EPITECH – MscPro 2020

CASHMANAGER - Documentation technique Back-End

Groupe 7

Lucas Boisbourdin, Alexis Barthelmebs, Bryan Lebar, Julien Barriere

13/12/2020

Table des matières

1. Résumé du document

Ce document est la documentation technique du Back-End de notre projet CashManager, réalisé dans le cadre du module DEV de 1ère année d'MscPro à Epitech.

Ce document est divisé en 4 parties :

- Présentation et documentation technique du Back-End de la solution
- Présentation et documentation technique Docker et Docker-Compose
- Présentation et documentation technique du processus CI/CD
- Présentation et documentation technique de l'hébergement du Back-End de la solution

Le repo de la solution : https://github.com/ReyLinn/Dev-CashManager

Le repo du Back-End : https://github.com/ReyLinn/Dev-CashManager-Server

2. Rappel sur le fonctionnement de la solution

2.1 Description de la solution

Ce projet a pour but de réaliser une application mobile permettant de scanner des articles dans un magasin, puis de payer en scannant le QR Code d'un chèque ou en scannant la puce NFC d'une carte bancaire.

Le projet portant plus sur le lien Front-End/Back-End, sur les tests, les commentaires de codes et la documentation, le cahier des charges du projet mentionnait que n'importe quelle méthode de paiement et d'authentification serait acceptée.

Le projet comporte donc deux grands pôles : le Back-End et le Front-End.

Ce document traitera de la partie Back-End.

3. Présentation globale et technique du Back-End

3.1 Technologie utilisée

Pour la partie Back-End de notre projet, nous avons décidé d'utiliser la technologie Microsoft ASP .Net Core permettant de construire aisément une application Web, dans notre cas une Api Json.

L'ASP .Net Core est le dernier framework .Net de Microsoft, codé en C#, il reprend ses prédécesseurs (l'ASP .Net et le .Net Framework) en plus puissant, plus modulable, et plus portatif car il peut être développé sur Windows, Mac OS et Linux.

Le projet nous incitait à nous diriger vers Java, étant donné que le Front-End est en Kotlin (framework Java), mais nous avons préféré le .Net Core car nous avions plus d'affinités avec le C# et la suite de développement Microsoft.

Nous utilisons le modèle MVC, mais sans le V, c'est-à-dire que nous avons supprimer les vues du projet, et nous utilisons uniquement le Controller, et les modèles.

Pour être plus précis, nos modèles ne sont utilisés que par l'ORM de l'application. L'ORM que nous utilisons est EntityFramework Core.

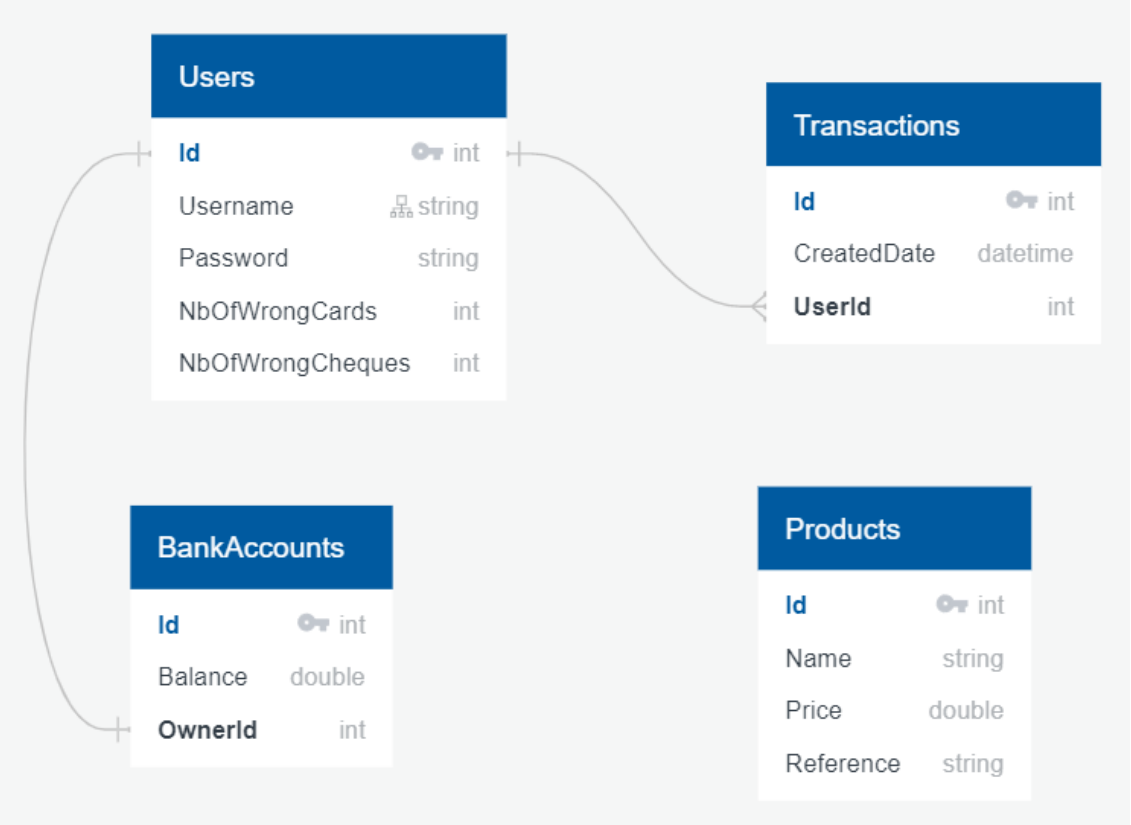
EntityFramework est développé par Microsoft, c'est un ORM très puissant et facile d'utilisation. Nous l'utilisons dans sa configuration par défaut : Code First. Cela signifie que nous créons nos modèles de données, puis nous générons des migrations et nous les appliquons à la base de données, ce qui créera les tables correspondantes.

3.2 Diagrammes UML

Vous pourrez retrouver les diagrammes UML dans le fichier UML.ppt, situé dans le dossier de rendu.

3.3 Schéma de notre base de données

Voici le schéma de notre base de données :



Nous avons donc une table Users qui servira de compte à nos utilisateurs. Ceux-ci auront nu nom d'utilisateur (Username) unique, un mot de passe (Password), un nombre de mauvaise carte de crédit (NbOfWrongCards) et un nombre de mauvais chèques (NbOfWrongCheques).
Chaque utilisateur (Users) à un compte bancaire (BankAccounts), c'est dont une relation One to One. Un compte bancaire à un solde (Balance) et un détenteur (OwnerId).

La table Transactions représente l'historique des transactions effectuées par un utilisateur. Cette table à comme propriété la data de création (CreatedDate), c'est-à-dire la date (et l'heure) à laquelle la transaction a été effectuée, cella nous permet de vérifier que l'utilisateur ne fait pas trop de requêtes de paiement à la minute.

Il y a une relation Many to One entre la table Transactions et la table Users.

Enfin, nous avons la table des produits (Products) qui contient le nom (Name) le prix (Price) et la référence (Reference) du produit.

Cette table est indépendante.

Les relations sont gérées par notre ORM, nous lui précisons quelles sont les relations dans le code, et ce que nous souhaitons récupérer lors des appels à la database.

Par exemple, dans la classe User, nous avons une propriété BankAccount, et une autre propriété List<Transactions>, qui seront fournies par l'ORM.

3.4 Test unitaires et tests d'intégration

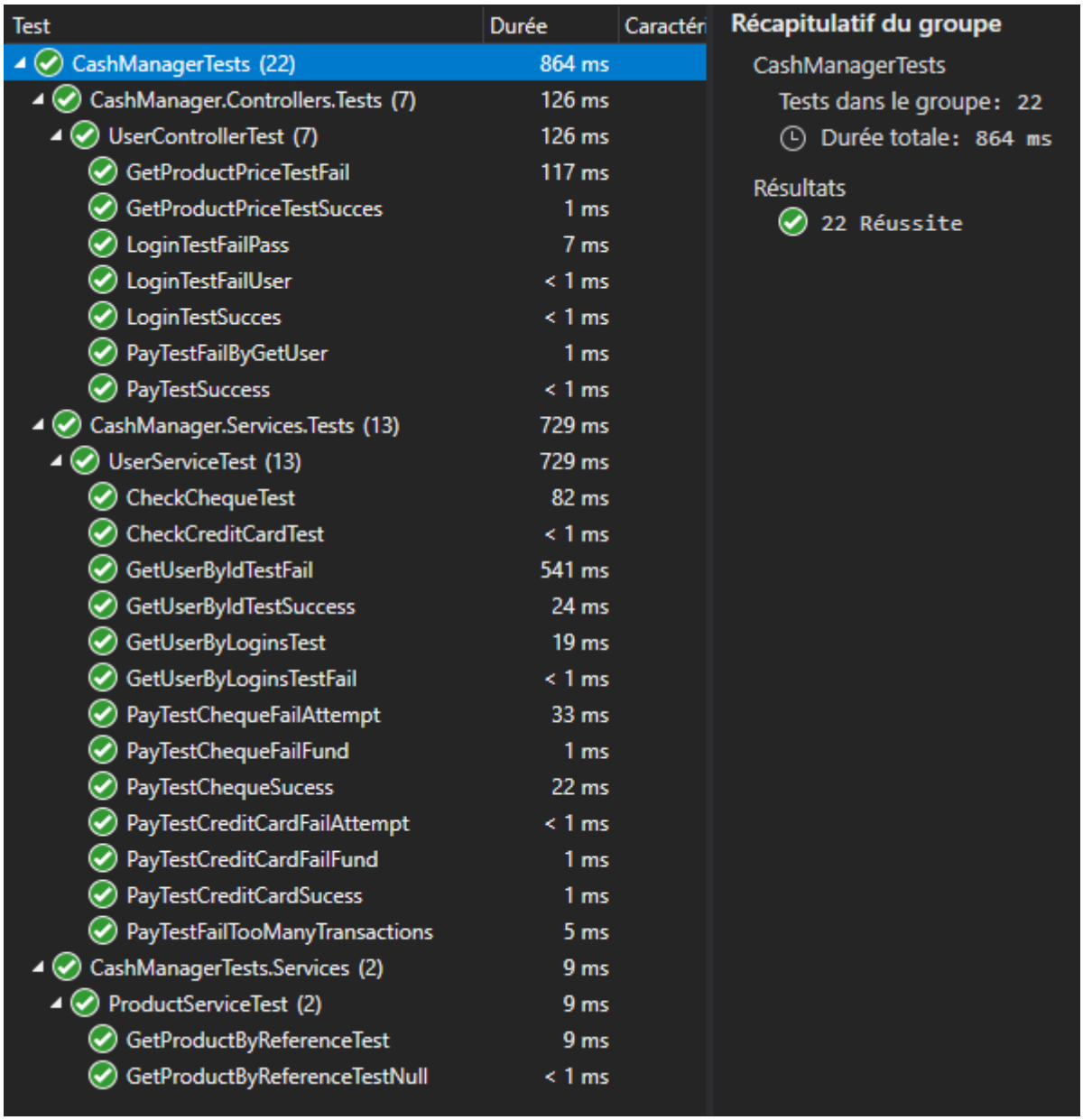
Comme demandé dans le cahier des charges nous avons implémenté des tests dans la solution.

Nous avons réalisé des tests d'intégration auprès de notre Controller, et des tests unitaires sur nos Services.

Pour effectuer ces tests nous avons utilisé XUnit.

Afin de ne pas avoir à créer une base de données propre aux tests nous utilisons un package pour notre ORM qui lui ajoute la fonctionnalité InMemory. Cela nous permet de simuler une basse de données selon notre modèle, dans la mémoire, et ainsi de ne pas impacter notre base de données existante.

Voici les résultats de nos tests :



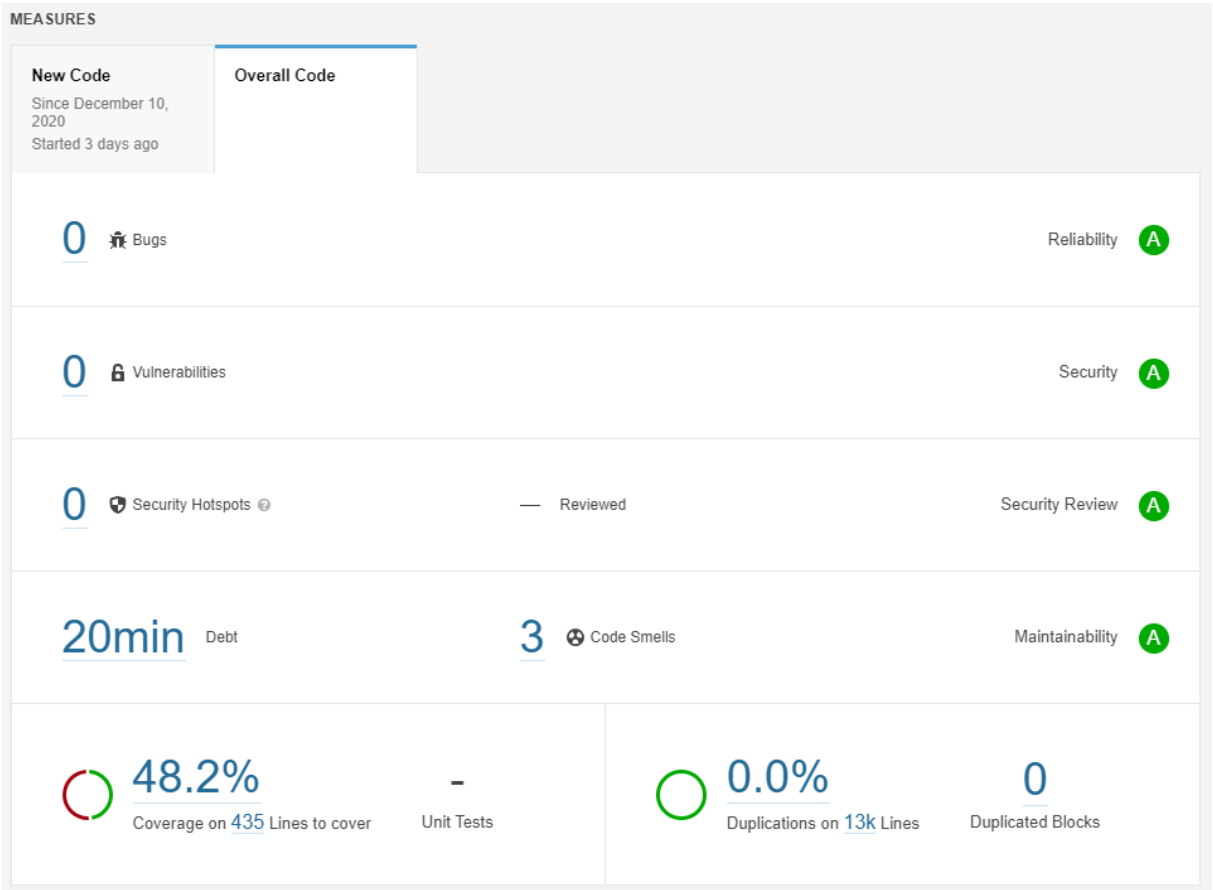
3.5 SonarQube

Afin de s'assurer de la qualité de notre code, de sa maintenabilité, nous utilisons SonarQube.

Sonar est un outil qui va s'attacher au processus de compilation de notre application, vérifier que tout se passe bien, et également lire notre code pour nous donner les problèmes qu'il relève.

Il est aussi capable de nous donner le pourcentage de duplication de code, et le pourcentage de code testé, pour cela, nous lui fournissons un fichier .opencover.xml qui renseigne toutes les parties du code que nous testons.

Voici le résultat de notre passe Sonar sur l'application :



Nous pouvons retenir que le résultat est très bon, les seuls Code Smells retenus sont des faux positifs car cela provient du fonctionnement de l'application en elle-même. La part de couverture du code est plutôt faible, mais cela des dû au fait que nous n'avons pas su restreindre les fichiers analysés par Sonar uniquement sur nos fichiers.

4. Présentation globale et technique Docker et Docker-Compose

4.1 Conteneuriser l'application

Afin de rendre le déploiement et le rendu du projet plus facile nous avons conteneurisé l'application.

Pour cela nous avons besoin de trois conteneurs, un conteneur avec l'application (le code .Net), un conteneur avec notre base de données, et un conteneur avec un serveur Nginx permettant d'avoir le https, et la redirection http vers https.

Pour lier les trois conteneurs, nous utilisons Docker-Compose, qui nous permet de spécifier des liens entre conteneurs, ainsi que des variables d'environnement, et de build ces conteneurs.

4.2 Conteneur de l'application

Ce conteneur aura en son sin l'application .Net, voici le Dockerfile correspondant :

CashManager > CashManager > CashManager.Dockerfile > FROM

```
1   FROM mcr.microsoft.com/dotnet/core/aspnet:3.1-buster-slim AS base
2   WORKDIR /app
3
4   FROM mcr.microsoft.com/dotnet/core/sdk:3.1-buster AS build
5   WORKDIR /src
6   COPY ["CashManager.csproj", "./"]
7   RUN dotnet restore "./CashManager.csproj"
8   COPY . .
9   WORKDIR "/src/."
10  RUN dotnet build "CashManager.csproj" -c Release -o /app/build
11
12  FROM build AS publish
13  RUN dotnet publish "CashManager.csproj" -c Release -o /app/publish
14
15  FROM base AS final
16  WORKDIR /app
17  COPY --from=publish /app/publish .
18  ENV ASPNETCORE_URLS http://*:5000
19  ENTRYPOINT ["dotnet", "CashManager.dll"]
```

Dans ce Dockerfile, nous utilisons l'image fournie par Microsoft, contenant le SDK .Net Core, ce qui nous permet de récupérer les packages utilisés par l'application.

Il faut noter que nous publions l'application, ce qui produit une .dll. C'est donc ce qui est utilisé après lorsque l'application n'est plus au stade de développement mais production voire recette.

4.3 Conteneur de la base de données

Le conteneur de la base de données utilise une image Microsoft pour SQL Server : « mcr.microsoft.com/mssql/server:2017-latest ».

Nous avons fait en sorte que notre application se connecte à cette base de données lors du déploiement en modifiant la connexion par défaut précisée dans le fichier appsettings.json.

De plus, nous avons ajouté une condition lors du démarrage de l'application, si la base de données n'a pas toutes les migrations d'appliquées, on les applique, voici le code :

```
using (var serviceScope = app.ApplicationServices.GetRequiredService<IServiceScopeFactory>().CreateScope())
{
    var context = serviceScope.ServiceProvider.GetService<ApplicationDbContext> ();
    if (context.Database.GetPendingMigrations().Any())
    {
        context.Database.Migrate();
    }
}
```

4.4 Conteneur Nginx

Le conteneur Nginx nous permet d'avoir un serveur de redirection pour notre application, ainsi nous n'avons pas à spécifier de certificat directement dans l'application, ce dernier est renseigné dans le conteneur Nginx, et il est créé sur le serveur hôte, dans notre cas. Il est copié dans le conteneur depuis le Dockerfile.

Nginx a besoin d'un fichier de configuration que voici :

CashManager > Nginx > nginx.conf

```
worker_processes 1;

events { worker_connections 1024; }

http {

    sendfile on;

    upstream web-api {
        server cashmanager:5000;
    }

    server {
        listen 80;
        server_name localhost;

        location / {
            return 301 https://$host$request_uri;
        }
    }

    server{
        listen 443 ssl;
        server_name localhost;

        ssl_certificate /etc/ssl/certs/localhost.crt;
        ssl_certificate_key /etc/ssl/private/localhost.key;

        location / {
            proxy_pass         http://web-api;
            proxy_redirect     off;
            proxy_http_version 1.1;
            proxy_cache_bypass $http_upgrade;
            proxy_set_header   Upgrade $http_upgrade;
            proxy_set_header   Connection keep-alive;
            proxy_set_header   Host $host;
            proxy_set_header   X-Real-IP $remote_addr;
            proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header   X-Forwarded-Proto $scheme;
            proxy_set_header   X-Forwarded-Host $server_name;
        }
    }
}
```

Dans cette configuration nous gérons le cas d'un appel http avec une redirection, et nous gérons la connexion sécurisée https avec un certificat général par OpenSSL.

4.5 Docker-Compose

Pour lier les trois conteneurs dont nous avons besoin nous utilisons docker-compose qui servira d'orchestrateur entre nos conteneurs, voici le code :

CashManager > Nginx > docker-compose.yml

```yaml
version: '3.4'

services:

  reverseproxy:
    build:
      context: .
      dockerfile: Nginx.Dockerfile
    ports:
      - "80:80"
      - "443:443"
    restart: always

  cashmanager:
    build:
      context: ../CashManager
      dockerfile: CashManager.Dockerfile
    depends_on:
      - reverseproxy
      - db
    expose:
      - "5000"
    restart: always

  db:
    image: "mcr.microsoft.com/mssql/server:2017-latest"
    environment:
        - ACCEPT_EULA=Y
        - SA_PASSWORD=1Secure*Password1
```

Ici nous définissons 3 services, la db, l'application et le proxy Nginx.

L'application est exposée sur le port 5000, port que nous écoutons dans la configuration Nginx, et Nginx écoute les ports 80 (http) et 443 (Https).

Il est donc possible de faire des requêtes sur l'URL (localhost/User).

5. Processus CI/CD

5.1 Travis-ci

Travis-ci est un service d'intégration continue (Continuous Integration / CI) et de déploiement continu (Continuous Deployment / CD).

Sur le principe le fonctionnement est simple, nous attachons le service sur un repo, Github dans notre cas, et lors de chaque push ou merge, Travis est activé et lance son processus. Il est possible de limiter Travis sur certaines branches, tout comme le déploiement peut se faire uniquement sur des branches, la dev, la prod ou la main par exemple. Nous avons par défaut Travis sur la branche main.

Nous utilisons Travis pour qu'il vérifie que le build de notre application ne retourne pas d'erreurs. Ensuite il effectue les tests, et si ces deux étapes se passent bien, il va construire nos images Docker avec notre fichier docker-compose, puis il va tagger les images créées (changer le nom), il va se connecter à un compte docker et push ces images sur docker hub, qui est un repository d'images Docker.

Ensuite il se connectera via ssh au serveur et lancera un script de deploiement.

Travis a besoin d'un fichier de configuration pour effectuer tout ce que nous voulons.

5.2 Le fichier .travis.yml

Voici le fichier qui détermine ce que Travis doit effectuer :

```
! .travis.yml
1   language: chsarp
2   mono: none
3   dist: xenial
4   dotnet: 3.1
5   addons:
6    ssh_known_hosts: 91.166.139.178
7    apt:
8        packages:
9        - sshpass
10  services:
11  - docker
12  install:
13  - cd ./CashManager
14  - dotnet restore ./CashManager/CashManager.csproj
15  - dotnet restore ./CashManagerTests/CashManagerTests.csproj
16  script:
17  - dotnet build ./CashManager/CashManager.csproj
18  - dotnet test CashManagerTests/CashManagerTests.csproj
19  after_success:
20  - cd ./Nginx
21  - docker-compose build
22  - docker tag nginx_cashmanager lucasboisbourdin/nginx_cashmanager_app
23  - echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_LOGIN" --password-stdin
24  - docker push lucasboisbourdin/nginx_cashmanager_app
25  - cd ..
26  - openssl aes-256-cbc -K $encrypted_db2095f63ba3_key -iv $encrypted_db2095f63ba3_iv
27    -in deploy_rsa.enc -out /tmp/deploy_rsa -d
28  - eval "$(ssh-agent -s)"
29  - chmod 600 /tmp/deploy_rsa
30  - ssh-add /tmp/deploy_rsa
31  - sshpass -p "$LUCAS_PASSWORD" ssh -o StrictHostKeyChecking=no lucas@91.166.139.178 Documents/Nginx/deploy_script.sh
```

Dans ce fichier, nous buildons uniquement le conteneur de l'application, car les conteneurs d'Nginx et de la db restent les mêmes, nous utilisons donc leur images (et dans le docker-compose nous utilisons également les images faites précédemment).

La partie « install » récupère les dépendances de nos projets, la partie « script » build l'appli et exécute les tests, la partie « after_success » construit et publie nos images Docker, puis se connecte en SSH et lance le script.

5.3 Le déploiement sur le serveur

Le déploiement sur le serveur est actuellement exécuté sur un serveur personnel, dont nous détailleront le système dans la prochaine partie.

Pour effectuer cette connexion via SSH, nous avons dû créer une clé publique et une clé privée. La clé publique est enregistrée auprès de l'host, et la clé privée a été encryptée par Travis. Cela nécessite d'installer Ruby puis le package Travis.

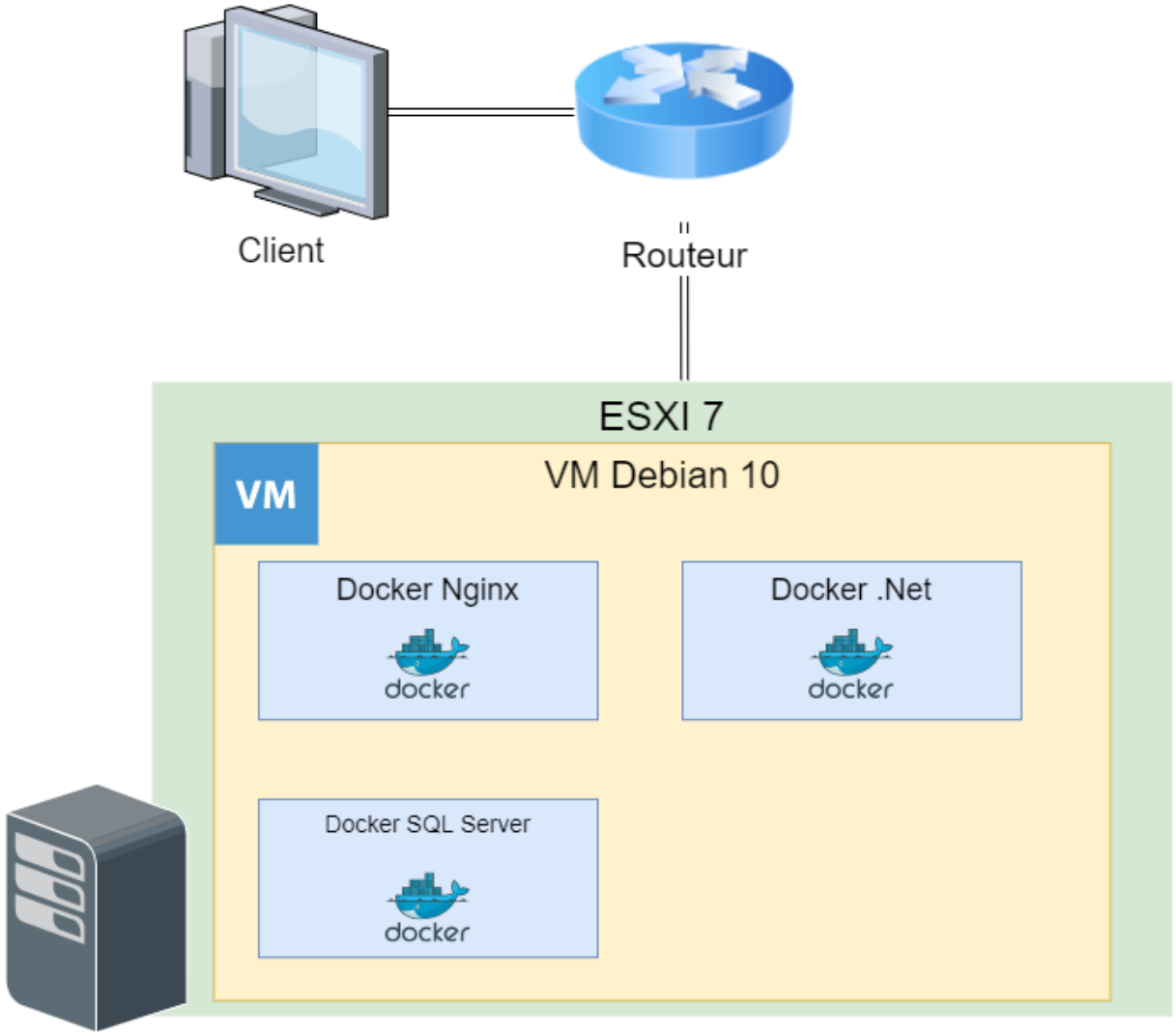
Pour encrypter la clé, nous devons nous connecter à Travis via CLI, générer une version encryptée de la clé privée (.enc) et l'ajouter à Travis (dans les settings, qui gardent la référence du fichier, ainsi que d'autres variables tels que les login docker hub).

Ensuite dans Travis nous utilisons OpenSSL pour décrypter le fichier, et ainsi l'utiliser pour la connexion SSH.

Nous utilisons le package sshpass (précisé dans addons) fin de passer le mot de passe de l'utilisateur via lequel nous nous connectons. Nous pouvons ensuite passer la commande d'exécution du script.

6. Serveur hôte

6.1 Architecture du système



Le serveur est un serveur Dell T320, avec un Hyperviseur de Niveau 1 : ESXI (version 7.0). Sur cet Hyperviseur il y a une VM Debian 10, dans laquelle nous avons installé docker, docker-compose, ce qui nous permet d'avoir un fichier docker-compose dans lequel on ne build pas mais où on pull les images créées précédemment.

6.2 Gestion des droits

Il a fallu dans un premier temps accorder les droits sudo puis les droits des commandes docker et docker-compose à l'utilisateur Lucas afin de pouvoir exécuter le script sans avoir besoin de sudo et de mot de passe.

Le script consiste à faire la commande docker-compose down, afin d'éteindre les conteneurs, ensuite faire docker-compose pull pour récupérer les nouvelles images, et docker-compose up pour les monter.

6.3 Gestion des ports

Pour des raisons de sécurité, le serveur n'est pas exposé au WAN, cependant il a fallu exposer la VM, nous avons donc ouvert les ports du routeur du réseau local dans lequel est le serveur. Nous avons donc redirigé les ports 8080 vers le port 80 (http) de la VM, le port 8443 vers le port 443 (https) de la VM, et enfin le port 22 vers le port 22 (SSH) de la VM.

Nous n'avons mis aucune sécurité particulière en place, car cette installation reste provisoire et nous manquions de temps.

6.4 Certificat auto-signé

Afin de pouvoir exécuter nos requêtes au serveur via https, il a fallu faire un certificat auto signé, pour cela nous avons utilisé OpenSSL.

La commande fût la suivante :

"sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout localhost.key -out localhost.crt -config localhost.conf -passin pass: {Votre mot de passe}"

Cela nous permet de récupérer un certificat que nous pourrons utiliser sur le serveur hôte.

7. Les routes

Il est important de noter que l'application possède un seed, qui rempli la base de données avec des données par défaut, ces dernières sont :

```
//We seed data to the User's table
modelBuilder.Entity<User>().HasData(
    new User
    {
        Id = 1,
        Username = "Username1",
        Password = "Password1",
        NbOfWrongCheques = 0,
        NbOfWrongCards = 0
    }
);

//We seed data to the User's table
modelBuilder.Entity<User>().HasData(
    new User
    {
        Id = 2,
        Username = "Username2",
        Password = "Password2",
        NbOfWrongCheques = 2,
        NbOfWrongCards = 2
    }
);

//We seed data to the BankAccount's table
modelBuilder.Entity<BankAccount>().HasData(
    new BankAccount
    {
        Id = 1,
        Balance = 10000,
        OwnerId = 1
    }
);

//We seed data to the Product's table
modelBuilder.Entity<Product>().HasData(
    new Product
    {
        Id = 1,
        Name = "Produit de test",
        Price = 10,
        Reference = "00000001"
    }
);
```

Ces données vous permettront d'effectuer des requêtes sur le serveur.

Voici la liste des routes accessibles :

- Route par défaut GET : https://91.166.139.178:8443/User/
- Route de Login (User 1) POST : https://91.166.139.178:8443/User/Login?username=Username1&password=Password1
- Route de prix d'un produit GET : https://91.166.139.178:8443/User/GetProductPrice?productReference=00000001
- Route de Paiement : https://91.166.139.178:8443/User/Pay?userId=1&ammount=50&isCreditCard=true