ECOM - GIZZA

Description du projet

Il s’agit d’une application de commande de pizza en ligne. Les utilisateurs auront la possibilité de constituer leurs menus et de passer des commandes. **L**’idée c’est donc d’avoir un système similaire à ceux que l’on retrouve dans certaines grandes chaînes alimentaires où les utilisateurs peuvent constituer des menus et passer leurs commandes en se servant de bornes dédiées à ces opérations.

Choix de technos pour le projet

### Back-end

JavaEE

Web Services REST

Base de **données** : postgreSql

Wildfly

Maven

### Front-end

Angular 4

HTML5 - ***CSS*** - Bootstrap - Materialize - SASS - UI/UX

Authors

Zodéhougan, Michaël

Hamidi, Hamza

Larreinegabe, Gerardo

Estrangin, Etienne

Boutaleb, Anis Sami

Ratnaparkhi, Gaurang

Les fonctions réalisées, réalisées et testées, et celles restant à réaliser

|  |  |  |
| --- | --- | --- |
| **Fonction réalisées** | **Fonction réalisées et testées** | **Fonction restant à tester et à réaliser** |
| Listing des produits (Pizza/ Boisson/ Dessert) | Testé | Admin/ Employés - gestion des commandes |
| Gestion de Panier (Ajouter/ Supprimer les produits) | Testé | Admin/ Employés - gestion de stock |
| Choisir plusieurs produits | Testé | Réinitialisation de mot de passe |
| Passer une commande | Testé | Modification d’information personnelle |
| Validation d’une commande | Testé | Gestion de l’espace personnelle |
| Création du compte (utilisateur) | Testé | Personnalisation de menu et pizza. |
| Faire un login avec un compte existant | Testé |  |
| Autorisation de admin | Testé |  |
| Passer une commande | Testé |  |
| Admin - gestion des catégories |  |  |
| Admin - gestion des produits |  |  |
| Admin - gestion des employés |  |  |

Les propriétés non fonctionnelles garanties par votre réalisation, et celles restant à considérer pour finaliser votre application

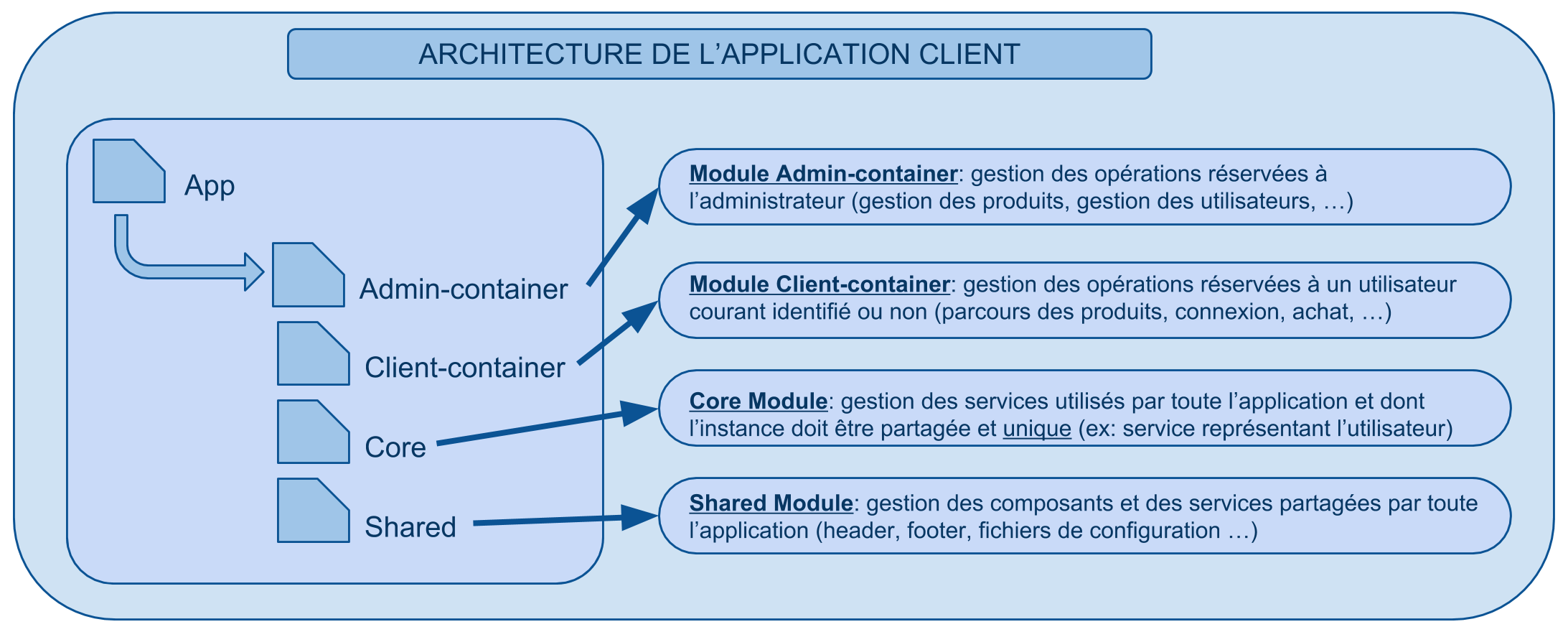
1. Hachage de mot de passe
2. Toutes les transaction sont via https
3. Le base de données est répliqué.
4. Load balancer de la charge. On assure l’haut disponibilité du site.
5. Authentication/ Authorization: Un utilisateur peut pas accéder la section.
6. Site compatible avec les navigateurs différents (IE 9+, Mozilla Firefox, Chrome, Safari 7, Android 4.4)

### Celles restant à considérer

1. Cache de requête avec REDIS ou EHCACHE
2. Integration continue
3. Progressive web application avec un Certificat officiel signé

L’application Client

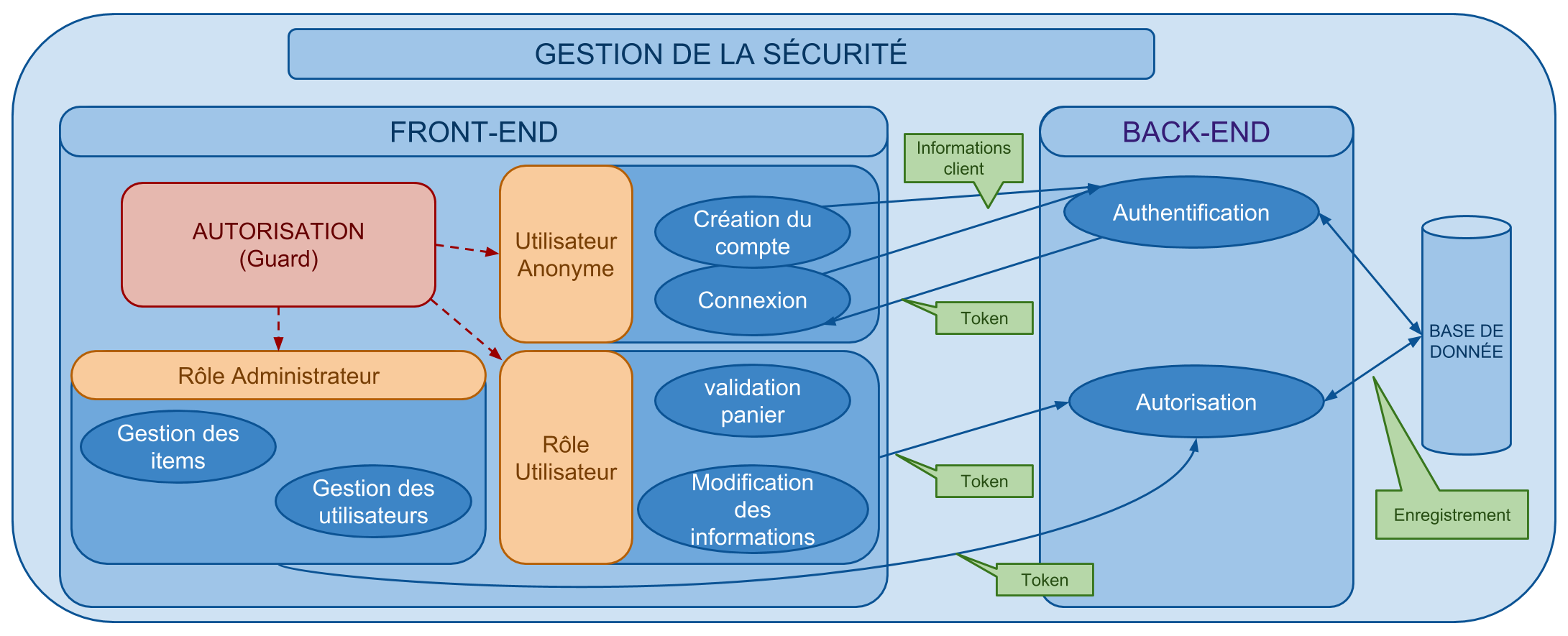
Architecture



Le schéma en haut représente l’architecture de l’application. Il contient 4 modules:

Le module admin ou client sera chargé selon le cas d'authentification tandis que le module Core et Shared sont partagés dans toute l’application.

Communication BackEnd FrontEnd



Le client envoie des requêtes pour avoir des ressources comme la liste des pizzas, boisson, etc. C’est des requêtes GET.

Pour faire une commande l’utilisateur doit avoir un compte.

**Cas1:**

Il crée un compte et donc remplit un formulaire et l'envoie au serveur il recevra alors une réponse avec un token. Au frontEnd nous faison un test pour voir si la réponse contient un token. Si c’est le cas, le token est donc enregistré dans le local storage. Ce dernier ne permet non seulement de persister la connexion de l’utilisateur, mais aussi est obligatoire pour toutes les requêtes utilisateur qui viennent après ( validation commande , modification informations utilisateur…).

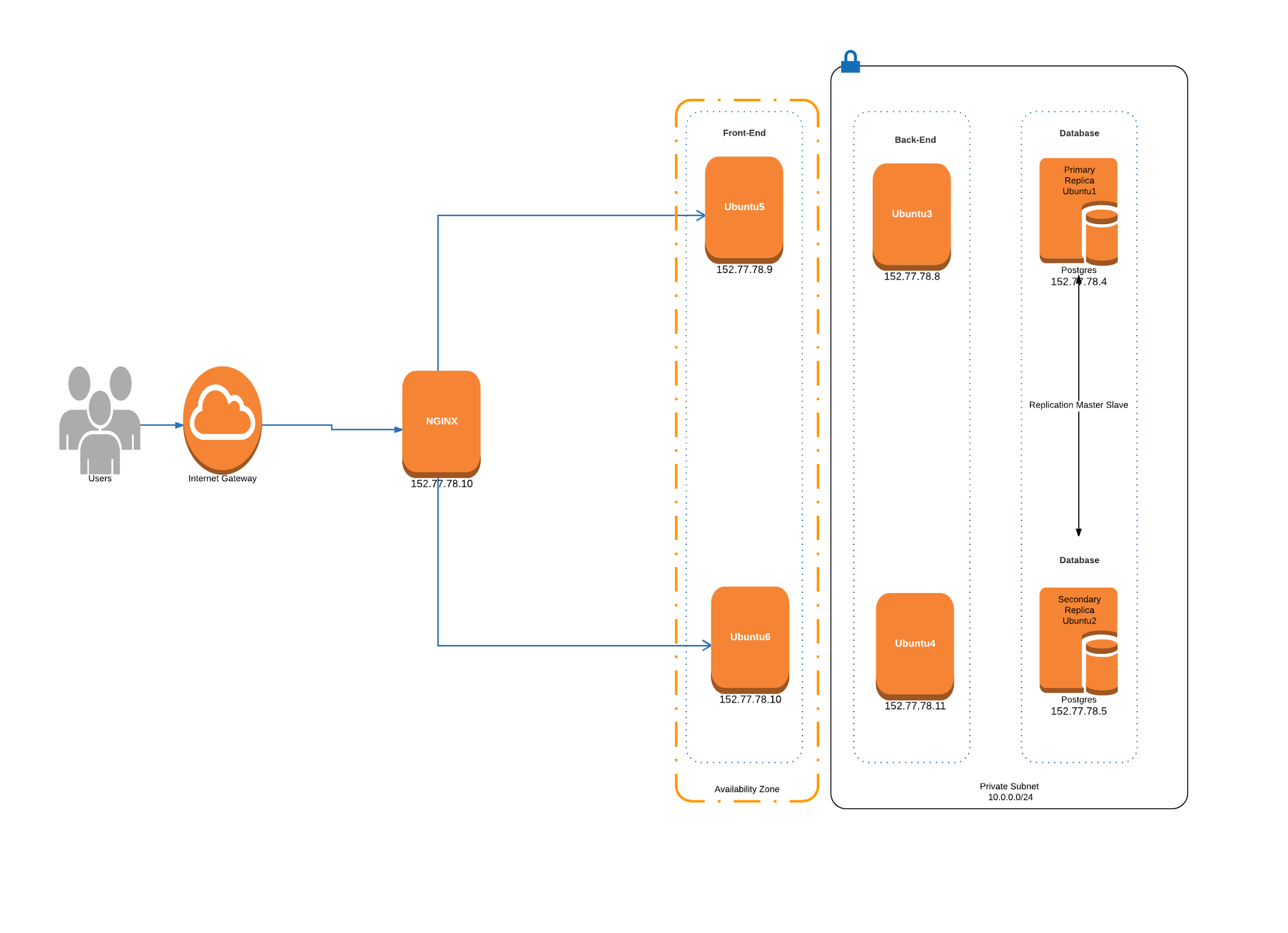
**Cas2:**

Il s’authentifie à ce compte et donc envoie un requête qui contient ses identifiants. Si les identifiants sont correctes, il recevra alors une réponse qui contient le token qui est stocké dans le localstorage. Pour les communications à venir on envoie le token dans header pour les requêtes utilisateur qui viennent après ( validation commande , modification informations utilisateur…).

Le token est encodé en base 64. On utilise la bibliothèque JWT en FrontEnd pour le décoder et après avoir le champs username et rôle (soit user soit admin).

Système

Architecture



L’architecture du système est composée pour deux partis principal:

1. La DMZ, avec les deux serveur FrontEnd
2. La Zone Sécurisée, avec deux serveurs Backends et la Base de Donnée Répliquée.
   1. Il y a un serveur qui fonctionne comme balanceur de charges pour les serveur FrontEnd et BackEnd.

Obs.: La Zone DMZ c’est juste une idée d'implémentation, elle est simulée.

Serveur d’Application BackEnd

Le serveur d’application utilisé est Wildfly. On l’a choisi parce qu’il respect tous les spécification de Java EE.

Serveur d’Application FrontEnd

On utilise Nginx comme:

* Serveur d’application
* Balanceur de charge
* Redirection pour le SPA au *root* et pour le *Path Location Strategy.*
* Balanceur de charge pour le Backend

Sécuritée

Le pare-feu de chaque serveur est activé et des règles sur les ports sont mis-en-place avec UFW.

Tous les ports sont fermés sauf:

* Le port 22 est ouverte dans tous les serveur.
* Le port dédié à chaque application est ouvert.
* BackEnd: Le port Http(80) et le port Https(443).
* FrontEnd: Le port Http(80) et le port Https(443).
* Base de Données: Le port postgresql(5432).

Le site fonction sur HTTPS, pour cela on a fait:

* Certificat auto signé.
* Redirection de requête http -> https.

On utilise des Token JWT pour la authentication et authorization avec des rôles.

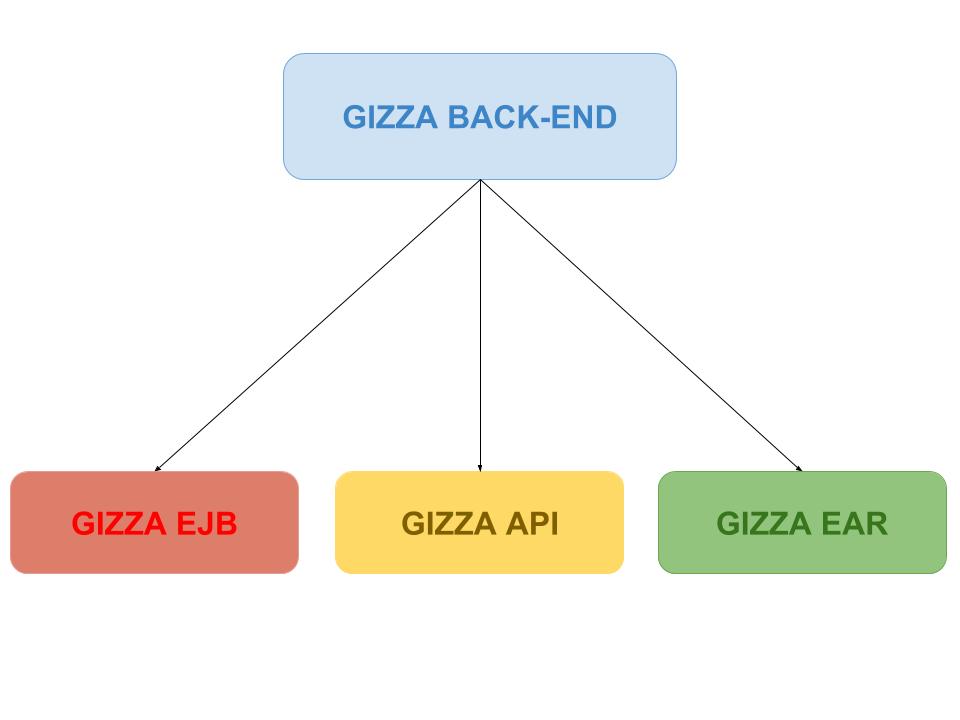
Base de Données

Le serveur choisi est Postgres. On a fait une Réplication Master - Slave Streaming

Présentation de l’application serveur

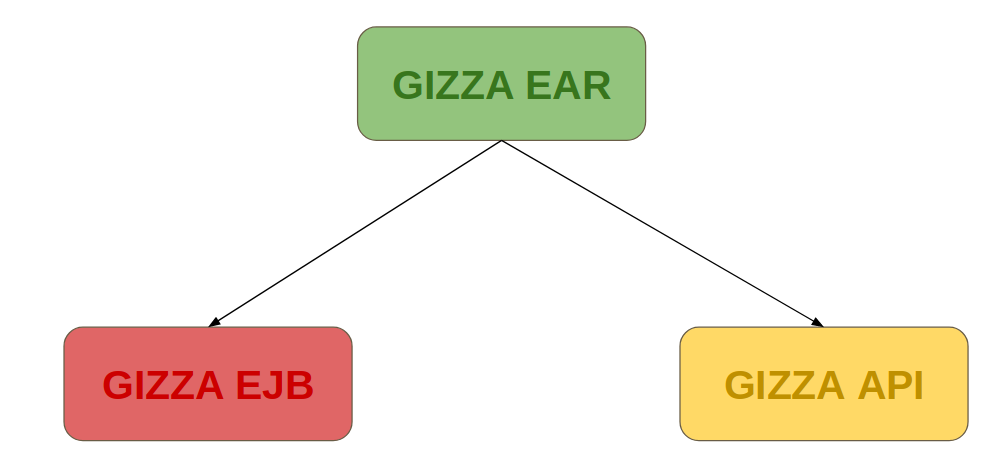
Le projet serveur que nous appelons GIZZA BACK-END, est un projet Maven composé de trois modules:

* Le module GIZZA-EAR
* Le module GIZZA-EJB
* Le module GIZZA-API



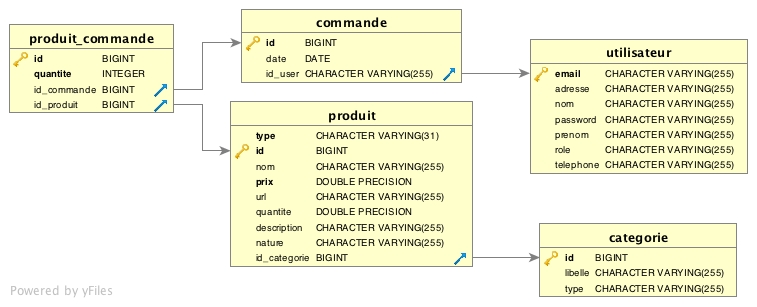
Module GIZZA-EAR

Le module GIZZA-EAR est celui que nous déployons sur le serveur. Il est composé des fichiers **EJB** et **WAR** obtenus à partir des builds respectifs des projets GIZZA-EJB et GIZZA-API. Le module GIZZA-EAR se présente comme suit:

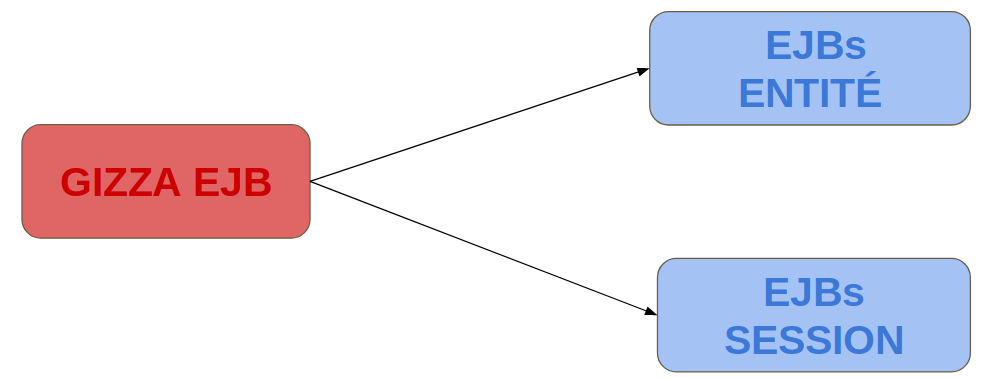


Module GIZZA-EJB

Le module GIZZA-EJB représente la couche d'accès aux données de l’application. C’est dans ce module que nous implémentons nos EJBs entités et nos EJBs sessions stateless. Les EJBs entités ont été réalisés conformément à notre modèle de données qui se présente comme suit:

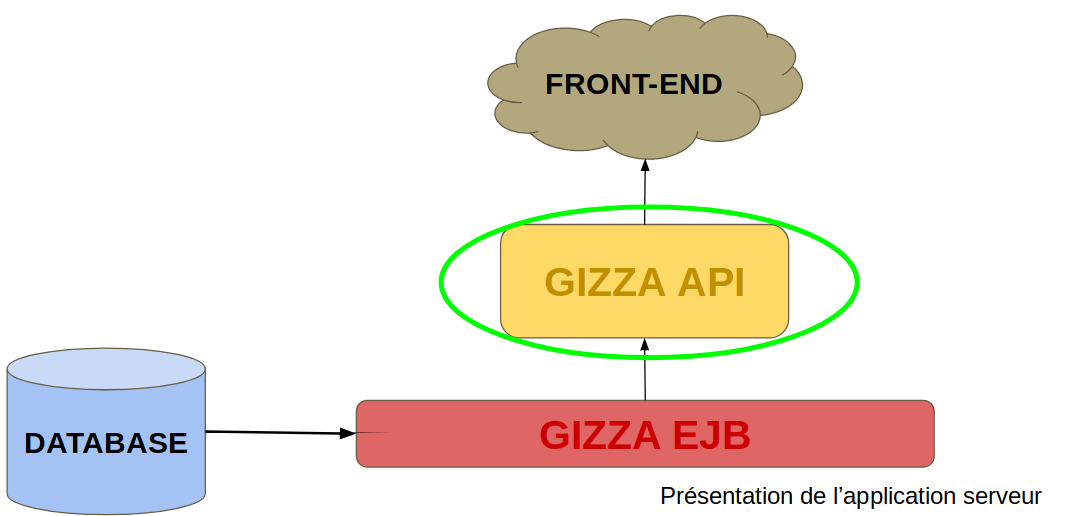


A chacune des tables figurant sur ce modèle correspond un Entity Bean pour le mapping objet relationnel. A chaque Entity Bean, nous associons un EJB stateless qui permet de réaliser toutes les opérations de CRUD (Create-Read-Update-Delete) sur cet Entity Bean grâce à un composant fourni par le framework JavaEE 7 appelé: **EntityManager**. Ce composant a la particularité de gérer tous les aspects liés à la transaction (atomicité, unicité cohérence, etc...) en plus de ceux liés à la persistance des entités. Ci-dessous un aperçu du projet EJB:



Module GIZZA-API

Le module GIZZA-API expose toutes les fonctionnalités développées en back-end sous la forme de services RESTFull. Ce module s’appuie sur les fonctionnalités d’accès aux données fournies par le module GIZZA-EJB pour mettre à disposition du Front-End les services proposées en Back-End. Il se présente comme suit:

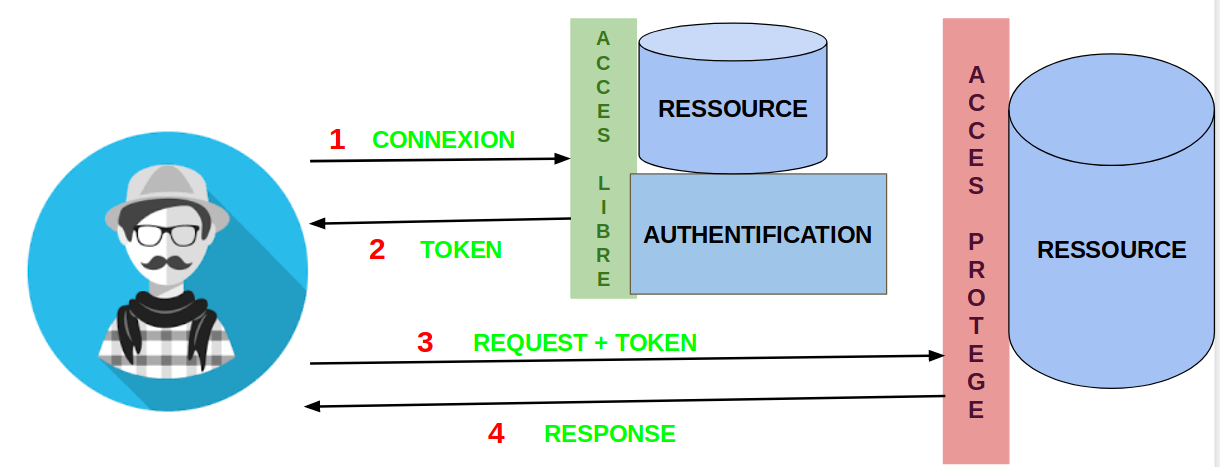


Sécurité au niveau du Backend

La gestion de la sécurité au niveau de l’application serveur se fait à deux: l’Authentification et l’Autorisation.

Un utilisateur peut consulter le site en mode anonyme (sans login ni mot de passe). Dans ce mode, il a accès aux ressources non protégées de l’application.

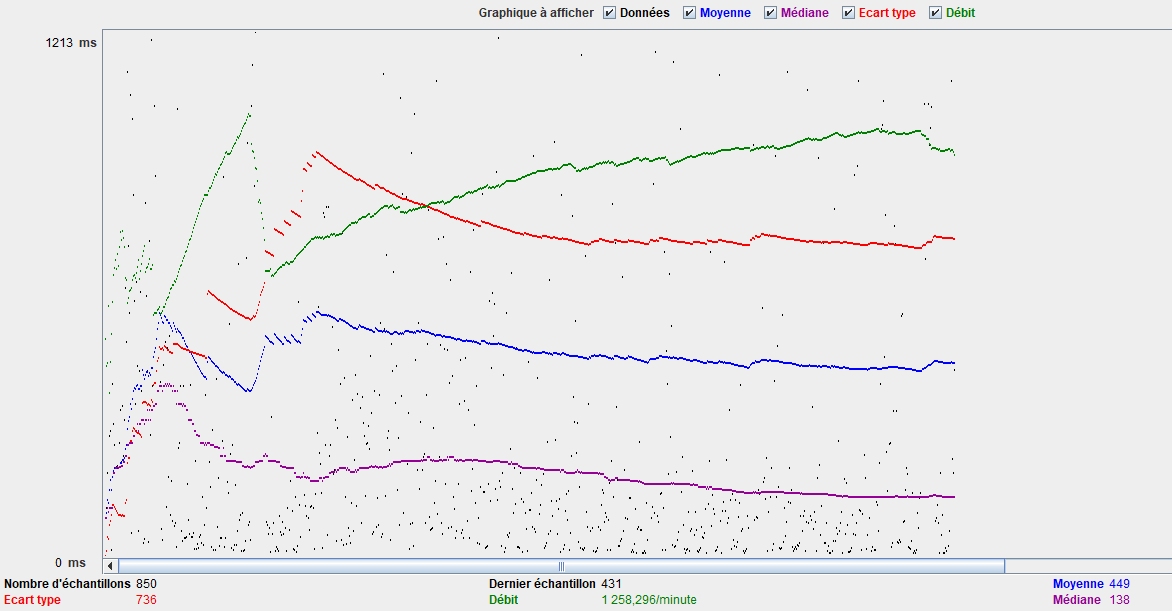
Lorsqu’un utilisateur se connecte, il reçoit un token jwt (Json Web Token) dont il se sert lors toutes ses requêtes sur une ressource protégée. Ce token est passé dans toutes les entêtes des requêtes du client vers le serveur. Le token est composé du login et du rôle de l’utilisateur. L’architecture mis en place au niveau de la sécurité se présente comme suit:



Tests de charge

Afin de tester le site Gizza, on a utilisé l’outil d’injection de charge Apache Jmeter.

On a commencé par établir un scénario fonctionnel qui se déroule comme tel : l’utilisateur commence par charger la page d'accueil, s’authentifie, sélectionne une pizza et enfin valide sa commande. Ensuite, on a établi 2 scénarios de tir de charge : le premier avec 10 utilisateur qui exécutent 5 itérations avec une monté en charge de 1s, le second avec 100 utilisateurs qui exécutent 50 itérations avec une monté en charge de 30s.

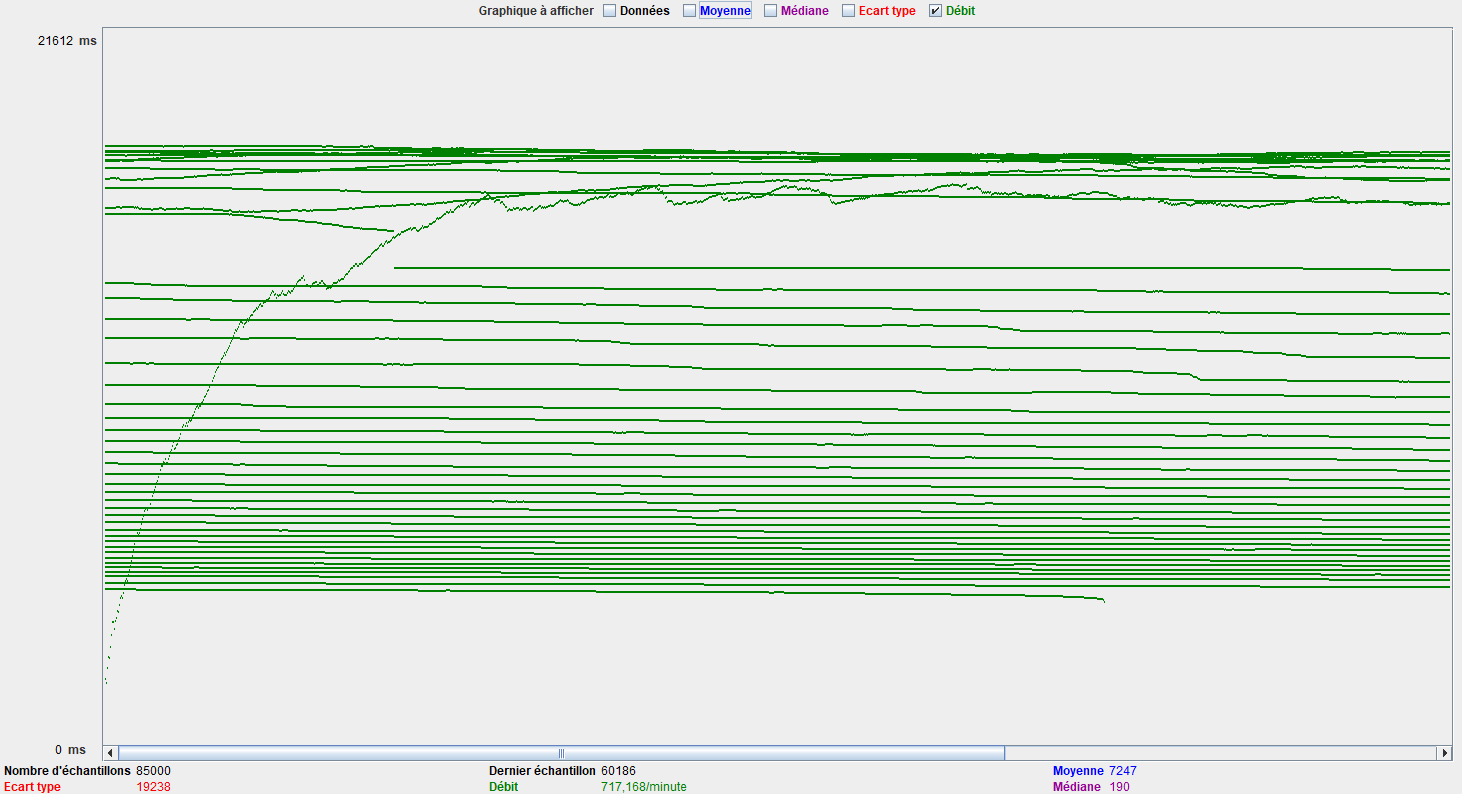
Le résultat du premier tir de charge est représenté dans le graphe suivant :

on constate que le débit augmente progressivement avec l’augmentation du nombre d’utilisateurs et qui se stabilise sans baisse brutal, même chose pour la moyenne du temps de réponse (qui est en Milliseconde) qui augmente et ensuite se stabilise. Le test a duré 3 minutes.

Les résultats du deuxième test sont présentés dans le tableau et les graphe suivant:

le tableau représente les résultats consolidé, on peut distinguer le nombre total de requêtes qui a été réalisé qui s'élève à 85000 avec un taux d'erreur de 5.15% sur la totalité mais un taux de presque 40%.

les graphe suivants montrent le débit ainsi que le temps de réponse moyen. Les deux courbe évoluent de manière proportionnelle. Le débit augmente avec l'augmentation du nombre d'utilisateurs, au bout de quelques itérations la courbe fait une cassure qui correspond au début de l’apparition des erreur. on peut le constater même sur le graphe du temps moyen de réponse.

La cause des erreurs a étés identifié. il s’agit d’une relation ManyToMany qui charge a chaque fois les commandes reliés au produit et donc la taille de l’objet augmente avec l’augmentation du nombre de commande. Cette erreur a été corrigé par la suite. 

Graphe du débit



Graphe de la moyenne du temps de réponse

