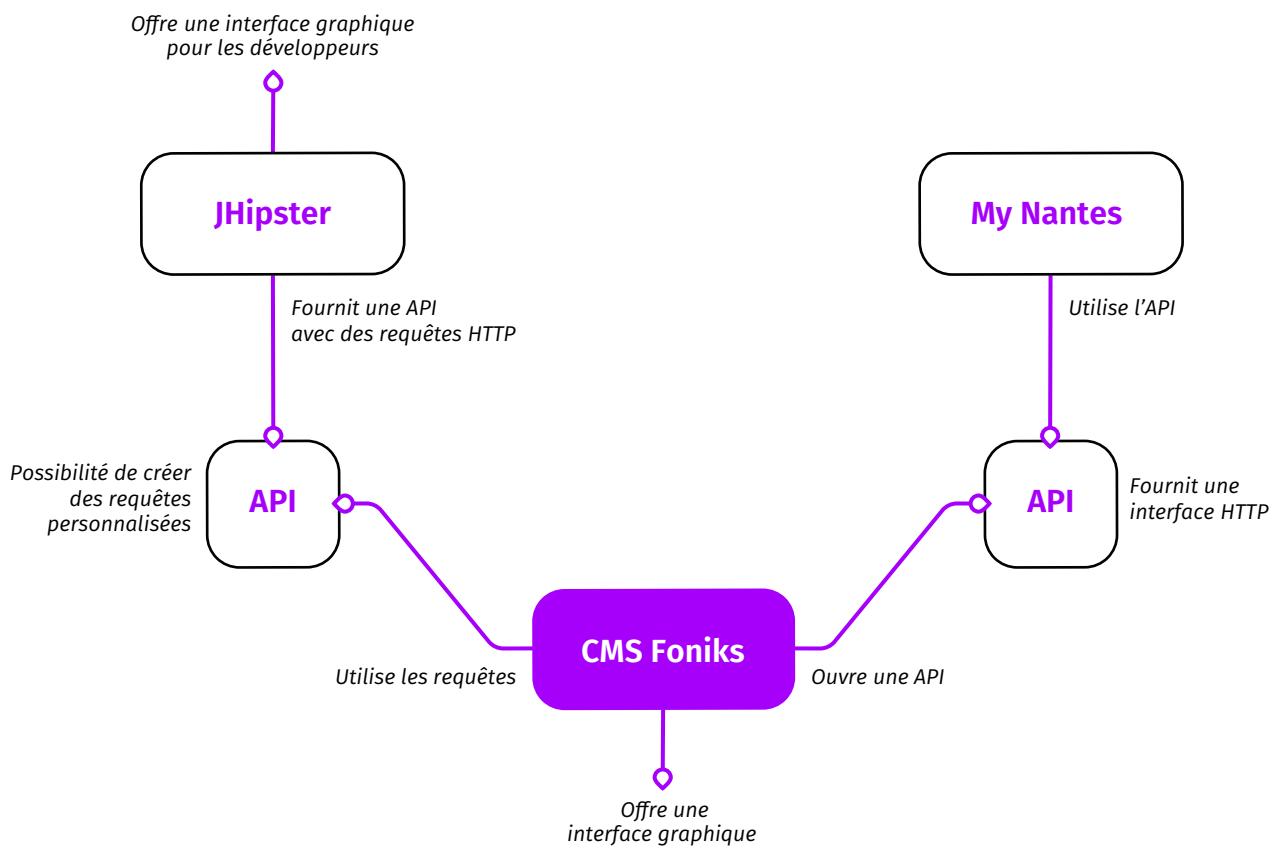
Elle nous permet notamment pour notre phase de développement de configurer notre dépôt Git, et de faire passer le code poussé (« push ») dans un chaîne d'intégration continue.



LE PROJET

1. Documentation

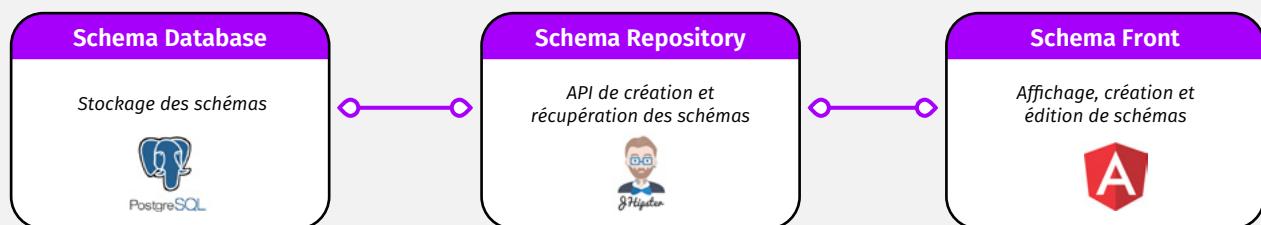
Le CMS Foniks dans l'ensemble de l'architecture



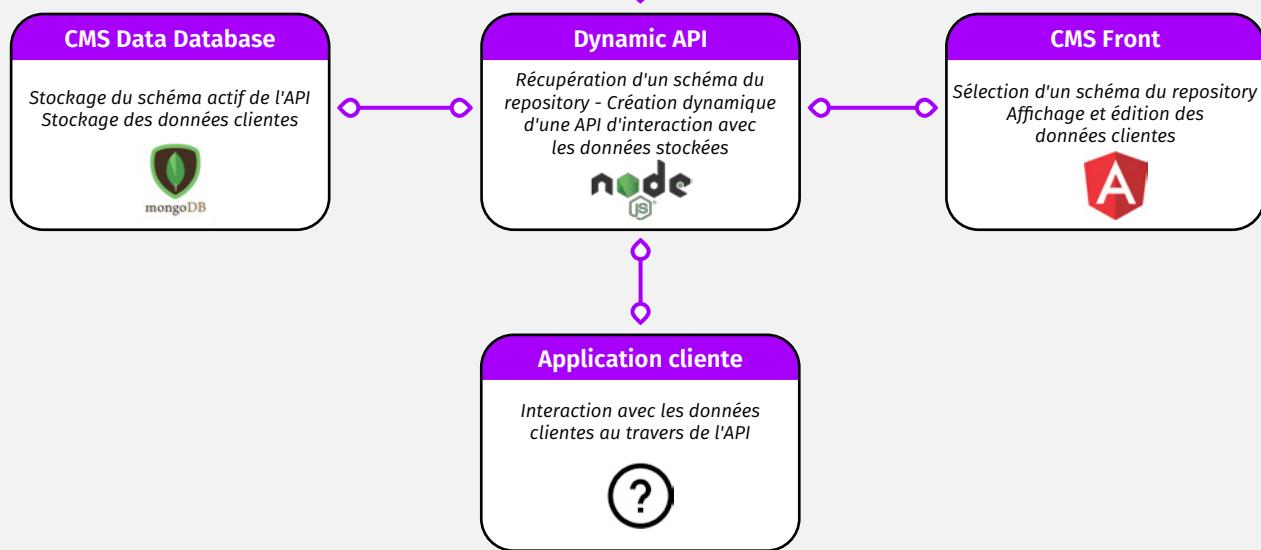
LE PROJET

Architecture du CMS Foniks

Provider Side



Client Side



2. Lancement du projet

Le back-end avec JHipster

Après l'installation du JDK de Java, celle de JHipster se fait par l'invite de commandes :

```
npm install -g generator-jhipster
```



À partir de là, il est possible de lancer un nouveau projet avec la commande **jhipster** dans un dossier préalablement créé dans le workspace. En référence à l'un des 3 piliers du projet, le nom choisi pour le projet est *Foniks* (signifiant « phoenix »). Plusieurs questions s'enchaînent alors sur les différentes options possibles, tout restant bien sûr modifiable par la suite. Voici les plus significatives :

Which type of application would you like to create?

2 types proposés : monolithique et microservices.

Le choix se porte sur une version **monolithique** car l'application est simple avec un objectif unique, qui ne nécessite pas pour l'instant l'ajout de complexité liée à une architecture microservices.

Which type of database would you like to use?

SQL répond aux besoins de l'application et est habituellement utilisé sur les projets JHipster développés chez Accenture.

Which production database would you like to use?

PostgreSQL est choisi pour les mêmes raisons que la question précédente. Ce choix s'avère très pertinent puisque ce système de gestion de base de données relationnelles est gratuit open source, et il est supporté par tous les principaux systèmes d'exploitation et langages de programmation. Il est fiable et performant (intégrité des données et gestion des requêtes), avec également une documentation et support de très bonne qualité.

Which development database would you like to use?

L'option la plus simple est choisie : **H2 running in memory**. La perte de données lors du reboot du serveur n'étant pas un désagrément majeur pendant la phase de développement

Would you like to use Maven or Gradle?

Maven et Gradle sont des builder d'application qui incluent un outil de gestion de dépendances (version de librairies). Nous choisissons **Maven**, la solution d'Apache spécialement conçue pour Java.

Une fois cette phase terminée, l'application est créée, puis accessible en local.



Comment JHipster fonctionne ?

Nous pouvons générer des entités, y ajouter des attributs, leurs types et leurs contraintes, ainsi que créer des relations entre les différentes entités.

Cela est possible de 2 façons :

- ① en ligne de commande, entité par entité
- ② à l'aide d'un outil graphique proposé par *JHipster*, le *JDL Studio*, qui génère un fichier implantable en une seule ligne de commande.

Cet outil, très bien conçu, permet de gagner beaucoup de temps. L'écriture est simplifiée à son maximum, et le code est visualisable directement à l'aide du diagramme de classe généré sur le même écran :

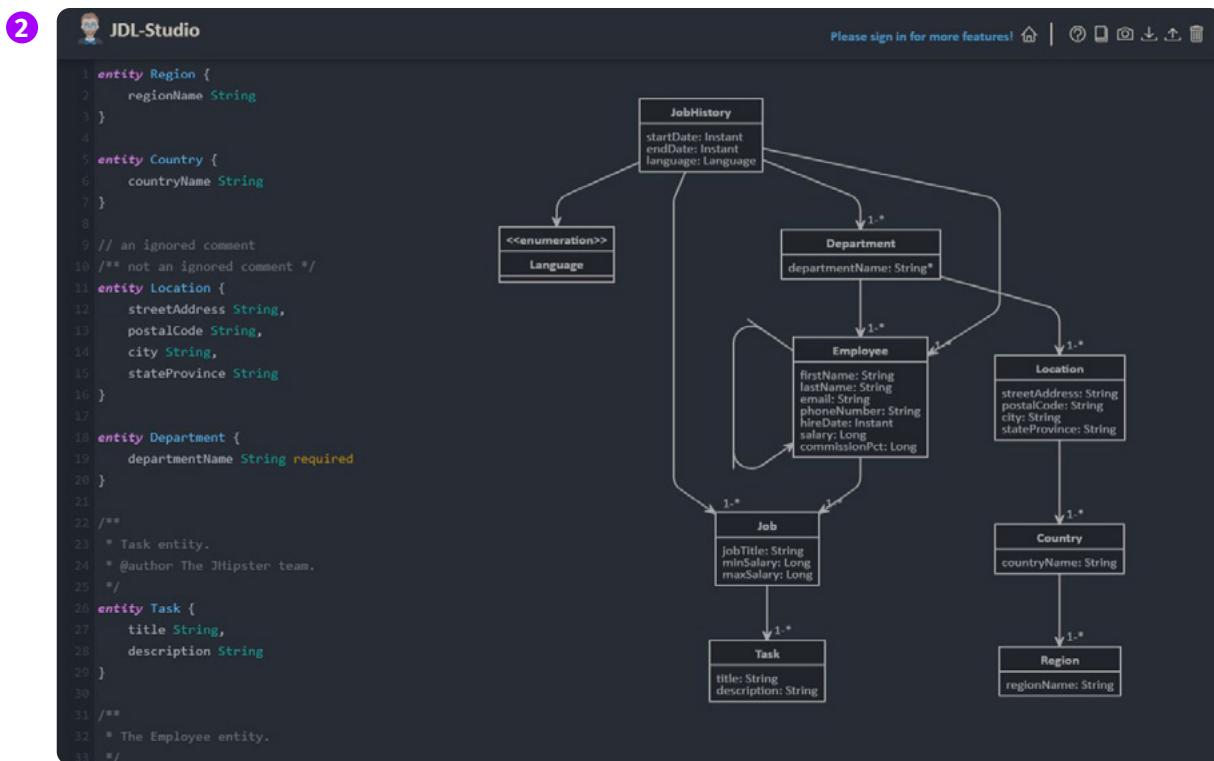


Diagramme de classes sur le JDL Studio



Le diagramme de classes

Pour pouvoir construire la structure, nous devons décomposer ce que représente un schéma de base de données avec pour objectif principal un système fonctionnel de création de schémas laissant ouvertes des possibilités d'évolution, dont le multi sites.

SCHEMA

- Un schéma représente la configuration logique de tout ou partie d'une base de données. Via le CMS Foniks, l'admin pourra en ajouter, accorder des droits à des utilisateurs qui pourront les utiliser.
Un schéma doit avoir un nom.

DATATYPE

- Un schéma est défini par une liste de types de données (datatype). Ce datatype doit avoir un nom.

ATTRIBUTE

- Un datatype est constitué d'un ensemble d'attributs. Un attribut doit avoir un nom, un type (à choisir dans une liste de types), ainsi qu'une ou plusieurs contraintes.

CONSTRAINT

- Une contrainte sera donc liée à une liste de types de contraintes, et pourra avoir une valeur.

LINK

- Un datatype peut être lié à un autre datatype. Le type de lien sera défini par une liste de types de lien.



Récupération du schéma d'un utilisateur

JHipster a généré une arborescence parfaitement. On peut faire des tests facilement, via l'interface graphique de JHipster ou via l'interface de la base de données H2. Il est possible d'ajouter, supprimer, modifier et visualiser dans nos 5 entités.

Pour commencer, on génère un schéma à l'aide du format JDL : **l'upload d'un fichier en format JSON**. L'utilisateur sera toujours de type **admin**, donc familier de méthodes plus techniques.



Un **JSON** (JavaScript Object Notation) est un format de données textuelles dérivé de la notation des objets du langage JavaScript. Il permet de représenter de l'information structurée.

Le choix du format JSON présente de nombreux avantages :

- C'est un format compréhensible par tous (humain et machine)
- Il ne dépend d'aucun langage, et permet une intégration simplifiée nativement par Javascript que nous prévoyons d'utiliser,
- Il peut stocker tous nos types de données,
- Il permettra d'extraire et de stocker les schémas en tant que fichier de sauvegarde,
- Il permet de tester rapidement sur des schémas variés et plus ou moins complexes, qui seraient longs à mettre en place autrement

C'est donc la meilleure façon d'écrire le schéma, pour que ce soit le plus clair et le plus simple possible, en partant du principe qu'une documentation précise sera fournie pour aider l'utilisateur.

Même logique exposée précédemment :

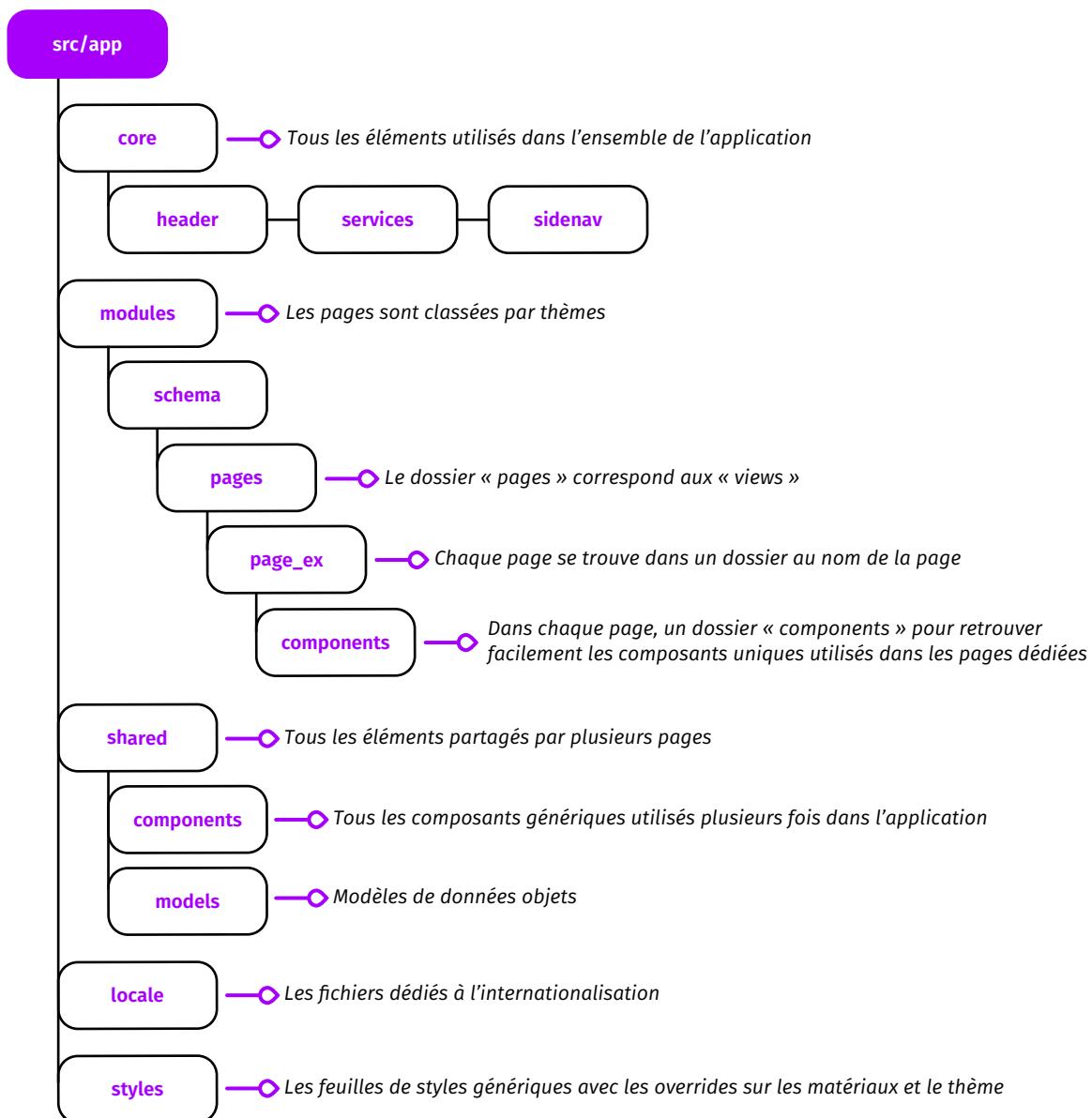
Commencer par le schéma, constitué d'un tableau de datatypes eux mêmes constitués, éventuellement d'un tableau de liens, et d'un tableau d'attributs, eux mêmes pouvant être constitués d'un tableau de contraintes.

Le schéma est composé de 2 datatypes **Client** et **Produit** ainsi que quelques attributs, un lien et deux contraintes (une avec valeur, l'autre sans).



Le front-end avec Angular

La partie front-end a été créée avec *Angular*. La structure sépare bien les différentes logiques. Elle est pratique et modulable, car elle est susceptible d'évoluer, si besoin.



3. Phase de développement

Il y a **4 grandes étapes** à réaliser pour récupérer et afficher un schéma :

1. Créer un modèle d'objets Typescript (le langage d'Angular pour la manipulation des données)
2. Transformer un fichier JSON (string) en objets Typescript
3. Une 1^{ère} page avec une interface qui upload un fichier JSON
4. Une 2^e page qui affiche le schéma à l'utilisateur

Le modèle d'objets Typescript et la désérialisation

Avec le JSON, il y a un flux de données sérialisées en string qui doivent être transformées en objets Typescript. Voici le processus mis en place avec la classe « Datatype » :

deserializable.ts

```
1 export default interface Deserializable {
2   |   deserialize(input: any): this;
3 }
```

➊ interface « Deserializable » pour pouvoir l'utiliser dans les modèles

datatype.ts

```
1 import Deserializable from "./Deserializable";
2 import { Link } from "./Link";
3 import { Attribute } from "./Attribute";
4
5 export class DataType implements Deserializable {
6   public id?: number;
7   public name: string;
8   public links: Link[] = [];
9   public attributes: Attribute[] = [];
10
11  deserialize(input: any): this {
12    (<any>Object).assign(this, input);
13
14    if (!!input.links) {
15      this.links = input.links.map(link => new Link().deserialize(link));
16    }
17
18    this.attributes = input.attributes.map(attribute =>
19      new Attribute().deserialize(attribute)
20    );
21
22    return this;
23  }
24}
```

➋ la Classe « Datatype » implémente Deserializable

➌ modèle Typescript

➍ méthode de désérialisation

➎ gestion des relations par mapping



L'upload du JSON

Après les modèles, les énumérations et les désérialisations, on peut passer à l'upload du JSON :

- ➊ Un **Service Angular** créé : « createSchemaService.ts »
- ➋ La classe est décorée avec « `@Injectable()` » et l'injecteur du service est défini « `: root` », pour que le service soit accessible dans toute l'application.
- ➌ Le schéma est initialisé comme **observable** et le « `BehaviorSubject` » permet de récupérer sa valeur. On trouve aussi les méthodes nécessaires à la récupération et l'édition du fichier avec un **get** et un **set**.
- ➍ Pour terminer, il y a la méthode de création du schéma qui prend en paramètres une variable de type « `string` ». La méthode `JSON.parse` analyse une chaîne de caractères JSON et construit l'objet décrit par cette chaîne.

➊ CreateSchemaService.ts

```

1 import { Injectable } from "@angular/core";
2 import { Schema } from "../shared/models/Schema";
3 import { BehaviorSubject } from "rxjs";
4
5 @Injectable({
6   //détermine quel injecteur fournira l'injection
7   providedIn: "root"
8 })
9 export class CreateSchemaService {
10   // initialisation du schema comme observable
11   // A Subject is both an observer and observable. A BehaviorSubject a Subject that can emit the current value
12   // permet d'envoyer la valeur du schéma
13
14   private schemaSubject = new BehaviorSubject<Schema>(new Schema());
15   schema = this.schemaSubject.asObservable();
16
17   constructor() {}
18
19   // remplacement du fichier schema.json d'origine par celui importé par l'utilisateur
20   setSchema(newSchema: Schema) {
21     this.schemaSubject.next(newSchema);
22   }
23
24   // retourne le schema
25   getSchema() {
26     return this.schema;
27   }
28
29   // creation du schema en type Objet Schema à partir du fichier json
30   createSchemaFromJson(schemaJson: string) {
31     const newSchema = new Schema().deserialize(JSON.parse(schemaJson));
32     this.setSchema(newSchema);
33   }
34 }
```



L'affichage du schéma

- ➊ Le service « createSchemaService.ts » est injecté dans un composant en paramètre du constructeur.
- ➋ Une méthode permet de réagir par un **event binding** à l'ajout d'un fichier. On vérifie qu'un fichier est bien présent grâce à l'**event target**, puis le fichier est lu de façon asynchrone à l'aide de l'objet **FileReader**. Ensuite, le résultat est envoyé à la méthode du service en le castant en string.
- ➌ Dans le template du composant, **le bind** de la méthode est effectué sur l'**input** du fichier. Il faut également réagir au bouton de validation vers une page affichant le schéma.

ChooseJsonFileComponent.ts

```

➊ 10  constructor(
11    // injection de createSchemaService
12    private creaSchemaService: CreateSchemaService
13  ) {}
14
15  //appel du Service
16  createSchemaFromJson(schemaJson: string) {
17    this.creaSchemaService.createSchemaFromJson(schemaJson);
18  }
19
➋ 20  // réagir à l'ajout du fichier
21  onJsonUpload(event) {
22    const reader = new FileReader();
23
24    if (event.target.files && event.target.files.length) {
25      const [file] = event.target.files;
26      // lecture du fichier
27      reader.readAsText(file);
28
29      // Quand lu, cast en string et envoi du contenu du fichier
30      reader.onload = () => {
31        this.createSchemaFromJson(reader.result as string);
32      };
33    }
34  }
35 }
36

```

chooseJsonFileComponent.html

```

➌ 1 <div class="ChooseFileForm">
2   <p>
3     <label for="file" id="ChooseAfile">Choose a file</label>
4
5     <!-- bindind à la méthode createSchemaFromJson -->
6
7     <input type="file" id="file" (change)="onJsonUpload($event)" />
8     <button id="GoButton">
9       GO
10      </button>
11   </p>
12 </div>

```



La sauvegarde du schéma dans la base de données

Le front-end fonctionnant correctement, on passe au back-end pour voir comment le schéma est sauvegardé dans la base de données. JHipster a généré dans notre package (com.accenture) :

- Un dossier « domain » contenant les classes métier, avec les **getters** et **setters**. Voici un exemple avec l'entité « Attribute », parfaitement générée à partir de notre JDL

domain/attribute.java

```
public class Attribute implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "sequenceGenerator")
    @SequenceGenerator(name = "sequenceGenerator")
    private Long id;

    @NotNull
    @Column(name = "name", nullable = false)
    private String name;

    @NotNull
    @Enumerated(EnumType.STRING)
    @Column(name = "type", nullable = false)
    private BaseType type;

    @ManyToOne(optional = false)
    @NotNull
    @JsonIgnoreProperties("attributes")
    private Datatype datatype;
```

- Un dossier « repository » avec des interfaces donnant accès aux données avec les méthodes du CRUD, où on peut ajouter des requêtes personnalisées en syntaxe JPQL, spécifique à Java. Voici le repository correspondant à la classe « Attribute » :

attribute/Repository.java

```
@Query("select attribute from Attribute attribute where attribute.datatype.id = :datatype")
List<Attribute> findAllAttributesOfDatatype(@Param("datatype") Long datatypeId);}
```

- Un dossier « rest » avec des fichiers **resource** pour chaque classe, utilise les méthodes du repository et gère les routes pour accéder aux données. Voici un exemple de requête personnalisée permettant de récupérer tous les attributs liés à un datatype. Une liste est renournée par un appel à la requête :

web/rest AttributeResource.java

```
@GetMapping("/attributes/datatype/{id}")
public List<Attribute> getAllAttributesOfDatatype(@PathVariable Long id) {
    log.debug("REST request to get all Attributes of Datatype " + id);
    return attributeRepository.findAllAttributesOfDatatype(id);
}
```



Back-end et front-end

Comment sont liés le back-end *JHipster* et le front-end *Angular* ?

Pour utiliser les requêtes **resource**, il faut les traduire afin qu'elles soient comprises par le modèle Typescript. Tout d'abord, on s'abstrait de la base de données. Cela permet d'anticiper les futures évolutions de l'application, notamment si le système de base de données change.

Le **pattern DAO** (Data Access Object) permet de faire le lien entre la couche d'accès aux données et la couche métier de l'application (les classes). Cette classe abstraite « *DataService* » est un service *Angular*. On y précise l'ensemble des méthodes qui devront être implémentées par un service de données, ces méthodes seront celles appelées partout dans l'application. Dans «*data.service.ts* », on trouve un exemple des méthodes à implémenter concernant la classe « *Attribute* » :

data.service.ts

```
abstract addAttribute(attribute: Attribute, datatypeId: number): Observable<Attribute>;
abstract getAttribute(id: number): Observable<Attribute>;
```

2 services sont utiles pour faire le lien avec *JHipster* :

- **JHipsterDataService** qui « extends » **DataService**, pour redéfinir les méthodes de *DataService*. Les requêtes sont envoyées et leurs réponses reçues au travers des données de l'application transformées afin qu'elles suivent les formats attendus par *JHipster* et formalisés par les classes **Payload**. C'est de la sérialisation.
- **JHipsterPayloads** où les données sont formalisées telles qu'attendues par *JHipster*

JHipsterDataService utilise **JHipsterPayloads** pour coller parfaitement au modèle du back-end

jhipster-data.service.ts

```
addAttribute(
  attribute: Attribute,
  datatypeId: number
): Observable<Attribute> {
  return this.getDatatypePayload(datatypeId).pipe(
    concatMap(datatype => this.addAttributeJH(attribute, datatype))
  );
}

getAttribute(id: number): Observable<Attribute> {
  return this.getAttributePayload(id).pipe(
    concatMap(attributePayload => this.getAttributeJH(attributePayload))
  );
}
```



La persistance des données

Le back-end et le front-end pouvant communiquer, un schéma uploadé sur l'application doit permettre la persistance des données. Dans le composant *Angular* gérant l'upload du fichier JSON et son envoi, cela est possible avec un **event** au clic sur le bouton de validation

choose-json-file.component.html

```

1 <div class="ChooseFileForm">
2   <p>
3     <label for="file" id="ChooseAfile">Choose a file</label>
4
5     <!-- bindind à la méthode createSchemaFromJson -->
6
7     <input type="file" id="file" (change)="onJsonUpload($event)" />
8     <button id="GoButton">
9       GO
10    </button>
11  </p>
12 </div>
```

Voyons la méthode appelée **onSubmit**. La méthode du service décrite précédemment est appelée et le routing vers une nouvelle page permet l'affichage du schéma. La méthode **onSubmitError** permet de gérer l'application en cas d'erreur.

choose-json-file.component.ts

```

onSubmit() {
  this.loading = true;
  this.creaSchemaService.createSchemaFromJson(this.loadedJson).subscribe(
    schema => this.router.navigateByUrl("schema/view/" + schema.id),
    error => this.onSubmitError()
  );
}
```

A l'écran, le schéma s'affiche comme prévu :

Schema : MyNantes		
People	News	Organization
lastName : STRING role : INTEGER firstName : STRING	title : STRING description : STRING	orgaName : STRING cityName : STRING groupName : STRING

Schéma « MyNantes » sur le front utilisateur



LE PROJET

L'affichage des données

Des pages ont été créées pour que l'utilisateur puisse charger des fichiers et observer le résultat :

- Une page permet d'uploader un fichier JSON

Create schema with JSON file

schema-format (2).json

OR

Create empty schema

Liste des schémas avec les ID générés

- Une page affiche une liste de schémas avec la méthode **getAllSchemas**,

My Schemas

ID	Name
1051	MyNantes
1052	Test

Liste des schémas avec les ID générés



LE PROJET

- Une page affiche les datatypes du schema. La sélection de schema datatype se faisant par URL, l'utilisateur peut naviguer via les URL avec les id des éléments

The screenshot shows a web interface titled "Schema : MyNantes". It displays three categories: "People", "News", and "Organization". Each category has its own set of datatypes:

- People:** lastName : STRING, role : INTEGER, firstName : STRING
- News:** title : STRING, description : STRING
- Organization:** orgaName : STRING, cityName : STRING, groupName : STRING

At the bottom center is a button labeled "Add a datatype".

Schéma « MyNantes » avec ses différents datatypes

- Une page affiche la liste des attributs et permet d'en créer de nouveaux

The screenshots show the "Attributes" section for the "People" schema.

Top Screenshot: A list view of attributes for the "People" schema. The attributes listed are:

- lastName STRING NOT_NULL, UNIQUE
- role INTEGER
- firstName STRING

Buttons at the bottom include "Add attribute", "Save", and "Delete".

Bottom Screenshot: A detailed edit view for an attribute named "age". The attribute details are:

- Attribute Name: age
- Attribute Type: INTEGER
- Attribute Constraint(s): NOT_NULL

Buttons at the bottom include "Add attribute", "Save", and "Delete".

Page d'édition des attributs



Le schéma dans la base de données

Il y a 2 façons de vérifier que le schéma est dans la base de données :

- 1 sur l'interface graphique de JHipster

ID	Name
1051	Test_db

- 2 sur l'outil de base de données

SELECT * FROM SCHEMA |

```
SELECT * FROM SCHEMA;
+----+----+
| ID | NAME |
+----+----+
| 1051 | Test_db |
+----+----+
(1 row, 1 ms)
```

Edit

Requête SQL sur les schémas

SELECT * FROM DATATYPE |

```
SELECT * FROM DATATYPE;
+----+----+----+
| ID | NAME | SCHEMA_ID |
+----+----+----+
| 1101 | Product | 1051 |
| 1102 | Client | 1051 |
+----+----+----+
(2 rows, 1 ms)
```

Edit

Requête SQL sur les datatypes

SELECT * FROM ATTRIBUTE |

```
SELECT * FROM ATTRIBUTE;
+----+----+----+----+
| ID | NAME | TYPE | DATATYPE_ID |
+----+----+----+----+
| 1151 | name | STRING | 1101 |
| 1152 | name | STRING | 1102 |
| 1153 | age | INTEGER | 1102 |
+----+----+----+----+
(3 rows, 1 ms)
```

Edit

Requête SQL sur les attributs



Les styles de l'application

Il doit y avoir une unité de composition pour tous les éléments (palettes de couleurs, maquette...) L'aspect de l'application est pris en charge par *Angular Material* et du SCSS.

Pour ces différents points, les solutions suivantes ont été mises en place :

- création de templates s'appliquant à toute l'application comme la sidenav et le header
- utilisation des composants *Angular Material* (tableaux, icônes, cards...)
- utilisation du SCSS pour la gestion des couleurs par un thème et pour les composants génériques

Angular Material est un module d'intégration qui permet d'obtenir facilement une interface responsive harmonieuse. Il s'installe en exécutant la commande suivante :

```
ng add @angular/material
```

Le fichier « theme.scss » est créé dans le dossier « styles » et est déclaré dans « angular.json ». Des palettes sont générées à partir des codes couleurs d'Accenture avec **Material Design** :

```
$custom-primary: (      $custom-accent: (  
  50: □#eee0f7,  
  100: □#d6b3ec,  
  200: □#ba80e0,  
  300: □#9e4dd3,  
  400: □#8a26c9,  
  500: □#7500c0,  
  600: □#6d00ba,  
  700: □#6200b2,  
  800: □#5800aa,  
  900: □#45009c,  
  A100: □#dcc7ff,  
  A200: □#bb94ff,  
  A400: □#9b61ff,  
  A700: □#8b47ff,  
  50: □#f7f7f7,  
  100: □#ececfc,  
  200: □#dfdfdf,  
  300: □#d2d2d2,  
  400: □#c8c8c8,  
  500: □#bebebe,  
  600: □#b8b8b8,  
  700: □#afafaf,  
  800: □#a7a7a7,  
  900: □#999999,  
  A100: □#ffffff,  
  A200: □#ffffff,  
  A400: □#ffd6d6,  
  A700: □#ffbdbd,
```

L'**import** et l'**include** est fait dans chaque composant nécessitant le thème :

```
@import "~@angular/material/theming";  
@include mat-core();
```

Puis, les imports de tous les composants sont déclarés où il est prévu d'utiliser ce thème

```
@import "../app/shared/components/custom-mat-icon/custom-mat-icon.component";  
@import "../app/core/sidenav/link-sidenav/link-sidenav.component";  
@import "../app/shared/components/custom-mat-table/custom-mat-table.component";
```



4. Développement d'une librairie Angular

De nombreux avantages

L'utilisation des librairies est devenue une pratique incontournable dans le monde du développement. Elles permettent d'éviter la duplication de code, de minimiser le temps et le coût de la maintenance. Dans *Angular*, l'intégration de « *ng-packagr* » permet de générer et de construire facilement des librairies.



Les composants créés dans la librairie sont réutilisables dans plusieurs projets avec la procédure de publication et de mise à jour avec NPM.

Les différentes étapes

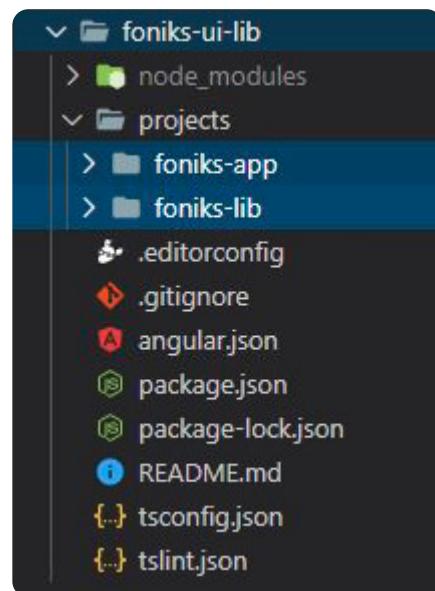
- 1 Tout d'abord, il faut créer un espace de travail vide dans lequel se trouveront la librairie et l'application permettant de tester en local les composants :

```
ng new foniks-ui-lib --create-application=false
crée un espace de travail, sans app

ng g library foniks-lib --prefix=nk
crée une librairie avec un préfixe qui permet
de créer des composants faciles à distinguer

ng g application foniks-app
crée une application Angular

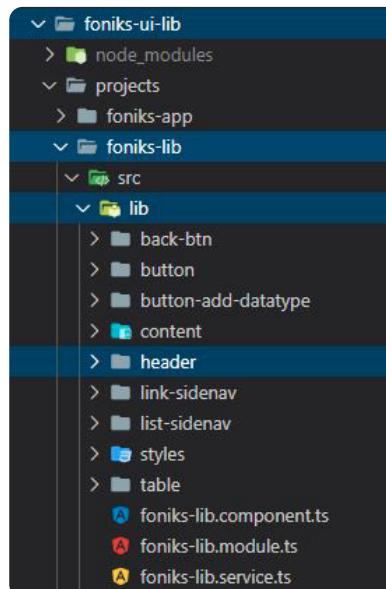
ng serve my-app
```



LE PROJET

- 2 Maintenant, on peut créer dans la librairie un composant qui sera utilisé dans les 2 fronts du CMS, par exemple un header :

```
ng generate component header --project foniks-lib --dry-run  
ng generate component header --project foniks-lib  
crée un component « header » dans la librairie « foniks-lib » et  
« dry-run » simule l'exécution sans apporter de modification aux fichiers existants
```



- 3 Dans « package.json », plusieurs scripts sont créés pour faire le build et le package de la librairie :

«lib-build»: «ng build foniks-lib»
«lib-watch»: «ng build foniks-lib --watch=true»
ce script va builder la librairie et «--watch=true» le met automatiquement à jour dès qu'une modification est apportée à la librairie

«npm-pack»: «cd dist/foniks-lib && npm pack»
ce script crée le package de la librairie

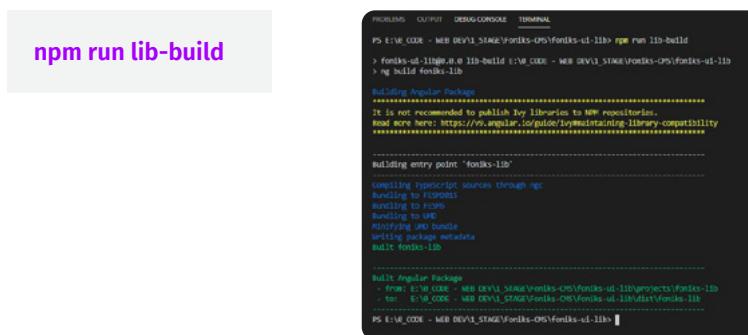
«lib-package»: «npm run lib-build && npm run npm-pack»
ce script lance le build et le package de la librairie

```
package.json x  
foniks-ui-lib > package.json > {} scripts > lib-package  
1 {  
2   "name": "foniks-ui-lib",  
3   "version": "0.0.0",  
4   "scripts": {  
5     "ng": "ng",  
6     "start": "ng serve",  
7     "build": "ng build",  
8     "test": "ng test",  
9     "lint": "ng lint",  
10    "e2e": "ng e2e",  
11    "lib-build": "ng build foniks-lib",  
12    "lib-watch": "ng build foniks-lib --watch",  
13    "npm-pack": "cd dist/foniks-lib && npm pack",  
14    "lib-package": "npm run lib-build && npm run npm-pack"  
15  },  
16}
```



LE PROJET

- 4 Quand la librairie est finalisé avec un ou plusieurs composants, on peut la builder :

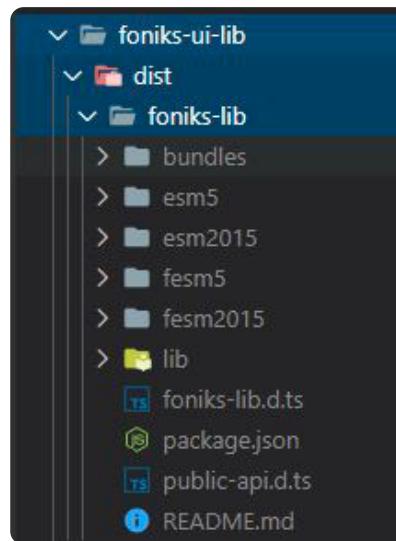


```
npm run lib-build

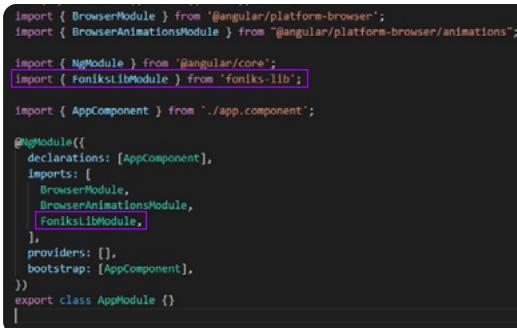
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS E:\V_CODE - WEB DEV\1_STAGE\Foniks-OHS\Foniks-ui-lib> npm run lib-build
> foniks-ui-lib@0.0.0 lib-build & tsc --noEmit --outDir ./dist
> ng build foniks-lib

Building Angular package
It is not recommended to publish Ivy libraries to NPM repositories.
Read more here: https://v2.angular.io/guide/ivy#maintaining-library-compatibility
-----
Building entry point 'foniks-lib'
-----
Compiling typescript sources through tsc
Handling to ESM
Handling to ESM
Building to ESM
Building to ESM
Building esm bundle
Writing package metadata
Built foniks-lib
-----
Built Angular Package
+ From: E:\V_CODE - WEB DEV\1_STAGE\Foniks-OHS\Foniks-ui-lib\projects\Foniks-lib
+ To: E:\V_CODE - WEB DEV\1_STAGE\Foniks-OHS\Foniks-ui-lib\dist\Foniks-lib
PS E:\V_CODE - WEB DEV\1_STAGE\Foniks-OHS\Foniks-ui-lib>
```

La librairie buildée se trouve alors dans le dossier « dist » qui contient le code final à exporter :



- 5 Pour tester la librairie en local dans « foniks-app », il faut l'importer dans « app.module.ts » :



```
import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { NgModule } from '@angular/core';
import { FoniksLibModule } from 'foniks-lib';

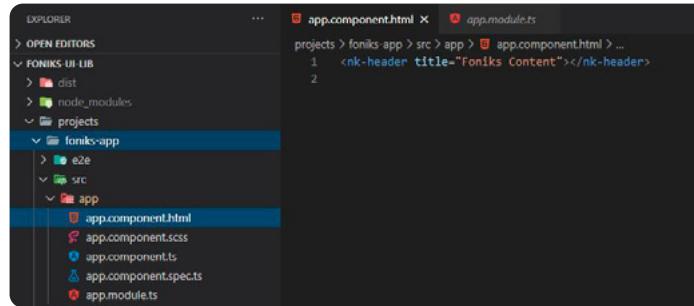
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FoniksLibModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```



LE PROJET

Puis, les composants nécessaires sont ajoutés dans « app.component.html » de « foniks-app ». Le préfixe « nk » devant chaque composant permet de connaître la librairie dont ils proviennent :



- 7 Pour faire le package de la librairie, on va sur le dossier « dist/foniks-lib » `cd dist/foniks-lib` et on lance la commande `npm pack`

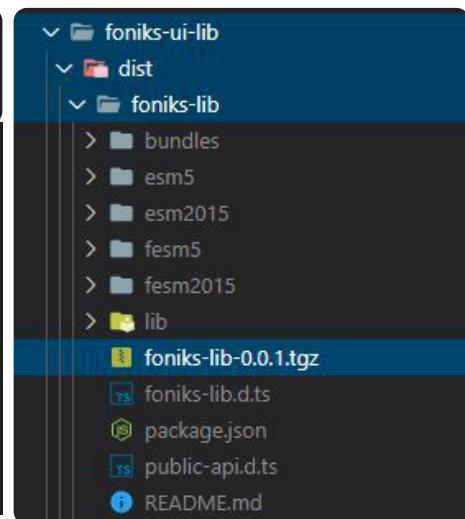
Un package « foniks-lib-0.0.1.tgz » est créé et pourra être exporté dans d'autres applications :

npm run npm-pack

le script «npm-pack» crée le package de la librairie
«npm-pack»: «cd dist/foniks-lib && npm pack»

```
PS E:\0_CODE - WEB\DEV\1_STAGE\Foniks-CHS\foniks-ui-lib> npm run npm-pack
> foniks-ui-lib@0.0.0 npm-pack E:\0_CODE - WEB\DEV\1_STAGE\Foniks-CHS\foniks-ui-lib
> cd dist/foniks-lib && npm pack

npm notice
npm notice package: foniks-lib@0.0.1
npm notice — Tarball: Contents —
npm notice 2.0kB es62015/lib/foniks-lib.component.js
npm notice 2.2kB es65/lib/foniks-lib.component.js
npm notice 476B es62015/foniks-lib.js
npm notice 476B es65/foniks-lib.js
npm notice 22.4kB fesm2015/foniks-lib.js
npm notice 24.3kB fesm5/foniks-lib.js
npm notice 12.6kB es62015/lib/foniks-lib.module.js
npm notice 12.8kB es65/lib/foniks-lib.module.js
npm notice 1.3kB es62015/lib/foniks-lib.service.js
npm notice 1.5kB es65/lib/foniks-lib.service.js
npm notice 27.9kB bundles/foniks-lib.umd.js
npm notice 11.3kB bundles/foniks-lib.umd.min.js
npm notice 3.1kB es62015/lib/header/header.component.js
npm notice 3.4kB es65/lib/header/header.component.js
npm notice 2.8kB es62015/public-api.js
npm notice 753B package.json
npm notice 16.3kB fesm2015/foniks-lib.js.map
npm notice 16.4kB fesm5/foniks-lib.js.map
npm notice 16.0kB bundles/foniks-lib.umd.js.map
npm notice 16.8kB bundles/foniks-lib.umd.min.js.map
npm notice 812B README.md
npm notice 39.4kB foniks-lib-0.0.1.tgz
npm notice 351B lib/foniks-lib.component.d.ts
npm notice 120B foniks-lib.d.ts
npm notice 2.5kB lib/foniks-lib.module.d.ts
npm notice 218B lib/foniks-lib.service.d.ts
npm notice 3770 lib/header/header.component.d.ts
npm notice 774B public-api.d.ts
npm notice === Tarball Details ===
npm notice name: foniks-lib
npm notice version: 0.0.1
npm notice filename: foniks-lib-0.0.1.tgz
npm notice package size: 89.4 kB
npm notice unpacked size: 304.7 kB
npm notice shasum: zbcc4566fb7ab6cc38107af21329725ed476d826
npm notice integrity: sha512-yfgNq11dfJw4[...]EuEVrgcdg+Q==
npm notice total files: 56
npm notice
foniks-lib-0.0.1.tgz
```



- 7 Quand le package de la librairie est fait, on peut l'installer dans l'application « foniks-front » :

```
npm install ..\foniks-ui-lib\dist\foniks-lib\foniks-lib-0.0.1.tgz
```

- 8 Comme on l'a fait en local dans « foniks-app », on répète les mêmes opérations en important d'abord le module dans « app.module.ts » :

```
import { FoniksLibModule } from 'foniks-lib'
```

Puis, on place les composants dont on a besoin dans « app.component.html » ou dans une autre fichier html :

```
<nk-header></nk-header>
```

le préfixe « nk » permet de choisir la librairie dont on a besoin



5. Les vulnérabilités de sécurité

N'importe quelle application peut être la cible d'attaques. Certaines consistent à hameçonner (phishing) un utilisateur ayant les autorisations nécessaires pour accéder à l'application, et à utiliser son authentification pour injecter du code malveillant via un système de requêtes. Cette attaque CSRF (Cross Site Request Forgery) est très dangereuse car elle se fait à l'insu de l'utilisateur. Afin d'anticiper certains problèmes, JHipster permet de mettre en place des services de sécurité afin de relever des failles qui pourraient ne pas être résolues.



Le **phishing** désigne une technique de fraude qui joue sur les failles humaines et non sur des faiblesses technologiques. Les pirates informatiques vont « à la pêche » aux identifiants, mots de passe ou informations bancaires, qui sont ensuite exploitées pour voler données, argent ou identité. Un simple email, d'apparence respectable, peut hameçonner un internaute qui est redirigé vers un site où il doit indiquer des données personnelles.

Le JSON Web Token (JWT)

Cette solution implémentée dans JHipster utilise un système de **token sécurisé** qui se traduit par la vérification de l'intégrité des données (login, mot de passe et autorisations) à l'aide d'une signature numérique ne pouvant être altérée.

Pour cela, JHipster utilise un système de clé qui est configuré dans les fichiers application **yml**. Il faut veiller à les stocker de façon sécurisée lors de la phase de production, la recommandation étant d'utiliser le **JHipster Registry** qui utilise un serveur **Cloud Spring Config**. Il faudra aussi modifier les mots de passe par défaut des comptes **user** et **admin**.



Envoi de code malveillant

Il y a dans l'application un input permettant d'envoyer un fichier JSON. On pourrait craindre le risque d'envoi de code malveillant, mais il y a une protection par le **parse** effectué lors de l'appel de la méthode dans le service *Angular*. Lors de ce procédé, le contenu du fichier JSON est transformé en objets de notre modèle. Si le contenu du schéma ne correspond pas à un schéma viable, l'utilisateur est redirigé vers la page d'accueil, comme prévu dans la méthode de gestion d'erreurs.

Pour les formulaires, une attaque XSS (Cross Site Scripting) est possible. Mais les frameworks modernes, dont *Angular*, offrent une protection puissante. La documentation *Angular* explique que le framework vérifie toutes les valeurs insérées dans le DOM et les nettoie avec « sanitization ». Cela supprime le code indésirable et retourne un résultat sécurisé dans l'application.



Les **XSS** (Cross Site Scripting) sont des injections de code qui vont avoir un impact sur le navigateur de l'utilisateur lors de leur exploitation.
Par exemple, un pirate peut injecter du JavaScript dans une page HTML afin de voler les cookies de session d'une victime pour la rediriger vers un autre site d'hameçonnage ou bien se connecter à l'application web en se faisant passer pour la victime.



CONCLUSION

1. Analyse du stage

A la fin du stage, ce projet a beaucoup avancé depuis sa phase de création. De nombreuses étapes sont à atteindre afin de pouvoir pérenniser l'application.

- Stockage des données en suivant la structure et les contraintes définies par les schémas créés
- Sauvegarde périodique du schéma et des données en prévision d'un besoin de restauration
- Créer l'API permettant la connexion avec l'application *MyNantes*
- Créer le 2e front pour gérer l'édition de contenu
- Vérifier le bon fonctionnement entre *MyNantes* et le CMS Foniks

J'ai eu la chance de travailler sur ce projet, sur lequel la réflexion technique a été constante et m'a appris beaucoup sur la gestion de schémas et de contenu. Le développement du CMS et de la librairie *Angular* a été source de nombreux questionnements, chaque problématique résolue entraînant une nouvelle question.



2. Compétences acquises

Après avoir appris *React* pendant ma formation, ce stage m'a permis de travailler avec *Angular* et son système de composants. J'ai été surpris par l'étendue des dossiers dans l'application, ainsi que par la gestion de modules et de composants génériques qui n'a pas été évident à prendre en main.

Cela m'a permis de travailler sur des aspects du développement que je n'avais pas du tout envisagés, comme la librairie *Angular* qui, j'espère, facilitera le travail des personnes qui reprendront le projet, et donc permettra d'accélérer les futurs développements et évolutions du CMS Foniks. Au delà du développement, j'ai été confronté aux bonnes pratiques à appliquer.

Ces 4 mois de stage m'ont permis de mettre en pratique tous les concepts fondamentaux du développement web appris au cours de ma formation, mais également l'architecture projet, la conception et l'apprentissage de nouvelles technologies.



