

Déployer ISIS Capitalist

Docker

Principe

Docker (www.docker.com) est une technologie qui permet de distribuer et de déployer des applications en les plaçant dans des conteneurs qui se comportent comme des machines virtuelles. A la différence de ces dernières, les conteneurs docker s'exécutent nativement sur les systèmes et sont donc très efficaces en ressources utilisées. Ils sont auto-suffisants ce qui signifie qu'ils contiennent tout ce dont l'application a besoin pour fonctionner, sans faire d'hypothèse sur la configuration de la machine qui les héberge.

Docker permet donc de rassembler dans un conteneur l'ensemble des fichiers et services dont votre application a besoin pour fonctionner. C'est l'étape de **build**.

Ce conteneur est ensuite placé dans un dépôt qui le rend accessible sur internet. C'est l'étape de **push**.

Pour déployer l'application, on télécharge le conteneur (**pull**) et on l'exécute (**run**).

Pour réaliser ces opérations, on utilise le *runtime* `docker` qui est un outil en ligne de commande. Cet outil est conçu pour s'exécuter sur des machines de type Unix mais il existe des outils qui nous permettent de l'utiliser aussi sur des machines de type Windows ou MacOS :

- Pour Windows et MacOS le plus simple est de télécharger et d'installer Docker Desktop:

| <https://www.docker.com/products/docker-desktop/>

- Pour Unix, selon la distribution les procédures sont indiquées ici :

| <https://docs.docker.com/engine/install/>

Le Docker Hub

Il existe un dépôt officiel recensant des milliers d'applications au format docker (cad sous forme de conteneurs) que vous pouvez *pull* et exécuter :

<https://hub.docker.com/search?type=image>

Le terme application est ici utilisé dans un sens très large. Un conteneur docker pouvant contenir n'importe quel ensemble de composants et d'outils logiciels, certains conteneurs contiennent des systèmes d'exploitation complets. D'autres contiennent des systèmes d'exploitation avec certains logiciels préinstallés. Par exemple le conteneur `node:lts-buster-slim` représente une ubuntu 18 (buster) avec node.js préinstallé.

Les conteneurs docker disponibles sur le *Docker Hub* ne servent pas qu'à installer des applications toutes prêtes. Ils servent aussi de base pour la création de vos propres conteneurs. En effet, comme nous allons le voir dans la section suivante, on construit toujours des nouveaux conteneurs en ajoutant des composants à des conteneurs existants.

Phase de build pour le backend ISIS Capitalist

L'étape de **build** permet de construire le conteneur correspondant à son application. Elle aboutie à la création d'une image qui contient tous les composants logiciels nécessaires à son fonctionnement.

Ces composants sont décrits dans un fichier au nom standardisé : le **Dockerfile**. Ce fichier est un script dont le déroulement construit une machine virtuelle en y installant les composants requis. Le **Dockerfile** commence toujours par une ligne `FROM` qui désigne le conteneur qui sert de base à l'installation. En général ce conteneur de départ est un des nombreux conteneurs disponibles sur le *Docker Hub* mais on peut aussi construire des conteneurs qui s'appuie sur des conteneurs privés.

Le projet ISIS Capitalist est composé de deux parties, frontend et backend, nous allons construire un conteneur pour chacune de ses parties.

Commençons par la partie backend. A la racine de votre projet créez un fichier **Dockerfile** doté du contenu suivant :

```
FROM node:16

# Create app directory
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are
# copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install

# Bundle app source
COPY . .

EXPOSE 4000
CMD [ "node", "index.js" ]
```

Analysons ce contenu. La première ligne `FROM` indique que nous partons du conteneur d'étiquette `node:16`. Ce conteneur disponible sur le *Docker Hub* représente une distribution linux équipée de node.js. Nous choisissons ce conteneur de départ car nous savons que notre application a besoin de node.js pour fonctionner.

La ligne suivante `WORKDIR /usr/src/app` nous permet de désigner un dossier du conteneur qui contiendra l'ensemble de notre application. On aurait pu choisir ici n'importe quel emplacement.

On copie ensuite le fichier `package.json` dans cet emplacement avec `COPY package.json .`. Pour rappel, ce fichier contient la liste des dépendances à installer pour que notre projet fonctionne.

On installe ensuite ces dépendances avec `RUN npm install`.

Puis on copie l'ensemble de nos sources dans le conteneur avec `COPY . .`. C'est cette commande qui permet au conteneur de rassembler en particulier l'ensembles des fichiers javascript qui constitue l'application.

La ligne `EXPOSE 4000` a un rôle purement déclaratif et sert à dire que le conteneur mettra son port 4000 à disposition de la machine qui l'exécutera.

Enfin la ligne `CMD ["node", "index.js"]` est aussi déclarative. Elle détermine l'instruction qui sera exécutée lors du démarrage du conteneur. On aurait pu la remplacer par `CMD ["npm", "start"]` qui dans le cadre de notre projet fait la même chose. Pour information il s'agit d'une commande par défaut. On pourra la modifier lors de l'exécution du conteneur.

Construisons à présent l'image du conteneur à partir de ce fichier. On utilise pour cela le programme `docker` et sa sous-commande `build` comme ceci :

```
docker build -t isiscapitalistbackend:latest .
```

L'option `-t` permet de préciser l'étiquette (*tag*) donné à l'image du conteneur, et le paramètre principal qui vient en dernier (ici `.`) désigne le dossier qui contient le fichier `Dockerfile`.

Dans un terminal, placez-vous à la racine du projet, exécutez la commande et observez sa sortie. On voit bien que docker télécharge (*pull*) les différentes couches de l'image de départ, et y ajoute les programmes et fichiers définis dans le Dockerfile.

Quand la commande se termine, la nouvelle image est créée. Vous pouvez le vérifier en tapant la commande `docker images` qui liste toutes les images docker disponible sur votre machine.

Phase de Run

Pour déployer notre image, nous la placerons plus tard dans un dépôt privé accessible sur internet. Mais avant cela, nous allons exécuter localement pour vérifier son fonctionnement.

Pour exécuter une image de conteneur, on utilise encore une fois la commande `docker`, cette fois avec sa sous-commande `run` et différentes options qui définissent le contexte d'exécution.

Dans sa forme la plus simple, voici comment on lance notre conteneur :

```
docker run isiscapitalistbackend:latest
```

Exécutez la commande, et observez sa sortie. Si tout se passe bien, le conteneur est lancé et affiche son message de démarrage. Problème l'adresse affichée n'est pas accessible. Ouvrez votre navigateur sur l'adresse <http://localhost:4000/grapqh> pour le vérifier. Le serveur ne répond pas.

La raison est que votre serveur tourne à l'intérieur du conteneur et que ce conteneur se comporte comme une machine autonome, indépendante de la machine qui l'héberge. La conséquence est que le port 4000 du conteneur, n'a rien à voir avec le port 4000 de votre machine.

Pour relier les deux ports, il faut mettre en place un transfert de port (*port forwarding*) de votre machine vers le conteneur. C'est ce que permet l'option `-p` de la sous commande `docker run`. On lui passe en premier le numéro de port de la machine hôte (votre machine principale) et ensuite le numéro de port du conteneur vers lequel il est transféré. Si vous voulez par exemple transférer le port 80 de votre machine (en supposant qu'il n'est pas déjà occupé par un autre programme) vers le port 4000 sur lequel le conteneur écoute on écrira donc (après avoir interrompu le `docker run` précédent) :

```
docker run -p 80:4000 isiscapitalistbackend:latest
```

Vérifiez que votre navigateur vous permet bien d'accéder à l'adresse <http://localhost:80/grapghl> (ou plus simplement <http://localhost/grapghl>) et que c'est bien votre backend qui répond.

Phase de Push

Maintenant que nous avons vérifié que notre conteneur fonctionne correctement, il est temps de le placer dans un dépôt accessible sur internet. Nous allons utiliser celui proposé par GitHub qui présente l'avantage d'être gratuit pour tous les conteneurs publics, et qui dispose de quotas suffisants pour les conteneurs privés (<https://docs.github.com/en/billing/managing-billing-for-github-packages/about-billing-for-github-packages>). On aurait pu utiliser d'autres dépôts, nous y reviendrons dans la section suivante.

Pour que l'exemple soit plus représentatif d'une utilisation réelle dans le cadre d'un projet de développement, nous souhaitons que nos conteneurs soient privés. Pour cela, vous devez associer un mot de passe à votre dépôt. Pour celui de GitHub, ce mot de passe prend la forme d'un token que vous devez créer via leur site.

Pour cela, vous pouvez suivre la procédure décrite ici : <https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>. En résumé vous devez vous connecter sur le site de GitHub, dans la partie **Settings** de votre compte, allez chercher la partie **Developer settings** puis la partie **Personal access tokens** et cliquez sur **Generate a new token**. Donnez lui une brève description, cliquez au moins sur la permission **write packages** et surtout notez la valeur finale affichée car vous ne pourrez pas la retrouver par la suite.

Armé de ce token et de votre nom de connexion chez GitHub, retournez dans un terminal local, et connectez-vous au dépôt avec la commande suivante (**ghcr.io** est l'adresse du *Container Registry* de GitHub) :

```
docker login ghcr.io
```

en utilisant le couple login/password correspondant à votre couple identifiant de connexion/token de GitHub.

Si la connexion a réussi vous pouvez maintenant poster et télécharger des images privées dans le dépôt GitHub. C'est ce que nous faisons à y plaçant l'image de notre conteneur backend en deux temps :

Tout d'abord nous donnons une étiquette supplémentaire à notre image, en la taguant avec un prefixe qui correspondra à son adresse dans le dépôt (vous remplacerez **nicolassinger** par votre nom de connexion chez GitHub):

```
docker tag isiscapitalistbackend:latest  
ghcr.io/nicolassinger/isiscapitalistbackend:latest
```

Puis nous réalisons le *push* qui uploades l'image dans le dépôt :

```
docker push ghcr.io/nicolassinger/isiscapitalistbackend:latest
```

Voilà c'est fait votre image est accessible sur internet pour vos futurs déploiement.

Phase de Push automatisée

Dans les sections précédentes, après avoir créé le fichier `Dockerfile`, nous avons réalisé un ensemble d'opérations manuelles consistant à construire l'image du conteneur, et à la placer dans un dépôt. Ces opérations sont automatisables grâce à des outils d'intégration continue (CI) comme *Jenkins*, *Circle CI*, *GitHub Actions*, *GitLab CI* ou autres. Pour information, ces tâches d'automatisation font partie des missions rattachées à ce que l'on appelle le *Devops*.

Comme vous êtes nombreux à placer votre projet sur GitHub, le plus simple est d'illustrer cela avec les *GitHub Actions* qui est l'outil de CI intégré à GitHub. Nous donnerons aussi un exemple avec GitLab pour ceux qui ont hébergé leur projet chez eux.

GitHub

Avec GitHub, l'automatisation passe par la création d'un fichier situé dans un dossier prédéfini `.github/workflow` situé à la racine de votre projet. Ce fichier au format *yaml* décrit les étapes qui doivent être réalisées lors de certains événements, comme par exemple un nouveau *push* des sources de votre projet.

Plus précisément, nous voulons définir un tel fichier pour qu'à chaque *push* sur la branche master, le conteneur docker soit reconstruit, et qu'il soit automatiquement placé dans le dépôt GitHub.

Plutôt que d'écrire à partir de zéro ce script d'automatisation, nous allons nous aider des actions prédéfinies par GitHub pour en générer la base que nous modifierons légèrement. Rendez-vous sur la page d'accueil de votre projet backend sur GitHub (s'il est bien hébergé là bas, sinon voyez plus bas si vous l'avez hébergé sur GitLab), et cliquez sur le menu `Actions` puis que la partie `New workflow`. GitHub vous demande alors de choisir parmi une liste de workflow prédéfini. Celui qui nous intéresse est dans la partie *intégration continue* et s'appelle *Publish Docker Container*. Cliquez sur son bouton `Configure` et modifiez le légèrement pour que le script démarre à chaque push sur la branche master, et supprimez la fin du fichier qui consiste à signer l'image du conteneur. A la fin le fichier doit ressembler à ceci :

```
name: Docker

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

env:
  # Use docker.io for Docker Hub if empty
  REGISTRY: ghcr.io
  # github.repository as <account>/<repo>
  IMAGE_NAME: ${GITHUB_REPOSITORY}

jobs:
  build:

    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
    # This is used to complete the identity challenge
```

```

# with sigstore/fulcio when running outside of PRs.
id-token: write

steps:
- name: Checkout repository
  uses: actions/checkout@v3

# Install the cosign tool except on PR
# https://github.com/sigstore/cosign-installer
- name: Install cosign
  if: github.event_name != 'pull_request'
  uses: sigstore/cosign-installer@1e95c1de343b5b0c23352d6417ee3e48d5bcd422
  with:
    cosign-release: 'v1.4.0'

# Workaround: https://github.com/docker/build-push-action/issues/461
- name: Setup Docker buildx
  uses: docker/setup-buildx-
action@79abd3f86f79a9d68a23c75a09a9a85889262adf

# Login against a Docker registry except on PR
# https://github.com/docker/login-action
- name: Log into registry ${ env.REGISTRY }
  if: github.event_name != 'pull_request'
  uses: docker/login-action@28218f9b04b4f3f62068d7b6ce6ca5b26e35336c
  with:
    registry: ${ env.REGISTRY }
    username: ${ github.actor }
    password: ${ secrets.GITHUB_TOKEN }







# Extract metadata (tags, labels) for Docker
# https://github.com/docker/metadata-action
- name: Extract Docker metadata
  id: meta
  uses: docker/metadata-action@98669ae865ea3cfffcbcaa878cf57c20bbf1c6c38
  with:
    images: ${ env.REGISTRY }/${ env.IMAGE_NAME }

# Build and push Docker image with Buildx (don't push on PR)
# https://github.com/docker/build-push-action
- name: Build and push Docker image
  id: build-and-push
  uses: docker/build-push-action@ad44023a93711e3deb337508980b4b5e9bc5dc5dc
  with:
    context: .
    push: ${ github.event_name != 'pull_request' }
    tags: ${ steps.meta.outputs.tags }
    labels: ${ steps.meta.outputs.labels }

```


Même s'il est un peu difficile de lire ce fichier quand on ne maîtrise pas encore les instructions des *GitHub Actions*, on devine qu'il réalise une série d'étape visant à construire à partir du Dockerfile de votre projet, un conteneur publié sur le dépôt GitHub.


Pour tester le fonctionnement de ce script, faites un nouveau *commit* de votre projet suivi d'un *push* sur GitHub et rechargez la page d'accueil. Vous devez voir un petit rond orange s'afficher dans l'entête du code (voir image ci-dessous).


 nicolasSinger	Update docker-publish.yml	 267bc20 now	 11 commits
 .github/workflows	Update docker-publish.yml		now
 public/icones	Progression		14 days ago
 userworlds	almost fonctionnal, no intense tests		12 days ago


En cliquant sur ce rond, vous pouvez accéder aux détails de l'automatisation en cours et distinguer les différentes étapes en cours de construction.

build
Started 21s ago

>  Set up job

>  Checkout repository

>  Install cosign

▼  Setup Docker buildx

```
106 #1 pulling image moby/buildkit:buildx-stable-1 4.8s done
107 #1 creating container buildx_buildkit_builder-5d0c0a02-8f12-46bf-9fb6-39d3c0bbd0b30
108 #1 creating container buildx_buildkit_builder-5d0c0a02-8f12-46bf-9fb6-39d3c0bbd0b30 0.7s done
109 #1 DONE 5.6s
110 Name:      builder-5d0c0a02-8f12-46bf-9fb6-39d3c0bbd0b3
111 Driver:    docker-container
112 Nodes:
113 Name:      builder-5d0c0a02-8f12-46bf-9fb6-39d3c0bbd0b30
114 Endpoint:  unix:///var/run/docker.sock
115 Status:    running
116 Flags:     --allow-insecure-entitlement security.insecure --allow-insecure-entitlement network.host
117 Platforms: linux/amd64, linux/amd64/v2, linux/amd64/v3, linux/386
118
119 ▶ Inspect builder
130 ▶ BuildKit version
```

☐ Log into registry \${{ env.REGISTRY }}

☐ Extract Docker metadata

☐ Build and push Docker image

☐ Post Checkout repository

☐ Post Setup Docker buildx

A la fin si tout s'est bien passé, le *build* est réussi et un nouveau package apparaît à droite de votre code :

main 1 branch 0 tags Go to file Add file Code

nicolasSinger Update docker-publish.yml ✓ 267bc20 6 minutes ago 11 commits

.github/workflows	Update docker-publish.yml	6 minutes ago
public/icons	Progression	14 days ago
userworlds	almost fonctionnal, no intense tests	12 days ago
.gitignore	almost fonctionnal, no intense tests	12 days ago
.gitlab-ci.yml	fix gitlab ci	11 days ago
Dockerfile	gitlab ci	11 days ago
index.js	almost fonctionnal, no intense tests	12 days ago
package-lock.json	Initial Commit	14 days ago
package.json	Initial Commit	14 days ago
resolvers.js	almost fonctionnal, no intense tests	12 days ago
schema.js	almost fonctionnal, no intense tests	12 days ago
world.js	Progression	14 days ago

Add a README with an overview of your project. Add a README

About Serveur GraphQL avec node pour ISIS Capitalist
0 stars 1 watching 0 forks

Releases No releases published [Create a new release](#)

Packages 1
isiscapitalistnodegraphql

Languages
JavaScript 98.6% Dockerfile 1.4%

Cliquez dessus pour obtenir l'adresse de votre conteneur dans le dépôt :



Serveur GraphQL avec node pour ISIS Capitalist

Install from the command line:

[Learn more](#)

```
$ docker pull ghcr.io/nicolassinger/isiscapitalistnodegraphql:main
```

Recent tagged image versions

main

↓ 0

Published 6 minutes ago · Digest ...

[View and manage all versions](#)

GitLab

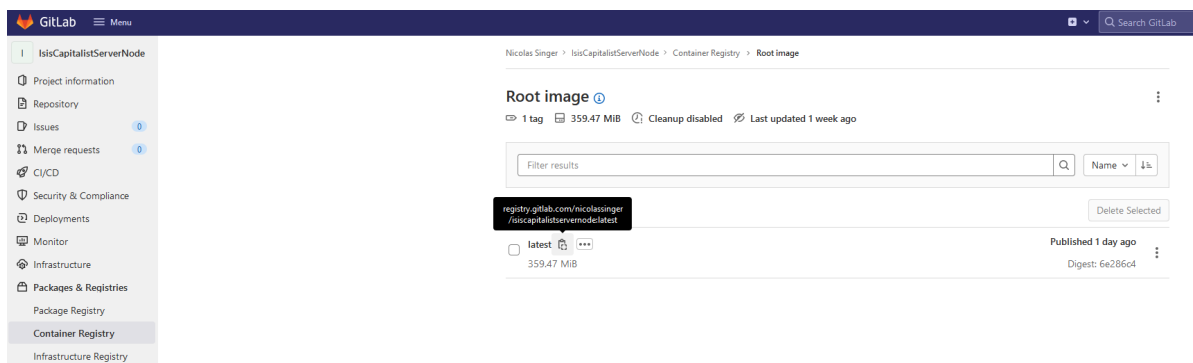
Chez GitLab, c'est le même principe. Le script d'automatisation s'appelle `.gitlab-ci.yml` et réside directement à la racine du projet. Sa syntaxe est plus simple que pour les *GitHub Actions*, voici son contenu pour automatiser la construction et le déploiement du conteneur dans le dépôt de GitLab.


```
image: docker:latest
services:
  - docker:dind

stages:
  - package

docker-build:
  stage: package
  script:
    - docker build -t $CI_REGISTRY_IMAGE .
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker push $CI_REGISTRY_IMAGE
```

Une fois ce fichier créé, les prochains push déclenchent l'automatisation. Pour obtenir l'adresse de votre conteneur dans le dépôt, rendez-vous dans l'espace **Packages & Registries** de GitLab puis dans la partie **Container Registry**



Conclusion

Une fois le Dockerfile défini, les outils d'intégration continue permettent l'automatisation de la construction et de la mise en dépôt des conteneurs. Cette automatisation peut bien entendu créer plusieurs conteneurs différents en fonction de la branche de développement. On peut alors avoir un conteneur qui représente la branche master, un autre pour la branche développement et encore un autre pour la branche de production.

Il ne nous reste plus qu'à voir comment on déploie un conteneur dans un environnement d'exécution.

Phase de déploiement

Nous avons déjà vu comment on exécute un conteneur en local, à partir de son image présente sur la machine devant laquelle on se trouve. Si l'image n'est pas présente sur la machine (parce que par exemple ce n'est pas la machine qui a servi à la construire), on récupère cette image avec un *docker pull*.

Par exemple si vous voulez lancer le conteneur sur une machine quelconque, installez docker sur cette machine et exécutez les deux commandes (pensez bien à remplacer dans ces commandes l'adresse de votre conteneur) :

```
docker pull ghcr.io/nicolassinger/isiscapitalistnodegraphql:main
```

```
docker run -p 80:4000 ghcr.io/nicolassinger/isiscapitalistnodegraphql:main
```

Cela suffit pour exécuter votre application. Notons d'ailleurs que le `docker pull` n'est pas nécessaire, le `docker run` étant capable de télécharger tout seul l'image manquante si elle bien précisée avec son adresse complète comme c'est le cas ici.

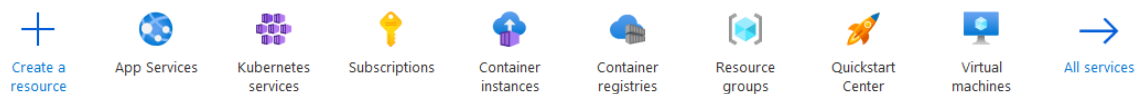
En production, notre conteneur sera probablement hébergé sur une machine serveur qui lui permettra d'avoir une adresse internet publique. Si vous disposez ou louez une telle machine, vous pouvez déployer le conteneur exactement comme en local. Mais vous pouvez aussi louer un service spécialisé dans l'exécution de conteneurs docker. De tels services sont fournis par les principaux fournisseurs *clouds*, comme Amazon AWS, Microsoft Azure, Google Cloud, OVH Public Cloud, Scaleway, etc. L'intérêt d'utiliser de tels services est de ne pas s'embêter à gérer une machine physique ou virtuelle. On déploie son conteneur via le service, en échange le service nous donne l'URL auquel le conteneur est hébergé, et nous permet de gérer son cycle de vie (démarrage, pause, stop, etc.)

Voyons un exemple avec Microsoft Azure. Si vous êtes étudiants, vous avez accès à un test du service sans carte bancaire. Commencez donc par vous inscrire chez Microsoft Azure avec un compte Microsoft (si vous n'en avez pas, créez en un) en suivant la procédure décrite ici : <https://azure.microsoft.com/fr-fr/free/students/>

Une fois l'inscription terminée, tout se gère à partir de la page d'accueil d'Azure : <https://portal.azure.com>

Nous allons créer une nouvelle ressource de type *App Services*. Utilisez le bandeau principal pour cela. Si vous ne voyez pas l'icône des *App Services*, cliquez sur *All Services*, vous la trouverez dans la partie *Compute*.

Azure services



Cliquez ensuite sur le bouton *Create* pour ajouter une nouvelle application. Dans le formulaire qui apparaît, remplissez le premier écran en précisant qu'il s'agit d'une application base sur un conteneur (partie *Publish*) et qu'elle doit être hébergée en France. Au final le premier écran doit ressembler à la copie écran ci-dessous :

Create Web App ...

[Basics](#) [Docker](#) [Monitoring](#) [Tags](#) [Review + create](#)

App Service Web Apps lets you quickly build, deploy, and scale enterprise-grade web, mobile, and API apps running on any platform. Meet rigorous performance, scalability, security and compliance requirements while using a fully managed platform to perform infrastructure maintenance. [Learn more](#) ↗

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ Azure for Students ▼

Resource Group * ⓘ ISISCapitalist ▼

[Create new](#)

Instance Details

Need a database? [Try the new Web + Database experience.](#) ↗

Name * SuperAppli ✓ .azurewebsites.net

Publish * ☐ Code ☒ Docker Container ☐ Static Web App

Operating System * ☒ Linux ☐ Windows

Region * France Central ▼

ⓘ Not finding your App Service Plan? Try a different region or select your App Service Environment.

App Service Plan

App Service plan pricing tier determines the location, features, cost and compute resources associated with your app. [Learn more](#) ↗

Linux Plan (France Central) * ⓘ ASP-ISISCapitalist-addf (B1) ▼

[Create new](#)

Sku and size * **Basic B1**
100 total ACU, 1.75 GB memory

Zone redundancy

An App Service plan can be deployed as a zone redundant service in the regions that support it. This is a deployment time only decision. You can't make an App Service plan zone redundant after it has been deployed [Learn more](#) ↗

Zone redundancy ☐ **Enabled:** Your App Service plan and the apps in it will be zone redundant. The minimum App Service plan instance count will be three.

☒ **Disabled:** Your App Service Plan and the apps in it will not be zone redundant. The minimum App Service plan instance count will be one.

[Review + create](#)

[< Previous](#)

[Next : Docker >](#)

Cliquez ensuite le bouton *Next: Docker* pour accéder au deuxième écran. C'est ici que vous précisez quelle est l'adresse du conteneur avec les identifiants d'accès si l'adresse est privée :

Create Web App ...

Basics Docker Monitoring Tags Review + create

Pull container images from Azure Container Registry, Docker Hub or a private Docker repository. App Service will deploy the containerized app with your preferred dependencies to production in seconds.

Options	Single Container
Image Source	Private Registry
Private registry options	
Server URL *	https://ghcr.io
Username	nicolasSinger
Password
Image and tag *	ghcr.io/nicolassinger/isiscapitalistnodegraphql:main
Startup Command ⓘ	

Remplacez évidemment les données de l'écran par vos identifiants GitHub et l'étiquette exacte de votre conteneur.

Terminez le processus en cliquant sur le bouton *Review + Create* puis sur le bouton *Create*. Attendez un moment que le déploiement soit terminé. Quand c'est le cas, cliquez sur *Go to ressource* pour obtenir l'écran de gestion de l'application.

Il nous reste simplement à expliquer que nous voulons rediriger le port 80 du service Azure vers le port 4000 de notre conteneur. Pour cela ouvrez le menu *Configuration* de l'application. Ajoutez une nouvelle propriété en cliquant sur *New application setting*. La nouvelle propriété à ajouter s'appelle `WEBSITES_PORT` et sa valeur doit être réglée sur 4000.

Au final la configuration de l'application doit ressembler à l'écran ci-dessous :

Application settings

Application settings are encrypted at rest and transmitted over an encrypted channel. You can choose to d

[+ New application setting](#) [👁 Show values](#) [✎ Advanced edit](#)

[🔍 Filter application settings](#)

Name	Value
DOCKER_REGISTRY_SERVER_PASSWORD	👁 Hidden value. Click to show value
DOCKER_REGISTRY_SERVER_URL	👁 Hidden value. Click to show value
DOCKER_REGISTRY_SERVER_USERNAME	👁 Hidden value. Click to show value
WEBSITES_ENABLE_APP_SERVICE_STORAGE	👁 Hidden value. Click to show value
WEBSITES_PORT	🔊 4000

Cliquez sur le bouton *Save* pour que les changements soit pris en compte et revenez à l'écran général via le menu *Overview*.

Repérez les URL qu'Azure a affecté à votre application.

URL : <https://superappli.azurewebsites.net>
Health Check : [Not Configured](#)
App Service Plan : [ASP-ISISCapitalist-addf \(B1: 1\)](#)
FTP/deployment username : No FTP/deployment user set
FTP hostname : <ftp://waws-prod-par-005.ftp.azurewebsites.windows.net/site/wwwroot>
FTPS hostname : <ftps://waws-prod-par-005.ftp.azurewebsites.windows.net/site/wwwroot>

Testez l'url en https à partir de votre navigateur en le terminant par `/graphql` :

Cela devrait fonctionner (le premier accès est un peu long)

Ca y est votre backend est en production chez Microsoft Azure. Si l'image de son conteneur est mis à jour, il suffit de redémarrer l'application pour que la nouvelle image soit *pull* par Azure.

Le frontend

Pour le frontend, c'est exactement pareil, il nous faut simplement définir le Dockerfile qui permet de construire la partie cliente de l'application. Le voici :

```
# build environment
FROM node:16-alpine as build
WORKDIR /app
ENV PATH /app/node_modules/.bin:$PATH
COPY package.json ./
COPY package-lock.json ./
RUN npm ci
```

```
RUN npm install react-scripts@3.4.1 -g --silent
COPY . ./
RUN npm run build

# production environment
FROM nginx:stable-alpine
COPY --from=build /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Il est constitué de deux parties :

- La première partie qui *compile* le projet en rassemblant les sources dans un gros fichier javascript/html/css (webpack). C'est e que fait la commande `npm run build`
- La deuxième qui crée un conteneur avec un serveur web (nginx) qui fournira ce fichier au navigateur.

A vous de faire le reste pour automatiser avec GitHub ou GitLab la création et la mise en dépôt du conteneur. Vous ferez attention de modifier dans le code l'adresse du serveur pour qu'elle pointe sur l'adresse de production du backend (hébergé chez Azure), à moins que votre client permette de saisir cette adresse dans son interface.

Créez ensuite une *App service* supplémentaire chez Microsoft Azure pour héberger et exécuter le conteneur frontend. Comme le Dockerfile du conteneur frontend le fait écouter sur son port 80, vous n'avez cette fois pas besoin de changer la configuration du service.

Vérifiez que tout fonctionne.