Test de performance

Projet Pedagogique

4IW - ESGI

Groupe:
Andre SBROCCO FIGUEIREDO
Jonathan RIVO
Belkacem MEZHOUD
Mohammed Amine ZIANI

Application développée

Nous avons développé une application web qui permet de gérer les ventes d'une boutique en ligne. Nous avons utilisé le framework Symfony pour coder le back-end, et React pour le front-end. La base de données que nous avons choisie pour ce projet contient 15 colonnes et 21642 lignes.

Id	Region	Country	Item type	Sales channel	Order priority	Order id	Ship date	Units sold	Unit price	Unit cost	Total revenue	Total cost	Total profit	Order date
1	Sub-Saharan Africa	South Africa	Fruits	Offline	M	443368995	7/28/2012	1593	9.33	6.92	14862.69	11023.56	3839.13	7/27/2012
3	Australia and Oceania	Papua New Guinea	Meat	Offline	M	940995585	6/4/2015	360	421.89	364.69	151880.4	131288.4	20592	5/15/2015
4	Sub-Saharan Africa	Djibouti	Clothes	Offline	H	880811536	7/2/2017	562	109.28	35.84	61415.36	20142.08	41273.28	5/17/2017
5	Europe	Slovakia	Beverages	Offline	L	174590194	12/4/2016	3973	47.45	31.79	188518.85	126301.67	62217.18	10/26/2016
6	Asia	Sri Lanka	Fruits	Online	L	830192887	12/18/2011	1379	9.33	6.92	12866.07	9542.68	3323.39	11/7/2011
8	Sub-Saharan Africa	Tanzania	Beverages	Online	L	659878194	1/16/2017	1476	47.45	31.79	70036.2	46922.04	23114.16	11/30/2016
9	Sub-Saharan Africa	Ghana	Office Supplies	Online	L	601245963	4/15/2017	896	651.21	524.96	583484.16	470364.16	113120	3/23/2017
10	Sub-Saharan Africa	Tanzania	Cosmetics	Offline	L	739008080	5/24/2016	7768	437.2	263.33	3396169.6	2045547.44	1350622.16	5/23/2016
11	Asia	Taiwan	Fruits	Offline	M	732588374	2/23/2014	8034	9.33	6.92	74957.22	55595.28	19361.94	2/9/2014
12	Middle East and North Africa	Algeria	Cosmetics	Online	M	761723172	2/24/2011	9669	437.2	263.33	4227286.8	2546137.77	1681149.03	2/18/2011

Back-end:

[Dépôt github]

API endpoints:

sales

Pour récupérer la liste complète des ventes (un très grand volume de données) il suffit d'envoyer une requête de type **GET** au endpoint suivant: **/sales**

sales-light

Le endpoint **/sales-light** répond à une requête **GET** et permet de récupérer uniquement les 10 dernières ventes, cette requête est beaucoup plus légère par rapport à la requête précédente.

heavy-operation

Le endpoint **/heavy-operation** permet de simuler une requête qui consomme beaucoup de ressources dans le serveur. (**GET**)

delete-sale

Permet au client de supprimer un élément. Cette requête est de type **POST**

Exemple de requête

```
//deletes sale with id=2
{
```

```
    "id": 2
}
```

add-sale

Permet de créer une vente. Le serveur attend un nombre aléatoire de secondes avant d'envoyer une réponse.
Cette requête est de type **POST**.

```
// POST request
{
    "region": "Europe",
    "country": "France",
    "item_type": "Clothes",
    "sales_channel": "Offline",
    "order_priority": "M",
    "order_id": 999562594,
    "ship_date": "7/28/2020",
    "units_sold": "1593",
    "unit_price": 5.5,
    "unit_cost": 4,
    "total_revenue": 2001.78,
    "total_cost": 11023.56,
    "total_profit": 3856.48,
    "order_date": "7/27/2012",

}
```

login

Afin de s'authentifier, le client doit envoyer une requête POST au endpoint suivant **/login**

```
// login POST request
{
    "username": "your_username",
    "password": "some-password",
  }
```

Front-end:

Dépôt github

La “Slow Page”

Le front-end comporte deux pages affichant les données. La page “Data” qui affiche les données de manière optimisée grâce à un mapping sur les données, et une page “Slow page” volontairement lente. Voici comment elle fonctionne :

Comme pour la page Data, nous avons un state de type Array qui contiendra les données, une fois le useeffect initial de la page déclenché.

```
useEffect(() => {
  window.scrollTo(0, 0);
  getUser() ? null : navigate("/login");
  fetchData();
}, []);
```

```
const fetchData = async () => {
  setLoading(true);
  let records = await fetch(SERVER_ADDRESS + "/sales", {
    method: "GET",
  });
  setData(await records.json());
  setLoading(false);
};
```

Une fois les données arrivées sont disponibles dans le State, au lieu de les afficher avec un mapping sur le State “data” comme nous le faisons dans la page “Data”, nous avons ajouté des étapes supplémentaires afin de ralentir le code.

Les données que nous affichons dans la page proviennent d’un State différent nommé “displayedData”, de type Array lui aussi. Nous réalisons un mapping pour afficher les données sur la page.

```
<tbody>
  {displayedData.map((i) => (
    <tr className="hover:bg-gray-100 cursor-pointer border-b-2 border-gray-300 text-c
      <td>{i.id}</td>
      <td>{i.region}</td>
      <td>{i.country}</td>
      <td>{i.itemType}</td>
      <td>{i.salesChannel}</td>
      <td>{i.orderPriority}</td>
      <td>{i.orderId}</td>
      <td>{i.shipDate}</td>
      <td>{i.unitsSold}</td>
      <td>{i.unitPrice}</td>
      <td>{i.unitCost}</td>
      <td>{i.totalRevenue}</td>
      <td>{i.totalCost}</td>
```

Cependant, pour ralentir le code, nous réalisons un transfert progressif entre le state “data” et le state “displayedData” :

```
const displayData = () => {
  for (let i = 0; i < data.length; i++) {
    let newData = displayedData;
    newData.push(data[i]);
    console.log("displayed data",newData)
    setdisplayedData(newData);
  }
};
```

```
useEffect(() => {
  displayData();
}, [data]);
```

Nous pouvons voir que nous passons chaque item de “data” un par un à “displayedData”, de ce fait, le state est sans cesse renouvelé jusqu’à atteindre son état final, au lieu d’être mis à jour en une seule fois. Ce procédé ralentit grandement la page qui met un certain temps à s’afficher. Nous aurions pu éventuellement rajouter un useEffect sur le state “displayedData” afin qu’à chaque boucle de la fonction “displayData”, cela déclenche le useEffect, mais cela aurait probablement crash, d’autant plus que la page était suffisamment lente.

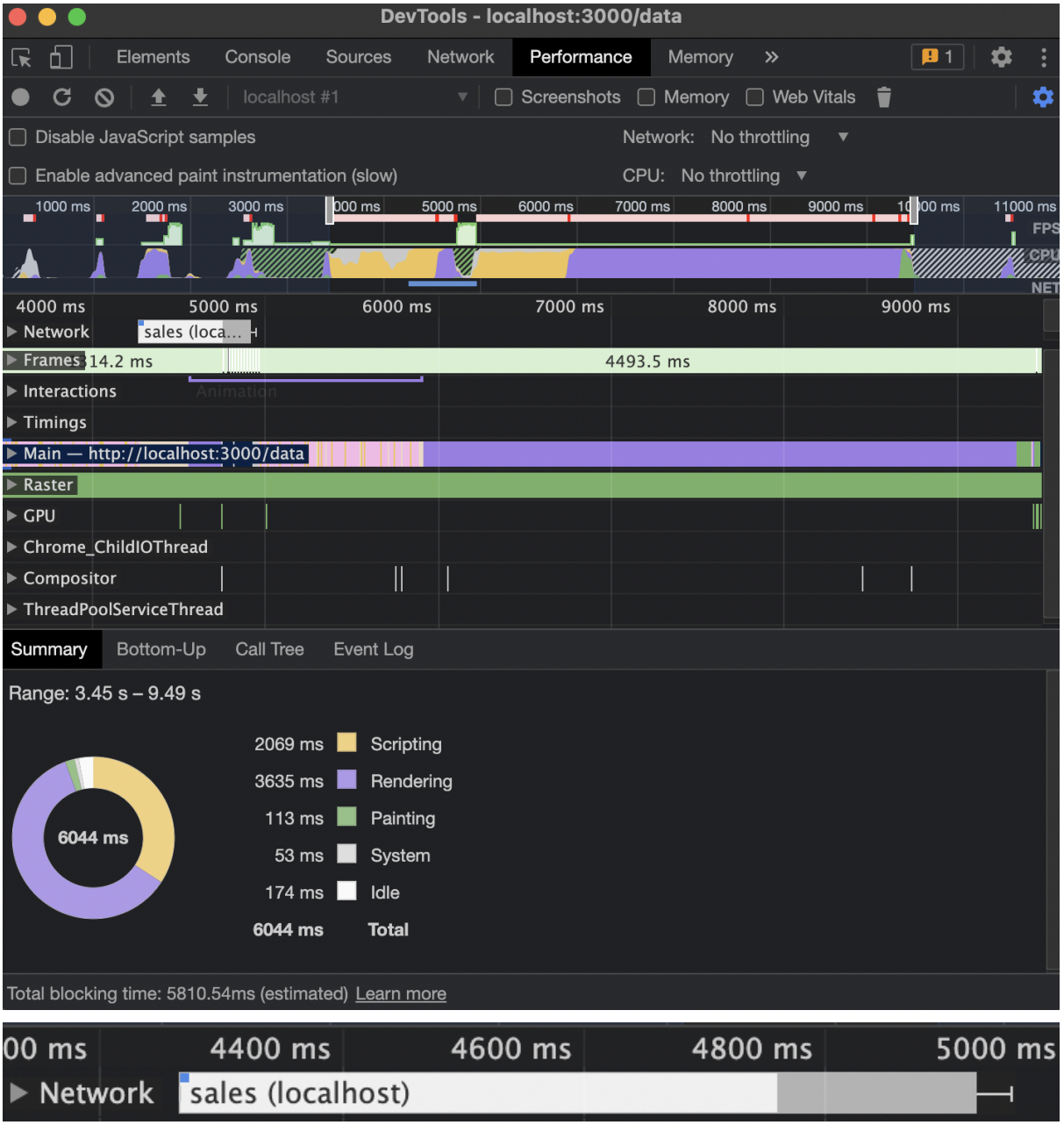
Notation Big-O de l’algorithme : O(n)

La vitesse d'exécution évolue linéairement par rapport au nombre d’enregistrement de la base (boucle for basée sur la taille du tableau de données).

Analyse de performance de la slow-page

En utilisant chrome-developer tools, il est possible de trouver la cause des ralentissements d'une page. Pour faire la comparaison des résultats, nous avons fait l'analyse sur la page normale (optimisée), et sur la page ralentie.

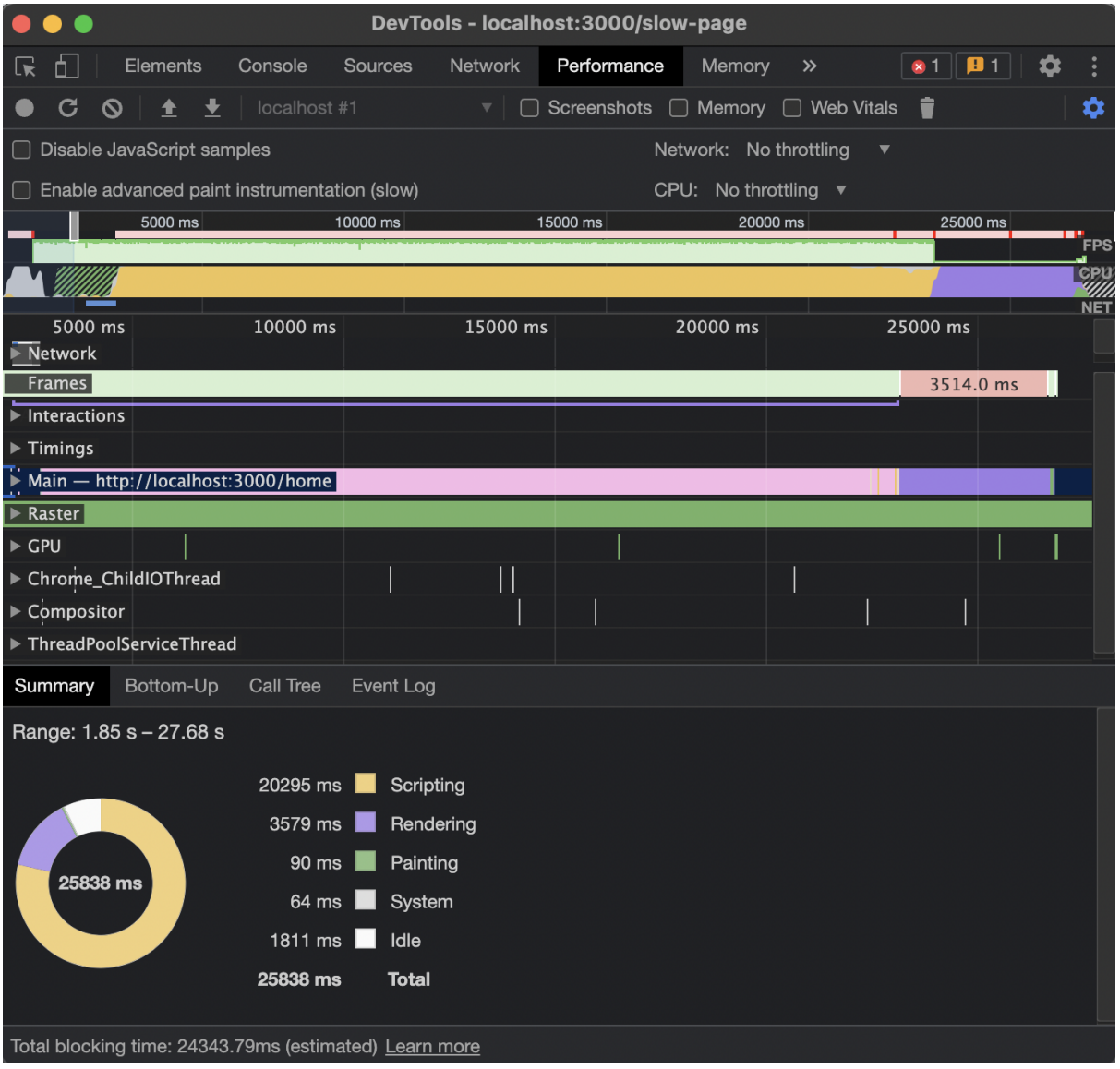
Chrome-developer tools - Normal page



Dans la page optimisée, nous observons que la requête pour récupérer le fichier prend environ 800 ms, scripting prend 2000 ms, et la majeure partie du temps est prise par le rendering (3635 ms).

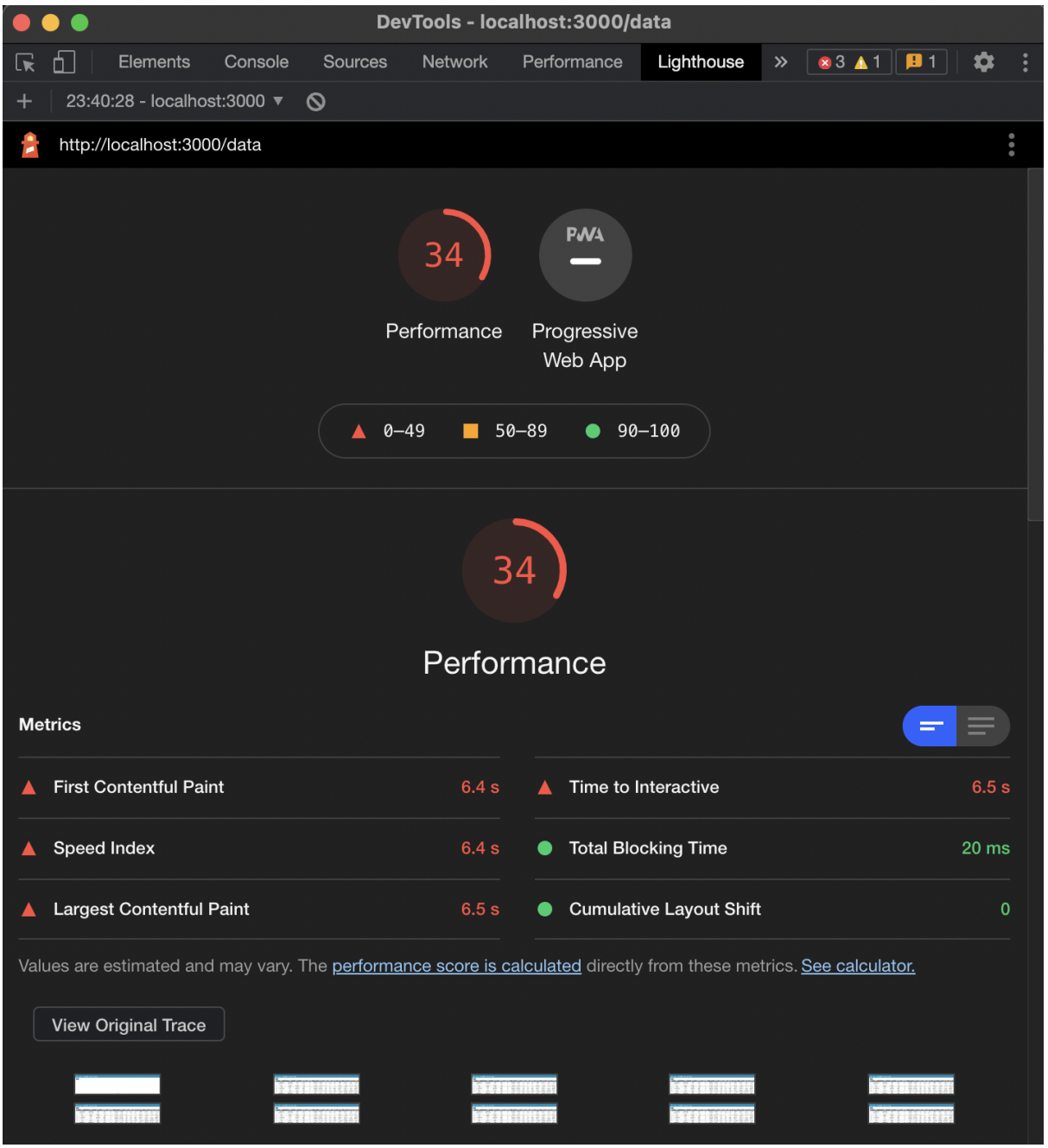
Chrome-developer tools - Slow page:

En faisant la même analyse sur la slow-page, on observe que le temps de network est toujours le même, mais le temps pris par scripting a augmenté considérablement (20.295 ms), ce qui nous donne une astuce de l'origine du ralentissement.



Lighthouse - Normal page

D'autres informations supplémentaires peuvent être trouvées avec l'outil "lighthouse", qui nous indique aussi où se trouve la source du problemme.


DevTools - localhost:3000/data
Elements
Console
Sources
Network
Performance
Lighthouse
23:40:28 - localhost:3000
http://localhost:3000/data
34
Performance
PWA
Progressive Web App
0–49
50–89
90–100
34
Performance
Metrics
First Contentful Paint 6.4 s
Speed Index 6.4 s
Largest Contentful Paint 6.5 s
Time to Interactive 6.5 s
Total Blocking Time 20 ms
Cumulative Layout Shift 0
Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.
View Original Trace

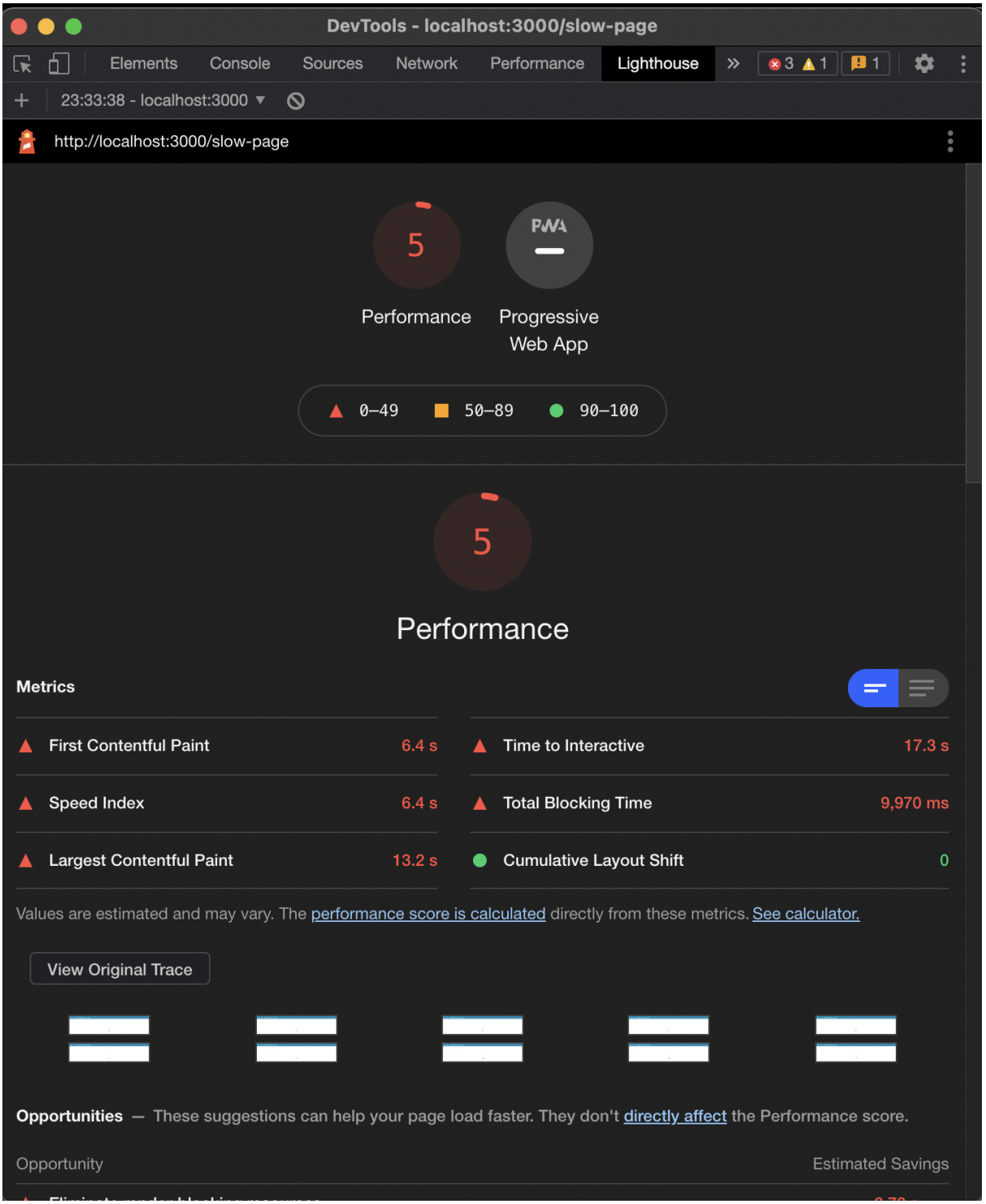
Opportunities — These suggestions can help your page load faster. They don't directly affect the Performance score.

Opportunity	Estimated Savings
▲ Eliminate render-blocking resources	2.73 s
▲ Remove unused CSS	2.72 s
▲ Remove unused JavaScript	1.04 s

Diagnostics — More information about the performance of your application. These numbers don't directly affect the Performance score.

- ▲ Serve static assets with an efficient cache policy — 2 resources found
- ▲ Avoid enormous network payloads — Total size was 7,537 KiB
- ● Avoid chaining critical requests — 2 chains found
- ● User Timing marks and measures — 478 user timings
- ● Keep request counts low and transfer sizes small — 13 requests • 7,537 KiB
- ● Largest Contentful Paint element — 1 element found
- ● Avoid long main-thread tasks — 2 long tasks found

Lighthouse - Slow page



Opportunities — These suggestions can help your page load faster. They don't directly affect the Performance score.

Opportunity	Estimated Savings
▲ Eliminate render-blocking resources	2.73 s
▲ Remove unused CSS	2.72 s
▲ Remove unused JavaScript	1.08 s

Diagnostics — More information about the performance of your application. These numbers don't directly affect the Performance score.

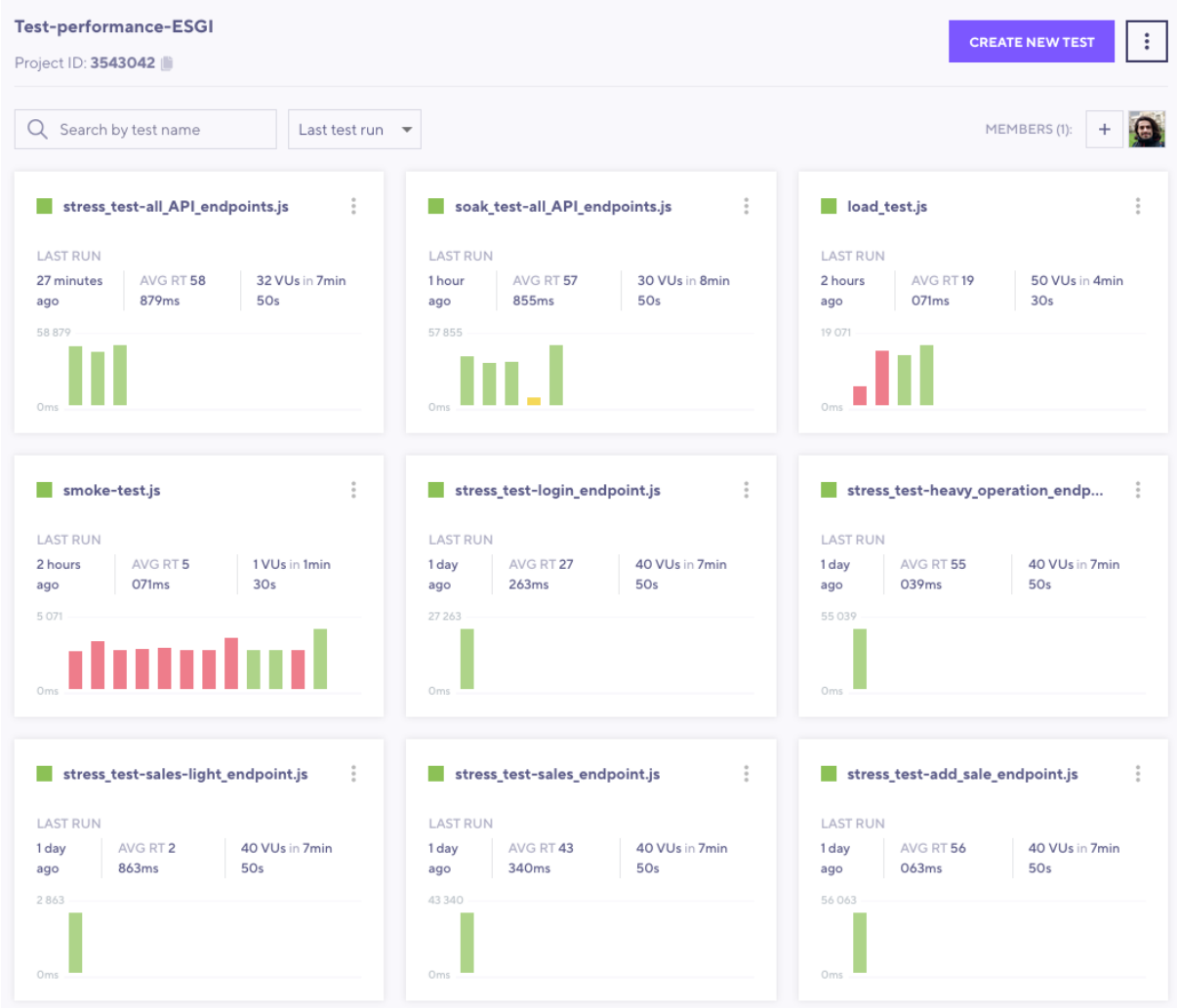
- ▲ Avoid enormous network payloads — Total size was 13,693 KiB
- ▲ Serve static assets with an efficient cache policy — 2 resources found
- ▲ Avoid an excessive DOM size — 310,415 elements
- ▲ Minimize main-thread work — 28.5 s
- ▲ Reduce JavaScript execution time — 24.5 s
- ● Avoid chaining critical requests — 2 chains found
- ● User Timing marks and measures — 172 user timings
- ● Keep request counts low and transfer sizes small — 13 requests • 13,693 KiB
- ● Largest Contentful Paint element — 1 element found
- ● Avoid long main-thread tasks — 4 long tasks found

Analyse de performance de l’API avec K6

[Dépôt github]

Tests réalisés

- Smoke test
- Load test
- Soak test
- Stress test:
    - All API endpoints
    - /add-sale endpoint
    - /delete-sale endpoint
    - /sales endpoint
    - /sales-light endpoint
    - /heavy-operation endpoint



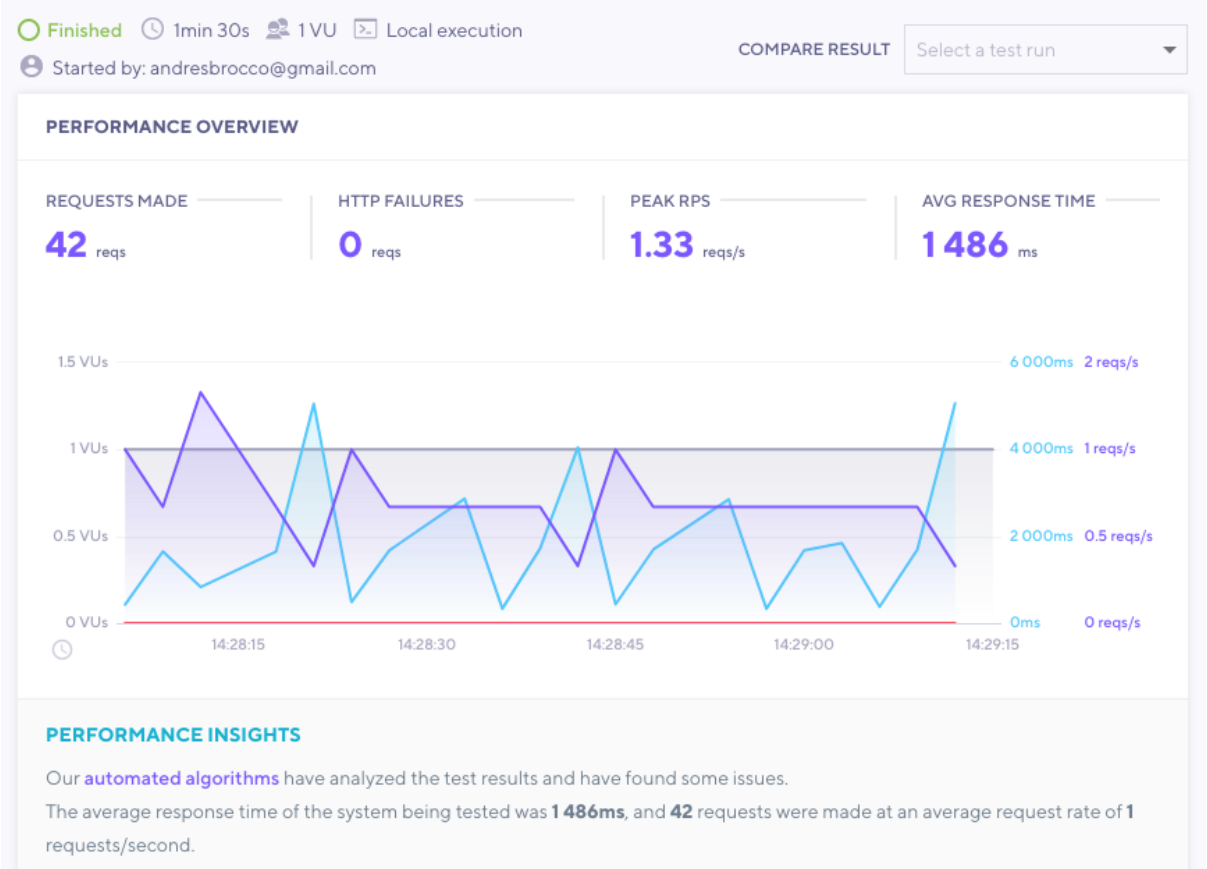
Procédure de réalisation de tests

Smoke test

Le but du smoke test était de valider que chaque requête à l'API fonctionne, et quantifier son temps moyen de réponse. Le résultat obtenu était:

API Endpoint	p(95)
/sales-light	~ 100 ms
/delete-sale	~ 100 ms
/sales	~ 1000 ms
/login	~ 1000 ms
/heavy-operation	~ 3500 ms
/add-sale	~ 5000 ms

Ces valeurs de temps moyen ont été utilisées comme "threshold" pour tous les tests suivants, parce qu'ils représentent le fonctionnement normal de l'API, sans aucune charge.



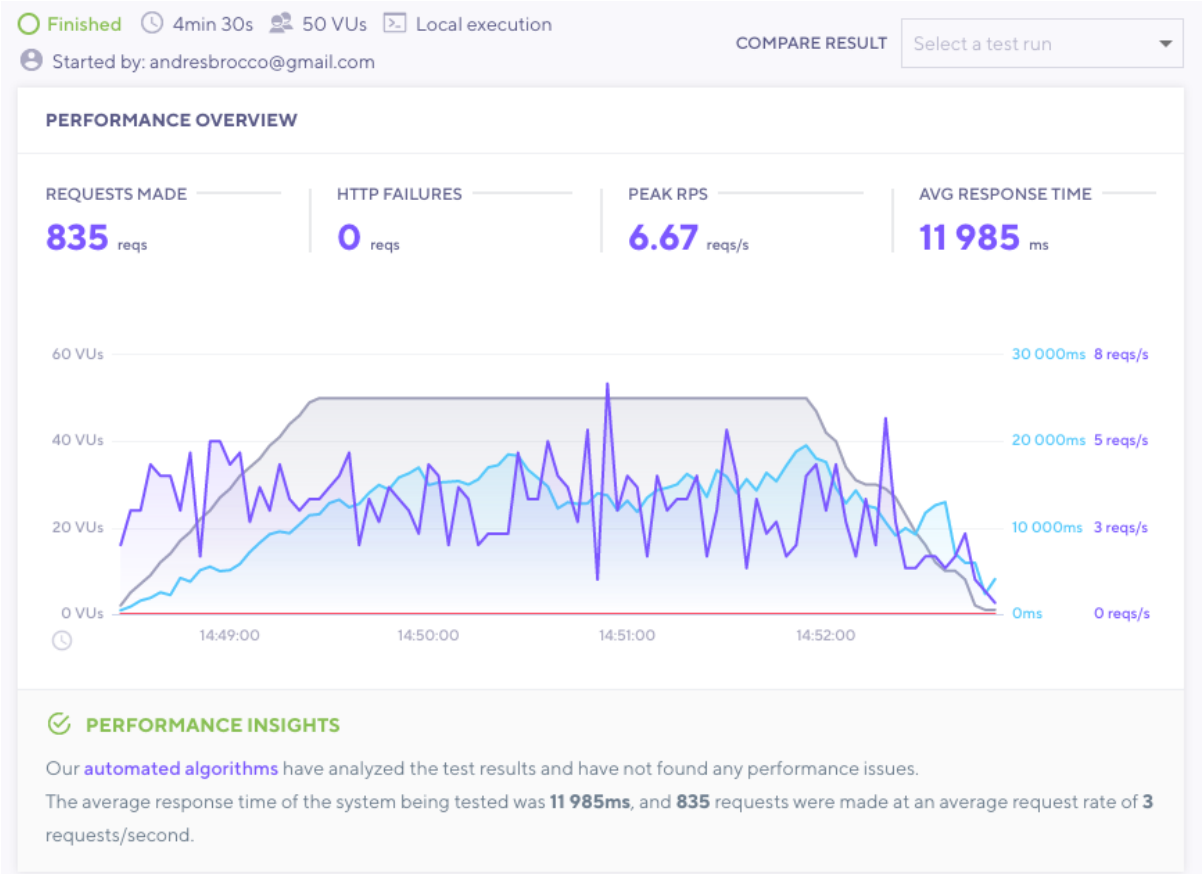
THRESHOLDS (6/6) | CHECKS (42/42) | HTTP (42/42) | ANALYSIS Compare metrics | SCRIPT View executed script | LOGS Execution logs

VIEW AS

	URL	METH...	STATUS	COUNT	MIN	AVG	STDDEV	P95	P99	MAX
✓	http://0.0.0.0:8082/sales	GET	200	7	603ms	654ms	61ms	749ms	760ms	762ms
✓	http://0.0.0.0:8082/login	POST	200	7	627ms	644ms	20ms	676ms	683ms	685ms
✓	0.0.0.0:8082/heavy-operation	GET	200	7	3 258ms	3 318ms	46ms	3 380ms	3 387ms	3 389ms
✓	0.0.0.0:8082/delete-sale	POST	200	7	44ms	51ms	11ms	69ms	75ms	76ms
✓	0.0.0.0:8082/sales-light	GET	200	7	32ms	38ms	3.65ms	42ms	42ms	42ms
✓	http://0.0.0.0:8082/add-sale	POST	200	7	2 053ms	4 208ms	1 219ms	5 072ms	5 072ms	5 072ms

Load Test

Le but du Load Test était d'évaluer le nombre de VUs supportés par le système. Après 4 itérations, nous avons observé que l'API a bien répondu jusqu'à 50 VUs (la limite maximum permise dans le compte gratuit de k6 cloud).



THRESHOLDS (6/6) | CHECKS (835/835) | HTTP (835/835) | ANALYSIS Compare metrics | SCRIPT View executed script | LOGS Execution

VIEW AS

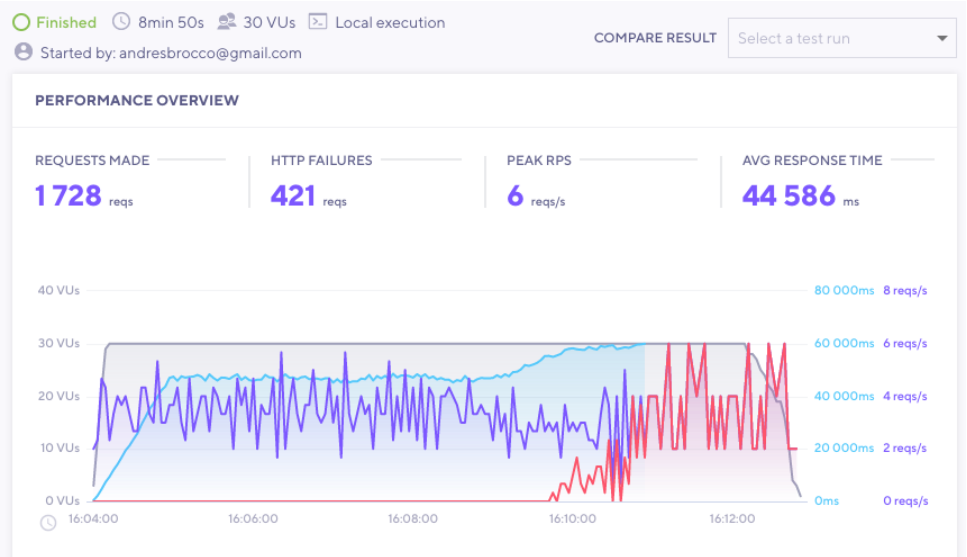
URL	MET...	STA...	CO...	MIN	AVG	STD...	P95	P99	MAX
.../sales-light	GET	200	147	30ms	10 784ms	4 639ms	15 916ms	18 119ms	19 391ms
0.0.0.0:8082/sales	GET	200	147	641ms	11 016ms	4 699ms	16 543ms	18 531ms	19 484ms
.../heavy-operation	GET	200	141	3 582ms	15 181ms	4 691ms	20 351ms	21 913ms	22 798ms
0.0.0.0:8082/add-sale	POST	200	122	1 619ms	13 619ms	4 890ms	20 479ms	22 110ms	22 780ms
.../delete-sale	POST	200	131	43ms	11 046ms	4 645ms	16 831ms	18 149ms	19 120ms
0.0.0.0:8082/login	POST	200	147	628ms	10 568ms	5 023ms	17 369ms	19 081ms	19 413ms

Mais on peut déjà déduire à partir des gros temps de réponses moyens (~ 10.000 ms) que le serveur recevait probablement de nouvelles requêtes avant de finir la précédente, ce qui a résulté en un temps majeur pour compléter le cycle de chaque requête.

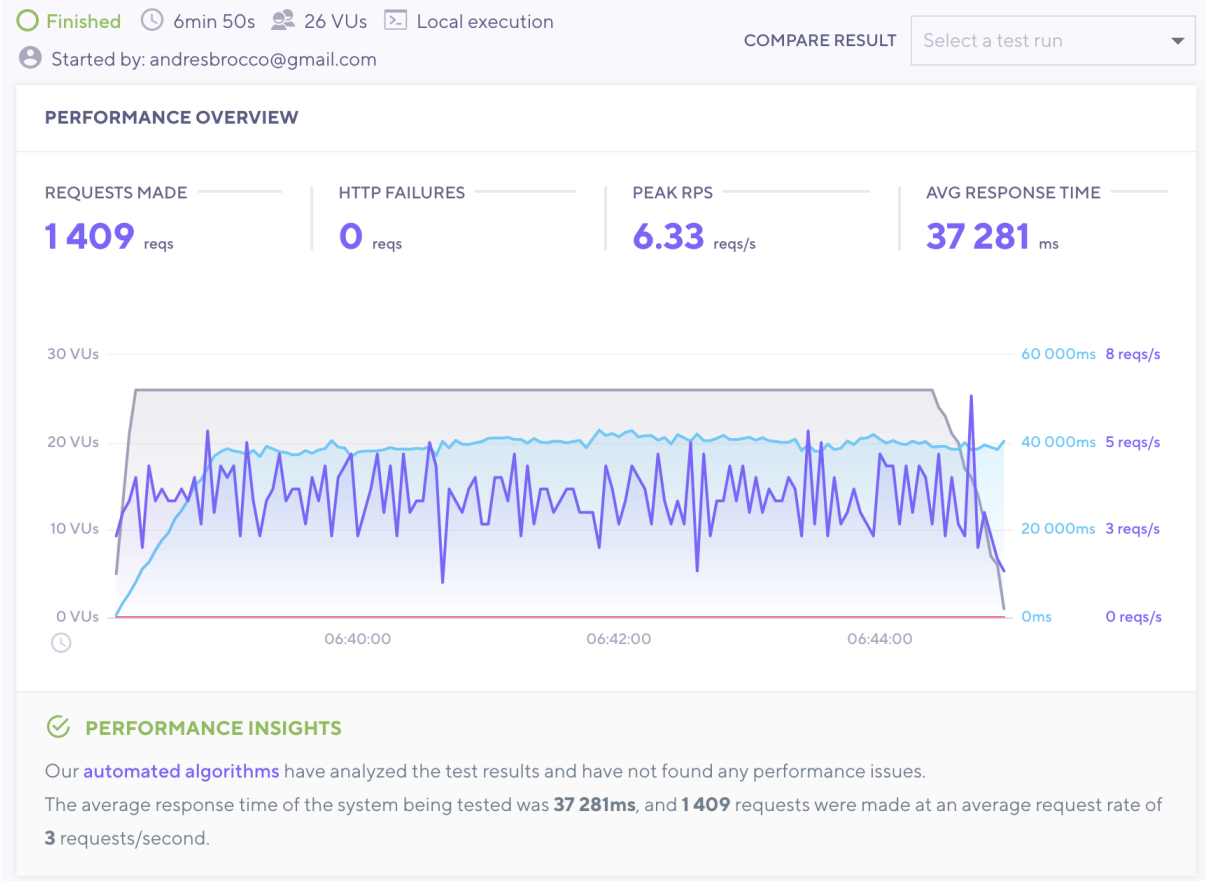
Une possibilité pour résoudre ce problème serait d'implémenter une sorte de “network manager” qui irait limiter le nombre de requêtes concurrentes (et répondrait avec une erreur 503 - Service Unavailable dans le cas où le serveur est surchargé).

Soak test

Après le point de départ de 50 VUs maximum, il faudrait tester la robustesse de l’API à gérer un nombre constant d’utilisateurs continuellement. C’est ici que nous avons commencé les Soak Test. Après 5 itérations, nous avons trouvé qu’il supportait 30 VUs pendant seulement 5 min. Après la 6ème minute, le serveur a présenté beaucoup de perte de requêtes:



Par contre, comme montré ci-dessous, le test a bien passé avec 26 VUs, ce qui défini que le nombre limite d'utilisateurs simultanés de la plateforme.

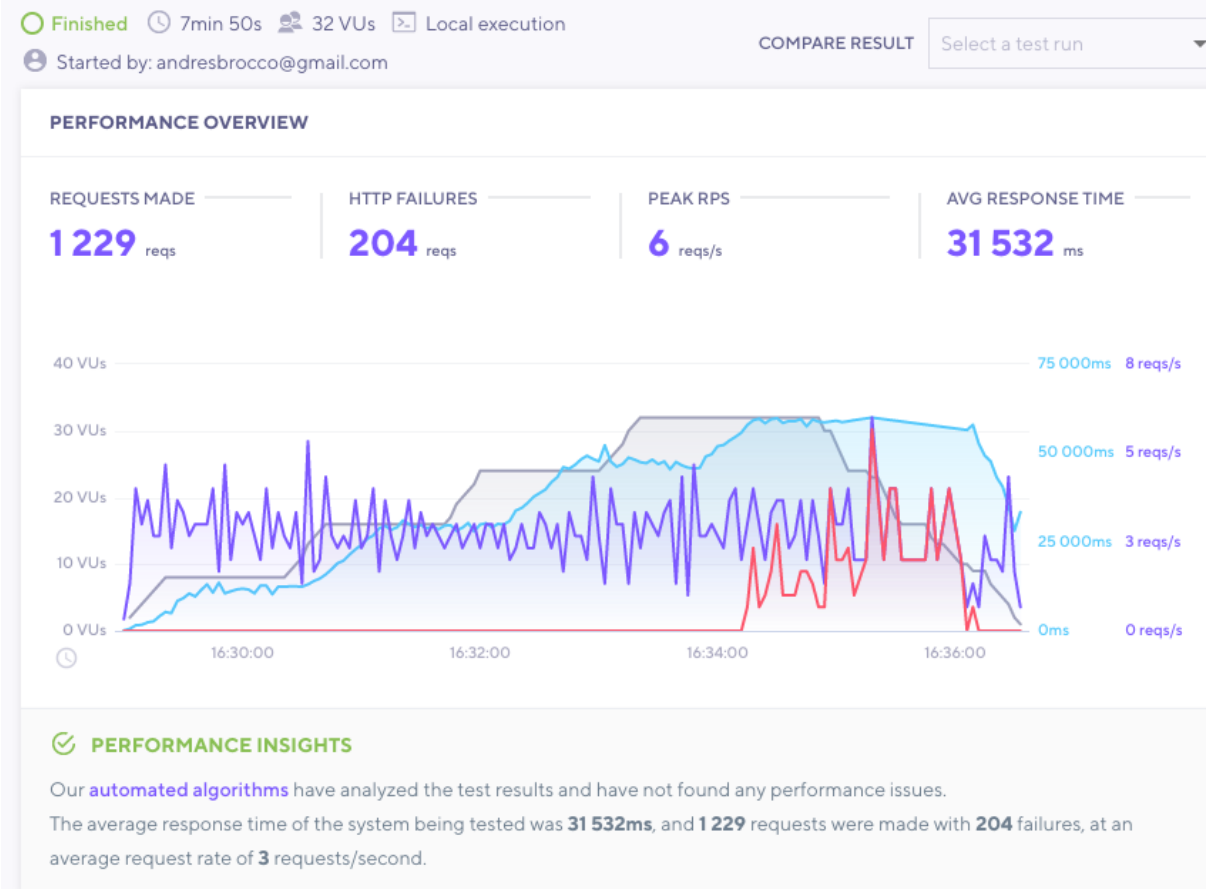


URL	METH...	STAT...	COU...	MIN	AVG	STDD...	P95	P99	MAX
Sales-light responses	GET	200	236	63ms	35 985ms	7 513ms	40 703ms	41 759ms	41 915ms
Login responses	POST	200	236	678ms	36 622ms	7 566ms	41 471ms	42 239ms	42 714ms
Heavy-operation responses	GET	200	233	4 061ms	39 871ms	7 715ms	44 543ms	45 311ms	45 533ms
Sales responses	GET	200	235	760ms	36 672ms	7 548ms	41 471ms	42 152ms	42 759ms
Add-sale responses	POST	200	233	2 655ms	38 494ms	7 588ms	44 543ms	45 577ms	46 484ms
Delete a sale responses	POST	200	236	54ms	36 090ms	7 512ms	40 703ms	41 727ms	41 957ms

Nous pouvons faire la même observation par rapport au temps de réponse moyen (monté ici jusqu'à 35.000ms)

Stress Test - All API Endpoints

Après avoir estimé la limite du nombre d'utilisateur simultanés (~ 26), est arrivé le moment d'évaluer comment le serveur se comporte dans une situation de stress temporaire: le test surpasse la limite d'utilisateurs simultanés jusqu'à 32 VUs pour 2 min.



	URL	METH...	STATUS	COUNT	MIN	AVG	STDD...	P95	P99	MAX
✕	Login responses	POST	0	30	60 001ms	60 006ms	0ms	60 006ms	60 006ms	60 006ms
✕	Delete a sale responses	POST	0	25	60 001ms	60 006ms	0ms	60 006ms	60 006ms	60 006ms
✕	Heavy-operation responses	GET	0	50	60 001ms	60 006ms	0ms	60 006ms	60 006ms	60 006ms
✕	Sales responses	GET	0	35	60 001ms	60 006ms	0ms	60 006ms	60 006ms	60 006ms
✕	Add-sale responses	POST	0	39	60 001ms	60 006ms	0ms	60 006ms	60 006ms	60 006ms
✕	Sales-light responses	GET	0	25	60 001ms	60 006ms	0ms	60 006ms	60 006ms	60 006ms
✓	Add-sale responses	POST	200	166	151ms	31 413ms	17 022ms	58 111ms	59 737ms	59 929ms
✓	Sales-light responses	GET	200	180	37ms	31 541ms	18 261ms	59 135ms	59 701ms	59 962ms
✓	Login responses	POST	200	175	765ms	31 631ms	17 867ms	59 391ms	59 928ms	60 000ms
✓	Heavy-operation responses	GET	200	154	3 481ms	32 255ms	16 244ms	54 105ms	58 096ms	59 753ms
✓	Sales responses	GET	200	170	707ms	30 828ms	17 470ms	58 623ms	59 903ms	59 973ms
✓	Delete a sale responses	POST	200	180	46ms	31 580ms	18 250ms	58 879ms	59 913ms	59 948ms

On peut observer qu'après 1 min en 32 VUs, l'API a commencé à perdre quelques requêtes, et même après avoir enlevé 24 VUs, l'API n'avait pas encore rattrapé la demande de requête.

La bonne nouvelle c'est que même après avoir surchargé le serveur, l'API a réussi à remonter sa performance, observée par les bonnes réponses obtenues à la fin du cycle.

Prochaines étapes de tests

Pour raffiner l'analyse de performance de l'API, le groupe a proposé de lancer des stress tests pour chaque endpoint. Cette situation exceptionnelle pourrait arriver par exemple dans le cas d'un "Black friday", où l'endpoint /add-sale et /login seront bien plus chargés que les autres. Malheureusement, nous n'avons pas trouvé le temps de le faire.

Analyse de performance de l'API avec Blackfire

Malheureusement l'équipe a essayé de mettre en place la sonde Blackfire, mais nous avons eu des soucis avec docker. Nous avons suivi la doc - mis en place la sonde (blackfire agent) dans le conteneur docker, créé un compte blackfire.io, enregistré nos credentials dans le .env - mais quand on essaye de lancer l'extension dans google chrome ou même par cli, on reçois toujour l'erreur suivant:

Pipeline sur Github

Malheureusement nous ne sommes pas très à l'aise avec docker pour faire un github actions workflow avec docker-compose.
Pour k6: https://github.com/marketplace/actions/k6-load-test
Pour pagespeed-insights: https://github.com/marketplace/actions/page-speed-insights