Documentation - Infrastructure systèmes et réseaux - projet LP DIWA

Rémi Venant <remi.venant@univ-lemans.fr>

Janvier 2023 - V1.0

Dans le cadre de votre projet, vous êtes amenés à réaliser une application web repartie reposant sur différentes technologies :

- Les frameworks **React** et **MobX** pour la partie *frontend* ;
- Le framework PHP **Symfony** pour la partie *backend* ;
- Un serveur de base de données relationnelle **MariaDB** (équivalent MySQL) ;
- Un serveur de base de données NoSQL orientée Document **MongoDB**.

Pour faciliter le processus de développement et de déploiement de votre application, une infrastructure logicielle reposant sur une pile de conteneurs **Docker** est mise à votre disposition,
ainsi qu'une **application de démonstration** exploitant la pile et les différentes technologies.

Ce document :

- décrit l'infrastructure pour le développement et celle pour la pré-production et la production ;
- décrit brievement la structure du dépôt git d'où est issue cette documentation ;
- fournie plusieurs commandes de base pour manipuler la pile.

Prérequis

Ce projet s'appuie uniquement sur **Docker** et sur son outil de gestion automatisée **docker-compose**. Pour installer ces logiciels sur votre machine, veuillez suivre la procédure suivante, selon votre système d'exploitation.

1. Installation de Docker (déjà présent sur les machines de l'IUT)

Système d'exploitation	Outil	Procédure d'installation
Linux Debian	Docker	https://docs.docker.com/engine/install/debian/
Linux Ubuntu	Docker	https://docs.docker.com/engine/install/ubuntu/
Linux (any distrib.)	docker-compose	https://docs.docker.com/compose/install (choisir la rubrique Linux)
	Docker &	

Windows	docker-compose	https://docs.docker.com/docker-for-windows/install/
MacOS	Docker & docker-compose	https://hub.docker.com/editions/community/docker-ce-desktop-mac/
Autre	Docker	https://docs.docker.com/engine/install/
Autre	docker-compose	https://docs.docker.com/compose/install/

2. Préparation de Docker (pour Windows / Mac)

Pour pouvoir exploiter pleinement les fonctionnalités qu'offre Docker sur Windows ou macOS, il vous est nécessaire de vous authentifier sur docker hub (notamment pour télécharger des images).

- Lancez docker (Application "Docker Desktop" sur Windows ou macOS) ;
- Une fois lancé, ouvrir les réglages de l'application ;
- Authentifiez vous avec votre compte Docker (si vous n'en avez pas un, un formulaire de création vous sera proposé).

3. Préparation de Docker - proxy Le Mans (uniquement pour les machines physiques de l'IUT)

Pour les machines physiques de l'IUT uniquement (et pas vos ordinateurs connectés sur le réseau WiFi eduroam), vous devez configurer Docker pour fonctionner derrière le proxy de l'Université du Mans

- Ouvrez l'interface de Docker Desktop ;
- Allez dans la partie "Resources" -> "proxy" ;
- Activez la configuration de proxy "manual" ;
- Configurez le proxy avec les informations suivantes :
    - *Web server* : vproxy.univ-lemans.fr:3128
    - *Secured Web Server* : vproxy.univ-lemans.fr:3128
    - *Bypass* : localhost,127.0.0.1

Structure générale du projet

Le projet, au sens arborescence de fichiers, contient :

- les fichiers nécessaires à la pile docker, quelque soit l'environnement (dev, pré-prod ou prod) ;
- cette documentation ;
- votre application (partie *frontend* et partie *backend*).

1. Arborescence de fichiers du projet

L'arborescence des fichiers est structurée ainsi :

- **api/** : dossier contenant votre API REST développée avec *Symfony*.
- **app/** : dossier contenant votre application web développée avec *React* (et tout autre framework qui vous sera utile pour la partie *frontend*), et packagé avec l'utilitaire *webpack*.
- **databasesInit/** : dossier contenant optionnellement des scripts d'alimentation de vos base de données à leur création (cf. ci-dessous)
- **doc/** : dossier contenant cette documentation, ses fichiers sources et un utilitaire de compilation pour celle-ci.
- **docker/** : dossier contenant les configurations des piles Docker et tout autre fichiers nécessaires aux différents conteneurs de la pile (dont les variables d'environnements)
- **docker/helpers** : scripts shell facilitant la manipulation des conteneurs de bases de données (connexion, dump).

2. Dépôt git initial

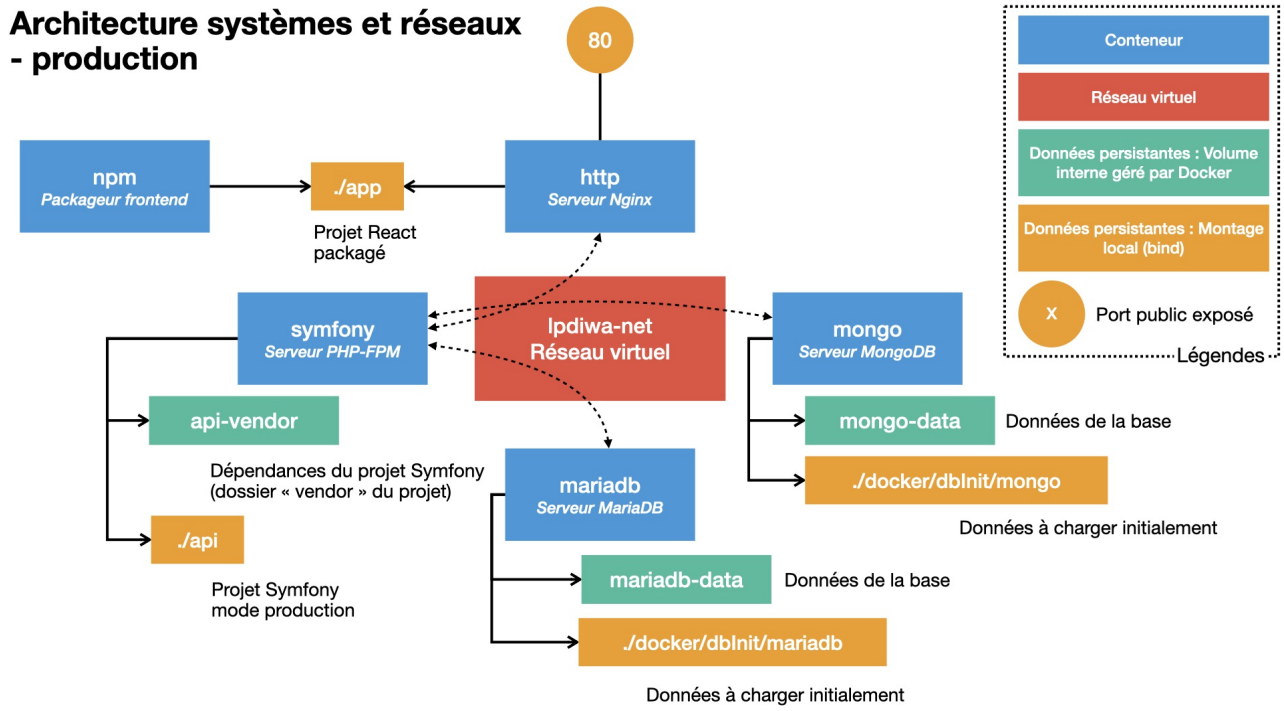
Le dépôt git de cette structure de projet propose **2** branches différentes que vous pouvez récupérer à votre guise :

- **main** : contient la structure de base de la pile docker, la structure initialisée des applications **backend** et **frontent**, les applications de démonstration. *Cette branche est celle conseillée pour commencer votre projet*.
- **structure-only** : contient la structure de base de la pile docker. Les applications *backend* et *frontend* ne sont pas initialisées.

Piles de conteneurs

Les deux piles de conteneurs utilisées pour votre projet sont décrites ici. La première est celle du développement tandis que la seconde est utilisé pour la production.

1. Pile de production



Au déploiement de la pile, tous les conteneurs suivants sont exécutés. Les volumes gérés par Docker et le réseau virtuel, s'ils n'existent pas déjà, sont créés. Chaque conteneur, volume et le réseau virtuel a un nom préfixé de "lpdiwa-project-prod" (pour faire la distinction avec les éléments de la pile de développement, décrite après).

- **mariadb** : serveur de bases de données relationnelles MariaDB.
    - Configuré avec le nom d'utilisateur, mot de passe et nom de base de donnée pour la production (informations décrites dans docker/env/**prod**/mariadb.env).
    - Utilise un volume géré par Docker "mariadb-data" pour stocker de façon pereinne les données.
    - Peut être alimenté en données, *à la création du volume uniquement* par tout script de chargement de données `.sql` contenu dans le répertoire docker/dbInit/mariadb. Le script docker/helpers/dump_mariadb.sh permet de générer automatiquement un dump de la base dans ce dossier
    - Ce serveur n'est accessible que depuis le réseau virtuel, interne à la pile "lpdiwa-net".
    - À l'exécution, le serveur se lance est reste à l'écoute de toute connexion.

- **mongo** : serveur de bases de données NoSQL orientées document MongoDB.
    - "Configuré" pour la production (informations décrites dans docker/env/**prod**/mongo.env).
    - Utilise un volume géré par Docker "mongo-data" pour stocker de façon pereinne les données.
    - Peut être alimentée en données, *à la création du volume uniquement*

par tout script de chargement `.js` ou `.sh` contenu dans le répertoire docker/dbInit/mongo. Un script *loader.sh* est déjà présent, permettant de charger les données issues d'un dump généré par le script docker/helpers/dump_mongo.sh
    - Ce serveur n'est accessible que depuis le réseau virtuel, interne à la pile "lpdiwa-net".
    - À l'exécution, le serveur se lance est reste à l'écoute de toute connexion.

- **symfony** : serveur d'exécution PHP-FPM. Ne gère que de l'exécution de code PHP.

    - Accède au dossier local **api** contenant le code de votre partie *backend*.
    - Utilise un volume géré par Docker "api-data", contenant les différents packages nécessaires de l'API (dossier **vendor** d'un projet Symfony), séparé du dossier local **api** pour des raisons de performance.
    - Ce serveur n'est accessible que depuis le réseau virtuel, interne à la pile "lpdiwa-net".
    - À l'exécution, le conteneur installe les différents packages décrits dans le fichier **composer.json** qui ne le seraient pas déjà dans le volume api-data (i.e. : `compose install`), puis le serveur se lance en mode production et reste à l'écoute de toute connexion.
    - Les variables d'environnement utiles à symfony (notamment pour se connecter aux bases de données) ne sont pas décrites dans le fichier .env (comme par habitude) mais injectées dans le conteneur via le fichier docker/env/**prod**/symfony.env (pour faciliter le passage du dev. à la prod.).

- **npm** : bundler de la partie *frontend*.

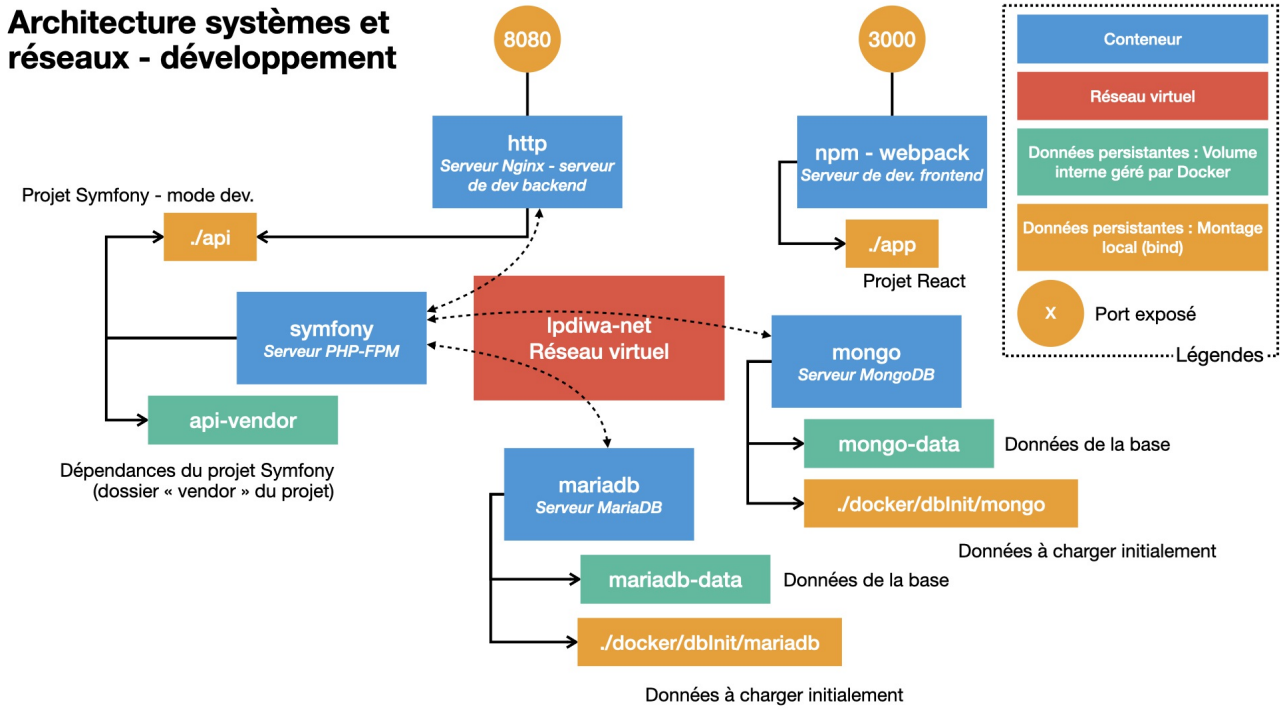
    - Accède au dossier local **app** contenant le code de votre partie *frontend*.
    - L'URL publique de l'API REST (Symfony) à laquelle doit accéder le navigateur exposant l'application cliente, est paramétrée par une variable d'environnement décrite dans docker/env/**prod**/npm.env.
    - À l'exécution, le conteneur installe les dépendances qui ne le seraientt pas déjà (i.e. : `npm install`) puis effectue un *build* de production de votre partie *frontend* peuplant le dossier **app/build** avec vos bundle js, la page index.html et tous fichiers nécessaire à votre application (images...), puis s'arrête. Également, un dossier **app/build/buildInfos** est généré contenant les rapports de build (utiles pour analyser le poids de votre application...). Ce dernier n'est jamais exposé publiquement.

- **http** : serveur HTTP Nginx de production

    - Accède au dossier local **app/build** contenant votre application *frontend*
    - toute requête destinée à l'url de préfix "/api/" est redirigée au serveur PHP-FPM **symfony**. Si la requête désigne un des fichiers présent dans **app/build**, le fichier est servi. Sinon la requête est redirigée vers

index.html (votre application *frontend* se chargeant du routage interne des vues)
- Ce serveur accède au réseau virtuel "lpdiwa-net", interne à la pile.
- Ce serveur est accessible de l'exterieur (machine hôte et réseau externe) par le port 80 de la machine hôte.

2. Pile de développement



Au déploiement de la pile, touts les conteneurs suivant sont exécutés. Les volumes gérés par Docker et le réseau virtuel, s'ils n'existent pas déjà, sont créés. Chaque conteneur, volume et le réseau virtuel ont un nom préfixé de "diwaDev" (pour faire la distinction avec les éléments de la pile de pré-prod/prod).

Les conteneur, volumes et le réseau virtuel sont donc bien différents que ceux de la pile précédente.

- **mariadb** : serveur de bases de données relationnelles MariaDB.
    - Configuré avec le nom d'utilisateur, mot de passe et nom de base de donnée pour le développement (informations décrites dans docker/env/**dev**/mariadb.env).
    - Utilise un volume géré par Docker "mariadb-data" pour stocker de façon pereinne les données.
    - Peut être alimenté en données, *à la création du volume uniquement* par tout script de chargement de données `.sql` contenu dans le répertoire docker/dbInit/mariadb. Le script docker/helpers/dump_mariadb.sh permet de générer automatiquement un dump de la base dans ce dossier
    - Ce serveur est accessible depuis le réseau virtuel, interne à la pile "lpdiwa-net", mais également sur l'interface de bouclage au port 3306

de la machine hôte.
    - À l'exécution, le serveur se lance est reste à l'écoute de toute connexion.

- **mongo** : serveur de bases de données NoSQL orientées document MongoDB.

    - "Configuré" pour le développement (informations décrites dans docker/env/**dev**/mongo.env).
    - Utilise un volume géré par Docker "mongo-data" pour stocker de façon pereinne les données.
    - Peut être alimentée en données, *à la création du volume uniquement* par tout script de chargement `.js` ou `.sh` contenu dans le répertoire docker/dbInit/mongo. Un script *loader.sh* est déjà présent, permettant de charger les données issues d'un dump généré par le script docker/helpers/dump_mongo.sh.
    - Ce serveur est accessible que depuis le réseau virtuel, interne à la pile "lpdiwa-net", mais également sur l'interface de bouclage au port 27017 de la machine hôte.
    - À l'exécution, le serveur se lance est reste à l'écoute de toute connexion.

- **symfony** : serveur d'exécution PHP-FPM. Ne fait que de l'exécution de code PHP.

    - Accède au dossier local **api** contenant le code de votre partie *backend*.
    - Utilise un volume géré par Docker "api-data", contenant les différents packages nécessaires de l'API (dossier **vendor** d'un projet Symfony), séparé du dossier local **api** pour des raisons de performance.
    - Ce serveur n'est accessible que depuis le réseau virtuel, interne à la pile "lpdiwa-net".
    - À l'exécution, le conteneur installe les différents packages décrits dans le fichier **composer.json** qui ne le seraient pas déjà dans le volume api-data (i.e. : `compose install`), puis le serveur se lance en mode développement et reste à l'écoute de toute connexion.
    - Les variables d'environnement utiles à symfony (notamment pour se connecter aux bases de données) ne sont pas décrites dans le fichier .env (comme par habitude) mais injectées dans le conteneur via le fichier docker/env/**dev**/symfony.env (pour faciliter le passage du dev. à la prod.).

- **npm** : serveur de développement de la partie *frontend*.

    - Accède au dossier local **app** contenant le code de votre partie *frontend*.
    - L'URL de l'API REST (Symfony) en mode prod (i.e.: http://localhost:8080/api) à laquelle doit accéder le navigateur exposant l'application cliente, est paramétrée par une variable d'environnement décrite dans docker/env/**prod**/npm.env.
    - À l'exécution, le conteneur installe les dépendances qui ne le seraientt pas déjà (i.e. : `npm install`) puis lance le serveur de développement de webpack et reste à l'écoute de toute connexion.

    - Ce serveur est accessible depuis la machine hôte (uniquement) par à l'adresse http://localhost:3000.

- **http** : serveur HTTP Nginx de développemnt

    - Accède au dossier local **api/public** contenant votre application *backend*, pour permettre de servir les interfaces web de dev de Symfony directement ainsi tout fichier statique (image, css) que vous pourriez être amené à manipuler pendant les phases de developpements de votre projet (pour tester une simple maquette HTML avec l'API par exemple).
    - Si la requête désigne un fichier présent dans **app/public** qui ne soit pas un fichier .php, le fichier est servi. Sinon la requête est redirigée vers le serveur php-fpm qui traitera la requête.
    - Ce serveur accède au réseau virtuel "lpdiwa-net", interne à la pile.
    - Ce serveur est accessible depuis la machine hôte (uniquement) par à l'adresse http://localhost:8080.

Manipulation et cas d'utilisation

En dehors de certain cas d'utilisation particuliers, l'ensemble des commandes de la pile se font via **docker-compose**.

Tous les exemples suivants sont pris avec la pile dev, mais fonctionnent de la même manière avec la pile prod.

1. Paramétrage du proxy

Que ce soit en environnemnt de développement ou en production, si vous êtes derrière un proxy non transparent, vous devez éditer le fichier .env utile à Docker à cet effet. Décommentez les 3 lignes de variables d'environnement HTTP_PROXY, HTTPS_PROXY et NO_PROXY et redéfinissez-les au besoin.

2. Lancement et arrêt la pile et mode développement

- Lancement : `docker-compose up`

    - Créer le réseau virtuel et les différents volumes gérés par Docker (s'ils n'existent pas déjà).
    - Lance les différents conteneurs.
    - Tous les conteneurs affichent leur sortie dans le terminal courant.
    - Le terminal courant étant occupé, un autre terminal doit être utilisé pour des commandes parallèles (ex.: arrêt de la pile).

- Création du schéma SQL : `docker-compose exec symfony php bin/console doctrine:migrations:migrate`

    - Ne fonctionne que lorsque la pile est lancée (notamment le service mariadb)

    - À ne faire qu'au premier lancement où si vous avez préalablement supprimé le volume mariadb-data

- Arrêt : `docker-compose down`

    - Stoppe et supprime les différents conteneurs ainsi que le réseau virtuel, mais conserve les volumes.
    - Si l'option **-v** est fournie, supprime également les volumes.

- Vérification de l'état de la pile : `docker-compose ps`

    - Lorsque la pile est lancé, tous les conteneurs devraient être "**running**" avec l'indicateur "**(healthy)**" à l'exception du service npm, qui n'a pas de système de vérification de son état.

La première fois que vous lancerez la pile, Docker rappatriera d'Internet toutes les images nécessaires aux différents conteneurs et créera ces conteneurs. Cette opération nécessite une bonne connexion à internet, et du temps. Soyez préoyant !

3. Lancement et arrêt la pile et mode production

- Lancement : `docker-compose -f docker-compose-prod.yml up`

    - Créer le réseau virtuel et les différents volumes gérés par Docker (s'ils n'existent pas déjà).
    - Lance les différents conteneurs.
    - Tous les conteneurs affichent leur sortie dans le terminal courant.
    - Le terminal courant étant occupé, un autre terminal doit être utilisé pour des commandes parallèles (ex.: arrêt de la pile).

- Création du schéma SQL : `docker-compose -f docker-compose-prod.yml exec symfony php bin/console doctrine:migrations:migrate`

    - Ne fonctionne que lorsque la pile est lancée (notamment le service mariadb)
    - À ne faire qu'au premier lancement où si vous avez préalablement supprimé le volume mariadb-data

- Arrêt : `docker-compose -f docker-compose-prod.yml down`

    - Stoppe et supprime les différents conteneurs ainsi que le réseau virtuel, mais conserve les volumes.
    - Si l'option **-v** est fournie, supprime également les volumes.

- Vérification de l'état de la pile : `docker-compose -f docker-compose-prod.yml ps`

    - Lorsque la pile est lancé, tous les conteneurs devraient être "**running**" avec l'indicateur "**(healthy)**" à l'exception du service npm, qui s'arrête après le build du *frontend* (et dont le code de sortie si tout s'est bien passé doit être 0).

La première fois que vous lancerez la pile, Docker rappatriera d'Internet toutes les images nécessaires aux différents conteneurs et créera ces conteneurs. Cette opération nécessite une bonne connexion à internet, et du temps. Soyez préoyant !

Conseil pour le développement (en mode développement)

---

1. Installation de dépendances pour Symfony

Pour installer une dépendance, nous lançons un autre conteneur **symfony** dédié que nous supprimons dès que l'installation est terminée. Par exemple la commande suivante installe la dépendance *symfony/make-bundle* en tant que dépendance de dev (avec l'option *--dev*, propre à composer) :

```
docker-compose run --rm symfony composer require symfony/maker-bundle --dev
```

Comme cette installation est réalisée dans un autre conteneur, elle peut être réalisée que la pile principale soit lancée ou non.

2. Installation de dépendances pour React

De la même manière, nous utilisons un conteneur temporaire npm pour installer une dépendance de l'application *frontend* :

```
docker-compose run --rm npm npm install eslint-plugin-babel --save-dev
```

La double occurence du terme "npm" n'est pas une erreur : le premier "npm" désigne le conteneur à lancer. Le second "npm" désigne la commande à executer dans ce conteneur. Comme cette installation est réalisée dans un autre conteneur, elle peut être réalisée que la pile principale soit lancée ou non.

3. Commandes avancées pour symfony

Avec symfony, de nombreuses commande vous sont proposées (ex.: créer un controlleur, une entité gérée par doctrine...). Ces commandes sont tout à fais executables via Docker.

Par exemple, les commandes suivantes permettent de créer une entité, préparer la migration de schéma et effectuer la migration.

```
docker-compose run --rm symfony php bin/console make:entity
docker-compose run --rm symfony php bin/console make:migration
docker-compose run --rm symfony php bin/console
doctrine:migrations:migrate
```

Gestion des BD

---

1. Accès à la base de données MariaDb

Que vous soyez en mode dev. ou prod., vous pouvez accéder en ligne de commande au SGBD MAriaDb en exécutant le script `./docker/helpers/goto_mariadb.sh`. Evidemment, le service **mariadb** doit être lancé.

En mode dev., le service est également accessible depuis le port 3306 de votre interface de bouclage (127.0.0.1). Vous pouvez donc utiliser un utilitaire exterieur comme phpMyAdmin ou tout autre client de SGBD.

2. Accès à la base de données Mongodb

Que vous soyez en mode dev. ou prod., vous pouvez accéder en ligne de commande au SGBD MAriaDb en exécutant le script `./docker/helpers/goto_mariadb.sh`. Evidemment, le service **mongo** doit être lancé.

En mode dev., le service est également accessible depuis le port 27017 de votre interface de bouclage (127.0.0.1). Vous pouvez donc utiliser un utilitaire exterieur.

3. Édition du code

Le développement de votre application ne nécéssite aucune changement de pratique de votre part, et s'en trouve même simplifié lorsqu'il s'agit de passer de l'environnement dev à l'environnement de pré-prod/prod.

3.1. Édition du code en phase de développement

Une fois la pile de développement lancé, vous pouvez éditer le code *backend* ou *frontend* comme vous le souhaitez.

Toute modification du code *frontend* sera détecté par le serveur de dev. webpack (conteneur **npm**), ce qui provoquera le rafraichissement automatique de vos pages de navigateur connectées à ce serveur.

Toute modification du code *backend* sera pris en compte par symfony à la prochaine requête. Attention, le temps de traitement de cette prise en compte rallonge considérablement le temps de traitement de la requête (plusieurs secondes). Une fois pris en compte, les requêtes suivantes sont exécutées de nouveau avec un temps normal.

3.1. Référence à l'API *backend* depuis le code *frontend*

Au sein de votre code React de la partie *frontend*, vous allez devoir accéder à l'API exposée par la partie *backend*. Comme l'url de l'API est différente en dev et en prod, une variable d'environnement est utilisée pour ne pas à avoir à modifier le code-source lors du passage d'un environnement à l'autre. Cette variable, nommée **API_EP_URI** est définie dans le fichier de configuration du conteneur **npm** (docker/env/<dev|prod>/npm.env).

Pour l'injecter dans votre code, il vous suffit d'y faire référence par la constante **APP_ENV.API_EP_URI**.
Par exemple, un fetch sur un service REST "livres" de la démo pourrait être

```
fetch(`${APP_ENV.API_EP_URI}/livres`).then(...)
```

Un exemple concret est donné dans l'application de démonstration (fichier app/src/model/ToDoListManager.js).

3.1. Problématique CORS en développement

En environnement de développement, votre application *frontend* est accessible depuis le serveur de developpement webpack tandis que votre API *backend* l'est depuis le serveur http. Ainsi vous vous trouvez en situation de *partage de ressources d'origines différentes (CORS, Cross-Origin Resource Sharing)*. Pour des raisons de sécurité, votre navigateur aborde un comportement différent dans ce cas. Lorsque votre application *frontend* fera une requête à l'API (depuis le navigateur, donc), celui-ci :

- commencera par faire une requête "*preflight*" au serveur (même URL, mais code HTTP différent "OPTIONS"), dont le but est de demander au serveur si cette requête est acceptable
- si le serveur répond par la positive (code 2**), et avec les en-têtes HTTP attendus, definissant si le site d'origine est autorisé pour demander cette requête, le navigateur effectura la véritable la requête.

La pile docker de développement est configurée pour permettre ce fonctionnement, notamment en injectant tous les en-têtes HTTP nécessaires aux requêtes CORS (configuration du serveur HTTP Nginx). Toutefois, il est nécessaire de répondre aux différentes requêtes "OPTIONS" depuis votre code *backend*. Un exemple d'un controlleur "idiot" répondant toujours par la positive pour ces requêtes est fourni dans l'application de démonstration (src/Controller/OptionsRequestsController.php).

Vous pouvez tout à fait laisser ce controlleur en production, qui sera sans effêt dans cet environnement puisque le serveur HTTP, configuré en production, n'injectera pas les en-têtes HTTP nécessaires pour permettre des requêtes CORS.