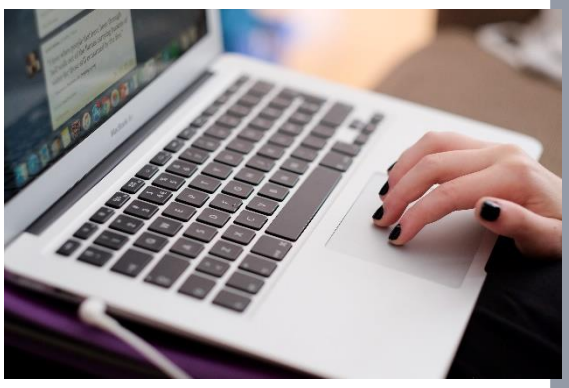


LUCIE CHEVANCHE

DOSSIER PROJET

11 SEPTEMBRE 2024

CONCEPTEUR DEVELOPPEUR D'APPLICATION

WILD CODE SCHOOL

TERASCOP

Remerciements

Je tiens tout d'abord à remercier toute l'équipe de la Wild Code School pour ce qu'elle m'a apporté durant la formation et plus particulièrement à notre formateur, Pierre. Merci pour ta patience et ton soutien. Je remercie aussi notre promotion pour l'entraide et la bonne ambiance durant cette année.

J'aimerais également remercier l'entreprise TERASCOP pour sa confiance et pour la bienveillance dont a fait preuve l'équipe avec moi.

Enfin, merci à ma famille, mon conjoint et mes enfants qui m'ont soutenu et accompagné tout au long de cette aventure qui ne fait que commencer.

Ils ont tous contribué à ma reconversion et méritent à ce titre, toute ma gratitude et mon respect.

Table des matières

1. Introduction

Ayant tout juste terminé une formation initiale de 5 mois à la Wild Code School, j'ai décidé de poursuivre mon apprentissage vers une alternance.

J'ai pu intégrer la société Terascop en septembre 2023. Je suis en contrat de professionnalisation de 12 mois en alternance sur un rythme de 3 semaines en entreprise, 1 semaine en formation.

Pour cette nouvelle formation je suis resté fidèle à la Wild Code School qui assure ma formation théorique.

Au sein de Terascop, nous sommes une équipe de 8 personnes : 2 codirigeants, 1 directeur technique et 5 développeurs repartis en 2 équipes (travaillant sur 2 projets différents).

Durant mon année d'alternance, j'ai pu travailler personnellement sur divers projets.

Ce dossier portera sur un projet pédagogique initialisé dans le cadre de ma formation à la Wild Code School. Il s'agit d'un projet réalisé en collaboration avec 3 autres développeurs en formation également. Nous avons découvert et mis en pratique de nombreux outils et langages différents comme GraphQl, Apollo, Jest, Github Actions et Docker.

En effet, les projets menés au sein de l'entreprise sont strictement confidentiels et ne peuvent être utilisés dans le cadre de ce dossier.

2. Liste des compétences mises en œuvre dans le cadre du projet

2.1 Développer une application sécurisée

Installer et configurer son environnement de travail en fonction du projet

Développer des interfaces utilisateur

Développer des composants métier

Contribuer à la gestion d'un projet informatique

2.2 Concevoir et développer une application sécurisée organisée en couches

Analyser les besoins et maquetter une application

Définir l'architecture logicielle d'une application

Concevoir et mettre en place une base de données relationnelle

Développer des composants d'accès aux données SQL et NoSQL

2.3 Préparer le déploiement d'une application sécurisée

Préparer et exécuter les plans de tests d'une application

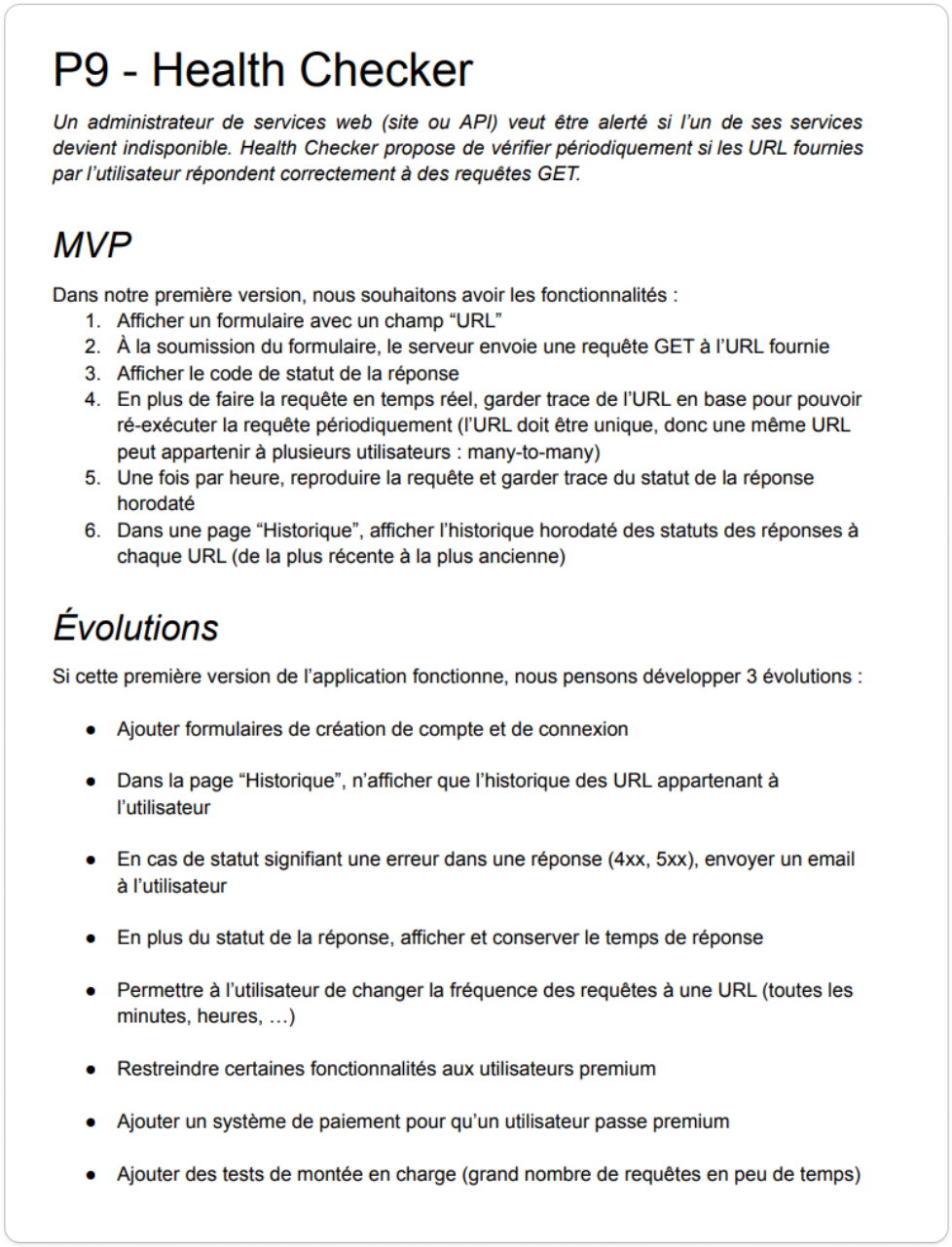
Préparer et documenter le déploiement d'une application

Contribuer à la mise en production dans une démarche DevOps

3. L ’expression des besoins du projet pour définir les objectifs

Dans le cadre de la formation, plusieurs projets nous ont été proposé afin de réaliser un projet de fin d’année en groupe.
Nous sous sommes positionnés en groupe pour travailler sur ce projet pendant les 6 derniers mois de la formation.

La fiche technique du projet était la suivante :



P9 - Health Checker

Un administrateur de services web (site ou API) veut être alerté si l’un de ses services devient indisponible. Health Checker propose de vérifier périodiquement si les URL fournies par l’utilisateur répondent correctement à des requêtes GET.

MVP

Dans notre première version, nous souhaitons avoir les fonctionnalités :

1. Afficher un formulaire avec un champ “URL”
2. À la soumission du formulaire, le serveur envoie une requête GET à l’URL fournie
3. Afficher le code de statut de la réponse
4. En plus de faire la requête en temps réel, garder trace de l’URL en base pour pouvoir ré-exécuter la requête périodiquement (l’URL doit être unique, donc une même URL peut appartenir à plusieurs utilisateurs : many-to-many)
5. Une fois par heure, reproduire la requête et garder trace du statut de la réponse horodaté
6. Dans une page “Historique”, afficher l’historique horodaté des statuts des réponses à chaque URL (de la plus récente à la plus ancienne)

Évolutions

Si cette première version de l’application fonctionne, nous pensons développer 3 évolutions :

- Ajouter formulaires de création de compte et de connexion
- Dans la page “Historique”, n’afficher que l’historique des URL appartenant à l’utilisateur
- En cas de statut signifiant une erreur dans une réponse (4xx, 5xx), envoyer un email à l’utilisateur
- En plus du statut de la réponse, afficher et conserver le temps de réponse
- Permettre à l’utilisateur de changer la fréquence des requêtes à une URL (toutes les minutes, heures, …)
- Restreindre certaines fonctionnalités aux utilisateurs premium
- Ajouter un système de paiement pour qu’un utilisateur passe premium
- Ajouter des tests de montée en charge (grand nombre de requêtes en peu de temps)

Fiche technique du projet

Pour résumer, le client souhaite pouvoir vérifier périodiquement le statut d'une ou plusieurs de ses URLs.

Dans le MVP attendu, il souhaite avoir comme fonctionnalités principales :

- Avoir un champ « URL »
- Envoyer une requête pour checker le statut de l'URL fournie
- Afficher le code de statut de la réponse obtenue
- Garder une trace de l'URL en base de données pour réexécuter la requête périodiquement
- Reproduire la requête une fois par heure
- Pouvoir afficher l'historique complet des statuts des réponses de chaque URL

Il était éventuellement possible de développer plusieurs évolutions également si le MVP était atteint :

- Une authentification avec une création de compte et une connexion
- Un envoi de mail en cas d'erreur dans une réponse
- En plus du statut de la réponse, afficher et conserver le temps de réponse également
- Permettre à l'utilisateur de changer la fréquence des requêtes à une URL
- Restreindre certaines fonctionnalités aux utilisateurs premium
- Ajouter un système de paiement pour passer en premium
- Ajouter des tests de montée en charge

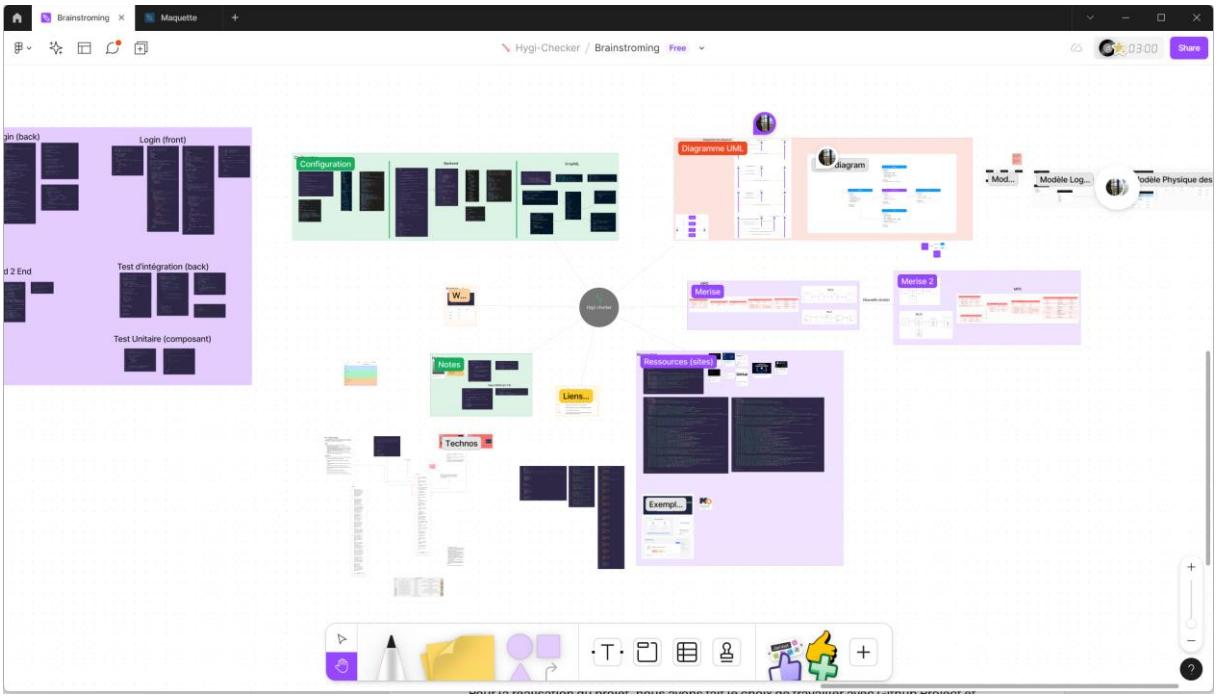
La totalité du MVP a été atteinte durant le développement de ce projet, ainsi que la très grande majorité des évolutions proposées.

Il ne resterait que l'ajout du système de paiement pour passer en premium, tout le fonctionnement étant déjà prévu, il ne reste que le système a proprement parlé à implémenter, ainsi que l'ajout des tests de montée de charge que nous n'avons pas eu le temps de mettre en place avant la fin.

4. La gestion de projet

4.1 Organisation de travail

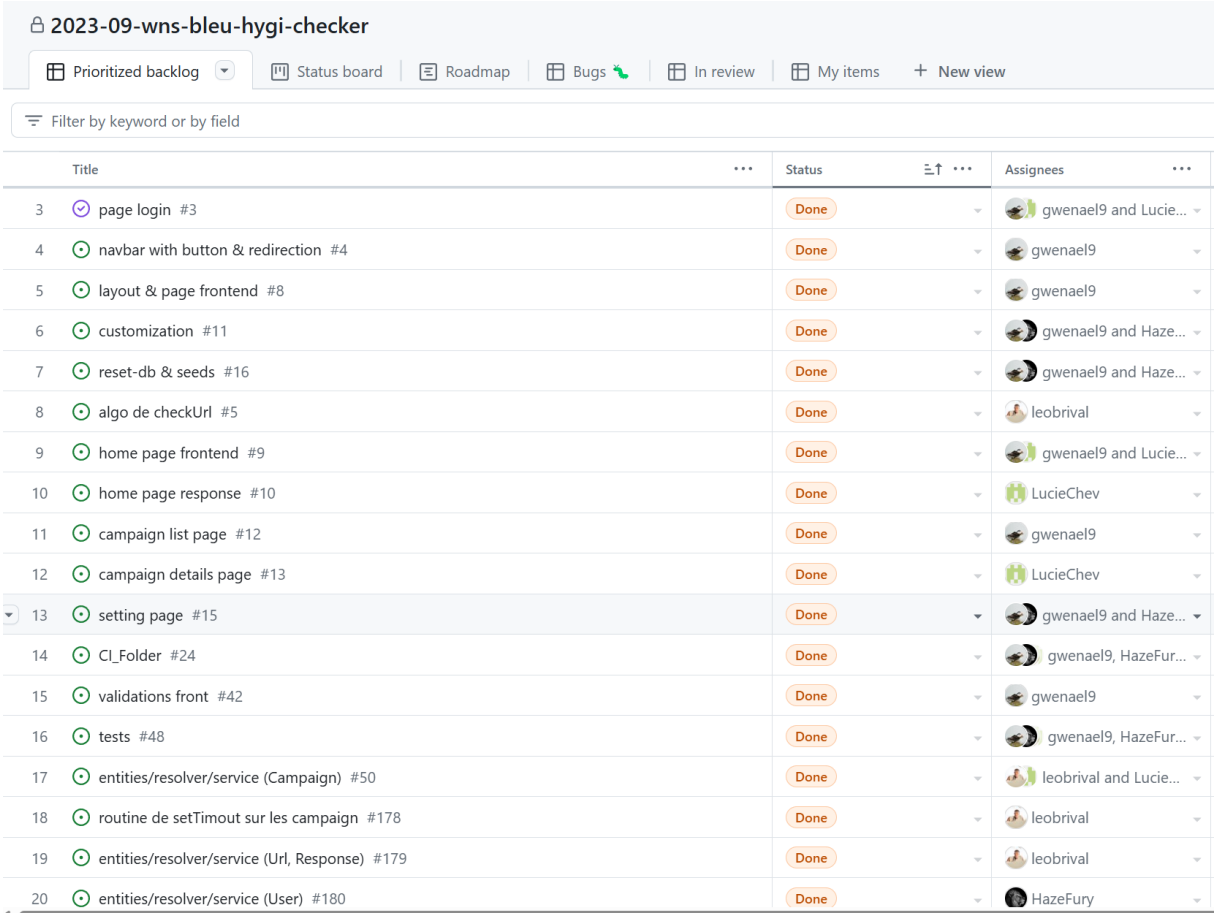
Nous avons le choix de mettre toutes nos réflexions et nos avancées sur un FigJam de Figma, pour garder une trace de nos ressources initiales ainsi que de nos différentes évolutions au cours du projet.



Partie de notre Brainstorming construit sur FigJam

Dès le début du projet, nous avons défini les grandes fonctionnalités. Puis, nous avons ensuite découpé ces points principaux en petites tâches.

Pour la réalisation du projet, nous avons fait le choix de travailler avec Github Project et de suivre notre développement comme cela.



2023-09-wns-bleu-hygi-checker

Prioritized backlog | Status board | Roadmap | Bugs | In review | My items | New view

Filter by keyword or by field

	Title	Status	Assignees
3	page login #3	Done	gwenael9 and Lucie...
4	navbar with button & redirection #4	Done	gwenael9
5	layout & page frontend #8	Done	gwenael9
6	customization #11	Done	gwenael9 and Haze...
7	reset-db & seeds #16	Done	gwenael9 and Haze...
8	algo de checkUrl #5	Done	leobrival
9	home page frontend #9	Done	gwenael9 and Lucie...
10	home page response #10	Done	LucieChev
11	campaign list page #12	Done	gwenael9
12	campaign details page #13	Done	LucieChev
13	setting page #15	Done	gwenael9 and Haze...
14	CI_Folder #24	Done	gwenael9, HazeFur...
15	validations front #42	Done	gwenael9
16	tests #48	Done	gwenael9, HazeFur...
17	entities/resolver/service (Campaign) #50	Done	leobrival and Lucie...
18	routine de setTimout sur les campaign #178	Done	leobrival
19	entities/resolver/service (Url, Response) #179	Done	leobrival
20	entities/resolver/service (User) #180	Done	HazeFury

Github Project, reprenant les différentes User Stories

Pendant les semaines d'école dédiées au projet, nous organisions une réunion de début de semaine. Ainsi, nous commencions par rappeler les tâches sur lesquelles nous étions en cours de développement, ou si ce n'était pas le cas, se répartir les tâches suivant la priorisation des fonctionnalités préétablies.

Chaque matin, nous commencions par un daily, ce qui permettait à chacun de communiquer le travail effectué et les points bloquants le cas échéant.

A la fin de la semaine, nous organisions une review pour présenter chacune de nos fonctionnalités à l'équipe et actualiser le backlog. Tous les mois, nous présentions notre travail au groupe de formation.

4.2 Workflow Github

Pour versionner notre code, nous avons utilisé Git en local et Github. Pour chaque tâche nous avons créé une branche Feature depuis la dernière version de Develop. Et nous avons ajouté au fur et à mesure les changements effectuées en local sur notre branche en remote. Quand la tâche était terminée nous demandions une Pull Request à un collègue pour que notre branche Feature soit mergée sur Dev.

A chaque fin de sprint, nous testions le travail effectué sur Develop en présentant nos fonctionnalités terminées à l'équipe.

A la fin de la review et avec la validation de l'équipe, nous mergions la branche Develop sur la branche Staging, puis une fois les tests effectués, nous pouvions merger également sur la branche Master.

Les types de branches :

- Feature :

Une branche Feature correspond à une fonctionnalité, elle est créée à partir de Dev et quand la fonctionnalité est terminée elle est mergée sur Dev.

Lors d'un commit sur une branche Feature, cela déclenche un pré-commit Husky :

```
#!/bin/sh
. "$(dirname "$0")/_/husky.sh"

fileList=$(git diff --staged --name-only :^package.json :^.husky/pre-commit :^frontend :^backend :^e2e-tests :^.github :^.gitignore)

echo ' '
echo '------------  🔑  START THE PRE COMMIT CHECK PROCESS  🔍   -----------'
echo ' '
echo ' ➡️  Checking staged files.'


if [ ! -z "$fileList" ]
then
  echo "❌ error: Unauthorized modifications in staged files ❌"

  echo "$fileList"

  echo "use 'git restore --staged ...' to fix the problem"

  exit 1
fi

echo ' ✅ Staged files checked successfully ! ✅'
echo ' '
echo ' '
echo ' ➡️  Start linting checks'
echo " [TIPS] if you have Prettier or ESlint issues, use 'npm run prettier-fix-all' or 'npm run eslint-fix-all' in the root folder to lint your code"
echo ' '

npm run lint

echo ' '
echo ' ✅ No error found, your code is well linted. Good job !!! ✅'
echo ' '

echo ' ➡️  Start unit tests'

cd ./frontend && npm run test

echo ' '
echo ' ✅ All tests passed, your code works fine !!! ✅'
echo ' '
echo ' 🚀🔥 Congrats !! Your code is clean, you can push it on Github 🔥🚀'
echo ' '
echo '------------  🔑 PRE COMMIT CHECK PROCESS FINISHED 🔍   -----------'
echo ' '

```

Script de pré-commit, déclenché à chaque commit

- Dev :

Un push sur cette branche déclenche les tests unitaires et les tests d'intégration.

- Staging :

Un push sur cette branche déclenche cette Action Github :

```yaml
name: Test, compile and push client and server to staging

on:
  push:
    branches: ["staging"]
  workflow_dispatch:

jobs:
  client-tests:
    uses: WildCodeSchool/2023-09-wns-bleu-hygi-checker/.github/workflows/tests-front.yml@main

  e2e-tests:
    uses: WildCodeSchool/2023-09-wns-bleu-hygi-checker/.github/workflows/e2e-tests.yml@main

  integration-tests:
    uses: WildCodeSchool/2023-09-wns-bleu-hygi-checker/.github/workflows/integration-tests.yml@main

  build-and-push-server:
    needs:
      - integration-tests
      - e2e-tests
    uses: WildCodeSchool/2023-09-wns-bleu-hygi-checker/.github/workflows/build-backend.yml@main
    secrets:
      DOCKERHUB_USERNAME: ${{ secrets.DOCKERHUB_USERNAME }}
      DOCKERHUB_TOKEN: ${{ secrets.DOCKERHUB_TOKEN }}
    with:
      image_name: hygichecker/hygi-checker-back

  build-and-push-client:
    needs:
      - client-tests
      - e2e-tests
    uses: WildCodeSchool/2023-09-wns-bleu-hygi-checker/.github/workflows/build-frontend.yml@main
    secrets:
      DOCKERHUB_USERNAME: ${{ secrets.DOCKERHUB_USERNAME }}
      DOCKERHUB_TOKEN: ${{ secrets.DOCKERHUB_TOKEN }}
    with:
      image_name: hygichecker/hygi-checker-front

  notify-vps:
    needs:
      - build-and-push-client
      - build-and-push-server
    runs-on: ubuntu-latest
    steps:
      - name: Call VPS webhook to update staging
        uses: wei/curl@master
        with:
          args: https://ops.0923-bleu-2.wns.wilders.dev/hooks/update-staging-hygi-checker"

```

Workflow GitHub Actions qui est utilisé pour tester, compiler et déployer sur Staging

- Master :

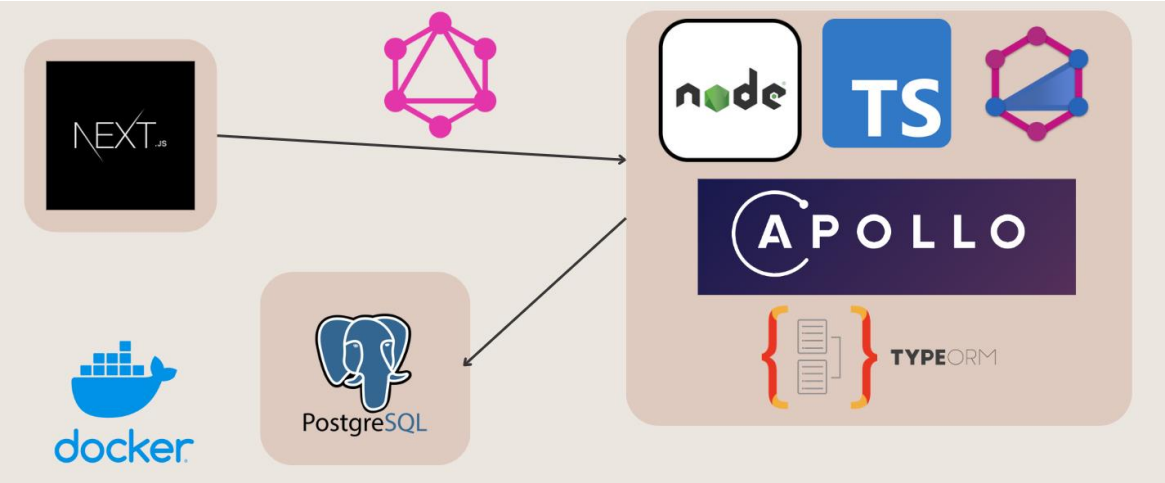
Un push sur cette branche déclenche cette Action Github :

```yaml
name: Test, compile and push client and server to production

on:
  push:
    branches: ["main"]

jobs:
  notify-vps:
    runs-on: ubuntu-latest
    steps:
      - name: Call VPS webhook to update production
        uses: wei/curl@master
        with:
          args: https://ops.0923-bleu-2.wns.wilders.dev/hooks/update-production-hygi-checker"

```

Workflow GitHub Actions qui est utilisé pour tester, compiler et déployer sur Master

5. L’environnement technique

5.1 Choix de l’architecture logicielle du projet



Architecture du projet

5.1.1 Frontend

Next.js : Framework React pour le rendu côté serveur (SSR) et le rendu statique.

Cela permet d'améliorer les performances et le SEO de votre application, car les pages peuvent être pré-rendues avant d'être servies à l'utilisateur.

Le système de routage basé sur le système de fichiers est simple et intuitif. Il vous suffit de créer des fichiers dans le dossier pages, et Next.js se charge du reste. Pas besoin de configurer manuellement les routes.

Next.js supporte nativement TypeScript, ce qui permet une transition fluide.

5.1.2 Backend

- **Node.js** : Environnement d'exécution JavaScript côté serveur.

Node.js permet d'utiliser le même langage, JavaScript, à la fois pour le développement frontend et backend. Cela simplifie le développement en réduisant le besoin d'apprendre un nouveau langage pour le côté serveur, et facilite la réutilisation du code entre le frontend et le backend.

Node.js exécute le code JavaScript très rapidement, ce qui contribue à de meilleures performances globales.

Node.js supporte TypeScript, permettant aux développeurs qui préfèrent le typage statique de l'utiliser tout en bénéficiant de l'écosystème riche de Node.js.

- **TypeScript** : Un sur-ensemble de JavaScript qui ajoute des types statiques, rendant le code plus robuste et maintenable.

TypeScript permet de définir des types pour les variables, les fonctions, les objets, etc. Cela permet de détecter les erreurs potentielles à la compilation plutôt qu'à l'exécution, réduisant ainsi les bugs dans le code. Le typage statique améliore aussi la lisibilité et la maintenabilité du code, car il rend les intentions du développeur plus explicites.

Grâce aux types, les éditeurs de code comme Visual Studio Code peuvent fournir une autocomplétion plus intelligente, des vérifications de type en temps réel, et des refactorisations plus sûres. Cela améliore l'expérience de développement en rendant le processus plus fluide et en aidant à prévenir les erreurs courantes.

Le système de types agit comme une forme de documentation intégrée. En spécifiant les types de données attendus, vous facilitez la compréhension du code par d'autres développeurs (ou par vous-même à l'avenir).

TypeScript aide à prévenir des erreurs courantes telles que les fautes de frappe dans les noms de variables, les erreurs d'assignation de types incorrects, ou l'utilisation de méthodes sur des objets inexistants. Cela contribue à la robustesse du code.

- **Apollo** : Une plateforme GraphQL utilisée pour construire un serveur GraphQL, facilitant les requêtes de données depuis le frontend.

Apollo Client est une bibliothèque puissante pour gérer les requêtes GraphQL et l'état global de l'application côté client. Il simplifie la gestion de l'état en centralisant les données de votre application dans un store unique, et offre des fonctionnalités comme la mise en cache, le suivi des mutations, et la gestion des erreurs.

Apollo Client gère automatiquement la mise en cache des résultats des requêtes GraphQL, ce qui améliore considérablement les performances en évitant les requêtes réseau redondantes. Les données sont synchronisées entre le cache et l'interface utilisateur, ce qui garantit une expérience utilisateur fluide.

Apollo prend en charge des fonctionnalités avancées de GraphQL comme la pagination, et les mutations. Cela permet de tirer pleinement parti de GraphQL dans des applications complexes.

Apollo dispose d'un écosystème complet avec des outils comme Apollo Server (pour créer des serveurs GraphQL) et Apollo Studio (pour la surveillance et l'optimisation des

APIs GraphQL). Cet écosystème permet de couvrir tous les aspects du développement avec GraphQL, du backend au frontend.

Apollo est conçu pour être modulaire. On peut choisir d'utiliser uniquement les parties dont on a besoin (par exemple, utiliser Apollo Client sans Apollo Server) et l'intégrer progressivement dans notre projet.

- **GraphQL** : Un langage de requête pour les APIs qui permet de demander uniquement les données nécessaires, améliorant l'efficacité des requêtes par rapport à REST.

Avec GraphQL, les clients peuvent spécifier exactement quelles données ils veulent dans une seule requête, ce qui évite de surcharger le réseau avec des données non nécessaires (over-fetching) ou de faire plusieurs appels pour récupérer toutes les données nécessaires (under-fetching). Cela réduit les temps de réponse et améliore les performances.

Contrairement à REST, où chaque ressource a généralement sa propre URL (endpoint), une API GraphQL n'a qu'un seul endpoint pour toutes les requêtes. Cela simplifie la gestion des routes et permet de centraliser toutes les opérations en un seul endroit.

GraphQL possède une introspection intégrée qui permet aux développeurs d'explorer l'API, de voir quelles données sont disponibles, et comment elles sont liées. Des outils comme Apollo Studio permettent d'explorer visuellement l'API et de tester des requêtes en temps réel, ce qui améliore l'expérience de développement.

GraphQL gère naturellement les relations entre les différentes entités, permettant de récupérer des données imbriquées en une seule requête.

- **TypeORM** : Un ORM (Object-Relational Mapping) pour TypeScript et JavaScript (ES7, ES6, ES5). Il permet de travailler avec des bases de données relationnelles (comme PostgreSQL) en utilisant des objets TypeScript.

TypeORM est conçu pour être utilisé avec TypeScript. Il utilise pleinement les types, ce qui permet de tirer parti de la sécurité du typage statique pour réduire les erreurs liées à la manipulation des données.

TypeORM supporte un large éventail de bases de données.

TypeORM gère de manière native les relations entre les entités. Cela permet de modéliser facilement des structures de données complexes tout en maintenant une forte cohérence et intégrité des données.

TypeORM est conçu pour être simple et intuitif. Les concepts de base tels que la création d'entités, la gestion des relations, et l'exécution de requêtes sont faciles à comprendre, ce qui réduit la courbe d'apprentissage pour les nouveaux utilisateurs.

En automatisant de nombreuses tâches liées à la gestion des bases de données (comme les requêtes SQL, la gestion des transactions, et les migrations), TypeORM permet aux développeurs de se concentrer davantage sur la logique métier plutôt que sur les détails d'implémentation.

- **TypeGraphQL** enrichit l'architecture décrite en fournissant un moyen efficace de créer des APIs GraphQL en TypeScript, avec une intégration fluide avec TypeORM et Apollo Server.

Il simplifie la gestion des schémas et assure une forte cohérence des types dans l'ensemble de l'application.

Au lieu de définir le schéma GraphQL manuellement en utilisant des chaînes de caractères, TypeGraphQL permet de définir les schémas en utilisant des classes et des décorateurs TypeScript. Cela rend le code plus propre, maintenable et orienté objet, tout en réduisant le besoin de duplicata entre les types TypeScript et les types GraphQL.

Les décorateurs TypeGraphQL (@ObjectType, @Field, @Resolver, etc.) permettent de déclarer facilement des types, des champs, des résolveurs, et des mutations. Cela améliore la lisibilité du code et simplifie la création d'APIs complexes.

Grâce au typage fort de TypeScript, TypeGraphQL permet de s'assurer que les types des données entre le client et le serveur sont cohérents. Cela améliore la sécurité des types et réduit les risques de bugs liés aux incohérences entre les types GraphQL et les types TypeScript.

5.1.3 Base de Données

PostgreSQL est un système de gestion de base de données relationnelle (SGBDR) open-source, puissant et extensible. Il stocke les données de l'application et est accessible via TypeORM dans le backend.

PostgreSQL est reconnu pour sa robustesse et sa fiabilité, offrant des fonctionnalités avancées telles que les transactions ACID (Atomicité, Cohérence, Isolation, Durabilité), qui garantissent l'intégrité des données même en cas de panne. Il dispose également de mécanismes puissants pour la récupération après sinistre.

PostgreSQL offre une multitude de fonctionnalités avancées telles que le support des types de données complexes : Il prend en charge les types de données personnalisés, les tableaux, les JSON, les UUID, et plus encore.

PostgreSQL offre des fonctionnalités de sécurité robustes, y compris le contrôle d'accès basé sur les rôles, le chiffrement SSL, l'audit des connexions, et la gestion fine des privilèges. Cela permet de sécuriser les données sensibles et de se conformer aux réglementations.

PostgreSQL est disponible sur toutes les principales plateformes, y compris Linux, Windows, et MacOs, ce qui le rend extrêmement versatile pour les déploiements sur différents systèmes.

5.1.4 Conteneurisation

Docker est une plateforme de conteneurisation qui permet d'empaqueter une application et ses dépendances dans un conteneur, assurant que l'application fonctionne de manière identique dans différents environnements (développement, test, production).

Les conteneurs Docker sont extrêmement portables. Une fois une application emballée dans un conteneur, elle peut être exécutée n'importe où Docker est installé, que ce soit sur un ordinateur local, un serveur, ou dans le cloud. Cela simplifie grandement le déploiement et la gestion des applications dans des environnements multiples.

Docker permet d'inclure toutes les dépendances nécessaires à une application (bibliothèques, runtime, outils, etc.) directement dans le conteneur. Cela garantit que l'application dispose toujours de l'environnement exact dont elle a besoin pour fonctionner, éliminant ainsi le problème des "ça marche sur ma machine".

Docker est un outil clé dans les pipelines CI/CD (intégration continue/déploiement continu). Il permet de construire, tester et déployer des applications rapidement et de manière cohérente, ce qui accélère le cycle de développement et améliore la qualité du code en facilitant l'intégration continue.

De plus, tous ces choix techniques s’accompagnent de communautés très actives qui contribue à leur développement continu, ainsi qu’un riche écosystème de plugins et d'extensions qui ajoutent des fonctionnalités supplémentaires.

5.1.5 Outils supplémentaires

- **Eslint** est un linter, un outil qui va analyser le code et vérifier qu'un certain nombre de règles préalablement fixées sont respectées. Grâce à cet outil, on peut donc coder toujours de la même manière, car on respecte le style de code défini. Sur Hygi-checker, le style de code suivi est celui d'Airbnb.

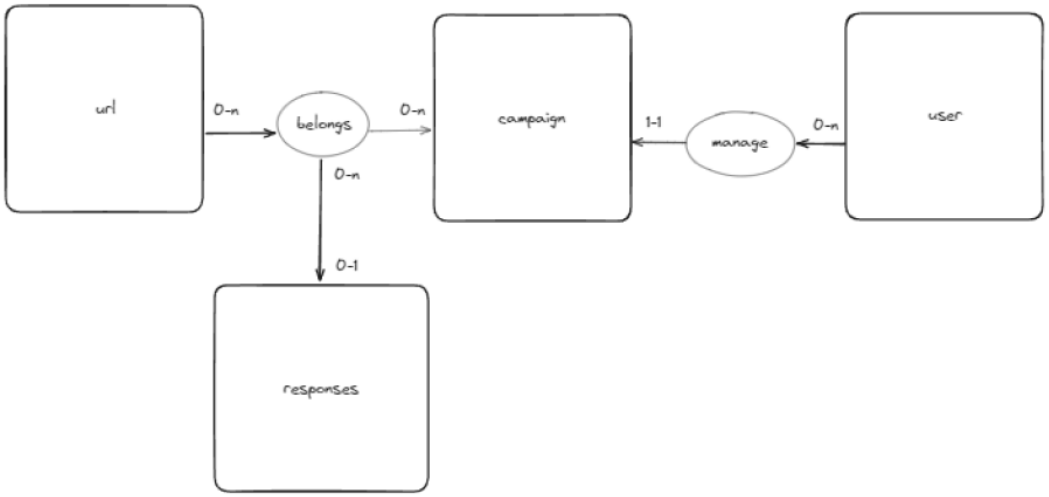
- **Prettier** est un code formater qui permet donc de formatter le code avec des règles définies.

- **ShadCN UI** est un ensemble de composants UI (interface utilisateur) open-source pour React, construit sur la bibliothèque Radix UI et souvent utilisé avec des frameworks comme Next.js. Les composants sont conçus pour être accessibles, flexibles et faciles à intégrer dans des projets React.

- **Tailwind CSS** est un framework CSS utilitaire qui permet aux développeurs de concevoir des interfaces utilisateur rapidement en utilisant des classes utilitaires directement dans le HTML. Contrairement aux frameworks CSS traditionnels comme Bootstrap, qui fournissent des composants préfabriqués, Tailwind se concentre sur l'approche utilitaire, où chaque classe fait une seule chose.

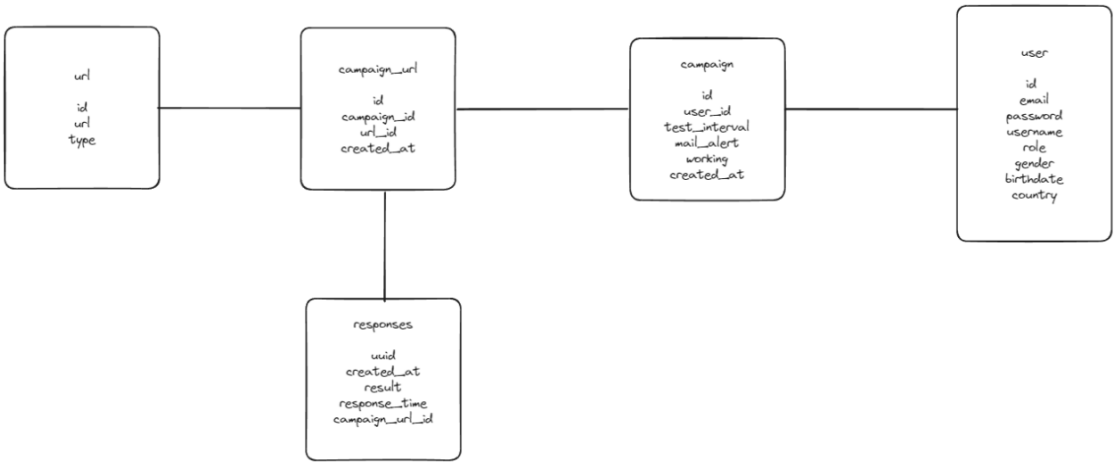
5.2 Modélisation des Données (Merise)

5.2.1 Modèle Conceptuel de Données (MCD)



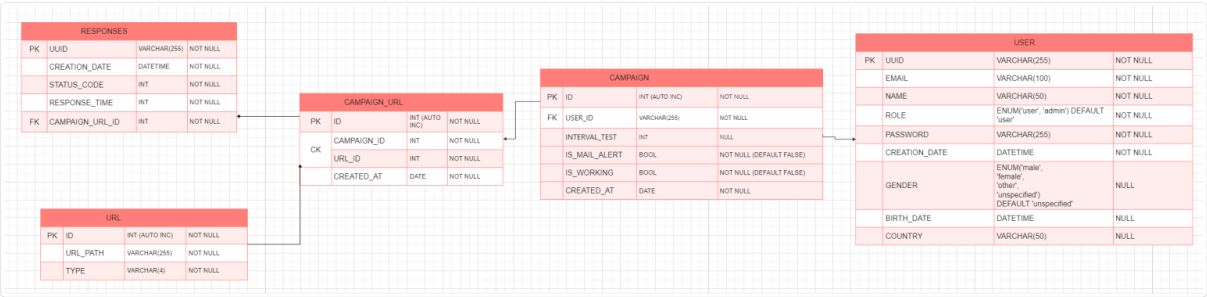
MCD du projet

5.2.2 Modèle Logique de Données (MLD)



MLD du projet

5.2.3 Modèle Physique de Données (MPD)



MPD du projet

1. **Table USER** :
    - **Champs** :
        - UUID : Identifiant unique de l'utilisateur (Primary Key).
        - EMAIL, NAME, PASSWORD, CREATION_DATE, ...
        - GENDER, BIRTH_DATE, COUNTRY : Informations personnelles supplémentaires.
    - **Relations** :
        - La table USER est reliée à la table CAMPAIGN via USER_ID.
2. **Table CAMPAIGN** :
    - **Champs** :
        - ID : Identifiant unique de la campagne (Primary Key).
        - USER_ID : Référence vers un utilisateur (Foreign Key vers USER).
        - INTERVAL_TEST, IS_MAIL_ALERT, IS_WORKING, ...
    - **Relations** :
        - Liée à USER par USER_ID.
        - Liée à CAMPAIGN_URL par ID.

3. **Table CAMPAIGN_URL** :
    - **Champs** :
        - ID : Identifiant unique (Primary Key).
        - CAMPAIGN_ID : Référence vers une campagne (Foreign Key vers CAMPAIGN).
        - URL_ID : Référence vers une URL (Foreign Key vers URL).
        - CREATED_AT : Date de création.
    - **Relations** :
        - Liée à CAMPAIGN et URL.
4. **Table URL** :
    - **Champs** :
        - ID : Identifiant unique de l'URL (Primary Key).
        - URL_PATH, TYPE.
    - **Relations** :
        - Liée à CAMPAIGN_URL via URL_ID.
5. **Table RESPONSES** :
    - **Champs** :
        - UUID : Identifiant unique de la réponse (Primary Key).
        - CREATION_DATE, STATUS_CODE, RESPONSE_TIME.
        - CAMPAIGN_URL_ID : Référence vers CAMPAIGN_URL (Foreign Key).
    - **Relations** :
        - Liée à CAMPAIGN_URL par CAMPAIGN_URL_ID.

Toutes les relations sont bien établies avec des clés étrangères, ce qui garantit l'intégrité des données.

5.3 Modélisation Orientée Objet (UML)

5.3.1 Diagrammes de Séquence

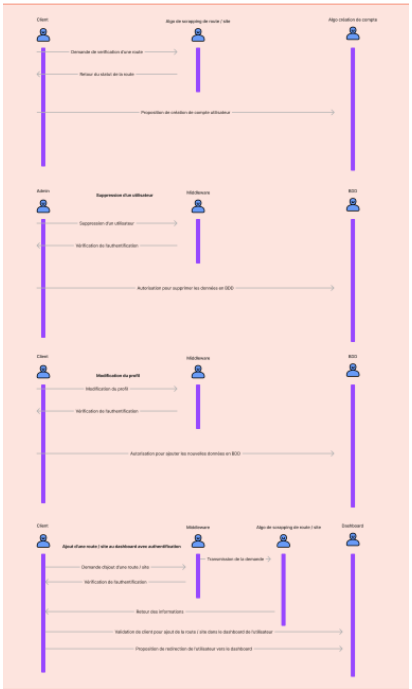


Diagramme de séquence du projet

1. **Gestion du compte utilisateur**

- **Acteurs** : Client, Middleware, Base de Données
- **Étapes** :
    1. Le client demande la vérification d'une URL.
    2. Le Middleware retourne l'URL si elle existe.
    3. Le Middleware propose la création d'un compte utilisateur si l'URL n'existe pas.

2. **Suppression d'un utilisateur**

- **Acteurs** : Admin, Middleware, BDD
- **Étapes** :
    1. L'admin demande la suppression d'un utilisateur.
    2. Le Middleware vérifie la demande de suppression.
    3. Le Middleware autorise la suppression en envoyant la demande à la BDD.

3. **Modification du profil utilisateur**

- **Acteurs** : Client, Middleware, BDD
- **Étapes** :
    1. Le client modifie son profil.
    2. Le Middleware vérifie la modification.
    3. Le Middleware autorise la modification en mettant à jour les données dans la BDD.

4. **Ajout d'une nouvelle URL ou modification d'une URL existante**

- **Acteurs** : Client, Middleware, Middleware Admin, Surveillant
- **Étapes** :
    1. Le client ajoute ou modifie une URL.
    2. Le Middleware valide l'URL et envoie les informations à l'Admin Middleware.
    3. Le Middleware Admin propose de valider ou de vérifier l'URL via un Surveillant.
    4. Le Surveillant valide ou demande une vérification supplémentaire de l'URL.

Acteurs impliqués :

- **Client** : Utilisateur qui interagit avec le système.
- **Admin** : Administrateur du système qui gère les utilisateurs.
- **Middleware** : Composant intermédiaire qui traite les demandes.
- **BDD (Base de Données)** : Stocke les données utilisateurs, URL, etc.
- **Surveillant** : Entité ou système responsable de la validation finale des URL ajoutées ou modifiées.

Ce diagramme illustre comment les actions des utilisateurs sont traitées par le système via un middleware, en interaction avec une base de données et d'autres acteurs pour effectuer des opérations comme la vérification des URL, la gestion des comptes utilisateurs, et l'ajout ou la modification d'URL.

5.3.2 Diagramme de Classes

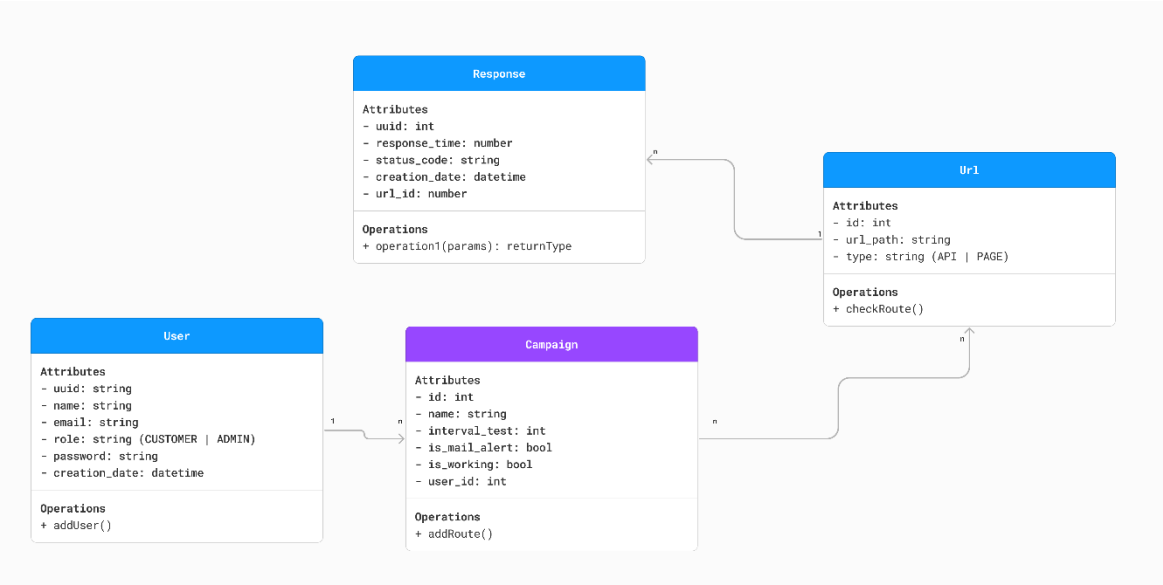


Diagramme de classes du projet

1. Classe User :

- **Attributs** :
  - Uuid, name, email, password, creation_date
- **Opération** :
  - addUser()

2. Classe Campaign :

- **Attributs** :
  - Id, name, interval_test, is_mail_alert, is_working, user_id
- **Opération** :
  - addRoute()
- **Relation** :
  - Une relation de type *One to Many* avec la classe User. Un utilisateur peut avoir plusieurs campagnes.

3. **Classe Url :**
    - **Attributs** :
        - Id, url_path, type
    - **Opération** :
        - checkRoute()
    - **Relation** :
        - Une relation de type *One to Many* avec la classe Campaign. Une campagne peut avoir plusieurs URLs associées.
4. **Classe Response :**
    - **Attributs** :
        - Uuid, response_time, status_code, creation_date, url_id
    - **Opération** :
        - Operation(params) : returnType
    - **Relation** :
        - Une relation de type *One to Many* avec la classe Url. Un URL peut avoir plusieurs réponses associées.

Les flèches indiquent des relations de type *One to Many*, où le "1" est du côté de la classe avec l'attribut simple, et le "n" du côté de la classe avec l'attribut de clé étrangère.

5.4 Contraintes du projet

5.4.1 Contraintes Techniques

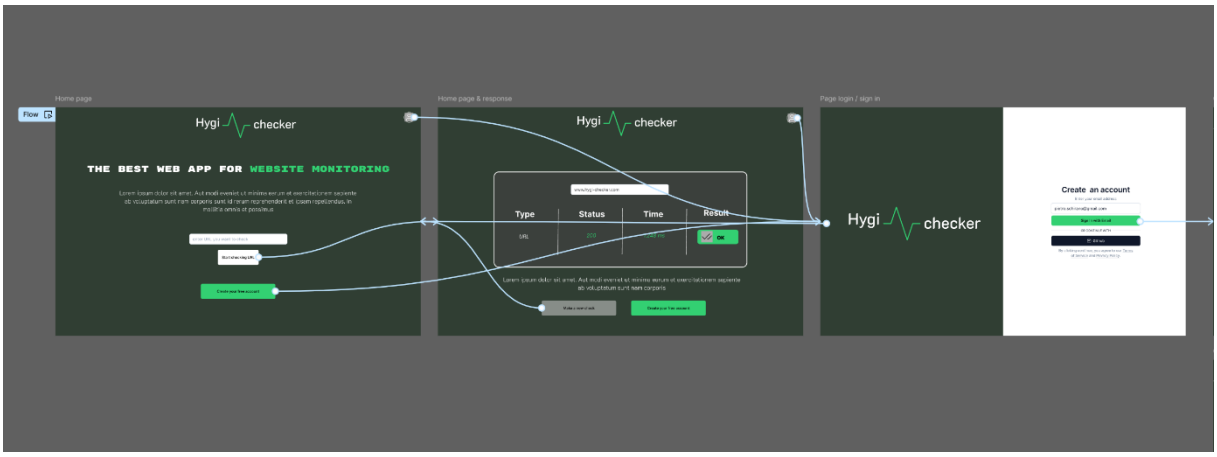
- **Compatibilité Navigateur** : Le projet doit être compatible avec les navigateurs modernes.
- **Sécurité** : Le projet doit respecter les meilleures pratiques de sécurité, notamment la protection contre les attaques XSS, CSRF, et SQL injection. L'utilisation de HTTPS est obligatoire.
- **Accessibilité** : Le site doit être conforme aux normes d'accessibilité WCAG.

5.4.2 Contraintes Fonctionnelles

- **Interactivité** : L'interface utilisateur doit être intuitive et réactive.
- **Responsivité** : Le projet doit être entièrement responsive, s'adaptant à une gamme d'écrans allant des téléphones mobiles aux écrans de bureau larges.
- **Intégration** : L'application doit pouvoir s'intégrer avec des API tierces, notamment pour la gestion des paiements.

5.5 Maquettes et enchaînement des maquettes

Nous avons réalisé une maquette sur Figma, je l'ai prototypé pour permettre de bien comprendre l'enchainement des différentes maquettes et afin de préparer les différentes étapes du développement, et de bien appréhender les interactions entre chaque pages et composants de l'application.



Maquettes du projet

- **Page d'accueil** : Cette page est l'écran de bienvenue où l'utilisateur peut entrer une URL pour vérifier son statut.
- **Page des Résultats** : Après la vérification de l'URL, l'utilisateur est redirigé vers une page affichant les résultats, y compris le code de statut HTTP et d'autres informations pertinentes.
- **Page de Connexion / Inscription** : Cette page permet aux utilisateurs de créer un compte ou de se connecter pour accéder à des fonctionnalités supplémentaires.

Maquettes du projet

- **Liste des Campagnes** : Cette page liste plusieurs campagnes de surveillance, avec des options pour gérer chaque campagne individuellement.
- **Détails d'une Campagne** : Une page détaillant les résultats d'une campagne spécifique, y compris des graphiques et des actions possibles (activer/désactiver, supprimer, etc.).

5.6 Sécurité de l'Application

5.6.1 Utilisation des JWT (JSON Web Tokens)

```
expressMiddleware(server, {
  context: async ({ req, res }) => {
    let user: User | null = null;

    const cookies = new Cookies(req, res);
    const token = cookies.get("token");

    if (token) {
      try {
        const verify = await jwtVerify<Payload>(
          token,
          new TextEncoder().encode(process.env.JWT_PRIVATE_KEY)
        );

        user = await new UserService().findUserByEmail(
          verify.payload.email
        );
      } catch (err) {
        console.error("il s'est produit une erreur dans l'index");
        console.error(err);
        const cookies = new Cookies(req, res);
        cookies.set("token");
        //potentiellement gérer l'erreur, est ce que l'erreur est liée au fait
      }
    }
    return { req, res, user, campaignManagerService };
  },
})
);      You, 4 weeks ago • code Leo
```

Extrait de code concernant le JWT

- Le code utilise des JWT pour authentifier les utilisateurs. Un cookie (token) contenant le JWT est vérifié à chaque requête via la fonction jwtVerify.
- Sécurité des JWT :
    - Le JWT est signé avec une clé privée (process.env.JWT_PRIVATE_KEY), ce qui garantit que le token n'a pas été altéré. Seuls les détenteurs de la clé privée peuvent générer des tokens valides.
    - Le token est déchiffré pour extraire les informations sur l'utilisateur (comme l'email) qui sont ensuite utilisées pour retrouver l'utilisateur dans la base de données.

5.6.2 Expiration et Renouvellement des Tokens :

- Il y a également une gestion de l'expiration des tokens JWT et un renouvellement des tokens expirés.
- Gestion des erreurs JWT : Le code inclut une gestion basique des erreurs liées aux tokens (par exemple, en supprimant le cookie contenant un token invalide ou expiré).

5.6.3 Protection des Routes avec CORS (Cross-Origin Resource Sharing)

```
await server.start();
app.use(
  "/",
  cors<cors.CorsRequest>({
    origin: [
      "http://localhost:3000",
      "https://studio.apollographql.com",
      "https://0923-bleu-2.wns.wilders.dev/",
      "https://staging.0923-bleu-2.wns.wilders.dev/",
    ],
    credentials: true,
  }),
```

Extrait de code concernant les CORS

- Le middleware CORS est utilisé pour restreindre les origines autorisées à accéder à l'API. Seules certaines origines spécifiques (comme http://localhost:3000, https://studio.apollographql.com, et les environnements de staging/production) sont autorisées à envoyer des requêtes.
- Credentials : true : Cette option permet de gérer les cookies d'authentification entre le client et le serveur, ce qui est essentiel pour les sessions sécurisées avec JWT.

5.6.4 Protection contre les Attaques XSS, CSRF, et SQL Injection

```
            case 'secure':
                cookie.secure = true;
                break;

            case 'httponly':
                cookie.httponly = true;
                break;
```

Extrait de code concernant les cookies

- **Httponly** : Lorsqu'un cookie est marqué comme httponly, il est inaccessible via le JavaScript côté client, ce qui aide à prévenir les attaques de type **Cross-Site Scripting (XSS)**.
- **Secure** : Les cookies marqués comme secure ne seront envoyés que sur des connexions sécurisées HTTPS. Cela protège contre les attaques de type **Man-in-the-Middle (MITM)** où les cookies pourraient être interceptés sur une connexion non sécurisée.

```
                case 'expires':
                    value = new Date(value);
                    // ignore date if can not parse it
                    if (value.toString() !== 'Invalid Date') {
                        cookie.expires = value;
                    }
                    break;
            case 'max-age':
                cookie.expires = new Date(Date.now() + (Number(value) || 0) * 1000);
                break;
```

Extrait de code concernant les cookies

- **Suppression des Cookies Expirés** : Les cookies expirés sont activement supprimés de la liste des cookies gérés, ce qui est une bonne pratique pour éviter l'accumulation de cookies obsolètes ou potentiellement non sécurisés.
- **Gestion de l'Expiration des Cookies** : Le code gère l'expiration des cookies, soit par un paramètre max-age, soit par une date expirée. En cas d'absence de ces paramètres, le cookie est défini avec une expiration par défaut basée sur la valeur de SESSION_TIMEOUT, assurant que les cookies ne restent pas actifs indéfiniment, ce qui est crucial pour la sécurité.

```
// if no expire date, then use sessionTimeout value
if (!cookie.expires) {
    cookie.expires = new Date(Date.now() + (Number(this.options.sessionTimeout || SESSION_TIMEOUT) || SESSION_TIMEOUT) * 1000);
}
```

Extrait de code concernant les cookies

- **Gestion des Cookies de Session** : Le code gère les cookies de session qui expirent après une certaine période d'inactivité (SESSION_TIMEOUT), contribuant ainsi à la sécurité en réduisant la fenêtre de temps pendant laquelle un cookie pourrait être utilisé après que l'utilisateur a quitté la session.

- **Protection contre les injections SQL** : Le code ne montre pas directement les interactions SQL, mais en utilisant TypeORM (via datasource), on bénéficie de la protection contre les injections SQL, car les ORM utilisent des requêtes préparées qui empêchent l'injection de code malveillant dans les requêtes SQL.

5.6.5 Bonnes Pratiques Générales

- **Gestion des Erreurs** :

Le code intègre une gestion des erreurs, notamment pour les erreurs de JWT. Cette gestion des erreurs, incluant des messages d'erreur spécifiques, renforce la sécurité.

- **Utilisation de https** :

Pour des raisons de sécurité, il est crucial de s'assurer que toutes les communications se font via HTTPS, surtout lors de la transmission de tokens JWT ou d'autres informations sensibles.

- **Mise à Jour des Packages** :

Il est important de garder les dépendances de son projet à jour, en particulier celles liées à la sécurité (comme express, jose, etc.), pour bénéficier des derniers correctifs de sécurité.

6. Exemple d’une fonctionnalité du projet

6.1 Introduction

Pour rappel, le client souhaite vérifier périodiquement le statut d’une ou plusieurs de ses URLs. L’objectif est de permettre aux utilisateurs de surveiller la disponibilité et les performances de leurs sites web ou services en ligne. Afin de répondre à ce besoin, une fonctionnalité clé du projet est la gestion des campagnes d'URL. Une campagne permet à l'utilisateur de regrouper et de suivre un ensemble d'URLs de manière centralisée, en automatisant les vérifications de statut à intervalles réguliers.

6.2 Conception de la Fonctionnalité

La fonctionnalité de gestion des campagnes permet aux utilisateurs de créer, modifier, et supprimer des campagnes. Chaque campagne contient une ou plusieurs URLs, que l'utilisateur souhaite surveiller. Les fonctionnalités principales incluent :

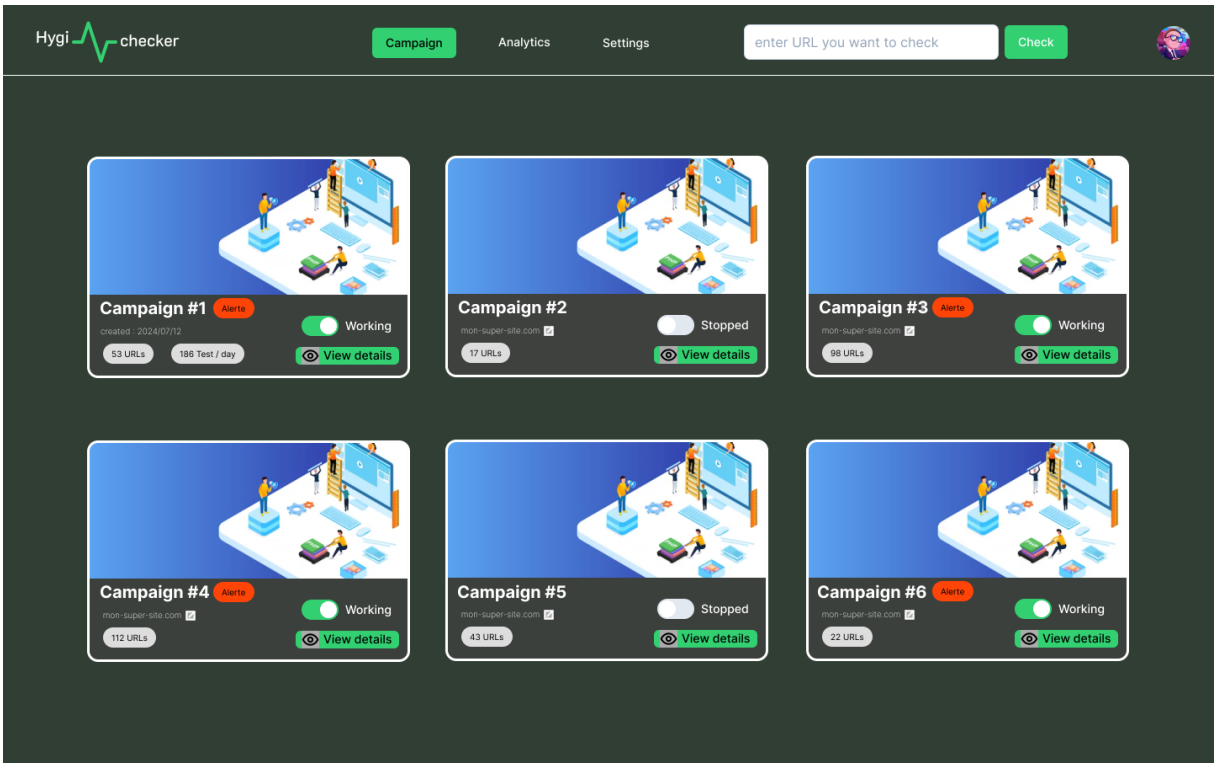
- **Création de campagnes** : L'utilisateur peut créer une nouvelle campagne en donnant un nom et en ajoutant les URLs qu'il souhaite suivre.
- **Gestion des URLs** : Les utilisateurs peuvent ajouter ou supprimer des URLs au sein d'une campagne. Chaque URL peut être configurée avec un intervalle de vérification spécifique.
- **Visualisation des résultats** : L'utilisateur peut visualiser les résultats des vérifications pour chaque URL dans une campagne, avec des informations telles que le code de statut HTTP, le temps de réponse, et l'historique des statuts.

Cette fonctionnalité est essentielle pour offrir une vue d'ensemble de la santé et de la disponibilité des ressources web surveillées par l'utilisateur. Elle permet d'assurer que les utilisateurs sont informés rapidement de tout problème avec leurs URLs.

6.3 Maquettage et prototypage

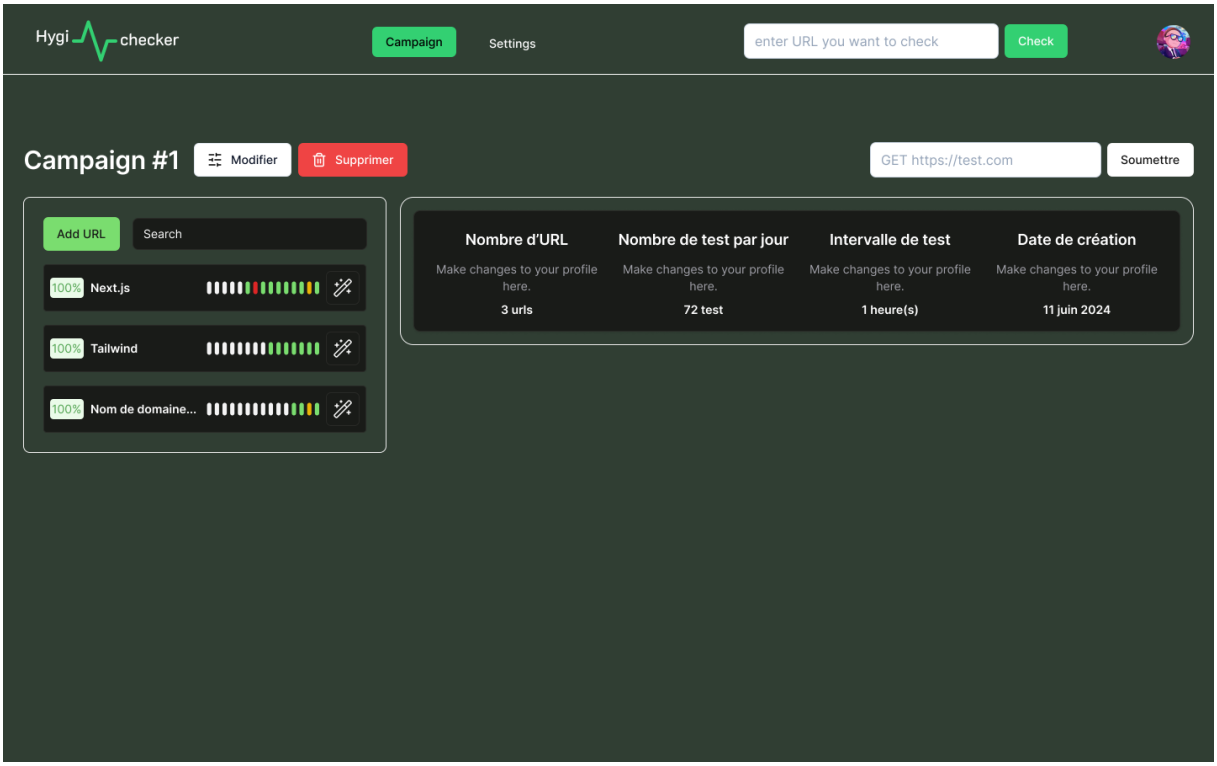
Avant de commencer le développement, nous avons réalisé des maquettes pour visualiser la conception de la fonctionnalité et recueillir des retours d'utilisateurs potentiels. Les maquettes ont été réalisées avec **Figma** et comprennent les écrans principaux suivants :

- **Liste des campagnes** : Cet écran affiche toutes les campagnes créées par l'utilisateur, avec des options pour ajouter une nouvelle campagne, ou modifier/supprimer une campagne existante.



Maquette de la liste des campagnes d'un utilisateur

- **Détails d'une campagne** : Cet écran montre les détails d'une campagne spécifique, y compris la liste des URLs surveillées, les résultats récents des vérifications, et les options pour ajouter ou supprimer des URLs.



Maquette des détails d'une campagne d'un utilisateur

Ces maquettes ont servi de guide pour le développement, en assurant que l'interface utilisateur serait intuitive et répondrait aux besoins des utilisateurs finaux.

6.4 Développement Front-End

Le front-end de la fonctionnalité a été développé en utilisant **Next.js** pour bénéficier des capacités de rendu côté serveur et de génération de pages statiques. Voici les étapes clés du développement front-end :

- **Définition de la mutation GraphQL** : Pour permettre la création et la gestion des campagnes, des mutations GraphQL ont été définies. Par exemple, la mutation CreateCampaign permet de créer une nouvelle campagne avec un nom et une liste d'URLs.

```
import { gql } from "@apollo/client";

export const CREATE_CAMPAIGN = gql`
  mutation CreateCampaign($input: InputCreateCampaign!) {
    createCampaign(input: $input) {
      id
      name
      image
      intervalTest
      isMailAlert
      isWorking
    }
  }
`;
```

Définition de la mutation GraphQL

- **Exécution de la mutation pour créer une campagne** : Lors de l'interaction utilisateur, par exemple lorsqu'il soumet un formulaire pour créer une campagne, cette mutation est appelée via Apollo Client. L'interface utilisateur est mise à jour pour refléter les changements.

```
});
export function AddCampaignForm() {
  const { toast } = useToast();
  const router = useRouter();

  const [fakeLoading, setFakeLoading] = useState(false);
  const [openForm, setOpenForm] = useState(false);

  const { refetch } = useCampaignsByUserIdQuery();

  const [createCampaign] = useCreateCampaignMutation({
    onCompleted: (data) => {
      const campaignId = data.createCampaign.id;
      setTimeout(() => {
        setFakeLoading(false);
        setOpenForm(false);
        router.push(`/dashboard/campaign/details/${campaignId}`);
        toast({
          title: `Campaign created successfully`,
          variant: "success",
        });
        refetch();      Marc-Antoine B, 2 months ago • Added the crea
      }, 1000);
    },
    onError: () => {
      toast({
        title: `Something went wrong. Please try again`,
        variant: "destructive",
      });
    },
  });
```

Exécution de la mutation pour créer une campagne

Ce code assure une expérience utilisateur fluide, où les utilisateurs peuvent facilement ajouter de nouvelles campagnes et voir instantanément les résultats de leurs actions.

6.5 Développement Back-End

Le back-end de la fonctionnalité repose sur **Node.js** et **TypeGraphQL** pour gérer les opérations sur les données. Voici les principales étapes du développement back-end :

- **Définition du type GraphQL** : Le type Campaign est défini pour décrire la structure des données renvoyées par les requêtes et mutations liées aux campagnes. Il inclut les informations sur la campagne.

```
export type CreateCampaignMutation = {
  __typename?: "Mutation";
  createCampaign: {
    __typename?: "Campaign";
    id: number;
    name: string;
    image?: string | null;
    intervalTest?: number | null;
    isMailAlert?: boolean | null;
    isWorking?: boolean | null;
  };
};
```

Définition du type : décrit la structure des données renvoyées par la mutation CreateCampaign

- **Implémentation de la Mutation GraphQL** : La mutation CreateCampaign est implémentée dans le back-end pour traiter les requêtes de création de campagne. Le service CampaignService est utilisé pour interagir avec la base de données et créer une nouvelle campagne.

```
@Mutation(() => Campaign)
async createCampaign(
  @Ctx() ctx: MyContext,
  @Arg("input") input: InputCreateCampaign
): Promise<Campaign | null | undefined> {
  if (ctx.user) {
    const user = await new UserService().findUserByEmail(ctx.user.email);
    if (!user) {
      throw new Error("Error, please try again");
    }
    return await this.campaignService.createCampaign(input, user);
  } else {
    throw new Error("You must be authenticated to perform this action");
  }
}
```

Implémentation de la Mutation GraphQL : traite cette mutation en vérifiant l'authentification de l'utilisateur et en utilisant un service pour créer la campagne dans la base de données.

Ce code vérifie d'abord que l'utilisateur est authentifié, puis appelle le service pour créer la campagne et l'enregistrer dans la base de données.

7. La présentation du plan de tests

7.1 Tests unitaires

Le test unitaire est une méthode de test qui consiste à tester des éléments ou des unités d'un logiciel. L'objectif du test unitaire est de valider que chaque unité du logiciel fonctionne comme prévu.

Les tests unitaires sont effectués pendant le développement d'une application par les développeurs. Les tests unitaires consistent à isoler une section du code et à en vérifier la bonne réalisation. Une unité peut être une simple fonction, une méthode, une procédure, un module ou un objet.

```
// campaignCard.test.tsx.snap

import CampaignCard from "@/components/CampaignCard";
import "@testing-library/jest-dom";
import { render, screen } from "@testing-library/react";

describe("CampaignCard component", () => {
  it("renders correctly", () => {
    const view = render(
      <CampaignCard
        campaign={{
          title: "teeeest",
          url: "urllll",
          urlNumber: 4,
        }}
      />,
    );
    expect(screen.getByText(/teeeest/)).toBeInTheDocument();
    expect(screen.getByText(/4/)).toBeInTheDocument();
    expect(view.baseElement).toMatchSnapshot();
  });
});

```

Test unitaire pour un composant nommé CampaignCard

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`CampaignCard component renders correctly 1`] = `
<body>
  <div>
    <div
      class="[w-400] flex flex-column p-2 gap-2 rounded-lg border"
    >
      <h2>
        teeeest
      </h2>
      <div
        class="flex flex-column gap-1"
      >
        <p>
          urllll
        </p>
        <p>
          4
        </p>
      </div>
    </div>
  </div>
</body>
`;

```

Capture d'écran d'un fichier de snapshot Jest

7.2 Tests d'intégration

Le test d'intégration désigne le processus consistant à tester les interfaces entre deux composants ou modules logiciels afin d'évaluer comment les données sont transférées entre eux.

Les stratégies de test d'intégration permettent aux équipes de développement et aux spécialistes informatiques de détecter les défauts susceptibles d'être introduits lors de l'intégration de deux ou plusieurs modules logiciels, ainsi que d'évaluer l'adéquation et la fonction globales des éléments logiciels combinés.

Les tests d'intégration interviennent généralement après les tests unitaires, qui consistent à tester les modules et les unités individuels. Une fois qu'il a été déterminé que chaque unité fonctionne de manière isolée, les tests d'intégration évaluent comment toutes les unités fonctionnent lorsqu'elles sont combinées.

Les tests d'intégration sont un processus incrémentiel, qui nécessite généralement que les testeurs intègrent les modules un par un et effectuent des tests à chaque étape.

```
// testResolver.test.fr

import { execute } from "../jest.setup";
import Test from "../src/entities/test.entity";
import getAdminContext from "./helpers/getAdminContext";
import addTest from "./operations/addTest";
import getTest from "./operations/getTest";

describe("TestRevsolver", () => {
  it("Placeholder Test", async () => {
    await Test.create({
      text: "1) texte du premier test d'intégration",
    }).save();
    await Test.create({
      text: "2) texte du deuxième test d'intégration",
    }).save();
    expect(await execute(getTest)).toMatchSnapshot(`
      {
        "data": {
          "tests": [
            {
              "id": "b84ba25f-edbf-4248-b229-c5a704e813cc",
              "text": "1) texte du premier test d'intégration",
            },
            {
              "id": "1ebd2d9a-a0c8-49d2-9bd3-0b691ff051e0",
              "text": "2) texte du deuxième test d'intégration",
            },
          ],
        },
      }
    `);
  });

  it("can create a test", async () => {
    const res = await execute(
      addTest,
      { data: { name: "test" } },
      await getAdminContext()
    );
    expect(res).toMatchSnapshot();
  });
});

```

Test d'intégration pour TestResolver

```
// jest.setup.ts

import { ASTNode, graphql, GraphQLSchema, print } from "graphql";

import db from "./src/lib/datasource";
import getSchema from "./src/schema";
import { Maybe } from "type-graphql";

async function clearDB() {
  const entities = db.entityMetadatas;
  const tableNames = entities
    .map((entity) => `"${entity.tableName}"`)
    .join(", ");
  await db.query(`TRUNCATE ${tableNames} RESTART IDENTITY CASCADE;`);
}

export let schema: GraphQLSchema;

export async function execute(
  operation: ASTNode,
  variableValues?: Maybe<{
    readonly [variable: string]: unknown;
  }>,
  contextValue = {}
) {
  return await graphql({
    schema,
    source: print(operation),
    variableValues,
    contextValue,
  });
}

beforeAll(async () => {
  await db.initialize();
  schema = await getSchema;
});

beforeEach(async () => {
  await clearDB();
});

afterAll(async () => {
  await db.destroy();
});

```

Fichier de configuration de tests pour un environnement de test GraphQL

7.3 Tests End to End

Les tests End to End (de bout en bout) sont une technique utilisée pour vérifier si une application (web, mobile…) se comporte comme prévu, du début à la fin.

Ils consistent à vérifier que l'utilisateur final puisse achever les principaux scénarios d'utilisation de l'application.

En effectuant des tests de bout en bout, vous pouvez vérifier un grand nombre de fonctionnalités en peu de temps. Grâce à ces tests e2e on peut s'assurer que tous les modules se connectent correctement entre eux.

Également, nous ne serions pas en mesure d'identifier les régressions potentielles à la suite de modifications du code concernant les parties que nous n'avons pas testées. Il est donc impératif d'effectuer ces tests de bout en bout à la fin de la phase de recette, une fois que tous les autres tests ont été effectués. Cela implique qu'il a d'autres types de tests a effectué qui couvrent finalement la grande majorité de l'application.

Le test E2E fait figure de dernière étape qui validera, ou non, le lancement définitif d'un site web ou d'une application.

8. La présentation d'un jeu d'essai élaboré par le candidat

Je fais le choix de vous présenter aujourd'hui, le test end-to-end que j'ai mis en place sur le projet, sur la fonctionnalité du login.

```
You, last month | 1 author (You)
1  import { connect, disconnect } from "./helpers";
2  import { clearDB } from "../../backend/src/lib/datasource";
3  import { test } from "@playwright/test";
4  import UserService from "../../backend/src/services/user.service";
5
6  test.beforeAll(connect);
7  test.beforeEach(clearDB);
8  test.afterAll(disconnect);
9
10 const email = "jane.doe@example.com";
11 const password = "Hygichecker69!";
12 const username = "Jane Doe";
13 const accepted_terms = true;
14
15 async function createUser() {
16   const userService = new UserService();
17   await userService.createUser({ email, password, username, accepted_terms });
18 }
19
20 test("can connect with correct credentials", async ({ page }) => {
21   await createUser();
22
23   await page.goto("http://localhost:3000/auth/login");
24   await page.getByTestId("login-email").fill(email);
25   await page.getByTestId("login-password").fill(password);
26   await page.getByRole("button", { name: "Log in" }).click();
27   await page.waitForURL("**/dashboard/campaign/lists");
28 });
29
```

Test end-to-end du login du projet

- **Imports** :
    - Connect, disconnect et clearDB sont importés à partir de modules personnalisés (voir pages suivantes).
    - Test est importé de @playwright/test, qui est un framework de test pour le navigateur.
    - UserService est importé d'un service utilisateur dans le backend.

- **Configuration des tests** :
    - Avant tous les tests (beforeAll), la connexion à la base de données est établie.
    - Avant chaque test (beforeEach), la base de données est effacée.
    - Après tous les tests (afterAll), la connexion à la base de données est fermée.
- **Variables** :
    - Des constantes sont définies pour email, password, username, et accepted_terms qui sont utilisées pour créer un utilisateur.
- **Fonction createUser** :
    - Cette fonction utilise un service utilisateur (UserService) pour créer un nouvel utilisateur avec les informations définies dans les constantes.
- **Test de connexion** :
    - Un test est défini pour vérifier si un utilisateur peut se connecter avec les bonnes informations d'identification.
    - Le test fait appel à createUser pour créer un utilisateur.
    - Ensuite, le script utilise Playwright pour automatiser la navigation sur une page de connexion, remplir les champs email et mot de passe, cliquer sur le bouton de connexion, et vérifier si l'utilisateur est redirigé vers une URL spécifique après s'être connecté avec succès.

Utilité : Ce type de script est utile pour tester l'authentification utilisateur dans une application web. Il permet de s'assurer que les utilisateurs peuvent se connecter correctement avec des identifiants valides.

Connexion et déconnexion à la base de données

- **Importation du module db :**
  - Le module db est importé depuis un fichier local. Ce module représente la configuration de la base de données.
- **Fonction connect :**
  - La fonction connect est une fonction asynchrone qui appelle db.initialize() pour établir une connexion avec la base de données.
  - await db.initialize() attend que l'initialisation de la connexion soit terminée avant de poursuivre. Cela signifie que cette fonction assure que la connexion à la base de données est prête avant que le reste du code ne s'exécute.
- **Fonction disconnect :**
  - La fonction disconnect est également asynchrone et appelle db.destroy() pour fermer la connexion à la base de données.
  - await db.destroy() attend que la connexion soit correctement fermée avant de continuer. Cela garantit que toutes les ressources sont correctement libérées avant de passer à l'étape suivante.

```typescript
import * as dotenv from "dotenv";
import { DataSource } from "typeorm";

import User from "../entities/user.entity";
import Test from "../entities/test.entity";
import Url from "../entities/url.entity";
import Response from "../entities/response.entity";
import Campaign from "../entities/campaign.entity";
import CampaignUrl from "../entities/campaignUrl.entity";
import AccessCheckResolver from "../resolvers/accessCheck.resolver";
import Premium from "../entities/premium.entity";

dotenv.config();

const db = new DataSource({
  type: "postgres",
  host: process.env.DB_HOST || "db" || "testDB",
  port: parseInt(process.env.DB_PORT || "0") || 5432,
  username: process.env.DB_USER || "postgres",
  password: process.env.DB_PASS || "postgres",
  database: process.env.DB_NAME || "postgres",
  entities: [
    User,
    Test,
    Url,
    Response,
    Campaign,
    CampaignUrl,
    AccessCheckResolver,
    Premium,
  ],
  synchronize: true,
});

export async function clearDB() {      You, 6 months ago • tests e2e & rou
  const entities = db.entityMetadatas;
  const tableNames = entities
    .map((entity) => `"${entity.tableName}"`)
    .join(", ");
  await db.query(`TRUNCATE ${tableNames} RESTART IDENTITY CASCADE;`);
}

export default db;
```

Fichier de configuration pour la base de données

Imports

- **Dotenv** : Le module dotenv est importé pour charger les variables d'environnement depuis un fichier .env. Ces variables sont ensuite utilisées pour configurer la connexion à la base de données.
- **DataSource** : DataSource est importé de typeorm. C'est la classe principale utilisée pour configurer et initialiser la connexion à la base de données avec TypeORM.
- **Entities et Resolver** : Plusieurs entités (User, Test, Url, Response, etc.) et un resolver (AccessCheckResolver) sont importés. Ces entités représentent les tables dans la base de données et sont utilisées par TypeORM pour interagir avec celles-ci.

Configuration de dotenv

- Dotenv.config() : Cette ligne charge les variables d'environnement définies dans un fichier .env pour les rendre disponibles tout au long de l'application.

Initialisation de la Source de Données (db)

- Une nouvelle instance de DataSource est créée avec plusieurs paramètres :
    - **type** : Spécifie le type de base de données utilisé, ici "postgres".
    - **host** : Le nom d'hôte ou l'adresse IP du serveur de la base de données. Il est défini par process.env.DB_HOST ou par défaut à "db" ou "testDB".
    - **port** : Le port utilisé par PostgreSQL, défini par process.env.DB_PORT, avec une valeur par défaut de 5432.
    - **username** : Le nom d'utilisateur pour se connecter à la base de données, défini par process.env.DB_USER avec "postgres" par défaut.
    - **password** : Le mot de passe pour se connecter à la base de données, défini par process.env.DB_PASS avec "postgres" par défaut.
    - **database** : Le nom de la base de données, défini par process.env.DB_NAME avec "postgres" par défaut.
    - **entities** : Un tableau des entités importées qui représentent les différentes tables dans la base de données.
    - **synchronize** : Mis à true, cela indique à TypeORM de synchroniser automatiquement le schéma de la base de données avec les entités définies.

Fonction clearDB

- **clearDB** : Cette fonction asynchrone est utilisée pour vider toutes les tables de la base de données. Elle est utilisée dans les tests pour s'assurer que chaque test commence avec une base de données vide.
    - db.entityMetadatas : Récupère les métadonnées des entités, qui incluent les noms des tables.
    - **tableNames** : Les noms des tables sont extraits des métadonnées, et formattés pour être utilisés dans une requête SQL.
    - **TRUNCATE** : La commande SQL TRUNCATE est utilisée pour vider toutes les tables spécifiées (${tableNames}) et réinitialiser les identifiants (RESTART IDENTITY CASCADE), ce qui supprime toutes les lignes dans les tables et remet les compteurs d'identifiants à zéro.

Exportation

- **export default db;** : Le db (l'instance de DataSource) est exporté par défaut, ce qui permet de l'importer et de l'utiliser ailleurs dans l'application pour gérer les opérations liées à la base de données.

9. Intégration continu (CI) et Déploiement continu (CD)

L'application web de notre projet a été hébergé sur un VPS fournit par la Wild Code School. Nous avons tout mis en place en équipe, de l'initialisation du VPS (gestion des utilisateurs, mot de passe, port de connexion), la mise à jour, à l'installation et le déploiement de notre projet. Nous avons également mis en place un déploiement continu.

Afin d'automatiser le déploiement et réduire les manipulations de commandes nous avons mis en place un déploiement continu en préproduction et en production. Pour cela nous avons utilisé différents outils, notamment les Actions Github, Docker, DockerHub, Caddy en reverse proxy, Webhook sur le serveur.

Github Actions

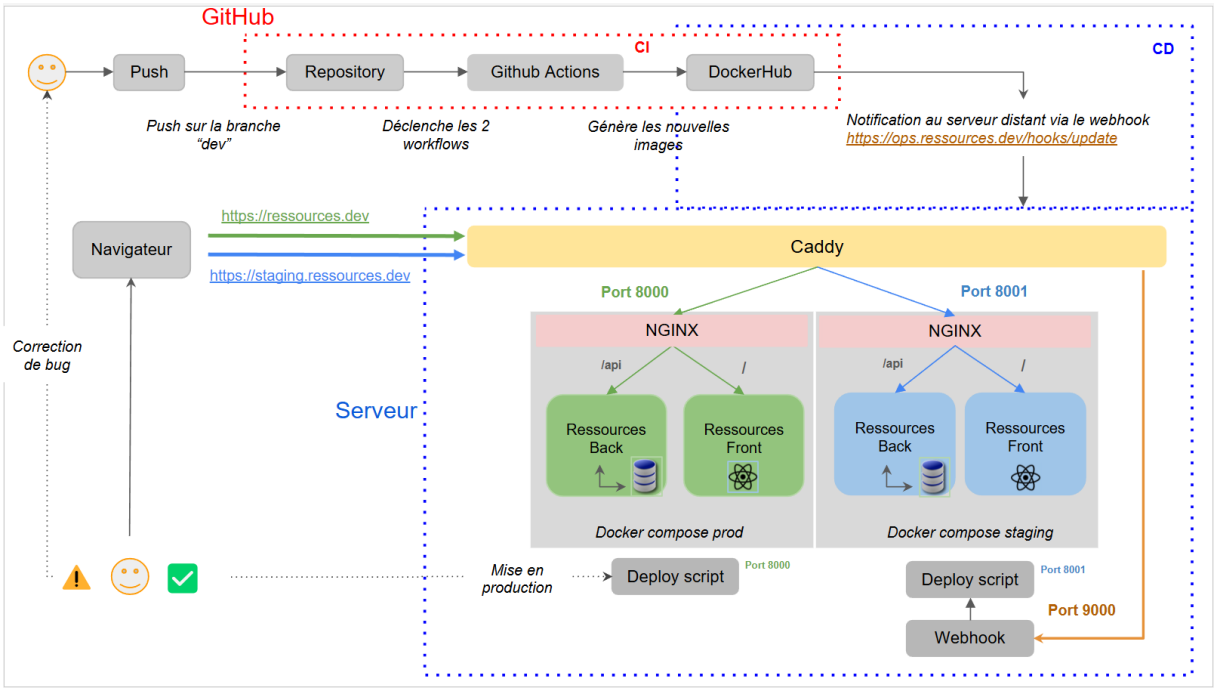
Lancé en 2019, cette solution de CI/CD est inclus dans les repositories et permet d'exécuter des workflows en se basant sur des événements, comme un push sur une branche. Le déploiement d'un pipeline Actions peut découler d'une configuration manuelle, c'est-à-dire l'édition de deux fichiers YAML associés à un dépôt ou le choix d'une recommandation effectuée par GitHub. Dans notre cas, nous avons utilisé cette solution pour automatiser le lancement des tests et pour builder et stocker les images dans DockerHub.

DockerHub

Comparable au Github pour Docker, ce service est un référentiel qui permet de créer, tester, stocker et déployer des images de conteneurs Docker. Aussi, il est possible d'utiliser DockerHub pour utiliser des images officielles de nombreux logiciels open source. C'est le cas notamment de l'image postgre que nous avons utilisé sur notre projet.

Webhook

Outil léger et configurable, Webhook permet de créer facilement des endpoints HTTP (hooks) sur un serveur pour exécuter des commandes configurées. Il est également possible de transmettre des données de la requête HTTP (headers, variables d'environnement, ou les variables de requête) à ces commandes. Enfin, il permet aussi de spécifier des règles qui doivent être satisfaites pour que le hook soit déclenché.



Fonctionnement global du déploiement continu

Conclusion

Ce projet a représenté une étape cruciale dans mon parcours de reconversion professionnelle, en me permettant de mettre en pratique les compétences acquises au cours de ma formation à la Wild Code School. À travers la conception, le développement et le déploiement de cette application web, j'ai pu explorer une variété de technologies modernes, telles que Next.js, Node.js, TypeScript, GraphQL, et Docker.

Le développement de cette application m'a non seulement permis de renforcer mes compétences en développement web, mais aussi d'acquérir une meilleure compréhension des processus de gestion de projet, notamment à travers l'utilisation de GitHub pour le versioning, la gestion des branches, et les workflows CI/CD. Travailler en équipe sur ce projet a également mis en lumière l'importance de la collaboration et de la communication dans un environnement de développement, des aspects que j'ai pu expérimenter tout au long de ce projet.

L'intégration de tests unitaires, d'intégration et end-to-end a permis d'assurer la qualité du code et la fiabilité de l'application, garantissant ainsi un produit final robuste et fonctionnel. De plus, les différentes mesures de sécurité mises en place, notamment l'authentification via JWT, la gestion des cookies sécurisés, et la protection contre les attaques courantes, ont contribué à renforcer la sûreté de l'application.

En rétrospective, ce projet a été une expérience riche et formatrice, m'offrant l'opportunité de me confronter à des problématiques réelles et de trouver des solutions adaptées. Il marque la fin de ma formation initiale et le début de ma carrière en tant que développeuse web, prêt à relever de nouveaux défis dans le monde professionnel.

Je remercie chaleureusement tous ceux qui m'ont accompagné et soutenu dans ce parcours, et je suis fière des compétences que j'ai pu développer et des résultats que j'ai pu obtenir. Ce projet n'est qu'un début, et je suis impatiente de continuer à apprendre et à contribuer à des projets futurs avec la même passion et détermination.