Labo 07 — Rapport

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
Université du Québec

Jean-Christophe Benoit
Rapport de laboratoire
LOG430 — Architecture logicielle
Montréal, 5 novembre 2025
École de technologie supérieure

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

Questions

Question 1

> Quelle est la différence entre la communication entre store_manager et coolriel dans ce labo et la communication entre store_manager et payments_api que nous avons implémentée pendant le labo 5 ? Expliquez avec des extraits de code ou des diagrammes et discutez des avantages et des inconvénients.

La différence principale entre la communication entre le store-manager et le payments-api et le store-manager et coolriel est que le store-manager envoie des évennements à travers un producteur d'évenements du côté du store-manager et un un consommateur d'évennements du côté de coolriel :

Envoi de l'évennement par Store manager :

```
user_event_producer = UserEventProducer()
user_event_producer.get_instance().send('user-events', value={'event':
'UserCreated',
                                        'id': new_user.id,
                                        'name': new_user.name,
                                        'email': new_user.email,
                                        'datetime':
str(datetime.datetime.now())})
```

Réception de l'évenement par coolriel :

```
registry = HandlerRegistry()
registry.register(UserCreatedHandler(output_dir=config.OUTPUT_DIR))
registry.register(UserDeletedHandler(output_dir=config.OUTPUT_DIR))
```

```
# NOTE: le consommateur peut écouter 1 ou plusieurs topics (str or array)
consumer_service = UserEventConsumer(
    bootstrap_servers=config.KAFKA_HOST,
    topic=config.KAFKA_TOPIC,
    group_id=config.KAFKA_GROUP_ID,
    registry=registry,
)
consumer_service.start()
```

Cependant, la communication vers le payments-api est faite à travers des requêtes HTTP :

```
response_from_payment_service = requests.post(
    'http://api-gateway:8080/payments-api/payments',
    json=payment_transaction,
    headers={'Content-Type': 'application/json'}
)
```

Question 2

> Quelles méthodes avez-vous modifiées dans src/orders/commands/write_user.py? Illustrez avec des captures d'écran ou des extraits de code.

Pour la génération d'un email html lors de la suppression d'un utilisateur, les méthodes suivantes ont été modifiées :

user_deleted_handler.py

```
def handle(self, event_data: Dict[str, Any]) -> None:
    """Create an HTML email based on user deletion data"""
    self.logger.debug(event_data)
    user_id = event_data.get('id')
    name = event_data.get('name')
    email = event_data.get('email')
    datetime = event_data.get('datetime')

    current_file = Path(__file__)
    project_root = current_file.parent.parent
    with open(project_root / "templates" / "goodbye_client_template.html",
'r') as file:
        html_content = file.read()
        html_content = html_content.replace("{{user_id}}", str(user_id))
        html_content = html_content.replace("{{name}}", name)
        html_content = html_content.replace("{{email}}", email)
        html_content = html_content.replace("{{deletion_date}}", datetime)

    filename = os.path.join(self.output_dir, f"goodbye_{user_id}.html")
    with open(filename, 'w', encoding='utf-8') as f:
        f.write(html_content)
```

```
        self.logger.debug(f"Courriel HTML généré à {name} (ID: {user_id}),
    {filename}")
```

write_user.py (dans store-manager)

```
    def delete_user(user_id: int):
    """Delete user in MySQL"""
    session = get_sqlalchemy_session()
    try:
        user = session.query(User).filter(User.id == user_id).first()
        if user:
            session.delete(user)
            session.commit()

            user_event_producer = UserEventProducer()
            user_event_producer.get_instance().send('user-events', value=
    {'event': 'UserDeleted',
                                            'id': user.id,
                                            'name': user.name,
                                            'email': user.email,
                                            'user_type_id': user.user_type_id,
                                            'datetime':
    str(datetime.datetime.now())})
```

Ensuite, pour ajouter le type d'utilisateur, les méthodes suivantes ont été modifiées :

user_controller.py

```
    def create_user(request):
        """Create user, use WriteUser model"""
        payload = request.get_json() or {}
        name = payload.get('name')
        email = payload.get('email')
        user_type_id = payload.get('user_type_id')
        try:
            user_id = add_user(name, email, user_type_id)
```

write_user.py

```
    def add_user(name: str, email: str, user_type_id: int):
        """Insert user with items in MySQL"""
        if not name or not email or not user_type_id:
            raise ValueError("Cannot create user. A user must have name, email
    and user type.")

        session = get_sqlalchemy_session()
```

```
    try:
        new_user = User(name=name, email=email, user_type_id=user_type_id)
        session.add(new_user)
        session.flush()
        session.commit()

        user_event_producer = UserEventProducer()
        user_event_producer.get_instance().send('user-events', value=
{'event': 'UserCreated',
                                        'id': new_user.id,
                                        'name': new_user.name,
                                        'email': new_user.email,
                                        'user_type_id':
new_user.user_type_id,
                                        'datetime':
str(datetime.datetime.now())})
```

user.py

```
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, nullable=False)
    email = Column(String, nullable=False)
    user_type_id = Column(Integer, ForeignKey('user_types.id'),
nullable=False)
```

user_types.py

```
class UserTypes(Base):
    __tablename__ = 'user_types'

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, nullable=False)
```

Question 3

> Comment avez-vous implémenté la vérification du type d'utilisateur ? Illustrez avec des captures d'écran ou des extraits de code.

J'ai ajouté le transfert du user_type_id dans le payload de l'évennement UserCreated. Ensuite j'utilise ce id avec un dictionnaire pour déterminer le message à afficher (le code ci-dessous montre les modifications apportées pour la création d'un nouvel utilisateur, mais les changements sont sensiblement les mêmes pour la suppression d'un utilisateur):

```
message_options = {
    1: "Merci d'avoir visité notre magazin. Si vous avez des questions ou
des problèmes concernant votre achat, n'hésitez pas à nous contacter.",
    2: "Salut et bienvenue dans l'équipe!",
    3: "Salut et bienvenue dans l'équipe!"
}

def handle(self, event_data: Dict[str, Any]) -> None:
    """Create an HTML email based on user creation data"""

    user_id = event_data.get('id')
    name = event_data.get('name')
    email = event_data.get('email')
    user_type_id = event_data.get('user_type_id')
    datetime = event_data.get('datetime')

    message = message_options.get(user_type_id)

    current_file = Path(__file__)
    project_root = current_file.parent.parent
    with open(project_root / "templates" / "welcome_client_template.html",
'r') as file:
        html_content = file.read()
        html_content = html_content.replace("{{user_id}}", str(user_id))
        html_content = html_content.replace("{{name}}", name)
        html_content = html_content.replace("{{email}}", email)
        html_content = html_content.replace("{{message}}", message)
        html_content = html_content.replace("{{creation_date}}", datetime)
```

Le template html a bien sûr aussi été modifé pour prendre le message en paramètre :

```
<body>
    <div class="container">
        <h1>🎉 Bienvenu·e {{name}}!</h1>
        <p>{{message}}</p>
        <p>Cordialement,</p>
        <p>Magazin du Coin</p>
        <p class="user-id">Votre ID Utilisateur: {{user_id}}</p>
        <p class="user-id">Votre adresse courriel: {{email}}</p>
        <p class="user-id">Crée à {{creation_date}}</p>
    </div>
</body>
```

Question 4

> Comment Kafka utilise-t-il son système de partitionnement pour atteindre des performances de lecture élevées ? Lisez cette section de la documentation officielle à Kafka et résumez les points principaux.

Les logs d'évennements pour chaque sujet (topic) sont divisés par partitions ordonnées, permettant à Kafka de distribuer ces partitions sur plusieurs machine pour que les lecteurs ou écriveurs puissent lire ou écrire de façon parallèle.

Vu que les partitions sont ordonnées et lues séquentiellement, cela augmente la vitesse de lecture.

De plus, la performance de lecture de Kafka reste constant même si la grandeur des données augmente, donc les performances ne se désintègre pas, même si on fait une rétention des évennements pour très longtemps.

Finalement, grâce à la consommation des évennements basée sur un offset, différents consommateurs peuvent lire la même partition sans affecter la lecture d'un autre consommateur.

Question 5

> Combien d'événements avez-vous récupérés dans votre historique ? Illustrez avec le fichier JSON généré.

J'ai ajouté et supprimé un utilisateur et donc je vois 2 évennements dnas le JSON généré :


```
{"event": "UserCreated", "id": 8, "name": "Jane Doe", "email":
"jd@exampled.ca", "user_type_id": 1, "datetime": "2025-11-06
21:22:05.325726"}
{"event": "UserDeleted", "id": 1, "name": "Ada Lovelace", "email":
"alovelace@example.com", "user_type_id": 1, "datetime": "2025-11-06
21:22:13.388895"}
```


Observations additionnelles

Problème(s) rencontré(s)

- Problème avec le chargement du schéma pour la table user types. Solution : Il faut ajouter un import de UserTypes lorsque User est utilisé même si cet import n'est pas utilisé, pour que SQLAlchemy charge le modèle de la table user_types et puissent créer des nouveaux utilisateurs avec un champ qui est un foreign key vers la table user_types.