Projet de session

Historique

#	Date	Description
1	25 septembre	Version initiale
2	25 septembre	Précisions sur les remises
3	25 septembre	Correction au diagramme de séquence
4	26 septembre	Instructions de remise sur Github
5	7 octobre	Ajout d'une exigence lors du paiement d'une commande et ajout de précisions sur la récupération des produits
6	11 octobre	Rajout d'une exigence pour la création d'une commande et quelques précisions
7	18 novembre	Deuxième remise
8	27 novembre	Précisions sur les erreurs de paiement
9	4 décembre	Correction de coquille sur le schéma JSON et la version de Postgres

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
- Déployer une application Web

Équipe

Le travail se fait individuellement ou en équipe de 2.

Échéances

Date	Description	Pondération
4 novembre 21h	Première remise	15%
16 décembre 21h	Remise finale	15%

Le projet de session représente 30% de la note globale du cours.

Évaluation

L'évaluation se fera à distance sans la présence du ou des membres de l'équipe.

Le code de l'application doit être hébergé sur Github.

Aux deux dates de la remise, à 21h, le dépôt sera cloné et c'est ce qui sera utilisé pour l'évaluation.

Vous devez garder le projet fonctionnel sur Heroku jusqu'au 6 janvier 2020.

Remise

La création du projet sur Github est automatisé grâce à Github Classroom.

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

1. Créer un compte Github si ce n'est déjà pas le cas.
2. Rendez-vous sur la page du projet de session :

   https://classroom.github.com/g/oH9-4nj2
3. Acceptez le travail, et identifier vous dans la liste en choisissant votre code permanent **Assurez-vous de ne pas prendre le code de quelqu'un d'autre!**
4. Créer une équipe individuelle, ou rejoignez l'équipe de votre collègue si vous faites la remise à 2.
5. Votre dépôt Github privé sera créé, vous pouvez commencer à l'utiliser.

Langage de programmation

Le langage de programmation pour le projet de session est Python. La version minimale qui sera utilisé est 3.6. Vous pouvez utiliser la version 3.7 si vous le désirez.

Le cadriciel de développement Web est Flask 1.1.

Le démonstrateur de laboratoire sera en mesure de vous apporter du support et du soutien technique pour ces deux technologies.

Le projet

Le projet consiste à développer une application Web responsable de prendre des commandes Internet. Cette application devra répondre à une API REST.

Le projet est séparé en deux. Les informations pour la première remise sont archivés et disponible sur Github.. La section pour la deuxième remise est disponible ci-bas.

Il s'agit de remises incrémentiels sur le même projet. Pour la deuxième remise, vous continuerez à utiliser le projet.

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 :

```
$ FLASK_DEBUG=True FLASK_APP=inf5190 REDIS_URL=redis://localhost
↪  DB_HOST=localhost DB_USER=user DB_PASSWORD=pass DB_PORT=5432
↪  DB_NAME=inf519 flask init-db
```

Les informations de la base de donnée (hôte, utilisateur, etc) sont données à titre d'exemple seulement.

Redis

L'application doit se connecter à une base de donnée Redis. À la différence de Postgres, une seule variable d'environnement va être exposé 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 inf5190 .
```

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=inf519 inf5190
```

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 fonctionnalités.

```
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 modifié. Afin de répondre à un besoin de résilience vous devez implémenter un système de mise à la 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 pas `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 :

```
$ FLASK_DEBUG=True FLASK_APP=inf5190 REDIS_URL=redis://localhost
↪  DB_HOST=localhost DB_USER=user DB_PASSWORD=pass DB_PORT=5432
↪  DB_NAME=inf519 flask worker
```

Lorsqu'une commande est entrain 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 entrain 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
```

Un erreur doit être retourné lorsqu'on essaie de modifier une commande qui est entrain 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" : "Montréal",
         "province" : "QC"
      },
      "email" : "caissy.jean-philippe@uqam.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 un erreur de paiement, l'erreur doit être persisté dans la base de donnée, et être retourné 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 retournés par le service distant sont persistés. 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" : "Montréal",
```

```
        "province" : "QC"
      },
      "email" : "caissy.jean-philippe@uqam.ca",
      "total_price" : 9148,
      "products" : [
        {
          "id" : 123,
          "quantity" : 1
        },
        {
          "id" : 321,
          "quantity" : 2
        }
      ],
      "shipping_price" : 1000,
      "id" : 6543
   }
}
```

Déploiement

Votre application doit être déployée sur Heroku. L'URL complet de l'application (i.e.: `https://<application>.herokuapp.com`) doit être fournie dans un fichier `HEROKU` à la racine de votre projet.

L'application déployée doit utiliser PostgreSQL (i.e.: `Heroku Postgres`) et Redis (i.e.: `Heroku Redis`).

Vous n'êtes pas obligé de payer pour Heroku, les forfaits de base gratuit pour l'application, ainsi que pour PostgreSQL et Redis sont suffisant pour cette remise.

Vous devez obligatoirement rajouter mon compte (`jean-philippe.caissy@uqam.ca`) en tant que collaborateur à votre application Heroku. Sinon je ne pourrai pas corriger l'objectif de déploiement et vous aurez 0% pour cette exigence.

Les instructions pour rajouter un collaborateur sont disponible ici : https://devcenter.heroku.com/articles/collaborating#add-collaborators

Le système de paiement en arrière plan doit également être fonctionnel sur Heroku.

Exigences techniques

1. **Un fichier nommé `CODES-PERMANENTS` doit être à la racine de votre projet et contenir le ou les codes permanents séparé par un saut de ligne**

- Donc pour un travail fait individuellement, le fichier doit simplement contenir votre code permanent
- Pour un équipe de deux, le fichier doit contenir les code permanent des deux étudiants, un par ligne.

2. **Un fichier nommé `HEROKU` contenant l'adresse de votre application sur Heroku**
3. Votre dépôt Github doit avoir été créé avec Github Classroom (les instructions sont dans la section **Remise**
4. Le projet devra rouler sous Python 3.6+ et Flask 1.11+
5. 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.
6. La base de données utilisée est PostgreSQL
7. Vous devez utiliser l'ORM `peewee` et le gestionnaire de tâches `RQ` (https://python-rq.org/)
8. Toutes les données doivent être stockés dans la base de donnée
9. La base de données doit être initialisée avec

   ```
   FLASK_DEBUG=True FLASK_APP=inf5190
   ↪ REDIS_URL=redis://localhost DB_HOST=localhost
   ↪ DB_USER=user DB_PASSWORD=pass DB_PORT=5432
   ↪ DB_NAME=inf519 flask init-db
   ```

10. À partir de la racine de votre projet, l'application doit pouvoir rouler avec la commande suivante :

    ```
    FLASK_DEBUG=True FLASK_APP=inf5190
    ↪ REDIS_URL=redis://localhost DB_HOST=localhost
    ↪ DB_USER=user DB_PASSWORD=pass DB_PORT=5432
    ↪ DB_NAME=inf519 flask run
    ```

    et le gestionnaire de tâche avec

    ```
    FLASK_DEBUG=True FLASK_APP=inf5190
    ↪ REDIS_URL=redis://localhost DB_HOST=localhost
    ↪ DB_USER=user DB_PASSWORD=pass DB_PORT=5432
    ↪ DB_NAME=inf519 flask worker
    ```

Critères d'évaluations

- 20% : Respect des **exigences techniques**

- 10% : `Dockerfile` et `docker-compose.yml` fonctionnels
- 10% : 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% : Déploiement sur Heroku
- 20% : Qualité du code

N.B. : Toutes les *exigences techniques* mentionnées ci-haut doivent être respectés, sinon c'est 0/20.

Première remise

Les informations sur la première remise sont disponibles sur Github : https://github.com/jpcaissy/INF5190/blob/b55de3f413234b3217d6df2f0cf9f5f52880f2dd/travail-de-session/enonce.md#premi%C3%A8re-remise.