DALHOUSIE UNIVERSITY

CSCI 5709
Advance Topics in Web Development
Assignment 3

Submitted By: Vishvesh Patel
Banner ID: B01021112

Github Link:- https://github.com/Viishveesh/CSCI5709/tree/main/Assignment3

Table of Contents

Focused feature

I have focused on the main component of my web application which is the authentication feature. I have created a several apis for the backend and frontned for serving the signup, login, request reset password and reset password screens. I have designed it using the ReactJS for the frontend and Flask for the backend. As in our group project, I took care of building the authentication system, I thought to apply the same here.

So, In the following report, there will be following apis :-
Backend APIs :-

1. /api/signup – for signup
2. /api/login – for login
3. /api/request-reset – for requesting for changing the password
4. /appi/reset-password – for reseting the password

Frontend APIS :-

1. / – Entry point of the application
2. /static/js/bundle.js – JS Bundle

Light Load Backend APIs :-

As mentioned in the assignment, I have installed the jmeter and create the one scenario for the light load. As mentioned in the figure 1, I configured it with the 10 users (threads) with 30 second of ramping up period.

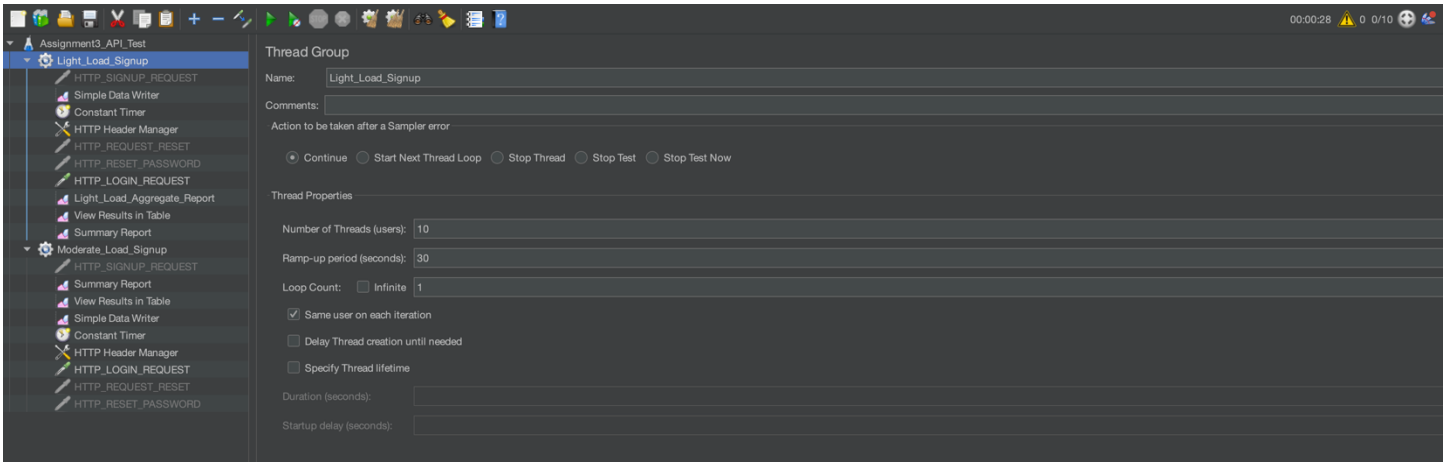


Fig 1: Light load for backend apis with 10 users with 30 second of ramping up time

Constant Timer (Think Time) :-

As mentioned in the document that the think time should be the 500ms. I have added the Constant Timer for that in which there is a Thread Delay of 500ms as mentioned in the figure 2.

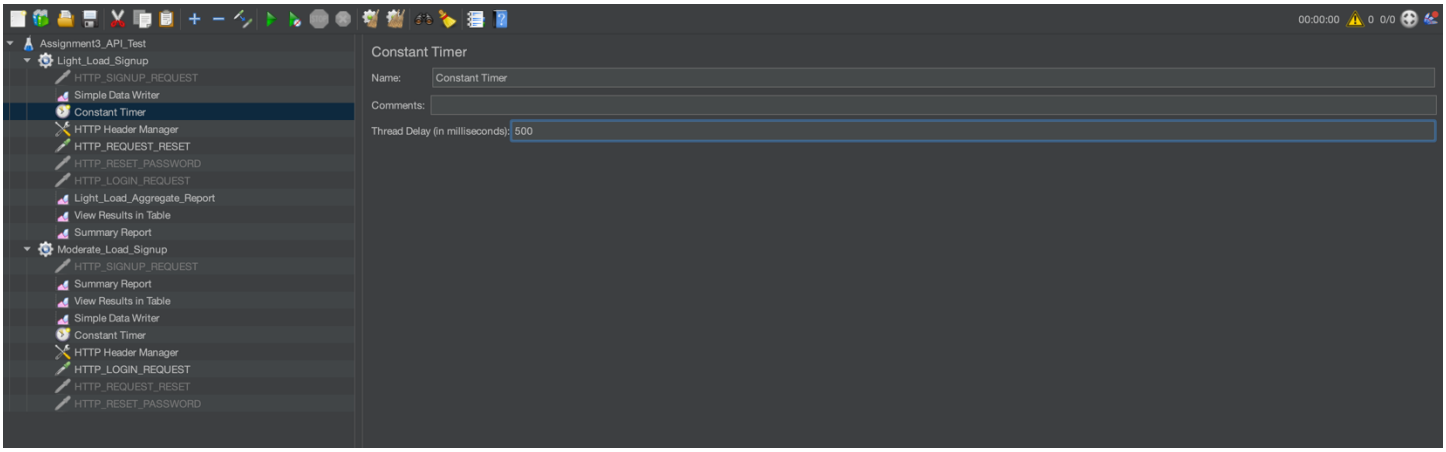


Fig 2. Constant Timer for the light load

Signup HTTP Request :-

Then under the same thread group, I have created the HTTP request for the signup page where I mentioned the server name, port address, http request method and the body data as showing in the figure 3.

Example body data :-

```
{
        "firstName": "Test${__Random(1,1000)}",
        "lastName": "User",
        "email": "test${__Random(1,1000)}@gmail.com",
        "password": "Test@2002",
        "role": "doctor"
}
```

This will generate the random email id for the signup and if there is an email id already registered and with the same email id if we are going to signup again then it will throw an error that the email already exists.

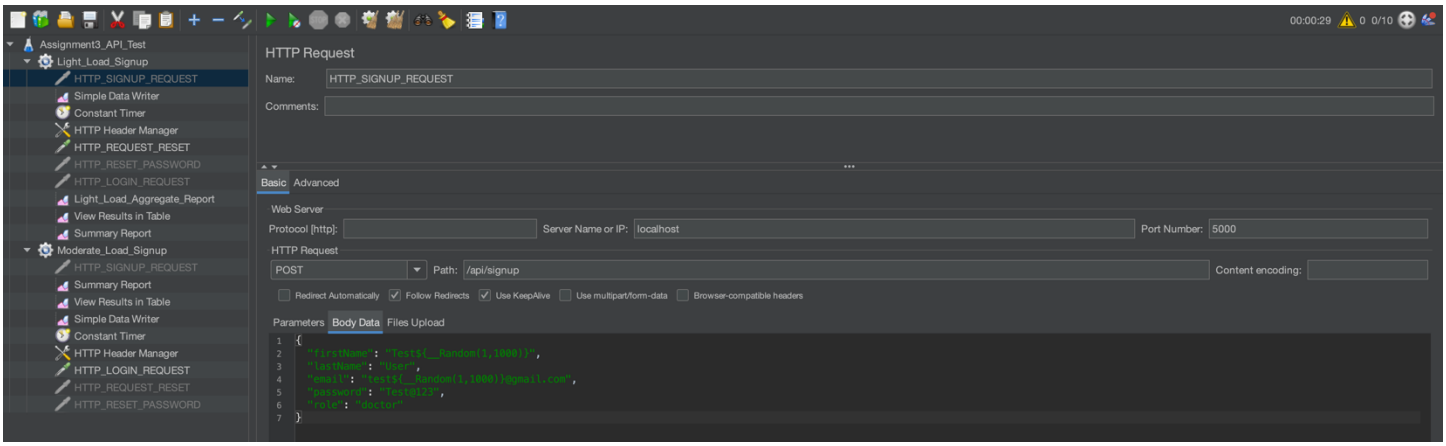


Fig 3: Signup HTTP Request

Login HTTP Request:-

I have also created the HTTP request for the login page where I mentioned the localhost as a server name, port address which in the case is 5000, http request method is POST, endpoint /api/login and body data as showing in the Figure 4.

Example of body data:-

```
{
        “email”: “rimiy67437@ikanteri.com”,
        “password”: “vishvesh2002@”
}
```

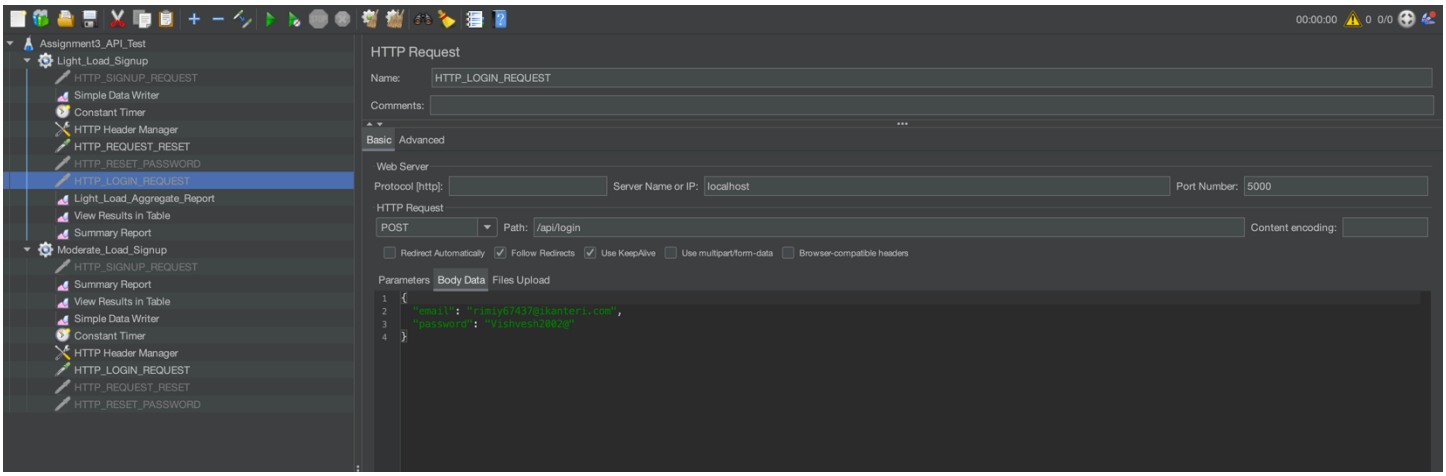


Fig 4: Login HTTP Request

Request Reset Password HTTP Request:-

I have created an endpoint /api/request-reset which asks for the email id of the user and if the email id exists then it first checks whether the user is registered with us or not if it’s registered it’s going to send the password reset link over their mail id. And if it’s not then it should give an error message to the user that the user is not exists.

I have the created the request for that endpoint for testing the performance of that. Configuration is described in the figure 5.

Example of body data:-

```
{
        “email”: “rimiy67437@ikanteri.com”
}
```

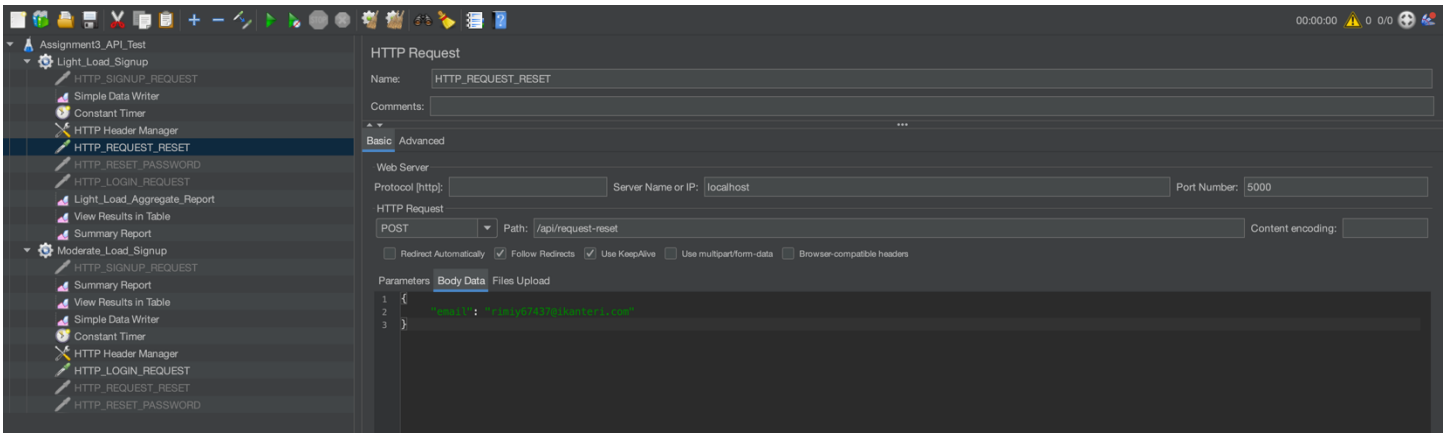


Fig 5: request-reset HTTP Request

Reset password HTTP Request:

Now, the user gets an email for resetting the password. There is one token in that URL which is being used to authorized an user to reset the password. In this request, my backend expects the token, new password and confirm new password values. For testing the performance I have created this HTTP request to test the /api/reset-password api.

Example of body data:-

```
{
        "token": "InBhdGVsdmlzaHZlc2g4NUBnbWFpbC5jb20i.aIJhsw.bG2gi1Bc_ubxEfNUN3LL1tE5rSM",
        "newPassword": "Hello2090@",
        "confirmPassword": " Hello2090@"
}
```

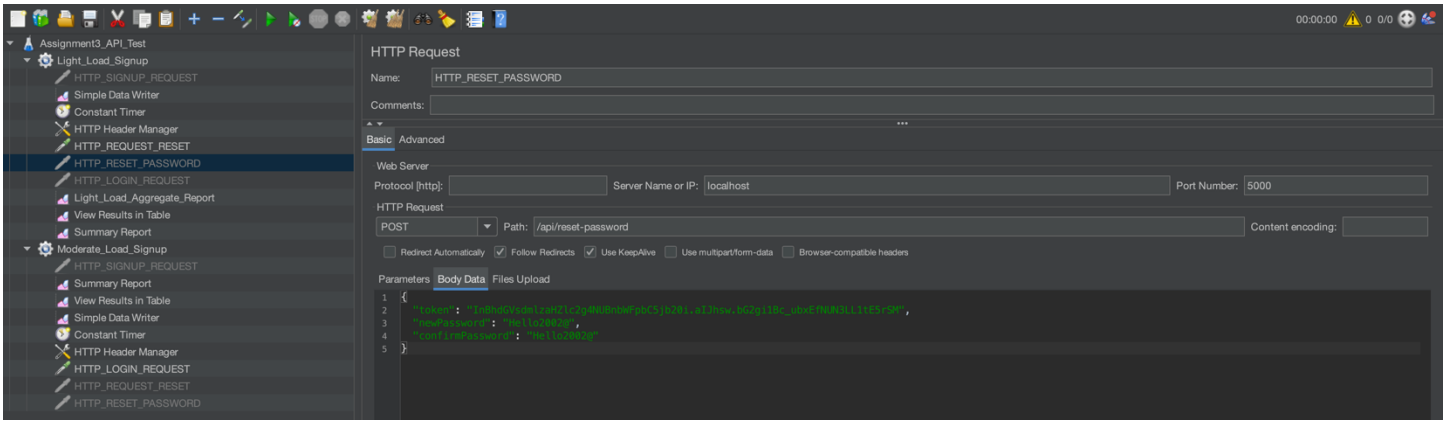


Fig 6: Reset password HTTP Request

CSV Result:-

Here I am including the screenshot of the report that's being generated using the Jmeter to capture the required things. This CSV report is for the light load thread group.

Fig 7: Aggregate report for the light load thread group

Moderate Load for backend APIs:-

As also we need to create the thread group for the moderate load, I have created that thread group in Jmeter with number of users as 50, ramping up period of 60 second as shown in figure 8 and Think time of 500ms as shown in figure 9.

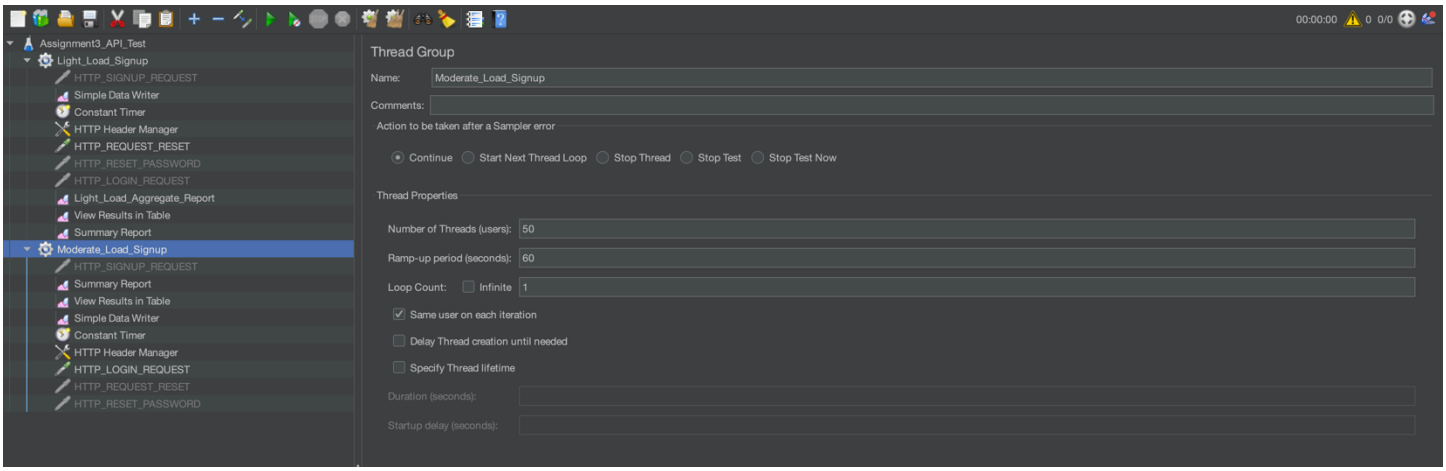


Fig 8: Moderate load thread group

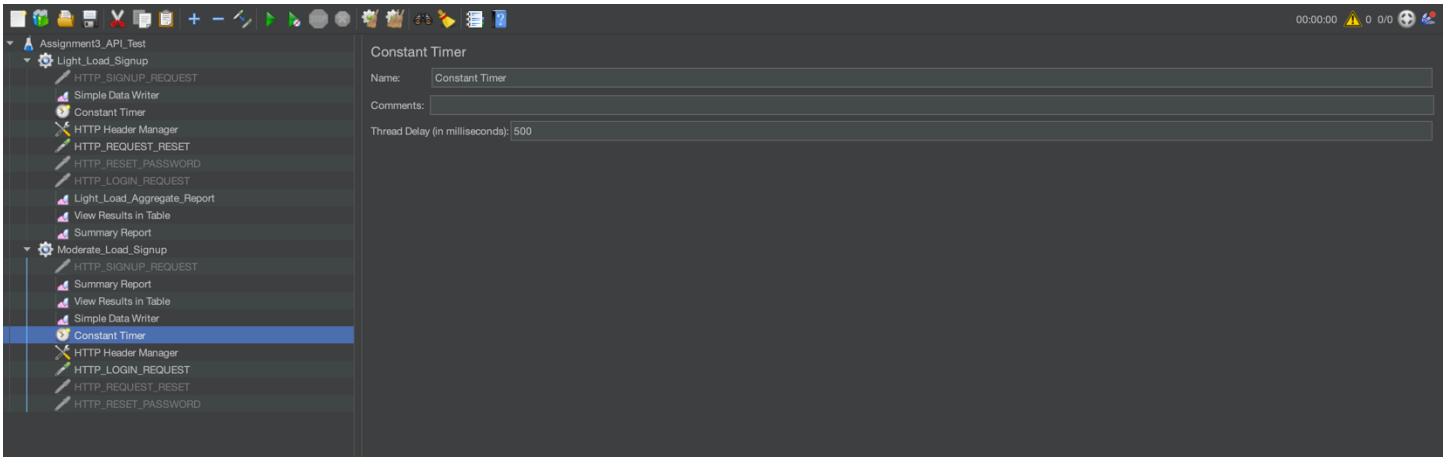


Fig 9: Think time of the moderate load thread group

Signup HTTP Request :-

Then under the same thread group, I have created the HTTP request for the signup page where I mentioned the server name, port address, http request method and the body data as showing in the figure 10.

Example body data :-

```
{
        "firstName": "Test${__Random(1,1000)}",
        "lastName": "User",
        "email": "test${__Random(1,1000)}@gmail.com",
        "password": "Test@2002",
        "role": "doctor"
}
```

This will generate the random email id for the signup and if there is an email id already registered and with the same email id if we are going to signup again then it will throw an error that the email already exists.

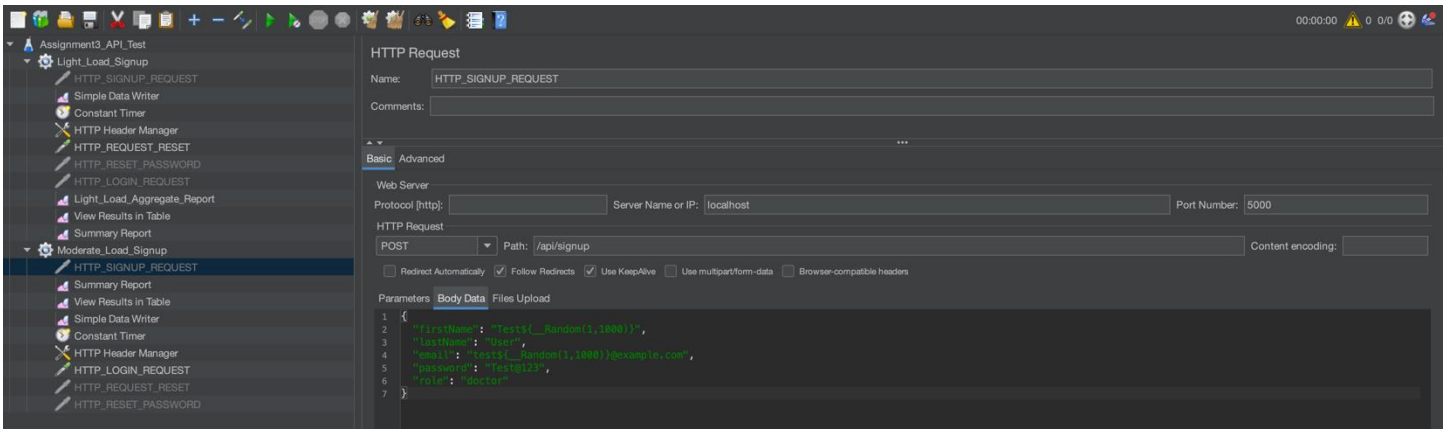


Fig 10: Signup HTTP Request for the moderate load group

Login HTTP Request:-

I have also created the HTTP request for the login page where I mentioned the localhost as a server name, port address which in the case is 5000, http request method is POST, endpoint /api/login and body data as showing in the Figure 11.

Example of body data:-

```
{
        “email”: “rimiy67437@ikanteri.com”,
        “password”: “vishvesh2002@”
}
```

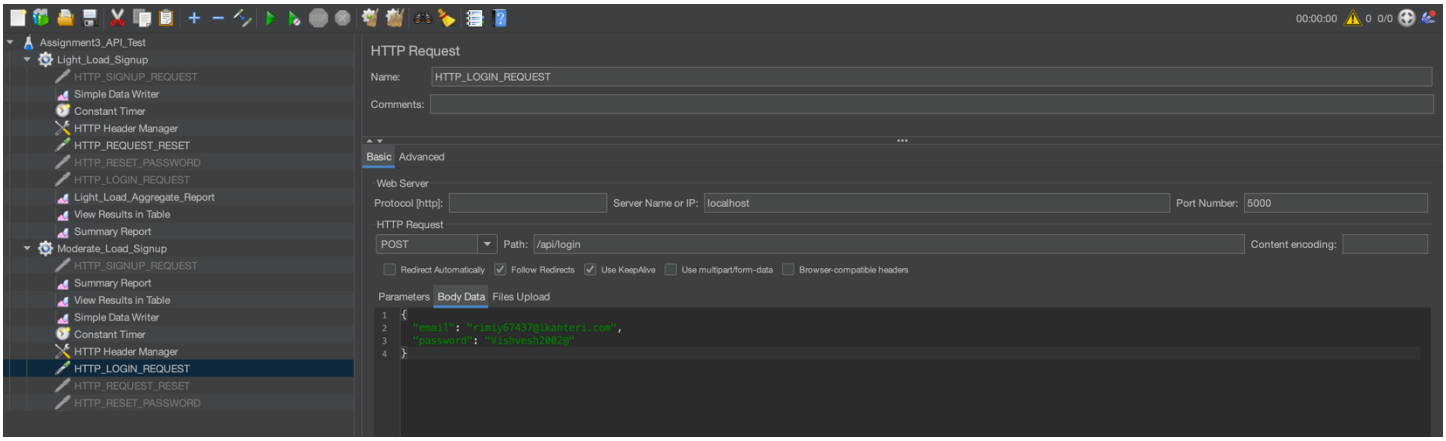


Fig 11: Login HTTP Request for the moderate load group

Request Reset Password HTTP Request:-

I have created an endpoint /api/request-reset which asks for the email id of the user and if the email id exists then it first checks whether the user is registered with us or not if it's registered it's going to send the password reset link over their mail id. And if it's not then it should give an error message to the user that the user is not exists.

I have the created the request for that endpoint for testing the performance of that. Configuration is described in the figure 12.

Example of body data:-

```
{
        "email": "rimiy67437@ikanteri.com"
}
```

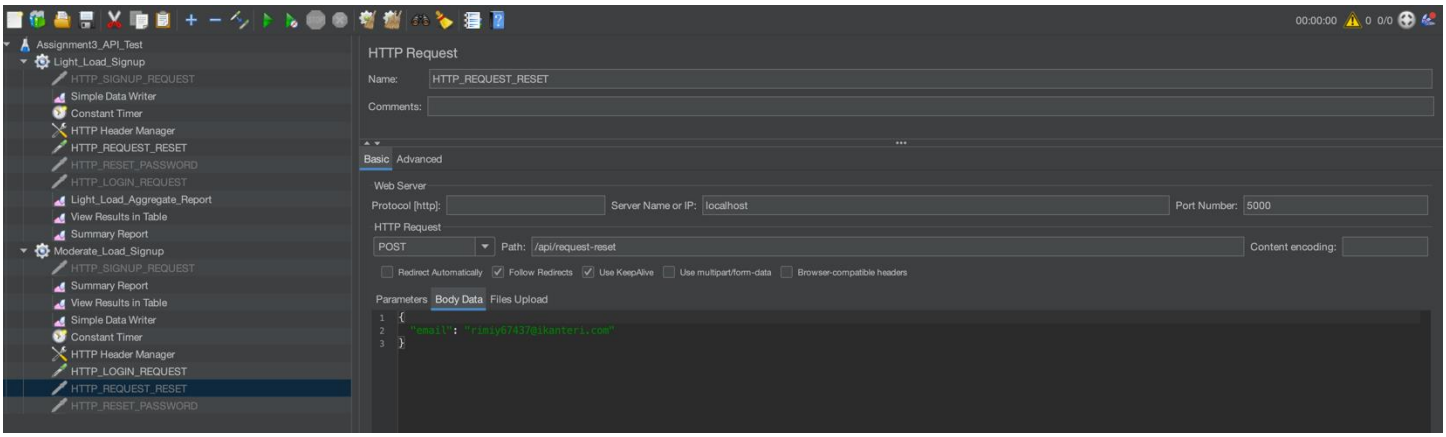


Fig 12: Request Reset Password HTTP Request for the moderate load group

Reset password HTTP Request:

Now, the user gets an email for resetting the password. There is one token in that URL which is being used to authorized an user to reset the password. In this request, my backend expects the token, new password and confirm new password values. For testing the performance I have created this HTTP request to test the /api/reset-password api.

Example of body data:-

```
{
        "token": "InBhdGVsdmlzaHZlc2g4NUBnbWFpbC5jb20i.aIJhsw.bG2gi1Bc_ubxEfNUN3LL1tE5rSM",
        "newPassword": "Hello2090@",
        "confirmPassword": " Hello2090@"
}
```

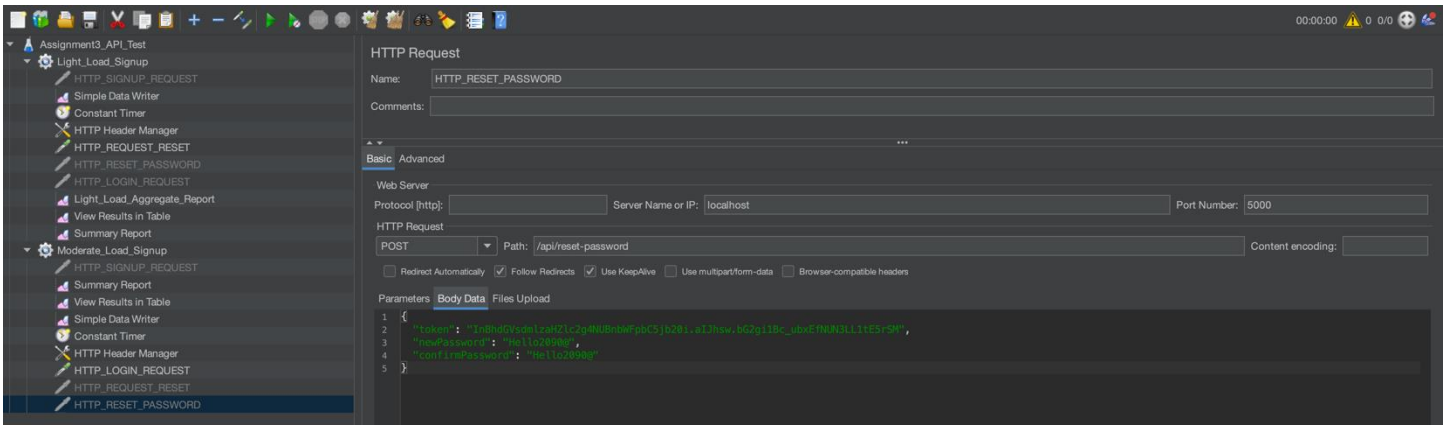



Fig 13: Reset Password HTTP Request for the moderate load group

CSV Result:-

Here I am including the screenshot of the report that's being generated using the Jmeter to capture the required things. This CSV report is for the light load thread group.

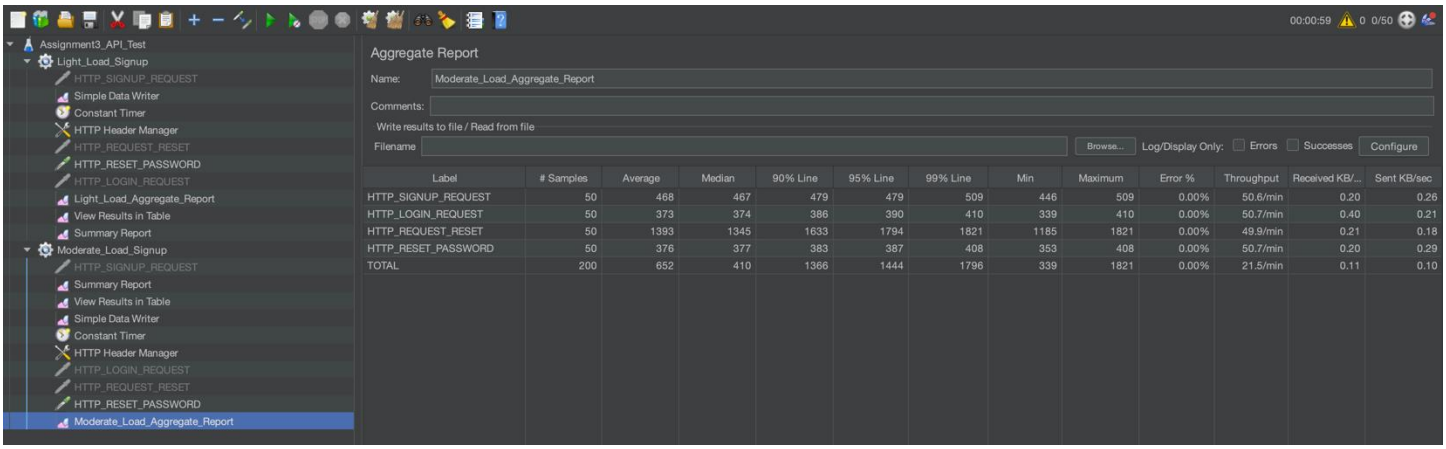


Fig 14: Report generated for the Moderate load thread group

Light Load Frontend APIs:-

As mentioned in the assignment for the frontend APIS, we need to test the frontend load for the light and moderate situations. I have created the 2 different load groups for the frontend and for the backend shown in the figure 15.

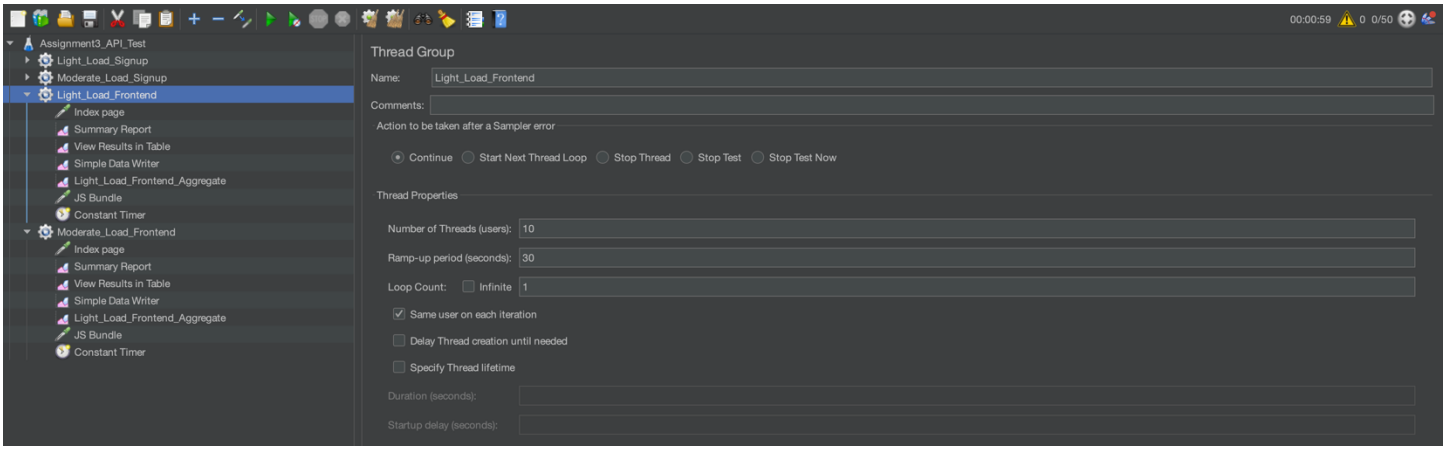


Fig 15: Light and moderate load group for the frontend apis

Configuration of the light load group :-

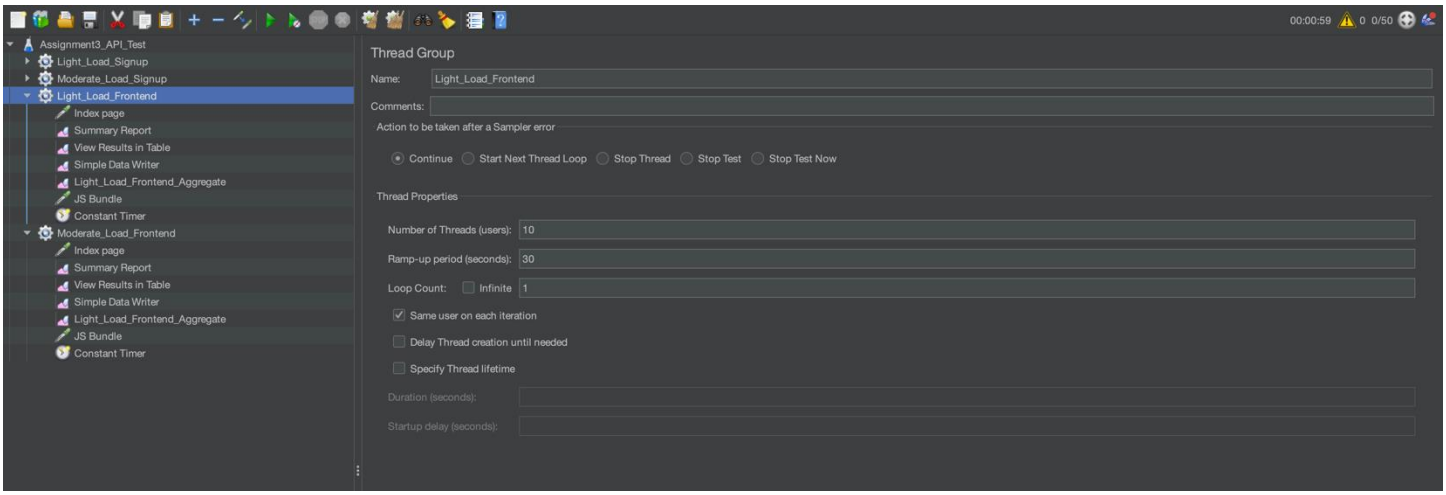


Fig 16: Configuration of the frontend light load group

Configuration of the moderate load group :-

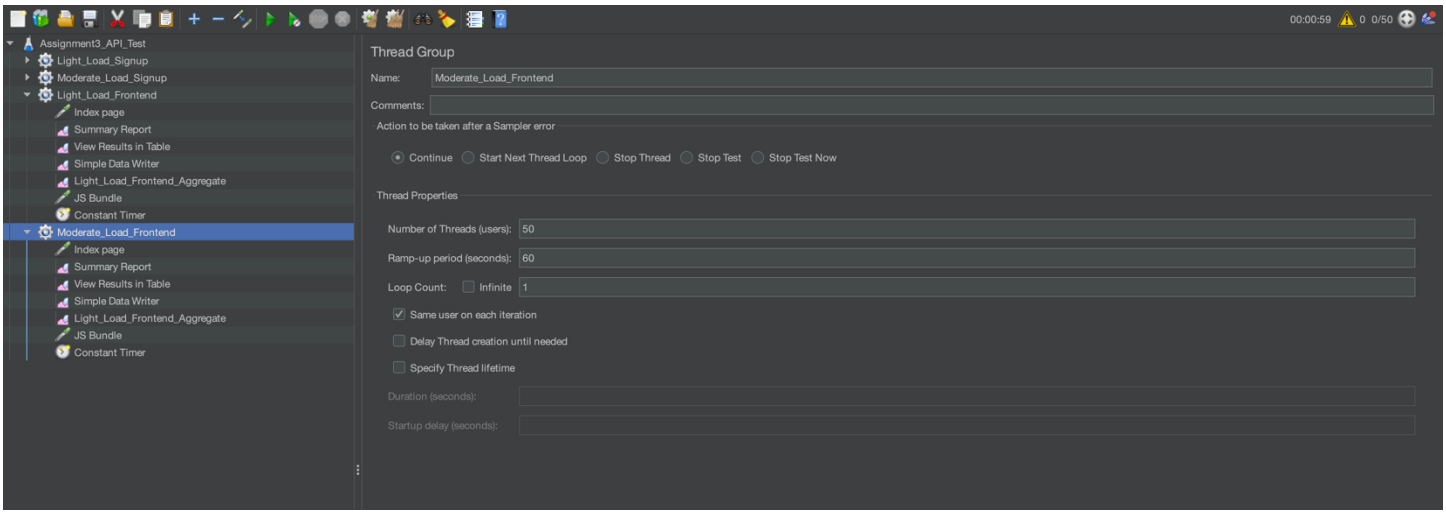


Fig 17: Configuration of the frontend moderate load group

I have added the constant timer for the both light and moderate groups and the configuration is shown in the figure 18.

Fig 18: Configuration of the constant timer

Entry point of the application:-

For the entry point testing, I ran the npm run build command to create the build version of the frontend and then configured the http request accordingly. As the server name would be the localhost and my react app runs on the port 3000. For the entry point the path should be / .

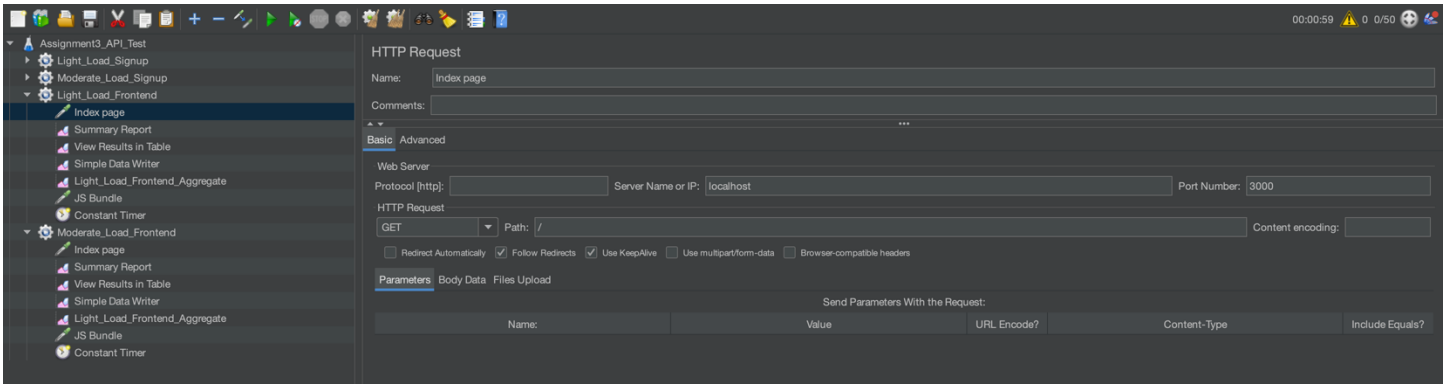


Fig 19: Configuration of the index page testing

JS Bundle:-

I have tested the JS bundle and the path for this is /static/js/bundle.js.

For this the number of users are 10 and the ramping up time is 30 seconds. Also the time delay is 500 ms which I already shown in figure 18.

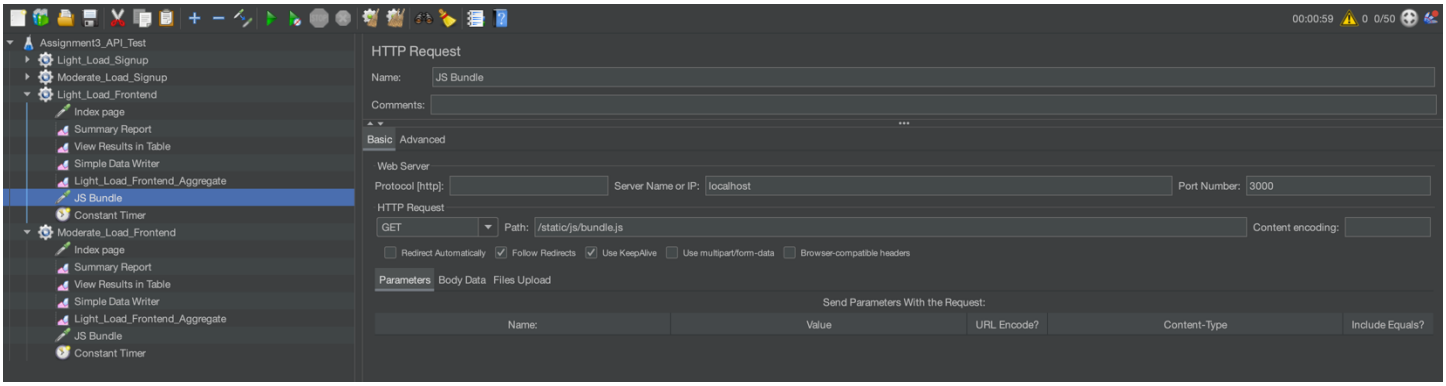


Fig 20: Configuration of the JS Bundle

CSV Report :-

The below is the generated CSV report for the frontend light load group.

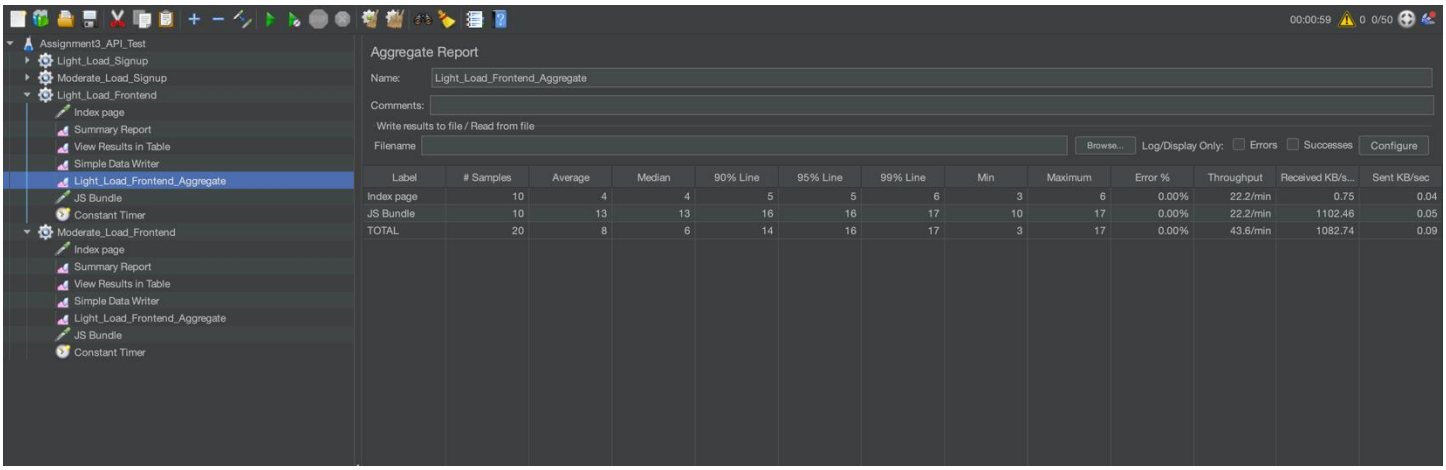


Fig 21: CSV Result for the frontend light load group

Moderate load Frontend APIs:-

I have added the number of users as 50 with the ramping up time of the 30 seconds. Also I have added the constant timer of the 500ms as shown in the figure 22 and 23 respectively.

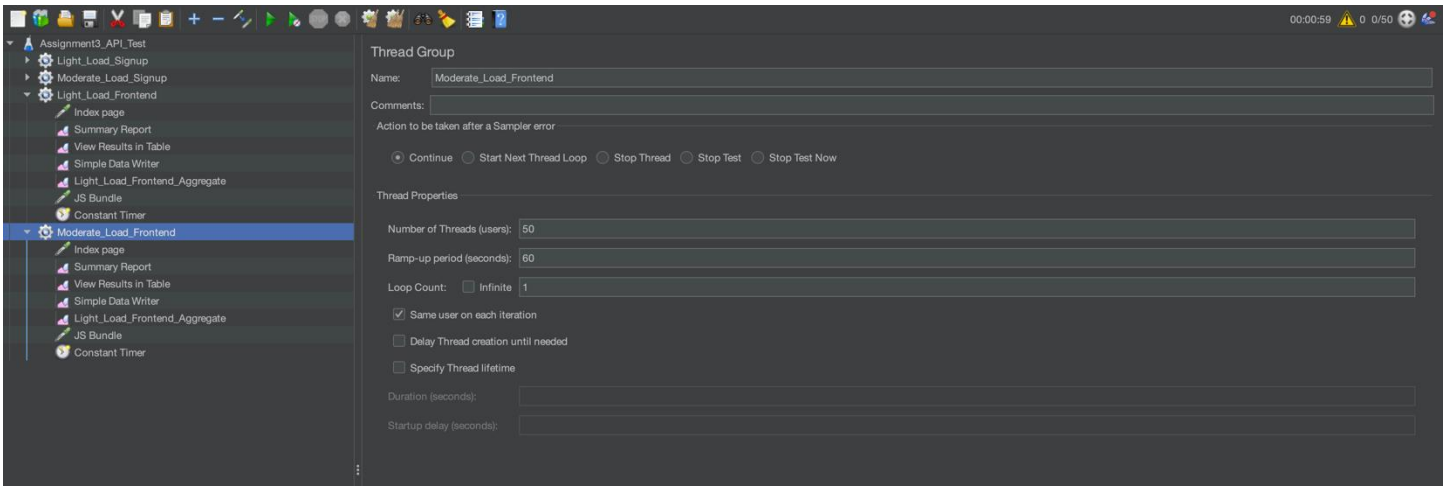


Fig 22: Configuration of the moderate load group

Fig 23: Configuration of the constant timer

Entry point of the application:-

For the entry point testing, I ran the npm run build command to create the build version of the frontend and then configured the http request accordingly. As the server name would be the localhost and my react app runs on the port 3000. For the entry point the path should be / .

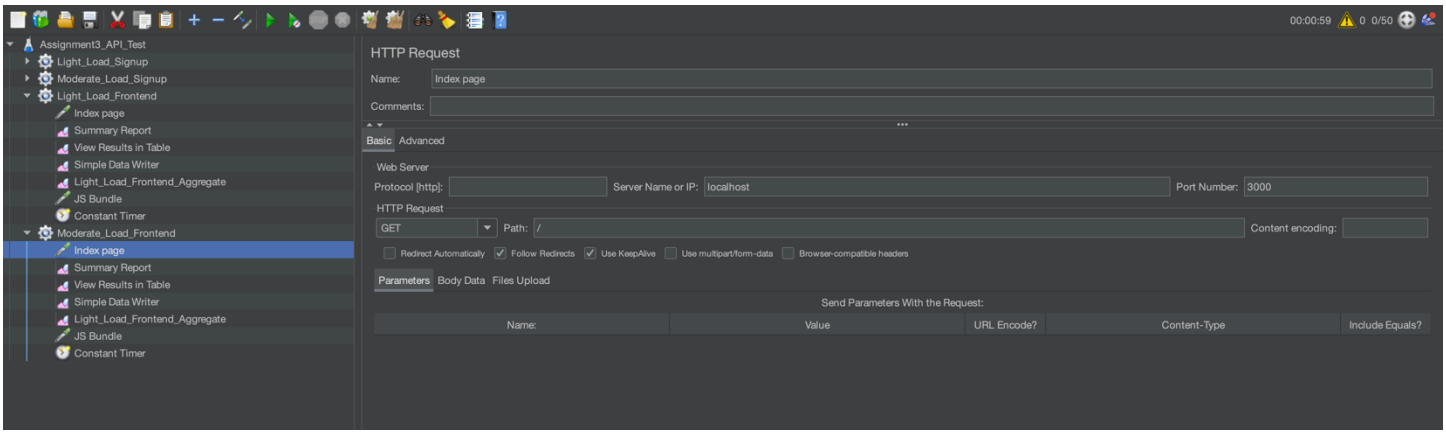


Fig 24: Configuration of the index page testing

JS Bundle:-

I have tested the JS bundle and the path for this is /static/js/bundle.js.

For this the number of users are 50 and the ramping up time is 60 seconds. Also the time delay is 500 ms which I already shown in figure 23.

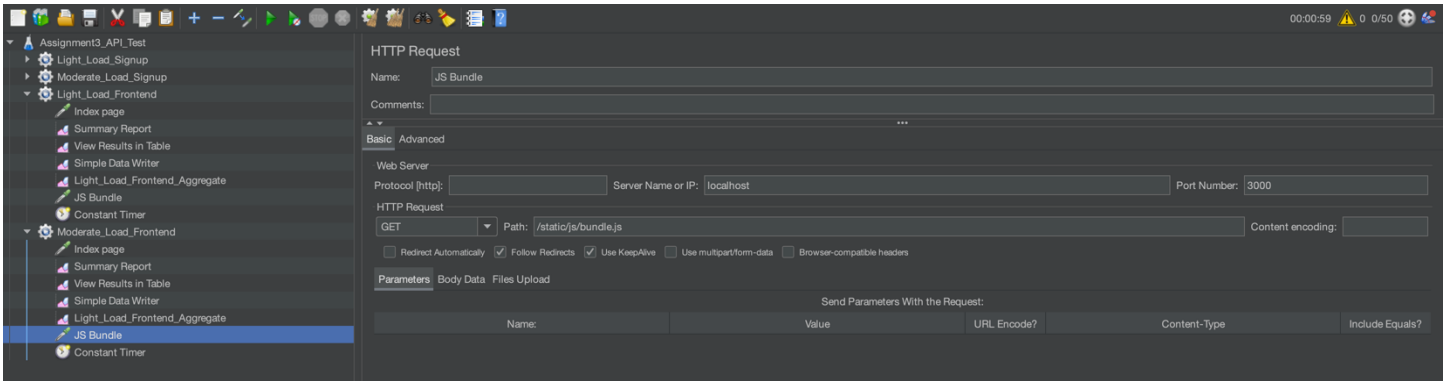


Fig 25: Configuration of the JS Bundle

CSV Report :-

Below is the CSV report of the frontend moderate load group.

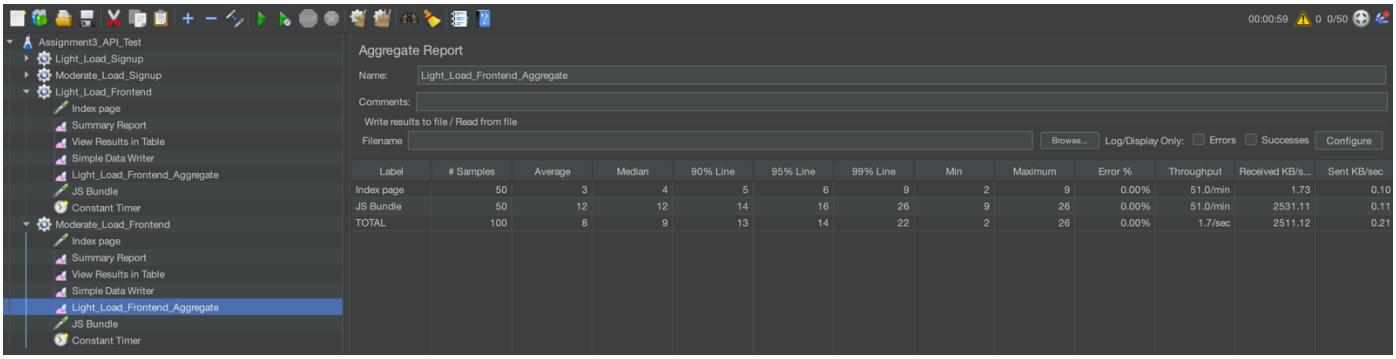


Fig 26: CSV Report for the moderate group load

Worst Bottlenecks :-

After analyzing the generated csv reports, I found the following bottlenecks.

1. **HTTP_REQUEST_RESET (Backend):-**
   This operation has the highest level of response times across light load (avg 1393 ms) and moderate load (1431 ms) as this operation involves the sending an email via an SMTP server due to which it introduces the latency in the communication and email service's processing time.

   This feature is critical feature which is going to help the user to reset their password. Slow performance here can frustrate the user potentially leading to poor user experience.

2. **HTTP_SIGNUP_REQUEST (Backend):-**
   This operation has the average responses time of the 468 ms for moderate and 485 for the light load groups. As this process involves encrypting the password using the bcrypt library and then inserting into the database, it takes some time as both functionalities are computationally expensive.

   User signup is the most critical part as it's the entry point of the application and leading an error or delay here can deter new users impacting the growth of an application.

3. **JS Bundle (Frontend):-**
   In this operation, it has the high data transfer rate (2531.11 in moderate & 1102.46 in light load) which shows that the large files are getting transferred which can slow down the page load time. The bundle size is likely very large which leads to the increases download times.

Client-Side Optimization

1. Lazy loading of the navbar and footer components :-

JMeter results show that the JS Bundle as a bottleneck with an average response time of 12-13 ms and a high data transfer rate which indicated the large javascript bundles. And I can see that the Footer and Navbar components are rendering in all three components.

So, I implemented the lazy loading for the Navbar and Footer components to defer their loading until needed. By splitting the navbar and footer into separate chunks the initial JS payload is reduced.

Changes I have done :-

I imported the Suspense and used React.lazy to dynamically import Navbar and Footer components as shown in the Fig 27.

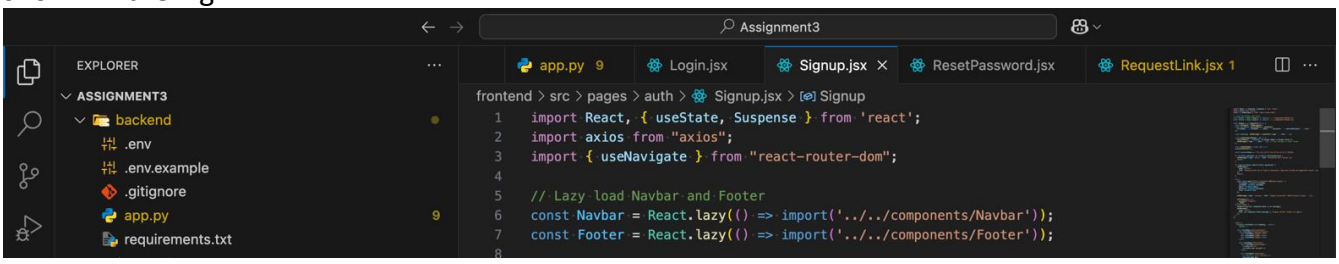


Fig 27: Lazy loading for the frontend

I then wrapped the component in Suspense with a fallback to display the loading text as shown in figure 28.

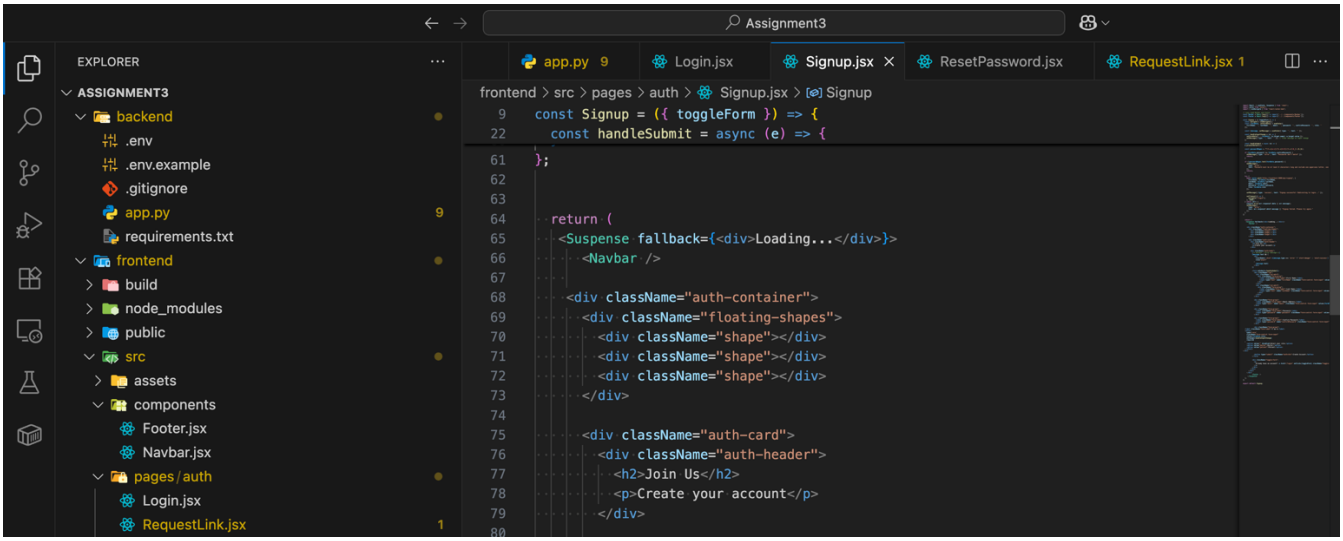


Fig 28: Wrapped component using Suspense

2. Caching Form Data in LocalStorage:-

The Login, Signup, Reset password all are handling form submissions that involve network requests which can be delayed if the users repeatedly enter similar data. Thus this can increase the loading time and repetitive form inputs increase user interaction.

To solve we can use the caching mechanism to cache the form data like email in localstorage to prefill fields on next visit reducing the user input times and improving the speed.

This can reduce the user interaction time and improve the performance of the frontend . It enhances the user experience by minimizing form entry delays.

I made changes in the code and initialized the email field in formData with localStorage.getItem('lastEmail') to prefill it if previously stored as shown in the figure 29.

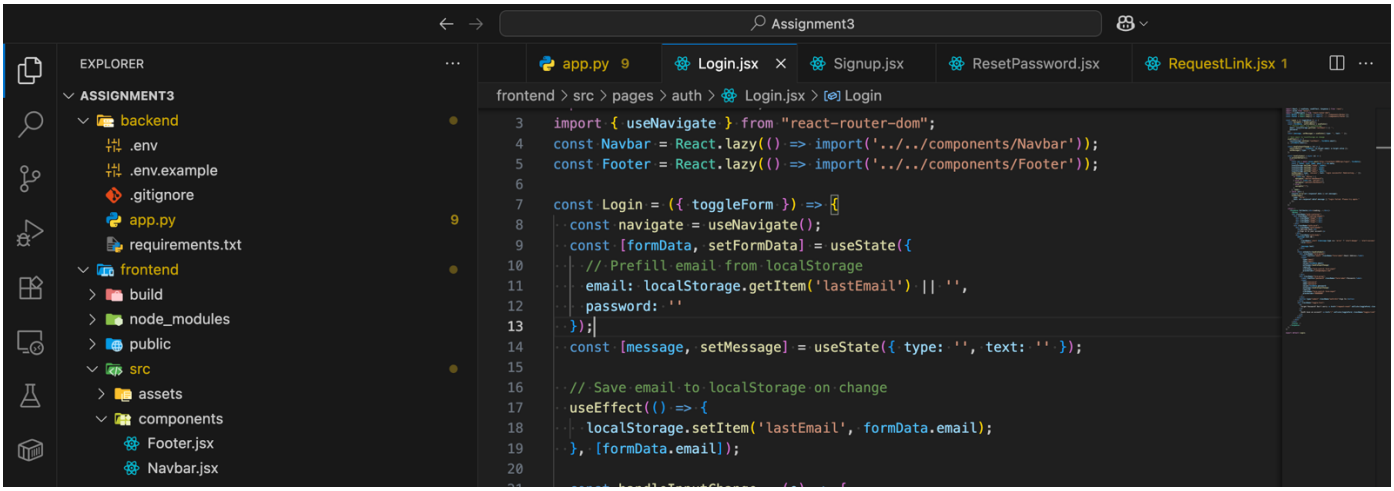


Fig 29: Caching form data in localstorage

Server side optimizing :-

1. Optimizing the Bcrypt hashing for password operations :-

As from the results it is likely that the password hashing is computationally expensive due to its default work factor which is typically 12 rounds. The high latency is partly due to this. Reducing the rounds to 10 can decrease the hashing time while maintaining sufficient security. This optimizes the both /api/signup and /api/reset-password endpoints.

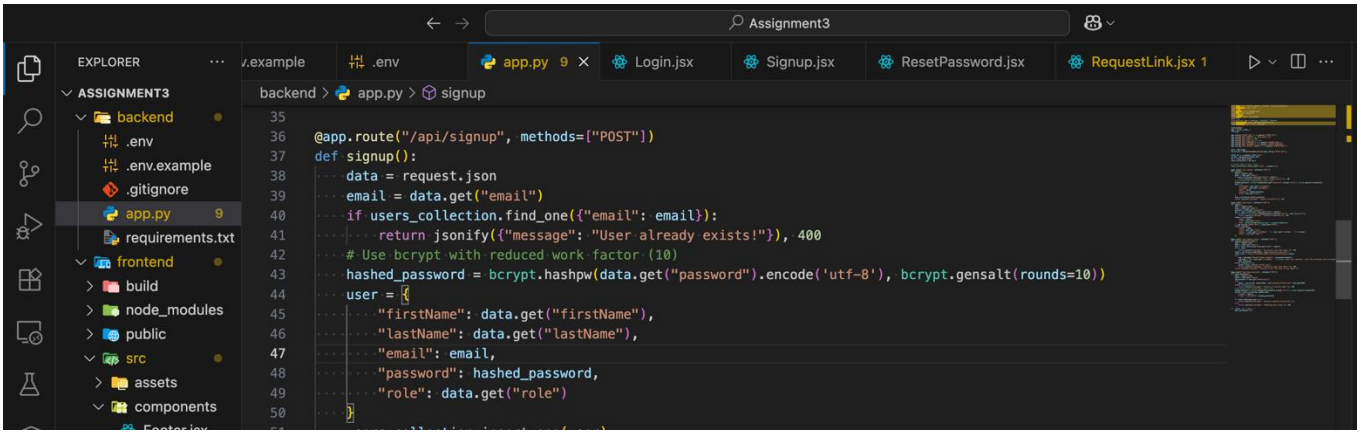


Fig 30: Changing the gensalt rounds to 10 from 12

2. Database indexing for the email queries:-

The signup request and login request endpoints have the high response times which is partly due to the mongodb queries on the email field. Without an index, Mongodb performs a full collection scan, which is slow for large datasets.

To solve this issue I have added an index on the email field in the users collection to speed up the lookups.

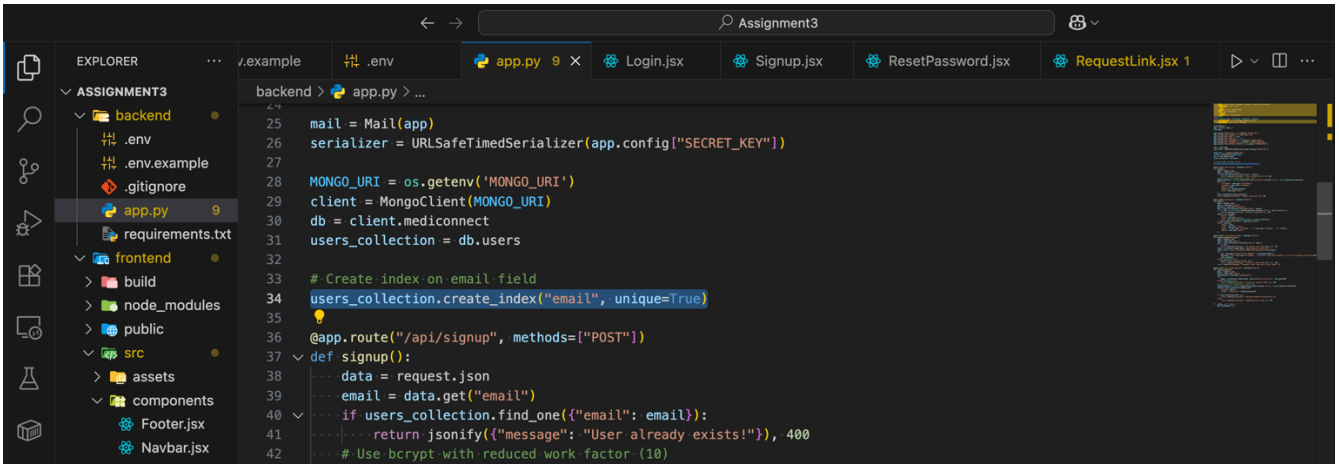


Fig 31: Adding indexing on email field

Table Comparison

Light Load

For the light load, I have tested all my backend APIs and frontend routes before applying the optimization and I also tested everything after applying the optimization. I am comparing the both the things in the table formats in below tables.

Light Load Backend Comparison

In Table 1, I have demonstrated the before and after comparison for all the HTTP requests. And we can clearly see the improvements in the below table.

Table 1: Comparison chart for light load backend apis

Label	Before Avg (ms)	After Avg (ms)	Before 95% (ms)	After 95% (ms)	Before Throughput	After Throughput	Error %
HTTP_SIGNUP_REQUEST	485.00	295.00	508.00	301.00	0.36	0.37	0.000%
HTTP_LOGIN_REQUEST	371.00	198.00	379.00	203.00	0.37	0.37	0.000%
HTTP_REQUEST_RESET	1431.00	1586.00	1523.00	1660.00	0.35	0.45	0.000%
HTTP_RESET_PASSWORD	374.00	203.00	386.00	206.00	0.37	0.37	0.000%
TOTAL	665.00	571.00	1437.00	1601.00	0.11	0.12	0.000%

Light Load Frontend Comparison

As I have improved the frontend as well, there is an improvement that we can see in the table 2.

Table 2: Comparison chart for light load frontend apis

Label	Before Avg (ms)	After Avg (ms)	Before 95% (ms)	After 95% (ms)	Before Throughput	After Throughput	Error %
Index Page	4.00	3.00	5.00	3.00	0.37	0.37	0.000%
JS Bundle	13.00	13.00	16.00	15.00	0.37	0.37	0.000%
Total	8.00	10.00	16.00	12.00	0.73	0.73	0.000%

Moderate Load Backend Comparison

As I am testing the light and moderate loads, i have tested the code with changes and without changes and got the below result as shown in table 3.

Table 3: Comparison chart for moderate load backend apis

Label	Before Avg (ms)	After Avg (ms)	Before 95% (ms)	After 95% (ms)	Before Throughput	After Throughput	Error %
HTTP_SIGNUP_REQUEST	468.00	286.00	479.00	298.00	0.84	0.85	0.000%
HTTP_LOGIN_REQUEST	373.00	198.00	390.00	210.00	0.84	0.85	0.000%
HTTP_REQUEST_RESET	1393.00	1542.00	1794.00	1704.00	0.83	0.83	0.000%

HTTP_RESET_PASSWORD	376.00	208.00	387.00	210.00	0.84	0.85	0.000%
TOTAL	652.00	558.00	1440.00	1657.00	0.36	0.29	0.000%

Moderate Load Frontend Comparison

Table 4: Comparison chart for moderate load frontend apis

Label	Before Avg (ms)	After Avg (ms)	Before 95% (ms)	After 95% (ms)	Before Throughput	After Throughput	Error %
Index Page	3.00	2.00	6.00	5.00	0.85	0.85	0.000%
JS Bundle	12.00	11.00	16.00	13.00	0.85	0.85	0.000%
Total	8.00	7.00	14.00	13.00	1.69	1.69	0.000%

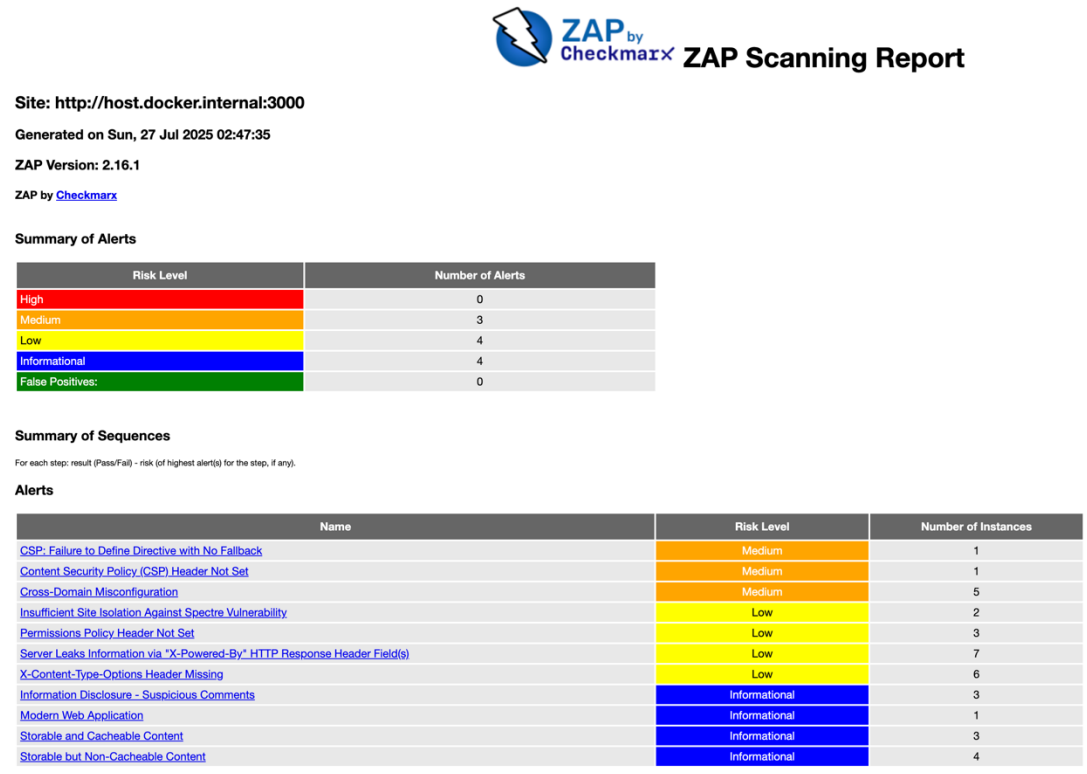
Analysis

The optimizations significantly improved the backend performance for most of the endpoints. HTTP_LOGIN_REUEST and HTTP_RESET_PASSWORD saw the largest improvements, with an average response time reductions of approximately 46% under both load group. HTTP_SIGNUP_REQUEST also improved by around 39% in both scenarios.

JS Bundle and index pages saw minor improvements in average and 95th percentile response times under moderate and light load.

OWASP ZAP Report

I have scanned my codebase using the OWASP ZAP and I got the following reports. The figure 32 shows the vulnerabilities in the frontend and the figure 33 shows the vulnerabilities in the backend.



ZAP by Checkmarx **ZAP Scanning Report**

Site: http://host.docker.internal:3000

Generated on Sun, 27 Jul 2025 02:47:35

ZAP Version: 2.16.1

ZAP by Checkmarx

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	3
Low	4
Informational	4
False Positives:	0

Summary of Sequences

For each step: result (Pass/Fail) - risk (of highest alert(s) for the step, if any).

Alerts

Name	Risk Level	Number of Instances
CSP: Failure to Define Directive with No Fallback	Medium	1
Content Security Policy (CSP) Header Not Set	Medium	1
Cross-Domain Misconfiguration	Medium	5
Insufficient Site Isolation Against Spectre Vulnerability	Low	2
Permissions Policy Header Not Set	Low	3
Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)	Low	7
X-Content-Type-Options Header Missing	Low	6
Information Disclosure - Suspicious Comments	Informational	3
Modern Web Application	Informational	1
Storable and Cacheable Content	Informational	3
Storable but Non-Cacheable Content	Informational	4

Fig 32: OWASP scan report for the frontend

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	2
Low	2
Informational	1
False Positives:	0

Summary of Sequences

For each step: result (Pass/Fail) - risk (of highest alert(s) for the step, if any).

Alerts

Name	Risk Level	Number of Instances
Content Security Policy (CSP) Header Not Set	Medium	2
Cross-Domain Misconfiguration	Medium	3
Permissions Policy Header Not Set	Low	3
Server Leaks Version Information via "Server" HTTP Response Header Field	Low	3
Storable and Cacheable Content	Informational	3

Fig 33: OWASP scan report for the backend

Vulnerability 1: Cross Domain Misconfiguration

There is no restriction in the CORS. By default it allows all the origins which is insecure. To fix it we need to configure the flask-cors to restrict the origins to the trusted routes only.

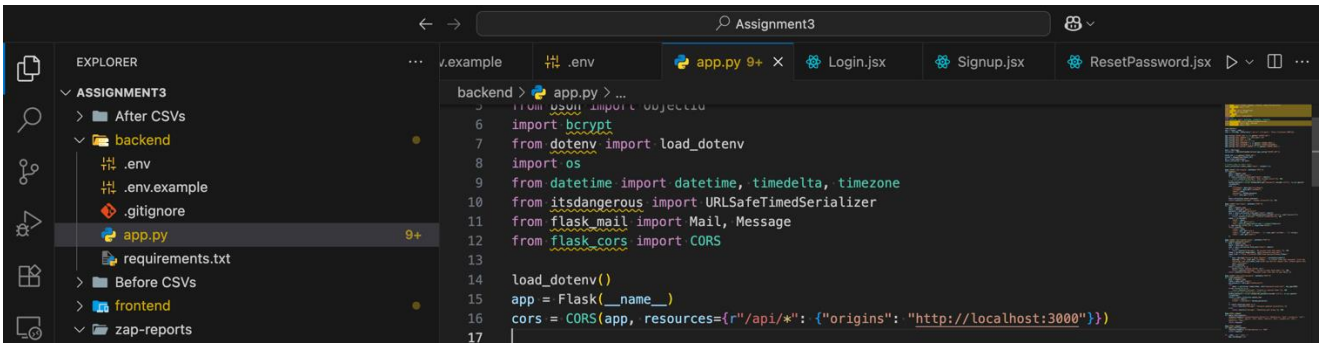


Fig 34: CORS restriction

Vulnerability 2: Permission policy header not set

The absence of Permissions-Policy headers allows browsers to enable the features like geolocation or camera access by default which is insecure. We need to add the Permissions-Policy header to disable unused features.

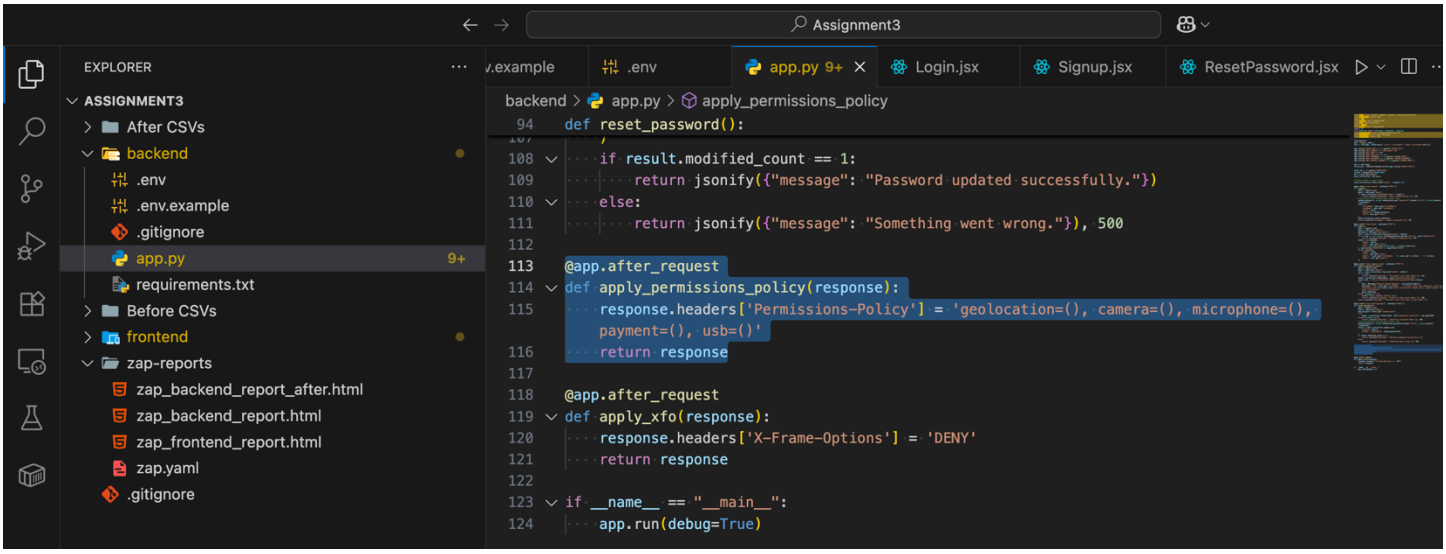


Fig 35: Add Permissions-Policy header

After fixing the both the vulnerabilities in the backend, I have scanned the codebase again and I got the following report in which the vulnerabilities were removed which I have fixed.

ZAP Scanning Report

Site: http://host.docker.internal:5000

Generated on Sun, 27 Jul 2025 03:20:55

ZAP Version: 2.16.1

ZAP by Checkmarx

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	1
Low	1
Informational	1
False Positives:	0

Summary of Sequences

For each step: result (Pass/Fail) - risk (of highest alert(s) for the step, if any).

Alerts

Name	Risk Level	Number of Instances
Content Security Policy (CSP) Header Not Set	Medium	3
Server Leaks Version Information via "Server" HTTP Response Header Field	Low	3
Storable and Cacheable Content	Informational	3

Alert Detail

Medium	Content Security Policy (CSP) Header Not Set
Description	Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks are used for everything from data theft to site defacement or distribution of malware. CSP provides a set of standard HTTP headers that allow website owners to declare approved sources of content that browsers should be allowed to load on that page — covered types are JavaScript, CSS, HTML frames, fonts, images and embeddable objects such as Java applets, ActiveX, audio and video files.
URL	http://host.docker.internal:5000
Method	GET
Parameter	

Fig 36: After applying the fixes

Monitor web application (Prometheus and Grafana)

Prometheus Configuration :-

To configure Prometheus and Grafana, I have used the docket compose and yaml file as shown in below figures.

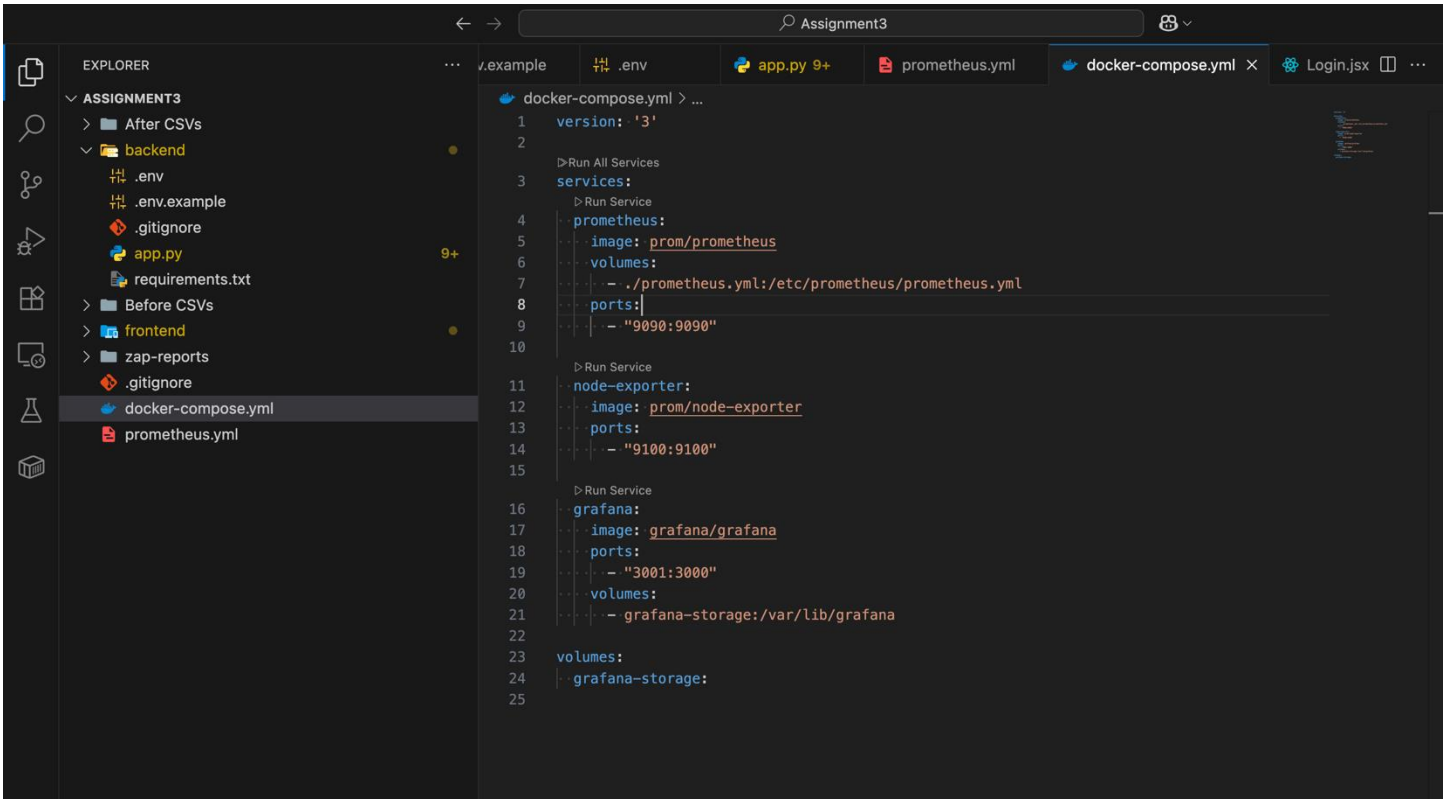


Fig 37: Docker compose configuration

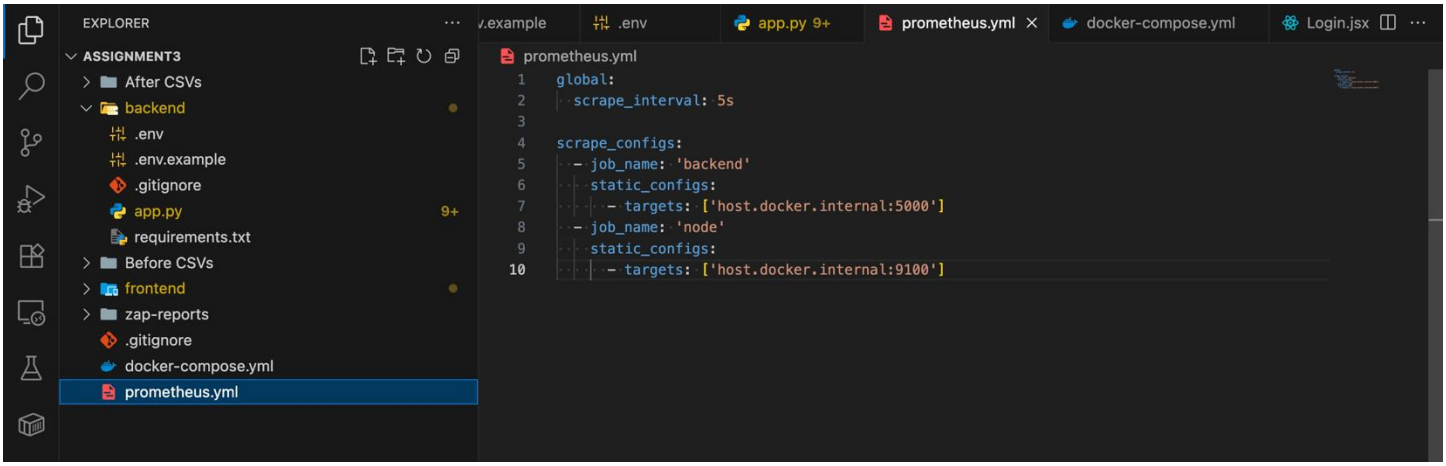


Fig 38: Prometheus YAML file

Grafana dashboard :-

Fig 39: Grafana Dashboard

CPU Utilization :-

For the CPU utilization I have created the panel as shown in the figure 40.

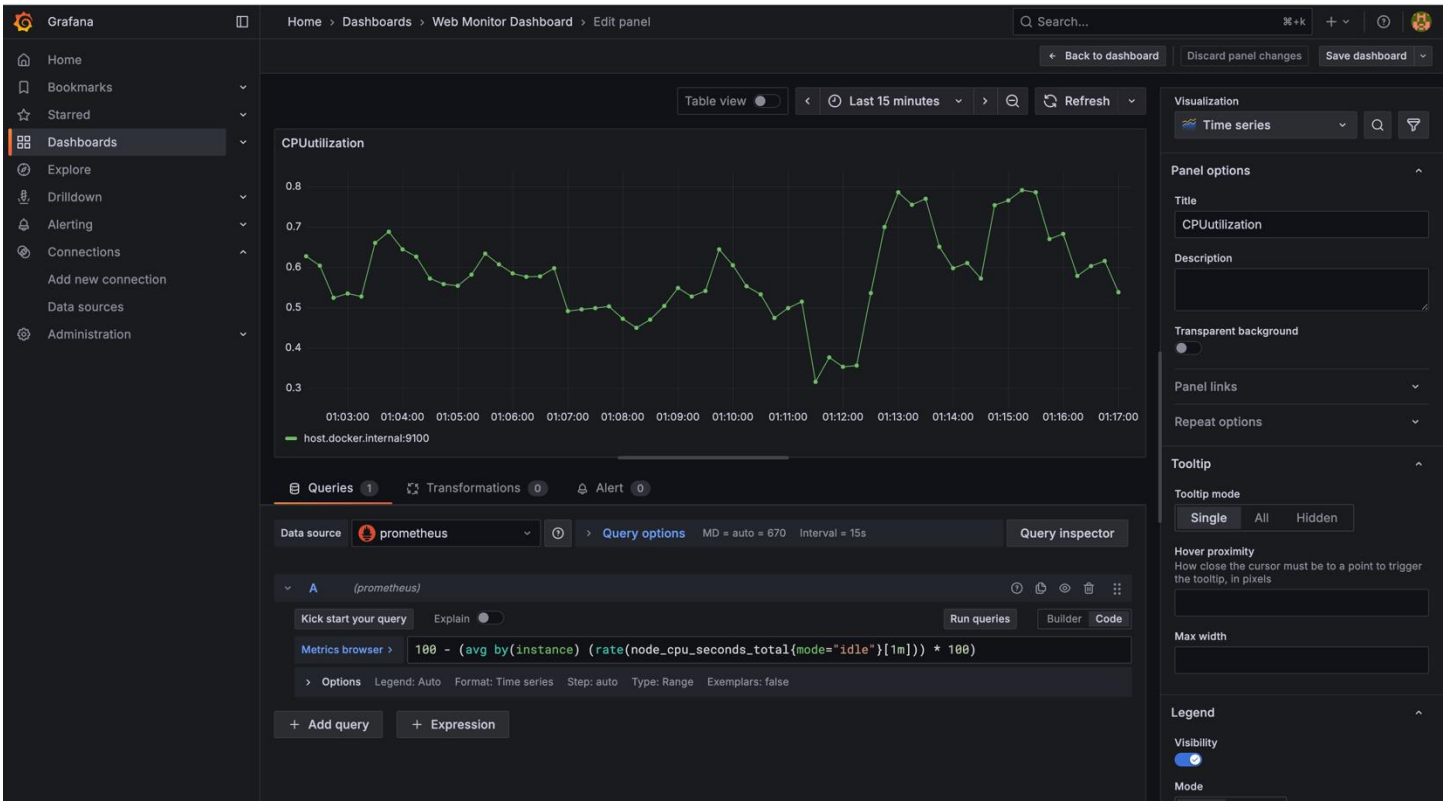


Fig 40: CPU utilization panel

Memory Utilization:-

For the memory utilization, I have created the panel as shown in figure 41.

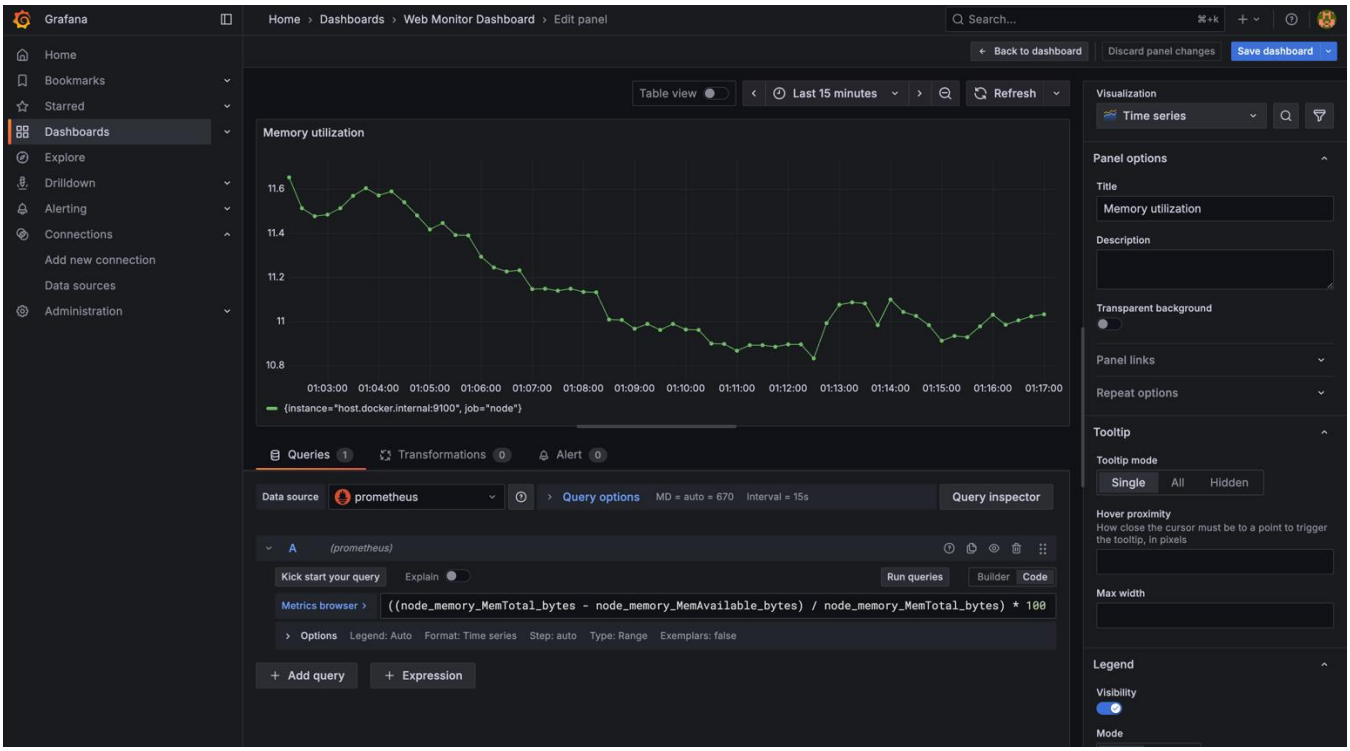


Fig 41: Memory utilization panel

Error Rate:-

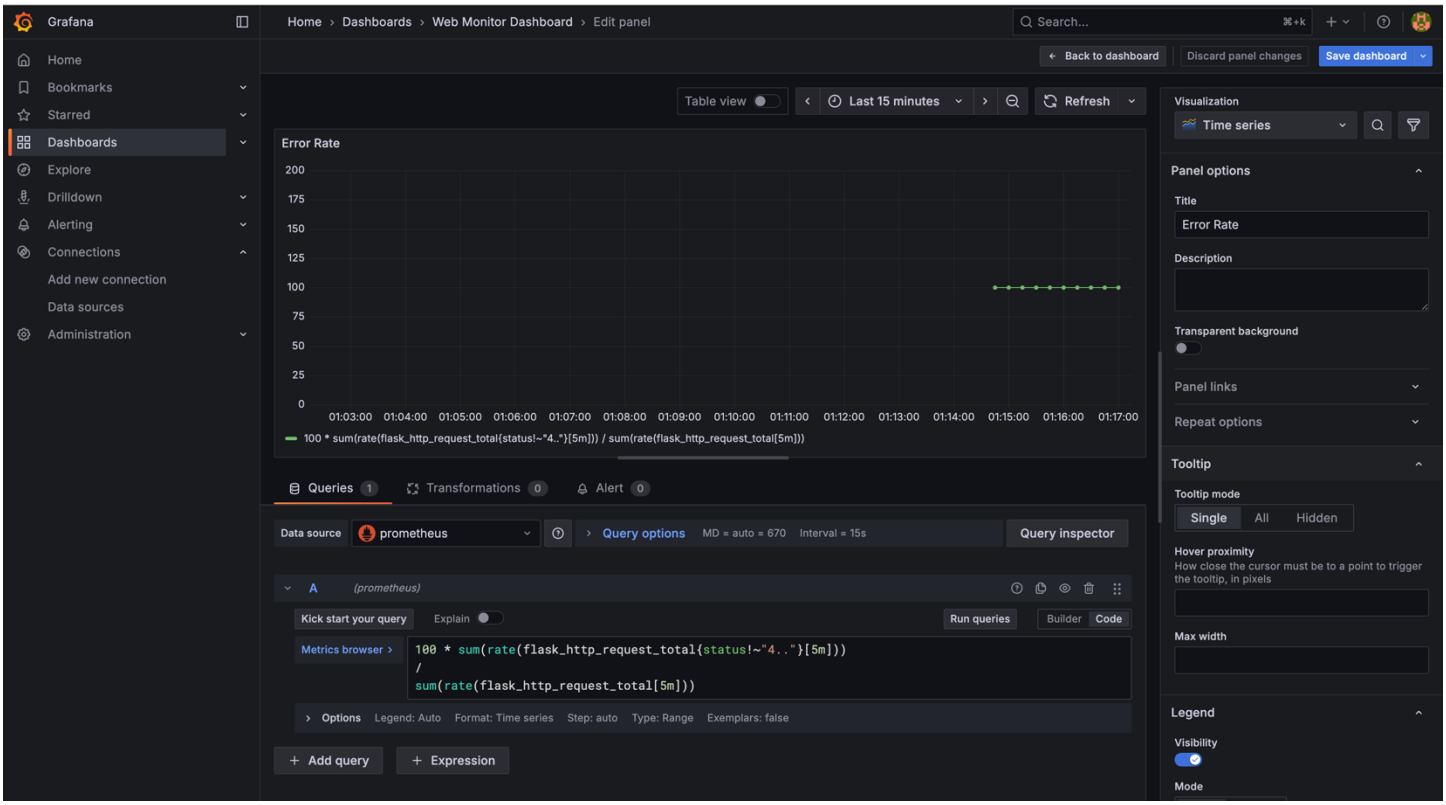


Fig 42: Error Rate panel

Request Latency:-

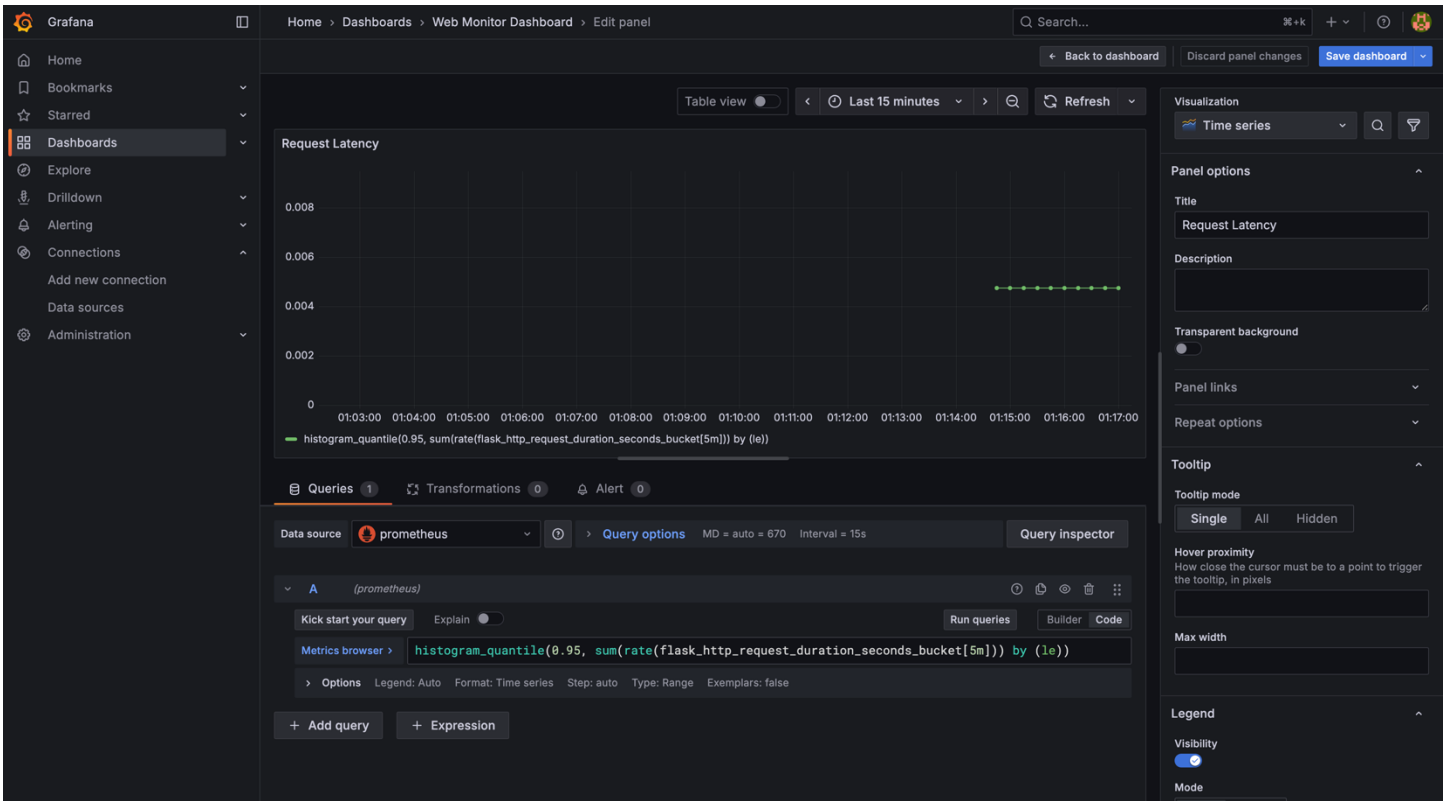


Fig 43: Request Latency panel