

Projet N°3 - Mise en production d'un modèle de Machine Learning

0. Infos utiles

Retrouvez nous directement sur slack/discord, ou pendant l'une des sessions Q&A aux horaires suivants (susceptibles de changer) :

- Session 1 : 02/06/2020 à 14h-17h
- Session 2 : 23/06/2020 à 14h-17h

Alternativement, vous pouvez nous contacter directement par email :

Long DO CAO : ldocao@krystals.ai

Nicolas GIBAUD : nicolas.gibaud@gmail.com

1. Contexte

Dans le projet ML1, vous avez développé un projet de prédiction de souscription d'un prêt bancaire à destination des équipes marketing. La prochaine étape est de vous assurer du bon déroulement de la mise en production de votre modèle en assurant sa disponibilité, sa facilité d'évolution, sa reproductibilité et sa stabilité. Vous mettrez en place toutes les techniques de data engineering que vous avez apprises au cours de cette formation (Conteneurisation, intégration et déploiement continu, tests unitaires, etc).

2. Objectifs

L'objectif est d'industrialiser le modèle de machine learning que vous avez développé pendant le projet ML1, à destination d'utilisateurs (des programmeurs) externes. Concrètement, il s'agit de construire une application qui prend en entrée un certain nombre de paramètres (les features), et qui renvoie un score (la prédiction) compris entre 0 et 1 représentant l'appétence du client au produit à travers d'une API REST. Lors de la mise en place de cette API, vous devrez être particulièrement vigilant aux notions suivantes qui doivent vous être maintenant familières :

- L'existence de tests unitaires
- La conteneurisation de l'application
- Le déclenchement du pipeline CI/CD à chaque commit

- Le déploiement automatisé de l'application sur un cluster kubernetes
- L'existence d'au moins 3 environnements (develop, staging et master¹)
- La protection des données sensibles (mot de passe) via des secrets

3.Format de restitution

A l'issue du projet, vous devrez délivrer les éléments suivants :

- Git repository
- Soutenance orale
- Documentation pour tester votre API (adresse IP, port, input et output attendus etc)

4.Guidelines et contraintes

Git

- Le code doit être sauvegardé comme un repository git
- Le repository git doit contenir 3 branches : develop, staging, master
- Le code final doit avoir au moins un tag (ex: v1.0) sur la branche master
- Les développements doivent être faits sur des branches distinctes des branches staging et master :
 - Idéalement sur des branches spécifiques (une branche par fonctionnalité développée)
 - A minima sur une branche develop
- L'intégration des développements depuis les branches spécifiques ou la branche develop vers les branches staging et master doit se faire selon le processus suivant :
 - Merge request (MR). Attention : par défaut, gitlab propose de détruire la branche de départ à l'issue du merge. Pensez à bien désactiver cette option le cas échéant (par exemple : staging > master)
 - Code review effectuée par les membres de votre équipe
 - Validation de la MR et déclenchement du CI/CD pour intégration à la branche staging puis master puis déploiement

Tests

- Nous tolérons une couverture de tests moyenne au regard de l'ampleur et de l'objectif du projet. En revanche, nous apprécierons évidemment si elle est élevée.

¹ Cette branche fera office de production



- Les tests unitaires doivent être indépendants les uns des autres (utilisez la technique de mock quand cela s'avère nécessaire)

Intégration et déploiement continue (CI/CD)

- Le build de l'image docker est commune à tous les environnements
- Chaque commit sur develop ou sur staging doit déclencher les tests unitaires (pas sur master)
- L'étape de déploiement doit se déclencher uniquement sur la branche staging ou master
- Chaque pipeline de CI doit être composé a minima des trois étapes suivantes:
 - Build : Build de l'image docker
 - Tests : Run des tests unitaires sur ce container sur l'environnement develop ou staging
 - Déploiement : Déploiement du container sur l'environnement de staging ou master
- Aucune variable personnelle ne doit être hardcodée ou sauvegardée dans git (on pense au fichier de config ou à la clé JSON). Elles doivent être sauvegardées séparément à l'aide des variables d'environnement dans gitlab ou dans les secrets d'un cluster kubernetes.
- Le pod doit lire dynamiquement les données socio eco depuis la base de données, il n'est pas autorisé de faire un dump et de l'utiliser

Vous ne serez pas évalué sur les performances du modèle, mais uniquement sur sa bonne mise en production.

5. Comment démarrer ?

Dans toute la suite de ce document, les commandes indiquées sont valables sur un environnement UNIX. Il est recommandé pour les utilisateurs de Windows d'utiliser WSL. Par ailleurs, vous devez remplacer GROUP_ID par le numéro de votre groupe (exemple: 2).

Vérification des permissions

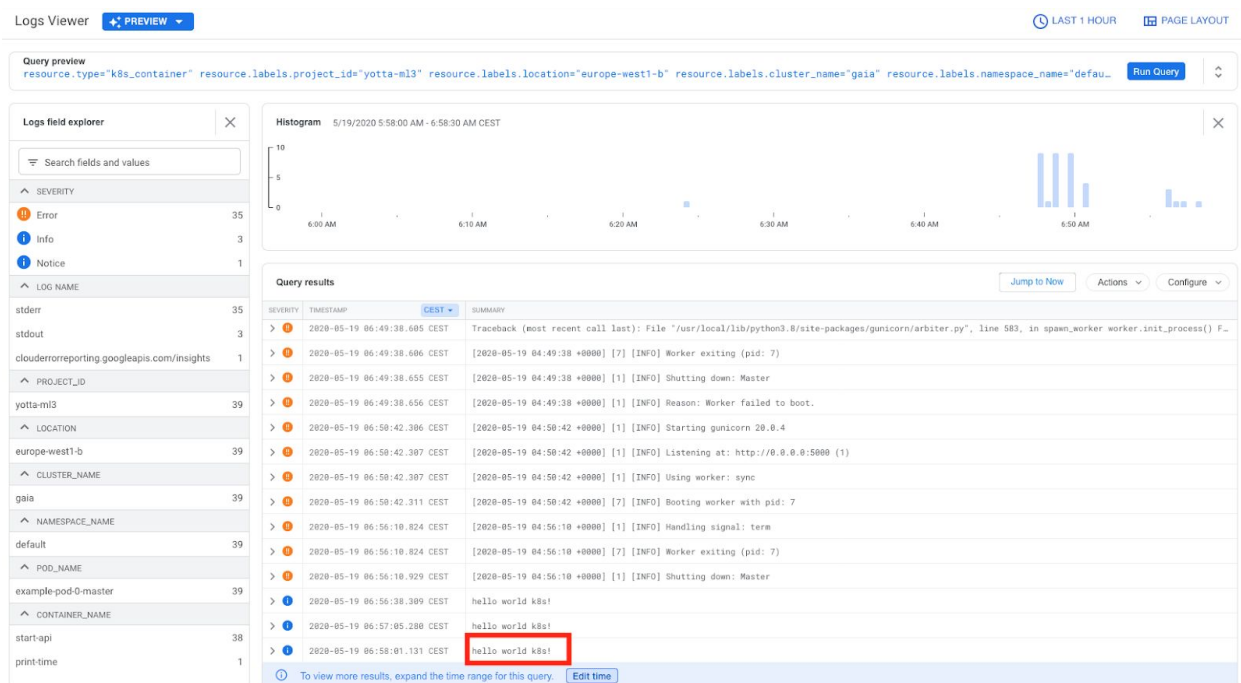
Vérifiez que vous avez bien accès aux ressources cloud suivantes :

- Gitlab git repository du code de départ :
<https://gitlab.com/yotta-academy/cohort-2020/projects/ml-prod-projects/chaos>
(NOTE : il se peut que vous ayez besoin d'intégrer votre nom d'utilisateur gitlab dans l'URL pour pouvoir cloner le projet:



https://<your_user_name>@gitlab.com/yotta-academy/cohort-2020/projects/ml-prod-projects/chaos.git)

- Gitlab git repository de votre groupe :
https://gitlab.com/yotta-academy/cohort-2020/projects/ml-prod-projects/chaos-GR OUP_ID
- Google cloud platform : accéder au projet google cloud 'yotta-ml3', puis accéder au cluster kubernetes > Workloads, et aller lire les logs de example-pod-0-develop , vous devriez y voir "hello world k8s!". Si le message ne s'affiche pas, il est possible que la fenêtre de temps soit trop récente. Réglez la fenêtre de temps pour afficher les logs du 20/05/2020 aux alentours de 10:28 (CEST).



En cliquant sur Kubernetes Engine > Workloads > example-pod-0-develop > container logs, vous devriez voir apparaître "hello world k8s!"

Environnement technique requis

Nous listons ci-après les logiciels nécessaires pour ce projet à installer en local sur votre ordinateur. A ce stade de la formation, vous devriez néanmoins avoir déjà tout installé :

- Python
- git
- docker
- google cloud sdk



- kubectl
- cloud sql proxy (voir section suivante)
- Option : [Dbeaver](#) (ou tout autre database manager qui peut se connecter à postgresQL)
- Votre éditeur de texte préféré (vim, emacs, VScode, Sublim text, Atom, etc.)
- Un fichier pickle (ou équivalent) de votre modèle de machine learning entraîné

Google cloud SDK

Pour rappel, vous pouvez installer le google cloud sdk avec la commande suivante

```
$ curl https://sdk.cloud.google.com > install.sh  
$ ./install.sh
```

Configurer votre google cloud sdk avec les informations suivantes (en utilisant gcloud init) :

1. Create a new configuration
2. Configuration name : yotta-ml3-group-GROUP_ID
3. Login with a new account : entrez vos login pour google cloud
4. Nom du projet : yotta-ml3
5. Configure compute region and zone : Y
6. Région : europe-west1-b

Docker

Ensuite, configurez docker pour qu'il utilise les credentials du google cloud SDK. Pour cela, utilisez la commande suivante ;

```
$ gcloud auth configure-docker
```

Acceptez la mise à jour du fichier de configuration de docker. Vous devriez voir les lignes suivante s'afficher à l'issue de la commande ci-dessus. Vérifiez que vous avez la même configuration dans votre fichier `$HOME/.docker/config.json`

```
(yotta-ml3-group-5) 11:15 ldocao@Longs-MacBook-Pro:chaos-5 gcloud auth configure-docker
Adding credentials for all GCR repositories.
WARNING: A long list of credential helpers may cause delays running 'docker build'. We recommend passing the registry nam
e to configure only the registry you are using.
After update, the following will be written to your Docker config file
located at [/Users/ldocao/.docker/config.json]:
{
  "credHelpers": {
    "gcr.io": "gcloud",
    "marketplace.gcr.io": "gcloud",
    "eu.gcr.io": "gcloud",
    "us.gcr.io": "gcloud",
    "staging-k8s.gcr.io": "gcloud",
    "asia.gcr.io": "gcloud"
  }
}
Do you want to continue (Y/n)? y
Docker configuration file updated.
```

Kubernetes

Enfin, connectez vous au cluster kubernetes de travail gaia avec la commande suivante :

```
$ gcloud container clusters get-credentials gaia
```

Vous pouvez vérifier que la connexion au cluster est bien effective en listant la commande suivante, qui liste les pods actifs :

```
$ kubectl get pods
```

Données confidentielles

Pour utiliser toutes les ressources de GCP, nous fournissons à chaque groupe des logins uniques. Il y a deux fichiers : fichier de configuration en YAML qui contient notamment les login de connexion à la base de données, et une clé JSON qui sera nécessaire au cloud sql proxy (voir section cloud sql proxy). Attention : vous devez considérer ces logins comme des données confidentielles. Ne les ajoutez pas à votre git repository ! Idéalement, sauvegardez les dans un gestionnaire de mot de passe.

Data et Cloud SQL proxy

Les données que vous allez utiliser (initialement en csv) sont désormais stockées dans une base de données postgresQL dans le cloud. Il s'agit d'un changement de format qui arrive

fréquemment lorsque vous passez du POC à l'industrialisation². Bien que les deux tables existent, vous n'aurez normalement besoin que de la table socio_eco dans le cadre de ce projet. Pour vous y connecter, GCP impose l'utilisation d'un proxy que vous devez mettre en place.

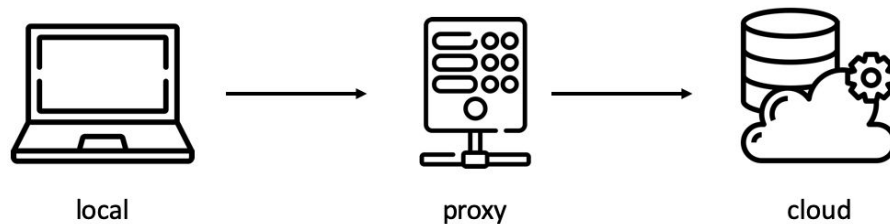


Schéma du rôle du proxy

Concrètement, suivez les étapes suivantes :

- Télécharger l'exécutable depuis cette page <https://cloud.google.com/sql/docs/postgres/sql-proxy>
- Ouvrir un terminal dans le même dossier où se trouve l'exécutable téléchargé, et lancer la commande ci dessous (assurez vous d'avoir bien configuré votre google cloud sdk au préalable) :

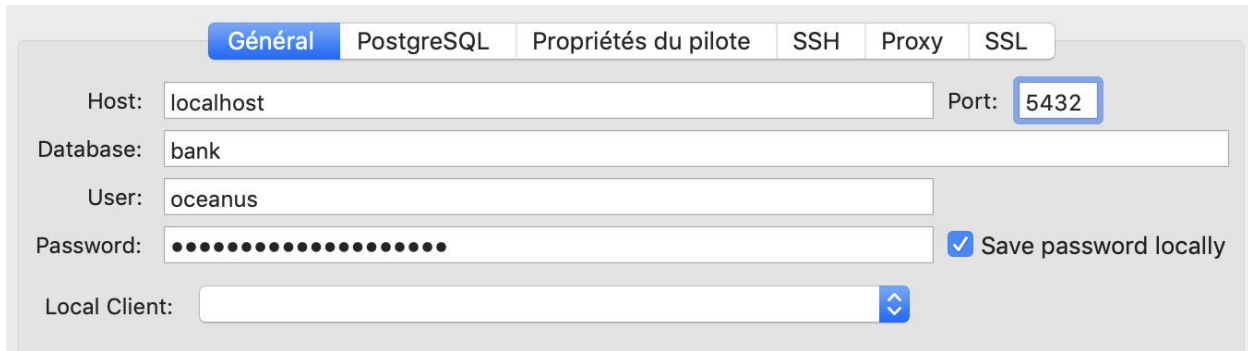
```
$ ./cloud_sql_proxy -instances=yotta-ml3:euope-west1:uranus=tcp:5432
```

Vous reconnaîtrez dans la commande ci-dessus (dans l'ordre) : le nom du projet, la région du cloud, le nom de la base de données, et le port d'accès. Le rôle du proxy est de simuler une base de donnée en local. Mais il s'agit en fait d'un lien vers la base de donnée distante. Vous devez maintenir cette fenêtre ouverte aussi longtemps que vous aurez besoin de la base. Il est possible que cette connexion soit interrompue et que ce lien soit rompu (par exemple à cause d'une connexion internet instable). Dans ce cas, coupez le proxy avec Ctrl+C, et relancez tout simplement la commande. Vous devriez voir dans votre terminal une ligne qui ressemble à :

```
$ ./cloud_sql_proxy -instances=yotta-ml3:euope-west1:uranus=tcp:5432
2020/05/19 03:07:09 Rlimits for file descriptors set to {8500
9223372036854775807}}
2020/05/19 03:07:12 Listening on 127.0.0.1:5432 for yotta-ml3:euope-west1:uranus
2020/05/19 03:07:12 Ready for new connections
```

² Souvent, ce changement s'accompagne d'autres modifications comme un changement de périmètre ou de noms de colonnes. Nous avons simplifié le problème en créant une copie iso des fichiers csv initiaux.

- Tester la connexion avec l'outil de votre choix. Vous pouvez utiliser soit psql en CLI (nécessite l'installation de postgresql-client en local), ou un database manager comme dbeaver (recommandé ici et décrit ci-après). Une fois l'application dbeaver ouverte, créez une nouvelle connexion en cliquant sur Database > New Database Connexion > PostgreSQL > Next. Entrez maintenant les coordonnées de la base de données que vous trouverez dans le fichier de configuration config.yml fourni. Constatez que vous avez bien accès aux données. Comme vous utilisez un proxy, veuillez à bien mettre :
 - Host: localhost
 - Port: celui indiqué à la fin de la commande proxy (5432 dans le cas de l'exemple donné ci-dessus)



Interface de dbeaver pour entrer les paramètres de connexion à la base de données

A noter que ce composant devra être présent également dans votre pod kubernetes pour que l'application puisse se connecter à la base de données.

Description du code de départ

Nous mettons à disposition un squelette du livrable du projet ML3 sous la forme d'un [repository gitlab](#). Il s'agit de votre code de départ qu'il faudra modifier. Il y a plusieurs catégories de fichiers à distinguer :

Modèle de machine learning

Dans le dossier chaos, vous retrouverez l'architecture DDD en couches standard : application, domain, infrastructure. On notera également la présence d'un dossier chaos/infrastructure/config (qui contient tout ce qui est nécessaire à la lecture d'un fichier de configuration en YAML), et le dossier de tests. Par défaut, le code va utiliser le fichier de configuration indiqué par la variable d'environnement "YOTTA_ML3_CONFIGURATION_PATH", ou le fichier chaos/infrastructure/config/config.yml



si la variable d'environnement n'est pas définie. Le fichier `config_template.yml` indique la structure que doit suivre le fichier de configuration attendu par l'app.

Dans le code de départ, nous avons ajouté un modèle de machine learning entraîné³ sous format binaire `chaos/domain/model.pkl` qui prend en entrée des données marketing et socio-économiques, et qui renvoie une prédiction d'appétence (un float entre 0 et 1). Ce modèle est utilisé à travers la classe `Customer` définie dans `chaos/domain/customer.py`. Il s'agit d'un exemple de comment charger un modèle de machine learning entraîné en production pour le mettre à disposition ensuite à l'application flask.

API flask

En plus du code précédent, nous avons rajouté un exemple d'un endpoint d'une API en flask qui prend un nombre en input, et qui renvoie le double de ce nombre en output (fichier `chaos/application/server.py`). Le format d'input attendu est le suivant :

```
{"question": 3}
```

Le format de la réponse est le suivant :

```
{"answer": 6} #returns 0 if an error occurred
```

D'une part, cela vous permet d'avoir un exemple de comment créer un endpoint API avec flask. D'autre part, nous utiliserons cet exemple simple pour l'exposer avec kubernetes.

Conteneurisation

Pour déployer votre application, il est nécessaire de conteneuriser votre application avec Docker. Un template de `dockerfile` fonctionnel est livré avec le code de départ. Vous devriez être familiarisé avec la plupart des commandes pré-écrites. A priori, ce projet ne nécessite pas que ce fichier soit modifié. La dernière ligne du template est :

```
CMD ["gunicorn", "chaos.application.server:app", "-b", "0.0.0.0:5000"]
```

Il s'agit d'un équivalent à la commande suivante qui lance un server flask :

```
CMD [ "python", "server.py" ]
```

³ En réalité, il s'agit d'un modèle totalement faux (d'ailleurs le code d'entraînement est fourni dans `chaos/application/train_model_for_appetence.py`).

Dans sa version standard, flask n'est pas prévu pour être déployé sur un cluster de production. En effet, il comporte quelques limitations majeures telles que [l'impossibilité de gérer des requêtes en parallèle parce qu'il est single-threaded](#). Nous proposons ici l'utilisation de gunicorn, un server web qui supporte les problématiques de production, en surcouche à flask. En pratique, cela ne change rien à votre code, sauf cette fameuse modification dans le dockerfile et l'installation de cette librairie python avec pip. Sa facilité d'utilisation (par rapport à nginx ou apache) en fait une solution que nous privilégions ici.

Pour des raisons de simplicité, toutes les image docker seront stockées sur le docker repository de GCP (équivalent de dockerhub), appelé container registry.

Kubernetes

Les fichiers relatifs à kubernetes (appelé Google Kubernetes Engine sur GCP, ci-après GKE) sont situés dans le dossier deployment à la racine du repository. Nous fournissons un exemple de pod minimaliste qui va démarrer un container à partir de l'image docker du repository. Dans cet exemple, le pod ne contient qu'un seul container, celui défini par le dockerfile. Vous remarquerez que le pod ne fait essentiellement... rien. Il affiche un message, puis reste inactif pour l'éternité⁴ (commande sleep infinity).

Notez que dans les metadata, le nom du pod est example-pod-0-develop. Lorsque vous définirez vos pods, prenez garde à bien remplacer 0 > GROUP_ID, et develop > ENVIRONNEMENT. Par exemple, le groupe 2 qui souhaite créer un pod sur l'environnement staging nommera ce pod : chaos-pod-2-staging.

Appliquez cette règle également aux autres variables du cluster. Par exemple, le groupe 2 devra nommer les secrets : chaos-secrets-2.

Par ailleurs, pour que l'application python (depuis GKE) puisse accéder aux autres ressources de GCP, et notamment à la base de données, il faut configurer ses droits d'accès. Cette opération étant un peu fastidieuse, nous les avons déjà pré-configurés. Pour votre information, cela s'est fait de la manière suivante :

1. Création d'un compte de service⁵ yotta-ml3 sur GCP
2. Génération d'une clé privée JSON que nous vous fournissons manuellement et individuellement par équipe.

⁴ La présence de sleep infinity est importante. En effet, si cette commande était absente, le pod serait considéré comme achevé (car il a fini d'exécuter toutes les commandes demandées), et serait tué par kubernetes. Comme il s'agit d'un pod, le container serait alors redémarré, et se retrouverait donc dans une boucle de redémarrage infinie.

⁵ Équivalent d'un compte utilisateur classique mais pour une application tierce (par exemple une application python par rapport à cloud sql). Généralement, on définit à l'avance un ensemble de droits le plus restreint possible nécessaire à son fonctionnement.

3. Modification de l'IAM GCP pour que le compte de service yotta-ml3 ait accès à cloud SQL

Pipeline de CI/CD

Nous fournissons à chaque groupe un repository gitlab dans lequel vous pusherez votre code, et depuis lequel seront lancés vos pipelines de CI/CD. Pour que gitlab puisse lancer ces pipelines, il a besoin des éléments suivants :

- Accès à GKE
- Accès au container registry
- Noeuds de calcul (fournis gratuitement par gitlab)

Pour simplifier la mise en place, nous avons déjà configuré gitlab pour qu'il ait accès au cluster kubernetes et au container registry. Pour votre information, cela s'est fait de la manière suivante (de manière similaire à la configuration décrite pour GKE) :

4. Création d'un compte de service gitlab sur GCP
5. Génération d'une clé privée JSON que nous avons copié dans les variables d'environnement CI/CD de gitlab
6. Modification de l'IAM GCP pour que le compte de service gitlab ait accès à GKE et au container registry

Dans le cadre du projet ML3, votre seule tâche est désormais de remplir le fichier .gitlab-ci qui définit la séquence des opérations déclenchées à la suite d'un commit sur une branche donnée. Ne modifiez pas le premier bloc (cf copie ci dessous) qui build et push l'image docker. En effet, ce bloc définit la politique de nommage des images docker et ne doit pas être changé pour garantir que chaque groupe n'écrase pas les images des autres.

```
10  build-docker-image:
11      stage: build
12      variables:
13          IMAGE_NAME: eu.gcr.io/yotta-ml3/$CI_PROJECT_NAME:$CI_COMMIT_BRANCH
14      script:
15          - echo "$GITLAB_IAM_PRIVATE_KEY" > key.json
16          - docker build . -t $IMAGE_NAME
17          - docker login -u _json_key -p "$(cat key.json)" $IMAGE_NAME
18          - docker push $IMAGE_NAME
```

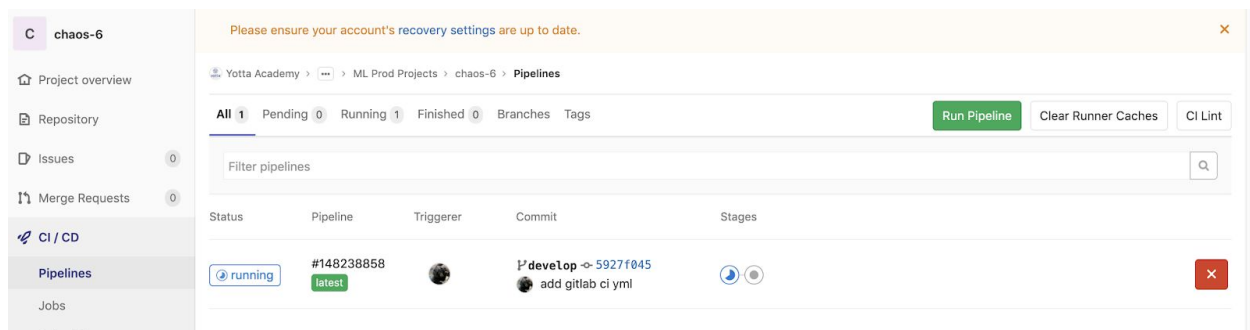
6. Les grandes étapes de développement

Nous vous proposons donc d'implémenter au fur et à mesure les éléments dans l'ordre suivant (vous êtes bien entendu libre de le faire différemment si vous vous sentez suffisamment à l'aise). L'idée est de mettre en place la chaîne dans sa globalité, puis dans un second temps d'introduire la complexité induite par la brique ML.

Prise en main du code de départ

Commençons tout d'abord par récupérer le code de départ qui doit fonctionner dans son intégralité. Procédez aux étapes suivantes dans l'ordre :

1. Clonez le code de départ et du votre (à cet instant vide) en local.
2. Créez tout de suite une branch git develop, et activez la.
3. Copiez le code de départ dans votre git repository sur la branche develop (attention à bien inclure les fichiers cachés tels que .gitlab-ci, mais pas le dossier .git/ !).
4. Committez et poussez le code. Cela devrait déclencher un pipeline CI/CD qui se finit avec succès. Nous vous recommandons de mettre le fichier config.yml dans le .gitignore et .dockerignore pour éviter toute fuite inopinée de données confidentielles.



5. Créez un environnement virtuel pyenv/virtualenv et installez les packages nécessaires à partir du requirements.txt fourni à cet effet. Veillez à bien utiliser la même version de python que celle indiquée en tête du dockerfile (Sinon les tests unitaires de base renverront une erreur). Comme nous utilisons l'une des dernières versions de python, il est possible que pyenv ne reconnaisse pas encore cette version. Choisissez alors la version la plus proche (exemple: 3.8.1 => 3.8-dev).
6. Installez localement la librairie chaos. Pour cela exécutez la commande suivante depuis un terminal à la racine de votre repository git :

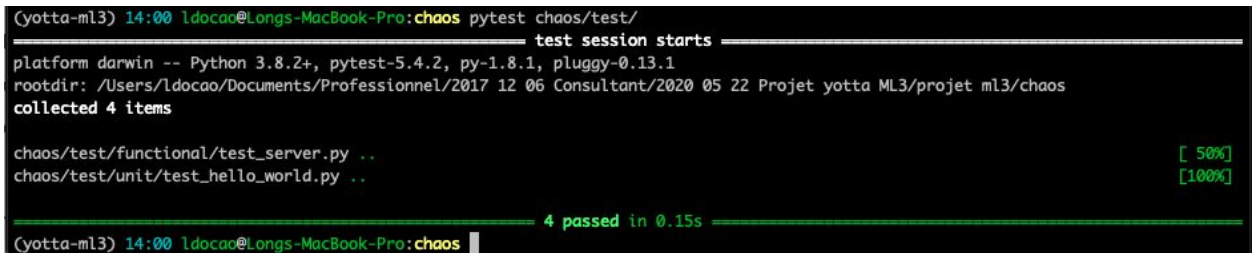
```
$ pip install -e ./
```

7. Au choix : placer le fichier config.yml dans le dossier chaos/infrastructure/config/ ; ou placez le ailleurs (par exemple : path/to/config.yml) et faites pointer la variable d'environnement YOTTA_ML3_CONFIGURATION_PATH dessus, en ajoutant la ligne suivante à votre .bashrc/.bash_profile :

```
export YOTTA_ML3_CONFIGURATION_PATH=/path/to/config.yml
```

8. Lancez les tests unitaires et fonctionnels pour vérifier le bon fonctionnement de cette API. Pour exécuter ces tests, vous devez tout d'abord lancer le serveur, et lancer les tests avec la commande pytest. Ils doivent tous passer avec succès.

```
$ python application/server.py #on terminal 1  
$ pytest tests/ #on terminal 2
```



```
(yotta-ml3) 14:00 ldocao@Longs-MacBook-Pro:chaos$ pytest chaos/test/  
test session starts  
platform darwin -- Python 3.8.2+, pytest-5.4.2, py-1.8.1, pluggy-0.13.1  
rootdir: /Users/ldocao/Documents/Professionnel/2017 12 06 Consultant/2020 05 22 Projet yotta ML3/projet ml3/chaos  
collected 4 items  
  
chaos/test/functional/test_server.py .. [ 50%]  
chaos/test/unit/test_hello_world.py .. [100%]  
  
4 passed in 0.15s  
(yotta-ml3) 14:00 ldocao@Longs-MacBook-Pro:chaos$
```

Les tests fonctionnels seront uniquement utilisés en local. D'ailleurs aucun test fonctionnel n'est attendu dans ce projet (voir la section Aller plus loin en fin de document).

9. Buildez votre première image docker, puis taggez la eu.gcr.io/yotta-ml3/chaos-GROUP_ID:develop, enfin pushez.
10. Modifiez le numéro GROUP_ID dans les fichiers relatifs à kubernetes : cette opération va nous permettre de distinguer chaque groupe sur un cluster commun à toutes les équipes Yotta. Plus précisément, modifiez le fichier deployment/example_pod_develop.yml et changez les chiffres 0 par votre GROUP_ID aux emplacements suivants (ligne 4 et 10) :

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: example-pod-0-develop
5    labels:
6      environment: develop
7  spec:
8    containers:
9      - name: start-api
10       image: eu.gcr.io/yotta-ml3/chaos-0:develop
11       imagePullPolicy: Always
12       command: [ "/bin/bash", "-c" ]
13       args: [ "echo hello world k8s! ; sleep infinity" ]
```

N'oubliez pas de modifier le chiffre 0 par votre GROUP_ID

11. Lancez le pod fourni kubernetes avec la commande

```
$ kubectl apply -f deployment/example_pod_develop.yml
```

Vérifiez que le pod a bien été créé avec la commande :

```
$ kubectl get pods
```

12. Connectez vous sur GCP avec votre navigateur web.

13. Vérifiez que votre pod est bien actif (status: **running**), et que les logs affichent bien "hello world k8s!"

Si vous avez effectué toutes ces opérations avec succès, félicitations ! Vous êtes maintenant prêts à apporter votre touche personnelle.

Transférer les secrets sur le cluster kubernetes

Créez un secrets qui contient à la fois le fichier de configuration et la clé JSON pour le cloud sql proxy appelé chaos-secrets-GROUP_ID. Pour cela, exécutez la commande suivante :

```
$ kubectl create secret generic chaos-secrets-GROUP_ID --from-file
```

Pour vérifier que vous avez créé le secret correctement, la commande `kubectl describe secrets chaos-secrets-GROUP_ID` devrait retourner les éléments suivants :

```
11:31 ldocao@Longs-MBP:~$ kubectl describe secrets chaos-secrets-0
Name:          chaos-secrets-0
Namespace:     default
Labels:        <none>
Annotations:   <none>

Type: Opaque

Data
====
config.yml:          162 bytes
yotta-ml3-00224c47f66d.json: 2296 bytes
```

Les secrets doivent contenir les deux fichiers fournis (config.yml et la clé JSON). Attention à bien indiquer votre numéro de groupe (ici 0)

Exposer une API simple de l'environnement develop

Nous allons maintenant exposer *manuellement* l'API exemple au monde extérieur.

1. Créez manuellement (c'est à dire hors pipeline CI/CD) le fichier de deployment yaml qui expose le endpoint exemple avec un nombre de replica de 1, et exécutez le via kubectl. Ce container a besoin du fichier de configuration pour bien fonctionner. Veillez donc à lui fournir un volume contenant les secrets transférés à l'étape précédente. A ce stade, si tout s'est bien déroulé, votre deployment devrait avoir le status **OK**, et ses logs devraient afficher :

> 	2020-05-19 14:48:08.566 CEST	[2020-05-19 12:48:08 +0000] [1] [INFO] Starting gunicorn 20.0.4
> 	2020-05-19 14:48:08.567 CEST	[2020-05-19 12:48:08 +0000] [1] [INFO] Listening at: http://0.0.0.0:5000 (1)
> 	2020-05-19 14:48:08.567 CEST	[2020-05-19 12:48:08 +0000] [1] [INFO] Using worker: sync
> 	2020-05-19 14:48:08.570 CEST	[2020-05-19 12:48:08 +0000] [7] [INFO] Booting worker with pid: 7

2. Ouvrez l'application au monde extérieur avec un service de type LoadBalancer. Attention à bien sélectionner les pods cibles à servir. Si tout s'est passé correctement, vous devriez voir que le service sert bien le pod que vous avez créé à l'étape précédente

Cluster	gaia	
Namespace	default	
Labels	app : api-deployment-5-develop	environment : develop
Cloud logs	chaos-deployment-5-develop	
Type	LoadBalancer	
External endpoints	35.241.152.150:5000 🔗	

LoadBalancer

Cluster IP	10.0.1.199
Load balancer IP	35.241.152.150
Load balancer	a5f4978039a9c11eaaddf42010a84007

Deployments

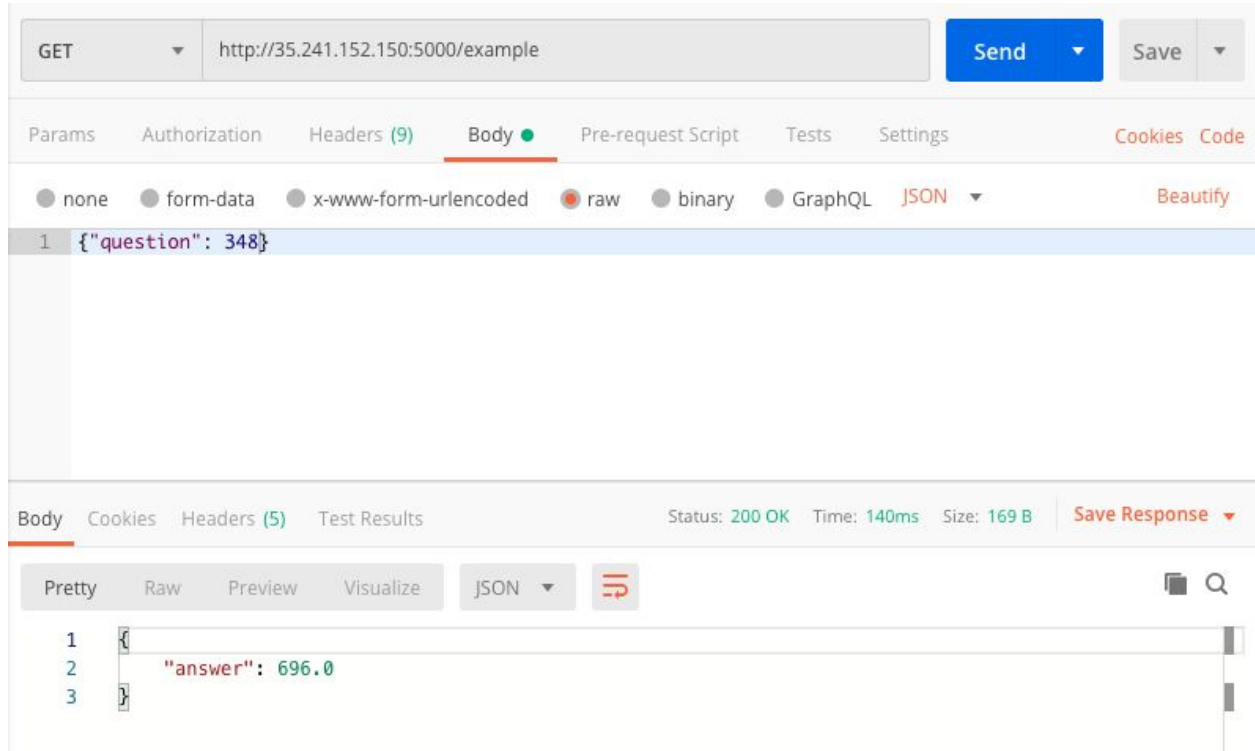
Name	Status	Pods
chaos-deployment-5-develop	✔ OK	1/1

Serving pods

Name	Status	Restarts	Created on ^
chaos-deployment-5-develop-745b88887-2kwgg	✔ Running	0	May 20, 2020, 3:21:04 PM

Le service est bien rattaché à un pod comme indiqué dans la section "Serving pods"

3. Tester avec postman ou un autre script depuis votre ordinateur local. Vous devriez pouvoir accéder à votre API depuis l'extérieur.



Exemple de test de l'API depuis un client externe au cluster (ici postman)

Déployer à partir du pipeline de CI/CD

Maintenant que vous avez le fichier YAML pour déployer un service et l'exposer, vous pouvez rajouter cette étape au pipeline de CI/CD. Modifiez donc le fichier `.gitlab-ci` pour que le déploiement se fasse automatiquement à chaque commit, après le build de l'image docker et le lancement des tests unitaires. Dans un premier temps, le déploiement se fera quelque soit l'environnement. Dans l'étape suivante, nous ajusterons les conditions de déclenchement.

Définir une politique de CI différente par environnement

Créez 3 fichiers YAML de deployment (un par environnement) : `develop`, `staging`, `master`. Dans le cadre de ce projet, cette distinction n'a que peu d'intérêt puisque les 3 environnements vont se comporter de la même manière (même base de données, même modèle etc). Cette étape vous prépare néanmoins à utiliser différents environnements en anticipation de projets de plus grande envergure (par exemple dans le cadre d'une mise à jour du modèle de ML).

Définissez maintenant une politique de pipeline CI/CD différente pour chaque environnement :

- Le build de l'image docker est commune à tous les environnements
- Chaque commit sur develop ou sur staging doit déclencher les tests unitaires (pas sur master)
- L'étape de déploiement doit se déclencher uniquement sur la branche staging ou master
- Les deployment de l'environnement master doivent avoir un replica⁶ de 2

Ajouter le endpoint API

Définissez un nouveau endpoint qui prend en entrée l'ensemble des données marketing qui vont vous servir à la prédiction pour un client donné, et qui renvoie un nombre. Autrement dit, l'input devrait être l'ensemble des valeurs utilisées pour faire la prédiction sur une ligne. Cela devrait ressembler à (évidemment l'exemple ci-après est très incomplet) :

Input :

```
{"age": 23,  
"balance": -10.3}
```

Output:

```
{"appetence": 0.82}
```

Dès maintenant, faites en sorte d'appeler la class Customer (même si pour le moment le modèle fourni en exemple ne prend en entrée que 3 features numériques complètement arbitraires) pour retourner une prédiction.

Ajouter le cloud sql proxy au pod

Désormais, l'application python va avoir besoin d'un accès à la base de données. Nous devons donc ajouter un container contenant le cloud sql proxy au pod du deployment. Ce pod contiendra donc les éléments suivants :

- Containers : votre pod sera constitué de 2 containers ; le premier contiendra le cloud sql proxy, le second votre application
- Volume : un volume doit être monté contenant les secrets (dans notre cas le fichier de configuration et la clé JSON)

⁶ On simule ici la résilience de notre environnement de production

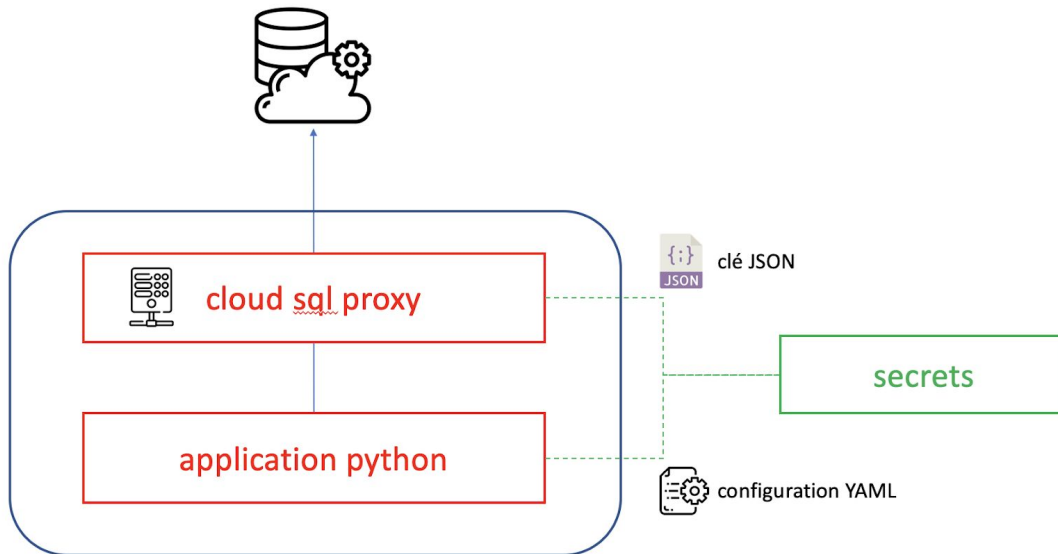


Schéma du contenu du service kubernetes : pod (bleu) avec ses deux containers (rouge), et le volume de secrets (vert) attaché à chaque container. Le volume de secrets fournit le fichier de configuration YAML au pod python, et la clé JSON au proxy pour se connecter à la base de données SQL.

Référez vous à la [documentation](#) pour créer ce nouveau container.

Insérer le véritable modèle ML entraîné

Remplacez le modèle d'exemple par le modèle ML entraîné à partir de votre projet ML1 sous un format binaire (pickle par exemple). Committez ce fichier au git repository. En principe, le code qui a servi à l'entraînement devrait être inclus dans le repository (et non le binaire). Nous faisons cependant ici le choix opposé pour simplifier le problème. Attention à bien inclure toutes les transformations des données que le modèle requiert.

Tester le code

Tout d'abord, écrivez tous les tests *unitaires* nécessaires à votre application. Notons que la notion d'unitaire est ici particulièrement importante. En effet, les tests seront lancés par le gitlab CI qui n'a aucune connaissance de l'environnement (pas d'accès à une base de données externes, ni au server flask, ni aux fichiers de configuration). Ajustez donc vos tests en conséquence. Nous tolérerons une couverture de tests moyen au regard de l'ampleur et de l'objectif du projet. En revanche, nous apprécierons évidemment si elle est élevée.

Pour vous aider, nous vous recommandons également d'écrire des tests *fonctionnels*, c'est à dire qui vont tester de bout en bout l'application (prenez par exemple le test fonctionnel fourni). Utilisez les en local pour vous aider à debugger votre code. Notez cependant que ces tests ne seront pas déclenchés au sein du pipeline CI/CD par soucis de simplicité du projet (voir la section Aller plus loin).

Tagger la version finale

Lorsque vous avez enfin achevé toutes ces étapes, taggez la version finale de votre code sur la branche master. C'est cette version que nous considérons comme faisant partie du livrable de ce projet. Félicitations, vous avez mis en production votre modèle de machine learning !

Conseils

- Utiliser des noms de git branch identiques à vos environnements de développement
- Tester toujours localement avant de lancer le pipeline de CI (par exemple faites un docker build, ou lancer les tests en local)
- Votre pipeline de CI/CD doit être le plus rapide possible pour que le débogage soit facile. N'hésitez pas à annuler certaines étapes ou à les simplifier pour tester la partie suivante. Si certaines d'entre elles sont indépendantes, faites les tourner en parallèle !
- Si votre pod est bloqué à l'étape de création, c'est qu'il lui manque probablement un élément pour démarrer, par exemple les secrets.
- Les fichiers de configuration YAML (gitlab-ci ou kubernetes) sont assez longs et propices à l'erreur. Utilisez les outils en ligne de vérification automatique : [kubeyaml](#) ou gitlab CI lint

Yotta Academy > ML Prod Projects > chaos > Pipelines

All 13 Pending 0 Running 0 Finished 13 Branches Tags Run Pipeline Clear Runner Caches CI Lint

Filter pipelines

Status	Pipeline	Triggerer	Commit	Stages	
passed	#147533707 latest		develop -> e8265e7d refactor: rename file to reflect ...		00:05:25 1 hour ago
passed	#147526256 latest		master -> e8265e7d refactor: rename file to reflect ...		00:05:36 1 hour ago
passed	#147526022		master -> 908aba61 refactor: use develop environm...		00:05:24 1 hour ago

- Attention : vous avez suffisamment de permissions pour détruire les images docker (les vôtres et celles des autres groupes) sur le container registry. Procédez donc avec prudence si vous tentez de supprimer une image.

7. Aller plus loin

A l'issue de ce projet ML3, vous noterez probablement que certaines parties pourraient être très largement améliorées. Nous listons ci après quelques points qui mériteraient une mise à jour, si vous en avez le temps...

- Ajout de tests fonctionnels : l'application n'est pas testée de bout en bout. Entres autres, l'API n'est pas testée. Cependant, cela requiert le déploiement d'un server test, ce qui complexifie beaucoup le pipeline de CI/CD.
- Séparation des databases par environnement : les environnements de develop, staging et production partagent la même base de données. Il faudrait les séparer. Cette amélioration est néanmoins hors périmètre car vous n'avez pas les droits suffisants.
- Entraînement dynamique des modèles : les modèles de machine learning pourraient être ré-entraînés si la base de données est modifiée
- Versioning des modèles : chaque nouveau modèle devrait être taggé et stocké séparément dans le cloud (par exemple google cloud storage) au lieu de faire partie de l'image docker
- Versioning des images docker : chaque image docker devrait être taggé différemment, par exemple en utilisant le hash du commit qui l'a généré
- Optimisation du chargement du modèle : pour chaque prédiction, le modèle est chargé depuis le fichier, ce qui peut augmenter le temps de réponse de l'API. Idéalement, il faudrait le charger une seule fois au démarrage de l'application.
- Log des erreurs : si une erreur arrive, il faudrait la capturer et l'envoyer vers une application tierce (ex: sentry)

Eléments à fournir aux participants

- Lien vers gitlab code de départ
- Lien vers gitlab repository du groupe
- Clé JSON application python > cloud SQL
- Fichiers config contenant les Informations de connexion à la database