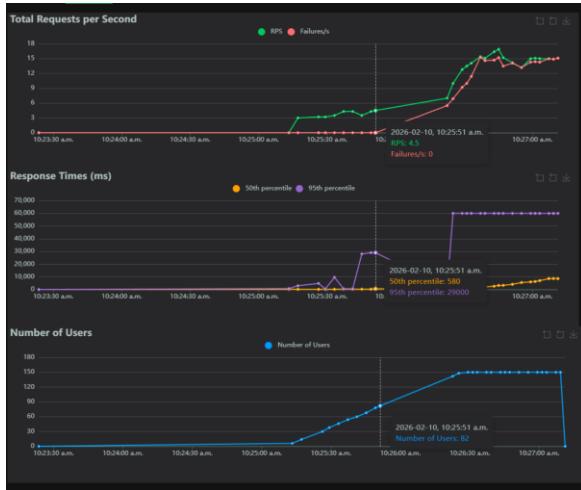


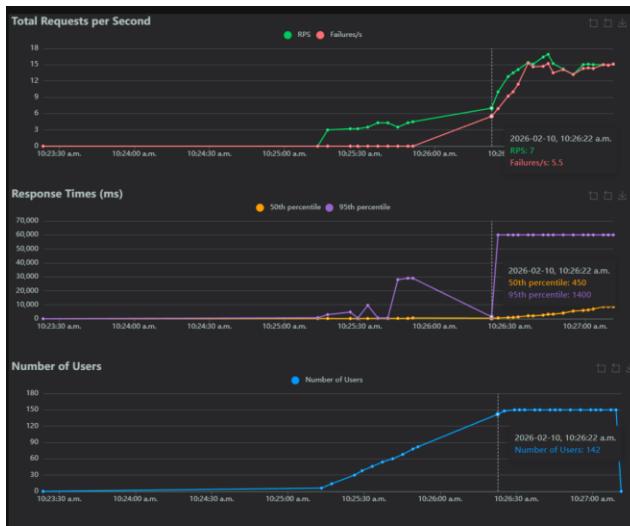
Ques1

Le Store Manager commence à échouer à environ 100-120 utilisateurs. On remarque qu'à 82 utilisateurs, il n'y avait pas encore d'échecs mais les temps de réponse étaient déjà très lents soit 29 secondes. Arriver à 142 utilisateurs, le taux d'échec était de 5.5 par seconde. Ça montre que le système n'est pas capable de gérer une charge élevée d'utilisateurs, peut-être à cause de la surcharge des connexions à la base de données MySQL ou un manque de ressources au niveau du serveur Flask

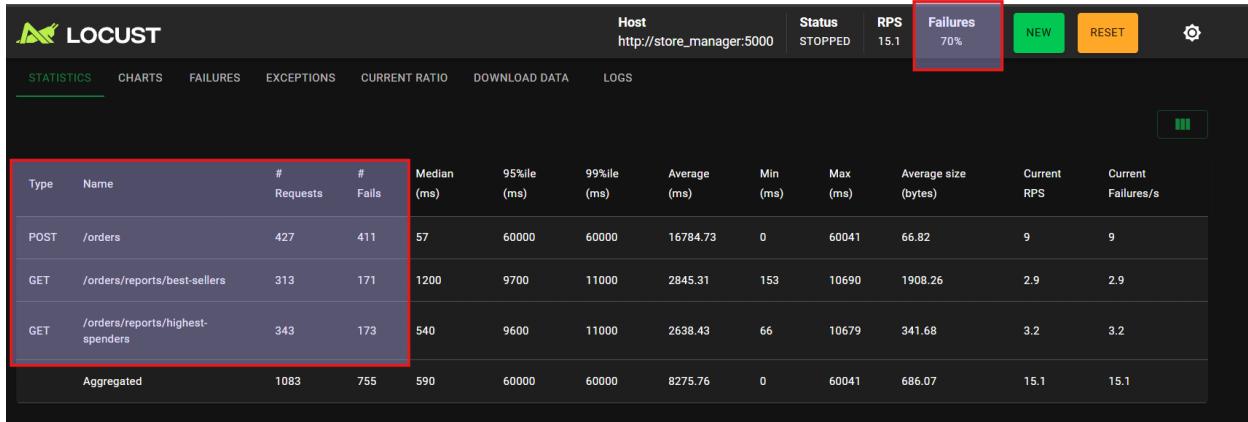
Cette image représente 82 utilisateurs avec un temps de réponse de 29-30 secondes.



Cette image représente 142 utilisateurs avec le taux d'échec de 5.5 par seconde.



Ques2



	Requêtes	Échecs	Taux d'échec
POST /orders	427	411	96%
GET /orders/reports/best-sellers	313	171	54%
GET /orders/reports/highest-spenders	343	173	54%

Les 3 endpoints testés échouent à 50% ou plus. L'endpoint POST /orders est le plus échoué avec un taux d'échec de 96% (411 échecs sur 427 requêtes). Ensuite, l'endpoint GET /orders/reports/best-sellers échoue à 54% (171 échecs sur 313 requêtes) et l'endpoint GET /orders/reports/highest-spenders échoue à 50% (173 échecs sur 343 requêtes). Sur le total on a 1083 requêtes dont 755 ont échoué. Ça représente un taux d'échec global de 70%. On remarque clairement que l'endpoint POST /orders échoue plus que les endpoints GET, ce qui nous laisse comprendre que les POST dans la base de données sont plus lourdes en ressources et surcharge plus rapidement le système.

Ques3

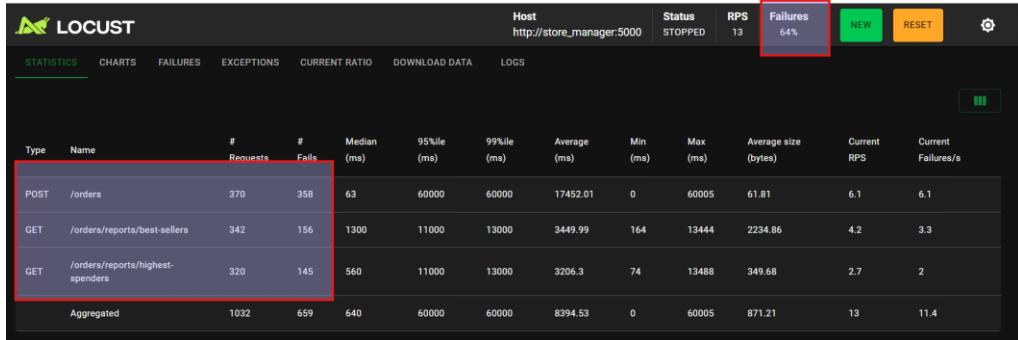
# Failures	Method	Name	Message
184	POST	/orders	CatchResponseError('Erreur : 500 - (mysql.connector.errors.OperationalError) 1040 (08004): Too many connections\n(Background on this error at: https://sqlalche.me/e/20/e3qB)')
112	POST	/orders	Exception('Aucune réponse (erreur ou timeout)')
112	POST	/orders	RetriesExceeded('http://store_manager:5000/orders', 0, original=timed out)
3	POST	/orders	CatchResponseError('Erreur : 500 - (mysql.connector.errors.DatabaseError) 1040 (HY000): Too many connections\n(Background on this error at: https://sqlalche.me/e/20/4xp6)')
171	GET	/orders/reports/best-sellers	CatchResponseError('Erreur : 500 - Aucun JSON dans la réponse. Message : <!DOCTYPE html><html lang="en"><title>500 Internal Server Error</title><h1>Internal Server Error</h1><p>The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.</p></v>')
173	GET	/orders/reports/highest-spenders	CatchResponseError('Erreur : 500 - Aucun JSON dans la réponse. Message : <!DOCTYPE html><html lang="en"><title>500 Internal Server Error</title><h1>Internal Server Error</h1><p>The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.</p></v>')

Les messages d'erreur affichés dans l'onglet Failures montre que le problème vient de MySQL. On le voit grâce aux types d'erreurs :

- **CatchResponseError('Erreur : 500 - (mysql.connector.errors.OperationalError) 1040 (08004): Too many connections (184 Failures)**) Cette erreur dit que MySQL a atteint sa limite maximale de connexions en même temps et refuse de nouvelles connexions.
- **CatchResponseError('Erreur : 500 - (mysql.connector.errors.DatabaseError) 1040 (HY000): Too many connections (3 Failures)**) : Même problème ici , trop de connexions.
- **Exception ('Aucune réponse (erreur ou timeout)')** (112 Failures): Le serveur ne répond plus car il est surchargé.
- **RetriesExceeded('http://store_manager:5000/orders', 0, original=timed out)** (112 Failures): Les requêtes ont dépassé le délai d'attente maximum.
- **CatchResponseError('Erreur : 500 Either the server is overloaded or there is an error in the application.)** (171 + 173 Failures) - Le serveur Flask retourne une erreur 500 parce qu'il ne peut plus traiter les requêtes.

Le problème principal vient alors de MySQL qui n'arrive pas à gérer le nombre de connexions en même temps. Quand MySQL refuse les connexions, le serveur Flask ne peut plus accéder à la base de données et retourne des erreurs 500. Et clairement, Redis n'est pas le problème dans ces erreurs vu que on avait désactivé la cache Redis plus haut dans le labo.

Ques4



# Failures	Method	Name	Message
147	POST	/orders	CatchResponseError(Erreur : 500 - (mysql.connector.errors.OperationalError) 1040 ([8004]): Too many connections\nBackground on this error at: https://sqlalche.me/e/20/e3q8j)
104	POST	/orders	Exception('Aucune réponse (erreur ou timeout)')
104	POST	/orders	RetriesExceeded('http://store_manager:5000/orders', 0, original=timed out)
3	POST	/orders	CatchResponseError(Erreur : 500 - (mysql.connector.errors.DatabaseError) 1040 ([HY000]): Too many connections\nBackground on this error at: https://sqlalche.me/e/20/4qpdj)
156	GET	/orders/reports/best-sellers	CatchResponseError(Erreur : 500 - Aucun JSON dans la réponse. Message : <!DOCTYPE html><html lang="en"><title>500 Internal Server Error</title><h1>Internal Server Error</h1><p>The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.</p>
145	GET	/orders/reports/highest-spenders	CatchResponseError(Erreur : 500 - Aucun JSON dans la réponse. Message : <!DOCTYPE html><html lang="en"><title>500 Internal Server Error</title><h1>Internal Server Error</h1><p>The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.</p>

	Requêtes	Échecs	Taux d'échec	Difference avec avant
Taux total	-	-	64%	-6%
POST /orders	370	358	96%	Identique
GET /orders/reports/best-sellers	342	156	54%	-9%
GET /orders/reports/highest-spenders	320	145	54%	-5%

En comparant les résultats avec le test d'avant, on remarque une petite amélioration des performances totaux. Le taux d'échec total est passé de 70% à 64%. Mais, pour l'endpoint POST /orders , le comportement est identique, le taux d'échec est encore très élevé 96% et le temps de réponse moyen est comparable au précédent. L'optimisation a eu un mauvais impact sur l'endpoint parce que le problème c'est encore la surcharge des connexions MySQL "Too many connections", pas le nombre de requêtes par commande. On voit une meilleure amélioration sur les endpoints de lecture Get avec leur taux d'échec réduit de quelque pourcent.

Ques5

Si on avait plus d'articles dans notre base de données par exemple 1 million ou plus par commande, le temps de réponse de l'endpoint POST /orders resterait stable avec le code optimisé, parce qu'on fait une seule requête SQL peu importe le nombre d'articles. Avec l'ancien code non-optimisé le temps de réponse augmentait en lien avec le nombre d'articles vu que chaque article nécessitait une requête séparée à la base de données.

```
25     try:
26         start_time = time.time()
27         # TODO: optimiser
28         product_prices = {}
29         products = session.query(Product).filter(Product.id.in_(product_ids)).all()
30
31         for product in products:
32             product_prices[product.id] = product.price
33
34         total_amount = 0
35         order_items = []
36
37         for item in items:
38             pid = item["product_id"]
39             qty = item["quantity"]
40
```

Ques6

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/orders	790	772	9328.02	0	60003	104	6.58	6.43
GET	/orders/reports/best-sellers	687	0	123.7	2	461	386	5.72	0
GET	/orders/reports/highest-spenders	706	0	110.36	3	462	418	5.88	0
Aggregated		2183	772	3450.31	0	60003	294.3	18.19	6.43

	Requêtes	Échecs	Taux d'échec	Difference avec avant
Taux total	-	-	35%	-29%
POST /orders	790	772	96%	Identique
GET /orders/reports/best-sellers	687	0	0%	-54%
GET /orders/reports/highest-spenders	706	0	0%	-54%

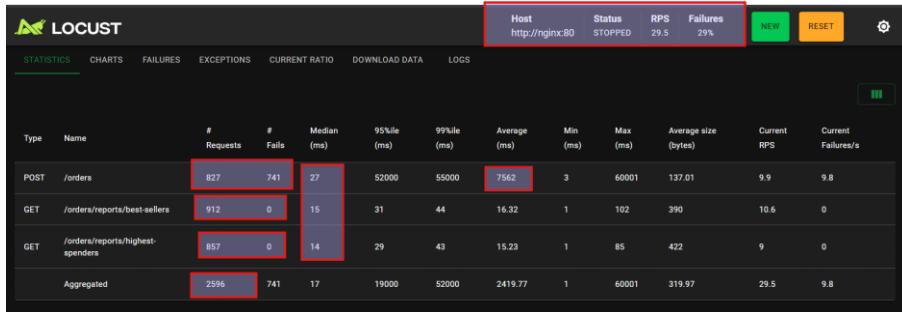
Les résultats montrent une énorme amélioration grâce au cache Redis. Le taux d'échec total est passé de 64% à 35%, donc une amélioration de 29%. Les endpoints de lecture GET /orders/reports/best-sellers et GET /orders/reports/highest-spenders ont maintenant 0% d'échec alors que c'était 54% avant. Leur temps de réponse moyen est passé d'environ 2700 ms à 80 ms, ce qui est une amélioration d'environ 95%. Le nombre de requêtes par seconde est passé de 9 à 51. Par contre, l'endpoint POST /orders reste problématique avec 98% d'échec 772 échecs sur 790 requêtes.

Ques7

Failure Summary				
# Failures	Method	Name	Message	Timestamp
115	POST	/orders	Exception('Aucune réponse (erreur ou timeout)')	2023-09-18T10:45:00Z
115	POST	/orders	RetriesExceeded('http://store_manager:5000/orders', 0, original=timed out)	2023-09-18T10:45:00Z
533	POST	/orders	CatchResponseError('Erreur : 500 - (mysql.connector.errors.OperationalError) 1040 (08004): Too many connections\n(Background on this error at: https://sqlalche.me/e/20/e3q8)')	2023-09-18T10:45:00Z
9	POST	/orders	CatchResponseError('Erreur : 500 - (mysql.connector.errors.DatabaseError) 1040 (HY000): Too many connections\n(Background on this error at: https://sqlalche.me/e/20/4xp6)')	2023-09-18T10:45:00Z

L'endpoint POST /orders reste très limité par MySQL. Même si les rapports utilisent maintenant Redis, chaque création de commande doit toujours écrire dans MySQL « insertion de la commande et des articles et mise à jour des stocks ». Encore une fois, l'erreur "Too many connections" de MySQL reste le problème primaire. Le cache Redis à régler le problème de lecture mais pas le problème d'écriture POST, parce que c'est obligatoire pour les écritures de passer par la base de données relationnelle pour que les données restent correctes.

Ques8



Avec l'équilibrage de charge Nginx, on remarque une amélioration des performances. Le taux d'échec total est passé de 35% à 29%, donc une diminution de 6 %. Les endpoints de lecture sont beaucoup plus rapides GET /orders/reports/best-sellers est passé de 124 ms à 15 ms et GET /orders/reports/highest-spenders est passé de 110 ms à 14 ms. L'endpoint POST /orders s'est aussi amélioré avec son temps de réponse passant de 9328 ms à 7562 ms et un taux d'échec passant de 98% à 90%. On voit aussi que le nombre total de requêtes a augmenté de 2183 à 2596. Toutes ces améliorations c'est grâce au fait que Nginx partage les requêtes entre 2 instances du Store Manager, ce qui permet de mieux balancer la charge.

Ques9

```
5   http {
6     upstream store_manager_nginx {
7       least_conn;
8       server store_manager:5000;
9     }
10 }
```

Dans le fichier nginx.conf, la politique d'équilibrage de charge utilisée **est least_conn** (least connections). Cette politique envoie les nouvelles requêtes vers le serveur qui a le moins de connexions actives. C'est une bonne stratégie quand les requêtes ont des temps de traitement variables, parce que ça évite de surcharger un serveur qui traite des requêtes trop longues.