

Networking

La communication entre conteneurs

v1.0.0 | 17/11/2024 | Auteur : Bauer Baptiste

Chapitre

Sommaire

1. Introduction	1
2. Présentation de l'application de démonstration	2
3. Communication	6
3.1. Préambule	6
3.2. Situation 1 : Du conteneur vers un serveur distant (HTTP)	6
3.3. Situation 2 : Du conteneur vers la machine hôte locale	7
3.4. Situation 3 : Du conteneur vers un autre conteneur	8
4. Mise en pratique : Création du conteneur et communication avec le serveur API REST	10
5. Mise en pratique : Faire communiquer le conteneur avec le localhost	15
6. Mise en pratique : Communication entre conteneurs [Solution basique]	18
7. Mise en pratique : Les Networks Docker [Solution élégante]	21
8. Les pilotes (drivers) de Docker Network	25

1. Introduction

Dans ce chapitre, nous allons explorer la manière dont les conteneurs peuvent communiquer entre eux. Pour cela nous allons nous appuyer sur une application de démonstration qui nécessite de communiquer avec des services distants : une API et une base de données.

Vous avez dit 'API' ?

Dans le cadre du développement d'applications, il est courant de créer une base de données, comme une base de données SQL, et de la solliciter pour obtenir des données que nous utilisons directement dans notre application. Cependant, il existe une autre approche : développer une application distincte, destinée exclusivement à gérer les données de la base de données, c'est-à-dire l'ajout, la modification et la suppression.

Cette application spécifique ne possède pas d'interface utilisateur. Au lieu de cela, les actions sont accessibles uniquement via des routes ou des URL qui transmettent, par le biais du protocole HTTP, des requêtes de publication (méthode POST), de modification (méthode PUT), de suppression (méthode DELETE) ou de sélection (méthode GET). Chaque action renvoie généralement une réponse du serveur au format JSON.

Par exemple, le service de Météo Nationale fournit des données météorologiques que vous pouvez intégrer dans votre application pour afficher les prévisions météorologiques. Ces données sont accessibles via une ou des URL que vous pouvez interroger pour récupérer les informations. On appelle cela une **API**.

Cette approche est communément appelée API REST (Application Programming Interface Representational State Transfer). Par exemple, l'URL <https://swapi.dev/api/people/1> permet de récupérer les données du personnage de Star Wars dont l'identifiant est 1.

Nous avons déjà vu comment conteneuriser une application simple. Nous allons maintenant compliquer un peu les choses en ajoutant des services externes à notre application.

Si vous avez bien compris le fonctionnement de Docker, vous devriez être en mesure de créer une image Docker contenant une application ainsi que sa propre base de données. Cependant, il est important de respecter un principe fondamental de Docker : un service équivaut à un conteneur. Par conséquent, si nous avons une application et sa base de données, nous devons créer une image pour l'application et une autre image pour la base de données.

Nous devons ensuite monter les deux conteneurs et les faire communiquer entre eux, ce qui peut compliquer un peu les choses.

Avant d'entrer dans le vif du sujet, nous allons d'abord présenter l'application de démonstration que nous allons utiliser pour illustrer la communication entre conteneurs.

2. Présentation de l'application de démonstration

Récupérez l'application via le fichier : `networks-starting-setup.zip` et décompressez le contenu :

En lisant le code source dans le fichier `app.js`, vous reconnaîtrez certainement des éléments de code que nous avons déjà vu dans les chapitres précédents :

- Nous créons notre propre API REST à partir des données récupérées par l'API `swapi`, par exemple avec cette route `/movies`

Fichier : `code/networks-starting-setup/app.js`

```
app.get('/movies', async (req, res) => {  
  // code  
});
```

- On consomme les données de l'API `swapi` grâce à la librairie JS `Axios`.

Fichier : `code/networks-starting-setup/app.js`

```
app.get('/movies', async (req, res) => {  
  try {  
    const response = await axios.get('https://swapi.dev/api/films');  
    res.status(200).json({ movies: response.data });  
  } catch (error) {  
    res.status(500).json({ message: 'Something went wrong.' });  
  }  
});
```

- L'application permet aussi de mettre des films ou des personnages en favoris et de les stocker en base de données `MONGODB` en utilisant la librairie `mongoose`.

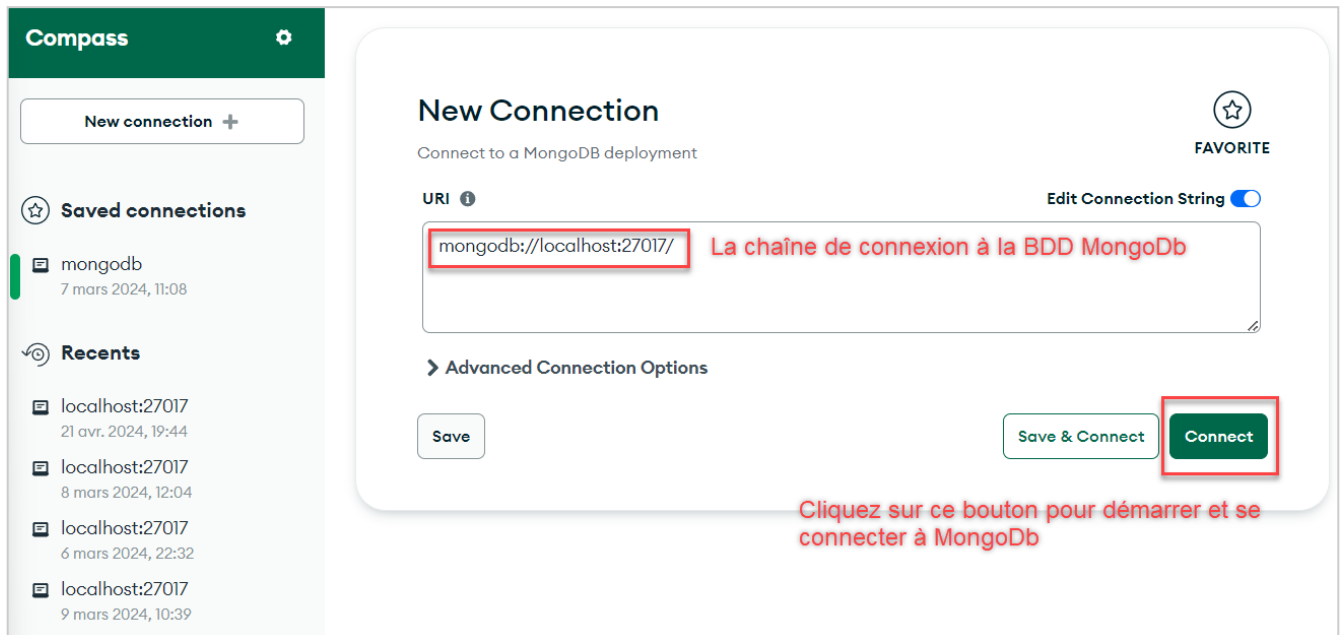
Fichier : `code/networks-starting-setup/app.js`

```
const mongoose = require('mongoose');  
// .. some code  
app.post('/favorites', async (req, res) => {  
  // .. some code  
});
```

Bien évidemment, nous pouvons tester notre application en local en installant MongoDB et son environnement, NodeJS, etc..

- Allez sur le site MongoDB et installez MongoDB Compass : <https://www.mongodb.com/products/tools/compass> Il s'agit d'une application avec une interface graphique (GUI) qui permet de créer et manipuler une base de données MongoDB. Je ne détaille pas l'installation ici.

- En lançant l'application, vous devriez voir cet écran :



- Maintenant, nous pouvons ouvrir un Terminal dans le dossier de notre application et installer les packages nécessaires :

```
PS C:\Users\baptiste\Downloads\APP> npm install
npm WARN deprecated axios@0.20.0: Critical security vulnerability fixed i
thub.com/axios/axios/pull/3410

added 100 packages, and audited 101 packages in 6s

14 packages are looking for funding
  run `npm fund` for details

1 high severity vulnerability

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
npm notice
npm notice New minor version of npm available! 10.2.5 -> 10.7.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v10.7.0
npm notice Run npm install -g npm@10.7.0 to update!
npm notice
PS C:\Users\baptiste\Downloads\APP> |
```

- Nous vérifions dans le code source que la chaîne de connexion à MongoDB soit la même que sur mon serveur local :

Fichier : code/networks-starting-setup/app.js

```
// .. some code
mongoose.connect(
  'mongodb://localhost:27017/swfavorites',
```

```
{ useNewUrlParser: true },  
(err) => {  
  if (err) {  
    console.log(err);  
  } else {  
    app.listen(3000);  
  }  
}  
};
```

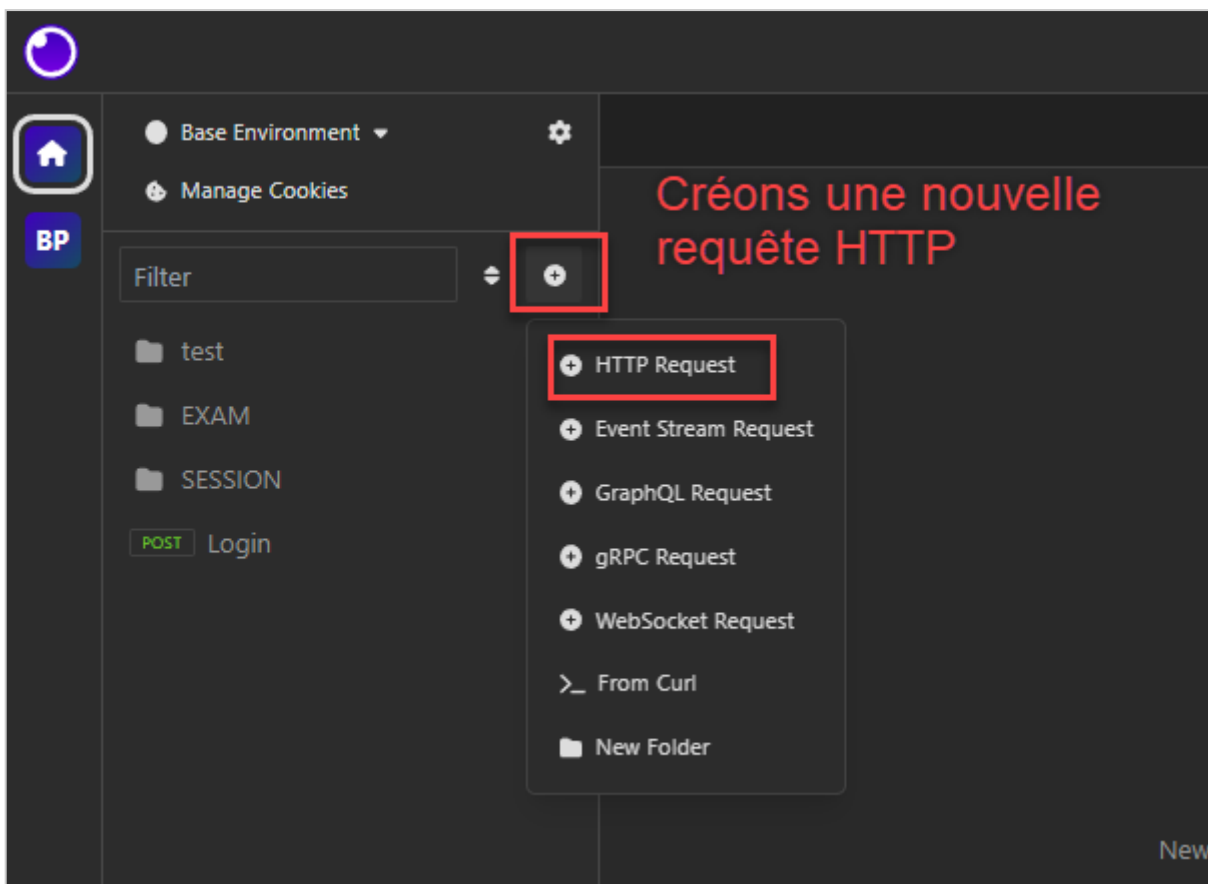
Et maintenant testons l'application :

```
node app.js
```

Comment tester notre application ? nous n'avons pas d'interface graphique (appelée aussi *UI*)!.

Notre application est une API ! Il faudra alors installer un logiciel qui nous permettra d'envoyer des requêtes HTTP facilement. Personnellement, j'utilise **Insomnia**, mais il en existe d'autres comme **PostMan**.

- Allez sur : <https://insomnia.rest/> et téléchargez l'application
- Une fois Insomnia lancé :



GET `http://localhost:3000/movies`

200 OK 592 ms 18.2 KB

Body Auth Query Headers Docs

Preview

1- {
2- "movies": [
3- "count": 6,
4- "next": null,
5- "previous": null,
6- "results": [
7- {
8- "title": "A New Hope",
9- "episode_id": 4,
10- "opening_crawl": "It is a period of civil war. Rebel spaceships, striking from a hidden base, won their first victory against the evil Galactic Empire. During the battle, Rebel
steal secret plans to the Empire's ultimate weapon, the Death Star, an armored space
power that can destroy an entire planet. Pursued by the Empire's sinister agents, Princess
board her starship, and flee to the planet Yavin. The mission is to destroy the plans and restore
galaxy....",
11- "director": "George Lucas",
12- "producer": "Gary Kurtz, Rick McCallum",
13- "release_date": "1977-05-25",
14- "characters": [
15- "https://swapi.dev/api/people/1/",
16- "https://swapi.dev/api/people/2/",
17- "https://swapi.dev/api/people/3/",
18- "https://swapi.dev/api/people/4/",
19- "https://swapi.dev/api/people/5/",
20- "https://swapi.dev/api/people/6/",
21- "https://swapi.dev/api/people/7/",
22- "https://swapi.dev/api/people/8/",
23- "https://swapi.dev/api/people/9/",
24- "https://swapi.dev/api/people/10/",
25- "https://swapi.dev/api/people/11/",
26- "https://swapi.dev/api/people/12/",
27- "https://swapi.dev/api/people/13/",
28- "https://swapi.dev/api/people/14/",
29- "https://swapi.dev/api/people/15/",
30- "https://swapi.dev/api/people/16/",
31- "https://swapi.dev/api/people/17/",
32- "https://swapi.dev/api/people/18/",
33- "https://swapi.dev/api/people/19/",
34- "https://swapi.dev/api/people/20/"
35-],
36- "planets": [
37- "https://swapi.dev/api/planets/1/",
38- "https://swapi.dev/api/planets/2/",
39- "https://swapi.dev/api/planets/3/"
40-],
41- "starships": [
42- "https://swapi.dev/api/starships/2/",
43- "https://swapi.dev/api/starships/3/",
44- "https://swapi.dev/api/starships/4/",
45- "https://swapi.dev/api/starships/5/",
46- "https://swapi.dev/api/starships/6/",
47- "https://swapi.dev/api/starships/7/",
48- "https://swapi.dev/api/starships/8/",
49- "https://swapi.dev/api/starships/9/",
50- "https://swapi.dev/api/starships/10/",
51- "https://swapi.dev/api/starships/11/",
52- "https://swapi.dev/api/starships/12/",
53- "https://swapi.dev/api/starships/13/",
54- "https://swapi.dev/api/starships/14/",
55- "https://swapi.dev/api/starships/15/",
56- "https://swapi.dev/api/starships/16/",
57- "https://swapi.dev/api/starships/17/",
58- "https://swapi.dev/api/starships/18/",
59- "https://swapi.dev/api/starships/19/",
60- "https://swapi.dev/api/starships/20/"
61-]
62- }
63-]
64- }
65- }

Nous allons faire une requête de type GET en localhost, sur le port 3000 (c'est le port d'écoute que l'on retrouve dans le code source).

Et nous allons suivre la route : "movies"

Votre requête qui porte le nom par défaut : "New Request"

Après avoir cliqué sur le bouton "Send" ... la requête HTTP GET est envoyée à notre application, qui retourne une réponse captée par Insomnia .

Nous obtenons donc la liste des films Starwars et d'autres données.

Vous pouvez tester d'autres routes que l'on retrouve dans le code source. Je vous laisse découvrir les choses.

Nous avons une meilleure compréhension de l'application de démonstration, et nous voyons qu'elle est beaucoup plus complexe que ce que nous avons utilisé jusqu'à présent.



Nous allons maintenant conteneuriser cette application !

En créant plusieurs conteneurs afin de respecter le principe :

- 1 service = 1 conteneur

3. Communication

3.1. Préambule

Après avoir étudié le fonctionnement de notre application, nous allons examiner les cas de figure où une **communication** sera initiée entre le conteneur et d'autres services.

Il y a 3 situations que l'on peut répertorier :

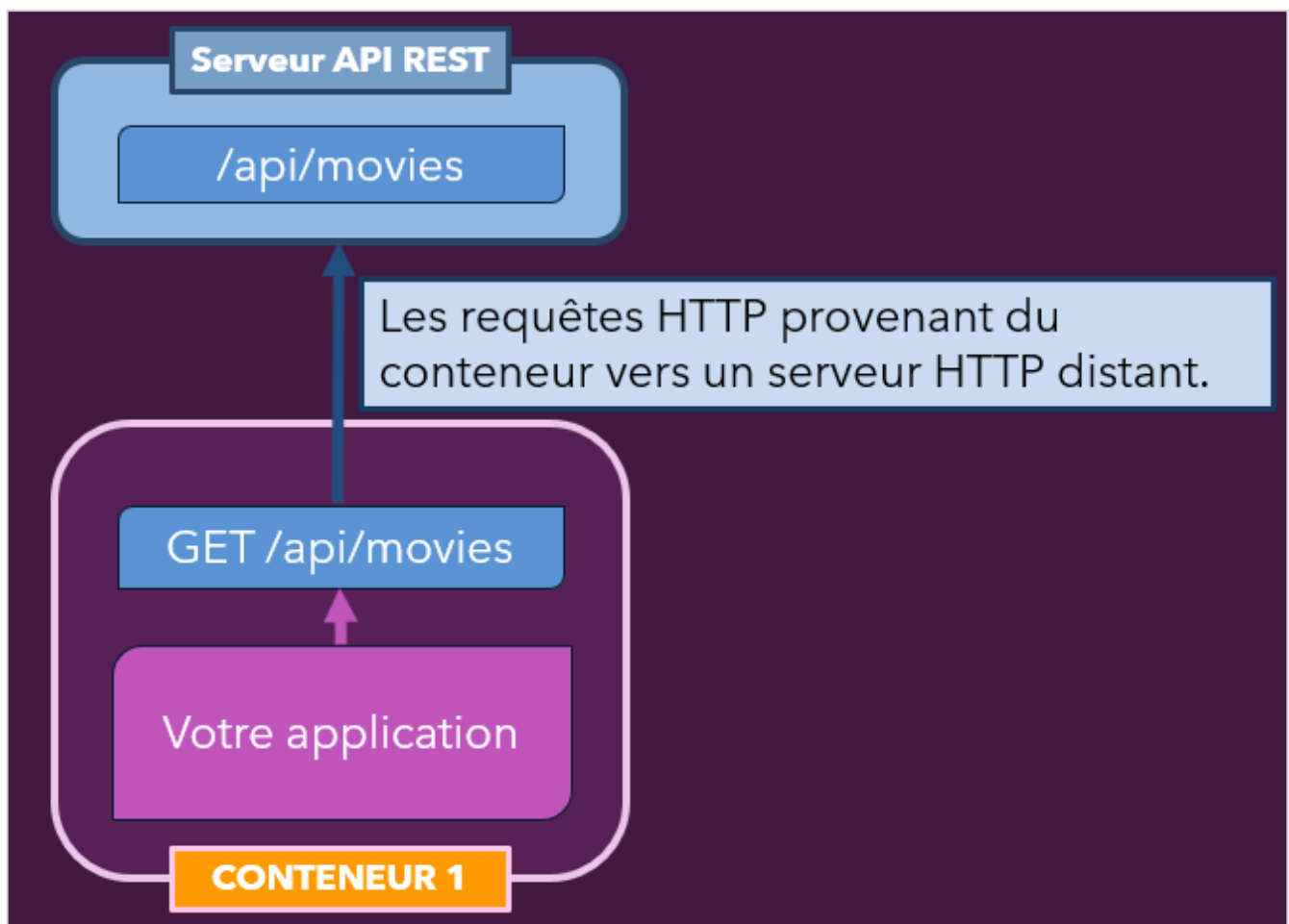
1. Du **conteneur** vers un **serveur distant** (*par exemple via HTTP vers une API*)
2. Du **conteneur** vers la **machine hôte locale** (*par exemple pour accéder à une base de données*)
3. Entre deux conteneurs



Mais avant de continuer, assurez-vous d'avoir fermé :

- Le serveur Node (**CTRL** + **C** dans le terminal)
- La fenêtre de MongoDB Compass.

3.2. Situation 1 : Du conteneur vers un serveur distant (HTTP)



L'API REST que nous interrogeons, n'est pas hébergée dans le conteneur et je ne l'ai pas non plus moi-même créée. C'est un service qui existe et que je désire **consommer** depuis mon conteneur.

Notre application va créer de nouvelles routes comme `/movies` ou `/people` par exemple, qui vont interroger l'API distante et retourner les données reçues.

En d'autres termes, notre application va devenir un proxy pour l'API distante. Elle va récupérer les données de l'API distante et les retourner à l'utilisateur.

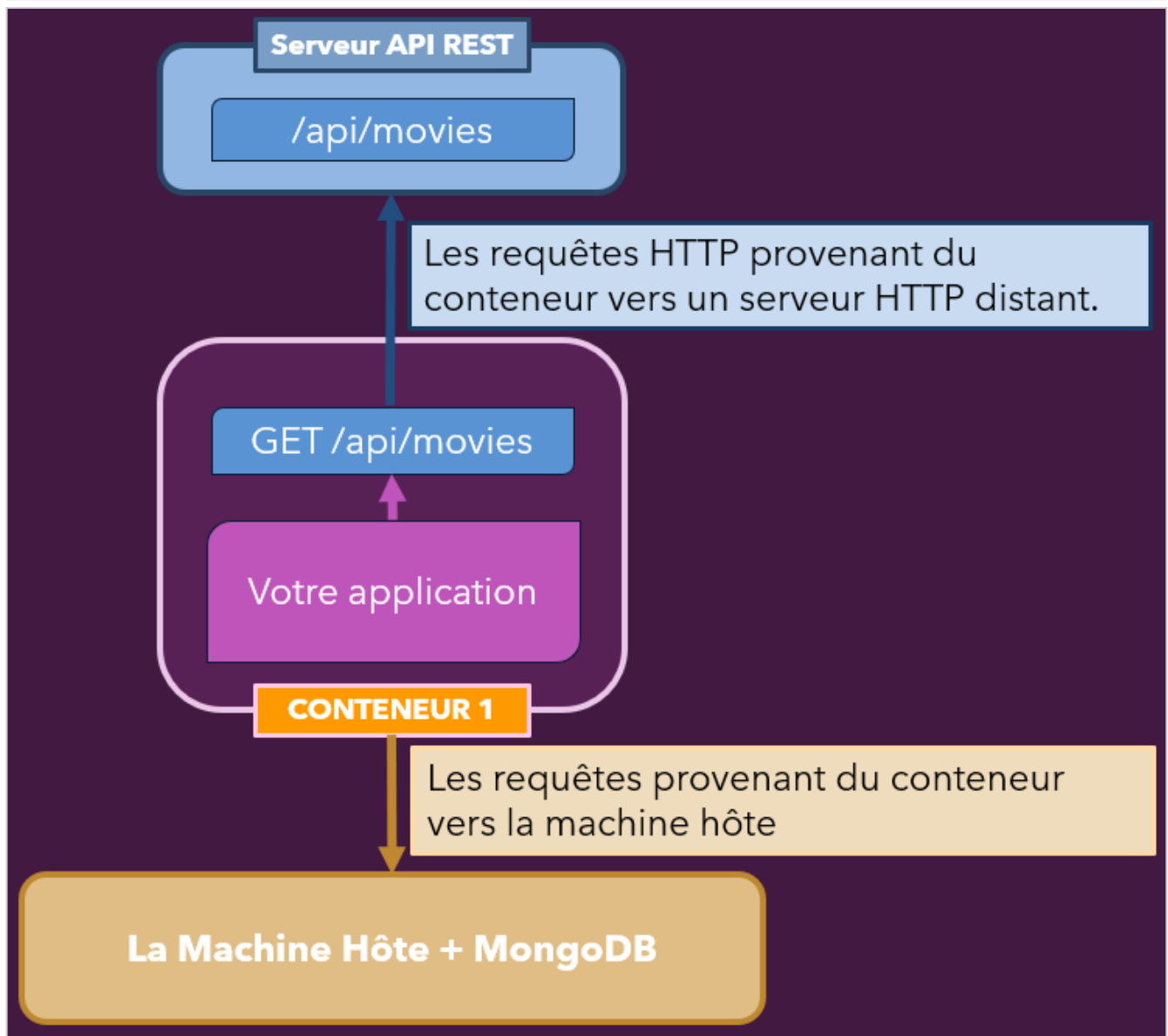
Il y a une communication **HTTP** entre notre application et l'API distante.



Et comme l'application est hébergée dans un conteneur, il faudra **s'assurer que la requête HTTP puisse sortir du conteneur et atteindre l'API distante**, puis, que la réponse puisse revenir à l'intérieur de notre conteneur.

3.3. Situation 2 : Du conteneur vers la machine hôte locale

Notre application doit pouvoir communiquer avec MongoDB installé sur la machine hôte pour effectuer des requêtes CRUD sur la base de données(**Create, Read, Update, Delete**)



```
mongoose.connect(
  'mongodb://localhost:27017/swfavorites',
  { useNewUrlParser: true },
  (err) => {
    if (err) {
      console.log(err);
    } else {
      app.listen(3000);
    }
  }
);
```

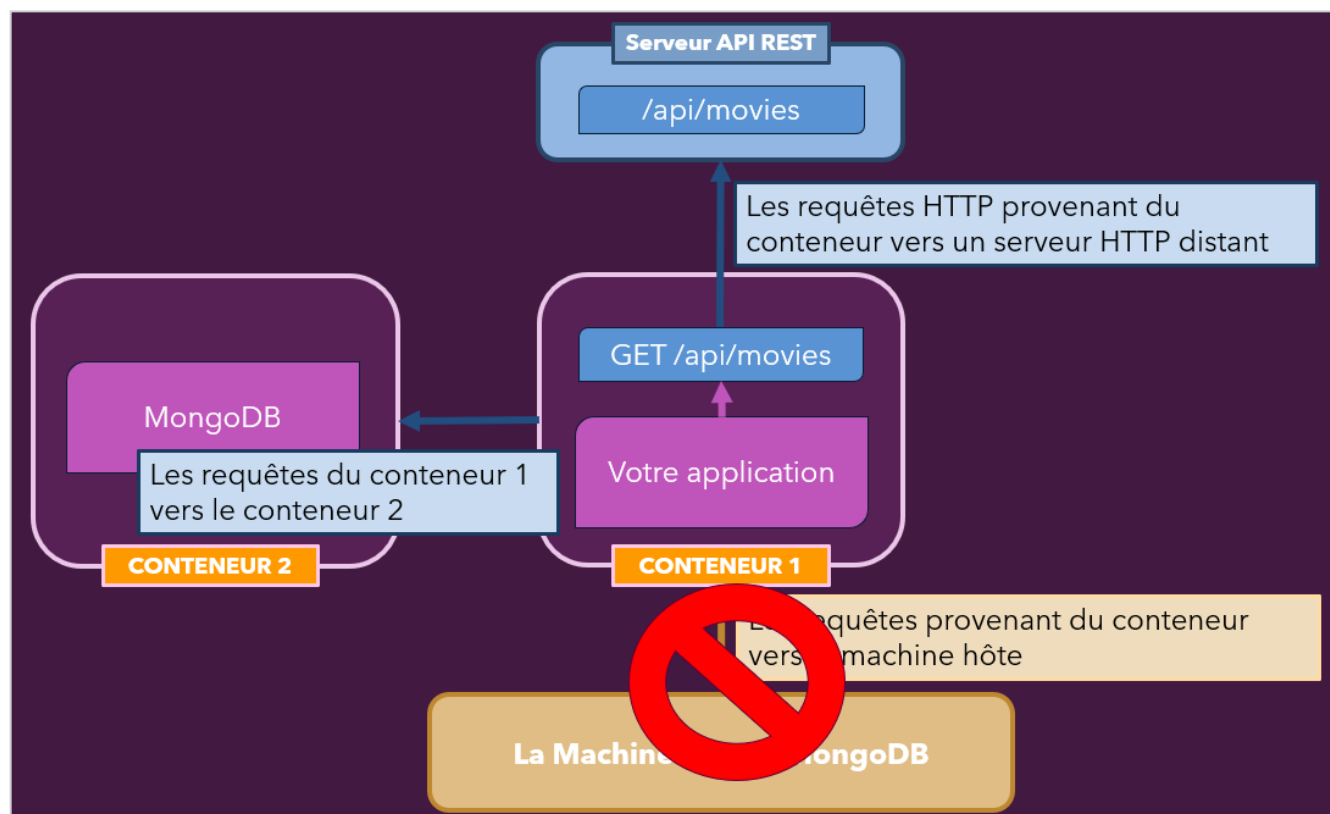
3.4. Situation 3 : Du conteneur vers un autre conteneur

Dans la situation 2, nous avons identifié le besoin de communiquer avec MongoDB installé sur la machine hôte.

Mais il est possible que le service de base de données soit lui aussi inclus dans un autre conteneur.

Et c'est d'ailleurs fortement recommandé !

Par conséquent, il faudra que le conteneur de notre application puisse dialoguer avec le conteneur de la base de données.



Donc, nous devons créer une image pour l'application NodeJs et une autre image pour la base de données MongoDB.

4. Mise en pratique : Création du conteneur et communication avec le serveur API REST

Dans le dossier de l'application de démonstration, supprimer le fichier `node_module` et `package-lock.json` si vous avez réalisé l'installation de l'application vue précédemment.

Nous allons maintenant créer une image avec le `Dockerfile` présent :

```
docker build -t favorites-node .
```

Ensuite lançons un conteneur basé sur l'image `favorites-node` :

```
docker container run --name favorites -d --rm -p 3000:3000 favorites-node
```



Rappels

- `--name` Permet de donner un nom au conteneur
- `-d` Démarre le conteneur en tâche de fond
- `--rm` Supprime le conteneur dès qu'il est arrêté
- `-p` Associé le port 3000 de la machine hôte avec le port 3000 du conteneur
- `favorites-node` nom de l'image

Il n'est pas nécessaire de définir de **volumes**, car l'application n'écrit rien dans des fichiers qui doivent persister après la suppression du conteneur. Les seules données qui seront écrites sont celles qui seront stockées dans la base de données, mais elles ne seront pas stockées dans ce conteneur.

En exécutant la commande, nous recevons un identifiant de conteneur. Vérifions :

```
docker container ps
docker container ps -a
```

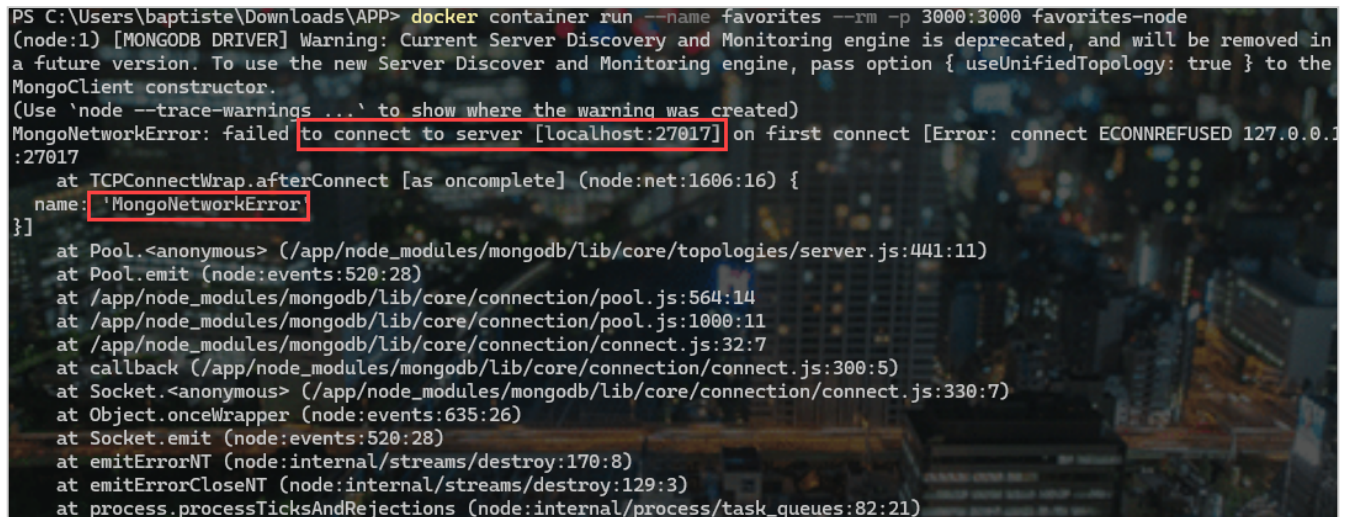
Nous remarquons que la liste est vide ! Aucun conteneur n'a été démarré ni créé après la commande précédente !

Cela est tout à fait normal ! En effet, nous avons démarré le conteneur avec le paramètre `--rm`, ce qui signifie que le conteneur n'a pas pu démarrer correctement, qu'il s'est arrêté et qu'il a été supprimé.

Regardons plus en détail d'où peut provenir ce problème, en lançant de nouveau la commande sans le mode détaché `-d`.

```
docker container run --name favorites --rm -p 3000:3000 favorites-node
```

Le conteneur se lance bien, mais l'application plante!



```
PS C:\Users\baptiste\Downloads\APP> docker container run --name favorites --rm -p 3000:3000 favorites-node
(node:1) [MONGODB DRIVER] Warning: Current Server Discovery and Monitoring engine is deprecated, and will be removed in
a future version. To use the new Server Discover and Monitoring engine, pass option { useUnifiedTopology: true } to the
MongoClient constructor.
(Use 'node --trace-warnings ...' to show where the warning was created)
MongoNetworkError: failed to connect to server [localhost:27017] on first connect [Error: connect ECONNREFUSED 127.0.0.1
:27017]
    at TCPConnectWrap.afterConnect [as oncomplete] (node:net:1606:16) {
  name: 'MongoNetworkError'
}
    at Pool.<anonymous> (/app/node_modules/mongodb/lib/core/topologies/server.js:441:11)
    at Pool.emit (node:events:520:28)
    at /app/node_modules/mongodb/lib/core/connection/pool.js:564:14
    at /app/node_modules/mongodb/lib/core/connection/pool.js:1000:11
    at /app/node_modules/mongodb/lib/core/connection/connect.js:32:7
    at callback (/app/node_modules/mongodb/lib/core/connection/connect.js:300:5)
    at Socket.<anonymous> (/app/node_modules/mongodb/lib/core/connection/connect.js:330:7)
    at Object.onceWrapper (node:events:635:26)
    at Socket.emit (node:events:520:28)
    at emitErrorNT (node:internal/streams/destroy:170:8)
    at emitErrorCloseNT (node:internal/streams/destroy:129:3)
    at process.processTicksAndRejections (node:internal/process/task_queues:82:21)
```

Il s'agit d'une erreur avec **MongoDb** ! Notre application tente de se connecter au serveur de base de données qui n'est pas inclus dans l'image.

Je vais lancer alors MongoCompass sur ma machine hôte et relancer le conteneur.

J'obtiens les mêmes messages d'erreurs !

Fichier : *code/networks-starting-setup/app.js*

```
mongoose.connect(
  'mongodb://localhost:27017/swfavorites',
  { useNewUrlParser: true },
  (err) => {
    if (err) {
      console.log(err);
    } else {
      app.listen(3000);
    }
  }
);
```

L'application, qui tourne dans le conteneur, tente de se connecter au serveur Mongo : **mongodb://localhost:27017/**. Etant cloisonnée dans le cadre du conteneur, **localhost** fait référence, dans l'application, au **localhost** du conteneur.

Pour le moment donc, notre conteneur est incapable de joindre le **localhost** et le port **27017** de la machine hôte !

Ouvrez donc le fichier **app.js** :

- Commentez les lignes 70 à 80. En Javascript, on met en commentaire un bloc de code avec le symbole : **/*** placé en début et ***/** placé à la fin du bloc.
- Ajoutez l'écoute de l'application sur le port 3000.

Fichier : code/networks-starting-setup/app.js

```
app.listen(3000);
/*
mongoose.connect(
  'mongodb://localhost:27017/swfavorites',
  { useNewUrlParser: true },
  (err) => {
    if (err) {
      console.log(err);
    } else {
      app.listen(3000);
    }
  }
);
*/
```

```
70  app.listen(3000);
71  /*
72  mongoose.connect(
73    'mongodb://localhost:27017/swfavorites',
74    { useNewUrlParser: true },
75    (err) => {
76      if (err) {
77        console.log(err);
78      } else {
79        app.listen(3000);
80      }
81    }
82  );
83  */
```

Reconstruisons l'image :

```
docker build -t favorites-node .
```

Créons de nouveau le conteneur :

```
docker container run --name favorites -d --rm -p 3000:3000 favorites-node
```

Vérifions :

```
docker container ps
```

Résultat :

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAME
6212cb109886	favorites-node	"docker-entrypoint.s..."	57 seconds ago	Up 56 seconds	0.0.0.0:3000->3000/tcp	favorites

L'application est fonctionnelle et testable ! Sauf bien entendu les deux routes qui nécessitent MongoDB :

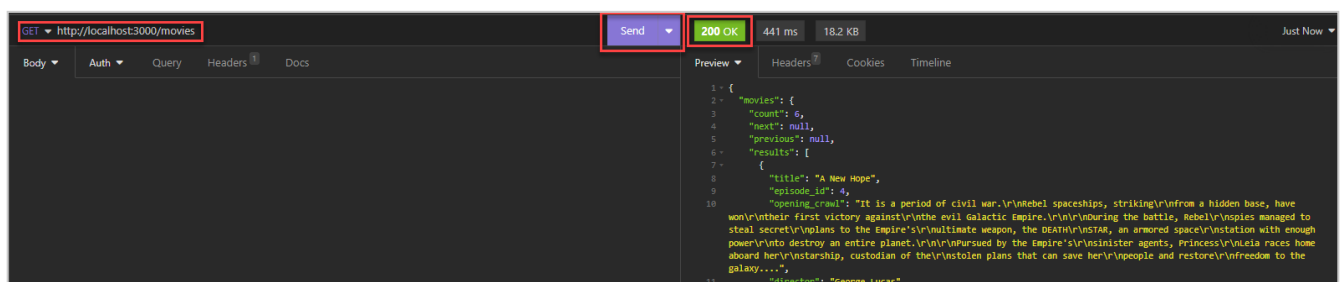
- GET /favorites
- POST /favorites

Mais les routes appelant seulement l'API externe sont testables :

- GET /movies
- GET /people

Ouvrons le logiciel **Insomnia** et testons la méthode GET sur l'URL :

<http://localhost:3000/movies>

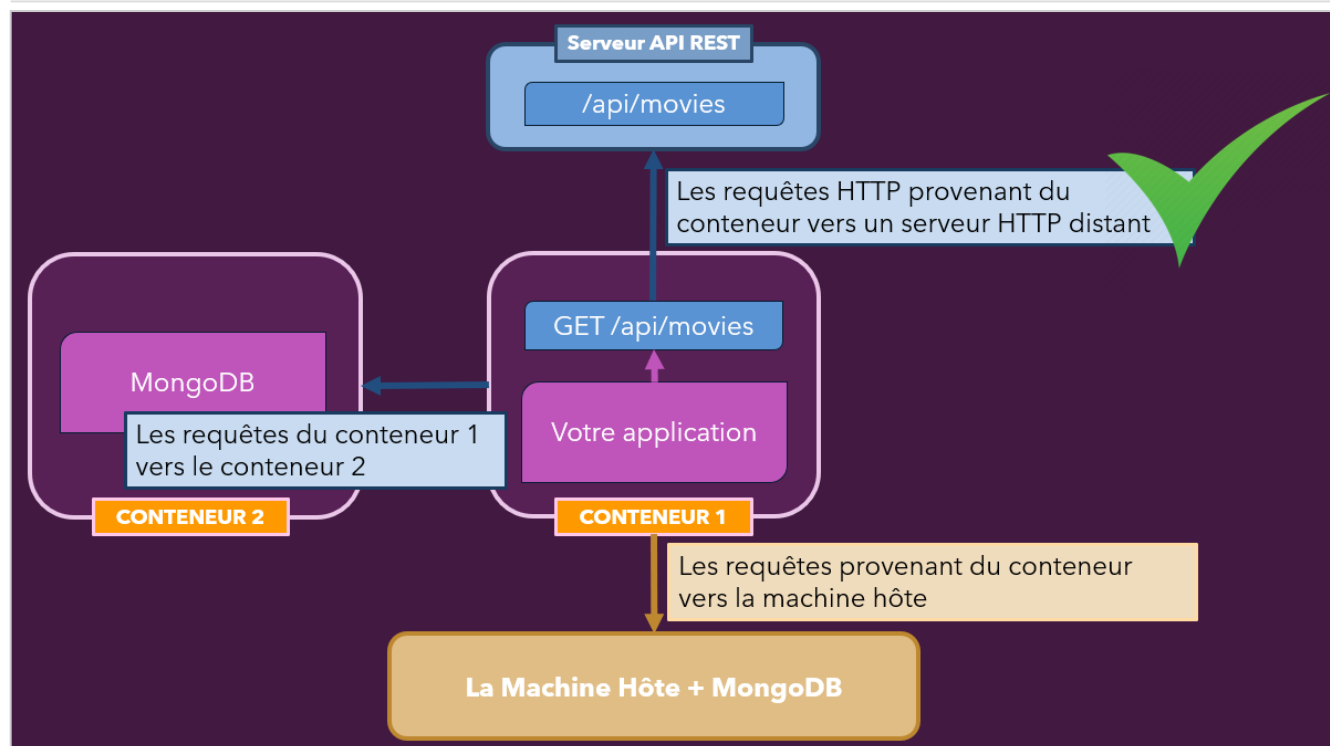


Nous constatons que notre conteneur communique parfaitement avec le **serveur HTTP distant** !



Il n'y a pas besoin de faire des modifications dans le code de l'application pour faire communiquer un **conteneur** avec une **API distante** par exemple !

A contrario, **il faudra surement faire quelques** modifications pour permettre **au conteneur de communiquer** avec le **localhost** de la machine hôte.



5. Mise en pratique : Faire communiquer le conteneur avec le localhost

Nous allons faire communiquer le service MongoDB qui tourne sur la machine hôte avec le conteneur.

Pour cela supprimons les modifications apportées précédemment dans le fichier `app.js` :

Fichier : `code/networks-starting-setup/app.js`

```
mongoose.connect(
  'mongodb://localhost:27017/swfavorites',
  { useNewUrlParser: true },
  (err) => {
    if (err) {
      console.log(err);
    } else {
      app.listen(3000);
    }
  }
);
```

La ligne qui pose un problème est la suivante :

`'mongodb://localhost:27017/swfavorites'` Plus précisément, le nom de domaine `localhost`.

Comme nous l'avons déjà mentionné, lorsque `localhost` est lu par l'application et par le moteur de traduction d'adresse IP de Docker, il est compris comme faisant référence à l'adresse IP du conteneur.

Or, nous voulons qu'il soit interprété différemment ! Nous voulons que `localhost` pointe vers l'adresse IP de notre machine hôte.

Heureusement, il existe une instruction spéciale comprise par Docker qui permet de remplacer le terme `localhost` dans notre code par `host.docker.internal`, ce qui permettra de cibler l'adresse IP de la machine hôte.

Fichier : `code/networks-starting-setup/app.js`

```
mongoose.connect(
  'mongodb://host.docker.internal:27017/swfavorites',
  { useNewUrlParser: true },
  (err) => {
    if (err) {
      console.log(err);
    } else {
      app.listen(3000);
    }
  }
);
```

```
);
```

Nous devons maintenant reconstruire l'image encore une fois pour que les modifications soient prises en compte.

Reconstruisons l'image :

```
docker build -t favorites-node .
```

Créons de nouveau le conteneur :

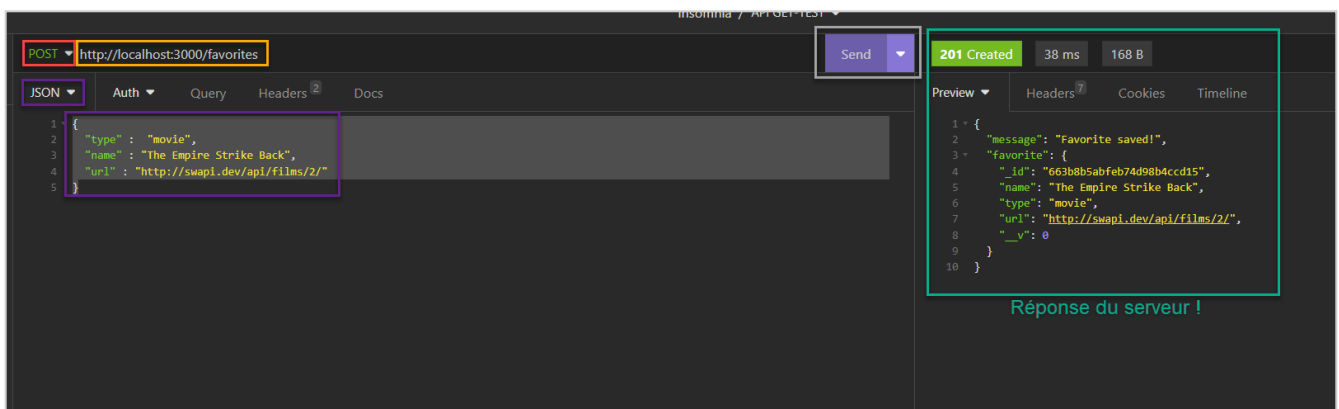
```
docker container run --name favorites -d --rm -p 3000:3000 favorites-node
```

Nous allons maintenant tester en postant un favori :

- Ouvrez **Insomnia** :
- Modifiez la requête HTTP en POST
- Ajouter l'URL : <http://localhost:3000/favorites>
- Modifiez le type du BODY de la requête en **JSON**
- ajoutez la structure **JSON** suivante :

```
{  
  "type" : "movie",  
  "name" : "The Empire Strike Back",  
  "url" : "http://swapi.dev/api/films/2/"  
}
```

- Appuyez sur "Send"

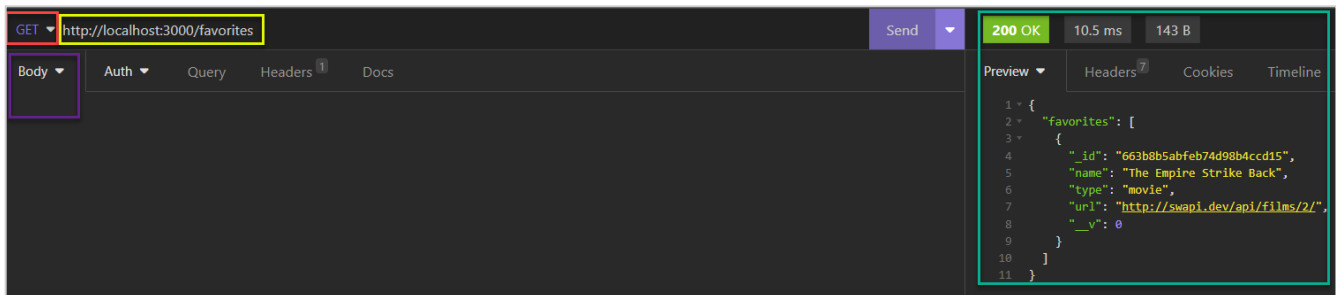


Cela fonctionne maintenant parfaitement !

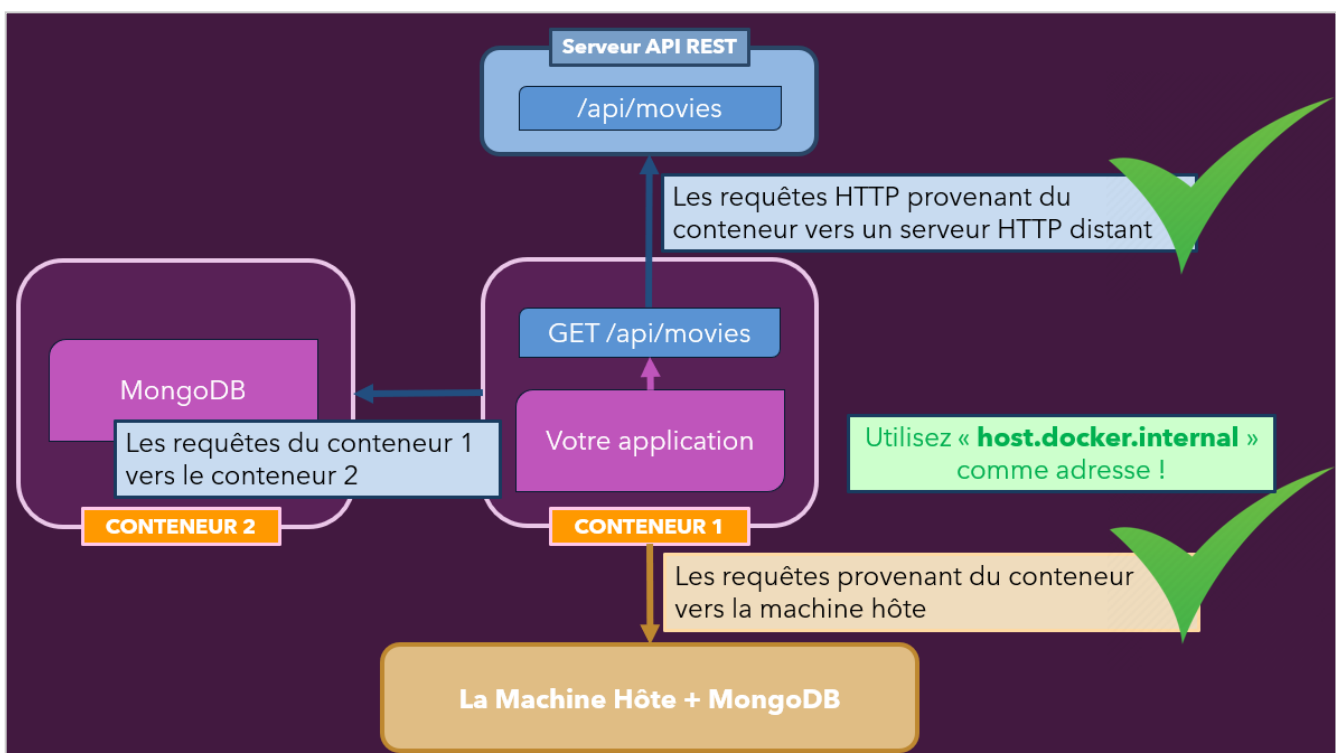
Nous pouvons même récupérer le favori ajouté :

- Modifiez la requête HTTP en GET

- Laissez l'URL : <http://localhost:3000/favorites>
- Modifiez le type du BODY en "No Body"
- Appuyez sur "Send"



Nous avons réussi à faire communiquer notre conteneur avec la machine hôte ! Mais il faut penser à modifier légèrement notre code !



Toutefois, cela n'est pas la solution idéale. Dans notre situation, il faudra créer un conteneur avec MongoDB et s'arranger pour qu'il puisse communiquer avec le conteneur de l'application !

6. Mise en pratique : Communication entre conteneurs [Solution basique]

Premièrement, nous créerons un nouveau conteneur basé sur une image officielle de MongoDB :

```
docker container run -d --name mongodb mongo
```

Maintenant que le conteneur est lancé, comment faire communiquer notre application du conteneur `favorites` avec MongoDB présent dans le conteneur "mongodb" ?

Nous savons déjà qu'il faudra éditer le fichier `app.js` et la ligne contenant cette chaîne de connexion :

```
mongodb://host.docker.internal:27017/swfavorites,
```

`host.docker.internal` peut être remplacé par l'adresse IP de `mongodb`, que l'on peut connaître en inspectant la configuration du conteneur :

```
docker container inspect mongodb
```

La commande retourne un fichier JSON organisant les données du conteneur.

Nous n'avons plus qu'à chercher la clé `IPAddress` !

Le fichier de configuration est long à parcourir. Nous allons modifier un peu la commande et intégrer un filtre.

`IPAddress` se trouve dans la catégorie : `NetworkSettings`, cela est pratique de le savoir, car nous allons pouvoir formater la réponse en n'affichant que la clé qui nous intéresse.

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' mongodb
```

L'adresse IP retournée est : `172.17.0.3`

```
PS C:\Users\baptiste\Downloads\APP> docker inspect --format '{{ .NetworkSettings.IPAddress }}' mongodb
172.17.0.3
```

Modifions maintenant notre fichier `app.js` de la sorte :

```
// some code before
mongoose.connect(
  'mongodb://172.17.0.3:27017/swfavorites',
  { useNewUrlParser: true },
  (err) => {
    if (err) {
```

```
        console.log(err);  
    } else {  
        app.listen(3000);  
    }  
}  
);
```

Pour tester, stoppons le conteneur **favorites**, et reconstruisons l'image.

```
docker container stop favorites
```

Puis :

```
docker build -t favorites-node .
```

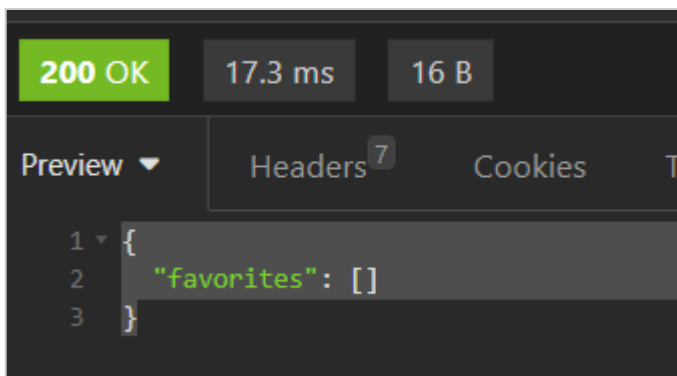
Lançons cette nouvelle version :

```
docker container run --name favorites -d --rm -p 3000:3000 favorites-node
```

Vérifions si nos deux conteneurs sont bien lancés :

```
docker ps
```

Et maintenant avec Insomnia, testons :



Le résultat est correcte puisque nous avons une nouvelle installation de MongoDB et nous n'avons pas encore inséré de données.

Nos deux conteneurs communiquent et échangent des informations ! Nous sommes arrivés à nos fins.

Mais cette solution est tout sauf pratique !

- Il faut chercher l'adresse IP du conteneur soit même.
- Quand l'adresse IP de MongoDB change, il faut recréer une image de l'application.

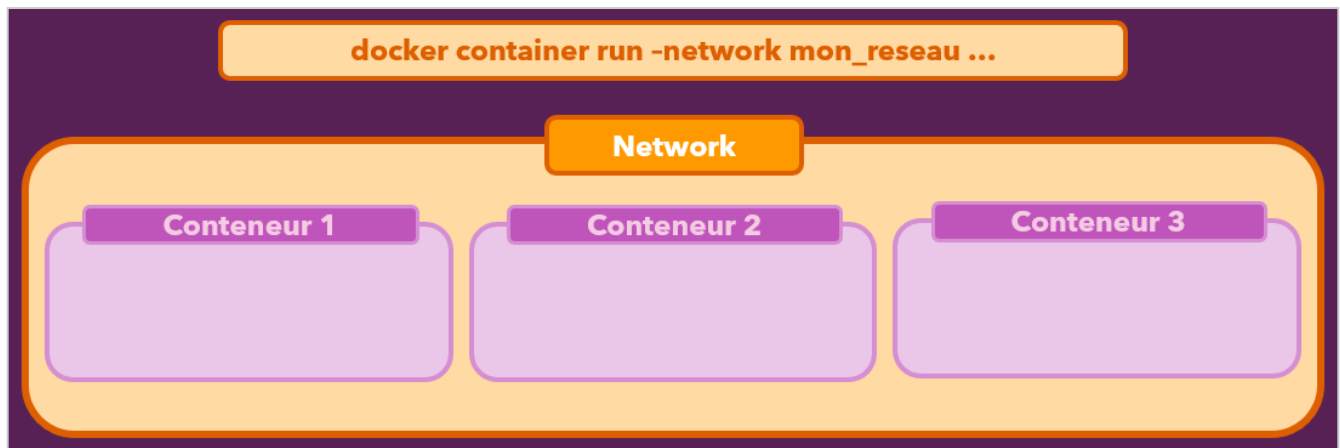
Voyons maintenant une manière élégante de procéder !

7. Mise en pratique : Les Networks Docker [Solution élégante]

Avec Docker, nous pouvons créer des réseaux appelés en anglais : **Networks**.

Lorsque nous avons plusieurs conteneurs qui vont devoir communiquer, nous allons pouvoir les réunir au sein d'un même réseau grâce au paramètre `--network` .

Docker ne crée pas automatiquement les réseaux quand nous montons plusieurs conteneurs. Il faut le spécifier manuellement.



Dans un **réseau interne Docker**, tous les conteneurs peuvent communiquer. Les adresses IP sont automatiquement résolues, c'est-à-dire **qu'elles sont attribuées et connues de tous les membres du réseau**.

Mettons en place un réseau pour les conteneurs de notre application.

Stoppons d'abord les conteneurs :

```
docker container stop favorites
```

puis

```
docker container stop mongodb
```

Et supprimons tous les conteneurs arrêtés :

```
docker container prune
```

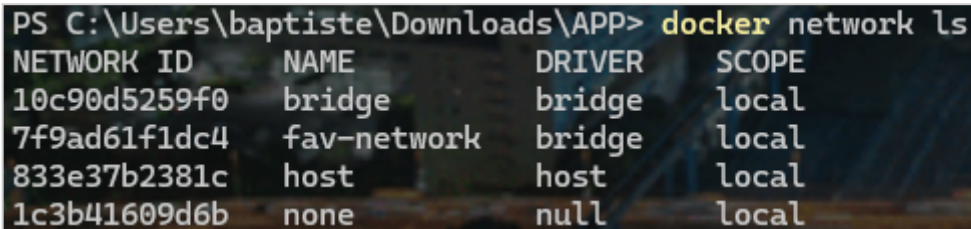
Maintenant voyons comment nous allons pouvoir intégrer notre premier conteneur `mongodb` dans un réseau nommé arbitrairement `fav-network` :

Il faut créer le réseau en lui-même avec la commande :

```
docker network create fav-network
```

Par curiosité, nous pouvons lister l'ensemble des réseaux existant sur notre Docker :

```
docker network ls
```



```
PS C:\Users\baptiste\Downloads\APP> docker network ls
NETWORK ID      NAME          DRIVER  SCOPE
10c90d5259f0    bridge       bridge  local
7f9ad61f1dc4    fav-network   bridge  local
833e37b2381c    host         host    local
1c3b41609d6b    none         null    local
```

Ensuite, créons un autre conteneur, en l'intégrant dans le nouveau réseau :

```
docker container run -d --name mongodb --network fav-network mongo
```



Mongodb est inclus dans un réseau Docker et sera utilisé par un autre conteneur. Vous remarquerez qu'il n'y a pas besoin ici d'exposer un port de mongodb vers une quelconque sortie.

Comme Mongodb sera dans le même réseau que notre application, le lien via l'adresse IP et le PORT du service sera automatiquement réalisé par DOCKER !

La prochaine étape sera de monter le conteneur de notre application de la même manière ! Mais il faut modifier le code source pour permettre la communication entre notre application et le serveur mongodb.

Il faut trouver une solution pour récupérer l'adresse IP du conteneur **mongodb** automatiquement.

Comme les conteneurs font partie du **même réseau Docker**, nous pouvons remplacer l'adresse IP du conteneur par le nom de ce conteneur: **mongodb** dans notre cas.

```
// some code before
mongoose.connect(
  'mongodb://mongodb:27017/swfavorites',
  { useNewUrlParser: true },
  (err) => {
    if (err) {
      console.log(err);
    } else {
      app.listen(3000);
    }
  }
);
```


Docker va alors automatiquement traduire ce nom de conteneur par la valeur de son adresse IP.

Faisons les modifications dans notre code, et montons une nouvelle image :



```
mongoose.connect(  
  'mongodb://mongodb:27017/swfavorites',  
  { useNewUrlParser: true },  
  (err) => {  
    if (err) {  
      console.log(err);  
    } else {  
      app.listen(3000);  
    }  
  }  
);
```

```
docker build -t favorites-node .
```

Lançons cette nouvelle version :

```
docker container run --name favorites --network fav-network -d --rm -p 3000:3000  
favorites-node
```

Nous pouvons vérifier si le processus s'est bien déroulé :

```
docker ps
```

La commande devrait montrer les deux conteneurs démarrés.

Puis les **logs**, qui nous indiqueraient si des problèmes de liaison entre les conteneurs du réseau seraient apparus.

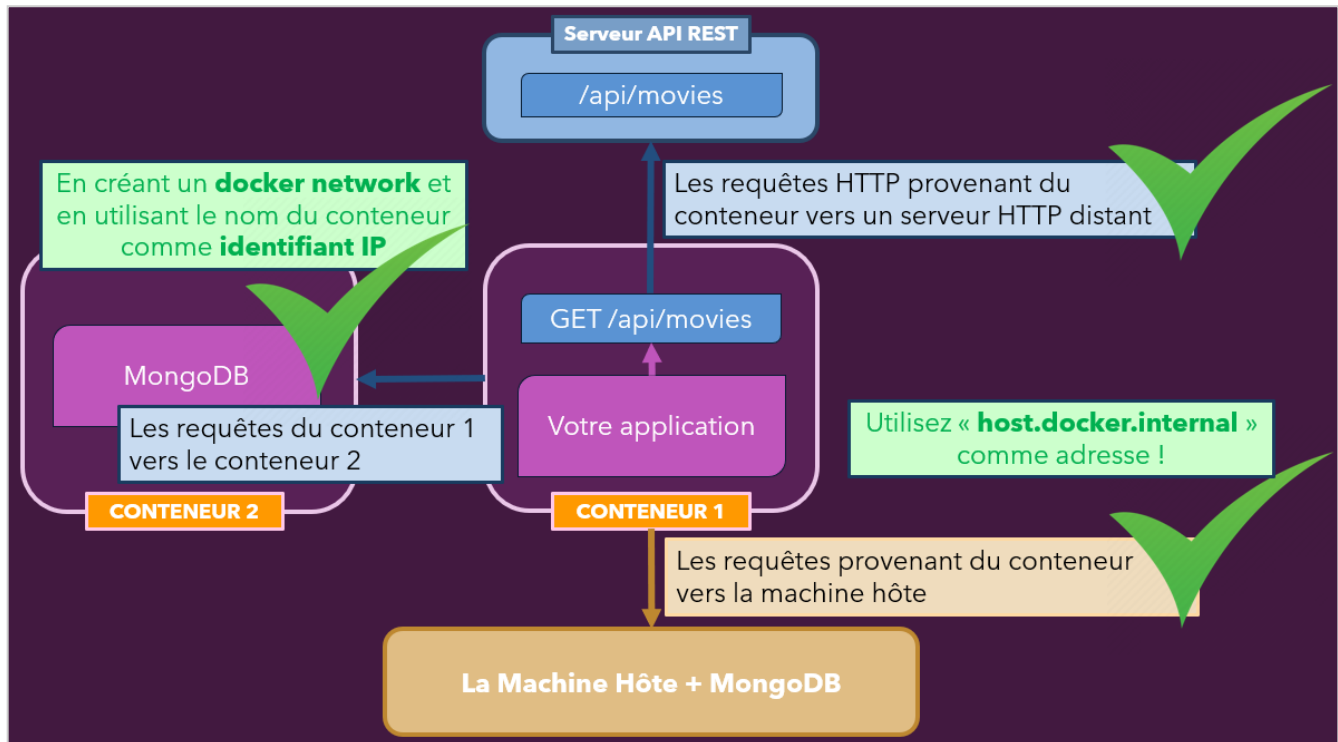
```
docker logs
```

Et au final, avec Insomnia, nous pouvons tester les routes !

En conclusion :

- Dans une communication de conteneur à conteneur, il faut créer un réseau, puis le **Docker Engine** se charge de faire la liaison automatiquement **sans que l'on ait à se soucier de l'exposition des ports**.

- L'**exposition des ports** n'est à réaliser que dans le cadre d'une communication entre un conteneur et la machine hôte.



Docker ne modifie pas le code source de notre application en remplaçant le nom de domaine par l'adresse IP du conteneur. Cependant, Docker contrôle l'environnement réseau et reçoit les requêtes HTTP entrantes et sortantes des conteneurs. Lorsqu'un conteneur envoie une requête au serveur mongodb, Docker tente de résoudre le nom de domaine du destinataire. Si le nom de domaine est remplacé par une adresse IP, Docker enverra la requête à cette adresse. Si le nom de domaine est un nom de conteneur, qui agit comme un nom de domaine local, Docker cherchera son adresse IP pour envoyer la requête.

8. Les pilotes (drivers) de Docker Network

Les réseaux Docker prennent en charge différents types de "pilotes" ou "**drivers**" en anglais, qui influencent le comportement du réseau.

Le pilote par **défaut** est le pilote "**bridge**"

- Il fournit le comportement présenté dans ce module, (c'est-à-dire que les conteneurs peuvent se trouver les uns les autres par nom s'ils se trouvent dans le même réseau).

Le pilote peut être défini lorsqu'un réseau est créé, simplement en ajoutant l'option `--driver`.

```
docker network create --driver bridge mon_reseau
```

Bien sûr, si vous souhaitez utiliser le pilote "bridge", vous pouvez simplement omettre l'option entière car "bridge" est le pilote par défaut.

Docker prend également en charge les pilotes alternatifs suivants - bien que vous utilisiez le pilote "bridge" dans la plupart des cas :

- **host** : pour les conteneurs autonomes, l'isolement entre le conteneur et le système hôte est supprimé (c'est-à-dire qu'ils partagent localhost en tant que réseau)
- **overlay** : plusieurs démons Docker (c'est-à-dire Docker en cours d'exécution sur différentes machines) peuvent se connecter les uns aux autres. Fonctionne uniquement en mode "Swarm" qui est un moyen obsolète / presque obsolète de connecter plusieurs conteneurs
- **macvlan** : vous pouvez définir une adresse MAC personnalisée pour un conteneur - cette adresse peut ensuite être utilisée pour la communication avec ce conteneur
- **none** : toute la mise en réseau est désactivée.
- **Third-party plugins** : vous pouvez installer des plugins tiers qui peuvent alors ajouter toutes sortes de comportements et de fonctionnalités

Comme mentionné, le pilote "**bridge**" a le plus de sens dans la grande majorité des scénarios.