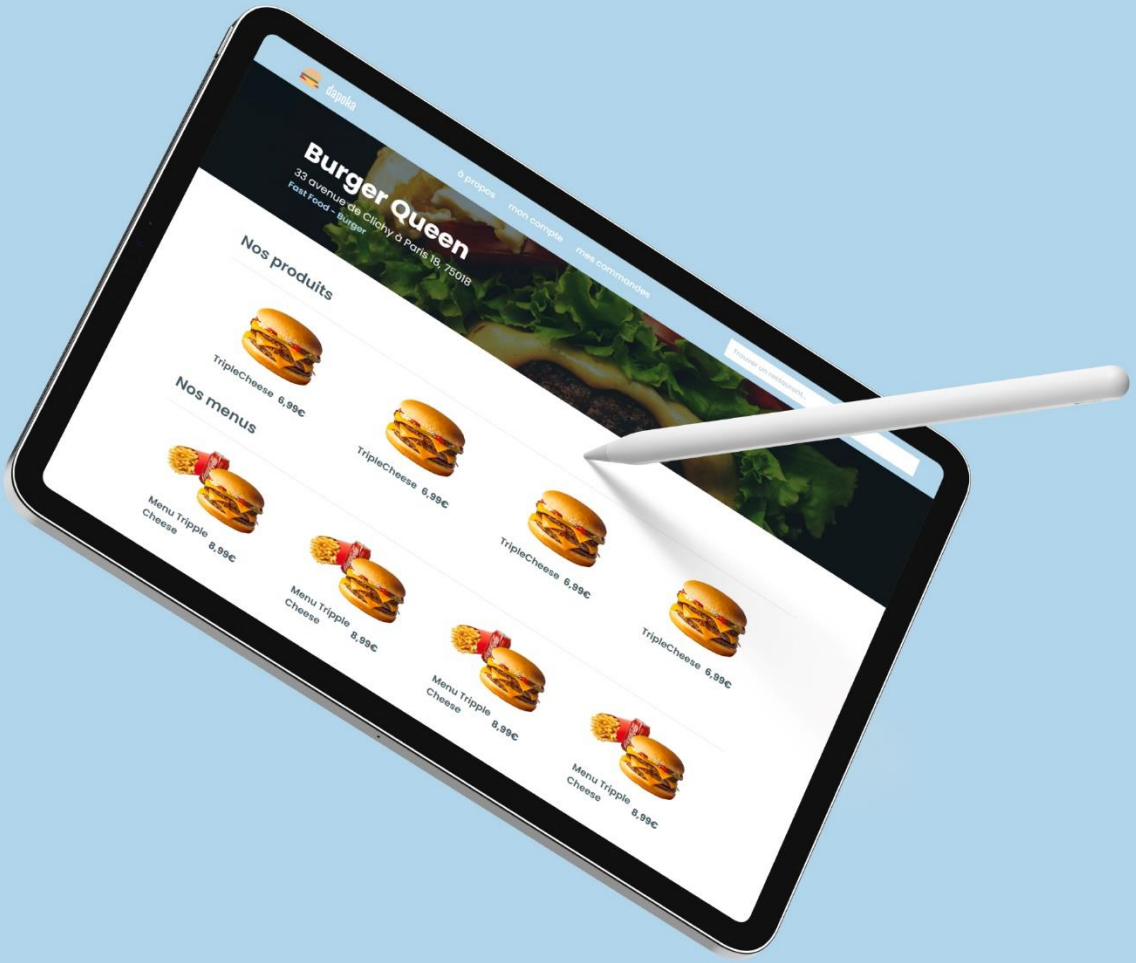
cesi ÉCOLE D'INGÉNIEURS

2022



Rapport de projet élective logiciels

Arthur LECRAS - Lucas AGNES - Dorian BUQUET

dapoka

Table des matières

Table des figures

Introduction

Ce rapport, présentes les différents choix techniques pour la mise en place de la solution finale. Nous commencerons par la reformulation du besoin, nous continuerons ensuite par l'identification des dates importantes : livrable et jalon, une présentation succincte de l'équipe sera présentée et pour finir le détail des spécifications techniques envisagé pour la solution technique.

Reformulation du besoin

Contexte

Tout le long de l'élective développement, nous avons été missionnés pour réaliser une application de livraison de restauration à domicile. Cette plateforme a pour mission de faire le lien entre les restaurateurs qui vendent leurs plats, les livreurs qui doivent transporter les commandes et les clients qui souhaitent se restaurer depuis leur domicile. Cette plateforme doit être simple d'utilisation et intuitive pour ses différents types d'utilisateurs.

Dans ce contexte, nous sommes Dapoka, provenant du grec ancien, de la contraction de "dô" et "apokathistēmon" (δῶ αποκαθιστημον) qui signifie "lier restaurant", la toute nouvelle plateforme Dapoka a pour ambition de devenir le vrai lien entre les consommateurs et les restaurateurs. Ces derniers ont énormément souffert durant la récente crise sanitaire mondiale, et avec les restrictions d'interdiction de déplacement des clients, il est devenu très difficile pour eux d'éviter la faillite. Dapoka facilitera donc la vente en apportant aux restaurateurs une nouvelle clientèle sans avoir besoin de se rendre sur place.

Besoins

Pour réaliser cette plateforme de livraison de restauration à domicile, plusieurs versions d'interfaces sont nécessaires en fonction du rôle de l'utilisateur. Qu'il soit client, restaurateur, livreur ou encore membre de certains services internes de Dapoka comme les services commerciaux, techniques et de développement, l'utilisateur aura accès à une interface adaptée à lui et son besoin.

Le client, celui qui commande un plat chez un restaurateur pour se faire livrer à domicile, a besoin d'une version de la plateforme avec la possibilité de choisir son restaurant, d'effectuer une commande avec un ou plusieurs articles, tout en pouvant modifier, supprimer et payer cette même commande. Le client doit également pouvoir créer un compte afin de réaliser des commandes. Il doit aussi en avoir la gestion, avec la possibilité d'en éditer les informations, mais aussi de le supprimer. Enfin, l'utilisateur client doit également pouvoir suivre l'état de sa commande, éditer son historique des commandes, mais aussi de parrainer ses proches.

Le restaurateur, quant à lui, possède des besoins totalement différents. Si comme le client, il doit être en mesure de gérer son compte, en allant de la création, de l'édition, de la suppression mais aussi du système de parrainage (entre restaurateurs dans ce cas de figure cependant) son interface est différente, car de nouvelles fonctionnalités lui sont consacrées. Le restaurateur pourra créer, modifier, supprimer, éditer un article comme par exemple un plat, une boisson ou même une sauce. Il aura également la possibilité de créer, de modifier ou de supprimer un menu (sous forme de composition d'articles, comme un menu comprenant un plat, un dessert et une boisson). Le restaurateur pourra aussi avoir une gestion des commandes reçues en ayant la possibilité de les visualiser, de les valider, mais également d'en suivre le processus de livraison. Enfin, les restaurateurs seront en mesure d'éditer l'historique des commandes, mais aussi d'avoir accès à des statistiques concernant leurs ventes.

Enfin, le livreur, comme les deux précédents rôles, aura la possibilité de créer, gérer et supprimer son compte tout en parrainant d'autres livreurs. Cependant, il aura accès à une interface utilisateur totalement différente. Il aura le choix d'accepter ou de refuser de prendre en charge une livraison, mais également d'acquitter cette même livraison.

Maintenant, que nous avons listé les besoins des principaux acteurs du cycle d'utilisation de notre plateforme, nous allons nous intéresser aux besoins des développeurs tiers et des services internes : commerciaux et techniques. Les développeurs tiers sont les derniers à avoir dans leurs besoins la création, l'édition et la suppression de compte. Ils doivent pouvoir éditer les composants disponibles de la plateforme et doivent également être en mesure de les télécharger.

Les membres du service commercial de Dapoka ont quant à eux des besoins totalement différents des utilisateurs classiques que nous avons énumérés plus haut. Ils ont besoin de pouvoir éditer, suspendre, supprimer des comptes clients, mais aussi d'avoir un accès aux statistiques de la plateforme et aux tableaux de bords de suivi de processus de commande en temps réel (avec par exemple, l'accès à la passation commande, à l'acceptation d'une commande, ou encore à l'acceptation et l'acquittement d'une livraison, et enfin le chiffre d'affaires transactionnel global en cours).

Enfin, le service technique devra être en mesure d'ajouter, supprimer des composants réutilisables, d'éditer les logs de connexions, les statistiques de performances des serveurs et micro-services, et les logs de téléchargement des composants réutilisables. Les membres du service technique de Dapoka doivent aussi pouvoir organiser les routes pour les résolutions des demandes entrantes et pouvoir déployer de nouveaux services ou micro-services sans interruption du service client.

Nous avons décidé de réaliser un diagramme bête à cornes, afin d'illustrer l'analyse fonctionnelle du besoin. Le diagramme bête à cornes est un schéma qui permet de visualiser l'utilité pour l'utilisateur d'un produit, en l'occurrence la plateforme Dapoka, afin de savoir si elle répond à des besoins. Cet outil qui nous permet de comprendre correctement les besoins des utilisateurs répond à trois questions :

- À qui rend-il service ?
- Sur quoi agit-il ?
- Dans quel but ?

C'est grâce à ce diagramme que nous allons pouvoir démontrer que la conception de notre plateforme est nécessaire.

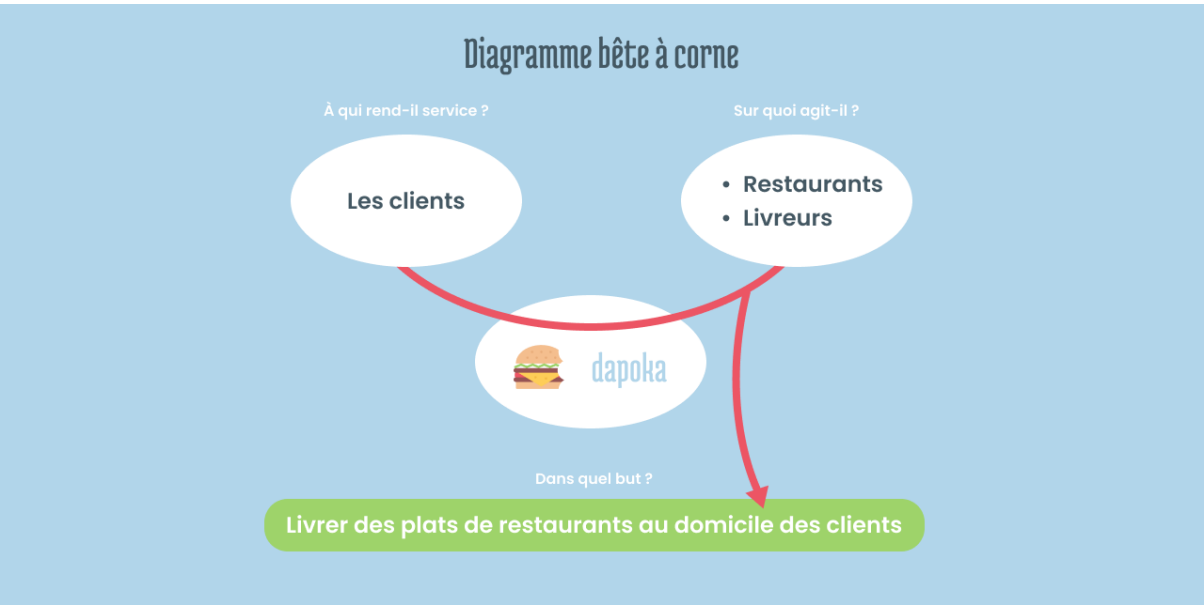


Figure 8 : Diagramme bête à corne

Identification des date importantes

Après la lecture en groupe du cahier des charges nous avons relevé deux types d'intervention, les jalons et les livrables.

Nous avons repéré quatre démonstrations :

1. la première s'effectuera le jeudi 9 juin, elle représentera le livrable n°0 et 1,
2. la deuxième se déroulera le mercredi 22 juin, nous y présenterons l'avancée du projet,
3. la troisième se passera deux jours après la première, le vendredi 24 juin et fera l'objet d'une démonstration technique,
4. la quatrième sera le vendredi 1 juillet, la présentation finale du projet ainsi qu'un récapitulatif éventuel des tâches non accomplies. Cette présentation fera l'objet des livrables 2 à 6.

Nous avons également décelé sept livrables :

1. livrable n°0 : le document récapitulant l'analyse du cahier des charges, à rendre le jeudi 9 juin,
2. livrable n°1 : le support de présentation montrant l'analyse du cahier des charges, à rendre le jeudi 9 juin,
3. livrable n°2 : le support de présentation de la présentation final, à rendre le vendredi 1er juillet,
4. livrable n°3 : l'oral de présentation, se déroulera le vendredi 1er juillet,
5. livrable n°4 : la solution finale, à rendre le vendredi 1er juillet,
6. livrable n°5 : les questions et réponses en groupe, se déroulera le vendredi 1er juillet,
7. livrable n°6 : les questions et réponses individuelles, se déroulera le vendredi 1er juillet.

Planning prévisionnelle

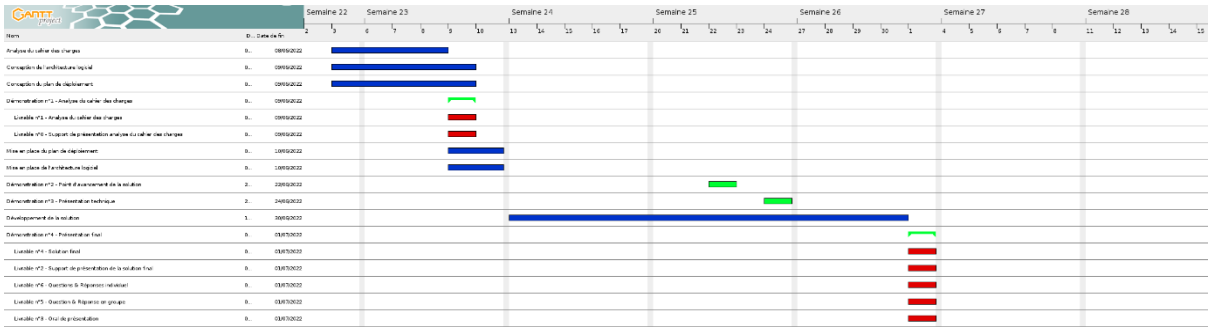


Figure 1 - Diagramme de Gantt prévisionnel et récapitulatif des phases du projet.

Planning réel

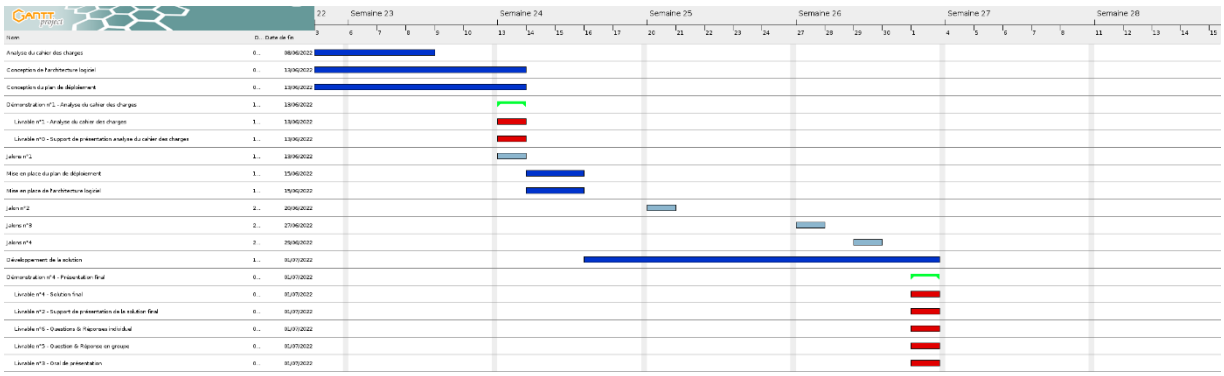
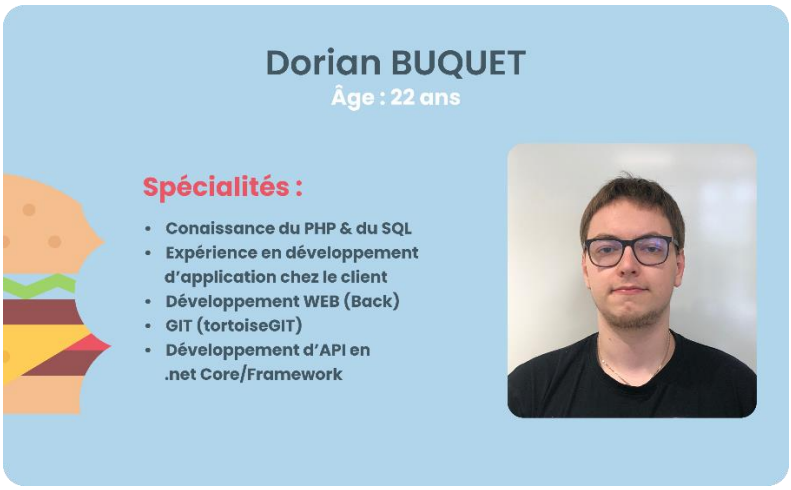


Figure 2 - Diagramme de Gantt et récapitulatif des phases du projet.

Présentation de l'équipe

Avant d'entrer dans le vif du sujet, voici une brève présentation des membres de notre équipe de projet menée par Arthur LECRAS. Comme vous pourrez le constater, nos profils différents vont pouvoir se compléter afin de mener notre projet à bien, une véritable richesse pour Dapoka.



Organisation du groupe et outils

Pour l'organisation du projet, nous avons opté pour la méthodologie Agile Scrum, car elle répond totalement aux besoins cycliques de gestion entre les prosits et les différentes phases de projets comme nous pouvons l'observer sur le schéma suivant :

Figure 3 - Schémas représentant les prosits et les phase de projet.

Pour planifier et organiser le projet avec la méthodologie Scrum, nous avons utilisé Trello pour l'organisation et la répartition des tâches, et Gantt pour effectuer un planning prévisionnel.

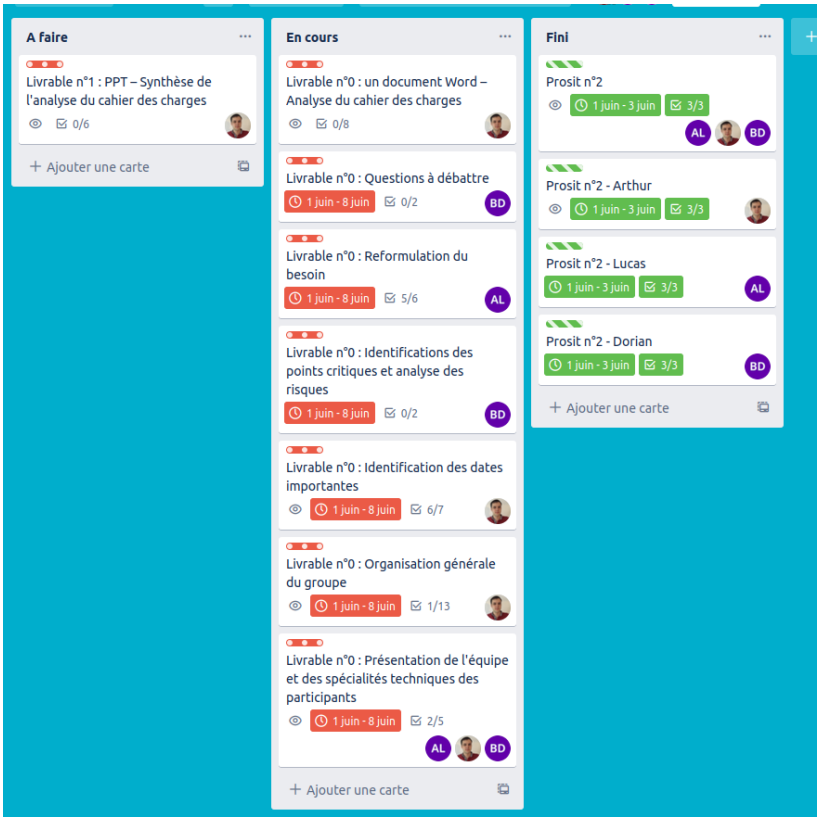


Figure 4 - Image du Trello.

Pour pouvoir communiquer ou débattre tout le long du projet, nous utilisons l'outil collaboratif discord.

Durant ce projet, nous aurons besoin de produire plusieurs documentations, pour cela, nous utiliserons la suite office 365 dont les documents résultants sont stockés sur un OneDrive afin de pouvoir y travailler à plusieurs. Un autre outil vient s'ajouter aux documents dans le but de produire des schémas explicatifs : Draw.Io.

Avant de commencer à construire les différents modules du projet pour obtenir l'application fonctionnelle, nous devons passer par plusieurs phases de réflexion. La première n'est autre que la phase de conception, et nous y utiliserons l'outil Visual Paradigm pour produire les différents diagrammes UML. La seconde phase de réflexion concerne le design des différentes interfaces graphiques et pour cela, nous utilisons l'outil collaboratif Figma.

À la suite de ces phases de réflexions, nous pourrons commencer à créer la solution technique, nous aurons besoin de plusieurs IDE, Visual Studio code et WebStorms selon les préférences de chacun pour le développement WEB et de Visual Studio pour développer l'application C#. Pour centraliser le code sources des solutions, nous utilisons le système de gestion de version Git héberger sur GitHub, l'outil Git kraken viendra en complément afin d'appréhender plus facilement Git.

Architecture

L'architecture générale et logicielle du projet nous est imposée, celle-ci doit respecter un équilibre entre une architecture micro-service et orienté service, respectant l'architecture logiciel "Layered pattern".

Ce type d'architecture comporte plusieurs avantages :

- La maintenance d'une fonctionnalité est moins lourde, en effet, toutes les fonctionnalités sont divisées en plusieurs couches dont certaines sont divisées en plusieurs micro-services. Par exemple : la mise à jour d'un composant graphique dans ce type d'architecture ne nécessitera pas la modification de la connexion aux bases de données, seuls les composants graphiques seront affectés par cette maintenance.
- Le développement de l'application est divisé en plusieurs projets, ce qui permet à une équipe de développement de travailler sur une même fonctionnalité, mais en parallèle par exemple.

Schéma globale

Nous avons, après analyse du cahier des charges, schématiser l'architecture de la manière suivante :

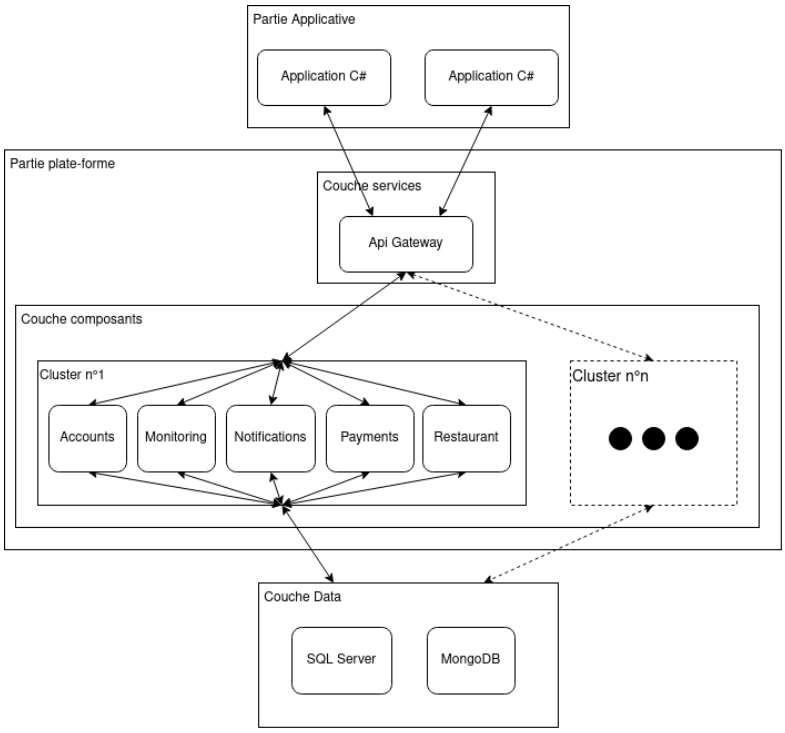


Figure 5 - Schéma global de la solution.

Application C#

Dans le cadre du projet Dapoka, notre groupe de projet s'est vu confié la mission de réaliser une application permettant aux commerciaux et aux techniciens d'effectuer certaines tâches telles que :

- Suspendre le compte de certains utilisateurs.
- Accéder aux diverses commandes en cours de réalisation/livraison.
- Accéder aux logs.

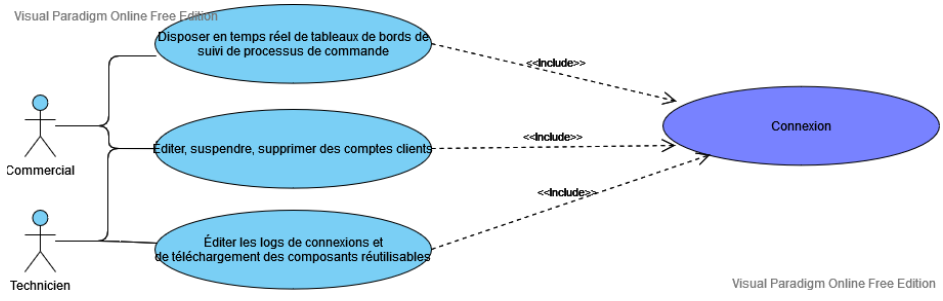


Figure 6 - Diagramme de cas d'utilisation de l'application C#.

Nous avons choisi de réaliser l'application en C# .Net WPF par soucis de praticité et de temps. L'application contient :

- Une page de login permettant aux utilisateurs dont le compte est actif de se connecter.
- Une page Main Windows avec la liste des utilisateurs ainsi qu'une barre de recherche.
- Une page permettant d'afficher les logs
- Une page affichant les commandes.

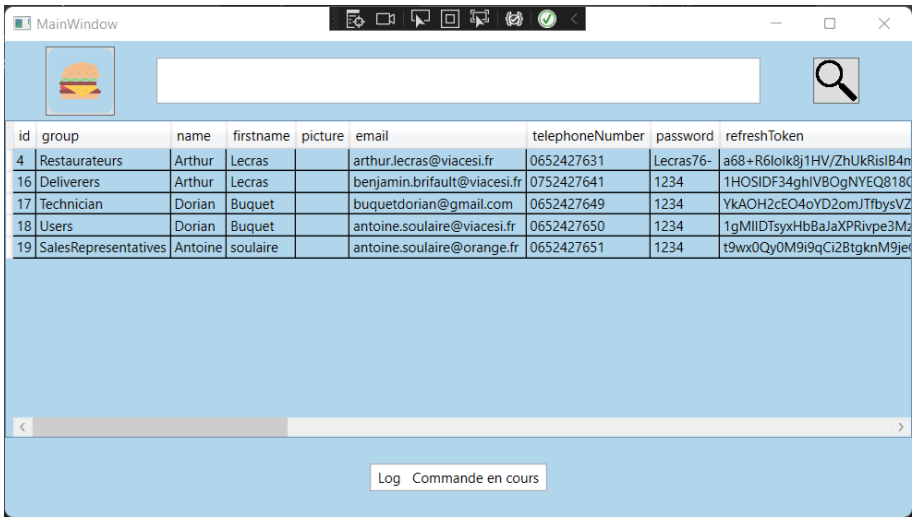


Figure 7 - Interface graphique de la gestion des utilisateurs.

La classe HttpClient a été créée avec un Design Pattern Singleton pour s'assurer de n'avoir qu'une instance d'HttpClient réalisant les appels à l'API pour récupérer les données.

La classe JsonDeserializer désérialise les données récupérées via le HttpClient pour qu'elles soient affichées dans les différents Windows.

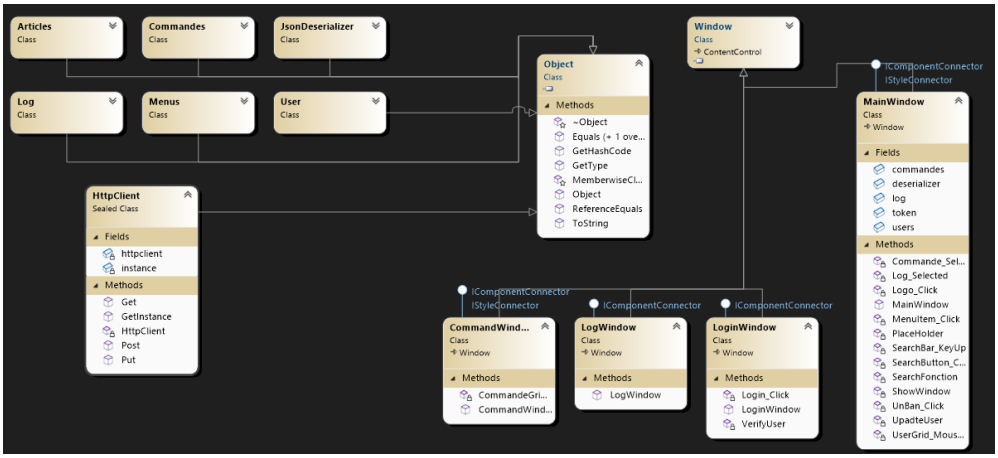


Figure 8 - Diagramme de classe de l'application C#.

Application VueJs

Ensuite, notre équipe devait réaliser la plateforme côté utilisateur du projet Dapoka, sous forme de site web. La plateforme est destinée à trois types d'utilisateurs :

- Les clients qui souhaitent commander des plats
- Les restaurants qui vendent leurs produits
- Les livreurs qui doivent effectuer leurs commandes

Les contraintes et les spécifications fonctionnelles étaient nombreuses pour chaque type d'utilisateur. La production de la plateforme Dapoka s'est donc déroulée en cinq étapes :

1. Création de la charte graphique
2. Conception de maquettes à l'aide du logiciel Figma
3. Réflexion du fonctionnement de l'UX et des différentes pages
4. Création du projet Vue.js et de l'ensemble des vues
5. Liaison avec l'API afin de la consommer

Figure 9 - Identité visuelle de dapoka.

Avant d'être réalisé, chacune des vues ont d'abord fait l'objet d'un maquettage minutieux afin de créer une expérience utilisateur adéquat, favorisant l'attention et incitant les utilisateurs à consommer les produits proposés par nos partenaires restaurateurs.

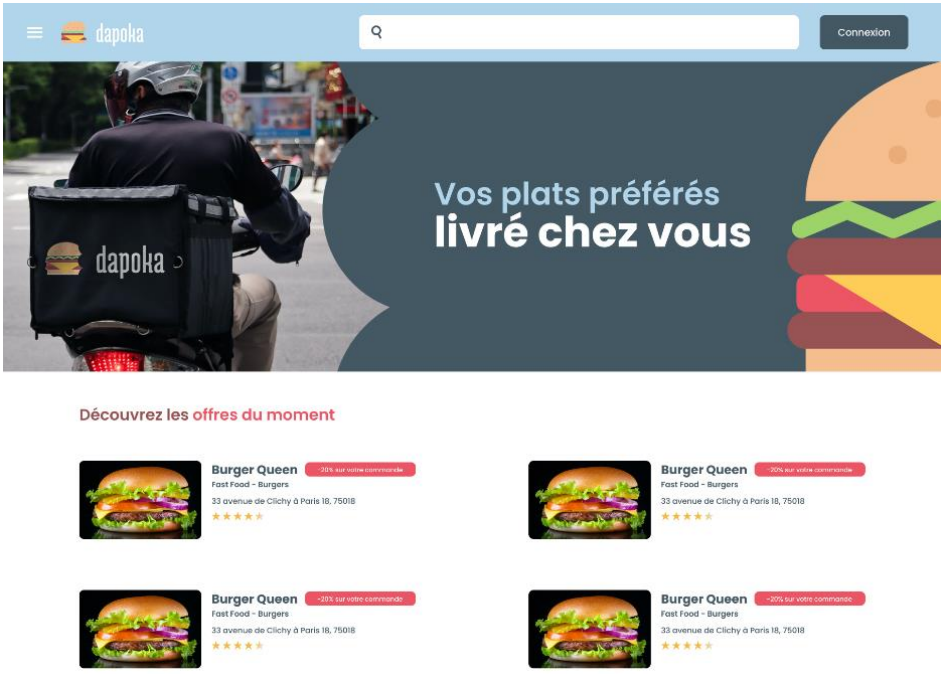


Figure 10 - Maquette de la page d'accueil.

Le projet a été réalisé avec la version 3.2 de Vue.js, l'organisation se divise en deux dossiers : un dossier components qui contient les divers composants utilisés par la plateforme et un dossier page contenant toutes les vues de la plateforme web.

Les vues sont responsives et codées en HTML/CSS. La navigation entre les pages se fait grâce à l'ajout de la bibliothèque Vue Router. Le Framework Bootstrap est également utilisé afin de peaufiner les vues et le responsive design.

La consommation de l'API se fait à l'aide de l'import de la dernière version d'Axios, qui permet de faire directement des commandes tel que POST ou GET pour récupérer directement les informations de la base de données et les modifier.

API Gateway

Une api Gateway a été mise en place pour deux raisons. Faire la liaison entre les clusters qui contiennent des instances docker des différents micro-services et le frontend (C# et/ou application VueJs) en utilisant un load balancing afin de garantir une continuité de service en cas de panne sur un micro-service. Et, Vérifier la validité des entrées d'une route : paramètre envoyé et gestion des tokens.

L'api a été développée à l'aide de NodeJS et Expresse en TypeScript, Axios a été utilisé pour la transmission des requêtes reçues au bon micro-service.

Avant l'envoi d'une requête aux différents services demandés, l'API vérifiait avec trois middlewares plusieurs paramètres, dont les caractéristiques sont définies dans un fichier de configuration json :

1. La validité des paramètres : headers, body et params.
2. La validité du token d'identification des applications : C#, VueJs, Proxy, etc...
3. La validité du token d'identification d'un utilisateur.

Prenons l'exemple de la route pour accéder aux menus d'un restaurant, le code dans le proxy sera le suivant :

```
menusRouter.get('/restaurant/menus/:id_restaurant', [
    header("token-api").exists().isString(),

    param("id_restaurant").exists().isNumeric()
], Utils.validateExpress, Utils.tokenApiMiddleware, Utils.tokenMiddleware, (req: Request, res: Response) => {
    Utils.setResponse(Utils.proxyTransport( config: {
        method: "GET",
        url: `3005/menus/${req.params.id_restaurant}`
    }), req, res,  api: "Restaurants");
});
```

Figure 11 - Exemple de route gérer dans l'api proxy.

Nous y voyons entre crochets, l'ensemble des règles qui vont être testées dans le middleware "Utils.validateExpress". Les deux autres "Utils.tokenApiMiddleware" et "Utils.tokenMiddleware" dépendront des règles inscrites dans le fichier json. Pour l'exemple précédent les règles sont :

```
{
  "path": "/restaurant/menus/:id_restaurant",
  "methode": "GET",
  "api-key": [
    "210acc4851961df7553144e8b1d77f6f"
  ]
},
```

Figure 12 - Exemple de format des règles dans permissions.json.

Cela signifie que seule l'application VueJs peut accéder à cette route, toutes les autres auront un refus. Nous avons choisi de faire un load balancing de sécurité, c'est-à-dire que celui-ci va envoyer un ping aux clusters afin de vérifier l'état de chacun et de mesurer la rapidité de la réponse et prendre le plus accessible.

Micro-services

L'ensemble des micro-services ont été développés avec les mêmes technologies que l'api Gateway : NodeJs et Express en TypeScript et l'utilisation d'Axios pour toutes requêtes externes au micro-service. Pour gérer l'accès aux bases de données, deux ORM ont été exploités : sequelize pour la base de données MS SQL server et mongoose pour la base de données non relationnel mongoDB.

Nous avons listé, cinq micro-services :

- Accounts : va encadrer la gestion des utilisateurs : création, modification, suppression, mais également l'authentification aux comptes, la génération d'un nouveau mot de passe en cas d'oubli et l'activation du compte.
- Monitoring : va gérer la publication et l'envoie des logs des micro-services.
- Payments : s'occupe de tout ce qui est lié au paiement de commande et remboursement.
- Restaurants : va laisser la possibilité aux restaurateurs de modifier les menus, articles de leur boutique et les autres utilisateurs pour y accéder afin de gérer les commandes, et accéder aux statistiques selon leur droit.

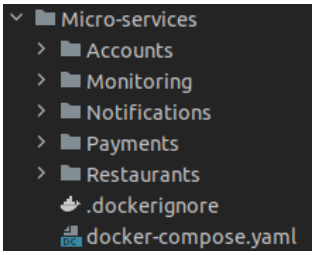


Figure 13 - Arboraissance des micro-services.

Sur l'image précédente, représentant l'arborescence, des projets micro-service, nous observons un fichier "docker-compose.yaml". Nous avons décidé afin de répondre convenablement aux bonnes pratiques de gestion du micro-service de conteneuriser l'ensemble des projets.

Infrastructure et déploiement

Stratégie (ou Architecture) de déploiement

Nous avons défini plusieurs critères pour définir un déploiement sans soucis :

- n'affecte pas ou très peu les utilisateurs de nos services, exemple, nous actualisons le système de paiement, les utilisateurs devront pouvoir continuer à payer,
- être capable de faire marche arrière en cas de problème, ou de non-satisfaction des modifications apportées par le déploiement d'une nouvelle version, exemple changement de design d'une fonctionnalité.

Pour répondre à ces deux contraintes, nous avons jugé que le déploiement canary correspondait le plus à ces contraintes. Il présente, cependant, le principal inconvénient d'être complexe et coûteux à mettre en place. Par ailleurs, il fonctionne en répartissant, une proportion d'utilisateurs sur l'ancienne version et la proportion restante sur la nouvelle version.

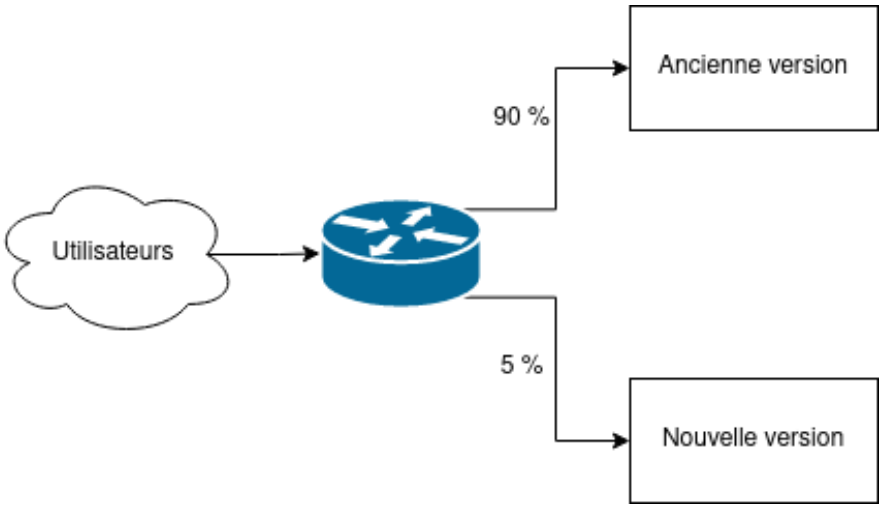


Figure 14 - Schémas d'explication du déploiement canary.

Infrastructure

Nous avons l'obligation de tester notre plateforme avant la mise en production définitive. Il nous faut donc une infrastructure capable de répondre à ce besoin. Pour cela, nous effectuons une étape entre le développement local et la mise en production que nous appellerons la mise en développement.

La mise en développement consistera à "push" le code source sur une branche nommée "development". Lorsque le code source sera reçu par cette branche, un workflow va déployer le code source de la même manière que la mise en production, mais dans un environnement intermédiaire.

Pour revenir, sur le choix de notre infrastructure, celle-ci doit contenir aux moins deux environnements :

- L'environnement de développement n'a pas besoin d'être accessible par le grand public, il sera alors encadré par l'infrastructure "On-premise".
- L'environnement de production est un environnement qui nécessite une attention particulière, car elle reçoit tous les utilisateurs et doit être assez robuste pour être doublé lors des déploiements. Pour cela, nous avons retenu le "private cloud" qui contrairement au "public cloud" de laisser uniquement le locataire utiliser les ressources de la machine.

L'ensemble des architectures devront pouvoir héberger plusieurs conteneurs Docker, il nous faut donc la possibilité d'installer des logiciels sur les machines. Pour l'infrastructure de l'environnement de développement, c'est assez facile, les serveurs sont hébergés au sein des locaux. Pour l'infrastructure de la production, nous il devra respecter le modèle IaaS.

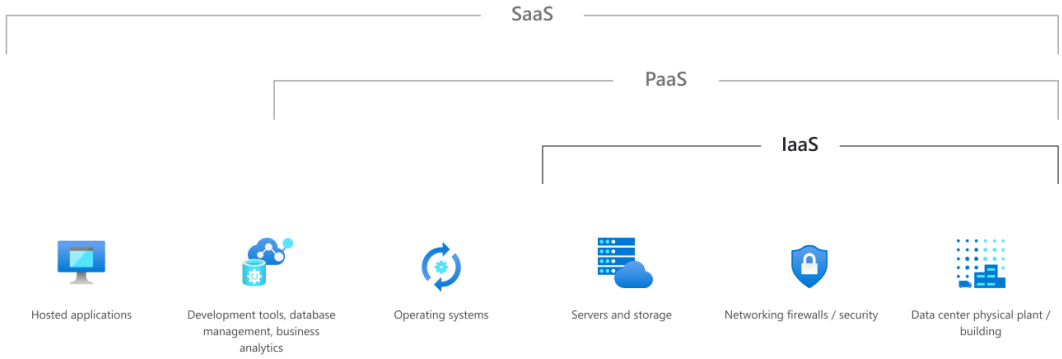


Figure 15 - Comparatif des différents modèles d'infrastructures.

Glossaire

Analyse fonctionnelle du besoin

L'analyse fonctionnelle du besoin est une analyse des fonctionnalités d'un produit selon les besoins de ses futurs utilisateurs..... 6

Diagramme bête à cornes

Le diagramme bête à cornes se présente sous forme d'un graphique permettant de démontrer si un produit ou service s'adapte aux besoins des utilisateurs. Il correspond à l'image d'une bête à cornes, plus précisément à la tête d'un taureau. La corne est notamment représentée par la ligne qui relie les deux bulles qui se trouvent au-dessus..... 6

Interface utilisateur

Une interface utilisateur, ou User Interface en anglais (UI), désigne l'ensemble des éléments graphiques et textuels qui permettent une interaction entre l'utilisateur et le site internet, l'application ou le logiciel. Cela inclut donc l'ensemble des items interactifs et « cliquables », tels que la barre de navigation du site, les boutons ou les liens 5

Plan de déploiement

Le plan de déploiement est un outil efficace pour les équipes responsables de l'amélioration de la qualité, car il leur permet de définir clairement la façon dont elles envisagent de planifier le déploiement des améliorations réalisées 18