

# LEVERAGING HEADLESS CONTENT MANAGEMENT SYSTEM AS A SERVICE IN A SERVICE-BASED ARCHITECTURE: ENHANCING USER EXPERIENCE AND OVERCOMING RESOURCE LIMITATIONS FOR START-UPS

ILMA ARIFIAN<sup>1</sup>, GEDE PUTRA KUSUMA<sup>2</sup>

<sup>1,2</sup>Computer Science Department, BINUS Graduate Program – Master of Computer Science, Bina Nusantara University, Jakarta 11480, Indonesia

E-mail: <sup>1</sup>ilma.arifiany@binus.ac.id, <sup>2</sup>i.negara@binus.ac.id

## ABSTRACT

Microservices and service-oriented architecture have acquired widespread recognition in recent years as efficient solutions for building scalable, resilient, and simple-to-maintain software systems. Although they offer an optimal solution for large organizations with intricate and dynamic systems, their suitability for startups may be uncertain because of the constraints a new company may encounter, such as budget and human resources. Hence, the architectural design should be modified following the specific requirements of the company in question. This case study focused on building a service-based architecture that would address a new company's constraints while still emphasizing the application's user experience. We implemented this service-based architecture by utilizing a headless content management system and building additional reusable services using the plugin and database feature of the content management system. This architecture allowed developers to accelerate and simplify the development of the company's backend services, enabling a focus on improving features impacting user experience. In our resulting architecture, each service operated on its own, with distinct responsibilities, lowering the reliance on one backend. This architecture also improved the website's performance, as shown by a fast response time, high throughput, and an overall good load speed.

**Keywords:** *Composable Architecture, Headless Content Management System, Microservices Architecture, Modern Web Development, Service-Oriented Architecture*

## 1. INTRODUCTION

Microservice and service-oriented architecture (SOA) are modern architectural systems known for their exceptional performance and are well-suited for enterprises with large and complex product offerings [1]. SOA is very well-suited for extensive, complicated, organization-wide systems needing integration with several applications and services. On the other hand, the microservices pattern is more suited for smaller, well-segmented web-based systems [2]. However, implementing both architectures is not always the right choice for start-up companies that still face various limitations, such as the limitation of human resources and architecture budget [3, 4]. Start-up companies sometimes encounter constraints such as limited human resources and budget, which render the implementation of both architectures unsuitable for them. The intricate nature of microservices may extend the development and maintenance period

because the services may require personnel with specialized skills in every area each service controls and supports [5]. Thus, companies might need a less intricate architecture than microservices and SOA, an architecture scalable to greater complexity when company constraints are addressed [4]. However, user experience (UX) is also a vital factor for startup companies to satisfy users of their applications, and it cannot be overlooked. Poor UX can lead the users to criticize the presentation of the product even if the product idea itself was good. Good UX can increase user acquisition, retention, and satisfaction. By prioritizing UX from the beginning, startups can create products that resonate with users, stand out in the market, and pave the way for long-term growth.[6].

This case study aims to create a service-based architecture which incorporates microservices and SOA method within company limitations. The object of this case study is a website for a new

company that helps people seeking work and helps organizations find employees. They have several products for which development is required: Events, Mentorships, Virtual Work Experiences, and CV Reviews. They also planned to create more products in the future, hence the necessity to have a scalable and easily developed application.

In the past, this company used the WordPress platform as the base for both the backend and the frontend, creating monolithic architecture. WordPress presents challenges. The complexity and limited adaptability of its code make it difficult to develop into a more important website platform, resulting in a steep learning curve [7]. Furthermore, because of the monolithic nature of the WordPress platform being used, data stored within it can only be accessed through the CMS and the integrated website itself. This limits the ability to access data from different websites or applications, thus hindering further growth [8]. Therefore, because of the growing number of features being developed, it became necessary to migrate from the monolithic structure of WordPress to a more modernized approach that would satisfy the company's needs. Third party plugins that helped WordPress handle additional features also slowed down the website, thus the use of this architecture also sacrificed user experience.

To summarize, building an inexpensive, intuitive, and adaptable system without sacrificing user experience is what this company needs. We propose a modern and less complex service-based architecture, where services with distinct responsibilities are developed, but not as loosely coupled as microservices. This architectural design can also be described as a composable architecture, as each component is reusable and integrable into different applications. A service-based approach will allow developers to divide the application into different necessary services as it grows.

For faster development, A headless content management system (Headless CMS) can serve as the basis for developing an appropriate website, meeting the company's requirement for rapid and efficient development. A Headless CMS is a content management system that separates the frontend, where content is presented, from the backend, where content is stored. Headless CMS can manage all aspects of data management and often feature a separate dashboard from the application [8, 9]. Headless CMS simplifies data and component structuring, letting developers build applications with improved efficiency. By leveraging the CMS's existing database, smaller services can be developed and integrated into the main dashboard with

simplicity, eliminating the need for separate frontend.

Headless CMS is responsible for presenting and processing all the website's content through its API. Therefore, its performance is crucial to the application's overall performance. A Headless CMS solely as a backend makes the application vulnerable to a single point of failure. One of the technical aspects related to UX that should be addressed, besides failure considerations, is the speed and security of the application. Speed is very important because of the high level of user interaction on the site. Therefore, further investigation is needed to determine whether this architecture requires additional services able to share tasks with Headless CMS. This case study not only aimed to build suitable architecture for start-up companies but also explored whether adding services would improve site performance or if a Headless CMS alone sufficed. By comparing the performance of the various services developed, the architecture with optimal performance can be identified. a clear overview of how to work with the CMS.

## 2. LITERATURE REVIEW

Numerous companies at an early stage adopted Monolithic Architecture because of its advantages in the speedy development of applications with minimal resource needs. Monolithic Architecture is a traditional software development approach where an application is built as a single unit. All components of the application, including the user interface, business logic, and data access layer, are tightly coupled and deployed together as a single entity [3]. Therefore, monolithic architecture is simpler to build, especially for small teams, and easier to deploy. Monolithic architecture requires lower operational overhead, as fewer resources are needed. But being a single unit means that if one application process experiences a significant spike in demand, the entire architecture must be upgraded. If the application faces troubles, then the whole architecture might be affected. Adding or improving functionality in a monolithic program becomes increasingly complex as its code base expands, leading to difficulties in adopting new technologies or frameworks for specific components because of technological constraints. High complexity limits the capacity for experimentation and presents obstacles to executing innovative ideas [10]. Therefore, monolithic architecture is not suitable for companies that aim for scalability and adaptability.

One way companies have dealt with this issue is by using Service-Based Architecture, in

which applications are divided into smaller, independent services [11]. Examples of this approach include Services-Oriented Architecture and Microservices. Service-oriented architecture (SOA) is a software framework that decomposes software components into various services that interact with each other as a cohesive entity. SOA aims to simplify complex software systems by transforming them into reusable services that can be accessed by various applications and users, known as service consumers. These services can be viewed as fundamental components for the development of new applications. Every component of an SOA service carries a unique responsibility and comes with an interface that encompasses input and output parameters, as well as the communication protocols for accessing it [12]. SOA services can be registered and reused in multiple application environments. This enables developers to make use of pre-existing functionality with no redevelopment.

SOA provides several benefits when compared to monolithic architecture. SOA enables companies to separate applications into independent services that can be modified, constructed, and deployed individually. This allows for enhanced flexibility in system development and maintenance. The independent nature of services ensures that modifications made to one service do not automatically impact other services or the overall program, hence enabling flexibility and scalability. This helps the maintenance and development of the application [3, 11].

Microservice architecture is another architecture that breaks an application into several services. These services can be deployed independently, loosely coupled, and are generated according to specific business requirements. This architecture is similar to SOA, as both are founded on the concept of services [13]. These services work by exchanging messages with each other through message passing. This architecture distinguishes itself from monolithic architecture and service-oriented architecture by its focus on scalability, independence, and semantic cohesion of each individual component inside the system. Microservice and SOA also differ in service granularity and communication method [2, 14].

While Microservices and SOA offer benefits, they also present drawbacks for start-up companies. For companies in the initial phases of development, this may provide considerable complexity right from the beginning. Every service requires its own infrastructure, including its database, servers, and environment configurations. The team's focus on validating their business idea

might be hampered by the need to undertake complex tasks such as inter-service communication management, data consistency maintenance, and fault tolerance implementation. Microservices' complexity may prolong development and maintenance, creating challenges for resource-constrained startups [3]. Start-ups often have small teams with limited technical expertise. Managing a distributed system like SOA or microservices requires specialized skills in areas such as DevOps, network security, and distributed systems. Monolithic architecture or simpler service-oriented designs can often meet the initial needs of a start-up without the overhead. Instead of resorting back to monolithic architecture, another solution might be possible. In this case, a Headless Content Management System might be utilized to create simpler Service-Based Architecture instead of creating services one by one which consume resources and time.

A Headless Content Management System is a CMS that separates where content is stored ('body') from where it is presented ('head'). The data generated by this CMS is delivered over a Representational State Transfer (REST) Application Programming Interface (API), enabling its utilization across several platforms. This enhances its adaptability and provides superior scalability compared to monolithic CMS [9, 15]. By employing headless architecture, developers have the freedom to select an appropriate technology and framework for developing the frontend layer. Content delivery is effortless with APIs, irrespective of whether it is a web app, mobile app, voice assistant, or another digital channel. [16].

Because it is independent from the presentation layer of the application, this decoupling aligns with the principles of service-based architecture, where services are modular and reusable. The design and architecture of Headless CMS are structured to allow seamless content integration with any platform, adhering to four fundamental principles: flexibility, performance, security, and affordability. Headless CMS platforms scale independently, as they focus solely on content storage and delivery. In a service-based architecture, this means the content service can scale based on demand without affecting other services. For example, during high traffic, the content delivery layer can be scaled independently of the transaction or authentication services.

Flexibility allows companies to quickly respond to the fluctuation of consumer expectations and new developments in technology or microservices. Application developers get the

freedom to choose the technologies and tools to be used in the application. The primary objective of a Headless CMS is to segregate the processes of content creation, storage, and management from the responsibilities of data display and delivery. Compared to a conventional content management system that merely provides template designs, this solution reduces worries regarding security, scalability, usability, and frontend technology [17]. These benefits make Headless CMS a natural fit for service-based architectures, where modularity, scalability, and flexibility are key. A Headless CMS can be deployed as a microservice within a larger ecosystem, interacting with other services via APIs. The CMS handles database, component, and content creation. This allows developers to concentrate on the application's frontend, resulting in quicker and more effective development.

Currently, there are a lot of Headless CMS platforms in the market. While there are benefits to using Headless CMS, there are some downsides too, depending on the Headless CMS platforms developers choose. For example, if the Headless CMS is cloud-based, the cost can get quite high too. Some Headless CMS platforms also have security concerns when their implementation is lacking. Some are also vendors locked in, making the possibility of migration in the future require significant work. Therefore, it's important to choose the Headless CMS that has handled those downsides and is suitable for the developers' and company's needs.

### 3. METHODOLOGY

Possessing a thorough comprehension of the user's responsibilities and workflows was crucial when developing an application that prioritizes the user as the primary focus. Therefore, before developing the website, we conducted a comprehensive interview with the stakeholders, including the UI designers, to gather the requirements [18]. Discussions with stakeholders established that the platform will serve Job Seekers, Mentors, and Business Companies, each with distinct roles and functions. We also identify the main actions users can take on the website, which are:

- a. Seeing the list and details of Events, Mentors, and Virtual Work Experiences.
- b. Registering and paying for Events, Mentorships, and Virtual Work Experiences
- c. Manage their accounts, profiles, and registrations

After gathering the requirements we needed, we continued designing the architecture by decomposing the system into services. We designed the services based on a domain-driven design principle where services are designed around business capabilities rather than technical layers [19]. To ensure service granularity, where services are small enough to be manageable but large enough to provide meaningful functionality, the services we design handle each defined activity. The primary services in the new architecture are:

- a. Catalog Service: Provides a list and details of each Event, Mentor, and Virtual Work Experience.
- b. Enrollment Service: Service that handles Event, Mentorship, and Virtual Experience registration.
- c. Management Service: Service that handles each product's Create, Update, and Delete methods, along with account and content management.

By separating the Catalog and Enrollment services, some core functionalities of the website can continue to be operational even when one service experiences failure. For example, users can still view program lists and enroll even if the Management service fails. As the organization grows and acquires additional resources, the Management Service can be segmented into smaller units. Furthermore, alongside these three services, various third-party services would be integrated as well. These include a Billing Service using Midtrans and a Scheduling Service using Calendly.

Following the initial design's completion, the next step involved implementing technical architecture. This architecture adopted REST API to communicate between components. Fig 1. provides a comprehensive and complete overview of what each service handles. Fig 2. illustrates the complete architecture, including the technologies used.

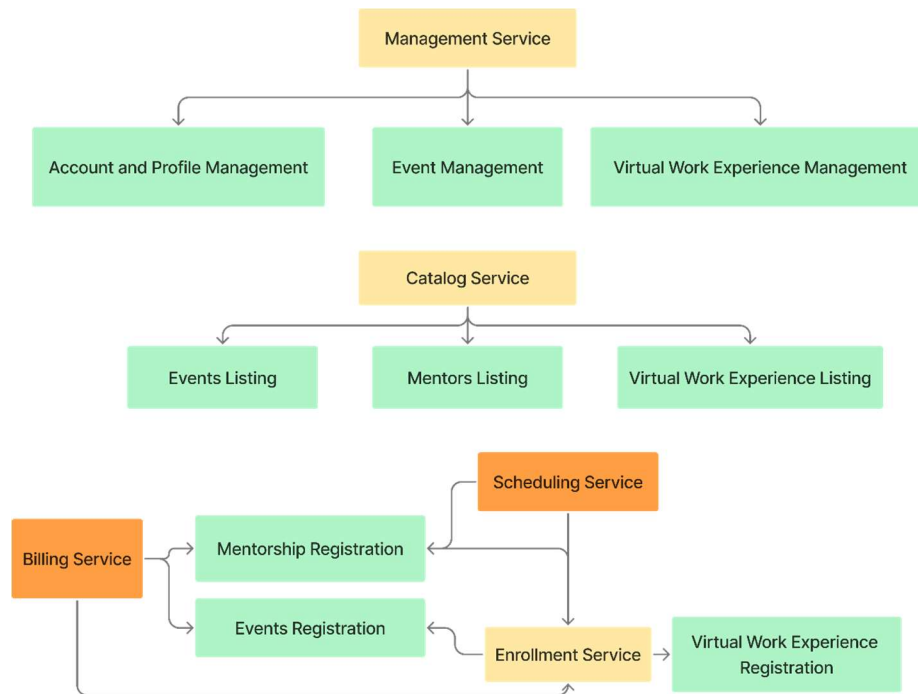


Fig 1: The Responsibilities of The Services

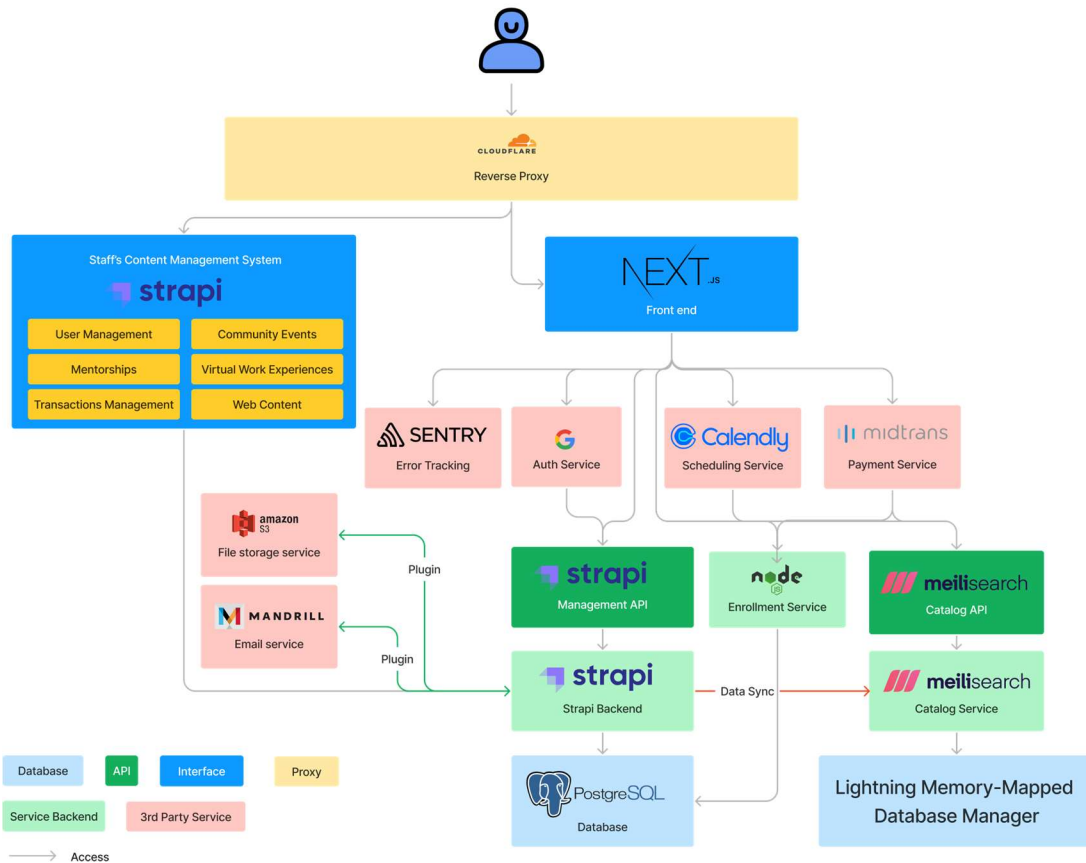


Fig 2: The Technical Design of Service-Based Architecture



To evaluate this architecture, as the website is still in development, we focused on the aspect of UX that can be affected by the architecture, which is speed [20]. We assessed the services and RESTful APIs we developed by conducting an API performance test with Apache JMeter and a website performance test with Google Lighthouse. Apache JMeter is free and open-source software designed for conducting functional behavioral testing and

measuring performance. It may analyze and assess the performance of web applications and services [21]. JMeter performs tests by sending HTTP requests to the server being tested and afterwards measuring the response time. Response time is calculated as the time from when the request is sent until the complete response is received. The steps to do an API call simulation can be seen in Fig 3.



Fig 3: Steps of Testing API Endpoints using JMeter

To test an API endpoint, first we need to define a thread group, which comprises the parameters that will affect the simulation. Here are the following parameters in a thread group:

- Number of Threads (Users): The number of virtual users that will simulate the API call. This determines the number of users who will simultaneously execute the call.
- Ramp-up Period: The time JMeter will take to start the threads. This controls how quickly the users are added to the test. A gradual ramp-up helps simulate real-world

scenarios where users do not all arrive at the same time.

- Loop count: Specifies how many times the simulation will be executed by each thread.

To determine how many executions are done in one test, we need to see the parameters. For example, when we set 20 virtual users with a ramp-up period of 2 seconds and a loop count of two, the total number of executions will be 40 executions. Each Rest APIs will have different parameters and will be specified in Evaluation. An example of how a thread group is established can be seen in Fig 4.

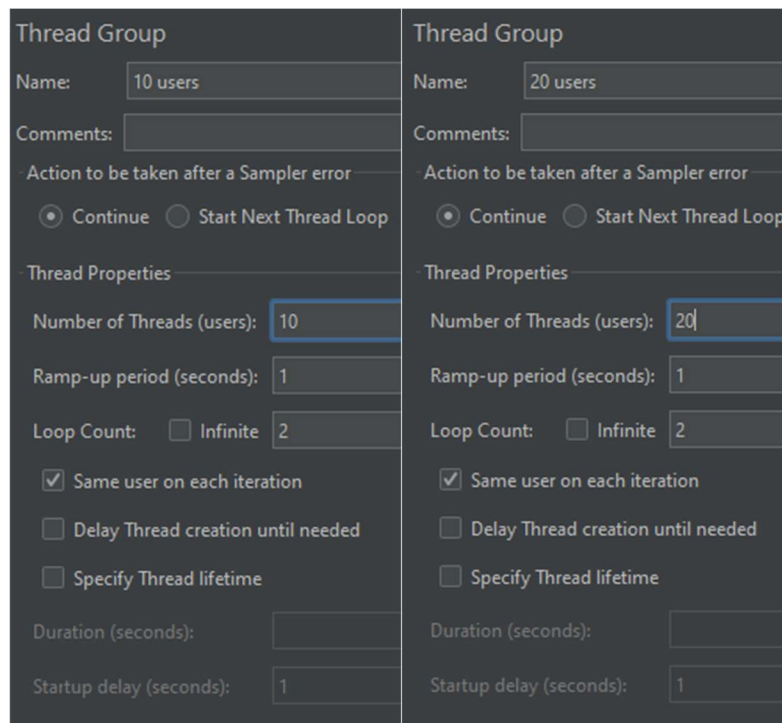


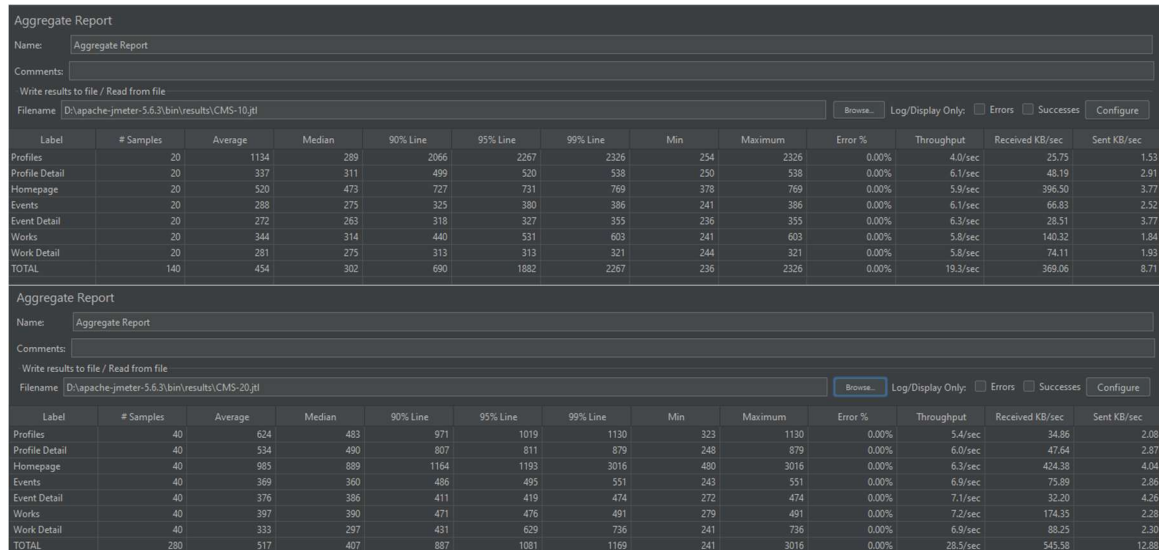
Fig 4: Example of Thread Groups in JMeter

HTTP Samplers are the setup for API calls, such as pinpointing the endpoints and determining the headers, while listeners are the reports needed. The final report can be accessed through an Aggregate Report. The summary and aggregate reports provide the following metrics:

- a. Throughput: The number of requests per minute the server has processed. Higher throughput indicates better API performance.

- b. Average: the total time is divided by the number of requests sent to the server.
- c. Error rate: Number of failed requests.
- d. Median: the number representing the time, where half of the server response time is lower than this number and half is higher.
- e. Deviation shows how much the server response time varies.

An example of aggregate reports can be seen in Fig 5.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/sec	Sent KB/sec
Profiles	20	1134	289	2066	2267	2326	254	2326	0.00%	4.0/sec	25.75	1.53
Profile Detail	20	337	311	499	520	538	250	538	0.00%	6.1/sec	48.19	2.91
Homepage	20	520	473	727	731	769	378	769	0.00%	5.9/sec	396.50	3.77
Events	20	288	275	325	380	386	241	386	0.00%	6.1/sec	66.83	2.52
Event Detail	20	272	263	318	327	355	236	355	0.00%	6.3/sec	28.51	3.77
Works	20	344	314	440	531	603	241	603	0.00%	5.8/sec	140.32	1.84
Work Detail	20	281	275	313	313	321	244	321	0.00%	5.8/sec	74.11	1.93
TOTAL	140	454	302	690	1082	2287	236	2326	0.00%	19.3/sec	369.06	8.71

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/sec	Sent KB/sec
Profiles	40	634	483	971	1019	1130	323	1130	0.00%	5.4/sec	34.86	2.08
Profile Detail	40	534	490	807	811	879	248	879	0.00%	6.0/sec	47.64	2.87
Homepage	40	985	889	1164	1193	3016	480	3016	0.00%	6.3/sec	424.38	4.04
Events	40	369	360	486	495	551	243	551	0.00%	6.9/sec	75.89	2.86
Event Detail	40	376	386	411	419	474	272	474	0.00%	7.1/sec	32.20	4.26
Works	40	397	390	471	476	491	279	491	0.00%	7.2/sec	174.35	2.28
Work Detail	40	333	297	431	629	736	241	736	0.00%	6.9/sec	88.25	2.30
TOTAL	280	517	407	887	1081	1169	241	3016	0.00%	28.5/sec	545.58	12.88

Fig 5: Example of Aggregate Report in JMeter

A response time under 1 second is preferable, while a response time of 1-2 seconds is still acceptable. 2-5 seconds is tolerable for non-critical actions, while over 5 seconds can frustrate for users [22, 23].

After the services were created and tested, we performed website testing to assess the impact of the API on the website's performance. Because the website is server-rendered where API calls are done in server, the quality of the API endpoints are important and affects the website quality. To assess how much the API endpoints affected the website, Google Lighthouse is used to test the website.

Google Lighthouse, sometimes known as Lighthouse, is a free tool designed to evaluate the performance of a website and assist developers in enhancing its performance [24]. Google Lighthouse categorizes numerous factors that impact its ranking, including fast First Contentful Paint (FCP), Total Blocking Time (TBT), Speed Index, Largest Contentful Paint, and Cumulative Layout Shift. The scoring criteria have been established by the Google Developers team.

In this case study, we investigated how the generated APIs affected the website load speed,

which can be discerned through FCP. FCP measures the time it takes for a web browser to display the initial part of the Document Object Model (DOM) content after a user visits a webpage. Time to First Byte (TTFB) is one of the factors that influences FCP. It measures the time between when a page request starts and when the first byte of data is received from the server. In the latest versions of Lighthouse, TTFB appears as an audit known as "Initial server response time". Each audit has its own scoring category, which can be seen in Table 1, Table 2, and Table 3. An example of a Lighthouse Report can be seen in Fig 6.

Table 1: Time to First Byte Criteria

Score	Category
Below 0.8s	Good
Between 0.8s to 1.8s	Needs Improvement
Above 1.8s	Poor

Table 2: First Contentful Score Criteria

Score	Category
Below 1.8s	Good

Between 1.8s to 3s	Needs Improvement
Above 3s	Poor

Table 3: Performance Score Criteria

Score	Category
0-49	Poor
50-89	Needs Improvement
90-100	Good

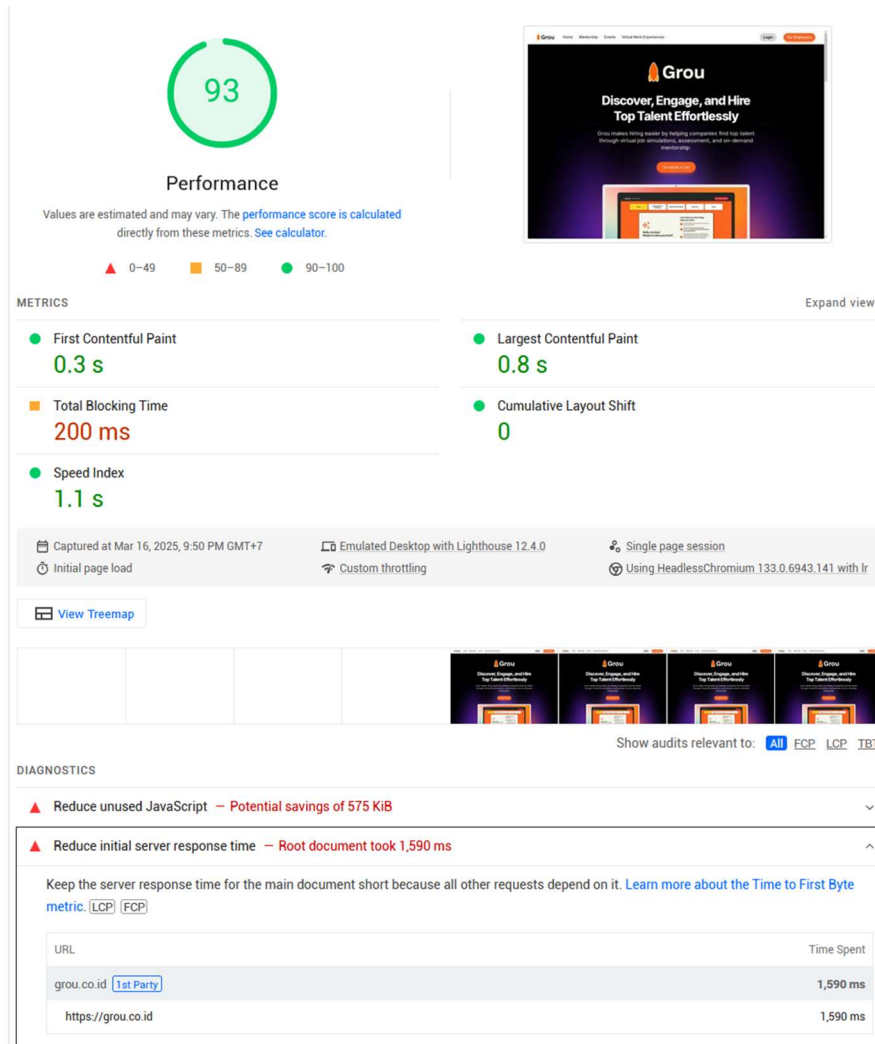


Fig 6: Example of Lighthouse Analysis Result

Variations in web and network technologies can impact the consistency of Lighthouse testing results, causing fluctuations in measurements even when the page content is the same. Lighthouse proposed reducing the impact of external influences by using either localhost or a machine within the same network, along with a dedicated device. In addition, they recommended executing Lighthouse numerous times and using

aggregate metrics such as Mean and Median instead of single tests [25]. Following that advice, we tested the pages on the same computer, using Lighthouse CLI, at various times. To assess the impact of transitioning from Strapi endpoints to services' endpoints on website performance, we conducted an in-depth evaluation of all the relevant endpoints of Strapi, Meilisearch, and Enrollment Service.



## 4. SERVICES DEVELOPMENT

### 4.1 Building the Headless Content Management System as the Management Service

After defining all product components, we started developing the Headless CMS. This CMS is the core of our service-oriented architecture. After researching some Headless CMS platforms, we chose Strapi as our chosen Headless CMS. Based on our research, Strapi has handled the possible downsides of Headless CMS mentioned in the Literature Review. Strapi is an open-source and free, allowing developers to expand and modify the codebase to meet their needs. It is also not locked in by one vendor, making migration to another platform easier. Because it can be self-hosted, developers can choose to deploy the CMS to an infrastructure fitting to their budget. Developers also have full control over security because of this self-hosted nature, on top of already providing security

features like CSRF protection, CORS configuration, and role-based access control.

Strapi provides an extensive dashboard, content-generating capabilities, and a REST API gateway [26]. In addition, Strapi provides authentication management and offers a straightforward way for developers to build plugins, therefore simplifying the integration of third-party services into the CMS. Different to WordPress plugins that extend both backend and frontend functionalities, Strapi plugins focus on backend functionality, such as adding new APIs and services, so it has less impact on the front end side.

Strapi speeds up the development of REST API and content architecture by providing the Content-Type Builder feature. This feature enables developers to create components for each product and quickly generates REST API endpoints for each component. An example of the Content Builder can be seen in Fig 7.

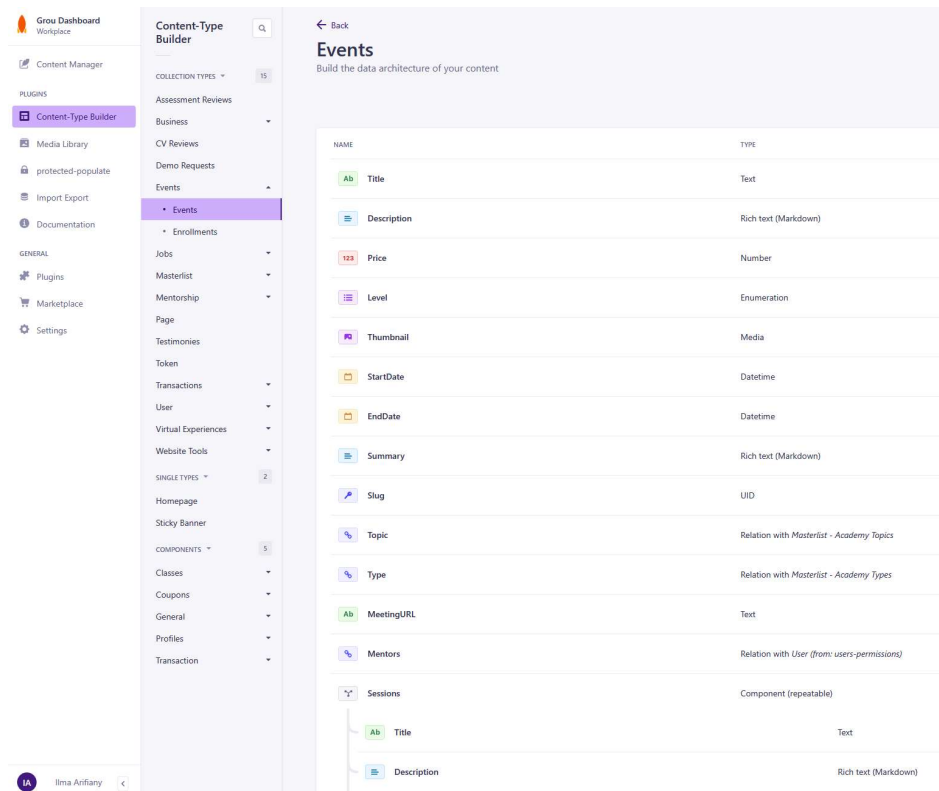


Fig 7: Content-Type Builder in the Company's Strapi CMS

Once the components had been created, developers could define user roles by referring to the user types determined by the product requirements. Developers could control the API access permissions that correspond to each role. Granting API access to a role allows all users assigned to that

role to retrieve and modify data through the specified API endpoint. To ensure extra security, a middleware was implemented in API endpoints that change data. This middleware restricts data modification or deletion to the data owner only,

preventing unauthorized access and manipulation of data by other users with different roles.

During development, we generated 74 API endpoints that would be used for various components of the websites. Out of those, 27 endpoints related to data that would be shown on the website and 8 endpoints related to enrollment that could be migrated to the other two services.

#### 4.2 Creating and Integrating Meilisearch as Catalog Service

For the catalog service, we decided to use Search-as-a-Service. Search-as-a-Service (SaaS) is a solution that handles indexing, querying, and delivering search results to an application. For example, in e-commerce, SaaS is used to display products and enabling filters, sorting, and search. This is suitable for displaying the company's products, such as events, mentors, and virtual work experiences [27]. SaaS is simple to integrate into Strapi via plugin. Meilisearch was chosen to serve as the underlying technology for the catalog service. Meilisearch is an efficient, fast, and feature-rich open-source and free search engine specifically built for seamless integration and implementation in diverse projects. Like Strapi, it is self-hosted, giving developers more freedom. Meilisearch was built using Rust and can be integrated into other programming languages, including Node.js, which

Strapi was built on. Meilisearch is a powerful platform that enables developers to effortlessly incorporate fast and accurate search capabilities into their applications, while also allowing for easy customization and scalability [28]. Its flexibility enables efficient management of huge amounts of data and traffic, making it suitable for both small and large-scale online applications. Meilisearch does not require access to our application's database because it has its own database called Lightning Memory-Mapped Database, which serves as its storage engine.

Meilisearch allows seamless integration into web-based applications for developers by offering a RESTful API. Meilisearch is neatly incorporated into Strapi via a plugin. Via the Strapi dashboard, we exported the specific data we wanted from the database to Meilisearch, arranging them into a searchable and filterable index. The integration of Meilisearch into Strapi can be seen in Fig 8. While Meilisearch boasts a straightforward implementation, optimizing and adapting the data fed into Meilisearch is vital to ensure compatibility with the organization's unique demands. This includes creating catalog names and defining filterable properties for each index. We indexed four specific sorts of data, which are Mentors, Virtual Work Experiences, and Community Events. These types of data correspond to the company's primary products.

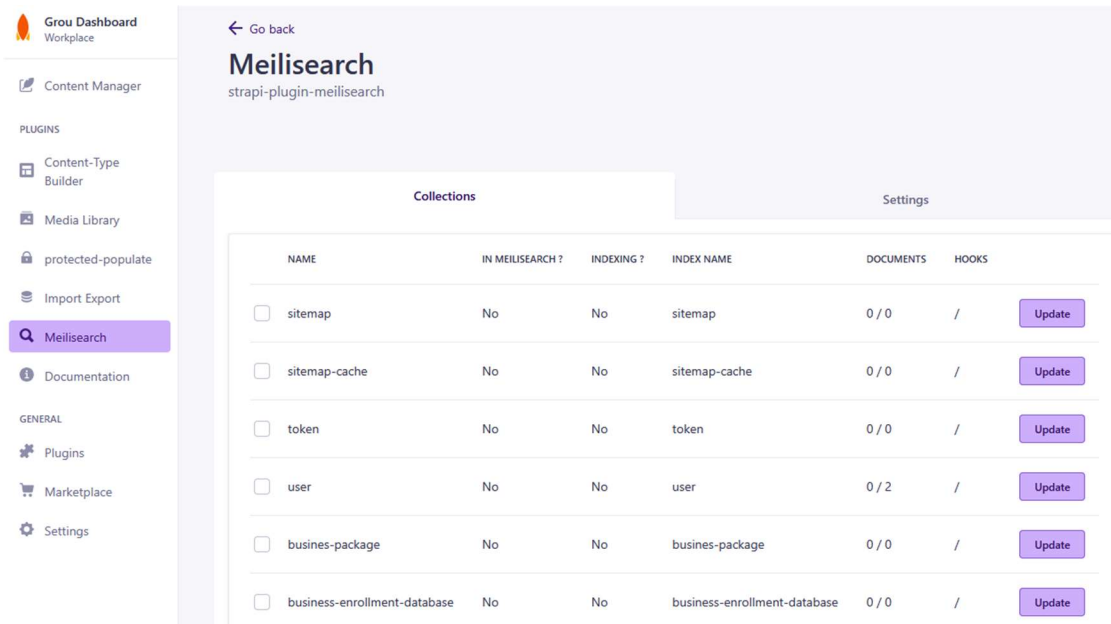


Fig 8: Integration of Meilisearch into Strapi

A catalog service is essential because Strapi might need to access multiple components and databases through a single endpoint, which can

create a significant burden on the application and database. For instance, the homepage requires access to different tables to get featured events, mentors,

and virtual work experiences. By condensing the featured data into a single entry in Meilisearch, the requirement to connect to various tables and access the database is avoided. By integrating Meilisearch, the need to depend on Strapi's API endpoints for displaying each product's catalog is reduced. It reduces the reliance on list APIs like Expertise, Event Types, Event Classes, and others. Previously, these API calls were used for constructing filter lists

on each product's catalog page. For example, filtering mentors based on their expertise and work experiences. By defining these attributes as filterable, Meilisearch automatically aggregates them to be displayed as filters in the instant search engine. An example of the simplification of tables into an index can be seen in Fig 9 while the result of the indexing can be seen in Fig 10.

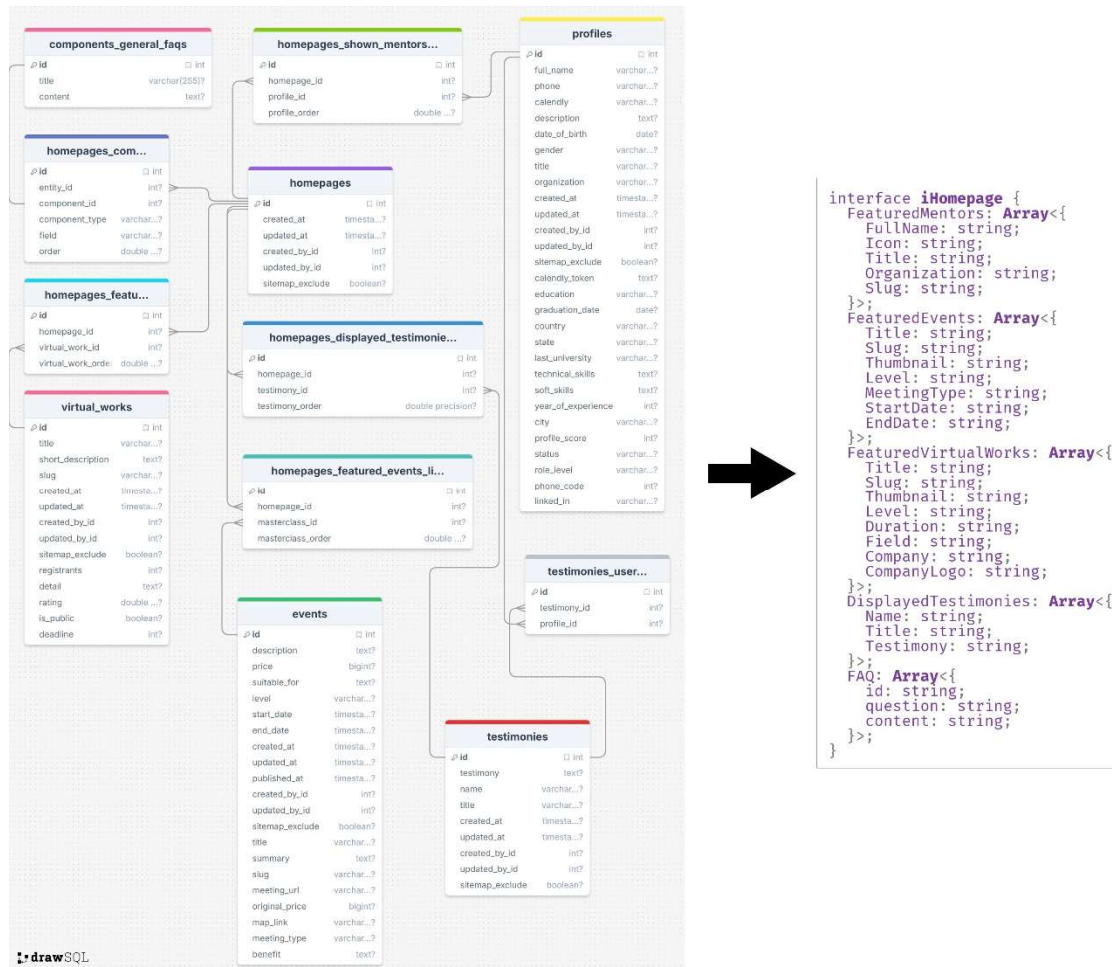
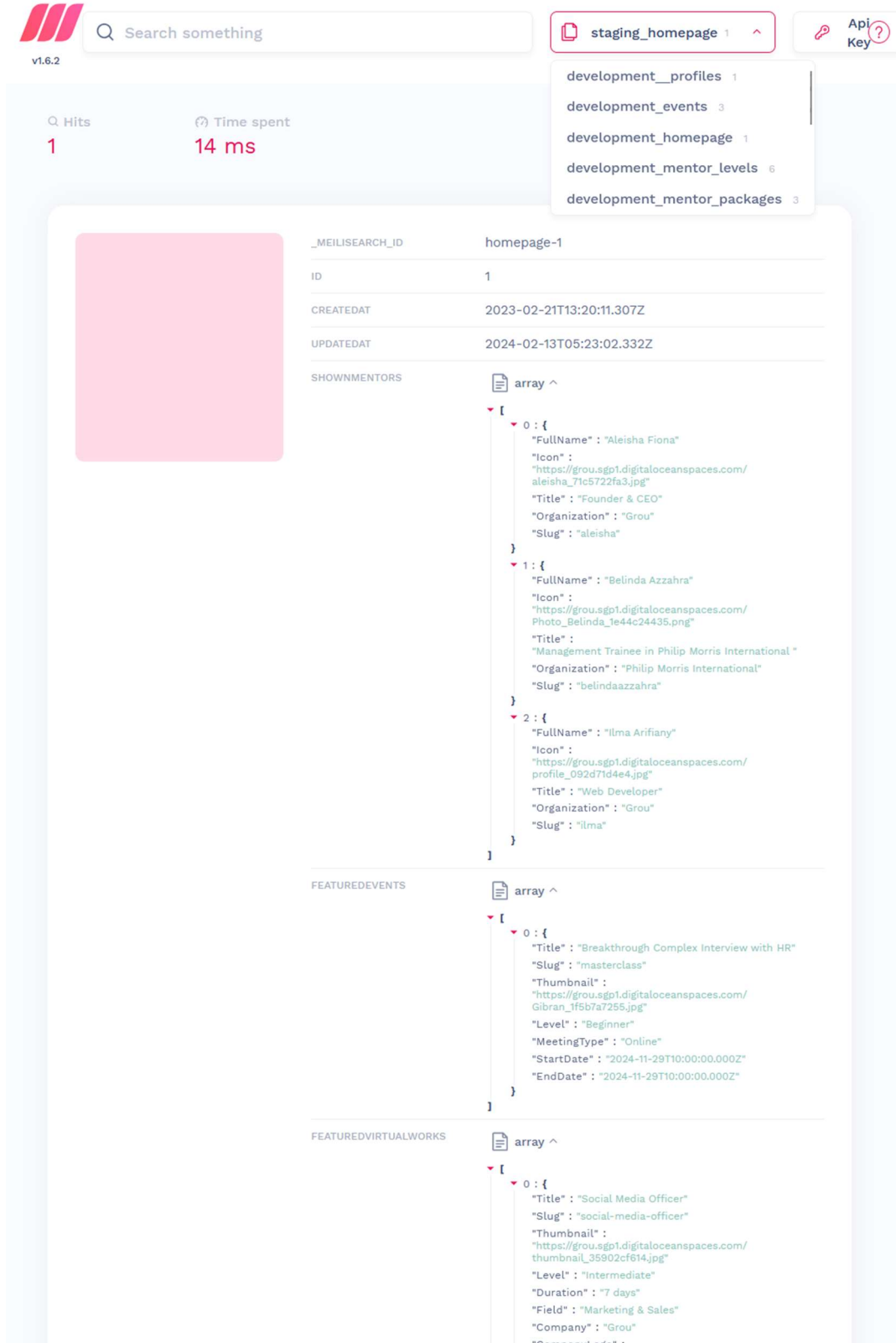


Fig 9: Example of Simplification of Tables to Index Displayed in Typescript Mode



Search something

staging\_homepage 1

development\_profiles 1

development\_events 3

development\_homepage 1

development\_mentor\_levels 6

development\_mentor\_packages 3

Q Hits 1

Time spent 14 ms

MEILISEARCH\_ID homepage-1

ID 1

CREATEDAT 2023-02-21T13:20:11.307Z

UPDATEDAT 2024-02-13T05:23:02.332Z

SHOWNMENTORS

array ^

```
[
  {
    "FullName": "Aleisha Fiona",
    "Icon": "https://grou.sgp1.digitaloceanspaces.com/aleisha_71c5722fa3.jpg",
    "Title": "Founder & CEO",
    "Organization": "Grou",
    "Slug": "aleisha"
  },
  {
    "FullName": "Belinda Azzahra",
    "Icon": "https://grou.sgp1.digitaloceanspaces.com/Photo_Belinda_1e44c24435.png",
    "Title": "Management Trainee in Philip Morris International",
    "Organization": "Philip Morris International",
    "Slug": "belindaazzahra"
  },
  {
    "FullName": "Ilma Arifiyany",
    "Icon": "https://grou.sgp1.digitaloceanspaces.com/profile_092d71d4e4.jpg",
    "Title": "Web Developer",
    "Organization": "Grou",
    "Slug": "ilma"
  }
]
```

FEATUREDEVENTS

array ^

```
[
  {
    "Title": "Breakthrough Complex Interview with HR",
    "Slug": "masterclass",
    "Thumbnail": "https://grou.sgp1.digitaloceanspaces.com/Gibran_1f5b7a7255.jpg",
    "Level": "Beginner",
    "MeetingType": "Online",
    "StartDate": "2024-11-29T10:00:00.000Z",
    "EndDate": "2024-11-29T10:00:00.000Z"
  }
]
```

FEATUREDVIRTUALWORKS

array ^

```
[
  {
    "Title": "Social Media Officer",
    "Slug": "social-media-officer",
    "Thumbnail": "https://grou.sgp1.digitaloceanspaces.com/thumbnaill_35902cf614.jpg",
    "Level": "Intermediate",
    "Duration": "7 days",
    "Field": "Marketing & Sales",
    "Company": "Grou",
    "CompanyLogo": ""
  }
]
```

Fig 10: Imported Data from Strapi to Meilisearch

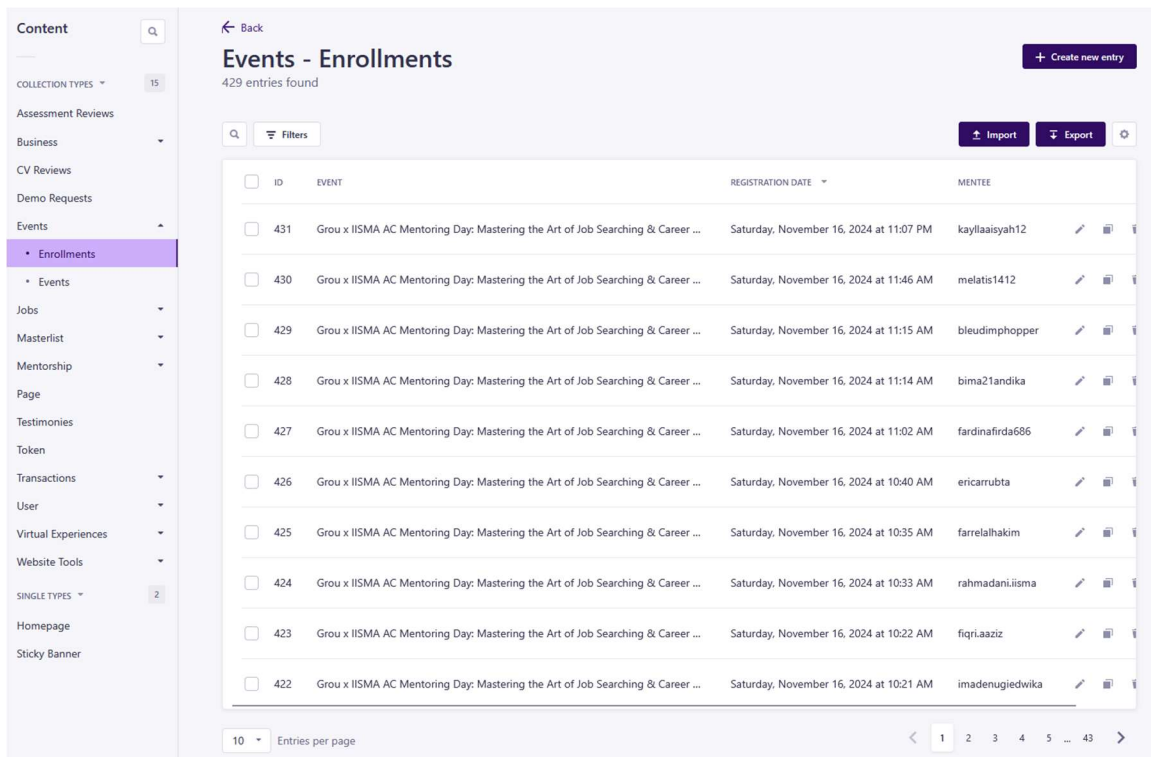
### 4.3 Building and Integrating NestJS as Enrollment Service

The development of a new API backend is necessary for the development of the enrollment service. After researching and comparing the performance of various programming languages, we have chosen NodeJS for its notable performance and employed NestJS as the framework to build the backend architecture [29, 30]. First, we used Strapi's content generation feature to establish the enrollment structures. By utilizing Strapi's automated generation of database schemas, developers are spared from manually creating these structures. Strapi's ability to create tables for inter-

component connections ensures the smooth integration of these entities within the new backend.

The NestJS application used and is built based on the database generated by Strapi, so the two services would share a database. When enrollment is done through NestJS, the data can be viewed on the CMS dashboard since they share the same database, eliminating the need for a new dashboard for enrollment data. This example can be seen in Fig 11.

For security, a new enrollment token is generated each time a user logs in. This ensures only authorized users access the APIs. This service handles event and work registration separately from the management service, lessening the management service's processing load.



ID	EVENT	REGISTRATION DATE	MENTEE
431	Grou x IISMA AC Mentoring Day: Mastering the Art of Job Searching & Career ...	Saturday, November 16, 2024 at 11:07 PM	kaylaaisayah12
430	Grou x IISMA AC Mentoring Day: Mastering the Art of Job Searching & Career ...	Saturday, November 16, 2024 at 11:46 AM	melatis1412
429	Grou x IISMA AC Mentoring Day: Mastering the Art of Job Searching & Career ...	Saturday, November 16, 2024 at 11:15 AM	bleudimphopper
428	Grou x IISMA AC Mentoring Day: Mastering the Art of Job Searching & Career ...	Saturday, November 16, 2024 at 11:14 AM	bima21andika
427	Grou x IISMA AC Mentoring Day: Mastering the Art of Job Searching & Career ...	Saturday, November 16, 2024 at 11:02 AM	fardinafirda686
426	Grou x IISMA AC Mentoring Day: Mastering the Art of Job Searching & Career ...	Saturday, November 16, 2024 at 10:40 AM	ericarrubta
425	Grou x IISMA AC Mentoring Day: Mastering the Art of Job Searching & Career ...	Saturday, November 16, 2024 at 10:35 AM	farrelalhakim
424	Grou x IISMA AC Mentoring Day: Mastering the Art of Job Searching & Career ...	Saturday, November 16, 2024 at 10:33 AM	rahmadaniisima
423	Grou x IISMA AC Mentoring Day: Mastering the Art of Job Searching & Career ...	Saturday, November 16, 2024 at 10:22 AM	fiqri.aaziz
422	Grou x IISMA AC Mentoring Day: Mastering the Art of Job Searching & Career ...	Saturday, November 16, 2024 at 10:21 AM	imadenugiedwika

Fig 11: Enrollments through Enrollment Service in Strapi Dashboard

## 5. EVALUATION

### 5.1 REST API Performance Test Result

For the first test, we performed tests on the RESTful API endpoints for Management Service (Strapi), Catalog Service (Meilisearch), and Enrollment Service (NestJS). We conducted multiple experiments using varying sample sizes. The Catalog Service's first iteration of the test consists of 10 virtual users for each API being evaluated. Successive iterations of the test add 10 virtual users to each test. Every test has a ramp-up

period of 1 second and 2 loops. The variables "Average" and "Median" refer to the mean and median response time (in milliseconds), respectively. "Error" shows the percentage of errors met during the test. "Throughput" stands for the number of requests processed per second. "Deviation" stands for the standard deviation of the response times.

We performed tests on the API endpoints of the Catalog Service that are expected to get many requests because of their common use on pages visited by guests and users. The API endpoints



include the home page, events catalog API, event detail, Virtual Work Experiences catalog, Virtual Work Experiences Detail, Mentors Listing, and Mentor Profile. The following table shows the outcomes of the API performance tests conducted on

Strapi catalog endpoints and Meiliseach's endpoints, compared to one another. For all tests, there was 0 error rate, while the rest of the variables' result can be seen in Table 4.

Table 4: Performance Test Result of Catalog Listing APIs

Total Samples	Average		Median		Throughput		Deviation	
	Strapi	Meiliseach	Strapi	Meiliseach	Strapi	Meiliseach	Strapi	Meiliseach
140	454ms	379ms	302ms	262ms	19.28/s	22.54/s	444.41	420.8
280	517ms	281ms	407ms	258ms	28.5/s	53.45/s	288.21	85.48
420	639ms	267ms	567ms	246ms	38.22/s	67.47/s	252.78	78.58
560	881ms	325ms	744ms	277ms	35.46/s	77.31/s	431.86	147.76

The result showed that Meiliseach achieved better results in load-testing scenarios with a larger number of samples, a shorter response time, and higher throughput. This suggests that there was a greater probability that using the catalog service would cause reduced load times at the front end. This

is because the data delivered by the catalog service has been optimized compared to the data fetched from the Strapi endpoints. For a clearer comparison, Fig 12 compares the numbers between Management Service APIs and Catalog Service APIs.

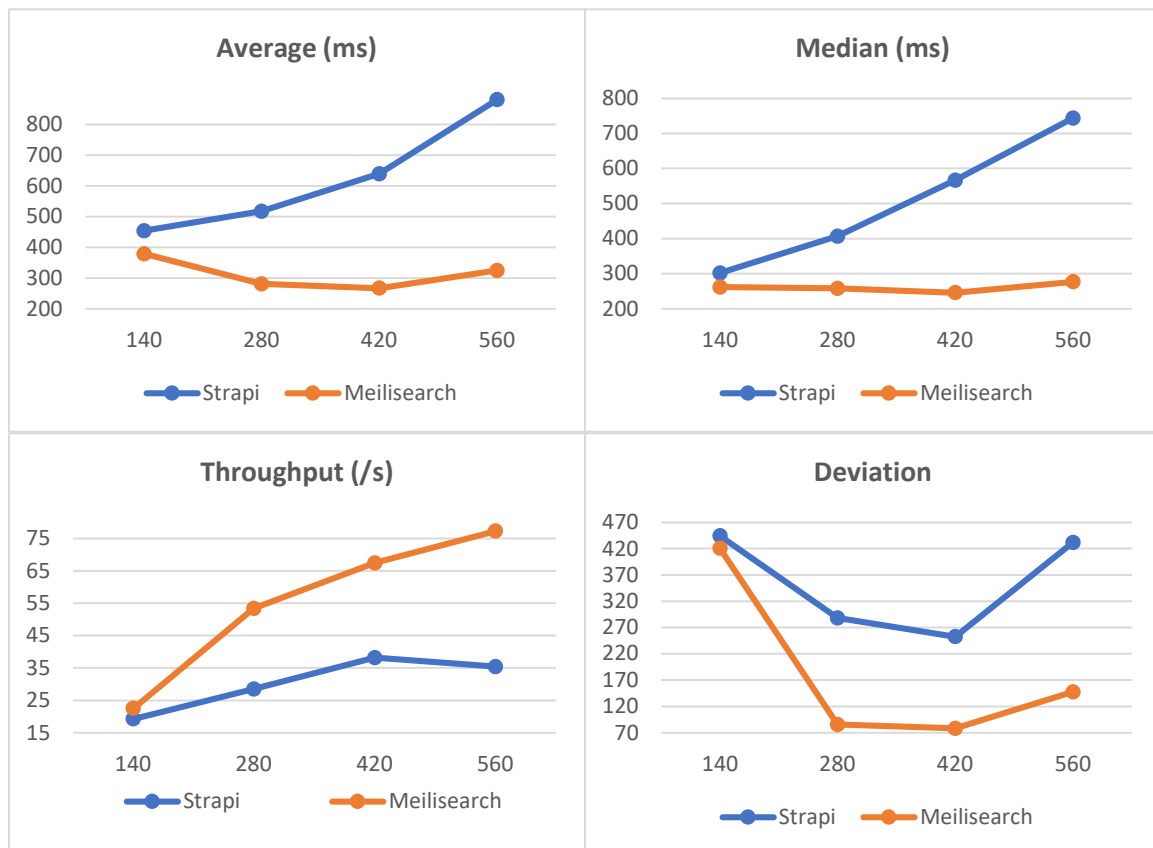


Fig 12: Graphs of Performance Test Result of Catalog Listing APIs

Next, we tested the enrollment APIs for Strapi and the Enrollment Service. For the first

iteration of the test, we performed tests on the APIs to retrieve enrollment data for Mentorship, Events,

and Virtual Work Experiences. The first test comprised 10 virtual users, and with each successive iteration, an additional 10 virtual users were added. The ramp-up period for all tests was set to 1 second, and the loop count was set to 2. The result of the test can be seen in Table 5. The test showed that the Enrollment Service shows superior performance at lower user load but deteriorates as it dealt with a

higher number of users visiting the same API endpoints. This suggests the service needs more optimization to efficiently handle a higher number of requests that occur when the website sees a larger user base. For a clearer comparison, Fig 13 compares the numbers between Management Service APIs and Catalog Service APIs.

Table 5: Performance Test Result of Enrollment Listing APIs

Total Samples	Average		Median		Throughput		Deviation	
	Strapi	NestJS	Strapi	NestJS	Strapi	NestJS	Strapi	NestJS
60	275ms	163ms	264ms	168ms	22.92/s	36.1/s	73.91	66.62
120	283ms	262ms	268ms	289ms	45.75/s	46.35/s	66.6	85.66
180	267ms	474ms	243ms	537ms	76.63/s	44.68/s	124.02	158.37
240	258ms	672ms	272ms	751ms	97.72/s	46.17/s	80.89	183.1

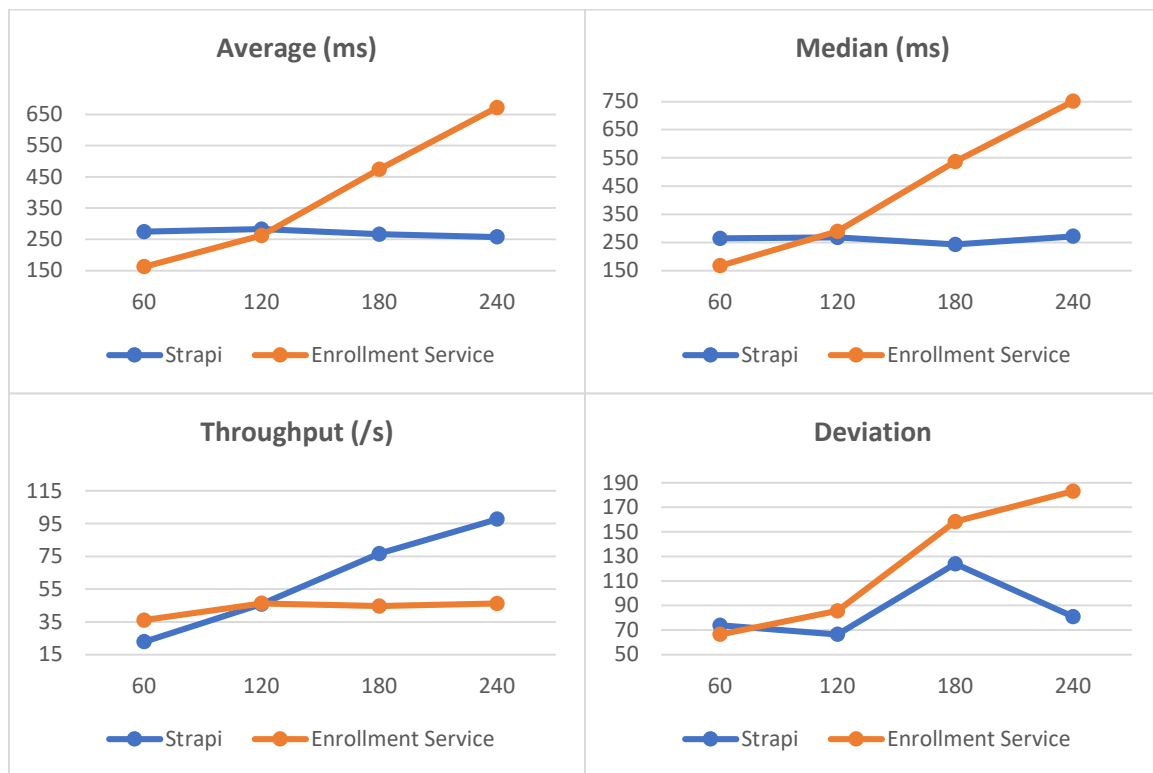


Fig 13: Graphs of Performance Test Result of Enrollment Listing APIs

The next test was performed for API Endpoints used for creating enrollments. The testing process involved two API endpoints: the transaction API, responsible for managing enrollment for Mentorship and Events, and the Virtual Work Experience Enrollment API. The first iteration of the test included 10 virtual users for each API being evaluated. Afterward, with each iteration, an

additional 10 virtual users were added to each test. Every test featured a ramp-up period of 5 seconds and a loop count of 1. The test results for the Strapi Enrollment endpoints and the Enrollment Service's endpoints were similar. The result of the test can be seen in Table 6. For a clearer comparison, Fig 14 compares the numbers between Management Service APIs and Catalog Service APIs.

Table 6: Performance Test Result of Enrollment Creation APIs

Total Samples	Average		Median		Throughput		Deviation	
	Strapi	NestJS	Strapi	NestJS	Strapi	NestJS	Strapi	NestJS
20	221ms	209ms	219ms	165ms	4.05/s	4.02/s	73.55	134.43
40	224ms	166ms	181ms	148ms	7.37/s	7.89/s	95.64	109.76
60	161ms	178ms	161ms	151ms	10.99/s	11.18/s	47.09	117.28
80	199ms	148ms	177ms	155ms	14.45/s	14.31/s	78.62	92.78
100	464ms	196ms	176ms	170ms	18.37/s	19.47/s	558.4	138.46

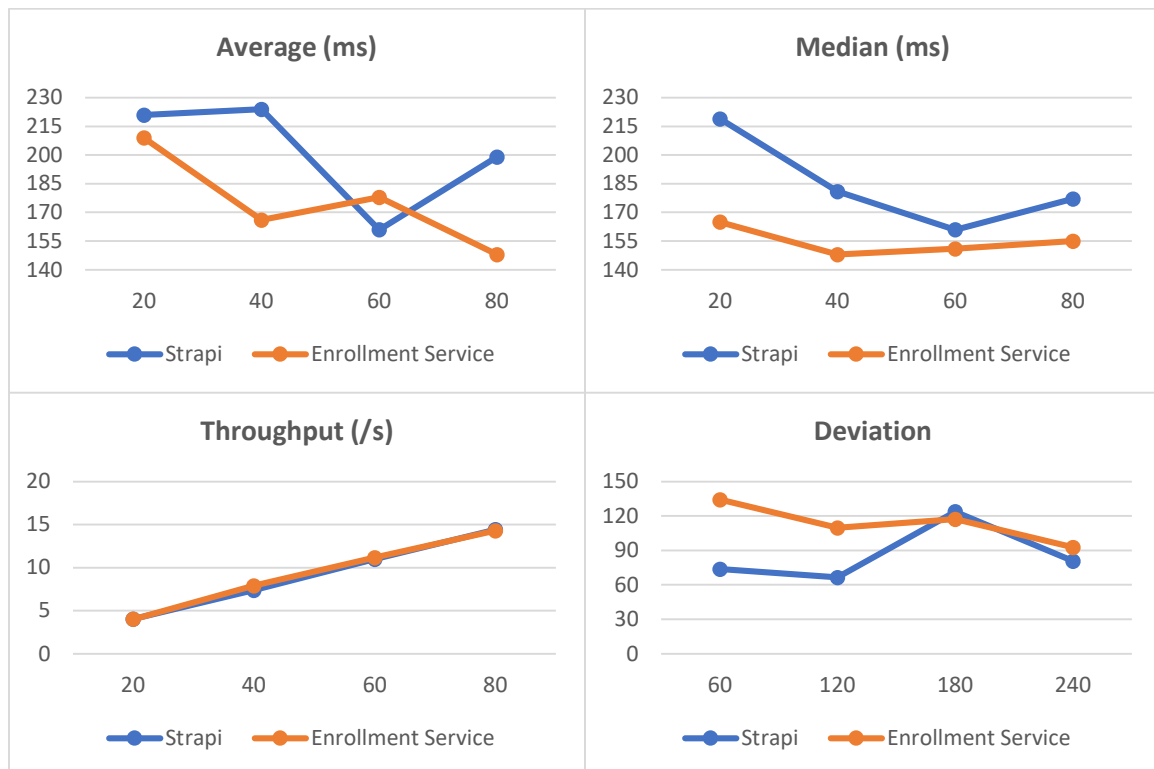


Fig 14: Graphs of Performance Test Result of Enrollment Listing APIs

## 5.2 Website Performance Test Result

A performance test of the developed website has been conducted to assess the impact of utilizing the services on user experience. As per Google Developers' suggestions, we tested the pages by running Lighthouse 10 times in two days. We calculated the mean, median scores, and standard deviation for each category. The pages we evaluated for this study include the pages that will be most often visited by both users and guests. These pages

included the home page (W1), events catalog (W2), event detail (W3), Virtual Work Experiences Catalog (W4), Virtual Work Experiences Detail (W5), Mentors Listing (W6), and Mentor Profile (W7). The first test is performed for Time to First Byte (TTFB), as this audit is the audit most affected by API performance. TTFB affects First Contentful Paint (FCP), so it was expected that TTFB test results would match FCP. Table 7 compares the TTFB test results on pages using 2 different services.

Table 7: Time to First Byte Result

Assessed Page	Average (in seconds)				Median (in seconds)				Deviation			
	Mobile		Desktop		Mobile		Desktop		Mobile		Desktop	
	Strapi	MS	Strapi	MS	Strapi	MS	Strapi	MS	Strapi	MS	Strapi	MS
W1	2.2	1.55	0.68	0.6	1.25	0.76	0.61	0.52	1.65	1.58	0.2	0.17
W2	0.64	0.7	0.56	0.56	0.52	0.74	0.52	0.56	0.29	0.17	0.21	0.11
W3	0.66	0.43	0.53	0.36	0.6	0.41	0.51	0.33	0.15	0.13	0.09	0.08
W4	0.49	0.75	0.5	0.38	0.48	0.46	0.49	0.39	0.12	0.7	0.09	0.08
W5	0.55	0.53	0.56	0.6	0.51	0.52	0.49	0.51	0.11	0.1	0.17	0.41
W6	0.48	0.63	0.44	0.31	0.44	0.48	0.42	0.3	0.12	0.34	0.07	0.05
W7	0.6	0.57	0.51	0.45	0.53	0.54	0.49	0.44	0.17	0.16	0.1	0.09

A clearer comparison between each point can be seen in Fig 15. Based on the categorization described in Table 1, Table 2, and Table 3, scores are divided into three criteria: red (poor), yellow (needs

improvement), and green (good/meets standards). This color assignment also applies to the figures following the next figure.

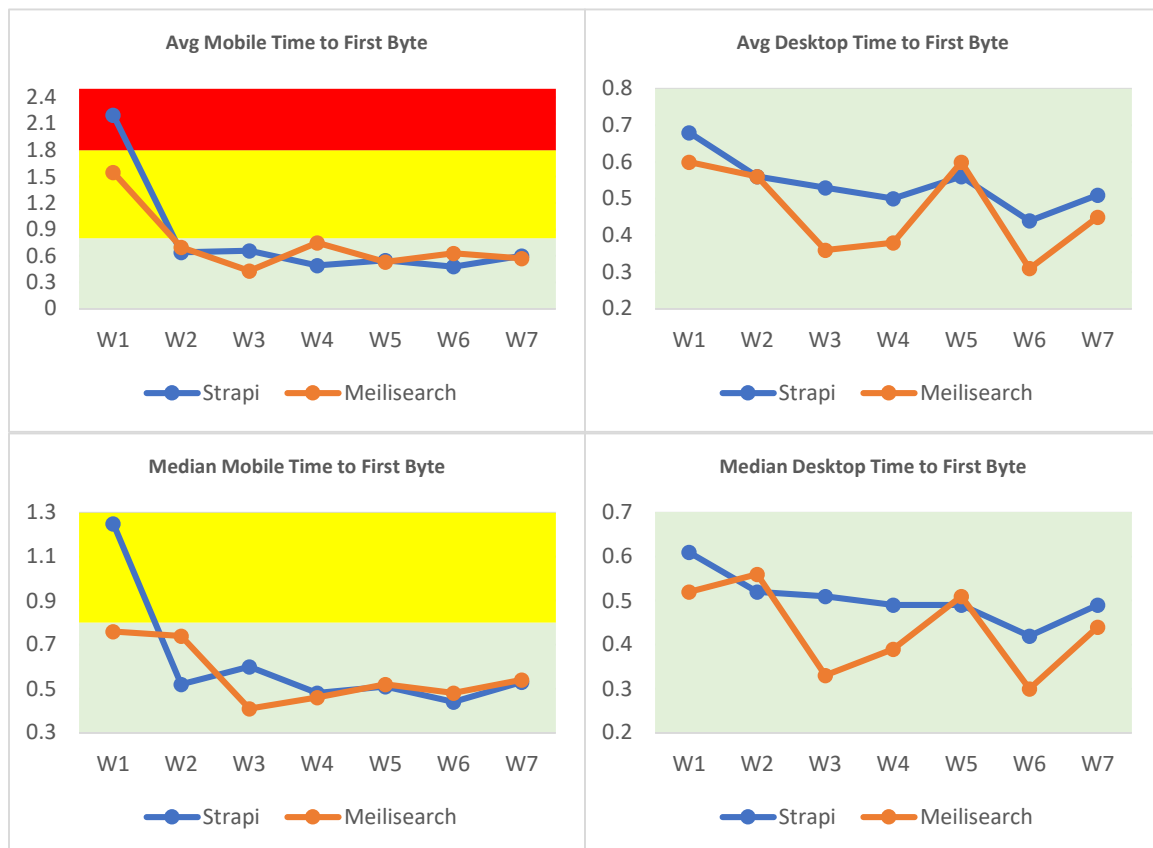


Fig 15: Comparison of Time to First Byte Time Result

The following test measured First Contentful Paint. Since FCP is considerably affected by TTFB, which was part of FCT criteria, the outcome mirrored the TTFB test outcome. The test results for FCP can be seen in Table 8. Similar to the

previous table, this table also compares the performance of the same page but using different services. Meanwhile, for the comparison of FCP results between pages using Strapi and Meilisearch, it can be seen in Fig 9.

Table 8: First Contentful Paint Result (MS = Meilisearch)

Assessed Page	Average (in seconds)				Median (in seconds)				Deviation			
	Mobile		Desktop		Mobile		Desktop		Mobile		Desktop	
	Strapi	MS	Strapi	MS	Strapi	MS	Strapi	MS	Strapi	MS	Strapi	MS
W1	3.1	3.17	1.35	1.46	3.05	3.1	1.3	1.25	0.42	0.33	0.21	0.37
W2	2.51	2.33	1.16	1.06	2.45	2.4	1.1	1.1	0.27	0.21	0.27	0.13
W3	2.34	2.3	1.29	1.06	2.4	2.3	1.15	1	0.39	0.29	0.35	0.14
W4	2.49	2.38	1.25	1.31	2.45	2.35	1.2	1.25	0.31	0.19	0.36	0.29
W5	2.59	2.48	1.12	1.14	2.5	2.35	1	1.05	0.24	0.31	0.2	0.3
W6	2.23	2.22	1.27	1.01	2.15	2.2	1.3	1	0.28	0.1	0.2	0.11
W7	2.57	2.3	1.23	1.06	2.4	2.25	1.15	1.1	0.37	0.25	0.33	0.11

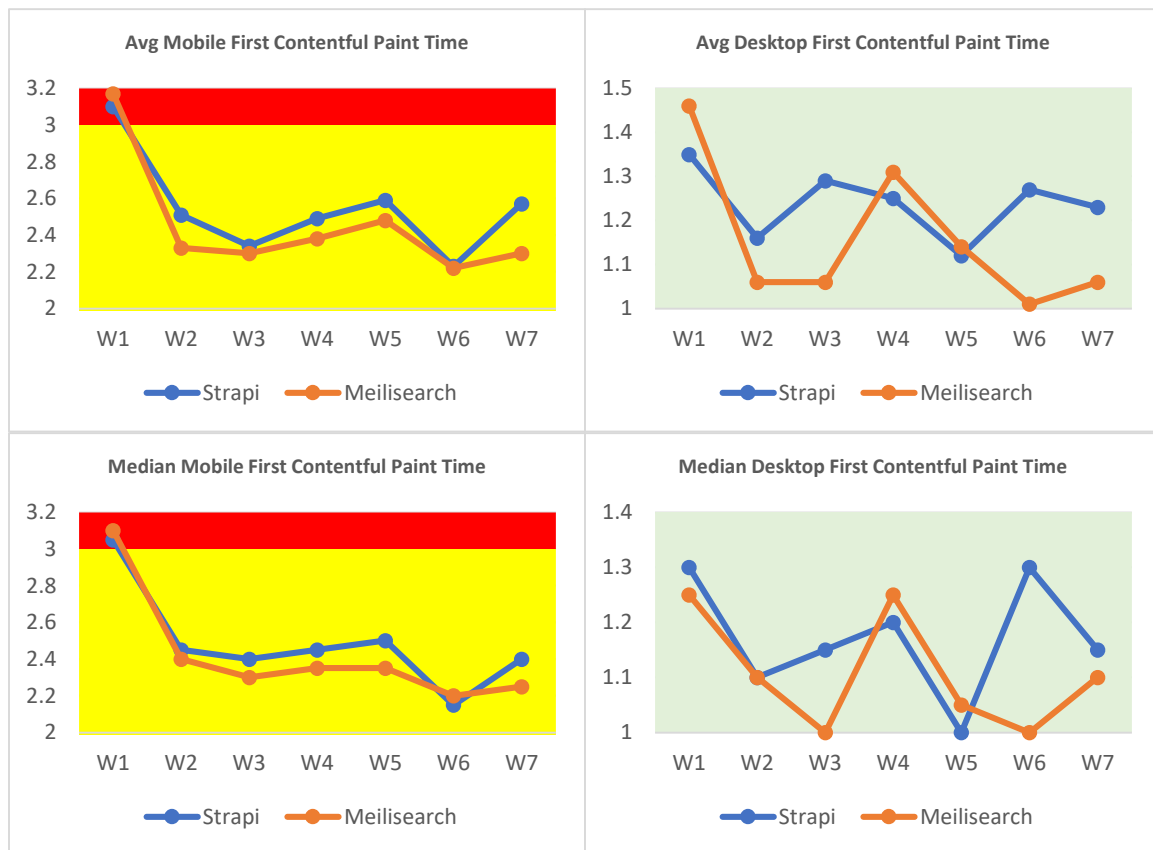


Fig 16: Comparison of First Contentful Paint Time Result

Lighthouse provides a final performance score for each page assessed. These scores can be seen in Table 9, while the clearer comparison can be seen in Fig 17. On mobile, the only poor score was for the home page using Strapi; while using Meilisearch, the score improved to only "need

improvement". This makes sense, given the amount of data required on the home page. It is important to note that FCP only contributes 10% of the total score as it also has other audits unrelated to services to be calculated into the final score.



Table 9: Performance Score Result (MS = Meilisearch)

Assessed Page	Average (in seconds)				Median (in seconds)				Deviation			
	Mobile		Desktop		Mobile		Desktop		Mobile		Desktop	
	Strapi	MS	Strapi	MS	Strapi	MS	Strapi	MS	Strapi	MS	Strapi	MS
W1	70.9	72.6	80.8	80	73.5	73.5	81	83.5	5.99	4.27	4.45	7.82
W2	79.4	82.6	84.7	87.8	80.5	84	85	89	7.77	4.94	7.25	4.33
W3	74.9	82.2	82.8	91.6	73.5	82	86.5	92.5	7.08	4.24	9.35	3.53
W4	77.7	77	84.6	83.6	79	78.5	83.5	84.5	7.14	5.93	7.94	6.2
W5	74.8	80.8	82.2	86.4	75	82	82.2	87.5	5.91	5.53	9.03	6.17
W6	87.1	85.4	85.5	90.5	89	87.5	84	93	6.14	5.5	5.5	4.27
W7	80.2	79.8	88.3	90.5	82	81.5	89.5	91	5.27	4.53	4.73	3.8



Fig 17: Comparison of Performance Score Result

Our test result indicates that pages using the Catalog Service demonstrate improved performance compared to pages using Strapi endpoints. The transition from only using Strapi to Meilisearch does not affect performance; instead, it improves the performance of specific web pages.

The Enrollment Service requires improvement in managing a larger user load. Therefore, Catalog Service can be integrated into

Service-Based Architecture, while Enrollment Service still needs to be worked on in the future to be more optimal.

## 6. CONCLUSION AND FUTURE WORK

Developing a minimum viable product for a startup with few resources presents difficulties, particularly when the company wants to focus on scalability and a positive user experience. This case

study showed that using a headless content management system offers an effective first step into creating a service-based architecture before expanding the architecture into a full SOA or Microservices. Most of previous research is mostly concentrated on creating services without considering using an existing solution and developing it into a service that can be used. This case study gives a possible ready-to-use and tested solution to start-up companies wanting to adopt a service-based approach but not having enough resources to create it. From this case study, we provide a guide on using Headless CMS and proving that the use of it will not regress the quality of the service-based architecture.

Employing Headless CMS as a service shows that development can be hastened while conforming to industry standards set by the headless content management system. The CMS's features, like database creation and easy component development, enable a small team of developers to turn it into a service in a few days, depending on the application's complexity. New product components are developed in under a day, freeing developers to focus on frontend optimization and user experience.

As the start-up expands, developing additional services using the CMS data structure becomes feasible, because developers designed the structure based on the consistent structure of the CMS. If the CMS allows developers to create plugins like Strapi, integrating services can be accomplished with convenience. In this case study, we seamlessly integrate Meilisearch as the catalog service and NestJS as the enrollment service using plugins.

In our architecture, each service can be deployed independently, with no impact on the other services. Proper implementation can enhance performance if the services are created with optimization and performance as the main priorities. In this study, the catalog service we developed outperformed the first Headless CMS API. However, the enrollment service performs at the same level as the initial CMS API, although it requires more enhancements to handle a larger number of users. This shows that the creation of services still needs to be optimized. To optimize the REST APIs developers might consider tools such as Load Balancers to assist services in handling many website visitors and optimize caching in the REST APIs.

However, even when the backend services have been optimized, website speed also depends on user connection, so further investigation into the overall user experience is required. Not only that, but

overall User Experience is also affected by the User Interface. Further investigation can be done by doing surveys with the real users of the website.

For consideration for future research and development, Headless CMS usually does not provide ways to connect to more than a pair of database servers. If a service shares a component with the CMS, a modification to that component necessitates corresponding updates to both services. The existing system uses a shared database server for two services, which heightens the risk of a single point of failure that might stop these services. Future endeavors can investigate the feasibility of employing multiple database servers within the services to alleviate the workload on the database server. To achieve proper microservice architecture in the future, developers must perform table and data migration once they use multiple database server styles.

## 7. AUTHORS' CONTRIBUTIONS

Ilma Arifiany was responsible for the conceptualization and formulation of the case study's objectives, the design of the methodology and architecture, the development of the website's services and frontend, the evaluation and analysis of the results, and the drafting of the research manuscript. Gede Putra Kusuma provided supervision and mentorship throughout the research process, reviewed the work, and offered critical feedback on both the research and the manuscript.

## REFERENCES:

- [1] Tapia F, Mora MÁ, Fuertes W, et al. From Monolithic Systems to Microservices: A Comparative Study of Performance. *Applied Sciences* 2020; 10: 5797.
- [2] Richards M. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, Inc., 2016.
- [3] Newman S. *Monolith to Microservices*. O'Reilly Media, Inc., 2019.
- [4] Paternoster N, Giardino C, Unterkalmsteiner M, et al. Software development in startup companies: A systematic mapping study. *Inf Softw Technol* 2014; 56: 1200–1218.
- [5] Kalske M, Mäkitalo N, Mikkonen T. Challenges When Moving from Monolith to Microservice Architecture. 2018, pp. 32–47.
- [6] Saad J, Martinelli S, Machado LS, et al. UX work in software startups: A thematic analysis of the literature. *Inf Softw Technol* 2021; 140: 106688.
- [7] Fitzgerald A. The Ultimate Guide to WordPress Plugins: 19 Examples & How They

- Work.,  
<https://blog.hubspot.com/website/wordpress-plugins> (2021, accessed 19 January 2024).
- [8] Lofthouse T. Creating a better WordPress Site: WordPress vs Headless WordPress vs Modern Headless CMS. *Skyward Digital*, <https://skyward.digital/blog/better-wordpress-sites-with-headless> (2023, accessed 19 January 2024).
- [9] Melvær K. Headless CMS Explained. *Sanity*, <https://www.sanity.io/headless-cms> (2023, accessed 20 January 2024).
- [10] Mahmood Z. The Promise and Limitations of Service Oriented Architecture. In: *International Journal of Computers*, <https://api.semanticscholar.org/CorpusID:39410584> (2007).
- [11] Tapia F, Mora MÁ, Fuertes W, et al. From Monolithic Systems to Microservices: A Comparative Study of Performance. *Applied Sciences* 2020; 10: 5797.
- [12] Bell M. *Service-Oriented Modeling: Service Analysis, Design, and Architecture*. Wiley & Sons, 2008.
- [13] Baresi Luciano and Garriga M. Microservices: The Evolution and Extinction of Web Services? In: Bucchiarone Antonio and Dragoni N and DS and LP and MM and RV and SA (ed) *Microservices: Science and Engineering*. Cham: Springer International Publishing, pp. 3–28.
- [14] Mazzara M, Bucchiarone A, Dragoni N, et al. Size matters: Microservices research and applications. *Microservices: Science and Engineering* 2020; 29–42.
- [15] Gardon DS. Why headless cms over monolithic cms? *Contenttrain*.
- [16] Heslop B. History of content management systems and rise of headless CMS., <https://www.contentstack.com/blog/all-about-headless/content-management-systems-history-and-headless-cms> (2023, accessed 19 January 2024).
- [17] Sobri NAN, Abas MAH, Yassin AIM, et al. Comparison between Headless CMS and Backend-as-a-Service Products for E-Suripreneur Backend. *Mathematical Statistician and Engineering Applications* 2022; 71: 928–938.
- [18] Keeling M. *Design It! : From Programmer to Software Architect*. 1st ed. Pragmatic Bookshelf, 2017.
- [19] Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.
- [20] Wojciechowski P. Technical Aspects of User Experience. *iRonin*.
- [21] Halili EH. *Apache JMeter*. Packt Publishing, 2008.
- [22] Galletta DF, Henry R, McCoy S, et al. Web Site Delays: How Tolerant are Users? *J Assoc Inf Syst* 2004; 5: 1 – 28.
- [23] Rempel G. Defining standards for web page performance in business applications. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. 2015, pp. 245–252.
- [24] Google Developers. Google Lighthouse, <https://developer.chrome.com/docs/lighthouse/overview> (2016, accessed 22 April 2024).
- [25] Google Developers. Lighthouse Variability, 22/04/2024 <https://developers.google.com/web/tools/lighthouse/variability> (2019, accessed 22 April 2024).
- [26] Gadhavi M. What is Strapi, and Why You Should Use it? *Radix*.
- [27] Srinivasan P. How to Leverage Search as a Service for Enhanced Results. *ClickUp*, <https://clickup.com/blog/search-as-a-service/> (2025, accessed 14 March 2025).
- [28] Chang X. The Analysis of Open Source Search Engines. *Highlights in Science, Engineering and Technology* 2023; 32: 32–42.
- [29] Damarjati YP, Raharjo WS. Performance and Scalability Analysis of Node.js and PHP/Nginx Web Application. *Informatika: Jurnal Teknologi Komputer dan Informatika*; 9. Epub ahead of print 2013. DOI: 10.21460/inf.2013.92.313.
- [30] Chaniotis IK, Kyriakou K-ID, Tselikas ND. Is Node.js a viable option for building modern web applications? A performance evaluation study. *Computing* 2015; 97: 1023–1044.