

# LOG430 - Rapport du laboratoire 02

ÉTS - LOG430 - Architecture logicielle - Hiver 2026 - Groupe 1

Étudiant: Yanni Haddar Nom github: mapleduck repo github: <https://github.com/mapleduck/log430-labo5> et <https://github.com/mapleduck/log430-labo5-payment>

## Questions

💡 Question 1 : Quelle réponse obtenons-nous à la requête à POST /payments ? Illustrez votre réponse avec des captures d'écran/du terminal.

Le body de la requête spécifie manuellement l'existence d'un order, et nous devrions donc recevoir un id de paiement pour pouvoir le process:

```
{
  "user_id": 1,
  "order_id": 1,
  "total_amount": 99.53
}
```

Nous recevons tout simplement un ID de payment. Ce test a été fait après 4 tentatives de payment process (j'ai pris trop d'avance par rapport à l'activité), il s'agit simplement de l'incrémentation normale:

The screenshot shows the Postman interface. On the left, the 'My Workspace' sidebar lists collections like 'LOG430 Labo05 Payment' and 'log430-labo5'. In the center, a 'LOG430 Labo05 Payment / /payments' collection is selected. A POST request to 'localhost:5009/payments' is shown with the method dropdown set to 'POST'. The 'Body' tab contains the following JSON payload:

```
{
  "user_id": 1,
  "order_id": 1,
  "total_amount": 99.53
}
```

The 'Headers' tab includes 'Content-Type: application/json'. The 'Test Results' tab shows a successful response with status '201 CREATED', duration '42 ms', and size '193 B'. The response body is displayed as:

```
1 {
2   "payment_id": 5
3 }
```

💡 Question 2 : Quel type d'information envoyons-nous dans la requête à POST payments/process/:id ? Est-ce que ce serait le même format si on communiquait avec un service SOA, par exemple ? Illustrez votre réponse avec des exemples et captures d'écran/terminal.

Nous envoyons en JSON des informations de paiement d'une carte de crédit:

The screenshot shows the Postman interface. On the left, the 'My Workspace' sidebar lists collections like 'LOG430 Labo05 Payment' and environments like 'log430-labo05'. In the center, a POST request is being made to 'localhost:5009/payments/process/:id'. The 'Body' tab shows a raw JSON payload:

```

1 {
2   "cardNumber": "999999999999",
3   "cardCode": 123,
4   "expirationDate": "2030-01-05"
5 }

```

The response tab shows a 200 OK status with a response time of 44 ms and a size of 225 B. The response body is:

```

1 {
2   "is_paid": true,
3   "order_id": 13,
4   "payment_id": 4
5 }

```

En SOA, ces informations seraient dans un format plus strict, en XML, avec des Enveloppes SOAP et un format tr's spécifique (il faut respecter un fichier WSDL).

💡 Question 3 : Quel résultat obtenons-nous de la requête à POST payments/process/:id?

On peut voir dans l'image ci-haut que l'API répond avec l'order ID, le payment ID, et la confirmation que l'order a été marqué comme payé (is\_paid = true).

💡 Question 4 : Quelle méthode avez-vous dû modifier dans log430-labo05-payment et qu'avez-vous modifiée ? Justifiez avec un extrait de code.

La méthode `update_order` dans `payment_controller.py` est celle qui a dû être modifiée (l'implémentation de la méthode devait être faite). En gros, la méthode fait un call à l'API pour mettre à jour la commande et le fait qu'elle a été payée. Il a aussi fallu adapter le call de `update_order` plus haut dans le fichier:

1 file changed +20 -3 lines changed

src/controllers/payment\_controller.py

```
@@ -41,8 +41,8 @@ def process_payment(payment_id, credit_card_data):
41     "payment_id": update_result["payment_id"],
42     "is_paid": update_result["is_paid"]
43   }
44 - # TODO: appelez la méthode correctement
45 - update_order(0, False)
44 +
45 + update_order(update_result["order_id"], update_result["is_paid"])

46
47   return result
48

@@ -54,4 +54,21 @@ def _process_credit_card_payment(payment_data):
54
55   def update_order(order_id, is_paid):
56     """ Trigger order update once it is paid"""

57 - pass
57 ⊕
57 + payload = {
58 +   "order_id": order_id,
59 +   "is_paid": is_paid
60 + }
61 + try:
62 +   response = requests.put(
63 +     "http://api-gateway:8080/store-manager-api/orders",
64 +     json=payload,
65 +     headers={'Content-Type': 'application/json'}
66 +   )
67 +
68 +   if response.ok:
69 +     logger.debug(f"Commande {order_id} mise à jour avec succès.")
70 +   else:
71 +     logger.error(f"Erreur lors de la mise à jour de la commande: {response.status_code}")
72 +
73 + except Exception as e:
74 +   logger.error(f"Erreur de connexion au Gateway: {e}")


```

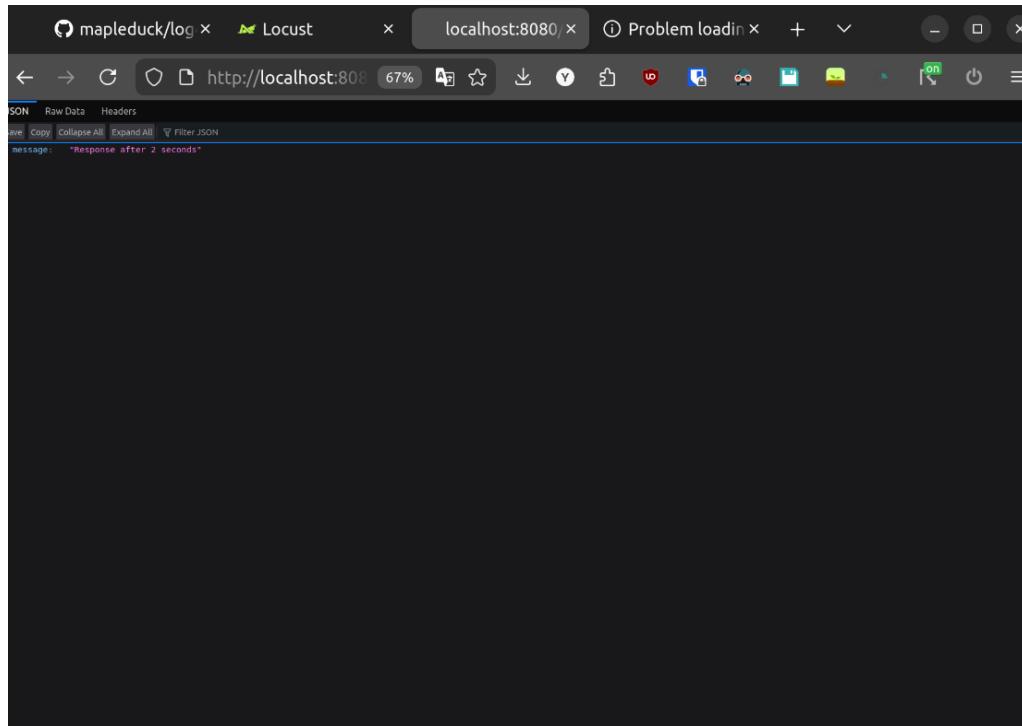
💡 Question 5 : À partir de combien de requêtes par minute observez-vous les erreurs 503 ? Justifiez avec des captures d'écran de Locust.

À partir de 6 requêtes par seconde.

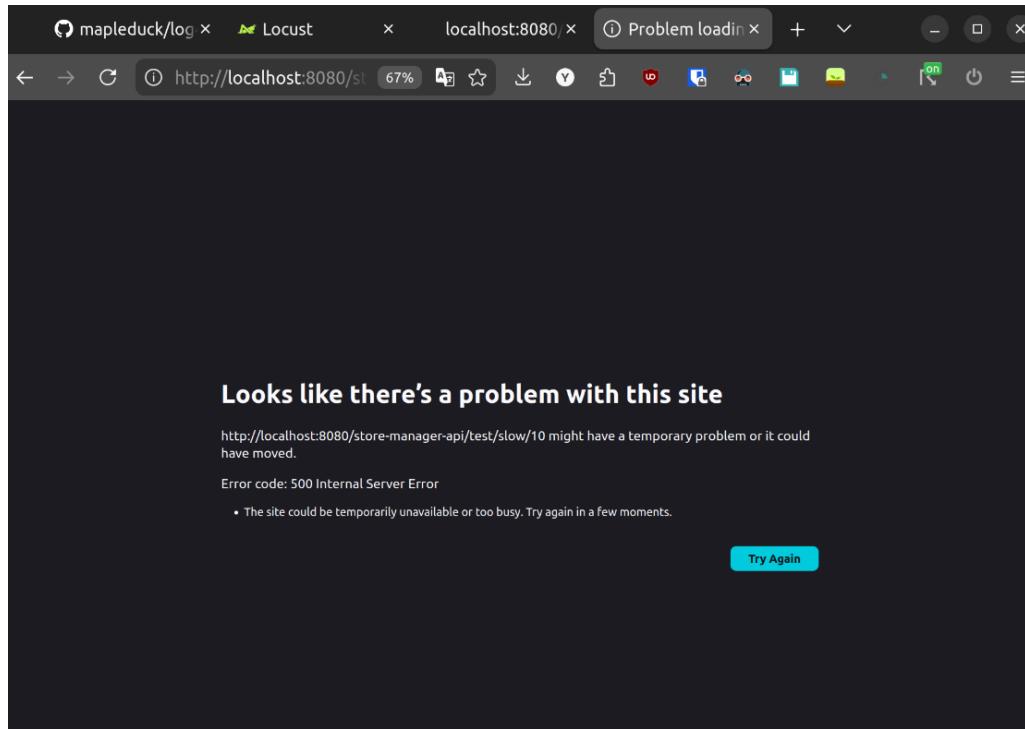


💡 Question 6 : Que se passe-t-il dans le navigateur quand vous faites une requête avec un délai supérieur au timeout configuré (5 secondes) ? Quelle est l'importance du timeout dans une architecture de microservices ? Justifiez votre réponse avec des exemples pratiques.

Voici la requête de secondes, elle fonctionne comme prévu et renvoie ceci après 2 secondes:



La requête de 10 secondes n'a jamais complétée. Le navigateur indique une erreur 500 après 5 secondes exactement.



Le timeout agit comme une protection qui empêche un service lent de causer un effet snowball, ou le service monopolise les ressources, causant d'autres services de aussi ralentir. En forçant un échec rapide, on protège la stabilité et la disponibilité de tout le système et on évite que la panne se propage. Ceci est d'une grande importance dans l'architecture microservice.

## Test de charge (activité 7)

Les tests de charge ont été effectués sur 120s avec 150 users peak et un spawn rate de 2 users par seconde. Les tests ont été effectués sur ma machine dû à un problème de connexion au réseau de l'école, que je réglerai d'ici le prochain labo.

Tentative #1 avec les paramètres par défaut, en faisant un POST sur les orders (voir locustfile ligne 37).

Très rapidement, un taux énorme d'erreur (>90%) a été atteint, quasiment que des erreurs 503. En regardant la console docker de KrakenD, le problème est évident:

```
[GIN] 2026/02/26 - 19:01:53 | 503 | 20.707µs | 172.21.0.5 | POST
"/store-manager-api/orders"
Error #01: rate limit exceeded
[GIN] 2026/02/26 - 19:01:54 | 503 | 18.792µs | 172.21.0.5 | POST
"/store-manager-api/orders"
Error #01: rate limit exceeded
```

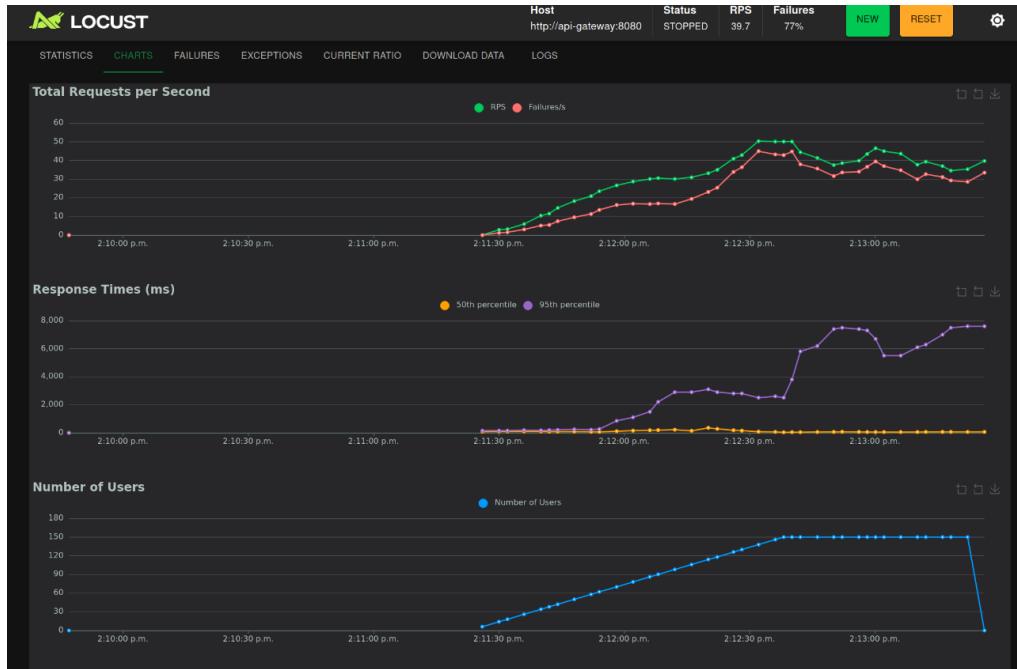
KrakenD avait un taux maximal de 200 requêtes par minutes, ce que notre test de charge oblitérait. J'ai donc modifié le taux maximal à une valeur plus libérale de 2000 par minute:

```
"max_rate": 2000,
```

De plus, les timeouts ont été mis à 15 secondes. Comme ça, on s'assure que le test de charge s'applique sur les failure points du backend et non simplement la config KrakenD.

## Tentative #2 avec les nouveaux paramètres KrakenD

Résultat: Taux d'erreur très stable de 50% pendant les premières 60 secondes, montant jusqu'à 77% vers la toute fin.



En regardant les taux d'erreurs, on voit que encore qu'une partie non négligeable (25%) des erreurs sont dues au rate limiter (503):

# Failures	Method	Name	Message
2513	POST	/store-manager-api/orders	CatchResponseError('Erreur 500: ')
654	POST	/store-manager-api/orders	CatchResponseError('Erreur 503: ')

C'est là que j'ai eu une réalisation. Lors du labo 2, il y avait dans le locustfile deux tests de GET et un test de POST, tous avec un weight de 1. Cela signifiait que pour chaque requête POST, il y avait en moyenne deux requêtes GET.

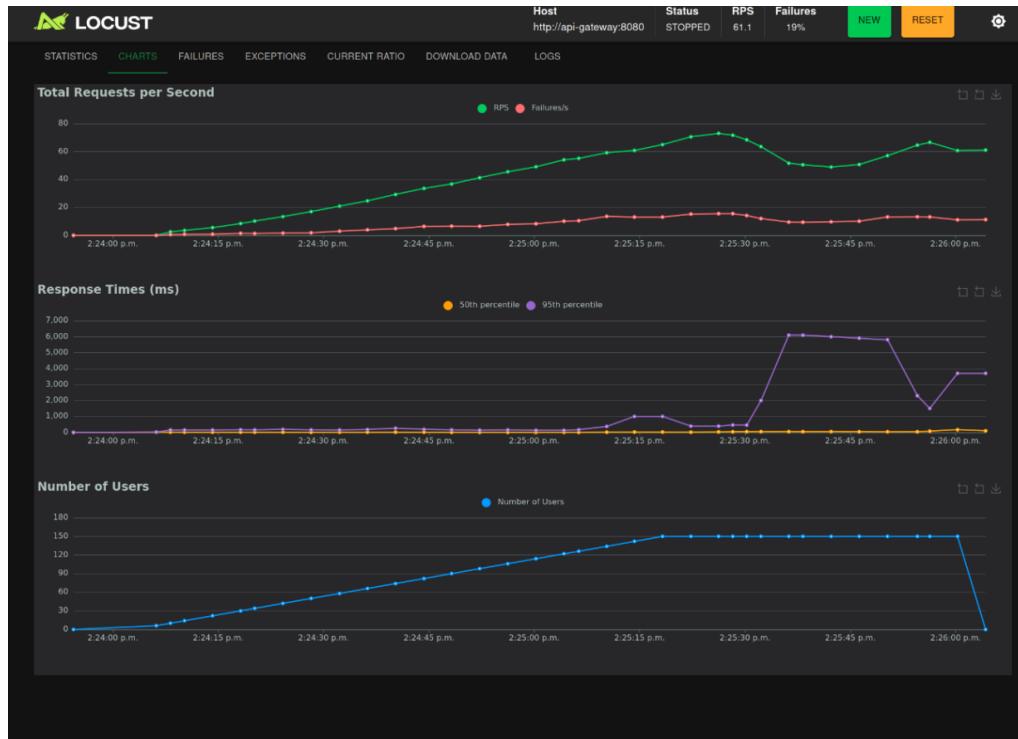
Or, mon locustfile actuel ne contient qu'une seule task, un POST. Cela veut dire que chacune des transactions effectuées par Locust est un POST. Les POST sont significativement plus lourds à handle, et cela explique le taux d'échec catastrophique comparé au labo 4, et le fait qu'il reste encore des rate limit errors.

Pour reproduire des conditions similaires au labo 4, j'ai donc rajouté un GET sur un order random entre 1 et 100 (pas le meilleur design, mais ça fera l'affaire, l'existence des orders 1 à 100 est garantie dans mon cas), et

j'ai donné à ce test un weight de 2, pour qu'il soit appelé deux fois plus souvent en moyenne que le POST.

### Tentative #3 avec le nouveau GET

Résultat: Le taux d'échec est descendu de façon significative, étant à 14-19% pour les requêtes overall et 40% pour les POST uniquement, ce qui ressemble beaucoup plus à mon labo 4:



Les erreurs 503 sont entièrement parties:

STATISTICS	CHARTS	FAILURES	EXCEPTIONS	CURRENT RATIO	DOWNLOAD DATA	LOGS
# Failures	Method	Name				Message
1065	POST	/store-manager-api/orders				CatchResponseError('Erreur 500: ')

Le haut taux d'erreurs sur les POST reste un problème, mais cela est une conséquence de rouler tout les services en plus du service de test sur la même machine. Des résultats plus positifs devraient avoir lieu lorsque la charge est balancée comme il faut.