Labo 08 — Rapport

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
Université du Québec

Jean-Christophe Benoit
Rapport de laboratoire
LOG430 — Architecture logicielle
Montréal, 13 novembre 2025
École de technologie supérieure

> J'ai crée ce PDF en utilisant le gabarit markdown (.md).

Questions

Question 1

> Comment on faisait pour passer d'un état à l'autre dans la saga dans le labo 6, et comment on le fait ici? Est-ce que le contrôle de transition est fait par le même structure dans le code?

Dans le labo 6, notre commande passait d'un état à l'autre à travers une requête envoyé au controleur Saga qui utilisait une boucle qui vérifie l'état actuel de la commande et exécute un handler dépendant de cet état. Ces handlers retournaient chacun un état de commande et la boucle était exécutée de la même façcon jusqu'à ce que la commande soit dans l'état `COMPLETED` (que ce soit à cause d'une réelle complétion de toutes les étapes ou parce que la commande a été annulée dû à un problème) :

On voit ci-dessous la boucle responsable du changement d'état :

```
while self.current_saga_state is not OrderSagaState.COMPLETED:
    self.logger.debug(f"Current sgag state: {self.current_saga_state}")
    if self.current_saga_state == OrderSagaState.CREATING_ORDER:
        self.current_saga_state = self.create_order_handler.run()
    elif self.current_saga_state == OrderSagaState.DECREASING_STOCK:
        self.increase_stock_handler =
DecreaseStockHandler(order_data["items"])
        self.current_saga_state = self.increase_stock_handler.run()
    elif self.current_saga_state == OrderSagaState.CREATING_PAYMENT:
        self.create_payment_handler =
CreatePaymentHandler(self.create_order_handler.order_id, order_data)
        self.current_saga_state = self.create_payment_handler.run()
    elif self.current_saga_state == OrderSagaState.INCREASING_STOCK:
        self.current_saga_state = self.increase_stock_handler.rollback()
    elif self.current_saga_state == OrderSagaState.CANCELLING_ORDER:
        self.current_saga_state = self.create_order_handler.rollback()
    else:
```

```
        self.is_error_occurred = True
        self.logger.debug(f"L'état saga n'est pas valide :
{self.current_saga_state}")
        self.current_saga_state = OrderSagaState.COMPLETED
```

Dans ce labo ci, le changement d'état est géré par l'envoi d'évennements pour le topic `order-saga-events` avec le courtier Kafka. Les évennements produits par chaque classe représente l'état actuel de la commande. Les évennements sont ensuite consommés par un handler qui lui produit un nouvel évennement, donc état, jusqu'à ce que l'évennement soit `SagaCompleted`.

On voit ci-dessus un exemple du passage de l'état OrderCreated à l'état `StockDecreased` ou `StockDecreaseFailed` : `write_order.py`

```
...

    add_order_to_redis(order_id, user_id, total_amount, items)

    # Déclencher l'événement OrderCreated
    event_data = {'event': 'OrderCreated',
                  'order_id': new_order.id,
                  'user_id': new_order.user_id,
                  'total_amount': new_order.total_amount,
                  'is_paid': new_order.is_paid,
                  'payment_link': new_order.payment_link,
                  'order_items': items,
                  'datetime': str(datetime.now())}
    return order_id

except Exception as e:
    # Déclencher l'événement OrderCreationFailed
    event_data['error'] = str(e)
    session.rollback()
    raise e
finally:
    OrderEventProducer().get_instance().send(config.KAFKA_TOPIC,
value=event_data) #<=== ici on passe à l'état OrderCreated, le premier
état de la Saga.
    session.close()
```

`order_create_handler.py`

```
def handle(self, event_data: Dict[str, Any]) -> None:
    """
    This method is here as a reference for the implementation of the
method handle.
    It will never be called if Sotre Manager is following normal
operation.
    Once you copy-paste the implementation, you can delete this method if
you want.
```

```
    """
    order_event_producer = OrderEventProducer()
    try:
        # La création de la comande a réussi, alors déclenchez la mise à
jour du stock.
        session = get_sqlalchemy_session()
        check_out_items_from_stock(session, event_data['order_items'])
        session.commit()
        # Si la mise à jour du stock a réussi, déclenchez StockDecreased.
        event_data['event'] = "StockDecreased"
    except Exception as e:
        session.rollback()
        # Si la mise à jour du stock a échoué, déclenchez
StockDecreaseFailed.
        event_data['event'] = "StockDecreaseFailed"
        event_data['error'] = str(e)
    finally:
        session.close()
        order_event_producer.get_instance().send(config.KAFKA_TOPIC,
value=event_data) #<=== Ici, on passe de l'état Order Created à l'état
StockDecreased ou StockDecreaseFailed
```

On peut évidemment observer que la transition n'est pas faite par la même structure de code, car nous sommes passé d'une boucle évaluant chaque état à une architecture contenant des producteurs, des consommateurs et un courtier d'évennements.

Question 2

> Sur la relation entre nos Handlers et le patron CQRS : pensez-vous qu'ils utilisent plus souvent les Commands ou les Queries? Est-ce qu'on tient l'état des Queries à jour par rapport aux changements d'état causés par les Commands?

Si on parle spécifiquement de l'utilisation de *queries* vs l'utilisation des *commands* lorsqu'un utilisateur place une commande, alors on peut clairement voir que les *commands* sont plus utilisées car il n'y a pas d'opérationde lecture de données lors de la Saga de placement d'une commande. Par exempl :

`order_created_handler.py` On utilise la *command* `write_stock.check_out_items_to_stock` pour diminuer le stock d'un produit

```
order_event_producer = OrderEventProducer()
    try:
        # La création de la comande a réussi, alors déclenchez la mise à
jour du stock.
        session = get_sqlalchemy_session()
        check_out_items_from_stock(session, event_data['order_items'])
        session.commit()
        # Si la mise à jour du stock a réussi, déclenchez StockDecreased.
        event_data['event'] = "StockDecreased"
    except Exception as e:
        session.rollback()
        # Si la mise à jour du stock a échoué, déclenchez
```

```
StockDecreaseFailed.
        event_data['event'] = "StockDecreaseFailed"
        event_data['error'] = str(e)
```

`stock_decrease_failed_handler.py` On va utiliser la *command* `write_stock.check_in_items_to_stock` pour augmenter le stock d'un produit.

Les *queries* faites pendant l'exécution d'une Saga de placement de commande vont récupérer l'état actuel de la commande dans Redis. Donc, si une commande a été créé, mais que le lien de paiement n'a pas encore été créé et ajouter à la commande, une requête pour lire cette commande va réfléter cet état :

`read_order.py`

```
def get_order_by_id(order_id):
    """Get order by ID from Redis"""
    r = get_redis_conn()
    raw_order = r.hgetall(f"order:{order_id}")
    order = {}
    for key, value in raw_order.items():
        found_key = key.decode('utf-8') if isinstance(key, bytes) else key
        found_value = value.decode('utf-8') if isinstance(value, bytes)
else value
        order[found_key] = found_value
    return order
```

Question 3

> Est-ce qu'une architecture Saga orchestrée pourrait aussi bénéficier de l'utilisation du patron Outbox, ou c'est un bénéfice exclusif de la saga chorégraphiée? Justifiez votre réponse avec un diagramme ou en faisant des références aux classes, modules et méthodes dans le code.

De la façon dont nous avions implémenté Saga orchestrée dans le labo 6, on ne pourrait bénéficier de l'utilisation du patron Outbox, car le code s'attend à une réponse synchrone du payments-api pour continuer la saga :

```
while self.current_saga_state is not OrderSagaState.COMPLETED:
    # TODO: vérifier TOUS les 6 états saga. Utilisez run() ou rollback()
selon les besoins.
    self.logger.debug(f"Current sgag state: {self.current_saga_state}")
    if self.current_saga_state == OrderSagaState.CREATING_ORDER:
        self.current_saga_state = self.create_order_handler.run()
    elif self.current_saga_state == OrderSagaState.DECREASING_STOCK:
        self.increase_stock_handler =
DecreaseStockHandler(order_data["items"])
        self.current_saga_state = self.increase_stock_handler.run()
    elif self.current_saga_state == OrderSagaState.CREATING_PAYMENT:
        self.create_payment_handler =
CreatePaymentHandler(self.create_order_handler.order_id, order_data) #
```

```
<=== Ici, on attend une réponse de notre handler, mais avec outbox, un
évennement serait produit.
        self.current_saga_state = self.create_payment_handler.run()
    elif self.current_saga_state == OrderSagaState.INCREASING_STOCK:
        self.current_saga_state = self.increase_stock_handler.rollback()
    elif self.current_saga_state == OrderSagaState.CANCELLING_ORDER:
        self.current_saga_state = self.create_order_handler.rollback()
    else:
        self.is_error_occurred = True
        self.logger.debug(f"L'état saga n'est pas valide :
{self.current_saga_state}")
        self.current_saga_state = OrderSagaState.COMPLETED
```

Cependant, il est possible de bénéficier du patron outbox avec Saga orchestré en changeant l'implémentation de l'orchestrateur pour qu'il utilise un consommateur d'évennements et des handlers asynchrones comme on a fait dans ce labo ci.

Question 4

> Qu'est-ce qui arriverait si notre application s'arrête avant la création de l'enregistrement dans la table Outbox? Comment on pourrait améliorer notre implémentation pour résoudre ce problème? Justifiez votre réponse avec un diagramme ou en faisant des références aux classes, modules et méthodes dans le code.

Si l'application s'arrête avant la création de la ligne dans la table Outbox, nous avons perdu un évennement et le paiement ne sera jamais créé, car kafka aura déjà auto-commiter le offset et le OutboxProcessor ne traitera jamais la requête.

Pour améliorer cela, il faut rendre toute cette opération atomique il faudra :

- Mettre auto-commit de kafka à False :

```
enable_auto_commit=False
```

- Commit l'offset seulement lorsque `session.commit` a roulé pour créer la ligne Outbox.
- Mettre en place une job asynchrone qui scan régulièrement la table Outbox pour traiter les items s'y trouvant.

De cette façon, on ne perd pas d'évennement et le consommateur avancera seulement l'offset lorsqu'on est sûr que la ligne Outbox a été créé.

Observations additionnelles

Problème(s) rencontré(s)

- Problème lors de l'ajout d'un consommateur d'évennement dans payments-api, car ce consommateur consommait les évennements du order saga avant le store manager et il n'avait pas de handler pour l'évennement initial OrderCreated.

    - Solution : Ajout d'un topic pour chaque service, donc nous avons un topic pour les events que le store manager veut consommer et un topic pour les events que payments-api veut consommer et iil n'y a plus de conflit entre les consommateurs d'events.
- Problème de consommation infinie d'events PaymentCreated lorsque l'event vient du payments-api.
    - Mon handler pour l'event PaymentCreated ne changeait pas la valeur event dans event_data, ce qui causait un feedback loop.
- Problème lors de la mise à jour des commandes auprès de redis pour que les tests passent.
    - Ajout de la mise à jour dans redis lors de l'ajout du payment link dans une commande et ajout d'un sleep pour que le processus asynchrone avec Outbox se complète avant de récupérer la commande.