GEN-Z WEB

Document d’architecture

SOMMAIRE :

1. Introduction :

Ce document a pour but de tracer l'ensemble des choix effectués pour répondre aux besoins de Knesh.
A travers celui-ci, nous aborderons dans un premier temps l'architecture web, puis les librairies de test et enfin les paradigmes à mettre en place.

2. Choix et justification de l'architecture de l'application

Dans cette partie, nous allons parler du choix effectué concernant l'architecture du projet.

La société Knesh exige dans un premier temps que le site soit disponible en permanence. Cette exigence induit que les technologies utilisées doivent être éprouvées et performantes.
Ensuite, elle mentionne un besoin de rapidité terme qui implique de réduire au maximum le temps de latence ainsi que d'avoir une interface facile d'utilisation.
Une troisième requête est positionnée autour des coûts du projet. Pour que le projet soit le moins cher à développer, il est judicieux d'utiliser au maximum les librairies déjà existantes.
En dernier, le client exprime un désir d'évolutivité, ce qui est sujet à utiliser des technologies connues et documentées pour faciliter le travail des futurs développeurs.

Ces différents besoins peuvent être mis en opposition avec les architectures existantes. Toutefois, il n'est pas intéressant de les lister dans ce document.
Nous vous proposons ce tableau ci-dessous récapitulant les avantages et inconvénients des trois candidates de ce projet :

Architectures	Avantages	inconvénients
Client-serveur	• Adaptabilité • Évolutivité	Possibilité de surcharge utilisateurs
Modulaire	• Évolutivité	Plantages entre modules qui induisent des dysfonctionnements systèmes
Orientée services	• Communications simples • A 100 % sur internet • Norme de sécurité stricte	Possibilité de surcharge du contrôleur de service web induisant des problèmes de performances

Comme le montre le tableau ci-dessus, l'architecture modulaire semble la moins appropriée pour le projet.
Habituellement utilisée pour des systèmes de chiffrement, elle possède aussi une évolutivité intéressante.
En revanche, si l'ajout de nouveaux modules provoquent des erreurs sur les anciens, les évolutions possibles de notre application vont s'en trouver plus complexes. Hors, un besoin du client est une application faible en coûts ce qui implique une facilité de modification et d'appréhension du code. Ces deux éléments suggèrent une rapidité de modification.

L'architecture client-serveur semble elle aussi une bonne solution pour les besoins de ce projet. Du serveur d'impression au serveur de messagerie, elle permet de traiter efficacement les différentes requêtes utilisateur.
Toutefois, dans la mesure où l'on souhaite dimensionner notre application pour un grand nombre d'utilisateurs, son inconvénient qui est la possibilité de surcharge du serveur par un nombre important de client, devient un handicap.
Il n'est pas réaliste de ne pas prendre en considération ce critère du système.

Seule l'architecture orientée service est réellement adaptée aux différents besoins énoncés du site.
Cette dernière pourrait être utilisée à travers un site de suivi de colis ou encore d'informations de vols pour des compagnies aériennes.
Elle supporte un grand nombre d'utilisateurs en simultané ce qui est un avantage de taille face aux deux précédentes architectures.
De plus comme l'indique « kinsta.com » cette dernière permettra de créer une application hautement évolutive et fiable.
Elle répond aux critères de rapidité, de disponibilité, d'évolutivité et de coûts et c'est pour cette raison qu'elle représente le meilleur choix.

Cette dernière comporte toutefois un défaut majeur qui est relatif à la taille des services.
Étant donné que notre application sera simple, cette optique n'est pas à prendre en considération.

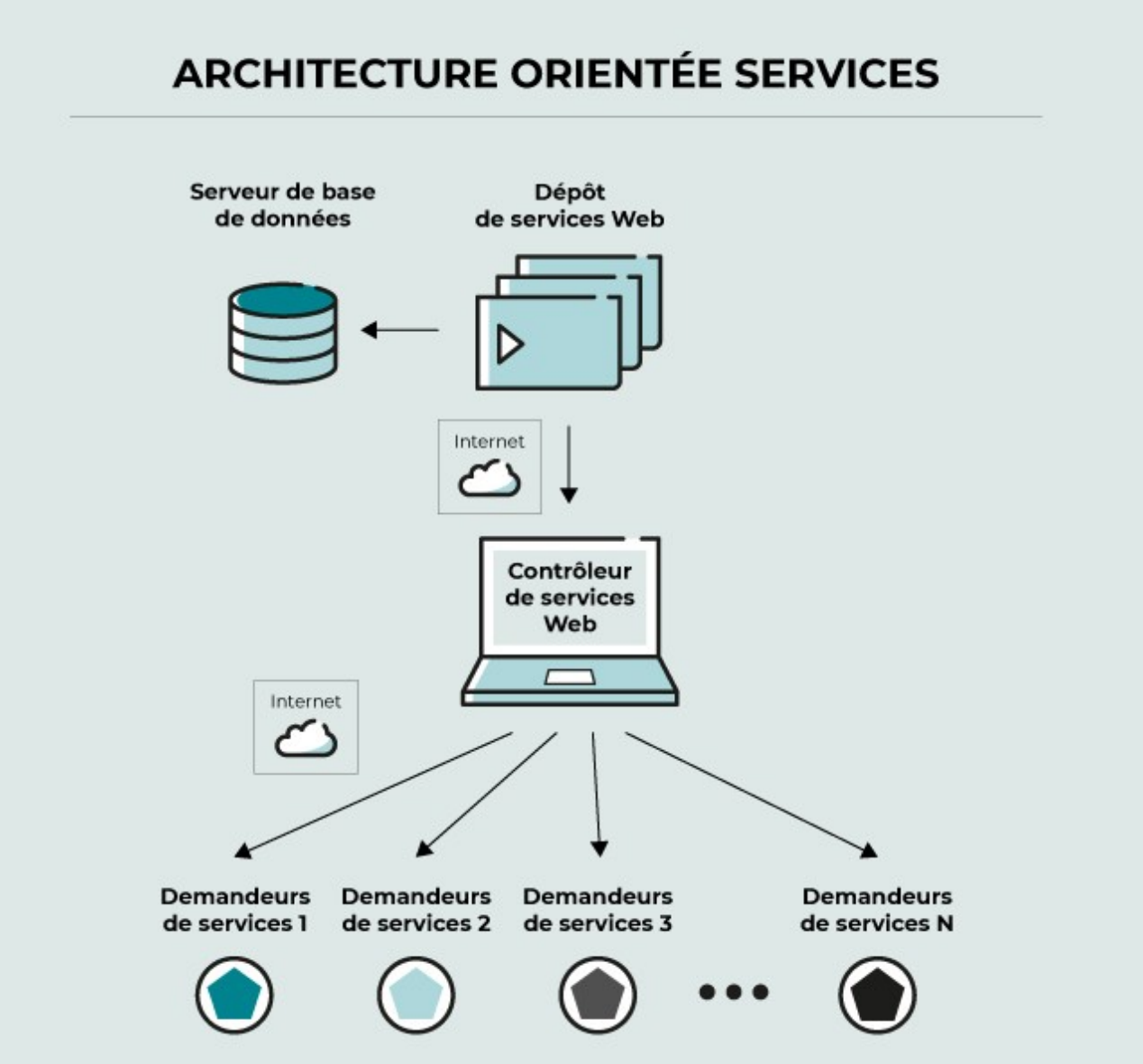

Figure 1: Schéma d'une architecture orientée service

2.1. Mise en œuvre de l'architecture orientée services

La mise en œuvre de cette architecture nécessite la création de services autonomes et réutilisables.
Un service aura une fonction spécifique et sera faiblement couplé avec les autres. Le terme couplage signifie que les services seront indépendants les uns des autres vis-à-vis de leurs fonctionnalités respectives. Ainsi, si l'on modifie un service, ses homologues ne seront pas impactés.

Pour revenir sur sa mise en œuvre, les technologies les plus utilisées sont les **API REST** (Representational State Transfer) et les services webs basés sur **SOAP** (Simple Object Access Protocol).
D'un côté nous avons **REST** qui est léger et adapté aux applications webs modernes et de l'autre **SOAP** qui est souvent utilisé dans les environnements d'entreprises où une garantie de sécurité est primordiale.
Dans le cadre de ce projet il conviendrai d'utiliser une **API REST**.

2.2. Communications entre services

En ce qui concerne la communication entre services, l'**API REST** n'échappe pas au standard HTTP.
Dans ces échanges d'informations, chaque service présente une interface dans laquelle les opérations liées à son utilisation sont décrites à la manière d'une documentation.

2.3. Gestion de la sécurité

La sécurité est très importante pour cette architecture en particulier.
Il est très facile d'imaginer des transferts d'informations sensibles entre deux services.
Admettons que notre service principal envoie à un service de livraison nom, prénom, adresse postale et numéro de téléphone d'un client. La perte de ces données serait critique pour la notoriété de notre site e-commerce, et pour le client.

De ce fait il est important d'axer notre application sur la sécurité. On peut séparer cette notion en trois parties.

La première consiste à considérer l'authentification et l'autorisation.
Chaque service doit être capable de vérifier que l'identité (token) est correcte et de comparer les droits d'accès aux actions souhaitées.
Cette première optique de sécurité n'échappe pas aux standards à savoir **Oauth2** ou **Jwt**. Ces deux mécanismes réalisent filtrages, authentifications ainsi que gestion des tokens d'accès.

Un deuxième aspect serait le chiffrement des communications.
Il est possible de paramétrer un cryptage des données avec un protocole tel que **SSL/TLS**.
Cette nouvelle notion nous permet de nous assurer que lors de l'interception d'un transfert de données par un pirate, il soit incapable d'interpréter ce qu'il a obtenu. En d'autre termes, ce protocole nous permet de rendre les données inutilisables par un tier non autorisé.

Enfin un aspect important est la sécurisation des API contre les attaques informatiques.
On pense notamment aux injections SQL, aux DdoS (attaques par déni de service) ou encore à une faille XSS pour les plus connues. Des actions préventives sont donc à mener pour prévenir ces attaques.
Par exemple, l'utilisation de pare-feux et la limitation de requêtes nous permet de garantir une application robuste face aux menaces.

En conclusion pour cette partie, c'est bien l'architecture orientée services qui est la plus pertinente pour les besoins du client.
En ce qui concerne sa mise en œuvre, il serait judicieux d'opter pour une API REST afin de coïncider au mieux avec l'application légère que l'on souhaite développer.
Les échanges d'informations entre services seront fait via des requêtes HTTP qui seront cryptés par le protocole SSL/TLS afin de sécuriser le transfert de données personnelles.
Enfin, le paramétrage d'un pare feux et la limitation de requête HTTP nous permettra d'avoir une application robuste et résistante face aux menaces informatiques grandissantes.

3. Choix et justification des librairies

Dans cette partie nous allons parler des différents choix que nous avons mené pour répondre au besoin de l'entreprise client.
Dans un premier temps nous évoquerons les librairies de test pour le Front-end puis le Back-end, et par la suite des librairies de composants visuels.

3.1. Librairies de test

3.1.1. Front-end

Dans l'optique de tester cette première composante qu'est le Front-end,il nous a été proposé d'utiliser **Jasmine**, **Jest**, **cypress** ainsi que **Protractor**.

Dans la continuité de la logique de coût suggérée par le client, il convient d'user d'un framework supportant l'ensemble des navigateurs Webs.
De ce fait, **protractor** ainsi que **cypress** ne remplissent pas cette condition. Le premier est fonctionnel uniquement sur Chrome d'après « testim.io » et le second ne l'est pas avec Safari d'après « testsigma.com ».

De ce premier élagage, il est tout aussi important de garder en vue le critère d'évolutivité.
Qu'un framework soit open-source ou soit développé par une grand entreprise comme c'est le cas de **Jest**, **Jasmine** et **Mocha** pourrait donc être intéressant.
Ici, le fait d'être open-source induit un développement via contribution de la communauté et le risque d'être abandonné un jour.
En revanche être développé par une grande entreprise est plus sécurisant sur la durée de vie, sécurité qui ne peut pas être appliquée à la gratuité de l'outil.

Enfin, parmi ces trois candidats il est crucial de piocher celui qui est le plus adapté à notre besoin.
D'après « browserstack.com » **Jest** est idéalement adapté aux applications React de complexité moyenne, **Mocha** est fait pour les applications Node.js complexes.
En revanche, Jasmine est taillé pour fonctionner avec des applications **Angular** simples.

C’est dans ce dernier cas de figure que nous nous trouvons et c’est sur ce choix que va être portée la librairie de test Front-end.

3.1.2. Back-end

Dans l’optique de tester la seconde composante qu’est le Back-end,il nous a été proposé d’utiliser **Selenium**, **cucumber** ainsi que **Junit**.

Cucumber ne semble pas être un choix pertinent.
Comme l’indique « testsigma.com », il n’est pas suffisant à lui même. Cette librairie à besoin d’une autre librairie telle que **Junit** ou **Selenium** pour fonctionner. Nous allons donc l’écarter pour continuer notre réflexion.

Une option confortable concernant le développement serait de pouvoir dérouler l’ensemble des tests en parallèle.
Cette fonctionnalité est embarqué par **Junit** mais ne l’est malheureusement pas pour Selenium d’après « testrigor.com ».

Evolutivité peut aussi allier simplicité.
Junit est conçu pour des projets java et ressemble à du typescript ce qui permettrait au développeur de facilement passer du Front au Back, dans le cadre d’une évolution, plutôt que d’utiliser deux langages fondamentalement différents.
Qui plus est pour coder notre API back-end, un exemple de framework adapté pour du java serait spring-boot qui embarque nativement **Junit 4**. Ce n’est toutefois pas la version la plus récente mais cette solution à le mérite d’être simple. C’est donc sur ce dernier que notre choix va porter.

3.2. composants visuels

Dans la continuité des idées évoqués dans les différents paragraphes ci-dessus, notre librairie de composants visuels se doit d’être documentée et avoir une communauté active.
Les trois candidates proposées sont **Angular Material**, **PrimeNG** et **NG-Zorro.**

NG-Zorro est une librairie chinoise utilisée entres autres par Alibaba dont une partie de la documentation est écrite en chinois. Ce point précis ne répond

pas au critère d’accessibilité que doit comporter la documentation.

Enfin notre choix de librairie va se baser sur une communauté active comme évoqué ci-dessus.
D’après NpmTrend, **Material** est le plus actif des trois librairies. **PrimeNG** reste une bonne alternative mais est largement devancée par **Material** au point de vue de la communauté. C’est donc sur ce dernier que notre choix va se porter.

4. Choix et justification des paradigmes de programmation

Dans cette dernière partie nous allons traiter des choix effectués concernant les différents paradigmes à mettre en place concernant encore une fois le Front-end puis le Back-end.

4.1. Front-end

Le Front comme son rôle l'implique sera de gérer la partie visuelle du site web. Ce dernier aura le rôle de l'Interface Homme Machine (IHM) est devra donc par conséquent réagir à des évènements. Des requêtes seront ainsi envoyées à notre API qui n'est autre que notre Back-end.
Ces deux caractéristiques évoquées ci-dessus correspondent au paradigme de programmation **réactive**.

Des données que nous allons échanger avec l'API, il y aura notamment des données transmises en JSON soit dans notre cas des objets. Nous nous devons de ce fait d'inclure le paradigme de programmation **orientée objet** à notre choix.

Notre application sera aussi structurée en composants qui auront un rôle spécifique. Chacun sera réutilisable et fonctionnera indépendamment des autres.
Cet aspect est celui du paradigme de programmation **déclarative**.

Enfin, compte tenu de la complexité de notre application, au sein de chaque composant nous trouverons une série d'instructions simples qui nous mène à attribuer à notre application le paradigme de programmation **impérative**.

4.2. Back-end

Le Back-end, comme son rôle l'implique sera de gérer tous les échanges d'informations entre la base de données et le Front-end.
A travers ces communications, nous allons principalement échanger des objets.

Il nous est d’ors et déjà possible de positionner un premier choix sur le paradigme de programmation **orientée objet**.

Ensuite, comme la structure d’une API le requiert, notre serveur comportera les différentes routes (requêtes) nécessaires au bon fonctionnement du Front-end.
Ceci implique que ces différents chemins seront distincts et donc d’utiliser le paradigme de programmation **procédurale**.