Projet de session

Informations générale

L'objectif du projet de session est de développer et déployer une application Web responsable du paiement de commandes Internet.

Le projet est divisé en deux étapes pour chacune des remises.

Objectifs

- Se familiariser avec le développement Web
- Développer une API REST
- Utiliser des services Web distants
- S'assurer de la résilience et de la performance d'une application Web
- Utiliser un gestionnaire de tâches

Équipe

Le travail se fait individuellement ou en équipe de 2.

Évaluation

La correction se fera à distance sans la présence du ou des membres de l'équipe ; toutefois, une **présentation** du fonctionnement de l'application aura lieu lors des semaines 14 et 15 (du 13 au 20 avril 2023).

Le code de l'application doit être hébergé sur Github dans un repo privé.

Remise

Date de remise : jeudi le 13 avril 2023.

Pour créer un projet, vous devez suivre les instruction suivantes :

1. Créer un compte Github si ce n'est déjà pas le cas.
2. Votre dépôt Github privé sera créé, vous pouvez commencer à l'utiliser.
3. M'ajouter comme collaborateur afin que je puisse accéder à votre repo. (adresse jgnault@uqac.ca)

Deuxième remise

La deuxième partie du projet de session se concentre sur la maintenance de l'application : déploiement, ajout de fonctionnalités, performance et résilience.

Base de donnée

Vous devez changer la base de donnée `sqlite` pour PostgreSQL. Les informations de connexion à PostgreSQL sont transmises par les variables d'environnements suivantes :

- `DB_HOST` : l'hôte pour se connecter à la base de donnée
- `DB_USER` : le nom d'usager de la base de donnée
- `DB_PASSWORD` : le mot de passe de la base de donnée
- `DB_PORT` : le port de connexion de la base de donnée
- `DB_NAME` : le nom de la base de donnée

Vous n'avez pas à migrer les données existantes.

N.B.: Vous n'avez pas à gérer la création de la base de donnée pour cette remise. Seulement la création des tables.

Initialisations des tables

Lors de la correction de cette remise, les tables de la base de donnée seront créées avec la commande suivante :

```
$ SET FLASK_DEBUG=True& SET FLASK_APP=api8inf349& SET REDIS_URL=redi
s://localhost& SET DB_HOST=localhost& SET DB_USER=user& SET DB_PASSW
ORD=pass& SET DB_PORT=5432& set DB_NAME=api8inf349
```

```
$ flask init-db
```

Les informations de la base de donnée (hôte, utilisateur, etc.) sont données à titre d'exemple seulement ; le localhost peut être remplacé par host.docker.internal.

Redis

L'application doit se connecter à une base de donnée Redis. À la différence de Postgres, une seule variable d'environnement va être exposée contenant l'URL de connexion.

e.g.: `redis://h:5326b83532892b4c@ec2-34-199-32-13.compute-1.amazonaws.com:6889` ou tout simplement `redis://localhost`

- `REDIS_URL` : l'url de connexion à Redis

Docker

Dockerfile

Vous devez produire un fichier `Dockerfile` valide à la racine du projet. Ce fichier doit produire une image Docker de votre application avec toutes les

dépendances requises pour rouler l'application (Python, Flask, Peewee, etc). Cette image ne doit pas contenir les services externes tel que Postgres, Redis.

Ce fichier doit pouvoir bâtir l'image Docker avec la commande suivante :

```
$ docker build -t api8inf349 .
```

Et l'application Web doit pouvoir être lancée avec la commande suivante :

```
$ docker run -e REDIS_URL=redis://localhost -e DB_HOST=localhost -e
DB_USER=user -e DB_PASSWORD=pass -e DB_PORT=5432 -e DB_NAME=api8inf3
49
```

Docker Compose

Vous devez également ajouter un fichier `docker-compose.yml` qui sera responsable de lancer les deux dépendances suivantes :

- PostgreSQL version 12
- Redis version 5

PostgreSQL doit utiliser un volume afin de persister les données entre chaque instanciation de l'image Docker.

Redis n'a pas besoin de volume.

Chacun des deux services doit exposer leurs ports respectifs :

- `5432` pour Postgres
- `6379` pour Redis

Commande

L'API de création d'une commande doit être modifiée pour permettre de créer une commande avec plus d'un produit.

```
POST /order
Content-Type: application/json

{ "products": [
        { "id": 123, "quantity": 2 },
        { "id": 321, "quantity": 1 },
    ]
}
```

Afin de garder une rétrocompatibilité, la création d'une commande avec un seul produit doit également être supporté.

Les exigences restent les mêmes qu'à la première remise.

Le format de réponse de l'affichage d'une commande **doit** être changé pour supporter cette nouvelle fonctionnalité.

```
GET /order/<int:order_id>
Content-Type: application/json
```

```
200 OK
```

```
{
   "order" : {
      "id" : 6543,
      "total_price" : 9148,
      "email" : null,
      "credit_card": {},
      "shipping_information" : {},
      "paid": false,
      "transaction": {},
      "products" : [
          {
             "id" : 123,
             "quantity" : 2
          },
          {
             "id" : 321,
             "quantity" : 1
          }
      ],
      "shipping_price" : 1000
   }
}
```

Les champs `total_price` et `shipping_price` doivent être adaptés également et respecter cette nouvelle fonctionnalité.

Résilience

Une fois qu'une commande a été payée, celle-ci ne peut pas être modifiée. Afin de répondre à un besoin de résilience, vous devez implémenter un système de mise en cache.

Lorsqu'une commande est payée, celle-ci doit être persistée dans la base de donnée `Postgres` et elle doit être mise en cache dans `Redis`.

Lors de l'affichage d'une commande avec `GET /order/<int:order_id>` vous devez vérifier en premier si la commande a été mise en cache dans `Redis`. Si c'est le cas, vous devez utiliser les informations de la commande à partir de `Redis`, et non `Postgres`.

Si la commande a été mise en cache, la route `GET /order/<int:order_id>` doit fonctionner sans `Postgres`.

Extraction du système de paiement

Vous devez extraire le système de paiement dans un gestionnaire de tâche en arrière-plan.

Pour ce faire vous devez utiliser la librairie de gestion de tâches `RQ` (RedisQueue): https://github.com/rq/rq

L'exécution des paiements doit se faire en arrière-plan.

La commande suivante sera utilisée pour lancer le gestionnaire de tâches :

```
$ SET FLASK_DEBUG=True& FLASK_APP=api8inf349& REDIS_URL=redis://loca
lhost& DB_HOST=localhost& DB_USER=user& DB_PASSWORD=pass& DB_PORT=54
32& DB_NAME=api8inf349

$ flask worker (ou $ python -m flask worker)
```

Lorsqu'une commande est en train d'être payée, le code HTTP 202 doit être retourné avec aucun corps de réponse.

```
PUT /order/<int:order_id>
Content-Type: application/json

{
   "credit_card" : {
      "name" : "John Doe",
      "number" : "4242 4242 4242 4242",
      "expiration_year" : 2024,
      "cvv" : "123",
      "expiration_month" : 9
   }
}

202 Accepted
```

Lorsqu'une commande est en train d'être payée, l'affichage de celle-ci doit retourner un code HTTP 202 avec aucun corps de réponse.

```
GET /order/<int:order_id>
Content-Type: application/json

202 Accepted
```

Une erreur doit être retournée lorsqu'on essaie de modifier une commande qui est en train d'être payée. L'erreur est le code HTTP 409.

```
PUT /order/<int:order_id>
Content-Type: application/json

409 Conflict
```

Et une fois que la commande est payée, celle-ci est retourné en format JSON avec un code HTTP 200.

```
GET /order/<int:order_id>
Content-Type: application/json

200 OK
Content-Type: application/json
```

```
{
   "order" : {
      "shipping_information" : {
         "country" : "Canada",
         "address" : "201, rue Président-Kennedy",
         "postal_code" : "H2X 3Y7",
         "city" : "Chicoutimi",
         "province" : "QC"
      },
      "email" : "jgnault@uqac.ca",
      "total_price" : 9148,
      "paid": true,
      "products" : [
         {
           "id" : 123,
           "quantity" : 1
         },
         {
           "id" : 321,
           "quantity" : 2
         }
      ],
      "credit_card" : {
         "name" : "John Doe",
         "first_digits" : "4242",
         "last_digits": "4242",
         "expiration_year" : 2024,
         "expiration_month" : 9
      },
      "transaction": {
          "id": "wgEQ4zAUdYqpr21rt8A10dDrKbfcLmqi",
          "success": true,
          "error": {},
          "amount_charged": 10148
      },
      "shipping_price" : 1000,
      "id" : 6543
   }
}
```

Erreur de paiement

Lorsque le service de paiement distant retourne une erreur de paiement, l'erreur doit être persistée dans la base de donnée, et être retournée sur l'objet `transaction`. Lors de l'appel `GET` sur la commande, on retourne quand même un statut HTTP 200.

N.B.: Seul les erreurs retournées par le service distant sont persistées. Le comportement pour les erreurs de validation côté client (commande déjà payée, champs manquant) ne changent pas.

```
GET /order/<int:order_id>
Content-Type: application/json

200 OK
Content-Type: application/json

{
   "order" : {
      "credit_card" : {},
      "transaction": {
          "success": false,
          "error": {
            "code": "card-declined",
            "name": "La carte de crédit a été déclinée."
          },
          "amount_charged": 10148
      },
      "paid": false,
      "shipping_information" : {
         "country" : "Canada",
         "address" : "201, rue Président-Kennedy",
         "postal_code" : "H2X 3Y7",
         "city" : "Chicoutimi",
         "province" : "QC"
      },
      "email" : "jgnault@uqac.ca",
      "total_price" : 9148,
      "products" : [
         {
           "id" : 123,
           "quantity" : 1
         },
         {
           "id" : 321,
           "quantity" : 2
         }
      ],
      "shipping_price" : 1000,
      "id" : 6543
   }
}
```

Interface utilisateur (front-end) HTML

Vous devez créer des pages Web permettant l'interaction avec les différents cas d'utilisation (use cases) de votre API. Pour cette fonctionnalité, les langages permis sont HTML et Javascript et/ou un outil de templating comme Jinja2.

Vous devez donc créer des pages Web permettant de contacter minimalement chacune de vos routes définies dans vos APIs avec les méthodes GET et POST (vous ne pouvez pas envoyer autre méthode via un formulaire HTML).

Exigences techniques

1. **Un fichier nommé `CODES-PERMANENTS` doit être à la racine de votre projet et contenir le ou les codes permanents séparés par un saut de ligne**
   - Donc pour un travail fait individuellement, le fichier doit simplement contenir votre code permanent
   - Pour une équipe de deux, le fichier doit contenir les code permanent des deux étudiants, un par ligne.
2. Un lien vers votre dépôt Github si celui-ci est public, ou encore m'ajouter comme collaborateur si votre dépôt est privé (recommandé, mon compte est jgnault@uqac.ca).
3. Le projet devra rouler sous Python 3.6+ et Flask 1.11+
4. À l'exception de Flask, peewee et RQ, il n'y a aucune restriction sur les paquets à utiliser. Toutes les dépendances nécessaires doivent être dans le fichier `requirements.txt`.
   - Lors de la correction, la commande `pip install -r requirements.txt` sera utilisée pour installer les dépendances Python.
5. La base de données utilisée est PostgreSQL
6. Vous devez utiliser l'ORM `peewee` et le gestionnaire de tâches `RQ` (https://python-rq.org/)
7. Toutes les données doivent être stockées dans la base de donnée
8. La base de données doit être initialisée avec la commande mentionnée précédemment
9. À partir de la racine de votre projet, l'application doit pouvoir rouler avec la commande mentionnée précédemment et le gestionnaire de tâche pareillement

Astuce : pour parvenir à communiquer entre votre application dans un conteneur, et Postgres/Redis dans un autre conteneur, vous devez remplacer les localhost par host.docker.internal dans votre commande docker run.

Critères d'évaluations

- 25% : Respect des **exigences techniques**
- 10% : `Dockerfile` et `docker-compose.yml` fonctionnels
- 5% : Modification de l'API pour permettre la création d'une commande avec plus d'un produit
- 10% : Résilience
- 20% : Extraction du système de paiement
- 10% : Interface utilisateur (front-end) HTML
- 10% : Qualité du code
- 10% : Présentation à l'enseignant