

# Fork Eat by Foodéfix

Maud Gellée

Marion Armangaud  
Thomas Lacaze

Carine Talandier  
Arsène Lapostolet

Jean Michel Remeur

## Contents

<b>Organisation dans l'équipe</b>	<b>2</b>
<b>Rôle fonctionnel de l'application</b>	<b>2</b>
<b>Analyse du domaine</b>	<b>2</b>
<b>Outils et langages</b>	<b>4</b>
Application Mobile . . . . .	4
Serveur d'application . . . . .	4
Prototype UI . . . . .	4
Serveur de base de donnée . . . . .	4
Infrastructure . . . . .	4
Conteneurisation d'application . . . . .	4
Traefik . . . . .	5
Drone . . . . .	5
Terraform . . . . .	6
Ansible . . . . .	6
Embarqué . . . . .	6
Golang . . . . .	6
Machine learning . . . . .	7
<b>Architecture technique et méthodologies de développement</b>	<b>7</b>
Back-end . . . . .	7
Architecture . . . . .	7
Validation et tests : Test Driven Development . . . . .	8
Sécurité . . . . .	9
Mobile . . . . .	9
Embarqué . . . . .	9
Infrastructure . . . . .	11
Continous Integration . . . . .	11
Continous Delivery . . . . .	13
Machine learning . . . . .	14
Recherches . . . . .	14
Implémentation . . . . .	17
<b>Problèmes rencontrés</b>	<b>17</b>
Front-end . . . . .	17
Back-end . . . . .	17
Mobile . . . . .	18
Embarqué . . . . .	18
Infra . . . . .	18
Machine learning . . . . .	19
<b>Amélioration possible</b>	<b>19</b>

Code source disponible ici (sous licence MIT) : ForkEat

## Organisation dans l'équipe

Nous avons organisé notre équipe en quatre pôles, en fonction des compétences et préférences de chacun :

- Pôle application mobile : Jean-Michel et Carine
- Pôle back-end : Marion et Arsène
- Pôle infrastructure et DevOps : Thomas
- Pôle Machine Learning : Maud

Le suivi et la gestion du projet a été organisé par Maud à l'aide de l'outil Monday.com.

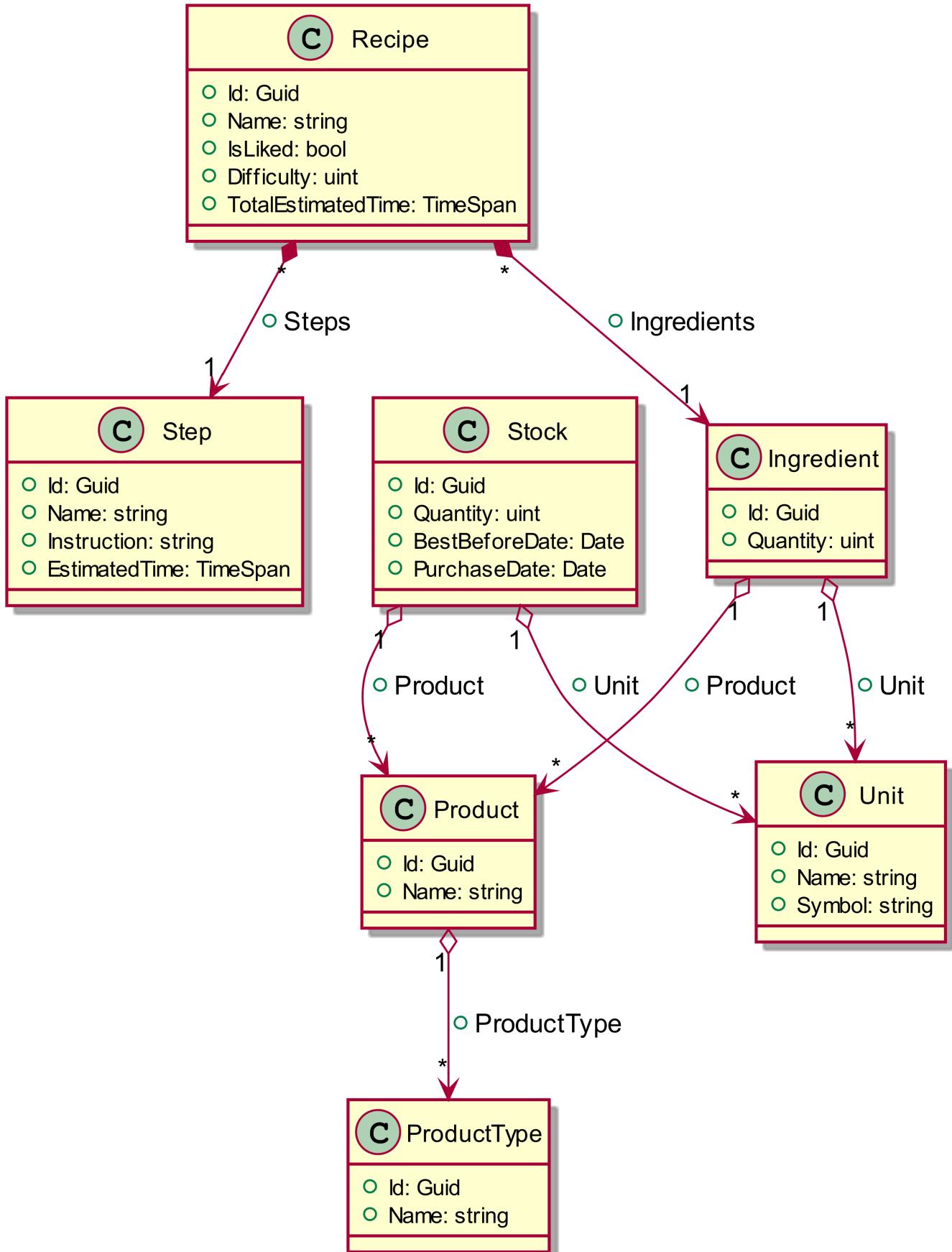
Pour la communication et chat et vocal nous avons utilisé un serveur Discord.

## Rôle fonctionnel de l'application

Nous avons créé une application open-source et autohébergeable de gestion de stock de nourritures, afin de limiter le gaspillage et de faciliter la planification des repas. Dans l'état actuel, notre application permet d'entrer les stocks de produits en base manuellement, de créer des recettes et de les chercher par noms ou ingrédients. De plus, nous avons commencé le développement d'un système de recommandation de recette, mais qui n'a pas pu être pas encore intégré à l'application.

## Analyse du domaine

Au fur et à mesure de notre avancée sur le projet, nous avons précisée nos connaissances du domaine pour arriver au modèle de donnée suivant :



# Outils et langages

## Application Mobile

Pour la partie mobile nous avons décidé d'utiliser Flutter comme outil de développement, car il apporte beaucoup d'avantages. Dans un premier temps, grâce au langage Dart, il est possible de rafraîchir rapidement l'affichage de l'écran en prenant en compte les modifications avec le "hot reload", ce qui évite d'avoir à fermer l'application et de la relancer. De plus, il possède toutes les fonctionnalités des plates-formes natives comme le défilement, la navigation, la police d'icônes, le texte, la vue Liste, etc. Mais également des widgets et designs qui lui sont propres ce qui le rend indépendant de la plateforme et lui permet de couvrir à la fois iOS et Android. Ainsi une application peut être conçue pour fonctionner sur les deux plateformes. L'écosystème est open source et tous les packages sont sous licence BSD ou MIT ce qui les rend totalement libre d'utilisation.

Dart est un langage de programmation optimisé pour plusieurs plateformes. C'est un langage orienté objet et asynchrone qui supporte le fonctionnement de code en parallèle et permet de fournir des réponses en temps réel.

## Serveur d'application

Le serveur d'application est écrit avec le langage C# sur le runtime cross-plateforme et open-source .NET 6 (nous avons commencé sur la version 5 puis continué sur la version 6 sortie en novembre). Nous avons choisi C# et .NET, car ce sont des outils modernes qui permettent de développer du code fiable, et ce avec un bon confort de développement.

Pour le protocole de communication avec le client, nous avons sélectionné REST, le standard en la matière de service Web. Notre application backend est donc une API Web REST développée le framework web fourni avec le runtime .NET, ASP .NET Core.

La sécurité de l'API est gérée par une authentification utilisant JSON Web Tokens. Cette option est la plus compatible avec ReST, respectant son caractère sans état.

Pour la persistance des données dans la base de donnée relationnelle, nous avons utilisé le standard de facto en la matière de mapping relationnel-objet pour .NET, Entity Framework Core.

Concernant les tests unitaires, nous avons sélectionné xUnit, l'un des trois frameworks de tests pour C# les plus utilisés, car nous avions déjà de l'expérience dessus.

Enfin, pour la documentation de l'application, nous avons utilisé l'intégration de Swagger pour ASP .NET qui génère à partir du code C# le JSON OpenAPI consommé par Swagger.

## Prototype UI

Figma

## Serveur de base de donnée

Pour cette application, nous avions besoin d'une base de donnée relationnelle. Ainsi, nous avons sélectionné PostgreSQL comme serveur de base de donnée. PostgreSQL un serveur de base de donnée relationnel moderne et open-source très performant et qui présente des fonctionnalités spécifiques intéressantes, comme un support pour la full-text search.

## Infrastructure

### Conteneurisation d'application

Nous avons utilisé Docker pour faciliter les déploiements, c'est un outil Open Source qui permet de créer différents conteneurs (plus souvent appelés containers). Dans chacun d'eux se trouve une ou plusieurs applications. Cet élément permet de déplacer facilement les containers et de les déployer sur différents serveurs.

Docker repose sur la décomposition des couches applicatives, cela permet à chaque application d'être maintenue de façon autonome mais aussi de disposer de différentes versions entre les instances.

Un container est une image d'une application à l'instant T. Les images sont disponibles soit sur un registre privé<sup>1</sup> ou public (Docker hub), soit il est possible de créer ses propres images à l'aide d'un fichier Dockerfile. Tout Dockerfile se base déjà sur une image créée au préalable.

---

<sup>1</sup>Ensemble d'images communautaires regroupées sur un site. (Équivalent à Github mais pour les images Docker).

Ainsi, le mécanisme d'image permet l'interopérabilité. Grâce au Docker Engine les images créées sont disponibles sur n'importe quel système d'exploitation : Windows, Linux, Mac OS. Cela permet de réaliser la migration d'un serveur à un autre très facilement et de faciliter la maintenance des couches applicatives.

Cet outil dispose d'un avantage conséquent face aux machines virtuelles (VM) : pour faire tourner un container il faut moins de ressources. En effet, sur une machine virtuelle, il est obligatoire de virtualiser le système d'exploitation (OS) comme Debian, Ubuntu, Windows alors que le container se base sur la technologie LXC (Linux Container) qui est une méthode de cloisonnement de l'OS. Ainsi chaque container est isolé les uns des autres mais ils partagent tous le même noyau. Un container n'inclut donc pas d'OS. C'est pour cela que Docker demande moins de ressources et permet de mettre plus de containers par serveur.

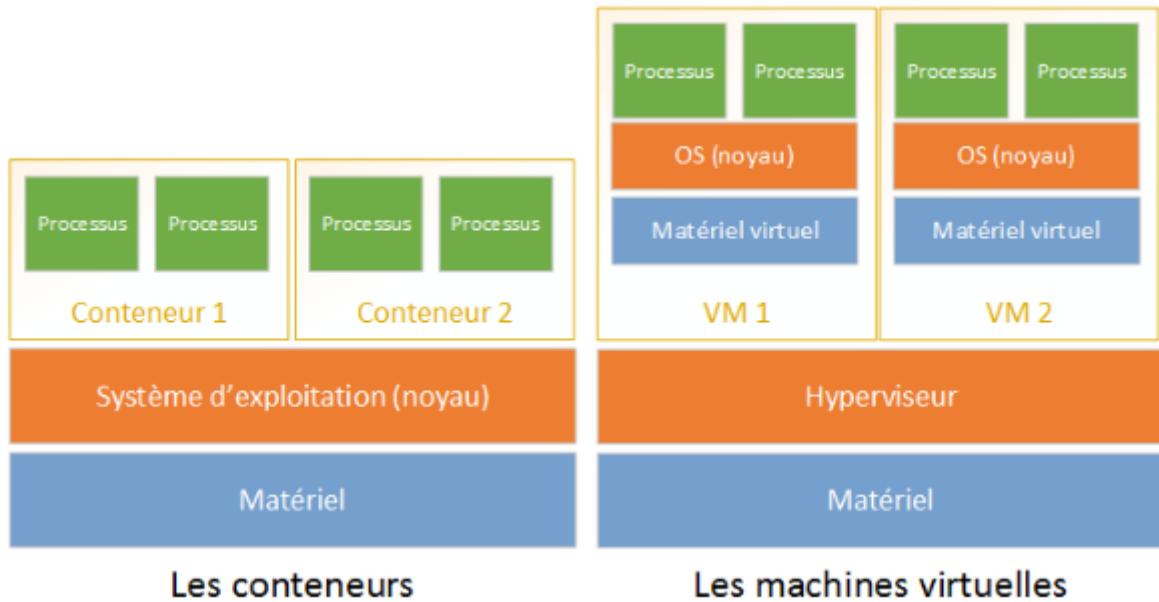


Figure 1: Différences entre Container et VM

## Traefik

L'un des problèmes avec les conteneurs est leur cycle de vie court. Un conteneur peut être détruit et recréé beaucoup de fois en un espace de temps réduit. Ainsi avec un reverse proxy traditionnel comme Nginx, il n'est pas possible d'associer un conteneur à un trafic défini, il faut forcément passer par un port. Traefik est un reverse proxy plus récent permettant de gérer dynamiquement divers fournisseurs (Docker, Kubernetes, AWS, ...). Il n'est donc pas nécessaire de passer par l'ouverture d'un port comme le fait Nginx.

Traefik est un edge router créé par Containous une société française fondée par Emile Vauge. Celle-ci propose Traefik en open-source sur Github, mais aussi Traefik EE pour Entreprise Edition permettant le load balancing et de la haute disponibilité.

Ainsi, prenons un exemple simple. Si nous n'utilisons pas un reverse proxy tel que Traefik, lors d'un déploiement d'un conteneur Docker, nous devons exposer le port du service en dehors du conteneur afin de pouvoir y accéder de l'extérieur. Si en plus nous voulons lui associer un sous domaine nous devons alors utiliser Nginx, ou un autre système afin de rediriger le trafic du sous domaine vers le port d'écoute du conteneur.

Dans cette opération rien n'est automatique. Cela devient fastidieux lors qu'on déploie des dizaines de conteneurs par jour. C'est dans cette problématique que Traefik a été conçu. Lancé dans un conteneur, Traefik est capable de rediriger le trafic en fonction des labels qui lui sont définis dans les conteneurs nécessitant un routing particulier. Le reverse proxy sait en "temps réel" où se trouvent les conteneurs, et il peut donc gérer le routing sans devoir redémarrer de conteneur, ni même ouvrir de port.

## Drone

Drone est un outil d'intégration continue permettant d'automatiser le déploiement d'applications. Drone est un outil comparable à Jenkins seulement, il a pour avantage d'être nativement compatible avec Docker : c'est la raison pour laquelle nous l'avons choisi pour déployer le front de notre application : avec Drone, il était bien plus simple de gérer les différentes images Docker auxquelles nous avons pu avoir recours.

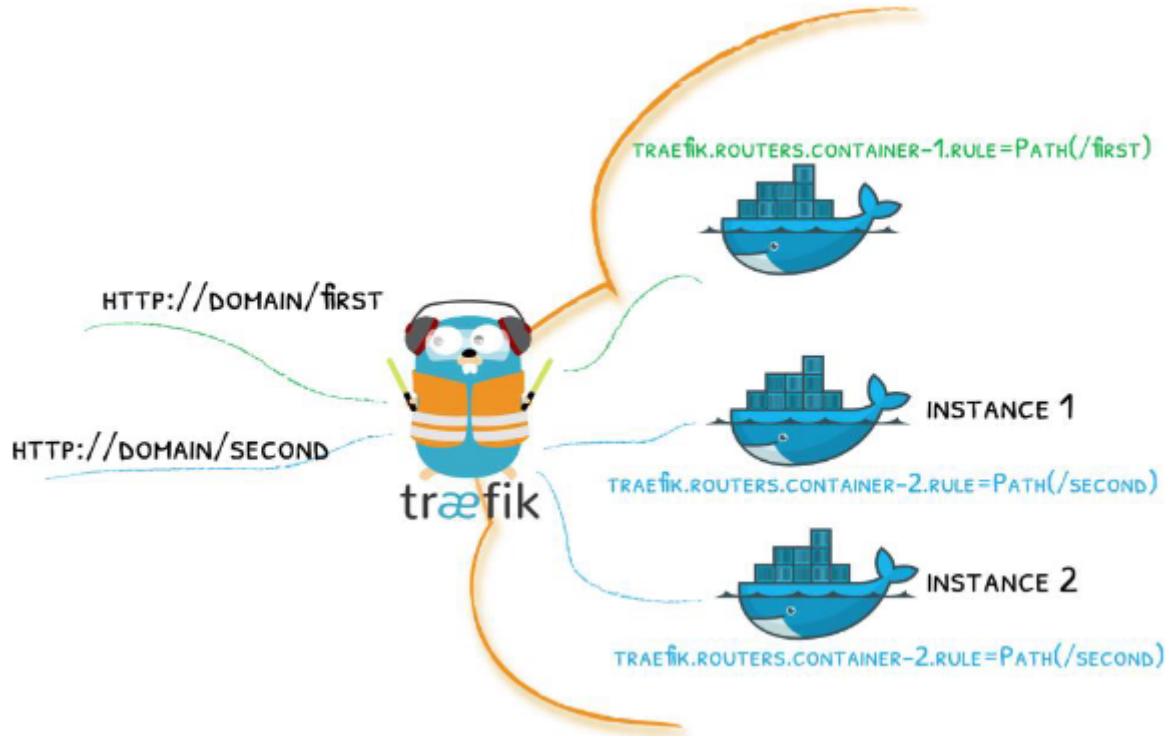


Figure 2: Schéma du fonctionnement de Traefik

## Terraform

Afin de déployer notre infrastructure sur un cloud, nous avons utilisé Terraform qui est un outil open source d'infrastructure as code développé par HashiCorp. Il permet de déployer et d'administrer une infrastructure avec du code déclaratif facile à apprendre et à comprendre. Terraform a un cycle de vie propre :

- **init** : permet de créer les dossiers ainsi que tous les fichiers de configurations
- **plan** : crée un plan d'exécution afin d'atteindre l'infrastructure désirée
- **apply** : phase où les modifications sont faites selon le plan
- **destroy** : supprime toutes les vieilles ressources de l'infrastructure

L'architecture de Terraform a deux composants principaux, le Terraform Core et les providers. Le Core utilise deux sources d'entrée pour faire son travail : la configuration Terraform que nous définissons (on définit ce qui doit être créé ou provisionné) et l'état (l'endroit où Terraform stocke l'état courant de l'infrastructure). Le Core permet de planifier le travail à faire et de comparer l'état actuel à l'état que l'utilisateur souhaite obtenir.

Les providers pour des technologies spécifiques (providers cloud comme AWS) ou pour des composants plus hauts niveaux (comme Kubernetes). Ainsi on a la possibilité de créer des infrastructures à des niveaux différents (créer une infrastructure AWS et déployer Kubernetes au-dessus puis créer des services à l'intérieur de Kubernetes).

## Ansible

En complément de Terraform nous avons besoin d'Ansible. C'est un outil open-source, permettant d'automatiser le provisionnement et la configuration de nos instances.

## Embarqué

### Golang

L'une des principales compétences que j'ai pu acquérir grâce à Eiffage est la programmation en Golang. Ce nouveau langage de programmation développé par Google depuis 2007 est là pour s'adapter à une évolution des autres langages et vient répondre à de nouveaux besoins. En effet, les logiciels sont de plus en plus complexes, et la force d'un langage réside dans le choix des fonctions et concepts qu'il intègre ainsi que la manière dont ils sont intégrés. Or le Go prend exemple sur des langages préexistants pour y prendre le meilleur, et se concentre également sur des fonctionnalités modernes.

Pour mettre en évidence la jeunesse de ce langage, la première version du Go date du 10 Novembre 2009 et sa première version stable de Mars 2012. Le langage C est apparu en 1972, il y a maintenant 47 ans.

Des éléments comme une syntaxe proche du C, une compilation rapide, ou encore un ciblage du monde actuel avec la programmation multi cœur, many-cœur complétés avec une gestion automatique de la mémoire apportent de nombreux bénéfices.

Les fabricants de matériel informatique ajoutent de plus en plus de coeurs aux processeurs pour en augmenter les performances. Tous les centres de données fonctionnent sur ces processeurs et nous devrions nous attendre à une augmentation du nombre de coeurs dans les années à venir. De plus, les applications d'aujourd'hui utilisent de multiples micro-services pour maintenir les connexions aux bases de données et gérer les files d'attente de messages et les caches. Ainsi, les logiciels développés et les langages de programmation utilisés, devraient pouvoir supporter facilement la concurrence et être évolutifs avec un nombre accru de coeurs.

Cependant, la plupart des langages de programmation modernes (comme Java, Python, etc.) sont issus de l'environnement single-threaded des années 90. La plupart de ces langages de programmation supportent certes le multi-threading, mais le vrai problème vient de l'exécution simultanée, du verrouillage des threads, des situations de compétition et des interblocages. Ces éléments rendent difficile la création d'une application multi threadée sur ces langages.

D'autre part, Go a été lancé en 2009 alors que les processeurs multi-coeurs étaient déjà disponibles. C'est pourquoi Go a été conçu en tenant compte de la concurrence. Go a des goroutines au lieu de threads. Ils consomment environ 2KB du tas. Ainsi, il est possible de faire tourner des millions de goroutines à tout moment. L'un des avantages les plus importants des langages C ou C++ par rapport à d'autres langages modernes de plus haut niveau comme Java ou Python est leur performance. En effet, les langages C et C++ sont compilés et non interprétés.

Le Go supporte des mécanismes souvent retrouvés dans des langages plus haut niveau comme le ramasse-miettes tout en présentant une syntaxe et une performance proche des langages de niveau inférieur comme le C ou le C++, le Go étant un langage compilé.

De plus, les logiciels développés de manière optimisée peuvent fonctionner sur du matériel moins cher et plus lent (comme les appareils IoT) ce qui élargit leurs domaines d'applications.

## Machine learning

Comme la plupart des implémentations de machine learning, nous avons choisi python comme langage de programmation. Pour faciliter le développement et la séparation claire des étapes, nous avons utilisé jupyter notebook comme outil de développement. De plus, pour gérer plus facilement les dépendances aux diverses bibliothèques utilisées, nous nous sommes aidé de miniconda pour créer un environnement indépendant des machines. La bibliothèque principale que nous avons utilisé est gensim, une bibliothèque open source, réputée pour ses modèles thématiques (topic model) non supervisés et ses modèles de NLP.

# Architecture technique et méthodologies de développement

## Back-end

### Architecture

L'architecture de notre application utilise largement la philosophie de développement logiciel "Clean Architecture".

La Clean Architecture est un ensemble de principes ayant pour objectif la séparation des responsabilités et le découplage des politiques métiers des détails d'implémentation. Elle a été théorisée et décrite par Robert C. Martin dans son livre "*Clean Architecture : a craftsman's guide to software structure and design*".

Le besoin de la Clean Architecture vient du simple besoin métier de minimiser l'effort humain requis pour développer et maintenir un système logiciel.

Dans la Clean Architecture, le système est représenté selon le découpage suivant :

Le principe le plus fondamental est le fait que les dépendances doivent toujours pointer de l'extérieur vers l'intérieur, car plus on avance vers le centre plus on monte en abstraction.

Les objectifs de la Clean Architecture sont les suivants :

- Indépendance des frameworks
- Testabilité
- Indépendance de l'UI
- Indépendance de la base de donnée

Ainsi, nous avons découpé l'application en deux principaux modules :

- ForkEat.Core : logique métier et logique application

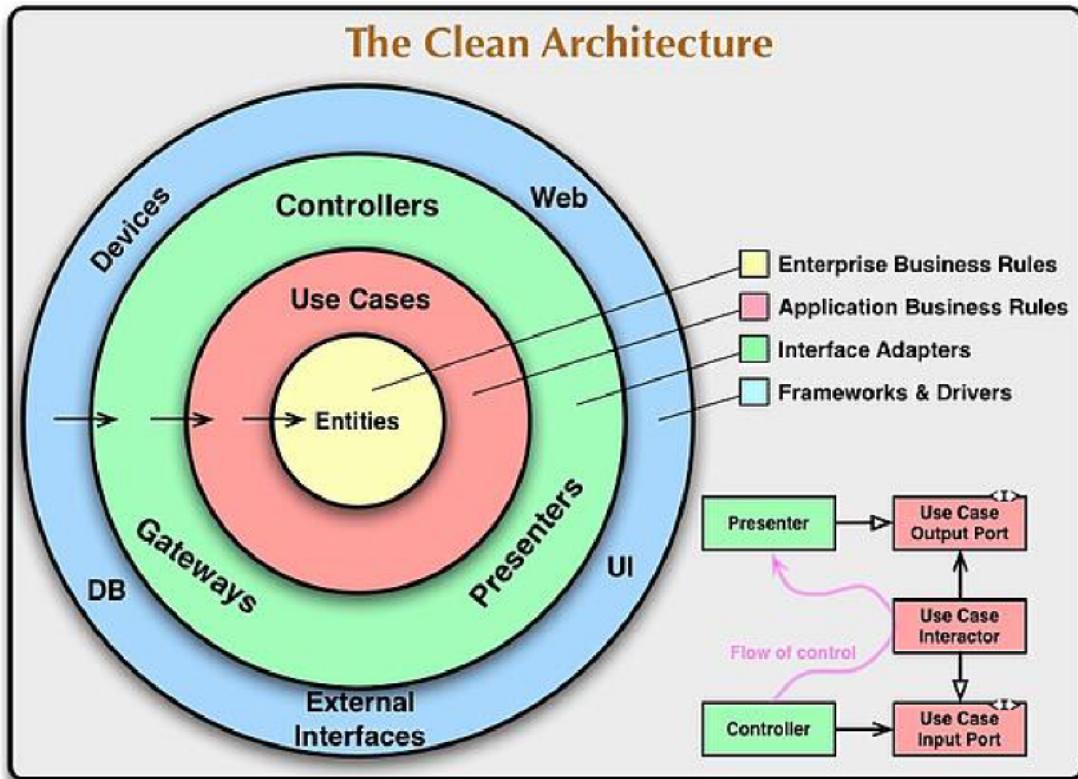


Figure 3: Clean Architecture

- ForkEat.Web : application web et persistance

Dans tout le projet, nous utilisons la programmation par contrat de service à l'aide d'interfaces. Pour éviter les couplages, nous utilisons l'injection de dépendances par constructeur et le conteneur d'inversion de contrôle standard du framework ASP .NET Core, Microsoft.Extensions.DependencyInjection :

```
public class ProductService : IProductService
{
    private readonly IProductRepository productRepository;
    private readonly IProductTypeRepository productTypeRepository;

    public ProductService(
        IProductRepository productRepository,
        IProductTypeRepository productTypeRepository)
    {
        this.productRepository = productRepository;
        this.productTypeRepository = productTypeRepository;
    }

    ...
}
```

Ainsi, nous avons :

- Les classes métier de l'espace de noms ForkEat.Core.Domain implémentent les règles métier (couche *Enterprise Business Rules*)
- Les classes service de l'espace de noms ForkEat.Core.Services implémentent la logique de l'application (couche *Application Business Rules*)
- Les classes contrôleur de l'espace de noms ForkEat.Web.Controllers définissent l'interface HTTP REST de l'application (*Controllers* de la couche *Interface Adapters*)
- Les classes repository de l'espace de noms ForkEat.Web.Database contient le code de persistance (*Gateway* pour la base de donnée de la couche *Interface Adapters*)

## Validation et tests : Test Driven Development

Pour le développement de l'application serveur, nous avons utilisé la méthodologie du Test-Driven Development.

Test-Driven Development est une méthodologie suivant laquelle on écrit le code de test avant d'écrire le code de production. Les tests sont ainsi une spécification du code.

Le TDD déclare les trois lois suivantes :

- Ne pas écrire de code de production avant d'avoir écrit un test qui échoue
- Ne pas écrire plus de code pour un test que ce qui est nécessaire pour le faire échouer
- Ne pas écrire plus de code de production que nécessaire pour faire passer un test

Le cycle court de développement suivant le TDD se découpe donc en trois étapes :

- Écrire un test qui échoue
- Écrire un test le code de production pour faire passer le test
- Procéder à un *refactoring* du code pour s'assurer qu'il soit propre

Les avantages de l'utilisation de TDD sont les suivants :

- Le code de production est par définition testable
- Le code de production est toujours couvert par au moins un test
- La couverture de tests est plus digne de confiance

Ainsi, dans notre application nous avons :

- 122 tests unitaires
- 46 tests d'intégration

Ces tests mettent en œuvre une couverture du code à hauteur de plus de 95%. Pour mesurer la couverture de code, nous avons utilisé un outil tiers appelé CodeCov, et nous l'avons intégré à notre pipeline d'intégration, ainsi, à chaque merge request, nous avions un rapport sur la couverture de code.

## Sécurité

Pour la sécurité nous avons utilisé la technologie JSON Web Tokens. Un token JWT un mécanisme de sécurité sans état, qui est devenu le standard de facto pour la sécurisation des API ReST. Il se compose d'un header, d'une payload et d'une signature. Le JWT est signé à l'aide d'un secret coté serveur pour en prouver l'authenticité, il expire au bout d'un certain temps. Au moment de sa distribution à l'utilisateur lors de la connexion, le token est encodé en base64, l'utilisateur va ensuite le présenter à chacune de ses requêtes pour s'authentifier. Le corps du token contient des informations permettant d'authentifier l'utilisateur pour pouvoir procéder à l'autorisation.

Au niveau technique, nous utilisons le package `Microsoft.AspNetCore.Authentication.JwtBearer` qui fourni une API pour signer des tokens, mais s'intègre également avec le mécanisme d'authentification du framework ASP .NET Core pour la vérification des tokens. Ainsi, les méthodes endpoint peuvent être facilement restreintes derrière l'authentification et leur code peut accéder à l'identité de l'utilisateur.

## Mobile

Nous avons choisi de faire une architecture clean patern dans le but de faire des tests unitaires que nous n'avons malheureusement pas eu le temps de faire.

// Détail de l'archi des fichiers

## Embarqué

Nous avons eu comme idée de développer une partie embarquée, le but étant avec un Raspberry pi équipé d'une caméra de scanner les produits afin de plus facilement les ajouter dans l'application. La création d'une solution embarquée utilisable par le client est toujours quelque chose très délicat. En effet, nous avons commencé par le développement d'une solution permettant à une personne de configurer le Wifi du Raspberry Pi automatiquement depuis son téléphone. Pour des raisons de performance nous nous sommes dirigés vers le langage Go

Le cas d'utilisation pour l'initialisation est le suivant :

- Une personne lance pour la première fois son Raspberry Pi afin de l'associer à son wifi
- Le Raspberry pi va automatiquement créer un Hotspot Wifi et afficher le SSID et le mot de passe sur l'écran.
- Une fois connecté au Wifi l'utilisateur, peut via l'application configurer le SSID et le mot de passe du réseau Wifi domestique de l'utilisateur
- Ainsi le Raspberry pi est connecté au Wifi de manière "User friendly"
- Si jamais le Raspberry Pi ne trouve pas le SSID enregistré, il va de nouveau recréer un réseau temporaire afin de recommencer l'appairage avec le réseau wifi domestique

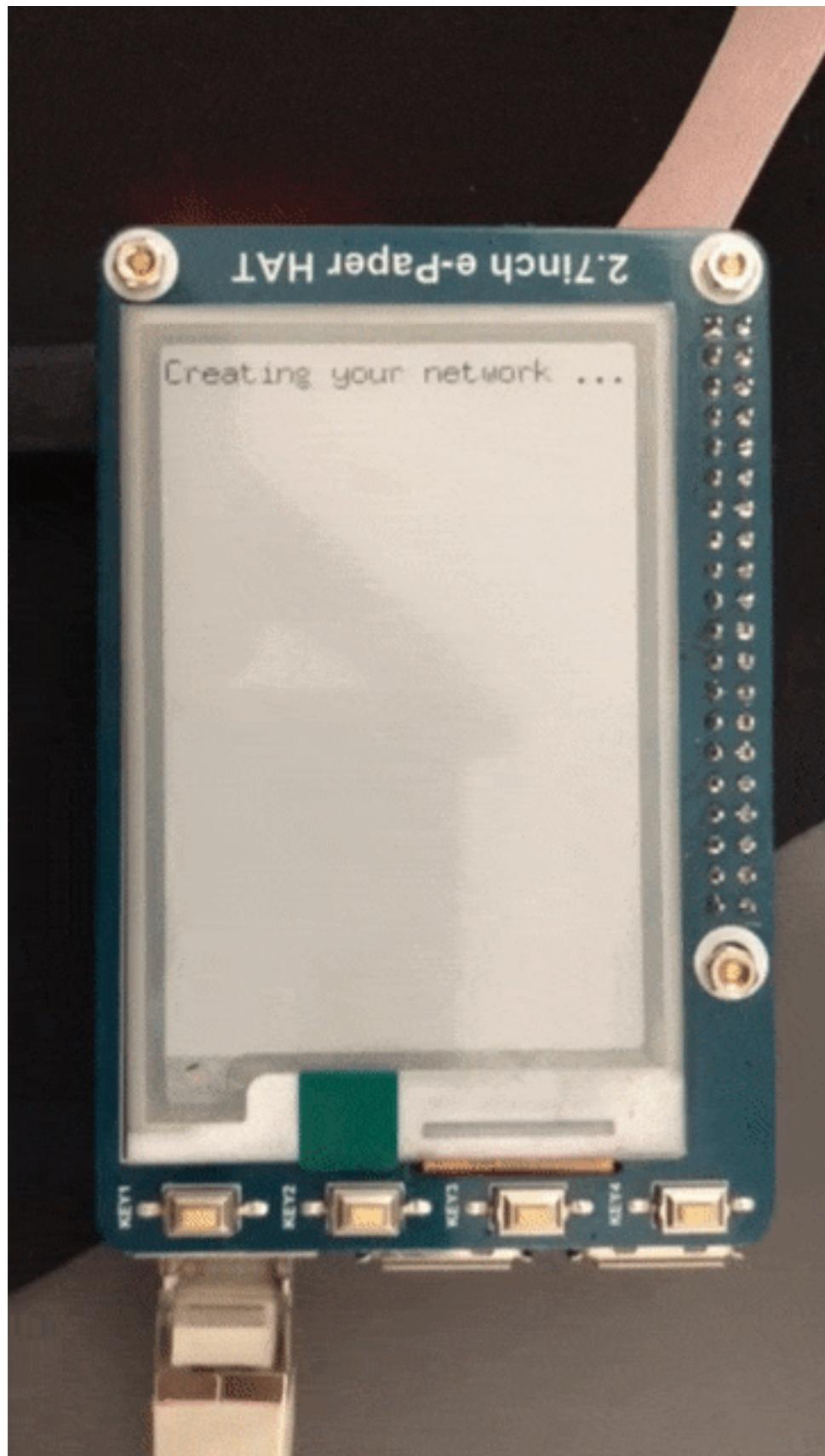


Figure 4: PI\_wifi

Une fois ceci fait l'utilisateur peut scanner le produit les l'ajouter ou le sortir du frigo en utilisant ainsi les boutons présents sur le Raspberry Pi.

## Infrastructure

La partie infrastructure est principalement celle qui détermine la "Quality of Service". C'est pour cela qu'il est important de bien dimensionner l'infrastructure. Ainsi afin de garantir un service optimal et fiable nous avons identifié deux types d'infrastructure. La première étant la CI soit la "Continuous Integration" qui nous permet à chaque itération de valider les ajouts avec une architecture la plus proche de la production. La seconde étant la CD soit la "Continuous Delivery" qui nous permet après chaque validation d'itération de pousser en production les nouvelles versions.

Pour effectuer cette mission, nous avons choisi Drone.io qui est une solution Open-Source et autohébergée qui nous permet de tester les différents services (tests unitaires, test d'intégration, construction de l'image). La grande force de cet outil est qu'il est nativement compatible avec Docker.

### Continuous Integration

La CI, Continuous Integration ou intégration continue en français est une étape permettant de tester continuellement et de manière automatique un projet tout au long de son cycle de vie.

Ainsi, il est possible grâce à l'intégration continue d'automatiser différentes tâches comme : - Lancer la suite de tests unitaires afin de vérifier chaque composant d'une application individuellement. - Lancer la suite de tests d'intégration afin de vérifier si l'interfaçage avec les différents composants de l'application ne présente aucune régression. - Lancer la suite de tests fonctionnels afin de vérifier si fonctionnellement parlant, l'application ne présente aucune anomalie ou de régression. Lorsque des tests d'intégration sont disponibles, cette étape est généralement exécutée après, afin d'utiliser l'infrastructure logicielle fraîchement déployée. - Analyser et remonter les possibles problèmes de qualité de code. - Remonter les possibles failles de sécurité embarquée par des modules tiers téléchargés par un gestionnaire de dépendances comme npm, gem, pip/Poetry, cargo, nuget, maven, etc ...

L'ensemble de ces tâches constitue un pipeline. Généralement, il est exécuté à la suite d'événements sur la forge comme la création d'une demande de fusion (Merge/Pull request selon la forge), lors d'un commit, ou encore lors de la création d'une étiquette (tag).

Si nous prenons l'exemple du backend, nous avons deux pipelines lors d'un commit sur la branch master:

- L'une qui va lancer une base de donnée en docker en parallèle, puis qui va lancer les tests unitaires et d'intégration. Puis nous lançons un test de code coverage afin d'éviter d'avoir une réduction de celui en fonction des itérations. Puis nous déployons l'image de l'intégration sur le Docker Hub grâce au Dockerfile.
- La seconde pipeline a un aspect de documentation, car elle permet de synchroniser les readme entre Github et le DockerHub.

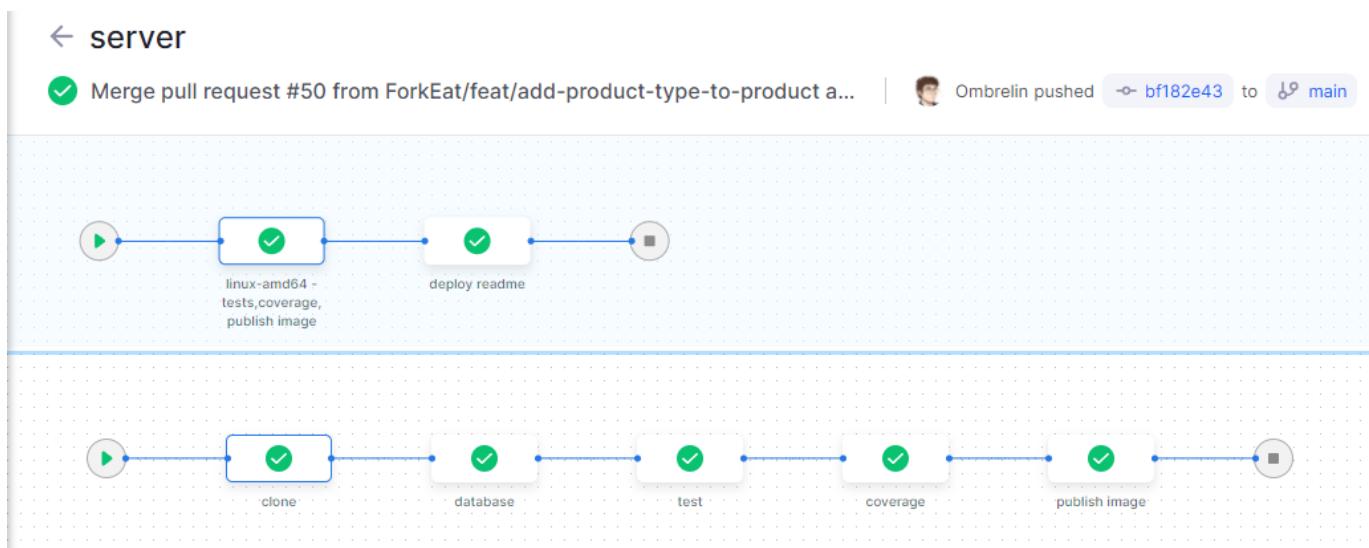


Figure 5: Pipeline

Cependant, nous nous commitons notre code sur une autre branch alors nous aurons uniquement le lancement des tests unitaires et d'intégrations avec le code coverage. Nous n'avons pas besoin de lancer le déploiement d'une image Docker. A contrario, si nous créons un tag sur le dépôt Github, la CI va lancer le déploiement de l'image en utilisant les tags comme version :

TAG		
<a href="#">latest</a>		
Last pushed 3 months ago by <a href="#">thomaslacaze</a>		
DIGEST	OS/ARCH	
<a href="#">95c8bce046d4</a>	linux/amd64	
TAG		
<a href="#">1.1.4</a>		
Last pushed 6 months ago by <a href="#">thomaslacaze</a>		
DIGEST	OS/ARCH	
<a href="#">9f76adc6f8e2</a>	linux/amd64	
TAG		
<a href="#">1.1</a>		
Last pushed 6 months ago by <a href="#">thomaslacaze</a>		
DIGEST	OS/ARCH	
<a href="#">9f76adc6f8e2</a>	linux/amd64	
TAG		
<a href="#">1</a>		
Last pushed 6 months ago by <a href="#">thomaslacaze</a>		
DIGEST	OS/ARCH	
<a href="#">9f76adc6f8e2</a>	linux/amd64	
TAG		
<a href="#">1.1.3</a>		
Last pushed a year ago by <a href="#">thomaslacaze</a>		
DIGEST	OS/ARCH	
<a href="#">c309f9993a35</a>	linux/amd64	

Figure 6: DockerHub

Voici à quoi ressemble le pipeline chargé de lancer nos tests sur la partie Backend (Les parties mobile et embarquée disposent, elles aussi de pipeline chargé de valider les itérations) :

```
services:
- name: database
  image: postgres:alpine3.15
  environment:
    POSTGRES_USER: user
    POSTGRES_PASSWORD: password
    POSTGRES_DB: test

steps:
- name: test
  image: mcr.microsoft.com/dotnet/sdk:6.0
  pull: always
  environment:
    JWT_SECRET: rdtfyguhijgfrdtyg45678
    TEST_DATABASE_URL: postgres://user:password@database:5432/test
    DATABASE_URL: postgres://user:password@database:5432/test
  commands:
    - sleep 15
    - cd ForkEat/
    - dotnet test --logger "console;verbosity=detailed" /p:CollectCoverage=true /p:CoverletOutputFormat=openapi

- name: coverage
  image: plugins/codecov
  settings:
    token:
      from_secret: codecov-token
  files:
    - ForkEat/ForkEat.Core.Tests/coverage.opencover.xml
  required: true

- name: publish image
  image: plugins/docker
  settings:
    auto_tag: true
    auto_tag_suffix: linux-amd64
    repo: thomaslacaze/forkeat-server
    username:
      from_secret: docker_username
    password:
      from_secret: docker_password
  dockerfile: ForkEat/ForkEat.Web/Dockerfile
  when:
    event:
      - push
      - tag
    branch:
      - master
      - main
```

## Continous Delivery

La CD désigne à la fois la livraison en continue (Continuous Delivery) ainsi que le déploiement continu (Continuous Deployment). La livraison continue permet de livrer une nouvelle version d'un logiciel en générant des paquets et binaires prêts à l'emploi pour les différents environnements dont l'application peut être exécutée. Le déploiement continu permet d'automatiser le processus de mise en production de la nouvelle version d'une application sur un ou plusieurs environnements comme un environnement de stage, pré-production ou de production

Nous avons deux types de Continous Delivery :

- La première est une solution auto-hébergée utilisant Traefik en tant que reverse proxy.
- La seconde utilise Terraform et Ansible afin de déployer la stack dans le cloud de Digital Ocean.

**Configuration de Traefik** Voici la configuration du conteneur Traefik nécessaire afin d'avoir un reverse proxy fonctionnel. Ensuite nous pouvons dire à notre conteneur backend les labels à utiliser afin de configurer Traefik sur le bon port. Il est aussi possible d'utiliser Traefik en tant que load-balancer d'avoir plusieurs instances et de rediriger les flux en fonction de la saturation de chaque instance.

Dockerfile

```
version: '2.1'
services:
  traefik:
    image: traefik:latest
    restart: always
    container_name: traefik
    ports:
      - 80:80
      - 443:443
    volumes:
      - /home/thomas/traefik/conf:/etc/traefik/conf
      - /var/run/docker.sock:/var/run/docker.sock:ro
      - /home/thomas/traefik/acme.json:/acme.json
      - /home/thomas/traefik/traefik.yml:/traefik.yml
    environment:
      - CF_DNS_API_TOKEN=${CF_DNS_API_TOKEN}
      - CLOUDFLARE_EMAIL=${CF_API_EMAIL}
      - CLOUDFLARE_API_KEY=${CF_API_KEY}
      - CF_API_EMAIL=${CF_API_EMAIL}
      - CF_API_KEY=${CF_API_KEY}
      - CF_ZONE_API_TOKEN=${CF_ZONE_API_TOKEN}
      - TZ=${TZ}

  forkeat-prod:
    image: thomaslacaze/forkeat-server:linux-amd64
    container_name: forkeat-prod
    restart: unless-stopped
    environment:
      - DATABASE_URL=${DATABASE_URL_PROD}
      - JWT_SECRET=${JWT_SECRET}
    labels:
      - traefik.http.routers.forkeatprod.rule=Host(`forkeat-prod.thomaslacaze.fr`)
      - traefik.http.services.forkeatprod.loadbalancer.server.port=80
```

**Configuration de Terraform avec Ansible** Tous les fichiers mentionnés sont dans le dossier `deploy` du backend

Nous voulions aussi utiliser un cloud distant afin d'appliquer certains outils vue en cours. Nous avons alors utilisé Terraform nous permettant de créer la stack nécessaire au bon fonctionnement de l'application : une machine virtuel linux, une base de données Postgres et un load-balancer. (Voir le fichier `deploy-backend-forkeat-server.tf`)

Une fois la stack déployé nous bon configurer la machine virtuel avec nos outils pour cela nous avons utilisé Ansible. Il est chargé d'installer Docker de lancer le conteneur du backend en fonction des paramètres de la base de données créer via Terraform.

## Machine learning

Pour développer un système de recommandation, il fallait déjà que nous comprenions plus en détail ce que c'était et quel type de système choisir. Il y a donc d'abord eu une importante phase de recherche afin de savoir ce que nous devions faire. Nous allons résumer ici tout ce que nous avons découvert au cours de cette phase.

### Recherches

Les systèmes de recommandation sont une sous-catégorie importante des algorithmes d'apprentissage automatique, et plus précisément des systèmes de filtrage de l'information. Un système de filtrage de l'information est un système qui gère un flux important d'informations, et qui est utilisé pour supprimer les informations redondantes ou indésirables, afin d'exposer les utilisateurs aux informations pertinentes. Un système de recommandation est créé pour prédire la préférence d'un utilisateur pour des articles divers. Il existe différents types d'approches de filtrage : le filtrage collaboratif et le filtrage basé sur le contenu.

**Filtrage collaboratif** Le filtrage collaboratif utilise les similitudes entre différents utilisateurs, l'idée derrière est la "sagesse de la foule". Pour prédire ce qu'une personne va aimer, nous allons étudier ce que d'autres utilisateurs similaires ont aimé. Si l'utilisateur A et l'utilisateur B ont aimé les mêmes articles dans le passé, il est fort probable qu'ils aient des goûts similaires et que nous puissions recommander à l'utilisateur A certains articles que l'utilisateur B a achetés, ou vice versa. Pour que cela fonctionne, nous avons besoin du profil de l'utilisateur, c'est-à-dire de l'historique des achats et des évaluations des articles, le modèle n'a pas besoin de comprendre quels sont les articles, ce qui est un avantage dans certains cas. Il existe deux approches pour le filtrage collaboratif :

- Explicite : l'utilisateur doit explicitement donner une opinion sur l'article, généralement avec une note, pour indiquer au modèle s'il a aimé ou non l'article. Cette méthode est utilisée par exemple par Netflix, elle permet au système de reconstruire facilement l'historique d'un utilisateur et d'éviter d'agrégner des informations qui ne sont pas liées à un utilisateur unique (plusieurs utilisateurs utilisant le même ordinateur).
- Implicite : ce système doit faire une analyse plus poussée pour comprendre le comportement du client et ses goûts, puisqu'il n'y a pas d'information explicite sur ce qu'il aime ou n'aime pas. Cela peut se faire en analysant le nombre de fois qu'un utilisateur a vu un article et en conservant l'enregistrement des achats par exemple (un article que vous avez vu plusieurs fois est susceptible de vous intéresser). Cette méthode est utilisée par Amazon (généralement achetée ensemble) et Facebook. Le principal avantage est qu'il n'est pas demandé à l'utilisateur d'évaluer ses articles et qu'aucune des données récoltées ne contient de biais de déclaration (si de nombreux utilisateurs ont aimé un article, l'évaluation de l'utilisateur peut être influencée).

**Avantages et inconvénients** Le plus grand avantage du filtrage collaboratif est qu'aucune connaissance du domaine n'est nécessaire, puisque la nature du produit n'a pas d'importance pour le système. De plus, ce système peut aider l'utilisateur à découvrir de nouveaux intérêts. Mais il a ses inconvénients comme le démarrage à froid (pour les nouveaux utilisateurs/articles), si un article n'est pas vu pendant la partie entraînement, le système ne peut pas créer de relation et ne sera pas en mesure d'interroger ce modèle (cependant certaines techniques comme la projection dans WALS peuvent aider à résoudre ce problème particulier). Il y a aussi le problème de la sparsité, sur internet, il y a tellement d'articles que même le plus populaire n'aura que quelques commentaires, ce qui rend le calcul plus difficile.

**Filtrage basé sur le contenu** Le filtrage basé sur le contenu exploite les similitudes entre les caractéristiques des produits et les préférences de l'utilisateur pour formuler des recommandations. Les préférences de l'utilisateur sont créées sur la base de ses actions précédentes, et les similarités entre les objets sont basées sur les mots-clés et les attributs calculés par le modèle. Par exemple, si un utilisateur achète un téléphone, le système de recommandation proposera probablement des étuis de téléphone, des films de protection d'écran ou d'autres accessoires de smartphone pour le bon modèle de smartphone, puisqu'ils auront probablement des mots clés communs comme smartphone, le nom du fabricant, le modèle, etc. Cette méthode est généralement adaptée aux applications qui contiennent des informations sur le produit (plus il y en a, mieux c'est), mais peu ou pas d'informations sur le profil de l'utilisateur.

**Avantages et inconvénients** Le principal avantage de cette méthode est que nous n'avons pas besoin des informations d'autres utilisateurs, contrairement au filtrage collaboratif, et que la recommandation est très pertinente pour l'utilisateur puisqu'elle est adaptée uniquement à ce que les utilisateurs ont acheté, la recommandation d'articles de niche est plus susceptible de se produire avec ce type de filtrage. Ce faisant, les recommandations sont plus compréhensibles par l'utilisateur. Avec le filtrage collaboratif, si la recommandation peut effectivement faire découvrir à l'utilisateur un nouvel intérêt, il est également possible que l'utilisateur ne soit pas du tout intéressé par ces nouveaux articles et ne comprenne pas pourquoi ils sont recommandés. Cependant, le système de recommandation sur mesure peut présenter un manque de diversité, ne recommandant que des éléments que l'utilisateur sait déjà qu'il va aimer. Le problème du démarrage à froid existe aussi d'une manière différente, un utilisateur doit avoir fait une action avant que le modèle puisse recommander avec pertinence, donc un nouvel utilisateur n'aura pas immédiatement une bonne recommandation. De plus, à chaque fois qu'un produit est ajouté, nous devons définir son mot-clé pour trouver des similarités entre les produits. Enfin, contrairement au filtrage collaboratif, nous avons besoin d'une connaissance du domaine pour définir les mots clés et les autres relations entre les articles, et ce pour chaque article, et ces liens entre les articles sont ce qui rend le système de recommandation bon ou mauvais, ils sont donc très importants.

Il existe d'autres techniques de filtrage (comme la recommandation par session ou la recommandation mobile), mais nous ne les verrons pas ici puisque les trois développées précédemment sont les principales.

Nous avions prévu d'avoir une version de l'application où de nombreux utilisateurs pourraient partager des recettes, créant ainsi un réseau entre les utilisateurs. Cependant, cela a été considéré comme trop compliqué puisque les algorithmes basés sur le contenu sont plus faciles à mettre en œuvre et qu'il s'agissait de notre premier système de recommandation (de plus, nous n'avions pas prévu d'avoir suffisamment d'utilisateurs pour créer un véritable réseau collaboratif entre eux, nous avons donc décidé de nous en tenir au filtrage basé sur le contenu). De plus, comme cette partie de l'application n'allait être mise en œuvre que tardivement dans le développement, nous ne pouvions pas attendre cette mise en œuvre pour travailler sur la partie Machine learning du projet. Nous avons donc opté pour un filtrage des données basé sur le contenu.

Pour ce projet, nous avons voulu mettre en œuvre un système de machine learning afin de recommander des recettes en fonction des informations de l'utilisateur. Nous avions deux idées de cas d'utilisation qui pouvaient être intéressantes pour notre application. Le premier était de recommander des recettes en fonction de ce que l'utilisateur avait à sa disposition dans sa cuisine, le second était de recommander des recettes en fonction des préférences et des goûts de l'utilisateur.

Nous avons pensé que le système de recommandation le plus simple serait celui basé sur les ingrédients restants. Sur la page d'accueil de l'utilisateur, nous enverrions au système de recommandation les ingrédients disponibles en ce moment dans la cuisine, et il vous proposerait une recette utilisant la plupart de ces ingrédients (avec la nécessité d'en ajouter certains bien sûr). Cependant, ce système ne semblait pas aussi intéressant que le second, nous avons donc décidé de passer au suivant.

**Difficultés prédictes** Nous avons décidé de nous tourner vers l'autre système de recommandation : baser les recommandations sur les goûts de l'utilisateur. La mise en œuvre de cette méthode posait cependant trois gros problèmes.

- Si nous voulions faire une recommandation sur ce que l'utilisateur aimait, nous avions besoin d'un indicateur pour savoir si un utilisateur aimait une recette ou non. Nous avions déjà mis en place un bouton "J'aime", mais il s'agissait davantage d'un usage favori que d'un indicateur permettant de savoir si l'on veut toujours quelque chose de similaire. Par exemple, je veux accéder rapidement à ma recette de chocolat préférée, mais je ne veux pas que mon application me propose que de la tarte au chocolat ou autre. Nous ne voulions pas mettre en place un système d'évaluation, car il n'était pas présent dans la première version de l'application et ce type de fonctionnalité est surtout utilisé pour les applications en ligne (applications avec de nombreux utilisateurs pour évaluer les recettes des autres utilisateurs). Nous avons donc pensé à un moyen de calculer combien de fois un utilisateur a vu la recette dans les dernières semaines/mois et dans l'ensemble (si un utilisateur fait une recette beaucoup de fois, il doit l'aimer) mais c'était en fait trop compliqué à mettre en œuvre dans le temps disponible, pour une première version au moins. Nous avons donc décidé d'envoyer au modèle une recette choisie pour sélectionner 10 recettes similaires.
- Il y avait maintenant le problème des données elles-mêmes. Comme tout algorithme d'apprentissage automatique, nous avions besoin d'un volume suffisant de données pour obtenir des résultats corrects. Et dans l'application elle-même, les recettes étant, dans les deux premières versions, ajoutées par l'utilisateur, nous n'avions pas assez de données pour entraîner notre modèle à trouver lui-même des similarités invisibles. Nous avons donc décidé d'importer un jeu de données de recettes préexistant au premier lancement de l'application (l'application a déjà prérempli des recettes) pour avoir un jeu de données plus conséquent sur lequel travailler.
- Un système de recommandation n'est pas un modèle d'apprentissage automatique comme les autres que nous avons vus auparavant, le jeu de données n'est pas séparé en deux parties : formation et test. La recommandation va trouver des similitudes entre différents éléments et baser sa recommandation sur ces similitudes. Ainsi, il n'est pas possible de séparer les données de l'utilisateur et les données importées, c'est pourquoi nous avons dû importer les données dans l'application, nous ne pouvions pas utiliser les nouvelles données uniquement pour la formation. Ainsi, chaque fois que l'utilisateur ajoute une nouvelle recette, nous devons régénérer un fichier et ré-entraîner le moteur pour le mettre à jour avec les nouvelles données. Ce n'est pas très optimisé et évolutif, mais c'est le plus gros problème des systèmes basés sur le contenu. Vous pouvez en fait l'ajouter en fonction de l'algorithme que vous utilisez en python, mais nous n'avons pas eu le temps de l'implémenter dans la version actuelle.

**NLP (Natural language processing)** Après toutes ces recherches, nous avons réalisé qu'avant d'implémenter un modèle de recommandation, il allait falloir utiliser les algorithmes NLP pour pré-processer les données, car les informations pertinentes d'une recette ne sont pas des simples chiffres ou mots (nous n'allons pas nous baser sur le temps de cuisson ou le nombre d'ingrédients), mais des textes relativement complexes.

Le NLP est un sous domaine de l'étude des langues et de l'intelligence artificielle qui s'intéresse aux interactions entre le langage humain et celui de la machine, en particulier comment programmer un ordinateur pour qu'il soit capable d'analyser une large quantité de langage dit naturel. Il permet de réaliser de nombreuses tâches, dans de nombreux domaines comme le traitement du texte et de la parole :

- La reconnaissance de caractères dans une image (optical character recognition)
- La reconnaissance automatique de la parole, permettant de retranscrire à l'écrit ce qu'une personne dit. C'est d'ailleurs un domaine considéré comme étant IA complet (des domaines de l'IA particulièrement complexe) à cause de la difficulté de percevoir des pauses entre les mots successifs.
- Le text to speech (ou synthèse vocale) qui est l'inverse de la reconnaissance vocale
- La segmentation de mots (ou tokenization) qui permet de séparer des paragraphes en mots (ce qui peut sembler simple pour des langages comme l'anglais ou le français, mais qui est bien plus complexe pour des langages à idéogrammes comme le japonais ou le chinois). Un token représente la plus petite unité de séparation, qui peut être un mot ou un groupe de mot d'une certaine taille (un token est égal à 3 mots par exemple). C'est une des

chose qui va nous servir pour notre problème avec l'utilisation de Bag of Words pour ensuite réaliser du data mining.

Il existe deux méthodes pour faire du NLP, la méthode statistique et la méthode basée sur les réseaux de neurones, qui est la méthode la plus utilisée aujourd'hui.

## Implémentation

Pour réaliser notre système de recommandation, nous avons décidé de nous baser sur les ingrédients, qui nous semblaient être l'information principale d'une recette. Nous aurions pu utiliser les étapes, qui contiennent généralement les ingrédients en plus de comment les utiliser (nous donnant ainsi plus de détails), cependant les étapes sont trop dépendantes du style d'écriture d'une personne, et contiennent parfois trop d'informations inutiles ou redondantes. Cela nous aurait demandé de passer beaucoup plus de temps sur le pre-processing, temps que nous n'avions pas, et qui aurait rendu le développement du modèle beaucoup trop complexe pour un premier modèle de recommandation.

Une fois cela décidé, nous sommes partis à la recherche d'un tutoriel pour nous aider à nous lancer dans notre système de recommandation. Nous tenons d'ailleurs à remercier Rachel Zhiqing Zheng dont les deux articles sur medium nous ont beaucoup aidé à comprendre comment faire fonctionner un système de recommandation de recettes et dont le code présent sur son github nous aura permis de partir sur des bases solides pour développer un système de recommandation.

Comme nous l'avons dit plus haut, le système de machine learning n'est pas intégré à notre application, il se résume donc en un jupyter notebook. Comme ce dernier est commenté et documenté, nous n'allons pas nous attarder sur le détail de chaque fonction. Cependant, nous allons expliquer dans les grandes lignes comment fonctionne le système.

Tout d'abord nous pré-processons les données en traitant les listes d'ingrédients et les mots de ces derniers, en remplaçant les espaces par des \_. Une fois que les termes ont été traités, on peut utiliser le modèle LDA (voir plus bas) pour générer des thèmes (10 dans notre projet) pour lesquels chaque recette aura un degré d'appartenance basé sur ses ingrédients.

Pour utiliser ce modèle, il suffit de sélectionner une recette parmi la base de données sur laquelle les thèmes ont été créés et de calculer les similarités entre la recette choisie et toutes les autres recettes de la base en utilisant un modèle de similarité qui va se baser en partie sur les thèmes de chaque recette. Le modèle nous donnera ainsi une liste de recette qu'il considère être les plus similaires à celle demandée. Pour augmenter la diversité des propositions, nous avons forcé le système à ne pas donner des recettes ayant le même thème principal (pas plus de 4 recettes venant du même thème).

**Latent Dirichlet Allocation** En NLP, LDA est un modèle génératif qui permet de ranger des observations (ici des listes d'ingrédients) dans des groupes non observables, permettant d'expliquer pourquoi certaines données sont plus ou moins similaires. En résumé, chaque document (recette) d'un corpus (base de données) est composé de mots/token (ingrédients), pour le modèle chaque document est un mélange d'un petit nombre de thèmes/sujets, que la présence de chaque mot est attribuable à l'un des sujets du document.

Par exemple, on pourrait considérer un thème comme étant un style de cuisine (asiatique, italienne, ...) ou correspondant à un régime alimentaire (végétarien, sans gluten, ...). Bien entendu, les thèmes ne veulent en réalité rien dire pour la machine, et n'ont pas forcément de sens aussi logique, mais cela permet d'avoir une idée de ce qu'est un thème.

**Le concept de Bag of Word** Le modèle de Bag Of Word est une représentation simplifiée d'un document utilisé en NLP, un texte est représenté comme étant un sac des mots qui le composent, sans considération pour la grammaire ou l'ordre des mots, mais en conservant leur multiplicité (le nombre d'apparitions du mot dans le texte). Le modèle LDA que nous avons utilisé avait besoin d'un Bag of Word.

## Problèmes rencontrés

### Front-end

### Back-end

Au niveau du serveur, nous avons eu des difficultés à implémenter une full-text search, nous pensions que PostgreSQL pourrait supporter cette dernière simplement sur des données structurées, mais ce n'était finalement pas le cas. Après avoir tenté différentes approches nous nous sommes rabattus sur une recherche par pattern de nom en utilisant la clause `LIKE` de SQL.

Aussi, nous avons eu des soucis pour l'exécution des tests d'intégration qui au début s'exécutaient en parallèle, les faisant échouer, car partageant la même base de donnée. Nous avons dû configurer de façon précise le framework de test xUnit afin de les exécuter en série.

## Mobile

Nous n'avons pas eu de difficultés particulières sur le développement. Syncro front & back difficile même si Swagger

## Embarqué

L'embarqué a été compliqué avec le manque de temps alloué à cette partie. De plus, elle nécessite d'avoir un back-end & front-end fonctionnelle afin de pouvoir développer des features complémentaires (configuration du wifi du Raspberry Pi via l'application). C'est pour cela qu'aujourd'hui, la partie embarquée a été mise de côté afin de mieux développer les autres stack.

Cela ne nous a pas empêché d'avoir des difficultés avec la partie embarquée. La lecture des codes barres des produits a été effectuée en Go avec utilisant l'outil Zbar associé à OpenCV. Cependant, la caméra du Raspberry Pi ne disposait pas d'autofocus, ce qui nous a rendu la tâche compliquée afin d'avoir une image nette. Nous avons ensuite utilisé une webcam USB disposant un autofocus, malheureusement la définition de la vidéo n'était pas suffisante. Avant d'être capable d'identifier que le problème provenait des caméras utilisées sur le Raspberry Pi, nous avons testé le bon fonctionnement du programme de reconnaissance de code barre sur un ordinateur portable avec une webcam de haute qualité avec succès.



## Infra

Au début le pipeline chargé d'effectuer les unitaires utilisait une base données distante. Ainsi elle était persistante même après la fin du pipeline. Cela nous a posé beaucoup de problème, parfois certains tests n'effaçait pas entièrement leurs champs ce qui entraînait les futures pipelines en erreur.

Nous avons ainsi choisi après d'utiliser une base de données temporaire qui est créée uniquement durant le pipeline puis supprimé automatiquement. Voici la solution utilisée, la création et destruction sont gérées par la Drone CI automatiquement

```
services:  
- name: database  
image: postgres:alpine3.15  
environment:  
POSTGRES_USER: user  
POSTGRES_PASSWORD: password  
POSTGRES_DB: test
```

## **Machine learning**

Pour la partie recommandation, la plus grosse difficulté à été de trouver des informations pour créer un modèle adapter à la recommandation de recettes. En effet, comme nous n'avons jamais eu de cours sur les systèmes de recommandation ainsi que la gestion d'information textuelle, il a fallu fait un gros exercice de recherche pour comprendre le fonctionnement de NLP ainsi que des systèmes de recommandation (assez différent de système de classification ou de régression classique que nous avions déjà vu). Une fois un tutoriel assez clair trouvé, il a fallu corriger les erreurs contenues ce qui a été assez difficile.

## **Amélioration possible**

Comme nous avons pu le voir entre l'application actuelle et celle que nous avions imaginée, nous n'avons pas eu le temps de développer toutes les fonctionnalités que nous avions souhaitées. Nous aurions particulièrement aimé réaliser la fonctionnalité permettant de planifier les repas et de les présenter sous forme de calendrier. Au niveau du machine learning, nous aurions aussi aimé avoir plus de temps pour mieux gérer les hyper-paramètres du modèle obtenu, afin d'améliorer les recommandations. En effet, même si elles restent pertinentes, certaines recommandations peuvent être assez difficiles à comprendre. De plus, il serait particulièrement intéressant de réussir à intégrer ce système à notre application, et de trouver un moyen d'ajouter des recettes au modèle sans avoir à le réentrainer entièrement.