

Dans cet atelier, nous allons utiliser les résultats obtenus lors de l'atelier précédent. Ensuite, et en construisant une API de ce modèle avec FastAPI, nous allons essayer de le déployer en utilisant deux types d'architectures de déploiement d'un modèle ML :

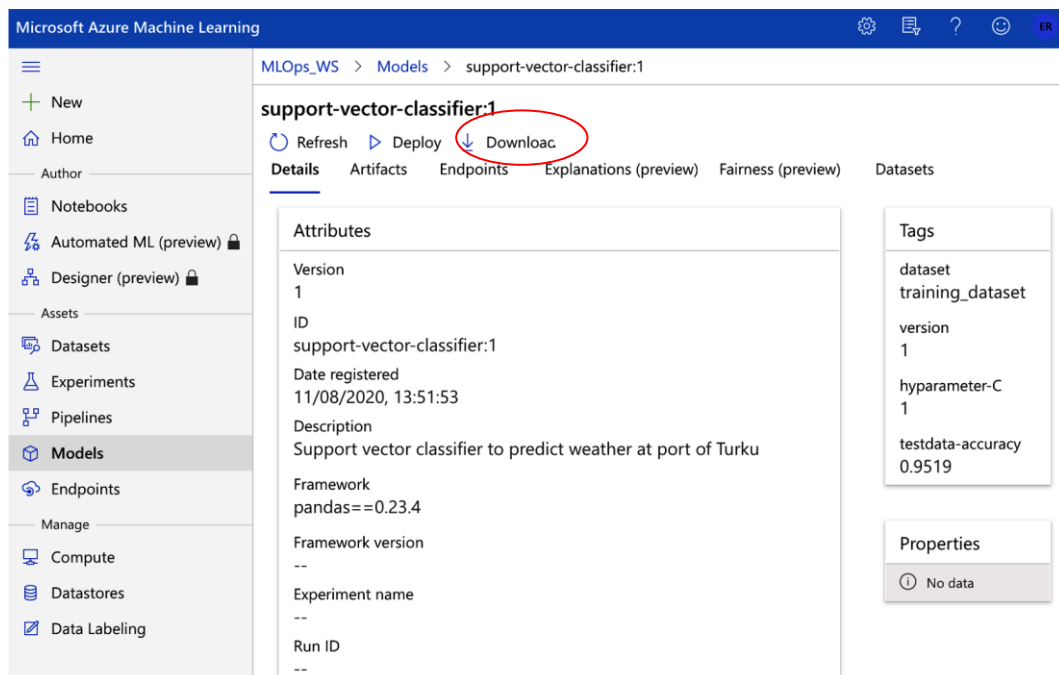
- Un service dédié au modèle via une API Rest
- Une API ML dédié en tant que microservice

Pour ce faire, nous allons utiliser plusieurs cibles d'inférence en tirant profit de plusieurs technologies cloud (PaaS et IaaS) :

- Heroku
- Microsoft azure à travers Azure container instances

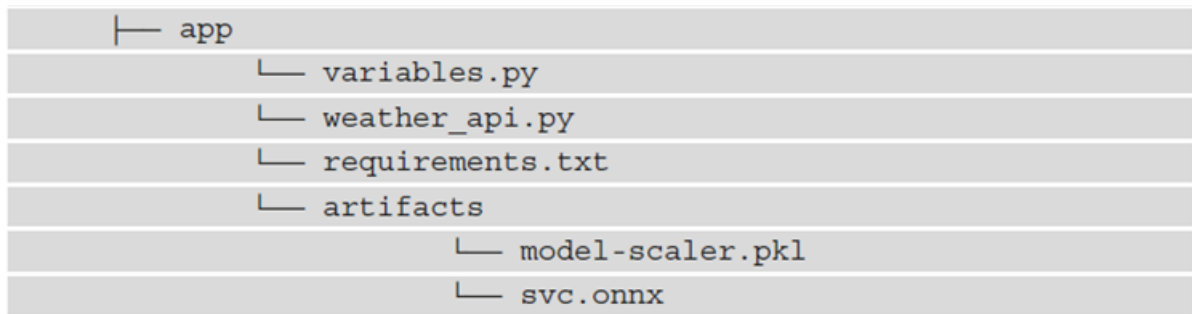
1. Construction de l'API et déploiement en tant que service

Dans l'atelier précédent, nous avons pu exécuter un pipeline pour produire un ensemble de modèles en opérant sur notre dataset « weather_data.csv ». Par ailleurs, les modèles produits peuvent être téléchargés sur « Microsoft Azure ML Studio ».



Nous allons utiliser ces modèles pour produire une API. Dans le dossier de l'atelier, ouvrez le dossier « API_Microservices ».

Dans ce dossier, vous allez trouver un ensemble de fichiers et dossiers comme le montre la figure ci-dessous :



- Variables.py

Dans ce fichier nous avons défini les entrées de notre API en utilisant la bibliothèque Pydantic.

En utilisant Pydantic, nous avons créé la classe « WeatherVariables » qui sera utilisée à son tour pour la validation des variables de notre API.

- Weather_api.py

C'est dans ce fichier qu'on va définir le service fastAPI. Les artefacts nécessaires sont importés et utilisés pour que le modèle puisse faire l'inférence.

- Requirements.txt

Ce fichier contient tous les packages requis pour exécuter notre service.

```

numpy
fastapi
uvicorn
pydantic
pickle5
scikit-learn==1.0.1
pandas
onnx
onnxruntime
  
```

Pour exécuter ce fichier, tapez la commande suivante en étant dans le dossier « app » :

```
uvicorn weather_api:app --reload
```

FastAPI utilise l'OpenAPI (en savoir plus : <https://www.openapis.org/>, <https://swagger.io/specification/>) Spécification pour servir le modèle. La spécification OpenAPI (OAS) est une interface standard indépendante du langage pour les API RESTful. En utilisant les fonctionnalités OpenAPI, nous pouvons accéder à la documentation de l'API et obtenir une vue d'ensemble de l'API. Vous pouvez accéder à la documentation de l'API et tester l'API à 127.0.0.1:8000/docs et cela vous dirigera vers une interface utilisateur basée sur Swagger « Swagger-based UI » (elle utilise l'OAS) pour tester votre API.

Pour voir la documentation interactive de l'application allez sur <http://127.0.0.1:8000/docs> . Plus que ça, vous pouvez tester avec d'autre valeurs sur le formulaire fournit grâce à Swagger UI.

Maintenant que nous avons exécuté notre ML API déployée localement, nous allons la déployer dans un PaaS à savoir Heroku.

Pour ce faire, ajoutez un fichier nommé « Procfile » dans le dossier de l'atelier (déjà présent dans le dossier) dont le contenu est le suivant :

```
web: uvicorn weather_api:app --host 0.0.0.0 --port $PORT
```

Les étapes de déploiement sont les suivantes :

1. Commencez par la création d'un compte sur <https://www.heroku.com>
2. Installez Git : <https://git-scm.com/downloads>
3. Installez Heroku CLI : <https://devcenter.heroku.com/articles/heroku-cli>, sur l'invite de commande tapez la commande suivante pour s'assurer que Heroku CLI est installé :

```
heroku --version
```

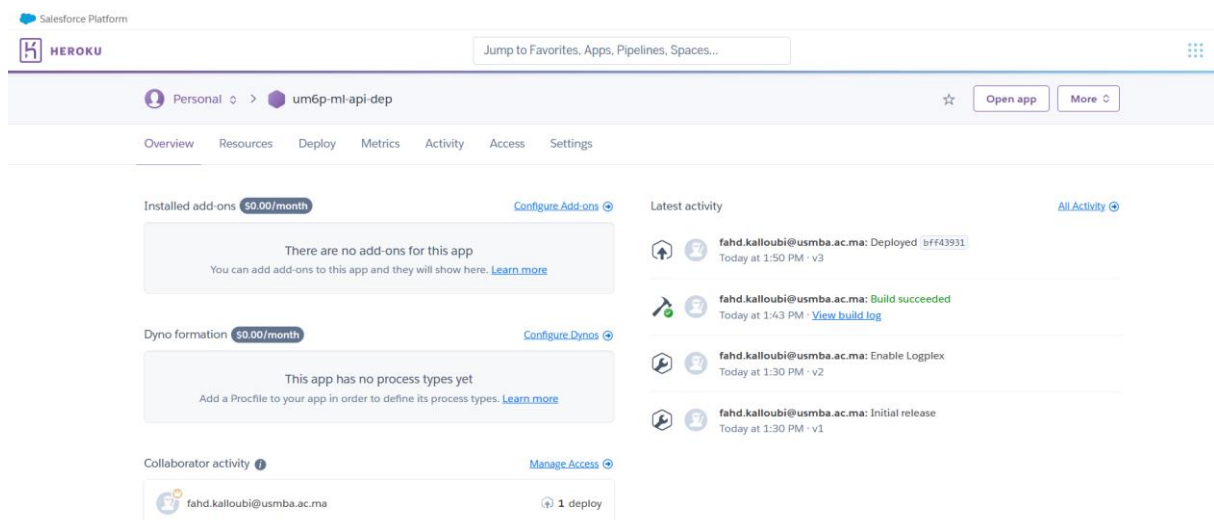
4. En étant dans le dossier « app », lancez la commande suivante pour créer une application Heroku (choisissez votre propre nom)

```
// en lançant heroku create, heroku se chargera de choisir un nom aléatoire  
// choisissez un autre nom de votre choix  
heroku create um6p-ml-api-dep
```

5. Lancez les commandes suivantes une après une afin de pouvoir charger notre code vers Heroku

- Git init
- heroku git:remote -a um6p-ml-api-dep
- git add .
- git commit -am "ML API um6p "
- git push heroku master

Sur le dashboard heroku, cliquez sur votre application ainsi créée.



Vous devez attendre que Heroku installe toutes les dépendances requises. Par la suite, et une fois le statut change vers « build succeeded », cliquez sur le bouton « open app » ou bien en ligne de commande tapez « heroku open ».

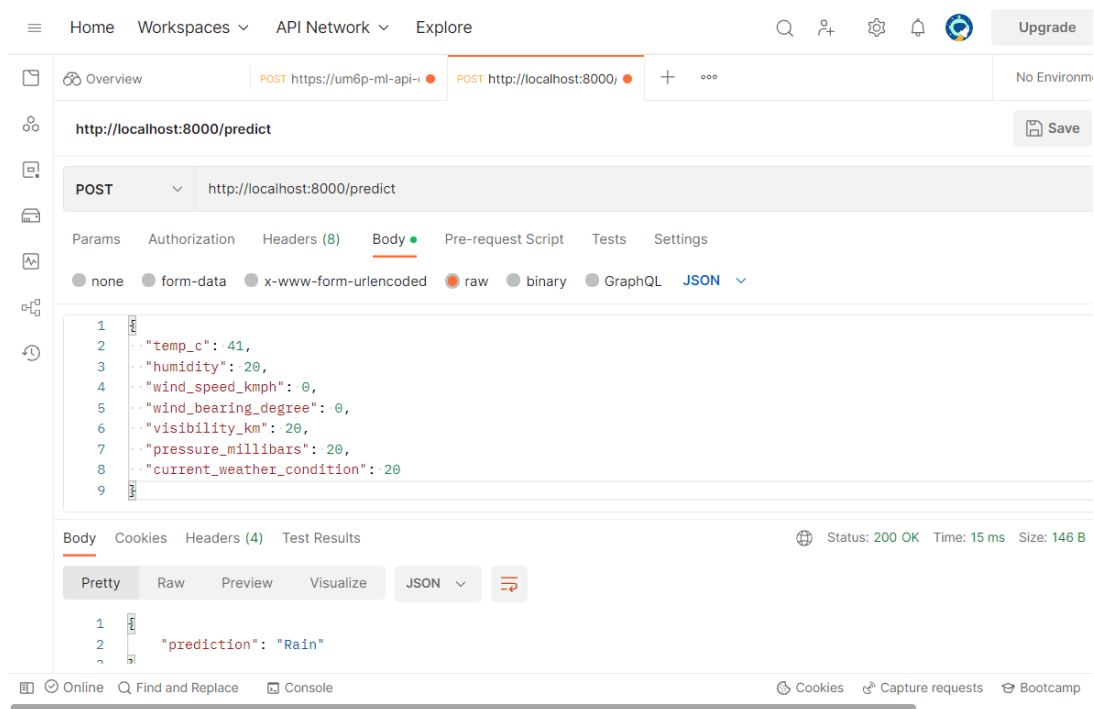


Nous avons bel et bien déployé notre service (i.e., API) sur le PaaS « Heroku » que nous pouvons le consommer soit en utilisant Swagger UI en visitant : <https://um6p-ml-api-dep.herokuapp.com/docs> ou bien via une application dédiée.

N.B : Malheureusement, le PaaS Heroku est devenu payant, alors nous allons utiliser le service déployé localement (i.e., <http://localhost:8000>)

1.1 Consommons notre API en utilisant Postman

Nous allons essayer maintenant un outil pour interroger notre API à l'aide des requêtes curl. Postman est une interface graphique facile à utiliser (lien de téléchargement : <https://www.postman.com/downloads/>).



1.2 Consommons notre API en utilisant une application dédiée

Comme flask, FastAPI dispose aussi du même moteur de template jinja2. De ce fait, nous allons créer une application basée sur FastAPI afin de consommer notre service à travers une interface Html conviviale.

Ouvrez le dossier « app_consuming ».

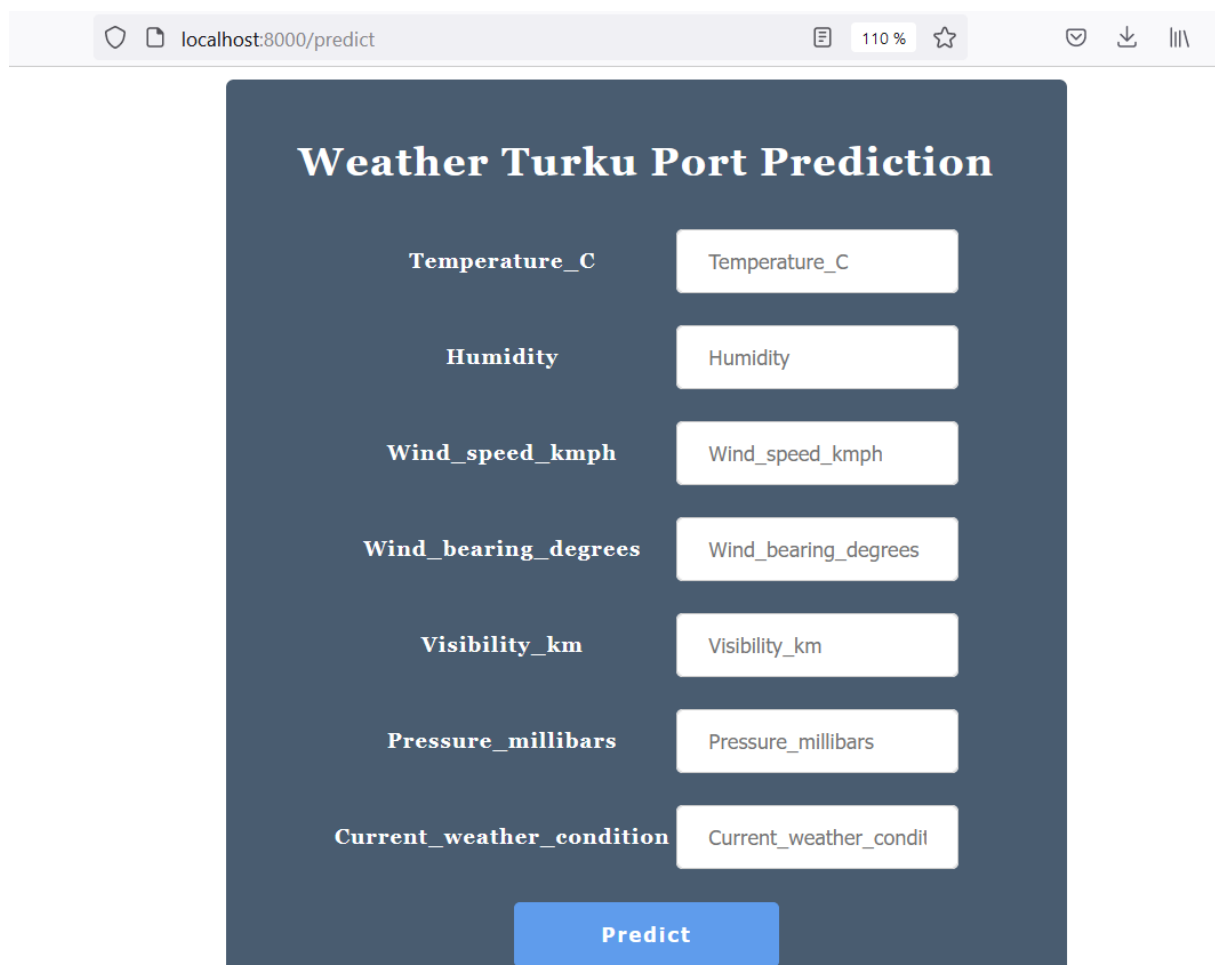
Installez les dépendances en lançant la commande : `pip install -r requirements.txt`

Comme vous le remarquez, le contenu du fichier « `weather_app.py` » est le même que celui de « `weather_api.py` » sauf que nous récupérons le contenu à partir d'un formulaire Html situé dans la page « `Template/index.html` ». Par ailleurs, ce contenu est envoyé ensuite en utilisant la bibliothèque « `Requests` » vers notre API « `https://um6p-ml-api-dep.herokuapp.com` ou `http://localhost:8000` ».

Pour exécuter l'application (en étant dans le dossier « `app_consuming` » :

```
uvicorn weather_app:app --reload --port 5000
```

Pour tester l'application, ouvrez <http://localhost:8000/>. Essayez d'introduire quelques valeurs afin de tester le bon fonctionnement du modèle.



The screenshot shows a web browser window with the address bar displaying 'localhost:8000/predict'. The page content is a form titled 'Weather Turku Port Prediction' on a dark blue background. The form consists of seven input fields, each with a label to its left: 'Temperature_C', 'Humidity', 'Wind_speed_kmph', 'Wind_bearing_degrees', 'Visibility_km', 'Pressure_millibars', and 'Current_weather_condition'. Each input field is a white box with its respective label inside. At the bottom of the form is a blue button labeled 'Predict'.

2. Déployer une API ML en tant que microservice

Maintenant, nous allons packager le service FastAPI de manière standardisée à l'aide de Docker. De cette façon, nous pouvons déployer l'image ou le conteneur Docker sur la cible de déploiement de notre choix.

Nous avons créé un fichier « Dockerfile » dans lequel nous avons utilisé une image officielle de fastAPI (tiangolo/uvicorn-gunicorn-fastapi) à partir de Docker Hub.

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7
COPY ./app /app
RUN pip install -r requirements.txt
EXPOSE 80
CMD ["uvicorn", "weather_api:app", "--host", "0.0.0.0", "--port", "80"]
```

Tout d'abord, nous utilisons une image officielle fastAPI Docker de Docker Hub en utilisant la commande FROM et en pointant sur l'image - tiangolo/uvicorn-gunicornfastapi:python3.7. L'image utilise Python 3.7, qui est compatible avec fastAPI. Ensuite, nous copions le dossier app dans un répertoire nommé app à l'intérieur de l'image/conteneur docker. Une fois le dossier app copié dans l'image/le conteneur Docker, nous installerons les packages nécessaires répertoriés dans le fichier requirements.txt à l'aide de la commande RUN.

Ensuite, nous EXPOSErons le port 80 pour l'image/le conteneur Docker. Enfin, nous allons faire tourner le serveur à l'intérieur de l'image/du conteneur Docker à l'aide de la commande CMD « uvicorn weather_api:app --host 0.0.0.0 --port 80 ». Cette commande pointe vers le fichier weather_api.py pour accéder à l'objet app de fastAPI et héberger le service sur le port 80 de l'image/du conteneur.

2.1 Exécutons notre API en tant que conteneur local

Pour tester notre API, nous allons suivre les étapes suivantes :

1. Nous allons commencer par la construction de notre image Docker. Pour ce faire, vous devez avoir Docker installé. En étant dans le dossier « API_Microservices », exécutez la commande suivante pour construire l'image Docker.

```
docker build -t fastapi .
```

Suites à l'exécution de cette commande nous allons procéder à la construction de notre image Docker en suivant les étapes prescrites dans le Dockerfile. Pour vérifier la création de notre image nous pouvons lancer la commande « docker images ».

```

C:\Windows\System32\cmd.exe
=> => extracting sha256:aebd27b2d86a5a3a2cbe186247911047a7e432b9d17daad8f226597c0ea4276 1.3s
=> => extracting sha256:3b81d4ff756ff433c0aa6bdd25335679c0e5f7baafba663151e57610912b564 2.9s
=> => extracting sha256:6e4f59a23b58b034aca9e284e24eb048a5bd8a08dfaf009cfd664804b5422e06 0.0s
=> => extracting sha256:5c6b61eb416d184070e2a3179d0fbf5195a965fb1a4051ba4ac497a42f711002 0.9s
=> => extracting sha256:41575650c078c48027e842a05cc34353cd2faee9c229c8353b56c67e72ab36c 0.0s
=> => extracting sha256:82af7648a68a3cdd74a95f3b8e467a68ad164374f726184cdedca6369af31c1c 1.7s
=> => extracting sha256:697e3cdf6fdb54685dd4b6ef30199efb83c70d5ba9060cd7beae411eef6c0664 0.0s
=> => extracting sha256:8d414b25a011a87ef0532614a30e5bfa150ccb1f6307ee8c9cea3d4dbc5be59 0.0s
=> => extracting sha256:3e3cfa737f563f22f21cab6f758afa05a8095c325ffbd1ce11060e1f4f51aa0 0.0s
=> => extracting sha256:1595db7eb9b17f8f89289166cfad8ff9dc7e39776bd627408ba5816d062c87da 0.0s
=> => extracting sha256:d2d3fcd4b870b1af2eb777f132032774d461109aa700325a3dcb351bc9aa14e6 0.0s
=> => extracting sha256:35f87fd78dc8b33cd8339a58d1da5f75fe2fd8c85478891e5e70e02d793e4c42 0.0s
=> => extracting sha256:146e266752fc4dc9c2073f27ca8769e23189df00d9b57b56aa59340e90b67a0 0.0s
=> => extracting sha256:57df244389c008c329cd8ee89a74f8f36e3984715e65607caca22c30d21efe83 2.1s
=> => extracting sha256:e8391aa94779ce81d31831f53777c61b634a0e7df78be5549e41edf932679f48 0.0s
=> [2/3] COPY ./app /app 0.3s
=> [3/3] RUN pip install -r requirements.txt 97.4s
=> exporting to image 12.5s
=> exporting layers 12.4s
=> writing image sha256:c4ec509ea1e18277d0d821449c2df4bd86f82ee2115580b93e61dbb0b5f29e28 0.0s
=> naming to docker.io/library/fastapi 0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

C:\migration\Cours UM6P\MLops DataOps\Labs\Lab 6\API_Microservices>docker images
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
fastapi              latest          c4ec509ea1e1   22 seconds ago 1.37GB
docker/getting-started latest          26d80cd96d69   10 days ago    28.5MB

C:\migration\Cours UM6P\MLops DataOps\Labs\Lab 6\API_Microservices>

```

2. Nous allons exécuter un container localement en exécutant la commande suivante :

```
docker run -d -p 80:80 --name weathercontainer fastapi
```

Pour afficher l'image créé « weathercontainer », vous pouvez lancer la commande « docker container ps --all ». Toutefois, vous pouvez voir que le service est servi sur le port 80.

```

Administrateur : Invite de commandes
C:\migration\Cours UM6P\MLops DataOps\Labs\Lab 6\API_Microservices>docker build -t fastapi .
[+] Building 109.0s (8/8) FINISHED
=> [internal] load build definition from Dockerfile 0.1s
=> => transferring dockerfile: 225B 0.0s
=> [internal] load .dockerignore 0.1s
=> => transferring context: 2B 0.0s
=> [internal] load metadata for docker.io/tingolo/uvicorn-gunicorn-fastapi:python3.7 20.7s
=> CACHED [1/3] FROM docker.io/tingolo/uvicorn-gunicorn-fastapi:python3.7@sha256:f5770422a8875fe3d677e1bda724ee 0.0s
=> [internal] load build context 0.1s
=> => transferring context: 436B 0.0s
=> [2/3] COPY ./app /app 0.2s
=> [3/3] RUN pip install -r requirements.txt 83.3s
=> exporting to image 4.3s
=> exporting layers 4.2s
=> writing image sha256:33d9c2c6ff3f9695556aa90765d0fc72ea91a7a6dd5c3ea4c4e7496b945e3028 0.0s
=> naming to docker.io/library/fastapi 0.0s

C:\migration\Cours UM6P\MLops DataOps\Labs\Lab 6\API_Microservices>docker run -d -p 80:80 --name weathercontainer fastapi
bb249c128052b86b9b4af07d0ae02fa3ff63f8cdb529b91e65b831b785c83eec

C:\migration\Cours UM6P\MLops DataOps\Labs\Lab 6\API_Microservices>docker container ps --all
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                    NAMES
bb249c128052   fastapi        "uvicorn weather_api..." 22 seconds ago Up 21 seconds  0.0.0.0:80->80/tcp       weathercontainer

C:\migration\Cours UM6P\MLops DataOps\Labs\Lab 6\API_Microservices>docker images
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
fastapi              latest          33d9c2c6ff3f   About an hour ago 1.46GB

C:\migration\Cours UM6P\MLops DataOps\Labs\Lab 6\API_Microservices>

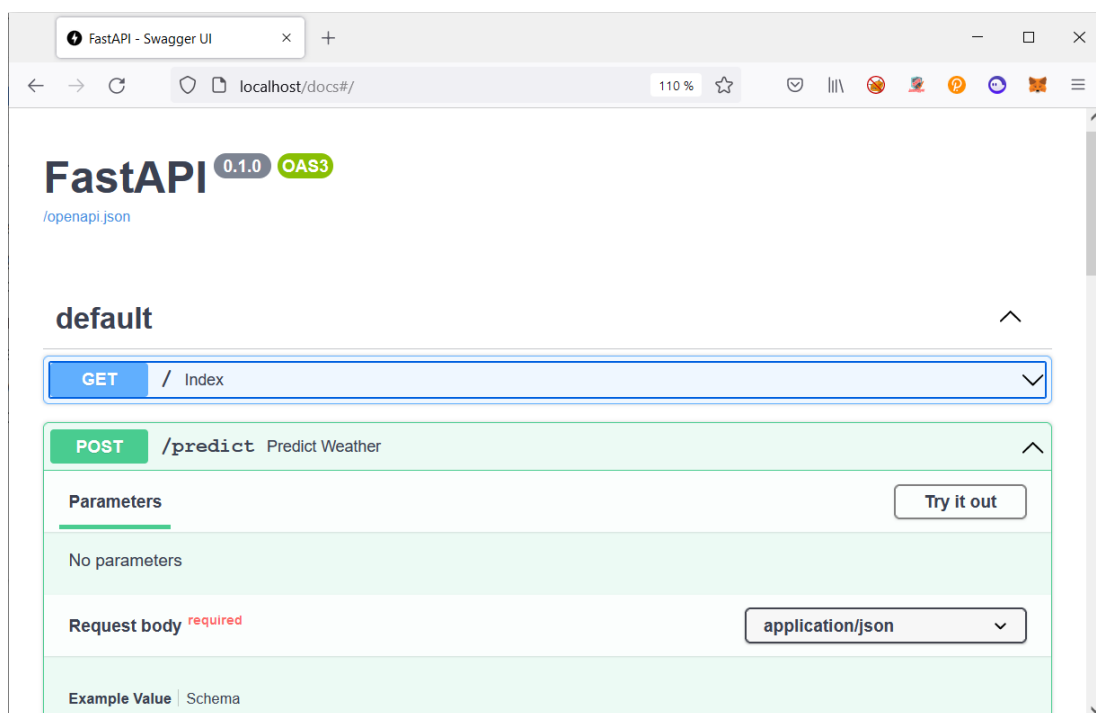
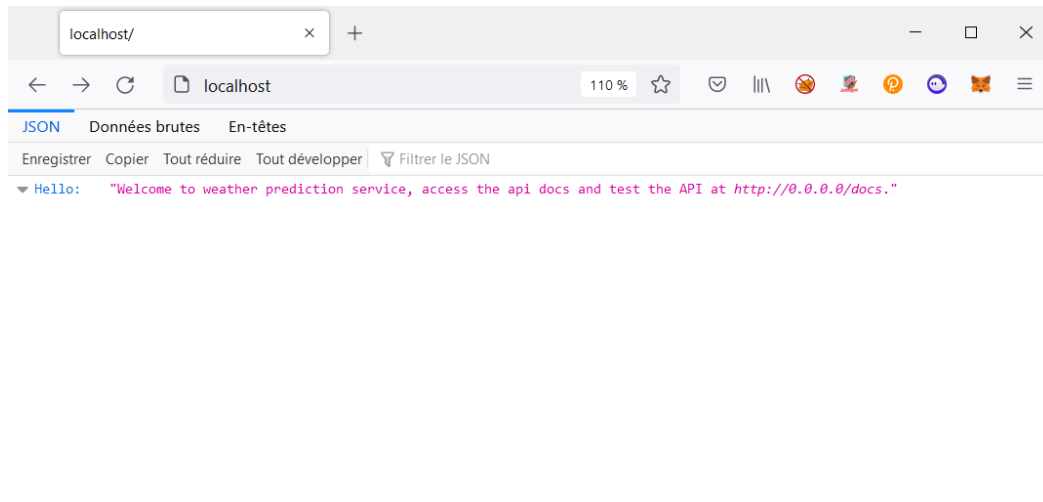
```

Vous trouvez ci-dessous quelques commandes Docker utiles :

docker container rm <ID-container>	Pour supprimer un container
docker container ps --all	Pour afficher tous les containers
docker logs <ID-container>	Pour afficher les logs d'un container (utile pour le débogage)

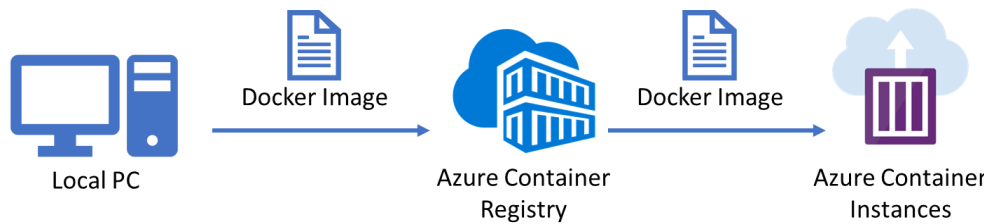
<code>docker rmi <ID-images></code>	Pour supprimer une image
<code>docker images</code>	Pour afficher toutes les images

Vous pouvez accéder le service et le tester à partir du navigateur en visitant « 0.0.0.0 :80 ».



3. Déployons notre conteneur sur Azure

Jusqu'à présent, nous avons pu construire notre image et nous l'avons exécuté en tant que conteneur local. Le processus qu'on va suivre afin de déployer notre image sur azure est décrit dans la figure ci-dessous :



Les étapes suivantes seront mises en œuvre afin de déployer notre API ML en tant que microservice sur azure :

1. Créer d'une instance Azure Container Registry
2. Marquer l'image
3. Uploader l'image vers Azure Container Registry
4. Déployer le conteneur de ACR vers Azure Container Instances
5. Afficher l'API sur le browser
6. Afficher les logs du conteneur

Avant de commencer :

- Installez Azure CLI (> 2.0.29) : <https://learn.microsoft.com/en-us/cli/azure/install-azure-cli> (le fichier .msi est donné dans le dossier de l'atelier), lancez la commande suivante pour s'assurer que Azure CLI est installé.

```
az --version
```

- Assurez-vous que Docker est installé

3.1 La création d'une instance Azure Container Registry

Dans le portail azure, cliquez sur « create resource » et puis choisissez « containers » dans la partie « categories » et ensuite cliquez sur « Container Registry ».

Nommez votre container registry « um6plab1 » et choisissez votre ressource group et cliquez sur « create ».

The screenshot shows the Azure portal interface for the 'um6plab1' Container Registry. The left sidebar contains navigation options like Overview, Activity log, Access control (IAM), Tags, Quick start, Events, and Settings. The main content area shows the 'Essentials' section with details: Resource group (um6p-ds), Location (France Central), Subscription (Azure for Students), and Subscription ID. It also lists the Login server (um6plab1.azurecr.io), Creation date (9/25/2022, 3:19 PM GMT+1), SKU (Standard), Provisioning state (Succeeded), and Soft Delete (Preview) (Disabled). Below this, there's a 'Usage' section showing 100 GiB included in SKU, 0.55 GiB used, and 0.00 GiB additional storage. An 'ACR Tasks' section provides information on building, running, pushing, and patching containers.

Sur l'invite de commande, nous allons nous connecter à notre container registry. Pour ce faire tapez la commande suivante :

```
az acr login --name um6plab1
// output : Login Succeeded
```

3.2 Marquer l'image pour ACR

Afin de mettre l'image dans un registre privé (ACR), on doit tout d'abord marquer l'image avec le nom complet du serveur de connexion du registre.

- Affichons d'abord le nom complet du serveur de connexion

```
az acr show --name um6plab1 --query loginServer --output table
// um6plab1.azurecr.io
```

- Affichons l'image que nous avons pu construire (i.e., fastapi)

Docker images

- Nous allons maintenant marquer l'image « fastapi » avec le nom complet du serveur (um6plab1.azurecr.io)

```
docker tag fastapi um6plab1.azurecr.io/fastapi:v1
```

Exécutez la commande docker images pour voir qu'une autre image a été créée avec le tag approprié

- Poussez l'image marquée vers ACR

```
docker push um6plab1.azurecr.io/fastapi:v1
```

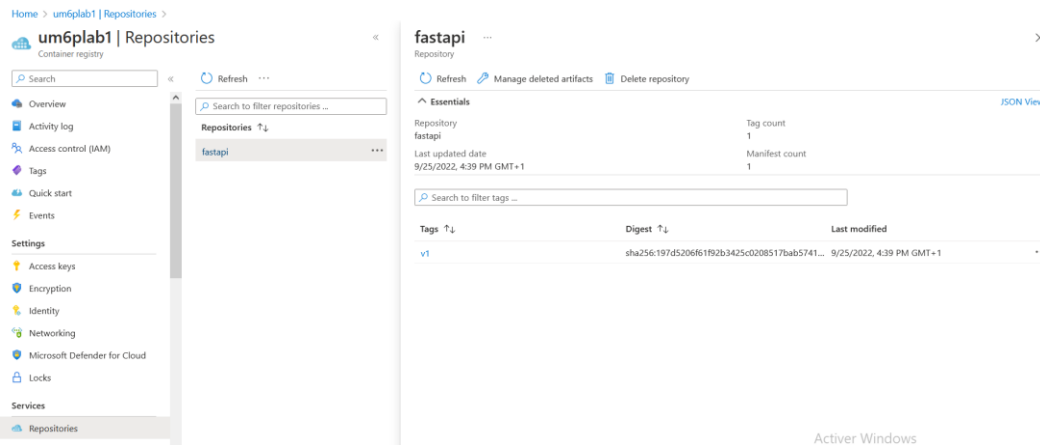
- Affichons les images dans ACR

```
az acr repository list --name um6plab1 --output table
```

- Affichons le tag pour notre image dans ACR

```
az acr repository show-tags --name um6plab1 --repository fastapi --output table
```

Dans le portail azure, cliquez sur votre ACR « um6plab1 » et puis sur « repository » afin d'afficher l'image ainsi créée



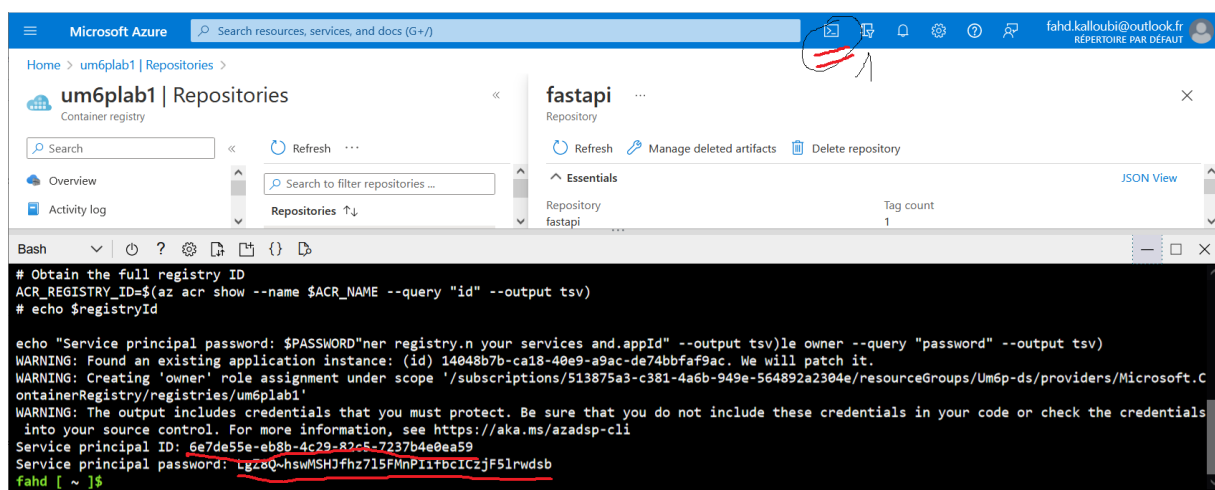
- Affichons le nom complet du serveur de connexion au registre de conteneurs

```
az acr show --name um6plab1 --query loginServer
```

3.3 Déployons le conteneur

Afin d'extraire notre image de ACR vers ACI nous devons créer et configurer un service principal Azure AD « active directory » avec des autorisations d'extraction sur votre registre. Ensuite, vous démarrez un conteneur dans Azure Container Instances (ACI) qui extrait son image de votre registre privé, en utilisant le service ainsi créé pour l'authentification. Pour de plus amples information visitez le lien suivant : <https://learn.microsoft.com/en-us/azure/container-registry/container-registry-auth-aci>

Dans le portail Azure, cliquez sur l'invité de commande « Cloud Shell » pour le lancer. Ensuite, copiez le code du fichier « azure-service-principal.sh » fournit dans le dossier de l'atelier et collez le dans l'invité de commande « cloud shell ». Suite à l'exécution de ce fichier, vous aurez en sortie les informations d'authentification du service principal que nous utiliserons pour se connecter à ACR comme le montre la figure ci-dessous.



Maintenant, nous allons utiliser la commande « az container create » pour déployer le conteneur.

- ✓ « um6plab1.azurecr.io » : est la valeur obtenue en lançant la commande précédente
- ✓ Les valeurs des paramètres « --registry-username » et « --registry-password » sont les valeurs obtenues en lançant le script précédent (soulignées dans la figure)

- ✓ « um6pdsaci » : est la valeur du serveur DNS que j'ai choisi

```
az container create \  
  --resource-group um6p-ds \  
  --name fastapi \  
  --image um6plab1.azurecr.io/fastapi:v1 \  
  --cpu 1 \  
  --memory 1 \  
  --registry-username 6e7de55e-eb8b-4c29-82c5-7237b4e0ea59 \  
  --registry-password 67P8Q~ozWjB7fcmUYfE77NW5DLT1SrtWYucHkc_U \  
  --registry-login-server um6plab1.azurecr.io \  
  --ip-address Public \  
  --dns-name-label um6pdsaci \  
  --ports 80
```

- Pour vérifier l'état du déploiement, lancez la commande suivante (remplacez Um6p-ds par le nom de votre ressource group)

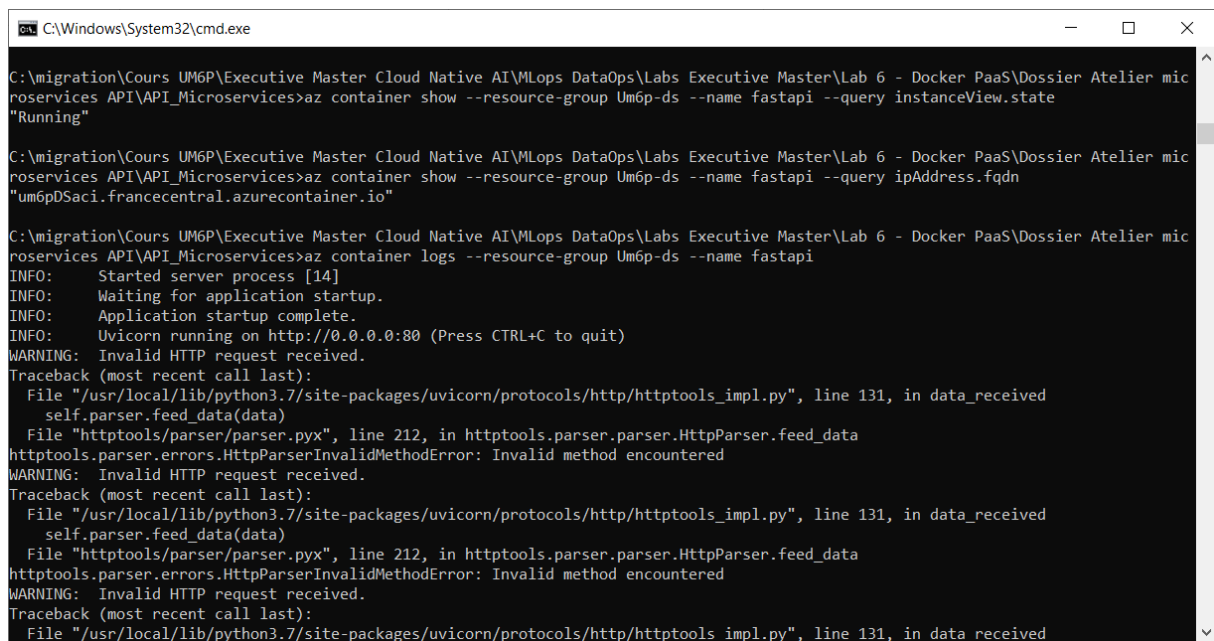
```
az container show --resource-group Um6p-ds --name fastapi --query  
instanceView.state
```

- Pour afficher l'application (lien de l'application)

```
az container show --resource-group Um6p-ds --name fastapi --query ipAddress.fqdn
```

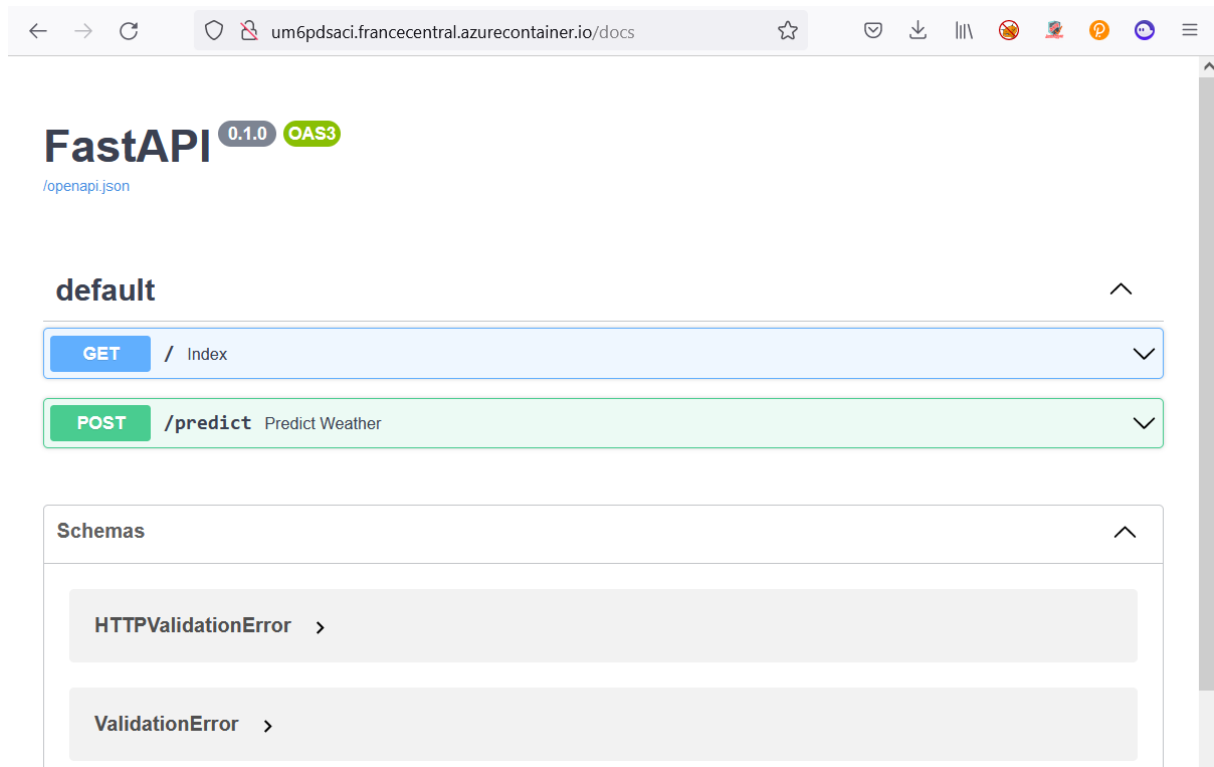
- Pour afficher les sortie logs du conteneur

```
az container logs --resource-group Um6p-ds --name fastapi
```



```
C:\Windows\System32\cmd.exe  
  
C:\migration\Cours UM6P\Executive Master Cloud Native AI\MLops DataOps\Labs Executive Master\Lab 6 - Docker PaaS\Dossier Atelier mic  
roservices API\API_Microservices>az container show --resource-group Um6p-ds --name fastapi --query instanceView.state  
"Running"  
  
C:\migration\Cours UM6P\Executive Master Cloud Native AI\MLops DataOps\Labs Executive Master\Lab 6 - Docker PaaS\Dossier Atelier mic  
roservices API\API_Microservices>az container show --resource-group Um6p-ds --name fastapi --query ipAddress.fqdn  
"um6pDSaci.francecentral.azurecontainer.io"  
  
C:\migration\Cours UM6P\Executive Master Cloud Native AI\MLops DataOps\Labs Executive Master\Lab 6 - Docker PaaS\Dossier Atelier mic  
roservices API\API_Microservices>az container logs --resource-group Um6p-ds --name fastapi  
INFO: Started server process [14]  
INFO: Waiting for application startup.  
INFO: Application startup complete.  
INFO: Uvicorn running on http://0.0.0.0:80 (Press CTRL+C to quit)  
WARNING: Invalid HTTP request received.  
Traceback (most recent call last):  
  File "/usr/local/lib/python3.7/site-packages/uvicorn/protocols/http/httptools_impl.py", line 131, in data_received  
    self.parser.feed_data(data)  
  File "httptools/parser/parser.pyx", line 212, in httptools.parser.parser.HttpParser.feed_data  
httptools.parser.errors.HttpParserInvalidMethodError: Invalid method encountered  
WARNING: Invalid HTTP request received.  
Traceback (most recent call last):  
  File "/usr/local/lib/python3.7/site-packages/uvicorn/protocols/http/httptools_impl.py", line 131, in data_received  
    self.parser.feed_data(data)  
  File "httptools/parser/parser.pyx", line 212, in httptools.parser.parser.HttpParser.feed_data  
httptools.parser.errors.HttpParserInvalidMethodError: Invalid method encountered  
WARNING: Invalid HTTP request received.  
Traceback (most recent call last):  
  File "/usr/local/lib/python3.7/site-packages/uvicorn/protocols/http/httptools_impl.py", line 131, in data_received
```

Afin d'afficher notre API conteneurisée et déployée sur ACI, collez la sortie de la commande d'affichage de l'application dans votre navigateur.



En utilisant l'application créée auparavant (i.e., `app_consuming`), consommer ce service en utilisant cette application.

3.4 Le Nettoyage des ressources

Pour éviter toute autre charge et si vous n'avez pas besoin des ressources créées dans cet atelier, vous pouvez exécuter la commande suivante :

```
az group delete --name um6p-ds
```

4. Déployer un modèle ML sur ACI en utilisant azure ML et MLflow

Dans le dossier « Azure MLflow Deploy » ouvrez le notebook « Train & Deploy local and ACI.ipynb » et suivez les étapes décrites dedans.