TUTORIAL

Conteneurisation d'une application Ruby on Rails pour le développement avec Docker Compose

Ruby on Rails Docker PostgreSQL Redis

By Kathleen Juell

Published on April 23, 2020 3.4k

Français

Introduction

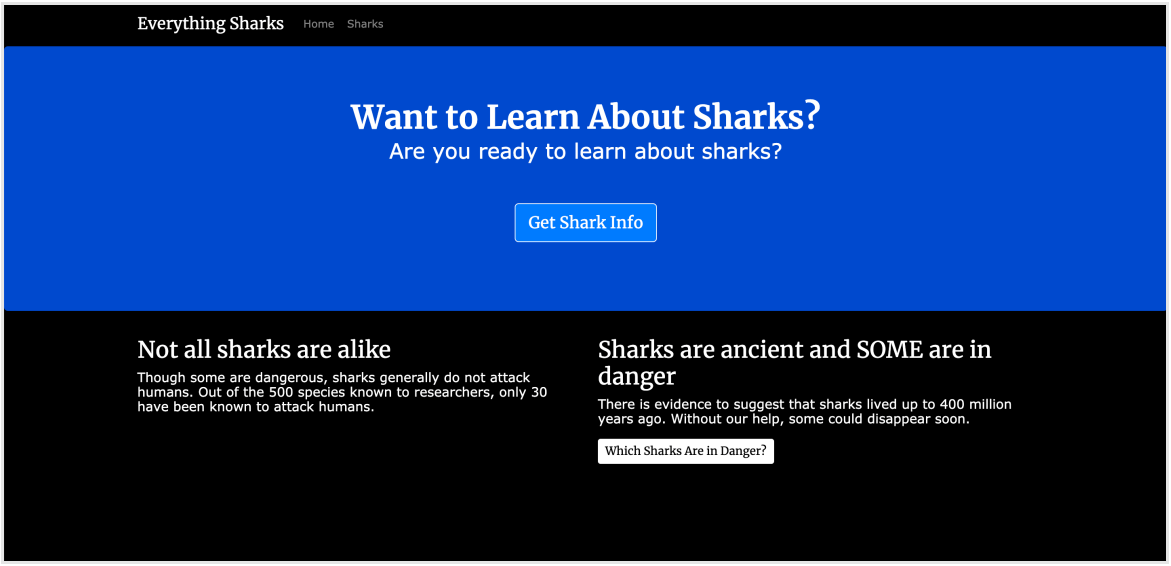
Si vous développez activement une application, l'utilisation de Docker peut simplifier votre flux de travail et le processus de déploiement de votre application en production. Le fait de travailler avec des conteneurs en cours de développement offre les avantages suivants :

- Les environnements sont cohérents, ce qui signifie que vous pouvez choisir les langues et les dépendances que vous souhaitez pour votre projet sans vous soucier des conflits de système.
- Les environnements sont isolés, ce qui facilite la résolution des problèmes et l'intégration des nouveaux membres de l'équipe.
- Les environnements sont portables, ce qui vous permet de mettre votre code en paquets et de le partager avec d'autres.

Ce tutoriel vous montrera comment mettre en place un environnement de développement pour une application Ruby on Rails en utilisant Docker. Vous allez créer plusieurs conteneurs - pour l'application elle-même, la base de données PostgreSQL, Redis, et un service Sidekiq - avec Docker Compose. L'installation se fera de la manière suivante :

- Synchronisez le code de l'application sur l'hôte avec le code dans le conteneur pour faciliter les changements pendant le développement.
- Données d'application persistantes entre les redémarrages des conteneurs.
- Configurez les travailleurs de Sidekiq de manière à ce que les emplois soient traités comme prévu.

À la fin de ce tutoriel, vous disposerez d'une application d'information sur les requins fonctionnant sur les conteneurs Docker :



Conditions préalables

Pour suivre ce tutoriel, vous aurez besoin des éléments suivants :

- Une machine ou un serveur de développement local fonctionnant sous Ubuntu 18.04, ainsi qu'un utilisateur non root avec des privilèges `sudo` et un pare-feu actif. Pour savoir comment les configurer, veuillez consulter le présent Guide de configuration initiale du serveur.
- Docker installé sur votre machine locale ou votre serveur, en suivant les Étapes 1 et 2 de Comment installer et utiliser Docker sur Ubuntu 18.04.
- Docker Compose installé sur votre machine locale ou votre serveur, en suivant l'Étape 1 de Comment installer Docker Compose sur Ubuntu 18.04.

Étape 1 - Clonez le projet et ajoutez les dépendances

Notre première étape consistera à cloner le référentiel rails-sidekiq à partir du compte GitHub de la communauté DigitalOcean. Ce référentiel comprend le code de la configuration décrite dans la section Comment ajouter Sidekiq et Redis à une application Ruby on Rails, qui explique comment ajouter Sidekiq à un projet Rails 5 existant.

Clonez le référentiel dans un répertoire appelé `rails-docker` :

```
$ git clone https://github.com/do-community/rails-sidekiq.git rails-docker
```

Naviguez jusqu'au répertoire `rails-docker` :

```
$ cd rails-docker
```

Dans ce tutoriel, nous utiliserons PostgreSQL comme base de données. Afin de travailler avec PostgreSQL au lieu de SQLite 3, vous devrez ajouter le `pg` gem aux dépendances du projet, qui sont listées dans son Gemfile. Ouvrez ce fichier pour le modifier en utilisant `nano` ou votre éditeur

préféré :

```
$ nano Gemfile
```

Ajoutez le gem n'importe où dans les principales dépendances du projet (au-dessus des dépendances de développement) :

~/rails-docker/Gemfile

```
. . .
# Reduces boot times through caching; required in config/boot.rb
gem 'bootsnap', '>= 1.1.0', require: false
gem 'sidekiq', '~>6.0.0'
gem 'pg', '~>1.1.3'

group :development, :test do
. . .
```

Nous pouvons également supprimer le gem `sqlite`, puisque nous ne l'utiliserons plus :

~/rails-docker/Gemfile

```
. . .
# Use sqlite3 as the database for Active Record
# gem 'sqlite3'
. . .
```

Enfin, décommentez le gem `spring-watcher-listen` en `cours de développement` :

~/rails-docker/Gemfile

```
. . .
gem 'spring'
# gem 'spring-watcher-listen', '~> 2.0.0'
. . .
```

Si nous ne désactivons pas ce gem, nous verrons des messages d'erreur persistants lors de l'accès à la console Rails. Ces messages d'erreur proviennent du fait que ce gem a des Rails qui utilisent `listen` pour chercher des changements dans le développement, plutôt que d'interroger le système de fichiers pour connaître les changements. Comme ce gem surveille la racine du projet, y compris le répertoire `node_modules`, il lancera des messages d'erreur sur les répertoires surveillés, encombrant ainsi la console. Toutefois, si vous êtes soucieux de préserver les ressources de l'unité centrale, désactiver ce gem pourrait ne pas vous convenir. Dans ce cas, il peut être judicieux de faire passer votre application Rails à Rails 6.

Enregistrez et fermez le fichier lorsque vous avez terminé de le modifier.

Avec votre référentiel de projet en place, le gem `pg` ajouté à votre Gemfile, et le gem `spring-watcher-listen` décommenté, vous êtes prêt à configurer votre application pour qu'elle fonctionne avec PostgreSQL.

Étape 2 - Configurez l'application pour qu'elle fonctionne avec PostgreSQL et Redis

Pour travailler avec PostgreSQL et Redis en développement, exécutez les opérations suivantes :

- Configurez l'application pour qu'elle fonctionne avec PostgreSQL comme adaptateur par défaut.
- Ajoutez un fichier `.env` au projet avec le nom d'utilisateur et le mot de passe de notre base de données et l'hôte Redis.
- Créez un script `init.sql` afin de créer un utilisateur `sammy` pour la base de données.
- Ajoutez un initialisateur pour Sidekiq afin qu'il puisse fonctionner avec notre service `Redis` conteneurisé.
- Ajoutez le fichier `.env` et d'autres fichiers pertinents aux fichiers `gitignore` et `dockerignore` du projet.
- Créez des seeds de base de données afin que notre application dispose de quelques enregistrements avec lesquels nous pourrons travailler lorsque nous la lancerons.

Tout d'abord, ouvrez le fichier de configuration de votre base de données, situé à l'adresse `config/database.yml` :

```
$ nano config/database.yml
```

Actuellement, le fichier comprend les paramètres par `défaut` suivants, qui sont appliqués en l'absence d'autres paramètres :

~/rails-docker/config/database.yml

```
default: &default
  adapter: sqlite3
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  timeout: 5000
```

Nous devons les modifier pour tenir compte du fait que nous utiliserons l'adaptateur `postgresql`, puisque nous allons créer un service PostgreSQL avec Docker Compose pour conserver nos données d'application.

Supprimez le code qui définit SQLite comme l'adaptateur et remplacez-le par les paramètres suivants, qui définiront l'adaptateur de manière appropriée et les autres variables nécessaires à la connexion :

~/rails-docker/config/database.yml

```
default: &default
  adapter: postgresql
  encoding: unicode
  database: <%= ENV['DATABASE_NAME'] %>
  username: <%= ENV['DATABASE_USER'] %>
```

```
  password: <%= ENV['DATABASE_PASSWORD'] %>
  port: <%= ENV['DATABASE_PORT'] || '5432' %>
  host: <%= ENV['DATABASE_HOST'] %>
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  timeout: 5000
. . .
```

Ensuite, nous allons modifier le paramétrage de l'environnement de `development`, puisque c'est l'environnement que nous utilisons dans cette configuration.

Supprimez la configuration existante de la base de données SQLite pour que cette section ressemble à ceci :

~/rails-docker/config/database.yml

```
. . .
development:
  <<: *default
. . .
```

Enfin, supprimez également les paramètres de la `database` pour les environnements de `production` et de `test` :

~/rails-docker/config/database.yml

```
. . .
test:
  <<: *default

production:
  <<: *default
. . .
```

Ces modifications des paramètres par défaut de notre base de données nous permettront de définir dynamiquement les informations de notre base de données en utilisant les variables d'environnement définies dans les fichiers `.env`, qui ne seront pas soumises au contrôle de version.

Enregistrez et fermez le fichier lorsque vous avez terminé de le modifier.

Notez que si vous créez un projet Rails à partir de zéro, vous pouvez définir l'adaptateur avec la commande `rails new`, comme décrit dans l'Étape 3 de Comment utiliser PostgreSQL avec votre application Ruby on Rails sur Ubuntu 18.04. Votre adaptateur sera ainsi placé dans le fichier `config/database.yml` et le gem `pg` sera automatiquement ajouté au projet.

Maintenant que nous avons référencé nos variables d'environnement, nous pouvons leur créer un fichier avec nos paramètres préférés. Extraire les paramètres de configuration de cette manière fait partie de l'approche à 12 facteurs du développement d'apps, qui définit les meilleures pratiques pour la résilience des apps dans les environnements distribués. À l'avenir, lorsque nous mettrons en place nos environnements de production et de test, la configuration des paramètres de notre base de données impliquera la création de fichiers `.env` supplémentaires et le

référencement du fichier approprié dans nos fichiers Docker Compose.

Ouvrez un fichier `.env` :

```
$ nano .env
```

Ajoutez les valeurs suivantes au fichier :

~/rails-docker/.env

```
DATABASE_NAME=rails_development
DATABASE_USER=sammy
DATABASE_PASSWORD=shark
DATABASE_HOST=database
REDIS_HOST=redis
```

En plus de définir le nom de notre base de données, l'utilisateur et le mot de passe, nous avons également défini une valeur pour la `DATABASE_HOST`. La valeur, `database`, fait référence à la `database` du service PostgreSQL que nous allons créer en utilisant Docker Compose. Nous avons également défini un `REDIS_HOST` pour spécifier notre service `Redis`.

Enregistrez et fermez le fichier lorsque vous avez terminé de le modifier.

Pour créer l'utilisateur de la base de données `sammy`, nous pouvons écrire un script `init.sql` que nous pouvons ensuite monter dans le conteneur de la base de données lorsqu'il démarre.

Ouvrez le fichier script :

```
$ nano init.sql
```

Ajoutez le code suivant pour créer un utilisateur `sammy` avec des privilèges administratifs :

~/rails-docker/init.sql

```
CREATE USER sammy;
ALTER USER sammy WITH SUPERUSER;
```

Ce script créera l'utilisateur approprié sur la base de données et lui accordera des privilèges administratifs.

Définissez les autorisations appropriées sur le script :

```
$ chmod +x init.sql
```

Ensuite, nous allons configurer Sidekiq pour qu'il fonctionne avec notre service `Redis` conteneurisé. Nous pouvons ajouter un initialisateur au répertoire `config/initializers`, où Rails recherche les paramètres de configuration une fois que les frameworks et les plugins sont chargés, qui fixe une valeur pour un hôte Redis.

Ouvrez un fichier `sidekiq.rb` pour spécifier ces paramètres :

```
$ nano config/initializers/sidekiq.rb
```

Ajoutez le code suivant au fichier pour spécifier les valeurs d'un `REDIS_HOST` et d'un `REDIS_PORT` :

~/rails-docker/config/initializers/sidekiq.rb

```
Sidekiq.configure_server do |config|
  config.redis = {
    host: ENV['REDIS_HOST'],
    port: ENV['REDIS_PORT'] || '6379'
  }
end

Sidekiq.configure_client do |config|
  config.redis = {
    host: ENV['REDIS_HOST'],
    port: ENV['REDIS_PORT'] || '6379'
  }
end
```

Tout comme les paramètres de configuration de notre base de données, ces paramètres nous donnent la possibilité de définir nos paramètres d'hôte et de port de manière dynamique, ce qui nous permet de substituer les valeurs appropriées au moment de l'exécution sans avoir à modifier le code de l'application lui-même. En plus d'un `REDIS_HOST`, nous avons une valeur par défaut pour `REDIS_PORT` au cas où elle ne serait pas définie ailleurs.

Enregistrez et fermez le fichier lorsque vous avez terminé de le modifier.

Ensuite, pour s'assurer que les données sensibles de notre application ne sont pas copiées dans le contrôle de version, nous pouvons ajouter `.env` au fichier `.gitignore` de notre projet, qui indique à Git quels fichiers ignorer dans notre projet. Ouvrez le fichier pour le modifier :

```
$ nano .gitignore
```

Au bas du fichier, ajoutez une entrée pour `.env` :

~/rails-docker/.gitignore

```
yarn-debug.log*
.yarn-integrity
.env
```

Enregistrez et fermez le fichier lorsque vous avez terminé de le modifier.

Ensuite, nous allons créer un fichier `.dockerignore` pour définir ce qui ne doit pas être copié dans nos conteneurs. Ouvrez le fichier pour le modifier :

```
$ .dockerignore
```

Ajoutez le code suivant au fichier, qui indique à Docker d'ignorer certaines des choses que nous n'avons pas besoin de copier dans nos conteneurs :

~/rails-docker/.dockerignore

```
.DS_Store
.bin
.git
.gitignore
.bundleignore
.bundle
.byebug_history
.rspec
tmp
log
test
config/deploy
public/packs
public/packs-test
node_modules
yarn-error.log
coverage/
```

Ajoutez également `.env` au bas de ce fichier :

~/rails-docker/.dockerignore

```
. . .
yarn-error.log
coverage/
.env
```

Enregistrez et fermez le fichier lorsque vous avez terminé de le modifier.

Dans une dernière étape, nous créerons des données de base afin que notre application dispose de quelques enregistrements au moment de son démarrage.

Ouvrez un fichier pour les données seed dans le répertoire `db` :

```
$ nano db/seeds.rb
```

Ajoutez le code suivant au fichier pour créer quatre requins de démonstration et un exemple de message :

~/rails-docker/db/seeds.rb

```
# Adding demo sharks
sharks = Shark.create([{ name: 'Great White', facts: 'Scary' }, { name: 'Megalodon', fa
Post.create(body: 'These sharks are misunderstood', shark: sharks.first)
```

Ces données de base permettront de créer quatre requins et un message qui est associé au

premier requin.

Enregistrez et fermez le fichier lorsque vous avez terminé de le modifier.

Votre application étant configurée pour fonctionner avec PostgreSQL et vos variables d'environnement créées, vous êtes prêt à écrire votre Dockerfile d'application.

Étape 3 - Rédigez les scripts du Dockerfile et d'Entrypoint

Votre Dockerfile précise ce qui sera inclus dans votre conteneur d'application lors de sa création. L'utilisation d'un Dockerfile vous permet de définir votre environnement de conteneur et d'éviter les divergences avec les dépendances ou les versions d'exécution.

En suivant ces directives sur la construction de conteneurs optimisés, nous allons rendre notre image aussi efficace que possible en utilisant une base Alpine et en essayant de minimiser nos couches d'image en général.

Ouvrez un Dockerfile dans votre répertoire actuel :

```
$ nano Dockerfile
```

Les images de Docker sont créées à l'aide d'une succession d'images en couches qui se construisent les unes sur les autres. Notre première étape sera d'ajouter *l'image de base* pour notre application, qui constituera le point de départ de la construction de l'application.

Ajoutez le code suivant au fichier pour ajouter l'image alpine Ruby comme base :

~/rails-docker/Dockerfile

```
FROM ruby:2.5.1-alpine
```

L'image `alpine` est dérivée du projet Alpine Linux, et nous aidera à réduire la taille de notre image. Pour savoir si l'image `alpine` est ou non le bon choix pour votre projet, veuillez consulter la discussion complète dans la section **Variantes d'images** de la page d'images du Docker Hub Ruby.

Quelques facteurs à prendre en compte lors de l'utilisation de `alpine` dans le développement :

- En limitant la taille des images, vous réduisez le temps de chargement des pages et des ressources, en particulier si vous limitez également les volumes au minimum. Cela permet de conserver la rapidité et la proximité de votre expérience utilisateur en matière de développement aussi proches que possible de ce qu'elles seraient si vous travailliez localement dans un environnement non conteneurisé.
- Le fait d'avoir une parité entre les images de développement et celles de production facilite la réussite des déploiements. Comme les équipes choisissent souvent d'utiliser des images Alpine dans la production pour des raisons de rapidité, le développement avec une base Alpine permet de compenser les problèmes lors du passage à la production.

Ensuite, définissez une variable d'environnement pour spécifier la version Bundler :

~/rails-docker/Dockerfile

```
. . .
ENV BUNDLER_VERSION=2.0.2
```

C'est l'une des mesures que nous prendrons pour éviter les conflits de versions entre la version par défaut de `bundler` disponible dans notre environnement et notre code d'application, qui nécessite Bundler 2.0.2.

Ensuite, ajoutez au Dockerfile les paquets dont vous avez besoin pour travailler avec l'application :

~/rails-docker/Dockerfile

```
. . .
RUN apk add --update --no-cache \
      binutils-gold \
      build-base \
      curl \
      file \
      g++ \
      gcc \
      git \
      less \
      libstdc++ \
      libffi-dev \
      libc-dev \
      linux-headers \
      libxml2-dev \
      libxslt-dev \
      libgcrypt-dev \
      make \
      netcat-openbsd \
      nodejs \
      openssl \
      pkgconfig \
      postgresql-dev \
      python \
      tzdata \
      yarn
```

Ces paquets comprennent, entre autres, des `nodejs` et des `yarn`. Étant donné que notre application sert des actifs avec webpack, nous devons inclure Node.js et Yarn pour que l'application fonctionne comme prévu.

Gardez à l'esprit que l'image `alpine` est extrêmement minimale : les packages listés ici ne sont pas exhaustifs de ce que vous pourriez vouloir ou ce dont vous pourriez avoir besoin en développement lorsque vous conteneurisez votre propre application.

Ensuite, installez la version `bundler` appropriée :

~/rails-docker/Dockerfile

```
. . .
RUN gem install bundler -v 2.0.2
```

Cette étape garantira la parité entre notre environnement conteneurisé et les spécifications du fichier `Gemfile.lock` de ce projet.

Définissez maintenant le répertoire de travail pour l'application sur le conteneur :

~/rails-docker/Dockerfile

```
. . .
WORKDIR /app
```

Copiez par-dessus votre `Gemfile` et `Gemfile.lock` :

~/rails-docker/Dockerfile

```
. . .
COPY Gemfile Gemfile.lock ./
```

La copie de ces fichiers en tant qu'étape indépendante, suivie `d'une installation groupée`, signifie que les gem du projet n'ont pas besoin d'être reconstruits chaque fois que vous apportez des modifications à votre code d'application. Cela fonctionnera en conjonction avec le volume de gem que nous inclurons dans notre fichier Compose, qui montera les gem dans votre conteneur d'application dans les cas où le service est recréé mais où les gem du projet restent les mêmes.

Ensuite, définissez les options de configuration pour la construction du gem `nokogiri` :

~/rails-docker/Dockerfile

```
. . .
RUN bundle config build.nokogiri --use-system-libraries
. . .
```

Cette étape construit `nokigiri` avec les versions des bibliothèques `libxml2` et `libxslt` que nous avons ajoutées au conteneur d'application dans l'étape `RUN apk add...` ci-dessus.

Ensuite, installez les gem du projet :

~/rails-docker/Dockerfile

```
. . .
RUN bundle check || bundle install
```

Cette instruction permet de vérifier que les gem ne sont pas déjà installés avant de les installer.

Ensuite, nous allons répéter la procédure que nous avons utilisée pour les gem avec nos paquets et dépendances JavaScript. Nous commencerons par copier les métadonnées du paquet, puis nous installerons les dépendances, et enfin nous copierons le code de l'application dans l'image du conteneur.

Pour commencer avec la section Javascript de notre Dockerfile, copiez `package.json` et `yarn.lock` de votre répertoire de projet actuel sur l'hôte vers le conteneur :

~/rails-docker/Dockerfile

```
. . .
COPY package.json yarn.lock ./
```

Ensuite, installez les paquets requis avec l'installation de `yarn` :

~/rails-docker/Dockerfile

```
. . .
RUN yarn install --check-files
```

Cette instruction comprend un drapeau `--check-files` avec la commande `yarn`, une fonction qui permet de s'assurer que les fichiers précédemment installés n'ont pas été supprimés. Comme dans le cas de nos gem, nous allons gérer la persistance des paquets dans le répertoire `node_modules` avec un volume lorsque nous écrivons notre fichier Compose.

Enfin, copiez le reste du code de l'application et lancez l'application avec un script entrypoint :

~/rails-docker/Dockerfile

```
. . .
COPY . ./

ENTRYPOINT ["./entrypoints/docker-entrypoint.sh"]
```

L'utilisation d'un script entrypoint nous permet de faire fonctionner le conteneur comme un exécutable.

Le Dockerfile final ressemblera à ceci :

~/rails-docker/Dockerfile

```
FROM ruby:2.5.1-alpine

ENV BUNDLER_VERSION=2.0.2

RUN apk add --update --no-cache \
      binutils-gold \
      build-base \
      curl \
      file \
      g++ \
      gcc \
      git \
      less \
      libstdc++ \
      libffi-dev \
      libc-dev \
      linux-headers \
      libxml2-dev \
      libxslt-dev \
```

```
      libgcrypt-dev \
      make \
      netcat-openbsd \
      nodejs \
      openssl \
      pkgconfig \
      postgresql-dev \
      python \
      tzdata \
      yarn

RUN gem install bundler -v 2.0.2

WORKDIR /app

COPY Gemfile Gemfile.lock ./

RUN bundle config build.nokogiri --use-system-libraries

RUN bundle check || bundle install

COPY package.json yarn.lock ./

RUN yarn install --check-files

COPY . ./

ENTRYPOINT ["./entrypoints/docker-entrypoint.sh"]
```

Enregistrez et fermez le fichier lorsque vous avez terminé de le modifier.

Ensuite, créez un répertoire appelé `entrypoints` pour les scripts de points d'entrée :

```
$ mkdir entrypoints
```

Ce répertoire comprendra le script de notre point d'entrée principal et un script pour notre service Sidekiq.

Ouvrez le fichier pour le script d'application entrypoint :

```
$ nano entrypoints/docker-entrypoint.sh
```

Ajoutez le code suivant au fichier :

rails-docker/entrypoints/docker-entrypoint.sh

```
#!/bin/sh

set -e

if [ -f tmp/pids/server.pid ]; then
  rm tmp/pids/server.pid
fi

bundle exec rails s -b 0.0.0.0
```

La première ligne importante est `set -e`, qui indique au shell `/bin/sh` qui exécute le script d'échouer rapidement s'il y a des problèmes plus tard dans le script. Ensuite, le script vérifie que `tmp/pids/server.pid` n'est pas présent pour s'assurer qu'il n'y aura pas de conflits de serveur lorsque nous lancerons l'application. Enfin, le script démarre le serveur Rails avec la commande `bundle exec rails`. Nous utilisons l'option `-b` avec cette commande pour lier le serveur à toutes les adresses IP plutôt qu'à l'adresse par défaut, `localhost`. Cette invocation fait que le serveur Rails achemine les requêtes entrantes vers l'IP du conteneur plutôt que vers le `localhost` par défaut.

Enregistrez et fermez le fichier lorsque vous avez terminé de le modifier.

Rendez le script exécutable :

```
$ chmod +x entrypoints/docker-entrypoint.sh
```

Ensuite, nous allons créer un script pour lancer notre service `Sidekiq`, qui traitera nos emplois Sidekiq. Pour plus d'informations sur la façon dont cette application utilise Sidekiq, veuillez consulter la section Comment ajouter Sidekiq et Redis à une application Ruby on Rails.

Ouvrez un fichier pour le script Sidekiq entrypoint :

```
$ nano entrypoints/sidekiq-entrypoint.sh
```

Ajoutez le code suivant au fichier pour démarrer Sidekiq :

~/rails-docker/entrypoints/sidekiq-entrypoint.sh

```
#!/bin/sh

set -e

if [ -f tmp/pids/server.pid ]; then
  rm tmp/pids/server.pid
fi

bundle exec sidekiq
```

Ce script lance Sidekiq dans le contexte de notre paquet d'application.

Enregistrez et fermez le fichier lorsque vous avez terminé de le modifier. Rendez-le exécutable :

```
$ chmod +x entrypoints/sidekiq-entrypoint.sh
```

Avec vos scripts d'entrée et votre Dockerfile en place, vous êtes prêt à définir vos services dans votre fichier Compose.

Étape 4 - Définissez les services avec Docker Compose

Grâce à Docker Compose, nous serons en mesure de faire fonctionner les multiples conteneurs nécessaires à notre installation. Nous définirons nos services *Compose* dans notre fichier `docker-compose.yml` principal. Un service dans Compose est un conteneur en cours d'exécution, et les définitions de service - que vous inclurez dans votre fichier `docker-compose.yml` - contiennent des informations sur la façon dont chaque image de conteneur sera exécutée. L'outil Compose vous permet de définir plusieurs services pour construire des apps multi-conteneurs.

L'installation de notre application comprendra les services suivants :

- L'application elle-même
- La base de données PostgreSQL :
- Redis
- Sidekiq

Nous inclurons également un bind mount dans le cadre de notre installation, de sorte que toute modification du code que nous apporterons pendant le développement sera immédiatement synchronisée avec les conteneurs qui doivent avoir accès à ce code.

Notez que nous ne définissons *pas* un service de `test`, puisque le test n'entre pas dans le cadre de ce tutoriel et de cette série, mais vous pourriez le faire en suivant le précédent que nous utilisons ici pour le service `sidekiq`.

Ouvrez le fichier `docker-compose.yml` :

```
$ nano docker-compose.yml
```

Premièrement, ajoutez la définition du service de l'application :

~/rails-docker/docker-compose.yml

```
version: '3.4'

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      - database
      - redis
    ports:
      - "3000:3000"
    volumes:
      - .:/app
      - gem_cache:/usr/local/bundle/gems
      - node_modules:/app/node_modules
    env_file: .env
    environment:
      RAILS_ENV: development
```

La définition du service `app` comprend les options suivantes :

- `build` : ceci définit les options de configuration, y compris le `contexte` et le fichier `docker`, qui seront appliqués lors de la construction de l'image de l'application par Compose. Si vous souhaitez utiliser une image existante provenant d'un registre comme Docker Hub, vous pouvez utiliser l'instruction `image` à la place, avec des informations sur votre nom d'utilisateur, le dépôt et l'étiquette de l'image.
- `context` : définit le contexte de construction de l'image - dans ce cas, le répertoire du projet en cours.
- `dockerfile` : spécifie le `Dockerfile` dans le répertoire de votre projet actuel comme le fichier que Compose utilisera pour construire l'image de l'application.
- `depends_on` : permet de mettre en place la `database` et les conteneurs `Redis` en premier, afin qu'ils soient opérationnels avant l'`app`.
- `ports` : recense le port `3000` sur l'hôte et le port `3000` sur le conteneur.
- `volumes` : nous incluons ici deux types de points de montage :
  - Le premier est un point de montage bind qui monte notre code d'application sur l'hôte vers le répertoire `/app` du conteneur. Cela facilitera un développement rapide, puisque toute modification que vous apportez à votre code hôte sera immédiatement versée dans le conteneur.
  - Le deuxième est un volume nommé `gem_cache`. Lorsque l'instruction `d'installation du paquet` se déroule dans le conteneur, elle permet d'installer les gem du projet. L'ajout de ce volume signifie que si vous recréez le conteneur, les gem seront montées sur le nouveau conteneur. Ce montage suppose qu'il n'y a eu aucune modification du projet, donc si *vous* apportez des modifications à vos gem de projet en développement, vous devrez vous rappeler de supprimer ce volume avant de recréer votre service d'application.
  - Le troisième volume est un volume nommé pour le répertoire `node_modules`. Plutôt que d'avoir des `node_modules` montés sur l'hôte, ce qui peut entraîner des divergences de paquets et des conflits de permissions dans le développement, ce volume assurera que les paquets de ce répertoire sont persistants et reflètent l'état actuel du projet. Encore une fois, si vous modifiez les dépendances Node du projet, vous devrez supprimer et recréer ce volume.
- `env_file` : indique à Compose que nous souhaitons ajouter des variables d'environnement à partir d'un fichier appelé `.env` situé dans le contexte de construction.
- `environment` : cette option nous permet de définir une variable d'environnement non sensible, en transmettant au conteneur des informations sur l'environnement Rails.

Ensuite, sous la définition du service `app`, ajoutez le code suivant pour définir votre service `database` :

~/rails-docker/docker-compose.yml

```
. . .
  database:
    image: postgres:12.1
    volumes:
      - db_data:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
```

Contrairement au service `app`, le service `database` tire une image `postgres` directement du Docker Hub. Notez que nous épinglons également la version ici, plutôt que de la mettre à `jour` ou de ne pas la spécifier (qui est `la plus récente` par défaut). Ainsi, nous pouvons nous assurer que cette configuration fonctionne avec les versions spécifiées ici et éviter des surprises inattendues en cassant les changements de code de l'image.

Nous incluons également un volume `db_data` ici, qui prolongera nos données d'application entre les démarrages de conteneurs. De plus, nous avons monté notre script de démarrage `init.sql` dans le répertoire approprié, `docker-entrypoint-initdb.d/` sur le conteneur, afin de créer notre utilisateur de base de données `sammy`. Une fois que le point d'entrée de l'image aura créé l'utilisateur et la base de données par défaut de `postgres`, il exécutera tous les scripts trouvés dans le répertoire `docker-entrypoint-initdb.d/`, que vous pourrez utiliser pour les tâches d'initialisation nécessaires. Pour plus de détails, consultez la section **Scripts d'initialisation** de la documentation sur les images PostgreSQL

Ensuite, ajoutez la définition du service `Redis` :

~/rails-docker/docker-compose.yml

```
. . .
  redis:
    image: redis:5.0.7
```

Comme le service de `database`, le service `Redis` utilise une image du Docker Hub. Dans ce cas, nous ne persistons pas dans le cache d'emploi de Sidekiq.

Enfin, ajoutez la définition du service `sidekiq` :

~/rails-docker/docker-compose.yml

```
. . .
  sidekiq:
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      - app
      - database
      - redis
    volumes:
      - .:/app
      - gem_cache:/usr/local/bundle/gems
      - node_modules:/app/node_modules
    env_file: .env
    environment:
      RAILS_ENV: development
    entrypoint: ./entrypoints/sidekiq-entrypoint.sh
```

Notre service `sidekiq` ressemble à notre service `app` à quelques égards : il utilise le même contexte et la même image de construction, les mêmes variables d'environnement et les mêmes volumes. Cependant, il dépend des services `app`, de `redis` et de `database`, et sera donc le dernier à démarrer. En outre, il utilise un `point d'entrée` qui remplacera le point d'entrée défini

dans le Dockerfile. Ce `point d'entrée` donne accès à `entrypoints/sidekiq-entrypoint.sh`, qui comprend la commande appropriée pour lancer le service `sidekiq`.

Pour terminer, ajoutez les définitions des volumes sous la définition du service `sidekiq` :

~/rails-docker/docker-compose.yml

```
. . .
volumes:
  gem_cache:
  db_data:
  node_modules:
```

Notre clé de volumes de niveau supérieur définit les volumes `gem_cache`, `db_data` et `node_modules`. Lorsque Docker crée des volumes, le contenu du volume est stocké dans une partie du système de fichiers hôte, `/var/lib/docker/volumes/`, qui est gérée par Docker. Le contenu de chaque volume est stocké dans un répertoire sous `/var/lib/docker/volumes/` et se monte sur tout conteneur qui utilise le volume. De cette façon, les données d'information sur les requins que nos utilisateurs créeront persisteront dans le volume `db_data` même si nous supprimons et recréons le service de `database`.

Le fichier terminé ressemblera à ceci :

~/rails-docker/docker-compose.yml

```
version: '3.4'

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      - database
      - redis
    ports:
      - "3000:3000"
    volumes:
      - .:/app
      - gem_cache:/usr/local/bundle/gems
      - node_modules:/app/node_modules
    env_file: .env
    environment:
      RAILS_ENV: development

  database:
    image: postgres:12.1
    volumes:
      - db_data:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql

  redis:
    image: redis:5.0.7

  sidekiq:
    build:
      context: .
      dockerfile: Dockerfile
```

```
    depends_on:
      - app
      - database
      - redis
    volumes:
      - .:/app
      - gem_cache:/usr/local/bundle/gems
      - node_modules:/app/node_modules
    env_file: .env
    environment:
      RAILS_ENV: development
    entrypoint: ./entrypoints/sidekiq-entrypoint.sh

volumes:
  gem_cache:
  db_data:
  node_modules:
```

Enregistrez et fermez le fichier lorsque vous avez terminé de le modifier.

Une fois vos définitions de service rédigées, vous êtes prêt à lancer l'application.

Étape 5 - Testez l'application

Une fois votre fichier `docker-compose.yml` en place, vous pouvez créer vos services avec la commande `docker-compose up` et ensemencer votre base de données. Vous pouvez également tester la persistance de vos données en arrêtant et en retirant vos conteneurs lorsque `docker-compose` est fermé et en les recréant.

Tout d'abord, construisez les images des conteneurs et créez les services en lançant le `docker-compose` avec le drapeau `-d`, qui fera fonctionner les conteneurs en arrière-plan :

```
$ docker-compose up -d
```

Vous verrez que vos services ont été créés :

```
Output
Creating rails-docker_database_1 ... done
Creating rails-docker_redis_1    ... done
Creating rails-docker_app_1      ... done
Creating rails-docker_sidekiq_1  ... done
```

Vous pouvez également obtenir des informations plus détaillées sur les processus de démarrage en affichant le journal de sortie des services :

```
$ docker-compose logs
```

Vous verrez quelque chose comme ça si tout a commencé correctement :

```
Output
sidekiq_1   | 2019-12-19T15:05:26.365Z pid=6 tid=grk7r6xly INFO: Booting Sidekiq 6.0.3
```

```
sidekiq_1   | 2019-12-19T15:05:31.097Z pid=6 tid=grk7r6xly INFO: Running in ruby 2.5.1p
sidekiq_1   | 2019-12-19T15:05:31.097Z pid=6 tid=grk7r6xly INFO: See LICENSE and the LG
sidekiq_1   | 2019-12-19T15:05:31.097Z pid=6 tid=grk7r6xly INFO: Upgrade to Sidekiq Pro
app_1       | => Booting Puma
app_1       | => Rails 5.2.3 application starting in development
app_1       | => Run `rails server -h` for more startup options
app_1       | Puma starting in single mode...
app_1       | * Version 3.12.1 (ruby 2.5.1-p57), codename: Llamas in Pajamas
app_1       | * Min threads: 5, max threads: 5
app_1       | * Environment: development
app_1       | * Listening on tcp://0.0.0.0:3000
app_1       | Use Ctrl-C to stop
. . .
database_1  | PostgreSQL init process complete; ready for start up.
database_1  |
database_1  | 2019-12-19 15:05:20.160 UTC [1] LOG:  starting PostgreSQL 12.1 (Debian 12
database_1  | 2019-12-19 15:05:20.160 UTC [1] LOG:  listening on IPv4 address "0.0.0.0"
database_1  | 2019-12-19 15:05:20.160 UTC [1] LOG:  listening on IPv6 address "::", por
database_1  | 2019-12-19 15:05:20.163 UTC [1] LOG:  listening on Unix socket "/var/run/
database_1  | 2019-12-19 15:05:20.182 UTC [63] LOG:  database system was shut down at 2
database_1  | 2019-12-19 15:05:20.187 UTC [1] LOG:  database system is ready to accept
. . .
redis_1     | 1:M 19 Dec 2019 15:05:18.822 * Ready to accept connections
```

Vous pouvez également vérifier l'état de vos conteneurs avec le logiciel `docker-compose ps` :

```
$ docker-compose ps
```

Vous verrez une sortie indiquant que vos conteneurs fonctionnent :

```
Output
         Name                        Command               State           Ports
-------------------------------------------------------------------------------------
rails-docker_app_1        ./entrypoints/docker-resta ...   Up      0.0.0.0:3000->3000/t
rails-docker_database_1   docker-entrypoint.sh postgres    Up      5432/tcp
rails-docker_redis_1      docker-entrypoint.sh redis ...   Up      6379/tcp
rails-docker_sidekiq_1    ./entrypoints/sidekiq-entr ...   Up
```

Ensuite, créez et ensemencez votre base de données et exécutez des migrations sur celle-ci avec la commande `docker-compose exec` suivante :

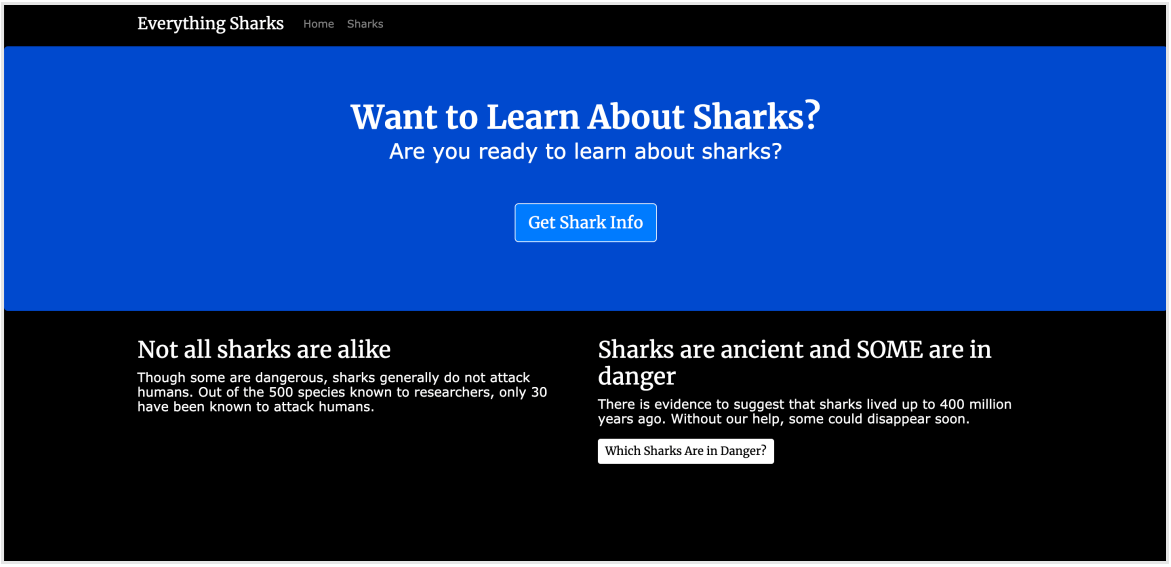
```
$ docker-compose exec app bundle exec rake db:setup db:migrate
```

La commande `docker-compose exec` vous permet d'exécuter des commandes dans vos services ; nous l'utilisons ici pour exécuter `rake db:setup` et `db:migrate` dans le cadre de notre offre groupée d'apps pour créer et amorcer la base de données et exécuter les migrations. Comme vous travaillez dans le développement, `docker-compose exec` vous sera utile lorsque vous voudrez exécuter des migrations sur votre base de données de développement.

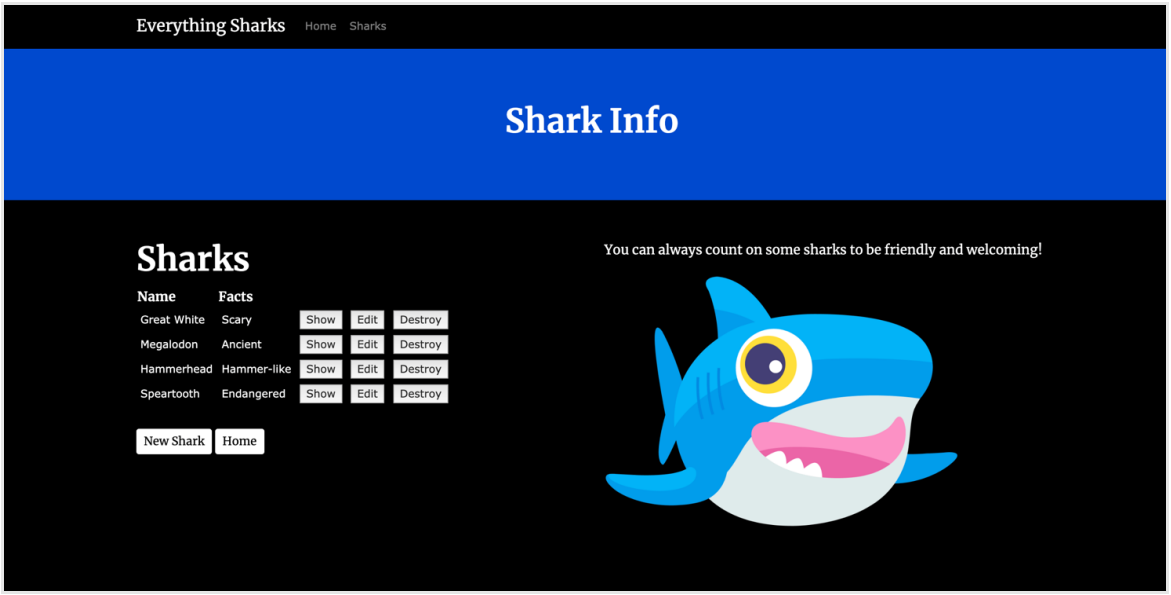
Vous verrez la sortie suivante après avoir exécuté cette commande :

```
Output
Created database 'rails_development'
Database 'rails_development' already exists
-- enable_extension("plpgsql")
   -> 0.0140s
-- create_table("endangereds", {:force=>:cascade})
   -> 0.0097s
-- create_table("posts", {:force=>:cascade})
   -> 0.0108s
-- create_table("sharks", {:force=>:cascade})
   -> 0.0050s
-- enable_extension("plpgsql")
   -> 0.0173s
-- create_table("endangereds", {:force=>:cascade})
   -> 0.0088s
-- create_table("posts", {:force=>:cascade})
   -> 0.0128s
-- create_table("sharks", {:force=>:cascade})
   -> 0.0072s
```

Lorsque vos services fonctionnent, vous pouvez visiter `localhost:3000` ou `http://your_server_ip:3000` dans le navigateur. Vous verrez une page d'accueil qui ressemble à ceci :



Nous pouvons maintenant tester la persistance des données. Créez un nouveau requin en cliquant sur le bouton **Get Shark Info**, qui vous amènera à la route des `requins / index` :

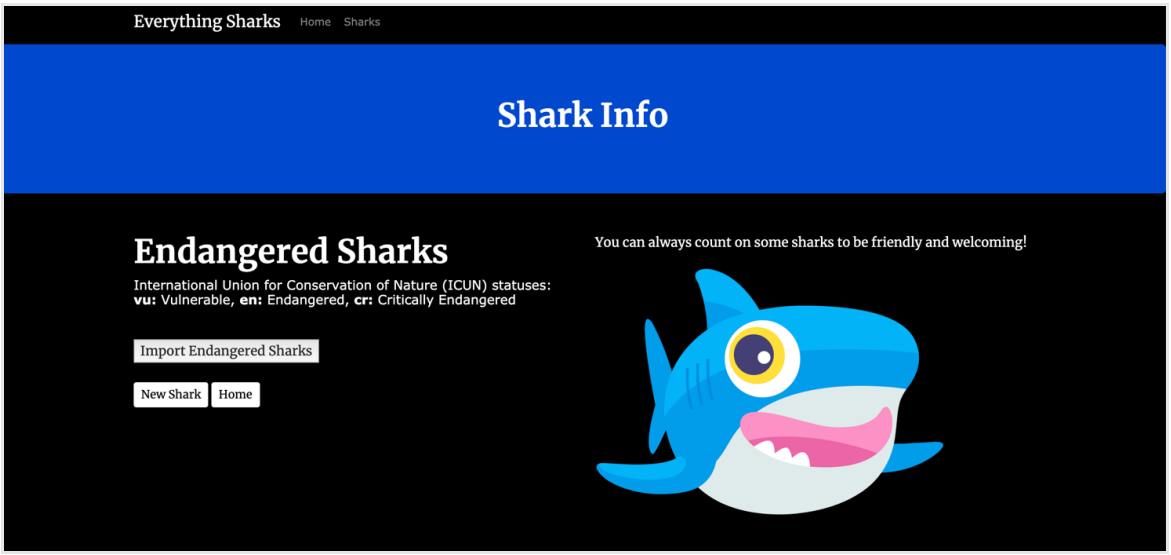


Pour vérifier que l'application fonctionne, nous pouvons y ajouter quelques informations de démonstration. Cliquez sur Nouveau Requin. Un nom d'utilisateur (**sammy**) et un mot de passe (shark) vous seront demandés, grâce aux **paramètres d'authentification du projet**.

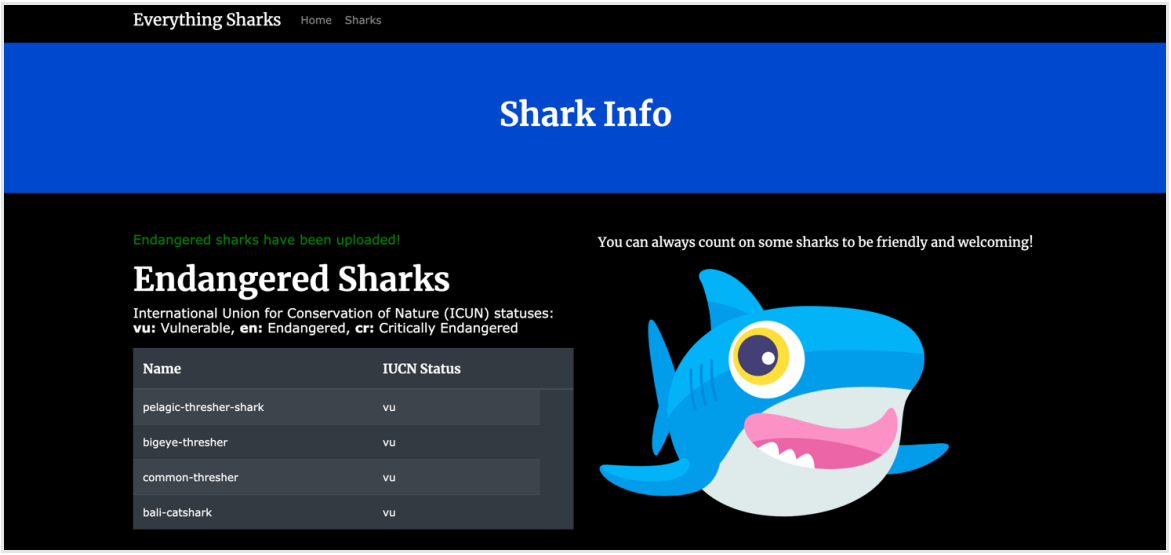
Dans la page **Nouveau Requin**, entrez “Mako” dans le champ **Nom** et “Rapide” dans le champ **Facts**.

Cliquez sur le bouton **Créer un Requin** pour créer le requin. Une fois que vous avez créé le requin, cliquez sur **Accueil** dans la barre de navigation du site pour revenir à la page d'accueil principale de l'application. Nous pouvons maintenant tester le fonctionnement de Sidekiq.

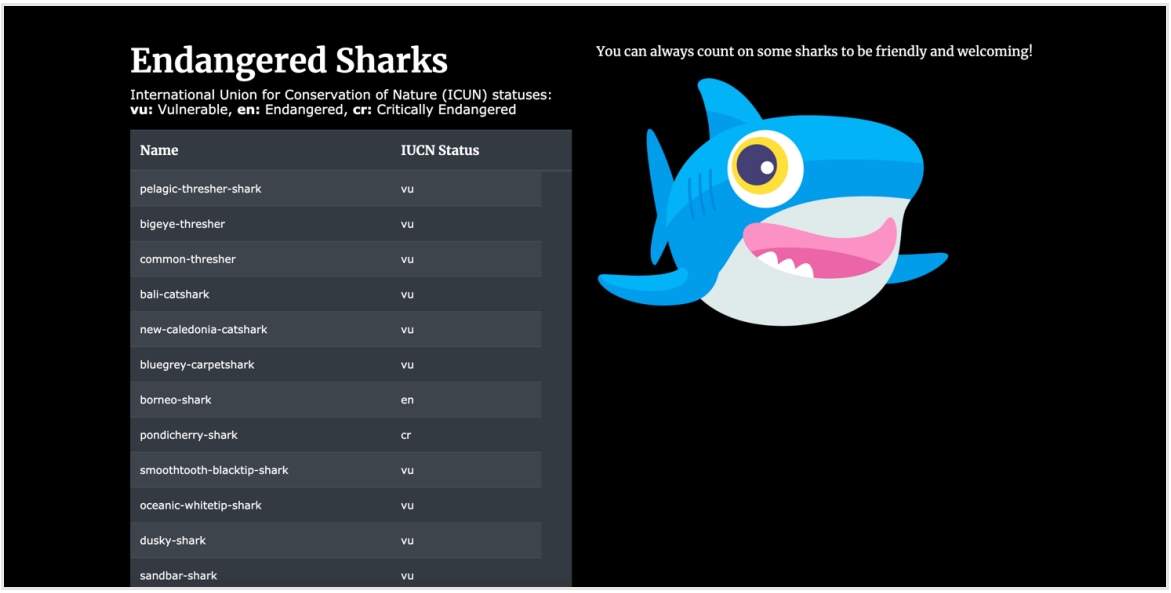
Cliquez sur le bouton **Quels sont les requins en danger ?** . Comme vous n'avez pas téléchargé de requins en voie de disparition, cela vous amènera à la vue `index` des `espèces en voie de disparition` :



Cliquez sur **Importer des requins en voie de disparition** pour importer les requins. Vous verrez un message de statut vous indiquant que les requins ont été importés :

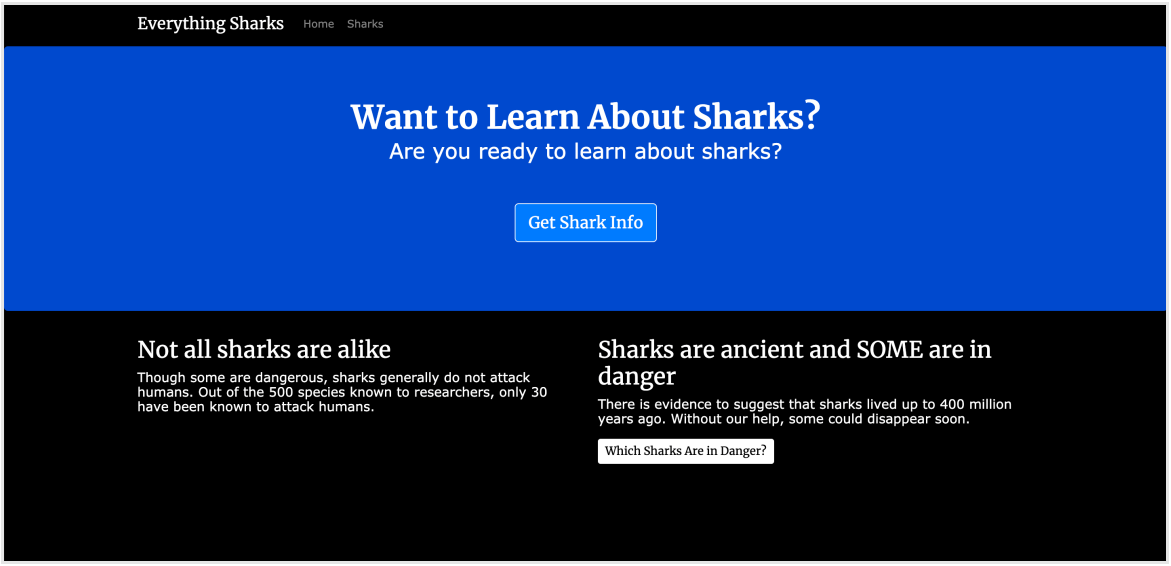


Vous verrez également le début de l'importation. Rafraîchissez votre page pour voir le tableau complet :



Grâce à Sidekiq, notre important téléchargement par lots de requins en voie de disparition a réussi sans verrouiller le navigateur ni interférer avec les autres fonctionnalités de l'application.

Cliquez sur le bouton **Accueil** en bas de la page, ce qui vous ramènera à la page principale de l'application :



De là, cliquez à nouveau sur **Quels sont les requins en danger ?** . Vous verrez à nouveau les requins téléchargés.

Maintenant que nous savons que notre application fonctionne correctement, nous pouvons tester la persistance de nos données.

De retour à votre terminal, tapez la commande suivante pour arrêter et retirer vos conteneurs :

```
$ docker-compose down
```

Notez que nous n'incluons pas l'option `--volumes` ; par conséquent, notre volume `db_data` n'est pas supprimé.

La sortie suivante confirme que vos conteneurs et votre réseau ont été retirés :

```
Output
Stopping rails-docker_sidekiq_1  ... done
Stopping rails-docker_app_1      ... done
Stopping rails-docker_database_1 ... done
Stopping rails-docker_redis_1    ... done
Removing rails-docker_sidekiq_1  ... done
Removing rails-docker_app_1      ... done
Removing rails-docker_database_1 ... done
Removing rails-docker_redis_1    ... done
Removing network rails-docker_default
```

Recréez les conteneurs :

```
$ docker-compose up -d
```

Ouvrez la console Rails sur le conteneur de l'`app`lication avec la console `docker-compose exec` et `bundle exec rails` :

```
$ docker-compose exec app bundle exec rails console
```

Lorsque vous êtes invité à le faire, inspectez le `dernier` enregistrement de Requin dans la base de données :

```
irb(main):001:0> Shark.last.inspect
```

Vous verrez le dossier que vous venez de créer :

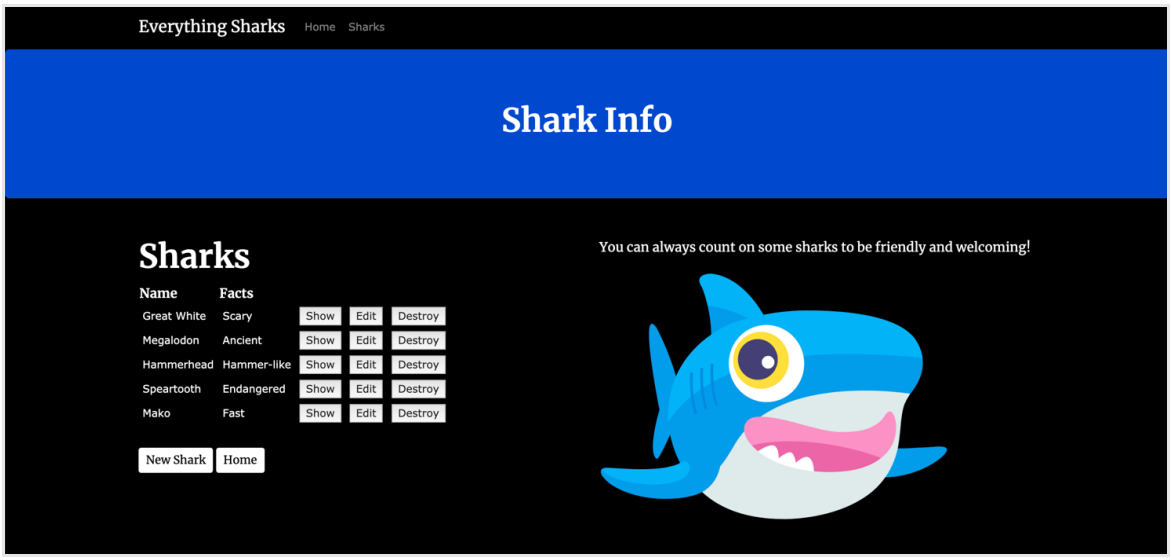
```
IRB session
  Shark Load (1.0ms)  SELECT  "sharks".* FROM "sharks" ORDER BY "sharks"."id" DESC LIMI
=> "#<Shark id: 5, name: \"Mako\", facts: \"Fast\", created_at: \"2019-12-20 14:03:28\"
```

Vous pouvez ensuite vérifier que vos requins `en voie de disparition` ont été conservés avec la commande suivante :

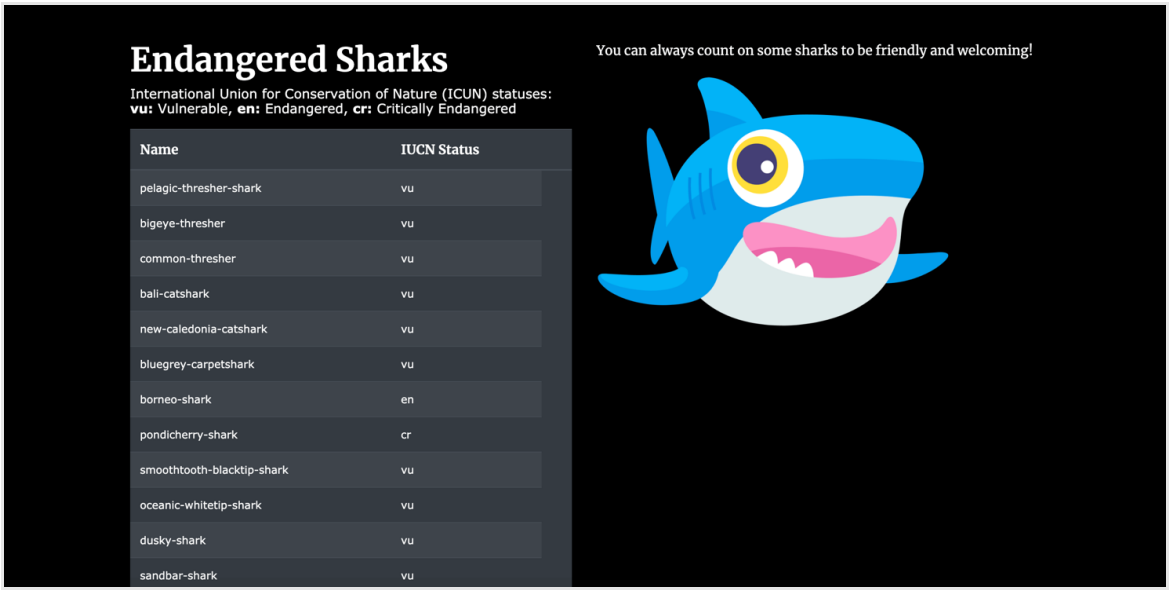
```
irb(main):001:0> Endangered.all.count
```

```
IRB session
   (0.8ms)  SELECT COUNT(*) FROM "endangereds"
=> 73
```

Votre volume `db_data` a été monté avec succès sur le service `de base de données` recréée, permettant à votre service `applicatif` d'accéder aux données sauvegardées. Si vous accédez directement à la page `d'index` des `requins` en visitant `localhost:3000/sharks` ou `http://your_server_ip:3000/sharks`, vous verrez également cet enregistrement s'afficher :



Vos requins en voie de disparition seront également sur le site `localhost:3000/endangered/data` ou `http://your_server_ip:3000/endangered/data view` :



Votre application fonctionne maintenant sur des conteneurs Docker avec la persistance des données et la synchronisation du code activées. Vous pouvez tester les changements de code local sur votre hôte, qui sera synchronisé avec votre conteneur grâce au point de montage bind que nous avons défini dans le cadre du service de `l'application`.

Conclusion

En suivant ce tutoriel, vous avez créé une configuration de développement pour votre application Rails en utilisant des conteneurs Docker. Vous avez rendu votre projet plus modulaire et portable en extrayant des informations sensibles et en découplant l'état de votre application depuis votre code. Vous avez également configuré un fichier `docker-compose.yml` standard que vous pouvez réviser en fonction de l'évolution de vos besoins de développement et de vos exigences.

Au fur et à mesure de votre développement, vous souhaiterez peut-être en savoir plus sur la conception d'apps pour les flux de travail conteneurisés et Cloud Native. Pour plus d'informations sur ces sujets, veuillez consulter les sections Architecture d'apps pour Kubernetes et Modernisation d'apps pour Kubernetes. Ou, si vous souhaitez investir dans une séquence d'apprentissage Kubernetes, veuillez consulter notre programme Kubernetes pour Développeurs Full-Stack.

Pour en savoir plus sur le code d'application lui-même, veuillez consulter les autres tutoriels de cette série :

- Comment construire une application Ruby on Rails
- Comment créer des ressources imbriquées pour une application Ruby on Rails
- Comment ajouter Stimulus à une application Ruby on Rails
- Comment ajouter Bootstrap à une application Ruby on Rails
- Comment ajouter Sidekiq et Redis à une application Ruby on Rail

Qu'avez-vous pensé de cette traduction? 👍 👎

Was this helpful? Yes No

Report an issue

About the authors

Kathleen Juell

Developer @digitalocean/community

Still looking for an answer?

Ask a question

Search for more help

Comments

0 Comments

Leave a comment...

Sign In to Comment

GET OUR BIWEEKLY NEWSLETTER

Sign up for Infrastructure as a Newsletter.

HOLLIE'S HUB FOR GOOD

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

BECOME A CONTRIBUTOR

You get paid; we donate to tech nonprofits.

Featured on Community Kubernetes Course Learn Python 3 Machine Learning in Python Getting started with Go Intro to Kubernetes

DigitalOcean Products Virtual Machines Managed Databases Managed Kubernetes Block Storage Object Storage Marketplace VPC Load Balancers