MICRO-FRONTENDS FOR WEB CONTENT MANAGEMENT SYSTEMS

OBASEKI ETINOSA OSASENAGA

19PCG02027

SEPTEMBER, 2021

MICRO-FRONTENDS FOR WEB CONTENT MANAGEMENT SYSTEMS

BY

OBASEKI ETINOSA OSASENAGA
19PCG02027
B.Sc Computer Science, Benson Idahosa University, Benin City

A DISSERTATION SUBMITTED TO THE SCHOOL OF POSTGRADUATE STUDIES IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF MASTER OF SCIENCE (M.Sc.) DEGREE IN COMPUTER SCIENCE IN THE DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES, COLLEGE OF SCIENCE AND TECHNOLOGY, COVENANT UNIVERSITY, OTA, OGUN STATE.

SEPTEMBER, 2021

ACCEPTANCE

This is to attest that this dissertation was accepted in partial fulfilment of the requirements for the award of Master of Science (M.Sc.) degree in Computer Science in the Department of Computer and Information Science, College of Science and Technology, Covenant University, Ota, Ogun State, Nigeria.

Mr. John A. Philip
(Secretary, School of Postgraduate Studies) Signature and Date

Prof. Akan B. Williams
(Dean, School of Postgraduate Studies) Signature and Date

DECLARATION

I hereby declare that this dissertation entitled **MICRO-FRONTENDS FOR WEB CONTENT MANAGEMENT SYSTEMS** was carried out by **OBASEKI ETINOSA OSASENAGA** with matriculation number **19PCG02027**. The project is centred on an original study in the Department of Computer and Information Sciences, College of Science and Technology, Covenant University, Ota, under the supervision of Dr. A.A. Oni. The concepts of this research project are the results of the research carried out by me. Ideas of other researchers have also been fully recognized.

OBASEKI ETINOSA OSASENAGA

Signature and Date

CERTIFICATION

This is to certify that this dissertation entitled **MICRO-FRONTENDS FOR WEB CONTENT MANAGEMENT SYSTEMS** was carried out by **OBASEKI ETINOSA OSASENAGA** and was supervised by and submitted to the Department of Computer and Information Sciences, College of Science and Technology, Covenant University, Ota, Ogun State.

Dr. Aderonke A. Oni
(Supervisor) Signature and Date

Dr. Oladipupo O. Oladipupo
(Head of Department) Signature and Date

Dr. Victor T. Odumuyiwa
(External Examiner) Signature and Date

Prof. Akan B. Williams
(Dean, School of Postgraduate Studies) Signature and Date

DEDICATION

This dissertation is dedicated to the God who makes things beautiful in his time and to everyone I call family who continued to believe in me long before I could remember to.

ACKNOWLEDGEMENT

I would like to thank my supervisor for her relentless push for quality and improvements. My gratitude to the Head of Department and Faculty of the Computer and Information Sciences Department for their direct and indirect impact on my academic and professional pursuits. The Almighty God for his inspiration and strength to finish this work.

TABLE OF CONTENT

ACCEPTANCE **ii**
DECLARATION **iii**
CERTIFICATION **iv**
DEDICATION **v**
ACKNOWLEDGEMENT **vi**
TABLE OF CONTENT **vii**
LIST OF FIGURES **x**
LIST OF TABLES **xii**
ABSTRACT **xiii**

CHAPTER ONE: **INTRODUCTION** **1**
1.1 Background Information 1
1.2 Statement Of The Problem 3
1.3 Aim & Objectives Of The Study 4
1.4 Research Methodology 4
1.5 Significance Of The Study 5
1.6 Scope Of The Study 5
1.8 Organisation Of The Dissertation 5

CHAPTER TWO: **LITERATURE REVIEW** **7**
2.1 Introduction 7
2.2 Content Management Systems 7
2.2.1 Web Content Lifecycle 8
2.2.2 Web Engineering Method For Content Management 9
2.2.3 Traditional Content Management Systems 12
2.2.4 Headless Content Management Systems 13
2.2.5 Extensibility In Content Management Systems 15
2.3 Architecture Evaluation 16
2.3.1 System Stability 16
2.3.2 Web Performance 18
2.3.3 Complexity Metrics 22
2.4 Micro-Frontends 24
2.5 Related Work 37

CHAPTER THREE: METHODOLOGY **44**
3.1 Introduction 44
3.2 Determine The Evaluation Metrics For Microfrontend Architecture In Literature 45
3.3 Develop Micro-Frontends For An Existing Headless Content Management System 51
3.4 Evaluate The Maintainability Of Microfrontend Architecture With A Content Management System 52
3.5 Adapted Model Architecture 53
3.6 Proposed Model Architecture 54
3.7 System Design And Development 54

CHAPTER FOUR: RESULTS **57**
4.1. Introduction 57
4.2. Multivocal Review Of Existing Literature 57
4.3 Implementation 59
4.4 Architecture Evaluation 74

CHAPTER FIVE: SUMMARY, RECOMMENDATIONS AND CONCLUSION **82**
5.1 Summary 82
5.2 Contribution To Knowledge 82
5.3 Recommendations 82
5.4 Limitations 82
5.5 Conclusion 83

REFERENCES **84**

APPENDIX A: SOURCE CODE **92**

APPENDIX B: APPLICATION PROGRAMMING INTERFACE RESPONSES **109**
WooCommerce All Products Response 109

LIST OF FIGURES

Figure	Title	Page
2.1	Market share of content management systems	8
2.2	Products of a Content Management System	9
2.3	Positioning of CMS-based web applications	10
2.4	Two-Tiered Client-Server Architecture	12
2.5	Architecture of Headless CMS	13
2.6	The differences between traditional web architecture and JAMstack web architecture	14
2.7	Example of system stability measurement	18
2.8	Web page performance metrics relative to the time of rendering	20
2.9	Overview of the micro-frontend approach	25
2.10	Types of rendering	26
2.11	Data Fetching and Server operations with Server requests	27
2.12	Data Fetching and Server operations with AJAX	29
2.13	Web proxy server to redirect HTTP requests per route	32
2.14	Contrasting hard and soft navigation	33
2.15	Shared application shell in the Unified Single Page App approach	36
2.16	Application workflow for ontology-based content management system	39
2.17	System architecture for proposed CMS with micro-frontends	40
2.18	System Management Domain Design Process	41
2.19	Routing and data fetch in micro-frontend based graduate information system	43
3.1	Workflow of the methodology	44
3.2	An overview of the multivocal literature review process as described	47

3.3	Overview of Proposed Architecture	54
3.4	Deployment structure	56
4.1	Temporal analysis of selected academic and grey literature	58
4.2	Product Listing Page	65
4.3	Cart and header	67
4.4	Aggregator dependency graph	75
4.5	Cart dependency graph	76
4.6	Product List dependency graph	77
4.7	Headless frontend dependency graph	78

LIST OF TABLES

Table	Title	Page
3.1	Grey literature quality criteria	49
3.2	Hardware Requirements for WordPress CMS	54
3.3	Software Requirements for WordPress CMS	54
3.4	Hardware Requirements for Micro-frontend layer	55
3.5	Software Requirements for Micro-frontend layer	55
3.6	Hardware Requirements for Headless layer	56
3.7	Software Requirements for Headless layer	57
4.1	Web Performance comparison for micro-frontend and headless layer	71
4.2	Complexity metrics for micro-frontends	72
4.3	Complexity metrics for headless monolith	72
4.4	Expert complexity assessment	74
4.5	Expert maintainability assessment	74
4.6	Expert performance assessment	75

ABSTRACT

Content Management Systems are a fundamental part of the modern world wide web. They are used to create various types of web applications. With the advent of service oriented architecture (SOA), it is commonplace for content management systems to be separate from the presentation layer that eventually displays the content. However, as the complexity of the system grows the frontend may become increasingly hard to maintain and scale. This study aims to apply the micro-frontend pattern to the presentation layer of headless web content management systems in order to provide improved maintainability to the frontend. A multivocal literature review which combines academic literature with grey literature is carried out in this study. The review is to determine the implementation strategies currently being used in research and industry as well as the approach to evaluation of micro-frontend architecture. This work provides a model architecture for applying micro-frontends to general purpose content management systems using WordPress as a case study. The success of the micro-frontend implementation is measured using system stability, web performance and code complexity metrics to compare against a functionally equivalent monolithic implementation. The results of the systematic review show the growing popularity of the micro-frontend approach as well as the different tools and techniques used in implementing the architecture. Client-side rendering and unified single page applications (SPA) are the dominant rendering and composition approaches of micro-frontend used in literature. The evaluation results that micro-frontends perform favourably compared to the headless approach. Micro-frontends had a maintainability index of 75.48 compared to an index of 74.64 for the monolithic version. In all the web performance metrics considered, micro-frontends posted a superior score than the monolithic versions. Micro-frontends did show a significant increase in the complexity of individual modules compared to the equivalent modules in the monolith.

Keywords: *Micro-frontends, Content Management System, Maintainability, Service Oriented Architecture*

CHAPTER ONE

INTRODUCTION

1.1 BACKGROUND INFORMATION

The volume of digital content being produced continues to explode. Many organisations and even individuals suffer from information overload and content chaos. These terms describe the inefficiency that currently exists in creating and consuming information. Within organisations, members of staff need to access documents, pictures, records and other forms of data from different parts of the organisation. These documents may be stored in different locations and systems and may need to maintain several versions in different languages and formats. The problem is also replicated for information that needs to be accessible from outside of the organisation. Additionally, around 80% of this information is unstructured (Ramalingam, 2016).

As part of efforts to solve issues of information overload and content chaos the practice of content management began to evolve. Websites process vast swathes of content to convey information to users. The amount of this information has continued to grow exponentially as the internet itself has continued to evolve. On the web the content management process goes through multiple phases that are collectively described as the web content lifecycle. Web content management systems support this process throughout its lifecycle (Benevolo & Negri, 2007).

Content management systems (CMS) are a web based software package concerned with providing an environment for the creation, editing and management of multimedia web content. The content management system is a key part of the architecture of any enterprise information system. Content management, with the rise of the internet and how indispensable it has become to operations and processes, is a key component of information management and is crucial to fields (Thrivani, Venugopal & Thomas, 2017). A content management system allows updates on web pages to be carried out by non-technical members of an organisation. This reduces the amount of back and forth required to effect changes while also freeing up technical manpower to work on more complex tasks (Ramalingam, 2016).

Web content management systems have evolved through three distinct stages each characterised by differences in how the content interacts with the presentation logic and business logic of the web application. Early on, content was tightly coupled to the presentation logic and required a programmer to edit the markup of the website in order to effect content changes. This naturally created bottlenecks and led to avoidable errors . The rate of change to information published on websites increased especially as web content became more dynamic and interactive. This makes it difficult, especially on larger websites, to continue to edit the markup while making content changes. This led to the second stage of content management evolution.

In the second stage of content management evolution, the content and the presentation layers were separated from the business logic layer. This led to the development of dedicated systems to manage the content and presentation. This was the birth of content management systems on the web. These content management systems, in the style of the prevalent web architecture, featured a tightly coupled content and presentation layer. The CMS generated the markup for the website from the content that had been entered by a non-technical administrator (Barker, 2016).

The third phase of content management systems features the separation of the presentation layer from the content management system entirely. As the world wide web evolved from plain text to support more and more multimedia capabilities, the complexity of the CMSes increased as well. As the design requirements became more complex, it became more difficult to implement these designs from the limits of the content system's templating. There was the rise of single page applications and the practice of offloading more logic to the client side of web applications (Puskaric *et al*., 2019). These practises made it possible for the next wave of CMS Evolution.

This most modern approach allows content to be completely divested of the client it will be rendered on. It also allows the same content to be used for multiple clients. This configuration also allows content based websites to take advantage of the scaling benefits of service oriented architecture (SOA) while being able to cater to different platforms the consumers may be on (Niknejad *et al*., 2020).

However, as the complexity of the underlying site increases, it can get more difficult to manage the code in the presentation layer. In situations with large teams especially, conflicts in stakeholder requirements and the implementation of those requirements begin to occur more frequently. Implementing conflicting requirements may lead to errors or slower development times. At such

points, it may be useful to consider a micro-frontend architecture to help solve some of these problems (Richardson, 2017; Kalske, Mäkitalo & Mikkonen, 2017).

As frontend techniques evolve, more options for creating dynamic web applications are available. A frontend may be developed as a single page application (SPA) or may be rendered on the server with server-side rendering (SSR) or may be a combination of both techniques as in isomorphically rendered applications. This is in addition to the original technique of statically rendered pages. These techniques result in monolithic frontends which become harder to scale and maintain as the application grows. This is especially true for large teams of maintainers (Peltonen, Mezzalira & Taibi, 2021).

Microservices are a variant of the service-oriented architecture that builds applications as a collection of loosely coupled services. It combines complex large applications in a modular way based on small functional blocks that communicate through a collection of language-independent application programming interfaces (APIs). Each functional block focuses on a single responsibility and function, and can be independently developed, tested, and deployed (Chen, 2018).

The micro-frontend architecture enables teams to develop independently, quickly deploy and test individually, helping with continuous integration, continuous deployment, and continuous delivery. Micro-frontends enable splitting of monolithic frontends into independent and smaller micro applications. This solves the frontend monolith problem and offers the benefits in the microservices architecture to the frontend. However, the micro frontends also bring some shortcomings and many companies are still hesitant to adopt Micro-frontends, due to the lack of knowledge concerning their benefits (Yang *et al*., 2019; Peltonen, Mezzalira & Taibi, 2021).

This work aims to explore the use of a micro-frontend architecture in a content management system for web applications with user generated content to solve the problems of complexity, requirement clashes and tight coupling to the content.

1.2 STATEMENT OF THE PROBLEM

Content management systems should provide extensibility and support new functionality. It should be easy to develop this new functionality while being flexible enough to be adapted to various user requirements. Allowing the easy creation of more scalable and powerful extensions or plugins for content management systems leads to more complex web applications being built easily (Laumer, Maier & Weitzel, 2017; George, 2015). On the frontend of content management systems developers are typically locked in to using the templating language of the system (Ang, 2019). In a headless content management system, the templating system may be replaced by one of the programmer's choice leading to monolithic frontends that are difficult to maintain and extend especially on large teams and codebases (Peltonen, Mezzalira & Taibi, 2021). Micro-frontend architecture could improve the developer experience of extending functionality in content management systems. There have so far only been limited attempts to apply the micro-frontend pattern to content management but the technique may prove to solve many challenges of modern content management systems (Wang et al., 2020).

1.3 AIM & OBJECTIVES OF THE STUDY

The aim of the study is to design, implement and evaluate a service layer that orchestrates and connects micro-frontend components to data from a content management system.

Following the aim, the objectives are:

1. To determine the evaluation metrics for micro-frontend architecture in literature.
2. To demonstrate the use of micro-frontends for an existing headless content management system.
3. To evaluate the maintainability and developer experience of micro-frontend architecture with a content management system.

1.4 RESEARCH METHODOLOGY

The study adopted micro-frontends architecture to implement the presentation layer of an existing general purpose Content management system.

A literature review was carried out to discover the evaluation methods used for micro-frontends as described in literature. This literature review was a multivocal literature review because in addition to using peer-reviewed materials it also sourced from grey literature in order to parse the industry view on the subject.

There are numerous systems, both proprietary and open source that can be used for content management. In this research, the most popular general purpose content management system, WordPress (W3Tech, 2021) was used to manage the content and provide a headless application programming interface (API) to access the content.

Micro-frontends responsible for consuming the content were developed using React, Vue and Node. The services layer was implemented in NodeJS, which is a JavaScript runtime, and is responsible for orchestrating, building and eventually rendering the various micro-frontend apps into a single macro-application. This layer is responsible for fetching data and coupling it with the appropriate frontend. NodeJS features such as HTTP requests and the filesystem API were used.

An e-commerce application was built as a case study. The application allowed products to be uploaded and managed in the content management system and pulled into a headless frontend where a user can view and add to a cart.

For the purpose of evaluation, the micro-frontend evaluation was contrasted against a feature-equivalent version of the case study system built as a frontend monolith using React. The maintainability of micro-frontends was evaluated based on system stability, web performance and code complexity evaluation.

1.5 SIGNIFICANCE OF THE STUDY

Software engineering research must be relevant to the industrial challenges of the day (Garousi, Borg & Oivo, 2020). Content management remains an important problem in the context of the world wide web and has seen significant improvement as web technology improves. Content management systems have been an important tool in making it possible for non-technical users to publish and manage content on the web. The proposed system in this study could make it possible to build and manage more complex presentation layers while also maintaining the benefits in developer experience and performance accrued from headless content management systems. This study also provides an evaluation of the micro-frontend architecture contrasting it against the headless pattern of presentation layers.

1.6 SCOPE OF THE STUDY

The scope of this work is limited to the frontend and user facing concerns of the content management system and does not cover such issues as database management, file and media handling or any concerns of the backend of the system.

1.8 ORGANISATION OF THE DISSERTATION

In addition to the introductory chapter, this dissertation is structured as follows: Chapter Two presents an extensive review of literature on content management systems, service oriented architecture and micro-frontends, related works and summary of the chapter. System methodology, design and modelling, architecture are described in Chapter Three. Chapter Four contains a description of the implemented system and testing/evaluation and a discussion of the results. Finally, Chapter Five presents the summary, contribution to knowledge, conclusion and recommendations for future work.

CHAPTER TWO

LITERATURE REVIEW

2.1 INTRODUCTION

In this chapter, the literature that relates to the approaches taken by other researchers in the area of content management systems, service oriented architecture within content management systems and micro-frontends is reviewed.

2.2 CONTENT MANAGEMENT SYSTEMS

A content management system is software that is concerned with the provision of an interface to publish, edit and modify information in a collaborative environment (Vaidya *et al*., 2013). Content management, with the rise of the internet and how indispensable it has become to operations and processes, is a key component of information management and is crucial to fields. A content management system allows updates on web pages to be carried out by non-technical members of an organisation. This reduces the amount of back and forth required to effect changes while also freeing up technical manpower to work on more complex tasks (Quadri, 2011).

Web content management systems are an important aspect of today's world wide web and WordPress, one of the most popular CMSes, accounts for over 40% of all websites. The distribution of content management market share is shown in Figure 2.1.

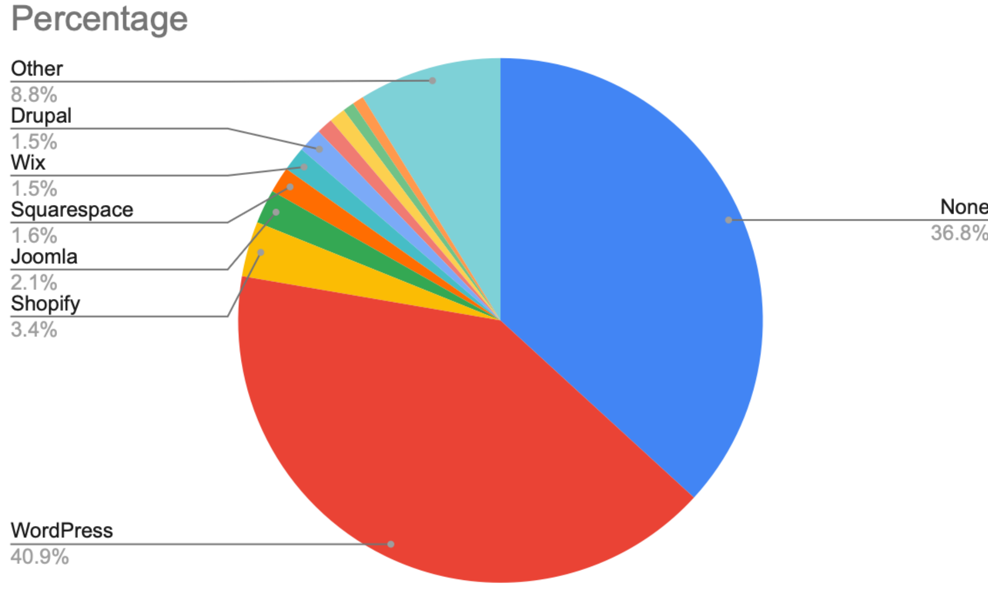


Figure 2.1: Market share of content management systems (W3Tech, 2021)

2.2.1 Web Content Lifecycle

Web content management systems support the multiple phases of the web content lifecycle, that is, the collect-manage-publish lifecycle of content management (Benevolo & Negri, 2007). Content is first collected and then managed before finally being published. Content management systems that support all the required activities are rare and vendors often only offer support for some of the features within the lifecycle (Robertson, 2004). The lifecycle and the activities that make up each stage are illustrated in Figure 2.2.

The collection system is made up of the processes and structures involved in obtaining the content. The processes involved include (Boiko, 2004):

i. Authoring: The process of creating the content.

ii. Acquisition: Means by which required information is obtained.

iii. Conversion: Transforming the acquired content into desired formats, removing unneeded aspects and translating into target markup.

iv. Aggregation: Organizing content based on selected metadata.

v. Collection Services: Any other ancillary services to support the collection process.

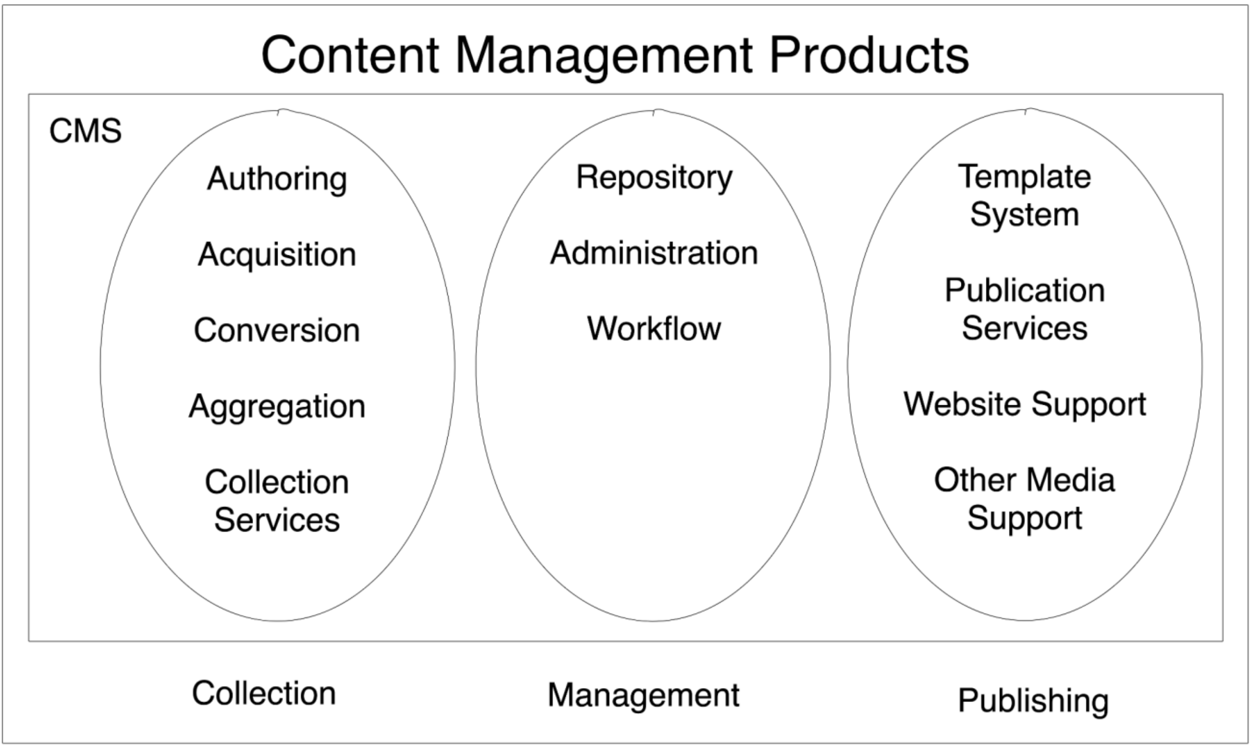


Figure 2.2: Products of a content management system (Benevolo & Negri, 2007)

The management system is responsible for storage and retrieval of the content and metadata collected in the first phase of the content management lifecycle. This layer houses the content in a repository and manages access to it via administration tools as well as other workflow functions (Benevolo & Negri, 2007).

The publishing system accesses content from the repository and presents it in its final form. This may be websites, mobile apps or even print. The publication system features templating engines that conform the platform agnostic content from the management system into formats and structures required by any publications (Boiko, 2004).

2.2.2 Web Engineering Method for Content Management

Content management systems often have special requirements that are not typical in other web engineering processes. Designers often build web applications based on best practises and methods learned from projects in other domains or projects. Such practises work well when designing customizations for completely new components, however, they are not always suitable for designing applications within the typical constraints of content management and indeed other product-specific software. The position of web content management systems in relation to other classes of web applications is explored in Figure 2.3.

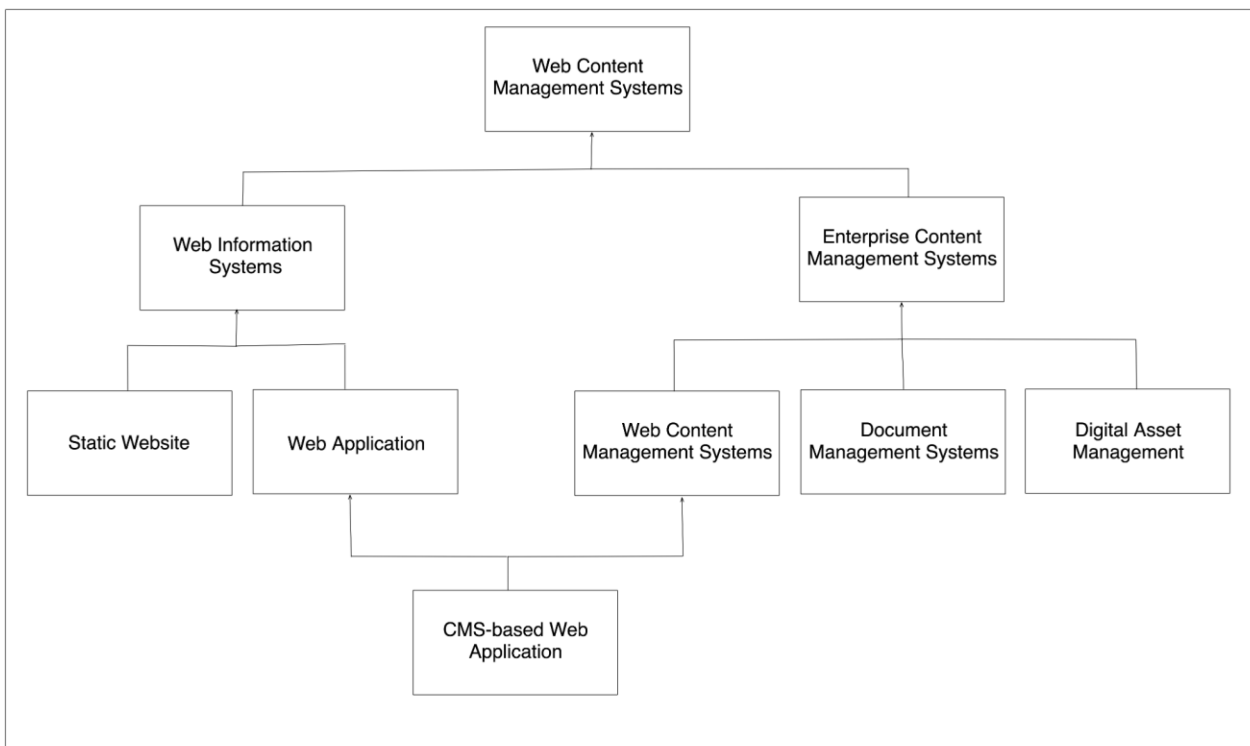


Figure 2.3: Positioning of CMS-based web applications (Souer, 2011).

Some of these specific issues include personalization, a hypertext frontend or presentation layer, caching, as well as the Content Management Software's specific product architecture and how its existing code implements all of these things. Souer *et al.* (2011) proposes the Web Engineering Method (WEM) which is a series of concepts, notations, process descriptions and techniques for the development and implementation of content management system-based web applications, which can be used by both researchers and practitioners. The method is subdivided into six phases.

i. **Acquisition phase**: In the acquisition phase customers requests are documented and outlined through means such as interviews, meetings or written documents. As the requests come together and understanding of the requirements begins to emerge. In traditional web engineering it is difficult to specify these requirements within the context of available features to the CMS while fulfilling the business logic (Souer, 2011). A feature list identifies the key requirements and maps them to the matching or closest resembling standard functionality offered by the CMS of choice. The requirements can then be formalised and understood by all stakeholders.

ii. **Orientation Phase**: The orientation phase of the Web Engineering Method (WEM) features the initialisation of project management structures including the participants at each stage, the target at the stage, the expected results, scope and constraints.

iii. **Definition Phase**: In the definition phase a product vision which includes a description and the aims of the application as well as the scope of the work. The feature list from the acquisition phase is detailed in more depth and an application model is generated. The application model gives a configuration of the CMS-based web application and consists of page navigation models, user interface models, functionality description, application workflow and a content reuse strategy. Nonfunctional requirements including permissions and roles, security and performance are also determined at this stage. This detailed view of the requirements minimises the chance of implementing the wrong requirements and in more complex projects these definitions are revisited frequently.

iv. **Design Phase**: The design phase maps out the requirements to the corresponding implementation strategy. Using the requirements resulting from the definition phase an acceptable architecture is formulated. In most projects the standard CMS features are all that is required and thus the architecture of the system will be the architecture of the CMS. For more tailored requirements extensions and customizations may be fitted to the architecture.

v. **Realisation Phase**: The realisation phase is the stage where actual implementation takes place. The frontend is integrated into the rest of the web application and the functionality is customised to meet user requirements as specified in the definition and design phases.

This stage may be repeated multiple times to hit the required targets while performing tests to ensure completeness.

vi. **Implementation Phase**: A CMS-based web application will typically be immediately deployed to the production environment. However, some users may require a staging environment where they can perform acceptance testing. Once the user is satisfied the project is deployed to the live environment and concluded.

2.2.3 Traditional Content Management Systems

A traditional content management system uses a monolithic architecture coupling the presentation layer and the content layer. This is consistent with the way traditional web applications are built. This monolithic architecture in a client server model is described in Figure 2.4. A monolithic application has all its services developed on a single codebase and whenever a change is made, the other services must also be guaranteed to be working (Villamizar et al, 2015).

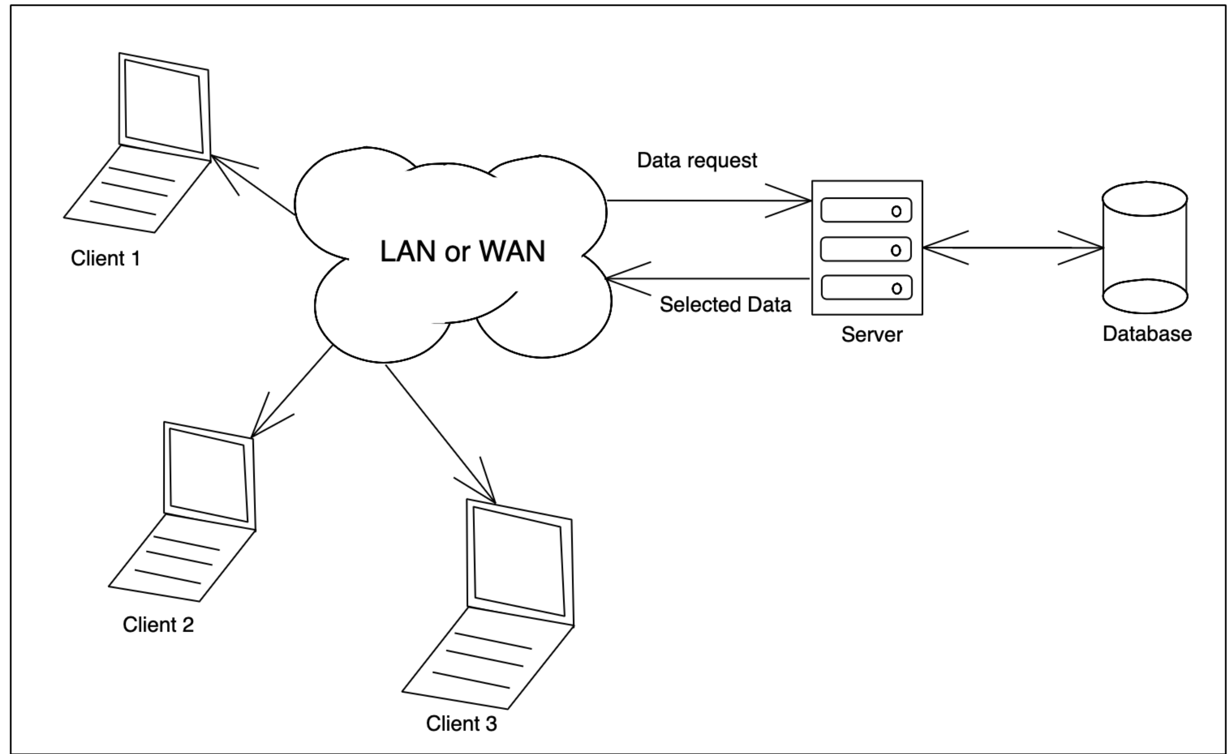


Figure 2.4: Two-Tiered Client-Server Architecture (Kumar, 2019)

2.2.4 Service Oriented Architecture and Content Management Systems

Service Oriented Architecture is an approach that favours building different parts of an application as self contained services with a communication layer exposed between them. In recent times, this approach has gained popularity in building web applications. It separates the server that typically handles application logic from the client that typically handles rendering. This allows content reuse on multiple clients such as serving a website and a mobile application (Niknejad *et al*., 2020). The recent addition of the WordPress REST (Representational State Transfer) API is a step forward in this direction. This API allows the use of WordPress as a headless CMS to build web apps while benefiting from all its core backend functionalities such as collaboration, content and user management (Cabot, 2018).

2.2.4 Headless Content Management Systems

A Headless CMS separates the content from the template that renders it. This separation is illustrated in Figure 2.5. The content is pulled into the template from an application programming interface (API) provided by the CMS (Attardi, 2020). Figure 2.6 shows the information flow for a headless website. The content can be pulled in at runtime or at compile time. The benefit of this approach is that complex user interfaces can be built and then the content can be updated without interacting with the template.

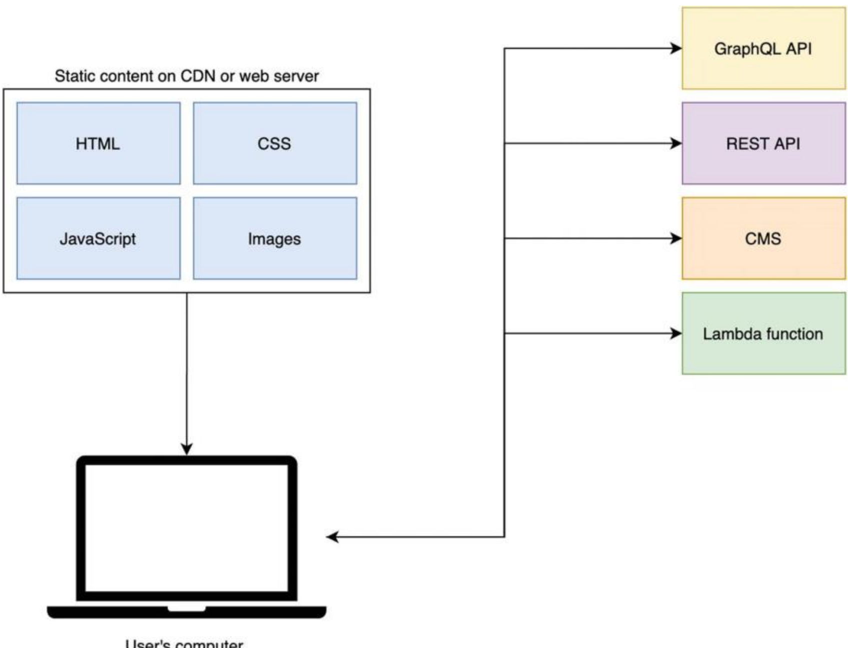


Figure 2.5: Architecture of Headless CMS (Attardi, 2020)

A Headless CMS is often paired with a JAMStack website. JAMstack is an increasingly popular web development philosophy that takes advantage of service oriented architecture to improve the web development process and web page download times. JAMstack is an acronym for the development stack consisting of JavaScript, APIs and Markup. JAMstack is based on using modern tooling and workflows for creating powerful websites that are easy to develop and deploy. It allows websites to achieve the speed and directness of static Hypertext Markup Language (HTML) websites while providing dynamic capabilities and interactivity using JavaScript and APIs. JAMstack websites can be deployed to content delivery networks and served directly to the client without managing any web servers. Thus JAMstack applications are far cheaper than typical server-side applications as the cost of serving static files is negligible and these assets are easily cached. JAMstack architecture emphasises a complete separation of the presentation layer from the backend or business logic layer of the application (Peltonen, Mezzalira & Taibi, 2021). The differences between JAMStack websites and traditional websites is illustrated in Figure 2.6.

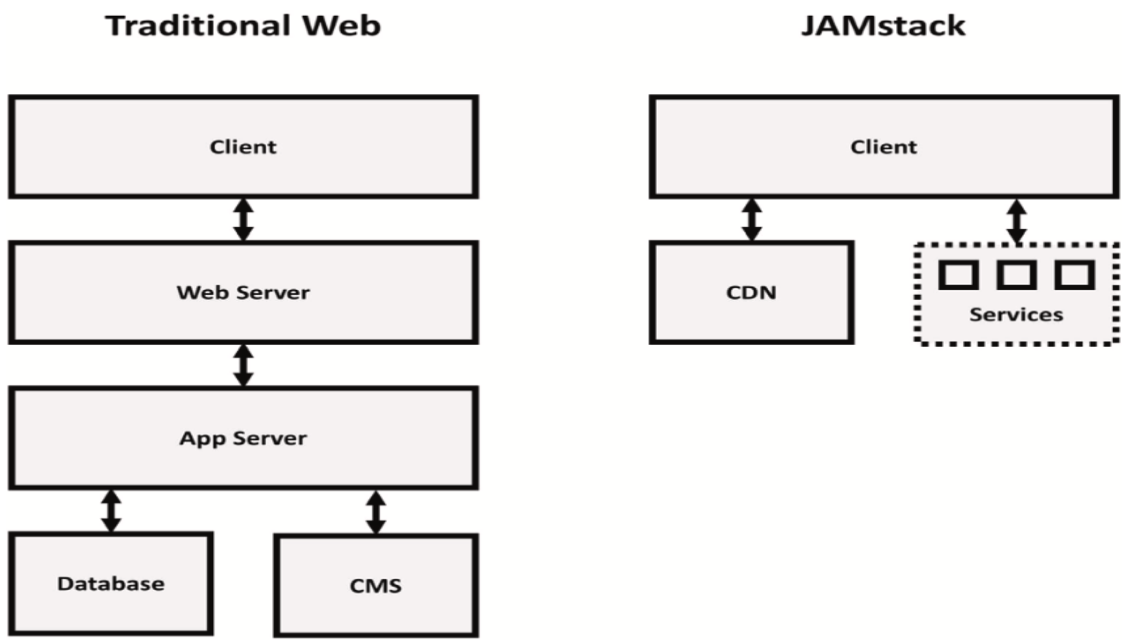


Figure 2.6: The differences between traditional web architecture and JAMstack web architecture (Peltonen, Mezzalira & Taibi, 2021)

2.2.5 Extensibility in Content Management Systems

In enterprise content management, employees will avoid using a system and will find workarounds if the satisfaction they gain from using the system is low. User satisfaction in enterprise content management systems is determined by information quality dimensions as well as system quality. Plugins and extensions can be used to bridge such gaps (Laumer, Maier & Weitzel, 2017).

Instead of writing new code, modern content management systems allow extensibility of their core functionality through a plugin system. Plugins are programs or parts of a program designed to add some functionality to a system. They extend software systems by allowing the provision of bespoke functionality that complement the core behaviour of the system. Developing web applications with a plugin system provides options for flexibility to developers (George, 2015; Mesa *et al.*, 2018). The three most common web content management systems, WordPress, Joomla and Drupal (W3Tech, 2021) all provide this functionality as well as repositories for accessing plugins created by others. WordPress in particular offers a significant number of plugins, with over 52,000 in active use, which have been used on the platform to build web applications with various functional requirements (Cabot, 2018; Martinez-Caro *et al.*, 2018; Mesa *et al.*, 2018). Content authors and marketers need the ability to change content quickly and easily. It is accessible from any computer that is connected to the Internet. CMS can be used for page updating on the fly, change or upload images, and add dynamic content to different files. It provides the developers with ready to use themes, and thus, it is a time-saving and easy operation as there is the requirement of less coding work (Bhowmik *et al.*, 2019). Plugin systems also allow software developers to create and extend their own programs based on existing library code which can reduce development time as well as allow more effort to be focused on developed functionality specific to the problem being solved by the system. The plugins created by software developers can be distributed to other developers thus promoting code reuse and again saving development time. Confidence in the quality of the software is also improved since it has been used severally and is considered battle-hardened. Additionally, plugins can also enable non-technical users to perform more complex tasks or to interface with more complex underlying systems (Martinez-Caro *et al.*, 2018; Mesa *et al.*, 2018). Plugin architecture when combined with visually developed applications has an even wider surface of application areas. Visual software allows modern applications to be more accessible especially to non-technical users. They lower the barrier to entry to creating and managing software. However, building graphical plugin systems especially for the presentation

layer can get very complex very quickly especially considering the constraints of the template system of the content management system (Ang, 2019; Liu & Su, 2020).

2.3 ARCHITECTURE EVALUATION

Evaluating the architecture of software is a significantly different activity from evaluating the lines of code or even blocks of code that make up the software. In a code artefact evaluation, more granular metrics such as lines of code per method and cyclomatic complexity are considered. These give an insight into the health of the code itself but may not reflect the state of the architecture. Architectural evaluation focuses on the larger context of the system as well as the structure of the artefacts and the relationship between them (Fontana, Ferme & Spinelli, 2012).

2.3.1 System Stability

Software system stability is a metric that investigates how much a system is impacted by any change. It is a measure of the degree to which modification in one part of the software will ripple out to affect other parts of the system. (Botella et al., 2004).

Software maintenance is all activity carried out on the software to ensure that it continues to function as expected or to change the existing functionality. Software maintenance can be very expensive especially without insight into how the non-functional attributes of the software impact its maintenance. The chosen architecture of a software system contributes immensely to the longevity of the project as well as the overall quality of the project. A stable architecture will allow the system to support updates and other maintenance without an extensive restructuring or rewrite (Pan & Wei, 2019, Salama & Bharsoon, 2017).

There are different contexts in which system stability can be examined. It can be looked at on the level of code or design or even architecture while considering the logical, physical or structural aspects of it. From an architectural point of view, stability is primarily viewed as the ability of a system to take on changes in its functional requirements or changes in its environment and constraints without the need for wholesale structural adjustment (Salama, Bharsoon & Lago, 2019). A low stability system is more susceptible to errors being introduced with every change. There is more testing and validation to be done on the system after every change since there are

many places a change could have cascaded and caused side effects. This makes software maintenance a more arduous task. Compare this with a system with high stability where there are almost no unintended effects from a change thus making it easier to test and validate a modification. Software that can handle changes without breaking is said to be robust (Bjuhr *et al.*, 2017).

To measure system stability the effect of a change in a software system is determined for each module in the system by inspecting the dependencies of the module and calculating the number of affected modules in the case of a change by using transitive closures. The average value from all the modules is taken as the overall stability of the system. The dependency information for every element is examined (Legunsen *et al.*, 2016). By calculating system stability relative to the size of the software project, we ensure that the metric is able to handle the addition or removal of modules.

Software architecture is a major determinant of system stability. It stands to reason then that monitoring the stability of the system can be used as a reliable proxy for measuring the health of the architecture. This is especially useful to watch over time to check for architectural degradation and check it before it becomes too expensive (Sturtevant, 2017). The system stability metric can be used to determine the critical paths for test coverage by revealing parts of the system with low stability. Areas with low stability thus need more concerted and rigorous testing approaches when a change is to be made there. In an example by Lattix Inc. (Barrow, 2019) a set of application code was deployed on some underlying framework and library code. Any changes to the framework or library code will affect the applications built on them. However, changes to application code will have a much lower impact on the overall application. It is thus important to have not only an overview of the stability of the software system but to also understand the sources of stability and instability in the system. In software with a layered architecture, the lowest layers tend to have the lowest system stability because any change to them affects the layers above. Therefore, the lower layers need much more testing (Botella et al., 2004).

System Stability	62.356%
Project	62.356%
Apps	73.184%
Component-1	71.029%
Component-2	70.234%
Component-3	99.221%
Component-4	72.653%
Framework	47.894%
Diag Service	41.269%
Link Service	53.465%
Management Service	53.983%
Schedular	51.799%
Driver	43.907%
HAL	39.833%
Util	36.043%

Figure 2.7: Example of system stability measurement (Barrow, 2019)

As shown in Figure 2.7, the project's overall stability is roughly 62%, however "Apps" has a stability of about 73%, while "Frameworks" and "Util" have much lower values. This implies that "Frameworks" and "Util" are dependencies in other components and thus modifications on them have a large impact and calls for rigorous testing (Barrow, 2019).

2.3.2 Web Performance

A study analysing the conversion rates of multiple e-commerce websites theorised that the conversion rate of the stores was strongly correlated with how acceptable the users perceived the wait time for the site's pages to complete loading. The conversion rate is the total number of customers who visit the online store compared to the amount of customers who end up making a purchase. The study posits that a drop off in conversion rate is possibly explainable by a commensurate increase in loading times (Stadnik & Nowak, 2017; Gao, Dey & Ahammad, 2017).

Google introduced the Speed Index (SI) metric to measure how quickly a web page reaches visual completeness. That is the speed to load the web page to a point that a user can see and interact with the page. The browser can progressively render artefacts sent to it allowing for partial page completion or asynchronous page loading behaviour. The nature of page rendering in modern browsers makes it possible for the user to perceive the page as loaded even though content is still being retrieved. This is because the page content above the fold can be prioritised while content below the fold or stretching beyond the viewport of the browser may continue to be retrieved. This means that Above The Fold time metrics likely form a more appropriate input than Page Load Time (PLT) for modelling the Quality of Experience for users browsing web pages. The Speed Index exclusively focuses on the effect of rendering times on user experience and does not include factors such as external distractions or task specifics. Many metrics exist to measure web page load times such as Speed Index. The SI metric along with other versions of it are established as the industry standard for web performance and testing (Hoβfeld, Metzger & Rossi, 2018).

Multiple web performance metrics including load time, fully loaded time, time to first byte (TTFB), start render time, speed index, first contentful paint (FCP), and time to interactive (TTI) are used as a proxy for measuring the user experience when it comes to the web page. These metrics capture different points in the page load lifecycle that determine whether the user's perception of the process is that it occurs quickly, normally or slowly (Ramakrishnan & Kaur, 2020). Figure 2.8 depicts the page load lifecycle and the metrics tracked at each point in the process.

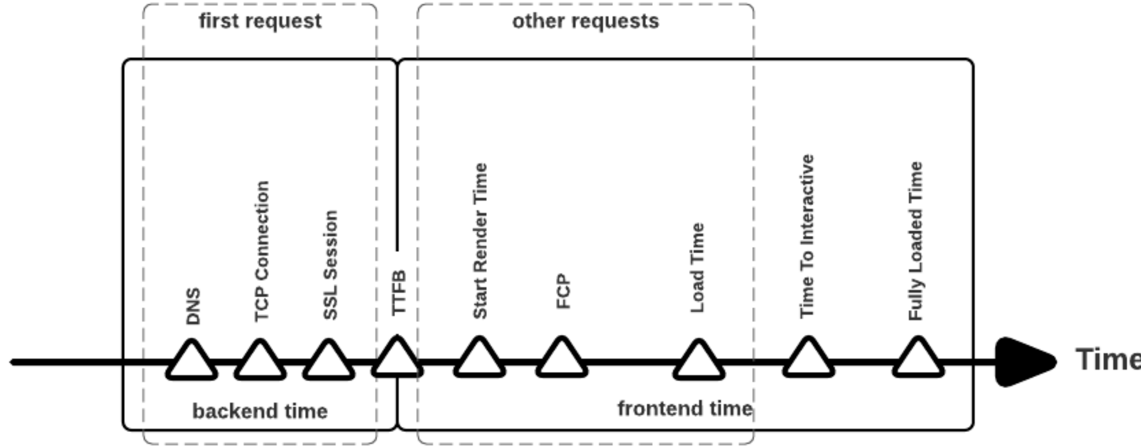


Figure 2.8: Web page performance metrics relative to the time of rendering (Ramakrishnan & Kaur, 2020)

Each of the performance metrics have distinct definitions and correspond to an event in the page loading lifecycle. The metrics are each explained below (Hoβfeld, Metzger & Rossi, 2018; Ramakrishnan & Kaur, 2020).

i. **Time To First Byte (TTFB)** measures the length of time from when a user lands on a page until the browser receives the first byte of the response. TTFB can also be called the backend time as it includes the time to fulfil server side processes including database queries and results, API calls to other web services and composing and interpolating HTML with dynamic data. After this event, all remaining processing time is considered frontend time (Ramakrishnan & Kaur, 2020).

ii. **The start render time** reports the time from the user navigating to the page up till the browser begins the process of rendering. This typically manifests as the browser showing the first bit of content on the screen.

iii. **First Contentful Paint (FCP)** tracks the time till the browser displays or paints the first content. FCP is a different measurement from the start render time which tracks when the rendering process starts. However for most browsers and web pages, the browser begins to paint content as soon as rendering begins so these metrics end up very close to each other. In many evaluations, the FCP is used to mean both situations (Ramakrishnan & Kaur, 2020).

iv. **Load time** is the duration from the start of the navigation, when a user lands on a page until the document and all of its embedded objects including resources like scripts, stylesheets and images have been completely downloaded from the remote server. The end of this period is marked by the browser firing the onload event (Hoβfeld, Metzger & Rossi, 2018; Ramakrishnan & Kaur, 2020; Grassi *et al*., 2021).

v. **Time To Interactive (TTI)** is a metric concerned with the amount of time from when the user first navigates to the page until when the page starts to respond to user input. This is often not immediate due to blocking actions in the critical rendering path (Ramakrishnan & Kaur, 2020).

vi. **Fully loaded time** considers the period from the browser's first request of the page until the final network activity is completed. This allows the metric to capture network activity resulting from lazing loading or deferring resource requests. Deferred resources are requested after the onload event has been triggered. Deferring non-critical resources or resources that are used further down on the page such as images can improve Above The Fold (ATF) loading time and thus perceived user performance (Hoβfeld, Metzger & Rossi, 2018).

vii. **Speed index** takes the approach of measuring the duration of visual completeness from an outside perspective. This metric is referred to as time integral because it takes into consideration the limitation of any metric defined at a single instance of time being able to capture all the complexity of the interaction between the user and the process of rendering the page. As an alternative, this metric and others like it, favour mathematical integration of load time over the various events in the course of rendering the page. The speed index is formally represented as the integral of the progressive visual state which is measured using multiple histograms of pixel-level changes to the page as seen. This approach is computationally expensive and multiple alternative indexes are developed with the same principle in mind (da Hora *et al*., 2018). The speed index is generally given as (Hoβfeld, Metzger & Rossi, 2018):

$$SI = \int_0^t (1 - R(t))dt \qquad (1)$$

$R(t)$ represents the visual completeness of the page using the mean pixel histogram difference ($MPHD$) between the page in current state given by I_t at a given time t and the state of the page after Above The Fold (ATF) rendering is completed given as I_T. $R(t)$ is given as:

$$R(t) = MPHD(I_T, I_t) \text{———} (2)$$

2.3.3 Complexity Metrics

Code complexity is an attempt to understand the relative complexity of a software project. By empirically calculating certain metrics an understanding of the effort required for maintaining and testing the system can be approximated (Masmali & Badreddin, 2020).

A basic metric is the source lines of code (SLOC). This metric tells the size of software and can be considered in different ways. It can be considered as the count of only imperative lines or could include assignment statements and comments. Since there are many ways to consider this metric, it is not very reliable as a way to compare software systems. It is a crude metric but can be used to compose more sophisticated measures (Toth, 2017).

Another important metric for software evaluation is cyclomatic complexity invented by Thomas J. McCabe in 1976. It is an empirical measure of software and is widely used in software evaluation. Cyclomatic complexity (CC) is found in graph theory as the cyclomatic number, which shows the number of regions in a graph. As a measure of software complexity it is the number of independent paths through a program. It is a proxy for the amount of effort required to exhaustively test the software since each path represents a different state the program can be in. This metric indicates how testable and maintainable a software system is. The cyclomatic complexity is calculated as:

$$\square = \square - \square + 2\square \text{———} (4)$$

M– Cyclomatic complexity

E– Number of edges

N– Number of nodes

P– Number of unconnected paths

Halstead in 1977 developed a set of software metrics known as complexity measures as a means to establish empirical methodology in software engineering.These metrics are meant to be independent of the execution platform and evaluates the implementation of the algorithm (Hariprasad *et al.*, 2017; Toth, 2017). The metrics are based on certain primitives described below.

η_1– Number of unique operators

η_2– Number of unique operands

N_1– Total number of operators

N_2– Total number of operands

From these primitives more complicated measures can be expressed:

i. **Program vocabulary**: The program vocabulary is defined as the sum of number of unique operators and number of unique operands. It is represented as:

$$\eta = \eta_1 + \eta_2 \quad (5)$$

ii. **Program length**: The program length is defined as the sum of the total number of operands and the total number of operators. It is represented as:

$$N = N_1 + N_2 \quad (6)$$

iii. **Halstead volume**: Halstead volume represents in bits the amount of space required to store the program. It is represented as:

$$V = Nlog_2(\eta) \quad (7)$$

iv. **Difficulty**: Program difficulty represents the difficulty of writing or reading the program. It is represented as:

$$D = \frac{\eta_1}{2} \times \frac{N_1}{\eta_2} \quad (8)$$

v. **Effort**: Halstead's effort measures the amount of work required to modify a program. It is calculated as the product of the volume, given in Equation 7, and the difficulty, given in equation 8. and A lower value as the Halstead effort typically means a simpler program to change. It is represented as:

$$E = D \times V \text{------------------} (9)$$

Maintainability Index (MI) is a measure that is composed of multiple metrics. It was proposed by Oman and Hagemeister and attempts to reduce relative maintainability to a single value (Hariprasad *et al.*, 2017; Toth, 2017). The Maintainability Index is composed of weighted Halstead metrics, McCabe's Cyclomatic Complexity and lines of code (LOC). The formula to calculate the metric is given as:

$$MI = 171 - 5.2 \times ln(V) - 0.23 * CC - 16.2 \times ln(L) \text{------} (10)$$

MI– Maintainability Index

V– Halstead Volume

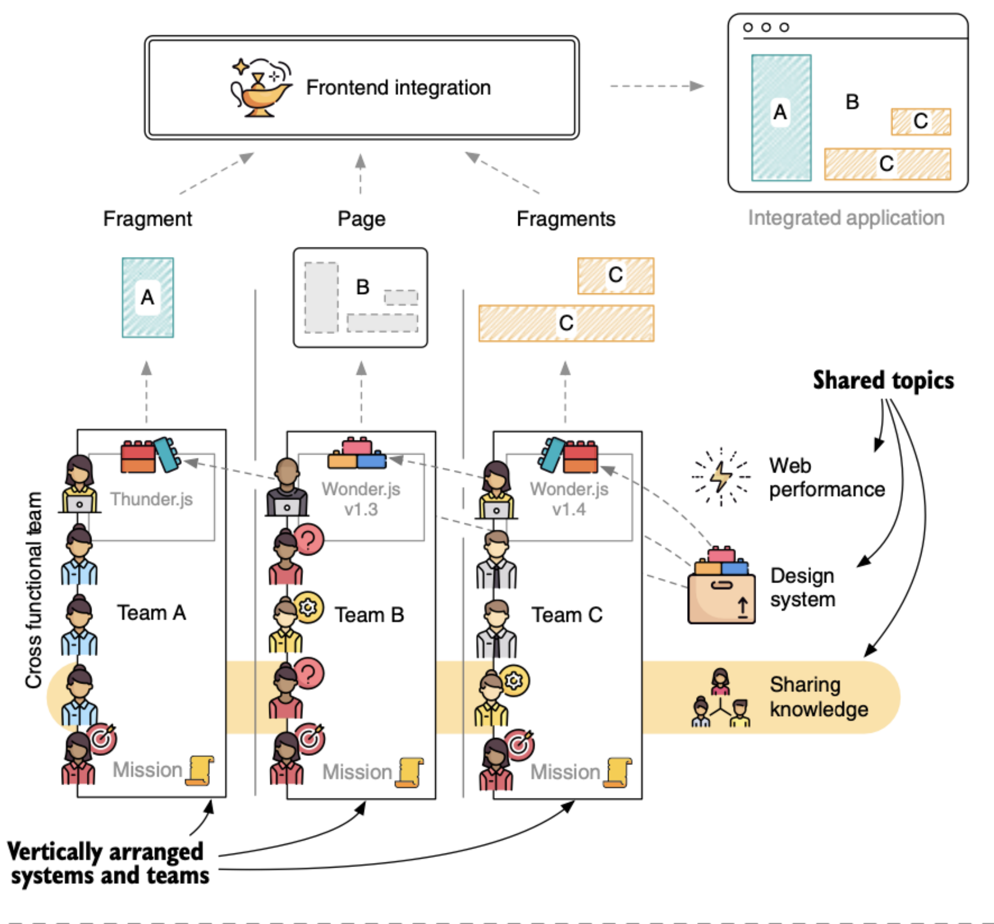
CC– Cyclomatic Complexity

L– Source Lines of Code

2.4 MICRO-FRONTENDS

Micro-frontends reason about a web application as a collection of different features whose responsibility is controlled by different teams. Each team has an independent business or function that they focus on. A team is cross-functional and develops its features end-to-end, from backend to frontend. Micro-frontends are a frontend software architecture where multiple standalone frontend applications which can be delivered on their own are composed into a greater whole (Jackson, 2019). Geers (2020) describes micro-frontends not as a definite piece of technology but more as an approach to organising and architecting software especially regarding the presentation layer. This approach allows the organisation of software engineering teams around a specific business goal such as customer acquisition rather than a specialisation in technology such as a frontend or backend team. This cross-functional team composition can lead to more productive

outcomes for product development and software engineering teams (McDonough, 2000). An overview of the approach is presented in Figure 2.9. In the micro-frontend architecture, each independent team is responsible for the HTML, CSS, and JavaScript required for a specific functionality. In achieving this, the team may make use of any frameworks, libraries or tools that solves their specific issues and they are comfortable with. There is no sharing of framework or library code and so each team is freed from the technical burden of the choices of other teams (Geers, 2020).



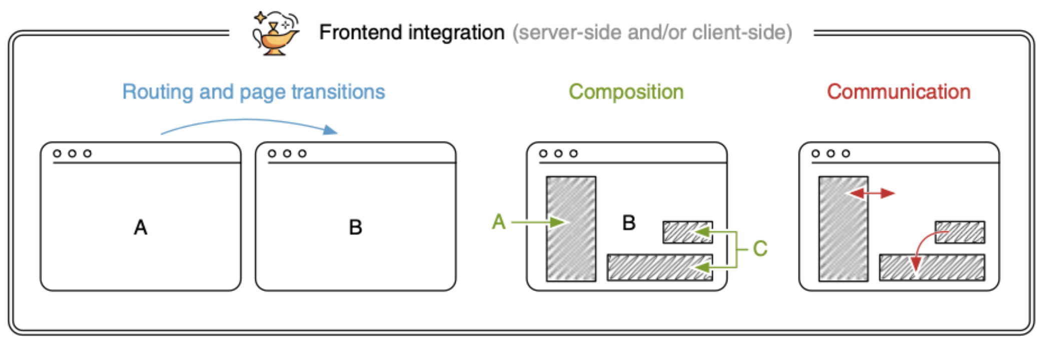


Figure 2.9: Overview of the micro-frontend approach (Geers, 2020)

Working on the frontend side of the application developers and software architects have a few architectural options to choose from including single-page applications, SPAs, in short, server-side rendering applications, or applications composed of static HTML files.

Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS) make up the documents that are the core of a web page. They control the content and visual appearance of the web page. HTML is either written directly by the programmer or programmatically generated by a web application. Rendering is the programmatic process of generating the web page from application code. The application that generates the document may either perform rendering on demand when a client requests that page or perform it before a request on the server.

2.4.1 Frontend Rendering Processes

Rendering processes may be organised into five (5) different classes based on where and when the process is carried out. In other words, they are classified based on the location and time of the rendering. The different classes of rendering are shown in Figure 2.10.

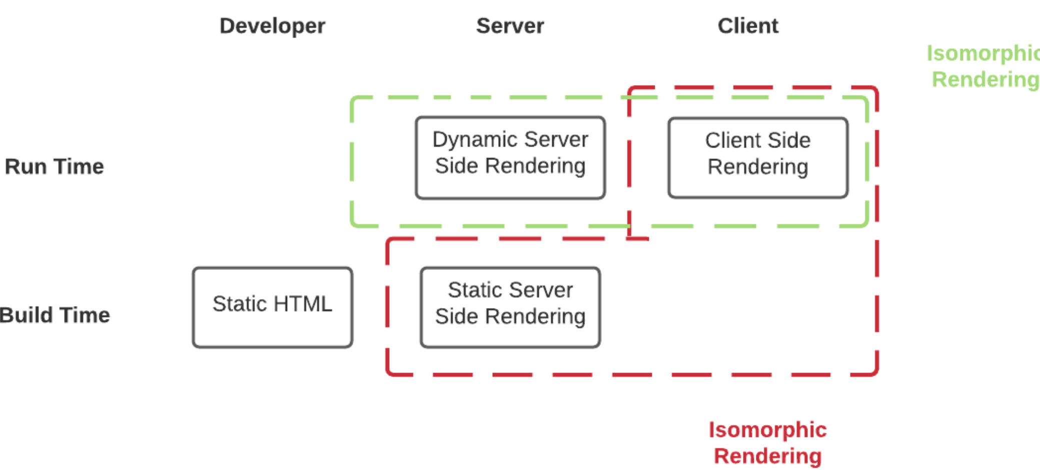


Figure 2.10: Types of rendering (Colliander-Celik, 2020)

Regarding the when, rendering can either be done at run time or at build time. Run time happens when the page has been requested by the user. The HTML is generated on demand based on the client's request. Build time rendering happens asynchronously before the page is requested and the HTML is stored to be sent to the client when requested. The where describes the location where the rendering takes place. In Figure 2.10, "Developer" indicates that it occurs on the developer's machine and is only applicable to static HTML which is written by hand. "Server" indicates that rendering was performed on a remote machine typically with significant resources. Routing with server rendering typically features round trips to the server with user action blocked while loading the new content as illustrated in Figure 2.11. "Client" indicates that rendering occurs on the client machine requesting the content. Static HTML is the rendering scenario where a programmer directly edits the HTML markup. This is the traditional method of developing web pages and all optimizations are made by hand. Static Server-Side Rendering (SSR) consists of using an intermediate templating language and then transforming this into HTML. The rendered HTML may then be stored on a different machine or network which will perform the delivery to the client. Static Server Side Rendering offers benefits to simplify the development workflow and allows for more complex logic than is possible with HTML alone. For both Client Side Rendering and Dynamic Server Side Rendering the HTML is generated at run-time, that is when a client requests it. It provides the advantage of being able to handle highly dynamic or contextualised cases. When Client Side Rendering is combined with a server side method, whether dynamic or static, it is known as isomorphic rendering.

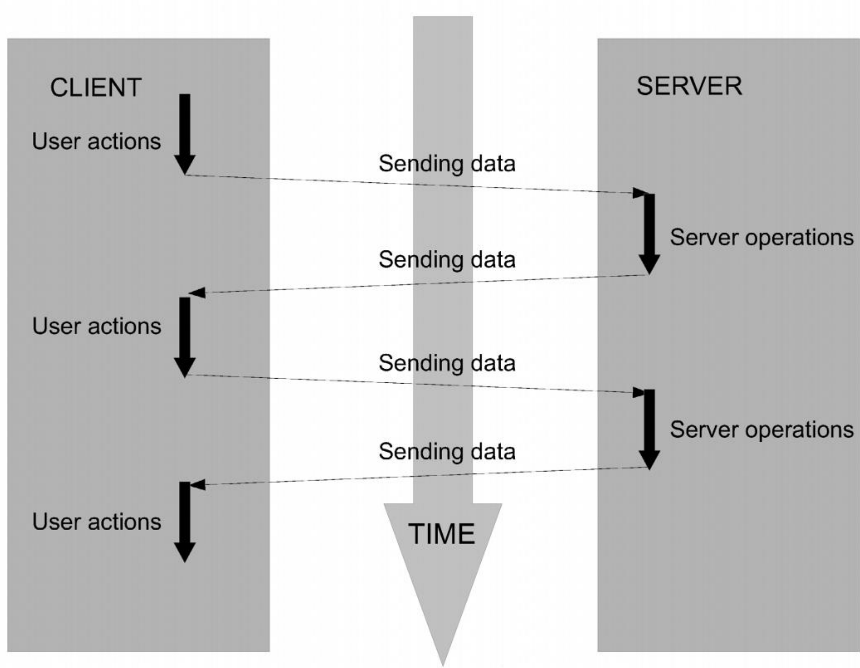


Figure 2.11: Data Fetching and Server operations with Server requests (Domański, Domańska & Chmiel, 2014)

2.4.1.1 Build Time Rendering

Both Static Rendering and Static Server Side Rendering are Build Time Rendering methods since they occur without being requested by the client and store the resultant HTML to be transmitted later. Static Server Side Rendering, also called pre-rendering, involves the transformation at build time of source code from one intermediate markup language such as markdown or JavaScript XML (JSX) into HTML. Jekyll, which is written in Ruby and used extensively by GitHub, is an example of a tool that translates markdown into HTML (https://jekyllrb.com/). Several templating languages such as PHP and JSX provide an extension of features to HTML, allowing for more imperative programming constructs such as looping or composition. This allows the use of more dynamic data and content within the template from sources such as a database or a computation. With this more powerful templating comes the option to run the rendering at build-time or at run-time. If none of the content depends on specific information from the request or the user, that is if everything regarding the content is already known, then prerendering is the preferred option.

2.4.1.2 Run Time Rendering

Run time rendering features the generation of the HTML on demand. Rendering is performed when the user makes a request. Run time rendering can either be dynamic server side rendering or client side rendering. Dynamic server side rendering has the generation happen on a remote machine and can use information from the request to customise the rendered content. The only difference between static and dynamic Server Side Rendering is the time at which the render is carried out. In client side rendering, the client receives application code to be executed on the client which then generates the HTML. Placeholder HTML may be sent on the first request which is then updated by the application code as needed. The browser exposes the Document Object Model interface to allow application code interact with and manipulate the markup as needed. Client side rendering allows for more interactivity as the rendered content can be updated in direct response to user action. The interactions are perceived to be faster since there is thus no need for a full page refresh. Such applications are described as Single Page Applications (SPA) since all the logic, including rendering and navigation can happen on a single page.

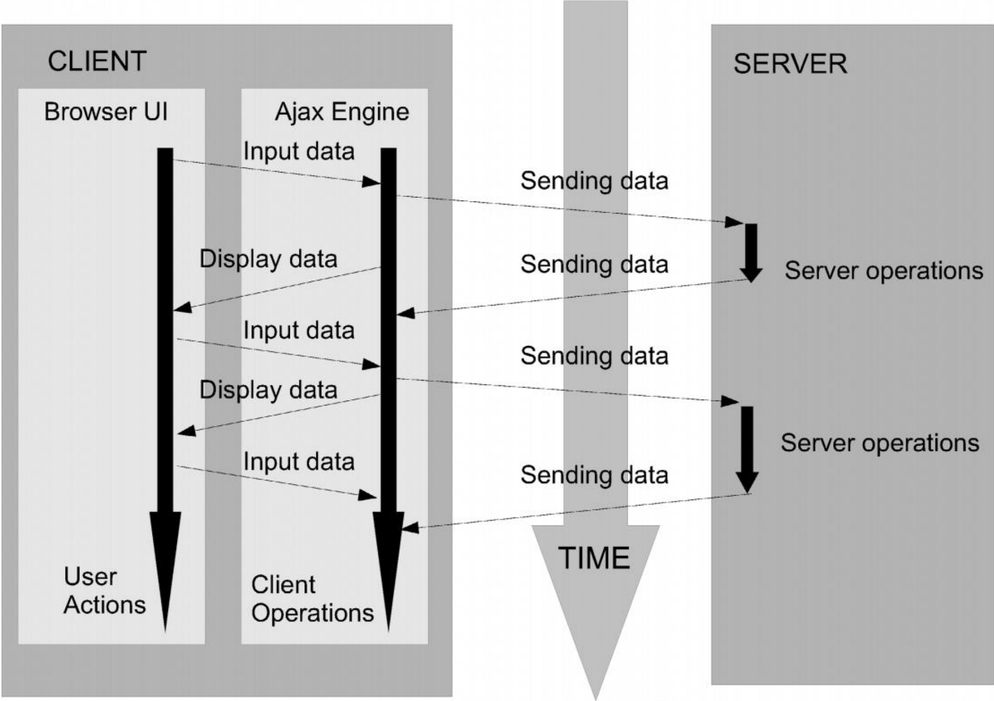


Figure 2.12: Data Fetching and Server operations with AJAX (Domański, Domańska & Chmiel, 2014)

Asynchronous JavaScript and XML (AJAX) provides a mechanism for the browser to request content from the server without a round trip to the server. The webpage is then partially updated with this new content. This mechanism is illustrated in Figure 2.12. The alternative is for the browser to send the request to the server and then receive a full HTML page as a response as illustrated in Figure 2.11 (Domański, Domańska & Chmiel, 2014).

2.4.1.3 Isomorphic Rendering

Isomorphic rendering which is also called universal rendering is the combination of multiple rendering techniques. It involves rendering on both the client and on the server. The content is rendered on the server and sent to the client along with application code that will allow rendering on the client. The first pass is thus faster as the initial HTML has enough content to engage the user. When the client side application code has loaded it can then take over the rendering cycle. This allows the benefits of both server side rendering which include a faster initial load and search engine optimization while also providing the benefits of a single page application including faster in-app navigation and better interactivity (Da Silva & Farah, 2018).

2.4.2 Frontend Monoliths

Over time these architectures might lead the project to become monoliths. This increases the complexity of the frontend application and making changes on part of the system may have unnecessary or unwanted effects on other parts. Codebases become huge, the application has a lot of dependencies and becomes tightly coupled, coordination between development teams becomes harder and slower, which leads to the law of diminishing returns. Increasing the number of developers on frontend teams will not affect the production rate, since the chosen architecture has set boundaries for developers (Paiva *et al.,* 2010).

Microservices are a variant of the service-oriented architecture that builds applications as a collection of loosely coupled services. It combines complex large applications in a modular way based on small functional blocks that communicate through a collection of language independent APIs. Each functional block focuses on a single responsibility and function, and can be independently developed, tested, and deployed (Chen, 2018). This makes the application easier to develop in parallel (Richardson, 2017). It also enables continuous delivery and deployment.

When a software company grows large enough and encounters problems with managing a huge codebase, it might be a good time to consider transitioning from a monolithic architecture to a microservices architecture. Microservices are good at handling complexity and size. Transitioning architectures will provide problems that have to be solved but may still be the better option. Monolithic Software is tightly coupled and can suffer from the drawbacks thereof. including difficulty in updating software, difficulty in testing, difficulty in scaling (Kalske, Mäkitalo & Mikkonen, 2017).

2.4.3 Quality Attributes of Micro-frontend Architecture

The advantages and disadvantages of this architecture are reflected in certain non-functional software attributes (Dragoni *et al*., 2017). These are:

i. **Availability**: Microservices are split into two or more as their complexity grows to preserve the ease of use when they are called. At the optimal service size, availability on a per service level is theoretically increased. Conversely, as the number of services increase, the system becomes more prone to failure on the basis of integrations, which leads to reduced availability.

ii. **Reliability**: Microservices offer less reliability than applications with an architecture that relies on in-memory message passing. This downside can be seen in any distributed system as networks cannot be assumed to be reliable.

iii. **Maintainability**: The loose coupling in microservices, as well as the inherent independence of each individual service increases maintainability significantly. It is much cheaper to modify services, fix errors or add new functionality, since there is less chance of an unintended effect being propagated by changes in one service.

iv. **Performance**: Network latency negatively affects the performance of microservices. In-memory calls are significantly faster than network calls and so as the number of calls over a network increases, performance will downgrade. If the context of the services are well defined, it is possible to achieve less degradation by sending fewer messages.

v. **Security**: Microservices suffer from similar vulnerabilities as service oriented architecture. As microservices use state transfer with various data-interchange formats, there needs to

be a focus on data security leading to additional overhead in building out encryption functionality. Providing authentication for third party services among different microservices and securely sending the data is another area of concern.

vi. **Testability**: The independence of components in a microservices architecture leads to better testing outcomes within each service since it can be done in isolation. Integration testing may be made more difficult especially when there are a large number of connections between services.

2.4.4 Composition of Micro-frontend Architecture

When the end user interacts with an application the experience has to be cohesive. It should not be apparent to the user that the application is an amalgamation of multiple independently developed micro-frontends. The process of integrating the micro-frontends into a single whole is known as composition. It is important to keep in mind that micro-frontend composition is a distinct process from rendering. Frontend integration describes the set of tools and techniques that are used to combine the separate micro-frontends into a cohesive experience for the end user of the application. It can be considered through the lenses of routing, composition and communication (Geers, 2020).

Based on the definitions of micro-frontends from literature, it is not specified whether the integration of micro-frontends should occur at build-time or at run-time. There are no inherent losses from adopting either direction and depends mostly on the end use case of the application being composed.

2.4.4.1 Linked Single Page Applications

A straightforward implementation is to develop separate pages using traditional approaches and simply linking between them. A web proxy can then be used to direct traffic to the appropriate page based on routing rules. This setup is visualised in Figure 2.13.

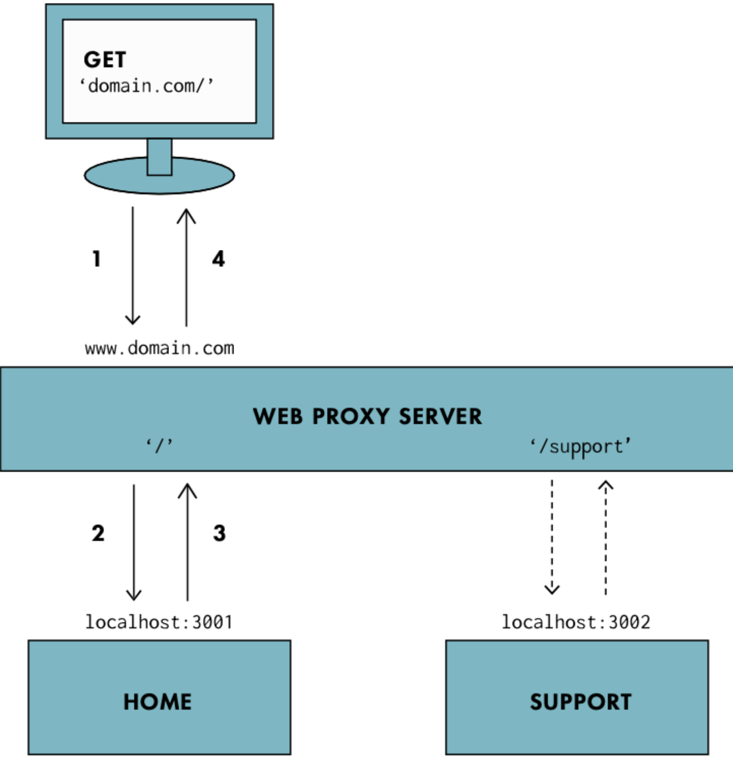


Figure 2.13: Web proxy server to redirect HTTP requests per route

Geers (2020) refers to this as Linked Pages when every page is a self contained application, and Linked SPAs when multiple pages are contained in a single SPA. Modern JavaScript frameworks typically have a dedicated routing library that allows the user to move between different pages of the application without a full refresh. The browser does not have to request and render a full, new HTML document only requesting the information it needs to update some parts of the page. Client-side navigation feels much faster and significantly improves the user experience. Only parts of the page that are changed have to be re-rendered. Static assets such as JavaScript and CSS are not fetched again. When the page is changed by the browser loading the complete HTML it can be considered a hard navigation. This is in contrast with a soft navigation that happens only on the client with the necessary data for the new page fetched over a network call. For the monolithic frontend it is usually one approach or the other. A monolith can either use server-rendered pages with each route being freshly fetched from the server or it can be a single page application with client-side routing. In the first case, all page changes are considered hard navigations while in the second case, all routing is soft navigation. In a micro-frontend enabled application the two

approaches can be combined using soft navigation between some routes and hard navigation between others. This approach is illustrated in Figure 2.14.

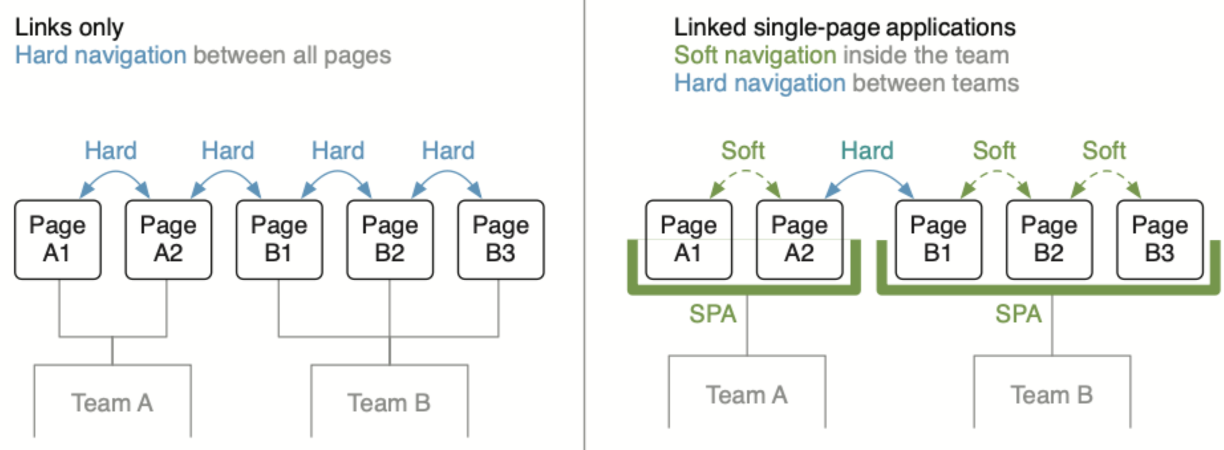


Figure 2.14: Contrasting hard and soft navigation (Geers, 2020)

In both of these methods the link between pages is all that each team needs to be aware of. The technical requirements to get this to work are very low. In some cases, it may not be acceptable to have to use hard navigation due to the perceived degradation in user experience.

2.4.4.2 Server-Side Fragment Composition

Server-side composition is also a viable method of delivering pages. Putting together page fragments on the server has been a popular and widely used method for a long time (Fagan, 2002; Harms, Rogowski & Lo Iacono, 2017). Large internet companies like Amazon, IKEA, and Zalando use this method (Geers, 2020). This can be achieved using Server Side Includes on the web server such as nginx or Apache, Edge Side Includes where transclusion occurs on the network such as on a CDN or specially developed tools such as Zalando Tailor, Hypernova or Podium. This technique is also effective for page-level distribution of an application.

2.4.4.3 Iframes

Techniques used client side for transclusion are by definition valid methods of micro-frontend integration (Fouh *et al.,* 2014). The most straightforward and direct frontend integration method is the use of iframes, which is a web-native standard and are supported directly by all modern browsers. Iframes have been part of the HTML standard since version 4.0 in 1998 (Asleson & Schutta, 2006). It however brings a number of serious limitations including performance overhead,

accessibility problems, search engine optimization problems, and an unpredictable layout. Each browsing context is a complete web document environment thus each <iframe> will use more memory, bandwidth and other computing resources. Therefore, although it is possible to use multiple <iframe>s on the page there is likely to be performance degradation (Mozilla, 2021).

2.4.4.4 Web Components

A much newer web-native standard is web components, which in practice is a suite of four web technology standards. They allow dynamic custom elements to be defined and registered, in an encapsulated scope. Web Components consist of three standards that together allow the creation of reusable custom markup with self-contained logic. Custom elements are a set of JavaScript APIs for defining the behaviour and structure of a custom element. These custom elements can then be used anywhere in the regular HTML document. The Shadow DOM is another set of JavaScript APIs that allow the connection of a separate, self-contained "shadow" DOM tree to a custom or native element. This shadow DOM is rendered separately from the regular DOM which allows for the encapsulation of functionality and styling for a custom component. Finally, HTML templates including the <template> and <slot> tags allow the creation and use of markup that is not contained on that specific page. These templates allow easy reuse across several pages (Mozilla, 2021). Web components are implemented on large products such as Youtube to allow interactivity in components (Geers, 2020). One of the disadvantages of using web components is that they are exclusively a client-side technology and cannot be integrated on the server-side. This exclusive client side rendering can lead to problems with accessibility, search engine optimization and site performance. Many frontend frameworks such as Polymer and Stencil are compiled down to web components, allowing developers to use friendlier language constructs and syntax. Three popular frontend frameworks, React, Angular and Vue all offer support for being encapsulated as web components as well as for consuming web components within their markup.

2.4.4.5 Unified Single Page Applications

There are dedicated micro-frontend frameworks that use other methods apart from web components. They typically feature a simple integration layer that handles composition of the unique micro-frontends as well as routing between micro-frontend boundaries. This integration layer is a shell application that stitches together applications written in other frameworks. This approach can be referred to as a Unified SPA, since it takes other Single Page Applications and

turns them into a coherent, whole application (Geers, 2020). This technique is illustrated in Figure 2.15 showing soft navigation between pages as well as a shared application infrastructure. Most of these meta-frameworks extract a significant migration cost since the root application and parts of the micro-frontend applications have to be written to the specification of the micro-frontend framework (Colliander-Celik, 2020).

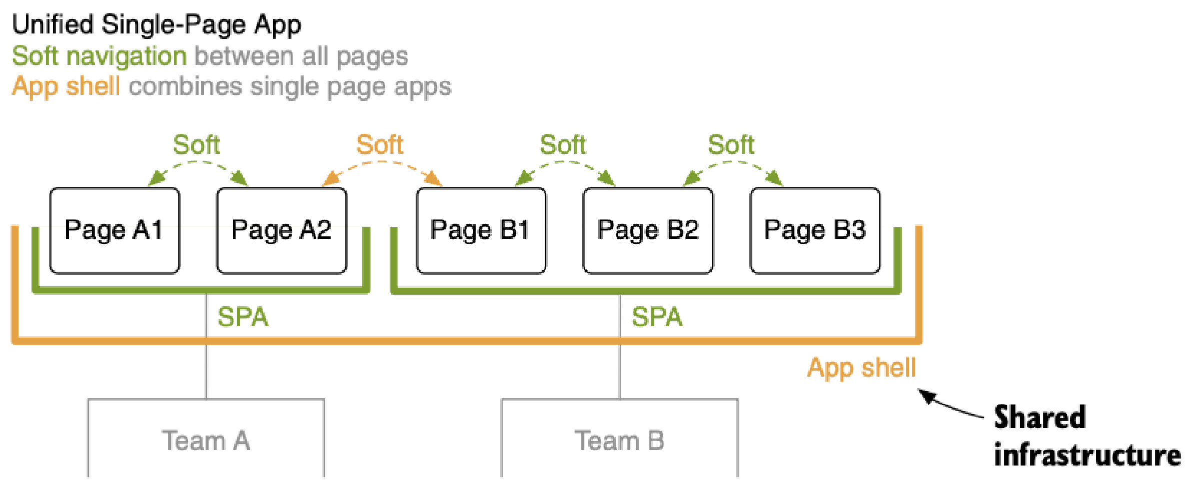


Figure 2.15: Shared application shell in the Unified Single Page App approach (Geers, 2020).

2.4.4.6 Webpack Module Federation

Module Federation is a frontend architecture that permits cross-loading of JavaScript applications and their dependencies at runtime. The application being requested and loaded is called a federated module. Dependencies are shared by the calling application and the federated module where possible. However, in the case of a required dependency for a federated module not being available in the host application, the dependency is downloaded and resolved from the origin server of the federated module (Jackson, 2020). Module Federation is available in Webpack 5 and is a major addition to the bundler that will allow an application to consist of more than a single deployable unit. Module Federation could have a potentially huge impact on the state of micro-frontends and JavaScript applications since Webpack is a popular bundler used in many frontend frameworks.

To conclude, similar to rendering techniques, micro-frontend composition techniques can be categorised based on the location of the composition: whether server side or client side. In contrast

to rendering techniques, composition can also be classified based on the depth of composition which can be fragment level composition or page level composition.

2.5 RELATED WORK

There have been multiple approaches to implementing flexible and extensible presentation layers in content management systems.

2.5.1 Lightweight Markup or Shortcodes

Dobrojević (2018) proposed the use of lightweight markup for composable user interfaces in content management systems. They are also called shortcodes and were heavily in use on forum websites and provided easy ways to format content by using the short, ubiquitous syntax to wrap around the content. The study considers Magma CMS which has as part of its core a shortcode parser to convert the special syntax into markup. While most popular content management systems have varying levels of implementation of shortcode functionality, Magma CMS supports the creation of page structure elements with the use of simple tags. This means that the markup behind elements such as menus, sliders, headers or footers is encapsulated in the shortcode. Developers can also encode behaviour in the tags. Additionally, tags can be composed to create more complex layout and behaviour. This allows content managers to link to related articles, generate reports and analyse content, or insert reusable components across pages.

2.5.2 Declarative Assembly of Web Applications using Déjà Vu

Perez De Rosso et al. (2019) developed a system of reusable fullstack components called concepts. A concept is a full-stack service composed of frontend components for the presentation layer, server side application and persistent data storage as well as "glue code" to hold it together. Each of the concepts are run in parallel and independently of each other with no direct message passing between the concepts. Conceptually speaking, an instance of a Déjà Vu concept is a state machine whose state changes are exclusively in response to user input from the frontend. Déjà Vu applications are created in a declarative manner and are composed by assembling the reusable concepts. The process is fully declarative using bindings which are expressed in a simple markup

templating language to perform concept synchronisation as well as data flow for both user input and rendered output. The template language is also used to structure the application layout thus determining the frontend components to be displayed as well as their position on the page or fragment. As a result, relatively complex applications can be quickly built by composing these pre-existing concepts.

2.5.3 Ontology based Content Management Systems

Vogt et al. (2019) described a novel approach using ontologies to annotate components and compose them using semantic programming techniques. An ontology is a formal representation of knowledge using domain specific concepts as well as the relationship between the concepts. This approach extends the use of ontologies to control the Semantic Ontology-Controlled Application for Web Content Management Systems (SOCCOMAS), an application for semantic web content management systems. These systems house content in the form of a tuple store knowledge base. The ontologies can then be used both in imperative and declarative form as an ontology-based language for describing the presentation layer as well as entities, interactions and all other processes within the CMS. This descriptive language allows the definition of views and inputs that make up the graphical user interface. A middleware or service layer uses these structures to generate the presentation layer code in AngularJS with a NodeJS backend for package management as well as local orchestration. The ontology-based language is also used to determine the appropriate HTML and CSS element to render. The elements are housed in a repository known as the GUI-Elements Catalogue which can be tailored to each instance of the CMS. Functionality and interactivity especially with form controls and behaviour are also expressed in the language of ontology. Figure 2.16 shows the full workflow of the ontology-based content management system. The complete content lifecycle (Benevolo & Negri, 2007) including drafts, published and revisions as well as entity management, logging, access control and permissions are all housed in terms of named graphs and directories and saved in the tuple store.

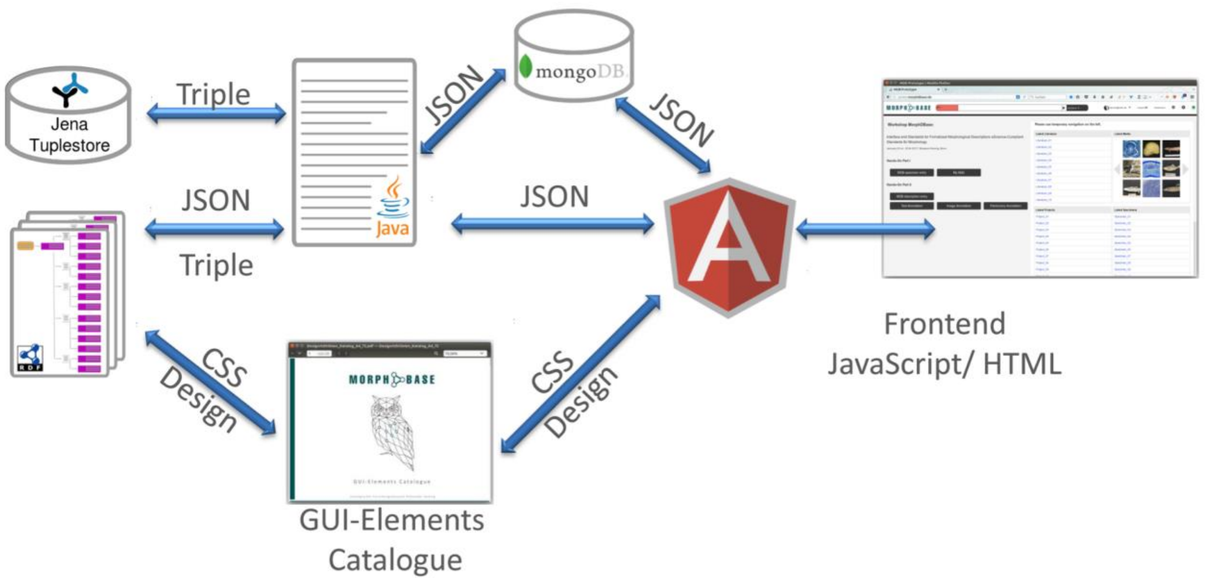


Figure 2.16: Application workflow for ontology-based content management system (Vogt *et al.*, 2019)

2.5.4 Micro-frontends in Content Management Systems

Yang *et al.* (2019) proposed a system design for a content management system based on micro-frontends. The content management system is divided into multiple submodules according to their function. The design makes use of a micro-frontend framework known as Mooa (https://github.com/phodal/mooa). The framework works with applications written in Angular, a JavaScript framework for frontend applications. The solution is optimised for legacy browsers including Internet Explorer 10. Mooa framework uses a master-slave architecture where multiple Angular applications can be instantiated simultaneously. One of the applications functions as the main application and initialises the other applications as well as managing critical features such as access control and permissions. The other Angular applications function as submodules and are only responsible for a narrow scope of business logic. This architecture is described in Figure 2.17. The main application fetches the configuration and settings from the server as part of initialization procedures. The configuration specifies the submodules and the main application is able to initialise each submodule and bind the lifecycle to an event bus. On a route change event, the main application checks to find a matching submodule for that route and if found, the submodule is loaded and control transferred to it.

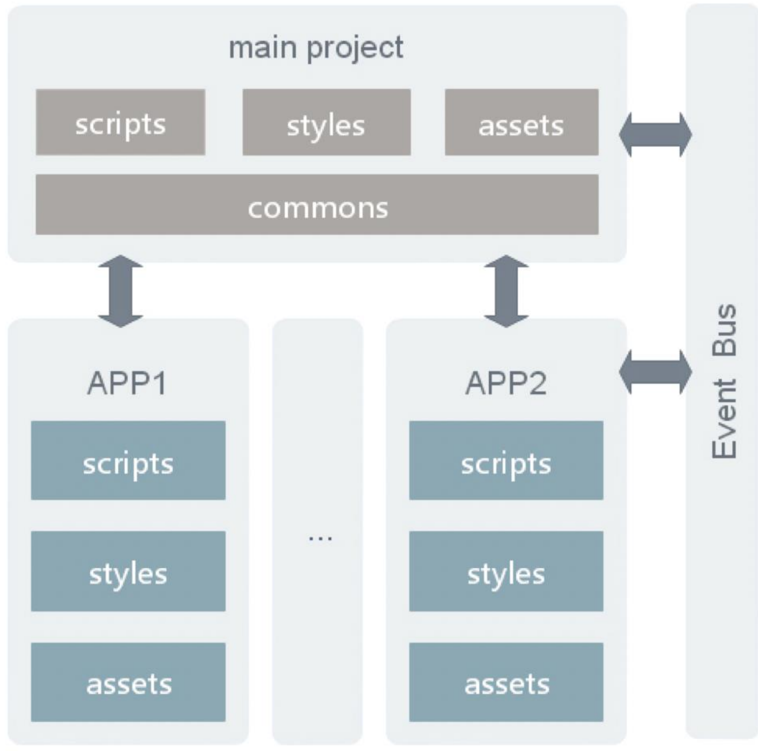


Figure 2.17: System architecture for proposed CMS with micro-frontends (Yang *et al*., 2019)

In this work, it was concluded that the micro-frontend-based content management system design enables teams to develop independently, quickly deploy and test individually, helping with continuous integration, continuous deployment, and continuous delivery. They report that while this architecture solved the frontend monolith problem in content management systems, it also brought some shortcomings. The mooa framework which the approach is implemented with only works using frontend applications built in Angular. This constrains the system against implementing multiple frontend technologies. Additionally, the integration of the multiple sub-applications is not as clean and organised as monolithic frontends. In modern Javascript applications, code splitting and bundling are important concerns as they affect overall performance but this was not factored into Mooa's design considerations. Due to the composition of the different frontend applications, there are also redundancies in the dependencies of each project which contributes to overall complexity as well as application size.

Wang *et al.* (2020) built on the above to propose a content management system for graduate records in the Chinese educational sector. Their approach favours Domain-Driven Design (DDD)

as it is a paradigm that has found success against complicated problem scenarios. The paradigm works by fostering a common terminology to describe the abstract problem space but in the business view and in terms of the software engineering effort. Software developers and business experts are thus able to clearly communicate and articulate issues. The business domain is divided into problem domain, bounded context and aggregate root. The application follows a similar separation of its artefacts. To begin with, the business domain is modelled according to the proposed system's requirements and constraints. Next the core business artefacts are extracted from auxiliary business services in order to form a "high cohesion and low coupling" business subdomain. The decoupling process is shown in Figure 2.18.

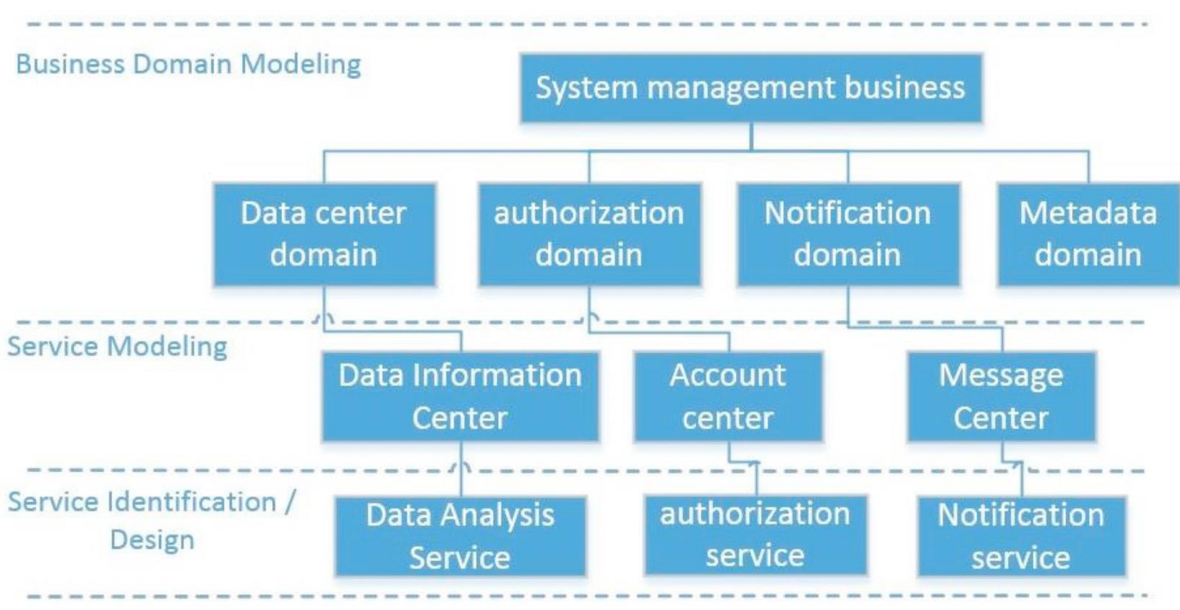


Figure 2.18: System Management Domain Design Process (Wang *et al*., 2020)

The graduate records system is divided into three business entities: school roll entity, student cultivation entity and system management entity. Considering the system management entity in detail showed the decoupling process of the core business domains and auxiliary services. The system management entity is divided into an authorization domain, a metadata domain, a data centre domain, and a notification domain. Central to each of these is the metadata domain which acts as the core domain for the system management entity. Each of the other business domains contains the corresponding context. When the business domain is identified while determining the

architecture, services are also identified and extracted as a model of the business processes. To illustrate, in the data analysis domain identified above, a data information centre provides centralised data analysis and management.

In the implementation of the system, the original school roll management as well as authorization management services are kept in their existing forms. They are implemented using the Model-View-Controller pattern and are served on the main application through an iframe container on the page. The native Window.postMessageAPI is used for cross submodule communication. For other submodules, web components are used in their development through the Angular framework. The resulting system supports self contained development and deployment for each submodule and posts no significant reduction in the quality of the user experience. In extending the system, new frontend technologies are supported. The system combines the submodules under the four major business domains specified and provides routing to different submodules. The system architecture design is shown in Figure 2.19. When the system is running, the main application of the program will obtain our application configuration in the system server, then initialise the submodules and bind their life cycles. To improve the user experience the submodules share and reuse basic components on the user interface as well as shared library code which are deployed and downloaded via a content delivery network. These measures reduce resource requests between submodules and improve overall page speed. Submodules are registered in a JSON file with the main application. Each entry in the file contains the details of a submodule including the name of the application, URL prefix and application entrypoint. Whenever submodules are added or removed, the configuration file needs to be updated although the main application will not require recompilation as it simply reads from the configuration file on demand. When the main application is initialised and has loaded the configuration file and active submodules it then generates the system navigation element and populates it with the relevant routes. When a matching route for a submodule is requested the graduate information system passes control to the relevant submodule and loads the code.

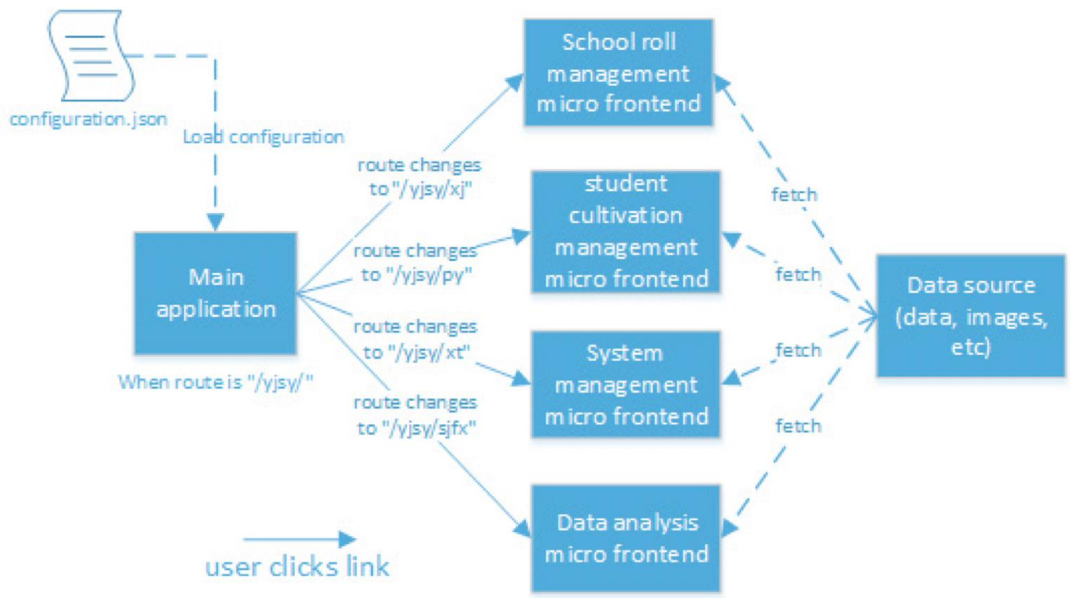


Figure 2.19: Routing and data fetch in micro-frontend based graduate information system (Wang *et al*., 2020)

CHAPTER THREE

METHODOLOGY

3.1 INTRODUCTION

This chapter introduces the methodology of the research. It also presents the overview of the system, requirements specification, analysis and design of the system. The framework and the deployment architecture are also discussed. Also, reflected in the proposed system is the integration of this system with popular open source content management systems. The workflow is depicted in figure 3.1.

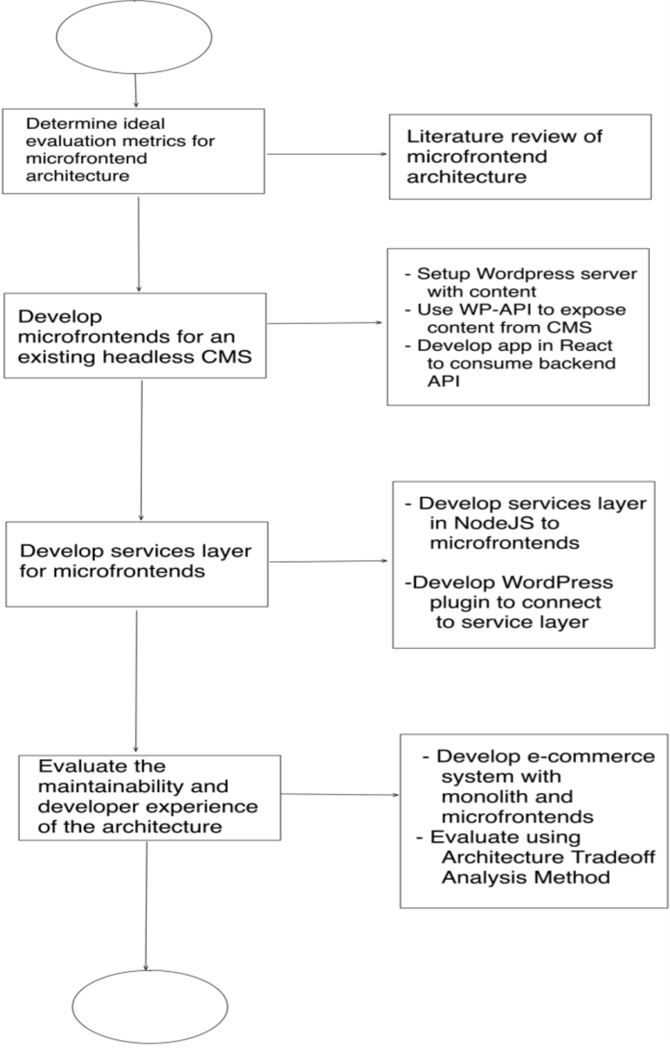


Figure 3.1: Workflow of the methodology

3.2 DETERMINE THE EVALUATION METRICS FOR MICROFRONTEND ARCHITECTURE IN LITERATURE

Review of literature on micro-frontends is carried out to discover the implementation and evaluation strategies used for this architecture. Software Engineering is a very industry oriented field of study. Thus it is vital to combine the cutting edge of the field in both academics and on the more practical aspect. The majority of software engineers do not publish their work in the typical forums of academia as such in order to adequately capture the current trends and ideas in the industry, it is necessary to include unpublished or non-peer reviewed work, which is referred to as grey literature, in this study. This sort of literature review is referred to as a multivocal literature review and in addition to using peer reviewed academic literature as is done in a typical systematic literature review, it also takes into consideration grey literature sources. This literature review method is especially helpful in closing the gap between academia and professional practice (Neto *et al.*, 2019, Garousi, Felderer & Mäntylä, 2019).

3.2.1 Research questions

The first objective of this study is to determine the evaluation metrics for micro-frontend architecture. Micro-frontends remain a novel area in both academia and industry and thus any research into them will need to draw heavily from both academia and industry.

Based on the goal of this work, the following research questions were generated:

1. How can micro-frontends be implemented?
2. How can micro-frontend architecture for web-based systems be evaluated against the software quality attributes of reliability, maintainability, performance, security and testability?

3.2.2 Design of the study

A Systematic Multivocal Literature Review (MLR) is the method adopted for this work since it is in a field heavily leaning toward software engineering practice. Typically, a multivocal literature review is focused on discovering the differences in opinion between academic researchers and practitioners. In this study, the aim of including grey literature is to synthesise the industry

recommended practises for building and evaluating micro-frontend architectures with a view toward further research in the area.

Here, peer-reviewed papers are described as academic literature, and other content including blog posts, white-papers and podcasts as grey literature.

The MLR process was adapted from Peltonen *et al*. (2021) where it was used to determine the benefits and motivations of adopting micro-frontend architecture. In this work it is used to discover practises for building and evaluating micro-frontend architectures. The process used in this study was based on seven steps and is depicted in Figure 3.2:

i. Selection of keywords and search approach
ii. Initial search and creation of initial pool of sources
iii. Reading through material
iv. Application of inclusion/exclusion criteria
v. Evaluation of the quality of the grey literature sources
vi. Creation of the final selection of sources
vii. Extraction of insights

This study followed a two-step (automatic and manual) search strategy. To execute the automatic search strategy, a query string was defined based on the keywords used in micro-frontend research. According to the research questions, the main keywords used are ‘‘micro frontends’’, ‘‘frontend microservices’’, and ‘‘frontend evaluation’’. The operators ‘‘OR’’ and ‘‘AND’’ were used to connect the primary keywords, synonymous terms, as well as key related terms.

In this study the keywords were used by adjusting word position or removing some of the terms in each iteration of the search process to obtain the relevant literature. The eventual search string applied was: (‘‘micro frontend” or ‘‘frontend microservices’’) AND (‘‘web architecture evaluation’’ or ‘‘frontend architecture evaluation’’). In running this query, this study makes use of the Scopus database provided by Elsevier.

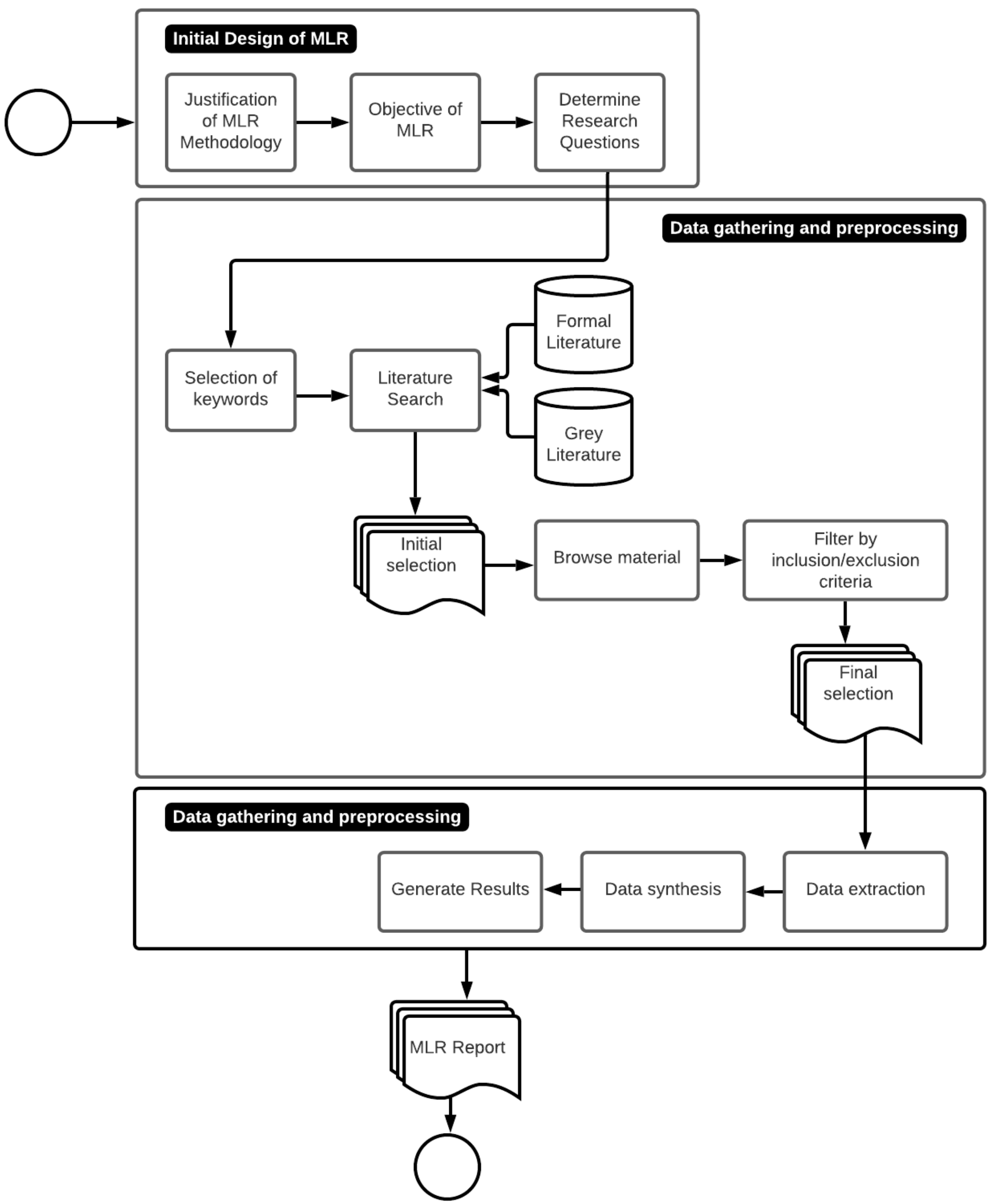


Figure 3.2: An overview of the multivocal literature review process (Peltonen *et al*., 2021)

To combat the bias inherent in using a single publication database, this study applied the forward and backward technique in the manual searching step. The technique entails attempting to trace

papers that cite the selected studies for the forward search while the backward search involves reviewing the referenced articles included in the selected studies.

After the initial selection is made from the sources, the articles are subjected to the inclusion/exclusion criteria to filter out those that would not add value to the study. For academic literature the inclusion criteria were as follows:

i. Papers published between 2015 and 2021
ii. Peer reviewed studies
iii. Papers with their full text versions available

The exclusion criteria for academic literature included:

i. Papers not using the term “micro-frontends” in the context of web engineering
ii. Papers not written in the English language
iii. Papers less than three (3) pages in length

The same query from the academic literature search was adopted for discovering grey literature. The search query is applied to Medium, Hashnode and Dev which enable this study to review work from industry practitioners. The forward and backward technique of searching outgoing links as well as resources linking to discovered materials was applied to broaden the search results. To ensure a high standard in the grey literature sources this study adopts the quality criteria put forward by Peltonen *et al*. (2021) and Garousi *et al.* (2019) for conducting a grey literature review. This involves screening the material according to the authority of the author, the quality of the applied methodology, the perceived objectiveness of the material, the date of publication, the contribution of new perspectives and the quality of the outlet it was published. The criteria is described in detail in Table 3.1. A ternary rating of 0, 0.5 or 1 was assigned to each criteria and then an average score was computed for each material. Materials with an average score lower than 0.5 were excluded.

Table 3.1: Grey literature quality criteria

Criteria Category	Questions	Options
Author's Authority	Has the author published other work in the field?	1: published over 3 materials in the field
		0.5: published 1 or 2 other materials
		0: no other published work
	Does the author have expertise in the area? (e.g., job title principal software engineer)	1: author job title is principal software engineer, cloud engineer, front-end developer or similar
		0: author job not related to any of the previously mentioned groups.
Methodology	Does the source have a clearly stated aim?	1: yes
		0: no
	Is the source supported by authoritative, documented references?	1: references pointing to reputable sources
		0.5: references to non-highly reputable sources
		0: no references
	Does the work cover a specific question?	1: yes
		0.5: not explicitly
		0: no
Objectivity	Does the work seem to be balanced in presentation	1: yes
		0.5: partially
		0: no
	Are statements in the work as	1: yes

	objective as possible?	0.5: partially 0: no
	Are the conclusions free of bias or is there vested interest? E.g., a tool comparison by authors that are working for particular tool vendor	1: no bias or vested interest 0.5: partial or small interest 0: strong bias
	Are the conclusions supported by the data?	1: yes 0.5: partially 0: no
Date	Does the item have a clearly stated date?	1: yes 0: no
Novelty	Does it enrich or add something unique to the research?	1: yes 0.5: partially 0: no
Outlet	Outlet control	1: high outlet control/ high credibility: books, magazines, theses, government reports, white papers 0.5: Moderate outlet control/ moderate credibility: annual reports, news articles, videos, Q/A sites (such as StackOverflow), wiki articles 0: low outlet control/low credibility: blog posts, presentations, emails, tweets

3.3 PROPOSED MICROFRONTEND ARCHITECTURE

The proposed model architecture is adapted from Attardi (2020) and describes the architecture of a headless content management system. As detailed in section 2.2.4 of this work, the headless CMS decouples the content engine from the rendering of the content. The content is then fetched via REST API onto the client and presented to the user. The adapted model architecture is illustrated in Figure 2.5. The frontend in this model is architected as a monolith which can lead to certain issues that have been detailed earlier in this work.

The proposed system's architecture uses the headless CMS approach favoured in the adapted work while introducing the concept of micro-frontends to the presentation layer. The proposed architecture is depicted in Figure 3.3.

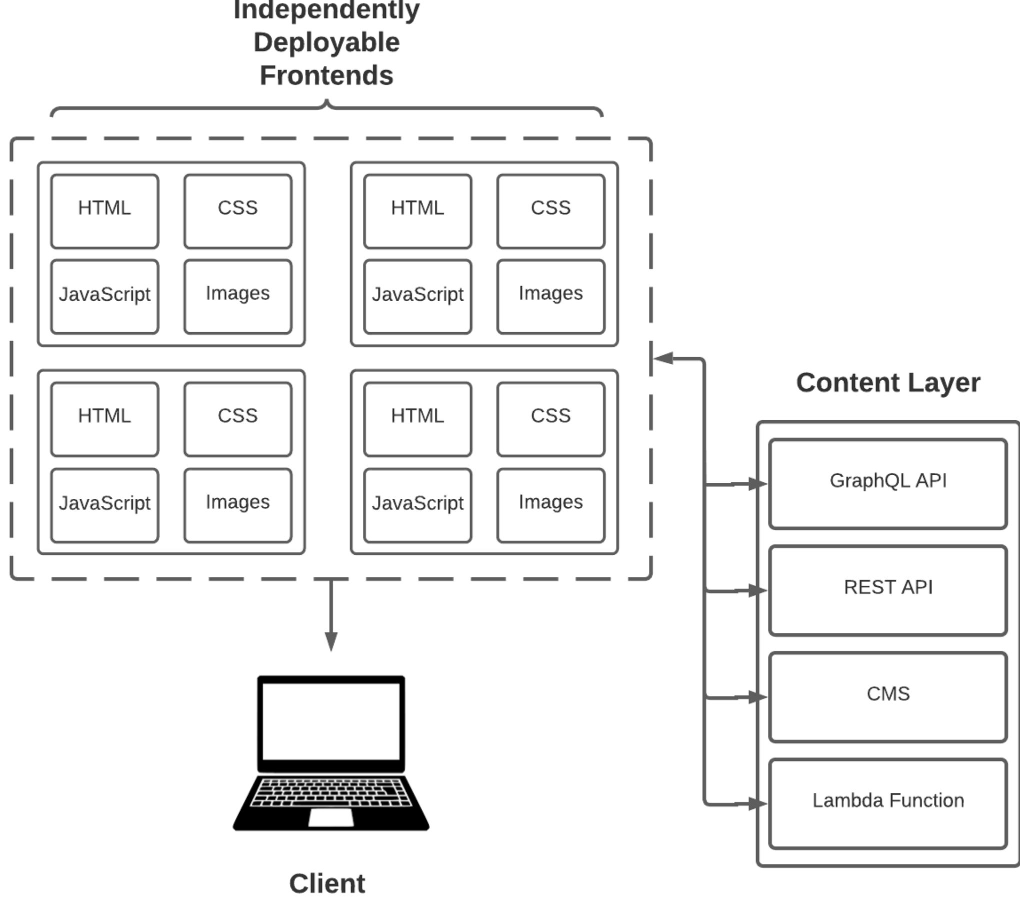


Figure 3.3: Overview of Proposed Architecture

3.4 SYSTEM DESIGN AND DEVELOPMENT

For this study a frontend to an e-commerce application powered by a content management system was developed. This frontend serves as a proof of concept of this architecture within a common use case for content management systems. The case study application was developed in two separate ways. Both applications encompass the exact same functionality and are expected to be nearly indistinguishable to users.

The first version is developed as a typical headless presentation layer for a WordPress site. It is built in React and uses the http library axios to perform data fetching from the WordPress REST API. The component based style of developing frontend applications is used and features splitting various parts of the application into components. There are five components defined in the first case study. They are: App, Cart, Header, Navbar and ProductList.

The second version features the use of micro-frontends in the presentation layer. This version breaks the monolithic app from the first case study into three (3) distinct frontend applications. As a way to demonstrate the flexibility of implementation afforded by this architecture, each of the constituents is implemented in a different frontend framework, although as noted in literature this is not strictly necessary for it to be considered a micro-frontend implementation. The three micro-frontends are labelled as Aggregator, Cart and ProductList.

3.4.1 Functional Requirements

Functional requirements describe how the system should behave. These requirements specify the required functionality of the system and serve as a guide to the design and implementation of the system. The following functional requirements exist for this project:

i. The user should be able to see a list of available products along with an image and price.

ii. The user should be able to add multiple products to their cart.

iii. When a user adds an item to the cart that already exists, the quantity of the product should be increased.

3.4.2 Non-functional Requirements

Non-functional requirements also known as quality attributes are features of the system that are concerned with how the system operates. This is in direct contrast with functional requirements which detail what the system should do. Non-functional requirements are very closely tied to the architecture. These are:

i. **Availability**: The ecommerce store should be highly available. It must be accessible to the user most of the time. In situations where a failure is unavoidable the system should degrade gracefully.

ii. **Maintainability**: The e-commerce store must be maintainable such that new modules can be added easily without affecting existing functionality.

iii. **Performance**: The ecommerce store should be highly performant. The user should not perceive noticeable lag in taking actions on the site.

iv. **Security**: User information and details entered on the ecommerce site must be secure and free from tampering or unauthorised access.

The content management system is language agnostic and serves content to other layers via a JSON Representational State Transfer (REST) API. With WordPress, this will be set up by leveraging the WP-API functionality to expose the required APIs. The services layer is responsible for composing and orchestrating the micro-frontends. This layer is built in NodeJS and will make use of Webpack and Babel for transpiling JavaScript as required.

3.4.3 System Requirements

The system requirements describe the software and hardware required to replicate the conditions in which the study was carried out. This section describes the conditions under which this implementation was performed as well as a description of the implemented software artefacts. Hardware requirements are expressed in terms of system specifications with a minimum value and a recommended value while software requirements are expressed in terms of version numbers.

3.4.3.1 WordPress Content Management System

WordPress is a popular web content management and was used as the data store and backend for the project. WordPress is used to publish a variety of websites and even more complex data driven applications such as e-commerce stores or membership management web applications.

Since version 4.5, released in 2016, WordPress has shipped with a native REST API implementation. This makes it possible to use WordPress as a headless content management system. The content management system returns the data we need via an Application Programming Interface (API) exposed in JavaScript Object Notation (JSON). A sample of the response is shown in Appendix B. Some fields have been truncated for brevity. WordPress is also the home platform of WooCommerce, the most popular open source e-commerce platform in the world. By combining the REST API with WooCommerce it is possible to headlessly build an e-commerce store managed by WordPress.

Table 3.2: **Hardware Requirements for WordPress CMS**

Requirement	Specification
Memory (RAM)	Minimum: 256MB Recommended: 4GB
Disk Space	Minimum: 1GB Recommended: 8GB
Processor	1 GHz

Table 3.2 describes the hardware requirements for the WordPress instance deployed for this study. The deployment to this server was done to a shared web host and thus the hardware is virtualized and shared by multiple users. This study does not consider the impact of the backend on performance as the same backend is used for both instances of the frontend which is the primary area of concern.

Table 3.3: Software Requirements for WordPress CMS

Requirement	Specification
WordPress Core	Version: 5.8
WooCommerce	Version: 5.4.2
PHP Interpreter	Version: 7.4
Apache Web Server	Version: 2.4

Table 3.3 shows the software requirements for the WordPress instance as used in this study.

3.4.3.2 Micro-frontend Presentation Layer

The presentation layer for the system is made up of an integration layer which couples multiple independently deployable applications. The application is initially server side rendered and then hydrated on the client in order to maintain a fast initial load time while providing interactivity on the client side.

Table 3.4: Hardware Requirements for Micro-frontend layer

Requirement	Specification
Memory (RAM)	Minimum: 1GB Recommended: 4GB
Disk Space	Minimum: 512MB Recommended: 2GB
Processor	1 GHz

Table 3.4 displays the hardware requirements for the micro-frontend layer. The application was developed on premise as opposed to in a cloud environment. The table contains minimum values which are required to replicate the conditions of this study but shows recommended values which reflect the actual conditions used.

Table 3.5: Software Requirements for Micro-frontend layer

Requirement	Specification
NodeJS Runtime	Version: 14.1
Express JS	Version: 4.16.4
Hypernova	Version: 2.5.0
React/ReactDOM	Version: 16.8.3
VueJS/Vue Server Renderer	Version: 2.6.6

Table 3.5 shows the software requirements of the micro-frontend presentation layer. The server side code is executed on a NodeJS runtime. ExpressJS provides the entry point and handles server side routing. It returns the shell app HTML which also includes the relevant micro-frontends. Hypernova is responsible for transmitting the string representation of the rendered micro-frontend to the integration layer. The integration layer features a hypernova instance and passes data to the required micro-frontend component. ReactDOM and Vue Server Renderer are responsible for performing the rendering of their respective frontend frameworks.

3.4.3.3 Headless Presentation Layer

To serve as a contrast, a second presentation layer for the same sample app is built using the Headless CMS approach. This approach features the CMS serving content as a JSON API which is consumed by the client at runtime. This monolith implementation of the presentation layer was built as a fairly conventional React application with a component based model.

Table 3.6: Hardware Requirements for Headless layer

Requirement	Specification
Memory (RAM)	Minimum: 1GB Recommended: 4GB
Disk Space	Minimum: 512MB Recommended: 2GB
Processor	1 GHz

Table 3.6 displays the hardware requirements for the headless presentation layer. Similarly to the micro-frontend layer, the application was developed on premise as opposed to in a cloud environment. This is to ensure similar deployment environments for both instances. The table contains minimum values which are required to replicate the conditions of this study but shows recommended values which reflect the actual conditions used.

The software requirements for the headless monolithic version of the presentation layer include the NodeJS runtime, Express JS and React/ReactDom. The exact versions of these requirements are specified in Table 3.7.

Table 3.7: Software Requirements for Headless layer

Requirement	Specification
NodeJS Runtime	Version: 14.1
Express JS	Version: 4.16.4
React/ReactDOM	Version: 16.8.3

The server side code and build process is executed on a NodeJS runtime. ExpressJS provides the entry point and handles server side routing. ReactDOM is responsible for performing the rendering of the React frontend frameworks.

The root component was responsible for fetching data and managing state that was then passed down to the various child components as required.

3.6 DEPLOYMENT STRUCTURE

The deployment structure is a three-tier layered architecture. It comprises the presentation layer, services layer, and the content layer, with all layers contributing to the total workability of the system. Figure 3.4 describes the three-tier layers and their distinct roles.

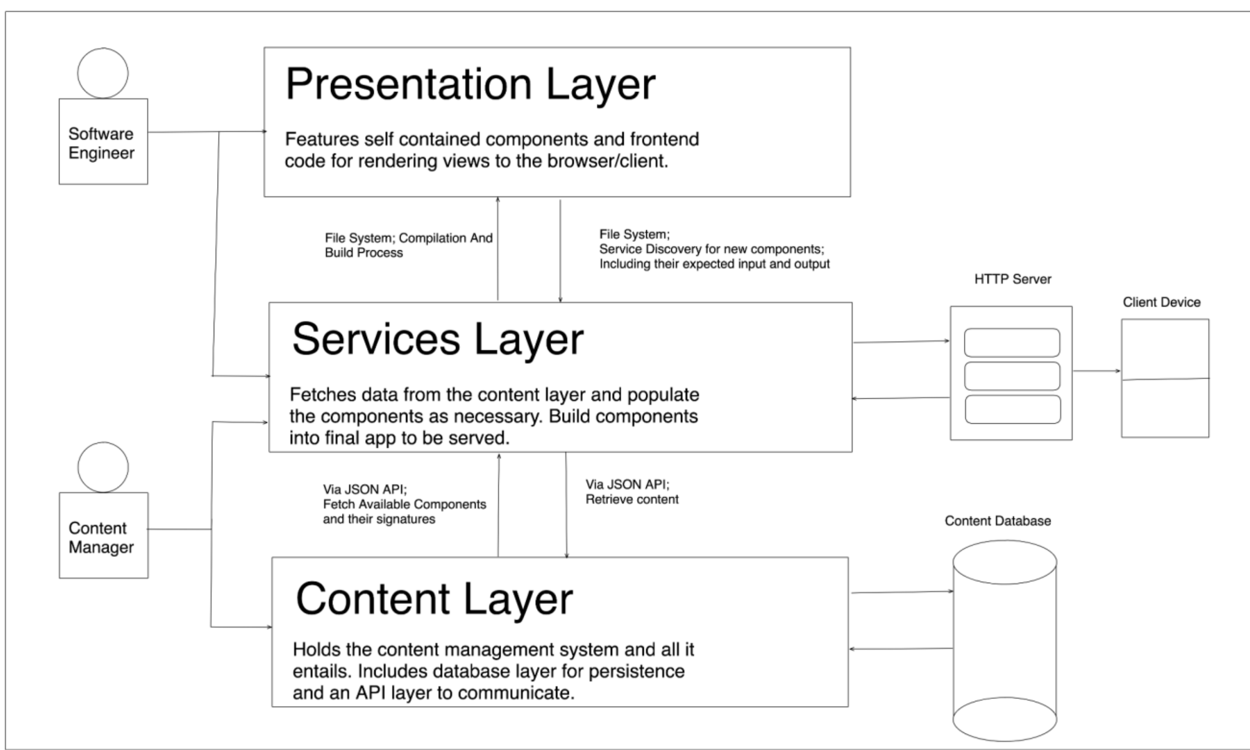


Figure 3.4: Deployment structure

The presentation layer is broken down into self contained micro-frontends which are tested and deployed independently. Each component is completely responsible for its internal functionality and does not rely on any other components. This single responsibility paradigm is crucial to the idea of micro-frontends in this architecture.

The content layer domiciles the content management system and communicates with the other parts of the system via JSON APIs to expose the content.

The services layer integrates the content from the CMS with the components in the presentation layer as needed. The services layer is also responsible for any build processes that need to be performed and compiles all the code and content into a single application. It queries the content layer and passes the results received to the appropriate components.

3.7 EVALUATE THE MAINTAINABILITY OF MICROFRONTEND ARCHITECTURE WITH A CONTENT MANAGEMENT SYSTEM

Two implementations of the case study systems will be carried out. The first will feature the monolithic frontend with the content pulled from the CMS headlessly via an API. The second implementation will feature the use of a micro-frontend for the presentation layer with content pulled from the content management system headlessly.

The monolithic frontend will be built in Javascript, with React as the framework of choice while the micro-frontend implementation is built using the proposed system with various independent components. The two implementations of the demo system will be evaluated side by side to determine how well they meet the specified non-functional requirements of **reliability, maintainability**, **performance**, **security** and **testability**. The stability of both systems will be evaluated. System stability measures the impact of change to the working of the system and systems with a higher stability are overall easier to maintain since changes are less likely to lead to unintended side effects. The complexity of the system will also be evaluated. Complex systems are harder to maintain and reason about. Thus reducing complexity is in the interest of a better system. The performance of the system will be evaluated by using the page speed metrics for the web.

The system stability evaluation is performed by running a dependency analysis on the high level modules recursively. Each module is inspected and the dependencies it includes are added as nodes with an edge to the original module while the included dependencies are scanned in turn. This process is carried out for each module and results in a dependency graph of the system. By inspecting the dependency graph we can infer the stability of the system in relation to its dependencies. To fulfil this objective the JavaScript tool dependency-cruiser (https://github.com/sverweij/dependency-cruiser) will be used.

The complexity analysis is carried out by parsing the code and creating an abstract syntax tree (AST) for each source file in the system. This AST is created by feeding JavaScript files into a JavaScript parser that performs the tokenization required to generate the AST. This process is also the intermediate step used in processing like code minification and IDE autocompletion. The generated AST is then analysed to obtain the complexity metrics including significant lines of

code, estimated errors, maintainability index and cyclomatic complexity. To achieve this objective the JavaScript evaluation tool plato (https://github.com/es-analysis/plato) will be used

Web performance analysis is important since this directly impacts the experiences of end users. If there is a significant degradation in performance, a new architecture will be rendered unusable because of that. Thus this step compares the web performance of the two resulting systems using the Google Pagespeed Index to determine their performance.

Finally, an expert evaluation will be carried out to determine the opinions of experts toward the model architecture. The evaluation takes the form of survey questions designed to gauge the experts opinion of the architecture in terms of performance, maintainability and complexity. All of which are also evaluated by the other methods in this study. Additionally, the survey contained a free form question to allow the experts to express their thoughts on the future of micro-frontend architecture especially in the context of content management systems.

CHAPTER FOUR

RESULTS

4.1 INTRODUCTION

The design and building of the architecture and the web application are discussed here. This chapter discusses the tools, software, and framework used for the implementation of the model and system.

Content heavy applications should typically be server side rendered to increase performance especially upon the first render as well as unlock other benefits such as search engine optimization. This work focuses on applying micro-frontends to content management systems and as such places a focus on initial server side rendering. The application will then be hydrated upon load to allow interactivity. Frontend applications support rich user interactions, by supporting intermediate states that may not be achievable with pure server side rendering (MacCaw, 2011).

To implement micro-frontends for the content management system it was imperative to choose a framework to orchestrate the implementation details and build steps of the various component frameworks.

4.2. MULTIVOCAL REVIEW OF EXISTING LITERATURE

This study performs a multivocal review of existing literature to find ways of evaluating micro-frontend architecture. The review took the form of a multivocal literature review to allow for incorporating non-published sources. After the initial selection of sources the total number of selected peer reviewed materials was 102. After the application of exclusion criteria the selection was reduced to 11 materials. For the grey literature search, the initial selection of sources yielded 429 materials. After the application of the grey literature quality criteria, the total number of selected materials was 126. The selected studies were then analysed to determine the method of rendering used, the method of micro-frontend composition, the communication technique between modules and the metaframework if specified.

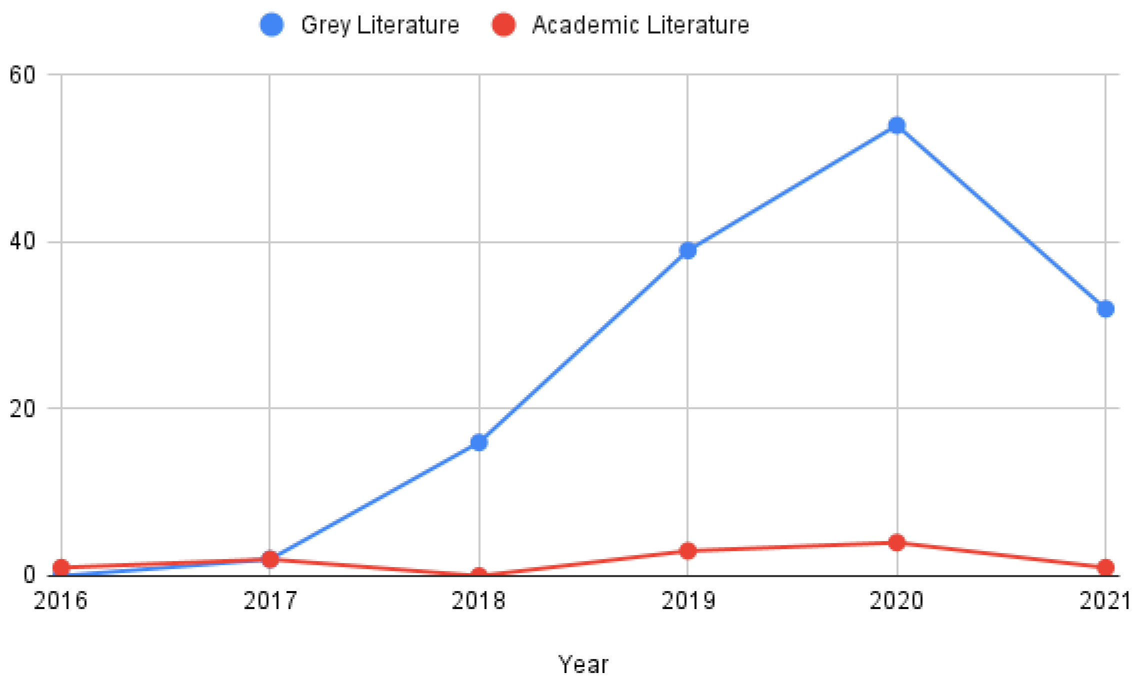


Figure 4.1: Temporal analysis of selected academic and grey literature

In terms of implementation, the reviewed work had a good distribution among both server and client side rendering as well as using different composition techniques. The identified rendering techniques from the review were client side rendering (52%), server side rendering (13%) and isomorphic rendering (17%) with 18% of the materials not reporting a rendering method. Each of these rendering methods are discussed in detail in section 2.4.1. Client side rendering as the most common rendering method is reflective of the shift to dynamic client-driven applications in recent years. The composition techniques used were categorised into Linked SPA (8%), Server Side Fragments (15%), Iframes (<1%), Web Components (13%), Unified SPA (24%) and Module Federation (18%). The composition methods of 21% of the selected materials could not be ascertained. Unified SPA was the most used method of composition especially considering that it is typically used in conjunction with micro-frontend frameworks. It is also possible to use the Unified SPA method with both server and client side rendering. Web components as an incipient browser standard represented the third most used composition method and was bested by the recently released Webpack Module Federation. However both methods were mostly used in conjunction with client side rendering. In the case of web components a major reason why most implementations were client-based could be the existence of mature server side templating and

transclusion techniques which may make it unnecessary to use Web Components. For Module Federation it is likely that as the technique gains popularity it will see adoption on server rendered projects as well, especially as popular frameworks that offer server side rendering such as NextJS (https://github.com/vercel/next.js/) announce support for it. Composition methods are discussed in detail in section 2.4.4. This study also explored the metaframeworks used in literature. While most materials considered did not report any metaframework (58%), single-spa (14%) was the most reported signalling that it may be the most mature micro-frontend framework available at this time. Other frameworks include Hypernova, Podium, Luigi, Ara, Templado and Ragu.

In reviewing related work on micro-frontends, it was observed that the major method of evaluation was by implementing the same system as a monolith and comparing the process of building them. The comparison was mostly made anecdotally rather than with empirical methods.

4.3 IMPLEMENTATION

The tools and methodologies used for the implementation of the e-commerce case study are discussed in this section.

Both implementations have the exact same styling. They both use the utility-based styling approach from the TailwindCSS library. The styling effects come out very similarly and both implementations look exactly the same. This suggests that it is possible to replace an existing system with a micro-frontend implementation without an immediate impact to the user interface and thus have end users unaware that a change has been made. Of course, user interface presentation is not the only factor that affects the user experience. Performance also plays a key role and the impact of the architecture on performance is discussed in section 4.4.

The micro-frontend layer uses unified single page applications as the method of composition as described in section 2.4.4.5 of this study. The root application is the part of the system responsible for orchestrating the other micro-frontends. The root application is responsible for data fetching to meet the data requirements of the included micro-frontends as well as the composition of them.

The root micro-frontend provides the static HTML that makes up the shell application and interpolates it with content from the micro-frontends once it is available before finally sending the full HTML to the browser. It also bundles client side JavaScript files from each of the included

micro-frontends in order to enable hydration and allow the pages becoming interactive once the bundles are loaded. This is known as isomorphic rendering and is described in detail in section 2.4.1.3.

The Product Listing page displays products available in the e-commerce store. It presents the product's name and a featured image. Also from the product listing page the customer selects the products they would like to purchase. These selections are then added to the cart. The rendered component is shown in Figure 4.2.

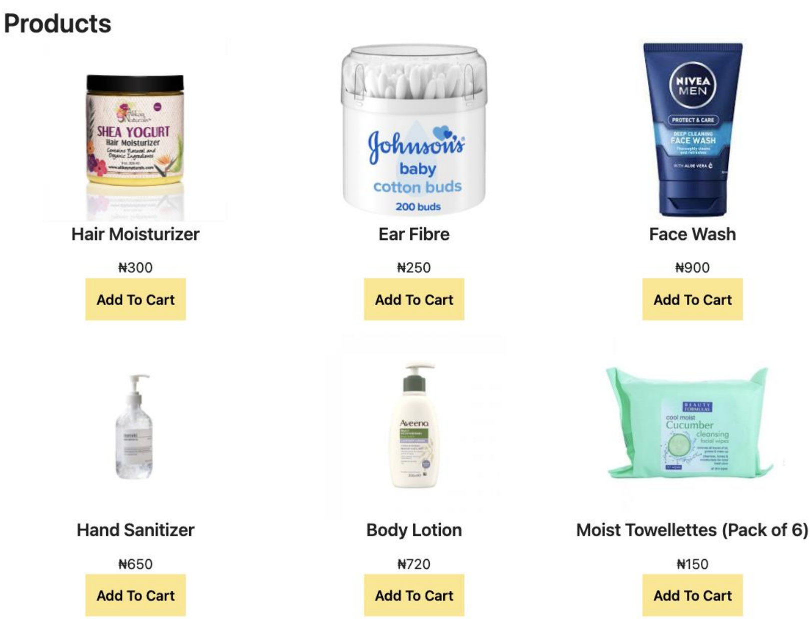


Figure 4.2: Product Listing Page

The product listing component is written in JavaScript framework VueJS. The component defines the markup for the product list within a template tag and also defines methods as required by the VueJS syntax. A method for adding products to cart is defined and attached to the button markup. The addition of products to cart is performed by using the native event model available in the browser. When the button is clicked the custom event is triggered and the details of the selected product are passed to the event to be broadcast for any active listeners on the page.

The Cart and Header fragment is at the top of each page and keeps track of which products the user has selected for purchase. The header also contains the store branding as well as links to other pages on the website. A screen capture of the rendered component is shown in Figure 4.3.

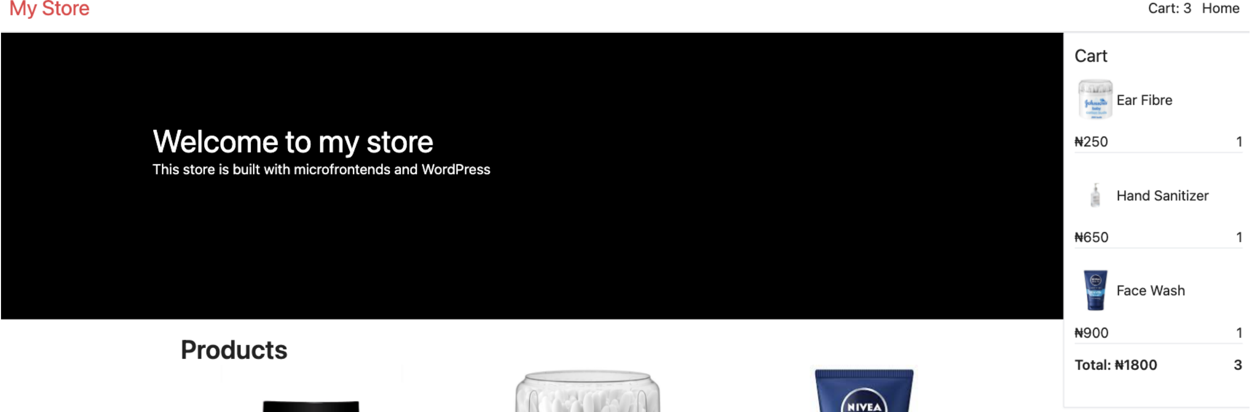


Figure 4.3: Cart and header

The cart component is written in the JavaScript framework React. The component is described as class-based and holds the current state of the cart. This state is defined upon initialisation of the component. React provides framework specific ways of updating the state and re-renders the component when the state changes. This framework specific method is used within this component. Once the component is ready an event listener is registered to listen for add to cart events. Whenever the browser fires such an event the component updates the cart state to reconcile the new items added to items already in the cart.

The implementation of the micro-frontend version of the case study showcases the use of two usually incompatible frameworks operating on the page while providing a model for non framework specific inter-component communication or message passing as described in the unified single page application approach (Geers, 2020).

4.4 ARCHITECTURE EVALUATION

The evaluation of the case studies takes place along three paths. The first is the system stability analysis which is based on inspecting the dependencies in each module and calculating the impact of a change in different modules based on the percentage of other modules that include the specific module as a dependency. The second evaluation approach features the use of page speed indices to measure the performance of each version of the application.

4.4.1 System Stability

The system stability measure offers insight into the impact of a change in a module of a system on other modules of the system. The system stability evaluation using dependency graphs revealed that as described in literature, a micro-frontend approach leads to high separation of concerns in the frontend. There were no dependencies between each of the modules. This was in contrast to the monolithic implementation which had a strong coupling of the components to its framework code. This evaluation also revealed a large difference in complexity. Each of the individual dependency charts for the micro-frontends was at least as complex as the dependency chart for the monolith.

4.4.1.1 Micro-frontend Presentation Layer

For the micro-frontend enabled version of the case study system the following dependency graphs shown in the Figure 4.4, Figure 4.5 and Figure 4.6 were obtained. The dependent graphs show various files within the module as nodes on the graph while the directed edges represent the importation of a module into another. By inspecting these edges we can see the relationship between modules and infer the degree of coupling or the dependence of these modules on one another.

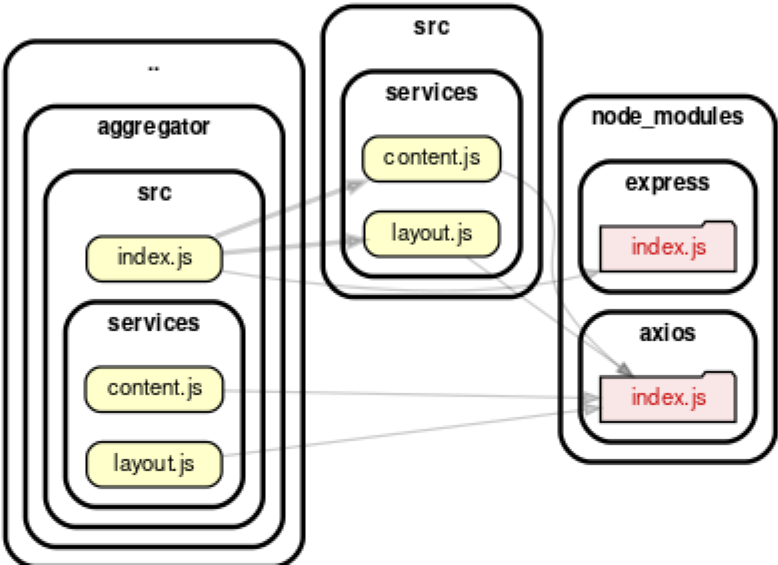


Figure 4.4: Aggregator dependency graph

The Aggregator module has dependencies between the layout and content submodule as well as axios from the node modules which contain dependencies external to the project. This module needs to display content from the other two submodules but avoids any code coupling between them. All of its dependencies are self-contained.

The Cart and Product List modules are similarly self-contained. As shown in Figure 4.5 and Figure 4.6 the components have internal dependencies but no edges linking to any of the other modules. This means that changes to the internal workings of the components should not cascade to affect other modules or components.

There are exactly zero (0) dependencies between the Aggregator, Cart and Product List modules. This is because the rendered content from both the Cart in React and the Product List in Vue are exposed as JSON over a REST API from the hypernova clients installed in both of these modules and shown as dependencies in Figure 4.5 and Figure 4.6 respectively.

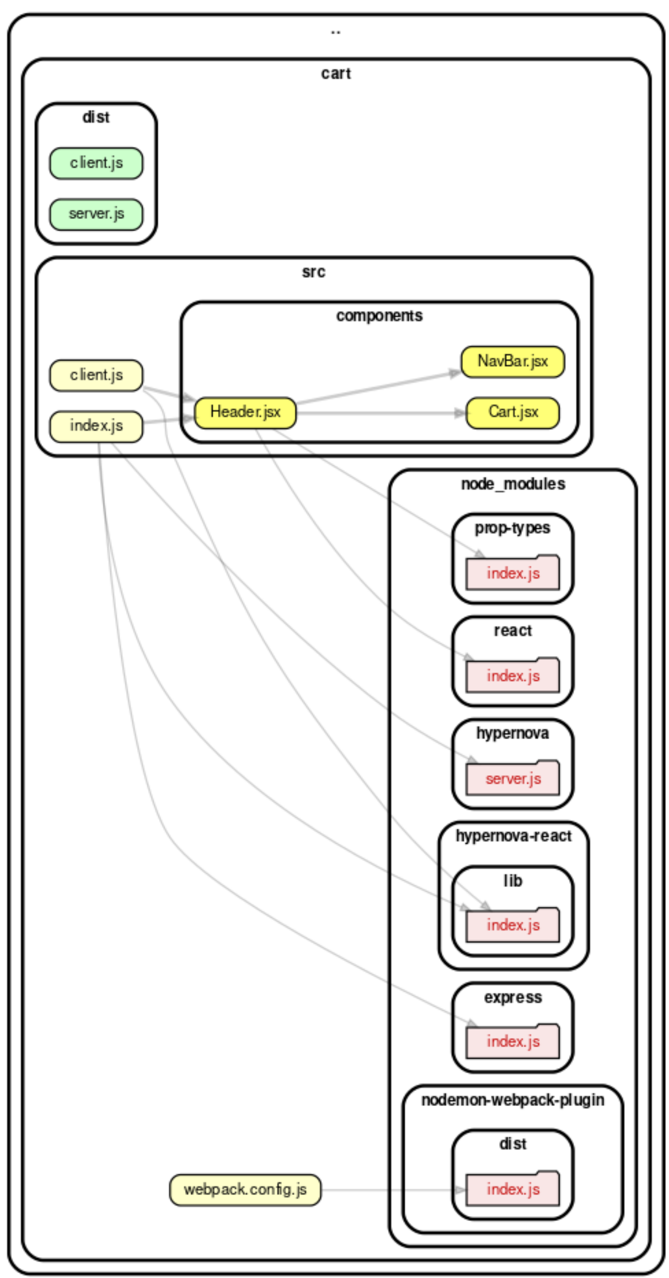


Figure 4.5: Cart dependency graph

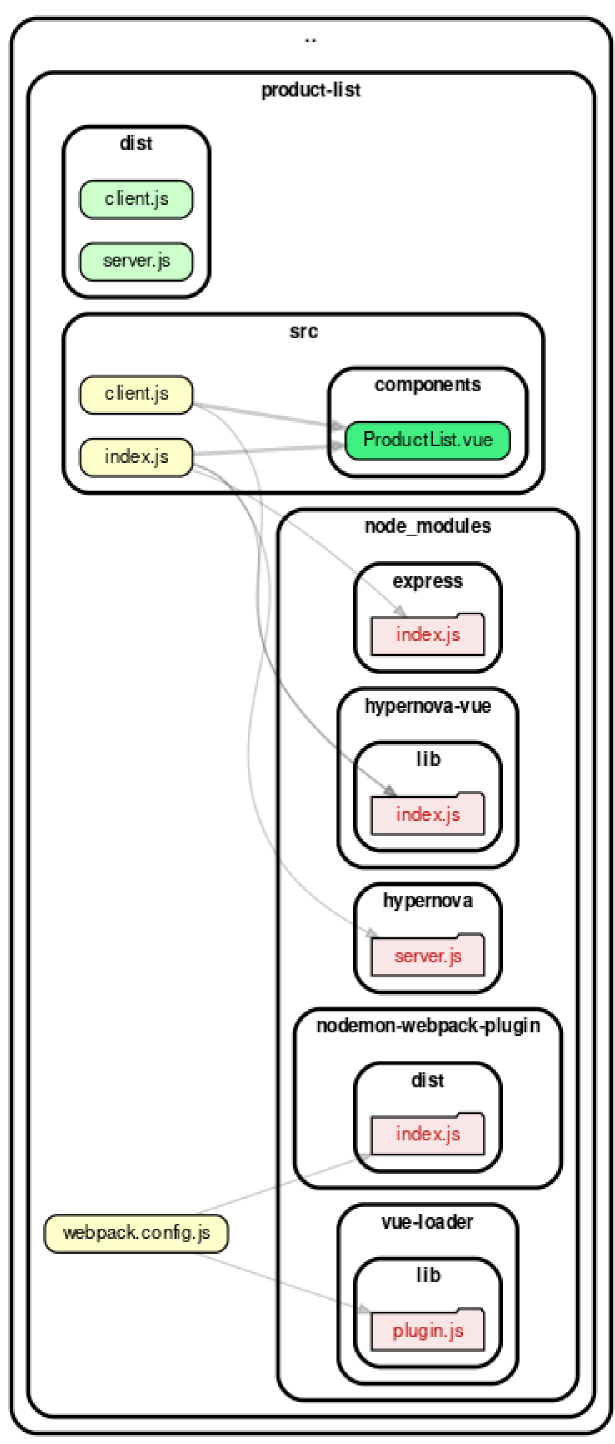


Figure 4.6: Product List dependency graph

4.4.1.2 Headless Presentation Layer

For the headless version of the case study system the following dependency graphs shown in the figures below was obtained.

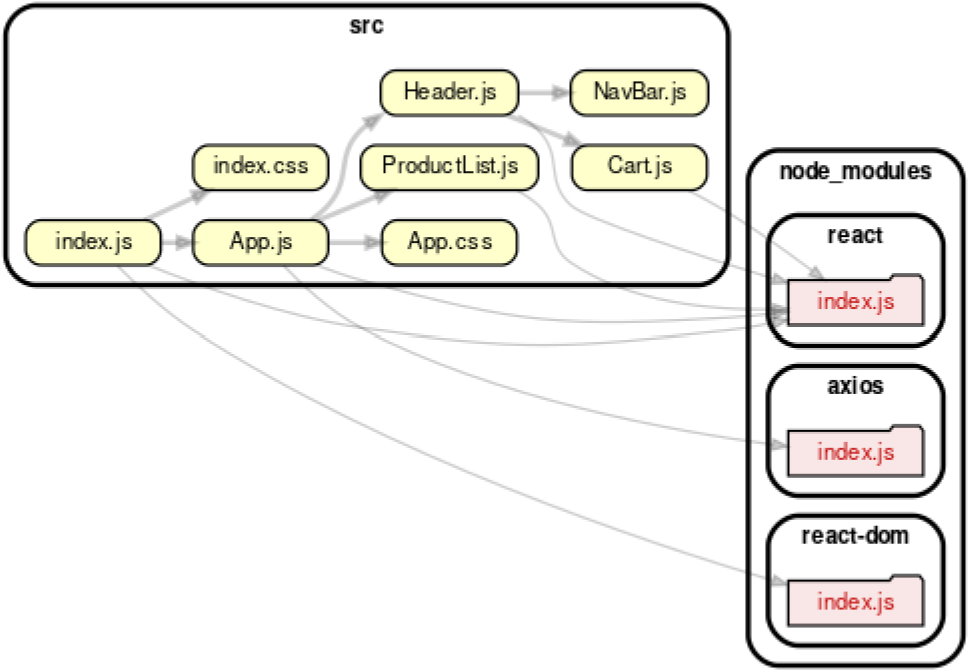


Figure 4.7: Headless frontend dependency graph

The modules in the headless implementation all share dependencies hence there are many edges across modules. The project external dependencies are also shared. This is in contrast with the micro-frontend version where the dependencies were self contained and duplicated. This duplication could lead to increased application size as multiple versions of external libraries are included. Techniques such as module federation discussed in Section 2.4.4.6 show promise in solving such problems.

4.4.2 Web Performance

The web performance metrics show how well a website is perceived by the user. Google's Lighthouse provides such metrics for the measurement of performance and was used here to evaluate both versions of the case study system. The measurement of performance reveals very similar results for both the monolithic and micro-frontend version of the case study. The

functionality of both instances are exactly the same and this coupled with the ability to apply the exact same styling to both versions make it possible to replace an existing monolithic frontend with one using the micro-frontend approach without suffering a difference in user experience.

Table 4.1: Web Performance comparison for micro-frontend and headless layer

Metric	Micro-frontend	Headless Monolith
First Contentful Paint	3.1s	5.3s
Time to Interactive	6.0s	7.5s
Speed Index	4.7s	6.0s
Total Blocking Time	10ms	70ms
Largest Contentful Paint	9.0s	7.6s
Cumulative Layout Shift	0	0.075
Overall Performance Score	63/100	55/100

The micro-frontend implementation performs better than the monolithic implementation on every metric. This shows that the micro-frontend implementation will offer superior performance especially since it is isomorphically rendered.

4.4.3 Complexity Metrics

Software complexity is a composite quality as it is made up of many other metrics. Measuring complexity helps us to empirically determine how maintainable a software system is. The complexity metrics provide insight into various modules and the associated complexity and risk with them. These metrics are based on well established measures from literature and have been discussed in section 2.3.3. The results here show much lower levels of complexity per module for the monolith than for the micro-frontend enabled version. This is attributable to the fact that each micro-frontend module is a full app and bundles the complexity for deploying that entire application with it. On the monolith side, the corresponding modules have much less bloat in them and thus have lower complexity scores. However, the cart and product list components in the monolith are essentially dumb modules since all of their logic and state is housed in the app module as shown in section 4.3.3 and in Code listing 4.4. Thus it is an equivalent comparison to look at

the average value of complexity for the micro-frontend against the value of complexity for the App component of the monolith.

4.4.3.1 Micro-frontend Presentation Layer

For the micro-frontend enabled version of the case study system Table 4.8 shows the complexity metrics which were computed.

Table 4.2: Complexity metrics for micro-frontends

Metric	Aggregator	Cart	ProductList
Total/Average Lines	105/35	31/15	28/14
Cyclomatic Complexity	1.34	2	2
Estimated Errors	0.183	0.07	0.085
Maintainability Index	73.94	76.44	76.06

To compute the overall maintainability index (MI) of the micro-frontend implementation, the average MI is computed using the individual values from each module. The average MI would be 75.48.

4.4.3.2 Headless Presentation Layer

For the headless version of the case study system Table 4.9 shows the complexity metrics which were computed.

Table 4.3: Complexity metrics for headless monolith

Metric	App	Cart	ProductList
Total/Average Lines	72/72	42/42	33/33
Cyclomatic Complexity	2	1	1
Estimated Errors	0.36	0.15	0.09
Maintainability Index	74.64	95.81	90.66

In the monolithic implementation, the cart and product list are to be considered submodules of the app module. This is because most of the logic and data for them is managed by the app module and passed to them as required. This explains the high scores they register in the metrics since they are essentially dumb modules.

For the purposes of comparison of the complexity values for the monolithic and micro-frontend versions of the case study, the maintainability index (MI) value of the app module will be compared against the average MI of the micro-frontends. The MI value for the monolith is 74.64 while the MI value for the micro-frontend is 75.48. The micro-frontend implementation performs better although they are within the same range of values.

4.4.3 Expert Evaluation

An expert evaluation was carried out to synthesise the opinions of experienced developers on the use of micro-frontends with content management systems. The questionnaire is divided into two parts. The first part required the participants to rate the level of the performance, complexity and maintainability of the micro-frontend architecture. The rating is on a scale of 1 to 5 with 5 representing a huge improvement in that category and 1 representing a huge deterioration in the category. The second part of the questionnaire contained four open ended questions and sought to discover what the experts thought about how content management systems might benefit from micro-frontend architecture, the impact of the architecture on developer experience, how micro-frontends might evolve in the future as well as any other thoughts on the architecture they intended to share.

A total of eight (8) participants consisted of seven (7) web developers with 3-5 years of professional experience as well as one (1) with over 5 years of experience. As web developers this group of respondents has proficiency in the underlying technologies for building micro-frontends and can thus offer meaningful insight as part of this evaluation.

Table 4.4: Expert complexity assessment

Complexity	Value	Frequency	Percentage
No complexity. It simplifies the existing model.	5	2	25
Some complexity, but still overall simpler than the existing model.	4	2	25
Not a lot of complexity. It is similar to existing models.	3	1	12.5
Significant complexity. It complicates the existing models	2	1	12.5
Too much complexity. It makes it difficult to use.	1	0	0
Missing	-	2	25
TOTAL	-	8	100

Table 4.10 reports the responses of the experts on how much complexity they associate with this architecture. There were two unfilled responses to this question.

Table 4.5: Expert maintainability assessment

Maintainability	Value	Frequency	Percentage
Very maintainable. It is easy to add new features and change existing ones.	5	2	25
Somewhat maintainable. Features can be added and changed but with some effort.	4	4	50
No impact on maintainability. It is similar to existing models.	3	2	25
Somewhat hard to maintain. Changing code in one aspect can lead to unexpected outcomes.	2	0	0
Too difficult to maintain.	1	0	0
Missing	-	0	0
TOTAL	-	8	100

Table 4.11 reports the responses of the experts on how much complexity they associate with this architecture. All respondents agreed that the architecture would at least match the maintainability of existing systems.

Table 4.6: Expert performance assessment

Performance	Value	Frequency	Percentage
High performance. This architecture will lead to a huge improvement in performance.	5	1	12.5
Good performance. This architecture will lead to a marginal increase in performance	4	3	37.5
No effect. This architecture will lead to no change in performance of the application.	3	2	25
Slightly worse performance. This will lead to marginal reduction in the performance of the application.	2	0	0
Low performance. This architecture will lead to a significant reduction in performance.	1	1	12.5
Missing	-	0	0
TOTAL	-	8	100

Table 4.12 reports the responses of the experts on the expected performance of micro-frontend architecture.12.5% of respondents believe that the architecture will lead to a significant reduction. The majority (37.5%) believe the architecture will lead to a marginal increase in performance.

Regarding the benefits that micro-frontends can provide to content management systems one respondent suggested that if bundle sizes could be significantly reduced in this architecture then it would lead to much faster loading times for content based applications. The developer experience could also be positively impacted by preventing the need for frontend developers to context switch between multiple domains of an application. Another response was that there could be difficulty in integrating work done by multiple teams and this could ultimately worsen the developer experience.

CHAPTER FIVE

SUMMARY, RECOMMENDATIONS AND CONCLUSION

5.1 SUMMARY

In this work a model architecture for the use of micro-frontends with content management systems was developed. To evaluate this architecture two versions of a case study system were built. The first using the model architecture and the second using a frontend monolith. The two systems were functionally equivalent and were compared using system stability, web performance and code complexity. The findings from the evaluation revealed that it is possible to overhaul an existing monolithic headless presentation layer and replace it with a micro-frontend layer without a regression in performance or user experience. Additionally, it was found that a micro-frontend presentation layer added significant complexity to individual modules compared to a monolithic approach although the dependencies for these modules were now self-contained.

This study also carried out a systematic review of literature in the field of micro-frontends and synthesised the techniques and tools currently being used to build micro-frontends in the industry and in academia.

Finally an expert evaluation was performed to determine the opinions of experts on the proposed model architecture and its implementation.

5.2 CONCLUSION

The research work in this study led to the development of a reference model for using a general purpose content management system, WordPress headlessly with a micro-frontend presentation layer. The results of evaluation showed that micro-frontend could have superior performance and maintainability than monolithic frontends. Additionally, it is possible to achieve the same visual styles in a micro-frontend as it is in a frontend monolith. A systematic review showed that client-side rendering was the most frequently used rendering method with micro-frontends. This work presented an isomorphically rendered implementation which is more suited to content based applications due to the advantages in performance and search engine discoverability. The results

of this study would be beneficial for organisations that intend to adopt micro-frontend architecture for their content based applications.

5.3 CONTRIBUTION TO KNOWLEDGE

This study has contributed to knowledge by proposing a model for the use of micro-frontend architecture with general purpose content management systems. The model architecture can be adapted to existing systems to migrate to a micro-frontend system as well as to build new applications.

This work offers an empirical evaluation of micro-frontend architecture based on maintainability and performance which has not been performed in any other work considered. The results of the evaluation show that micro-frontends can match and exceed the performance of monolithic applications. The results also provide evidence to support the claims found in literature regarding the improved maintainability of micro-frontend architecture over monolithic frontends.

Additionally, the results of the systematic review reveal industry trends in terms of tools and techniques for developing micro-frontends.

5.4 RECOMMENDATIONS

Although the metrics considered for this study in terms of stability, maintainability, performance and complexity are useful in a single observation, they are much more valuable when tracked over time. An avenue for further evaluation would be to monitor the growth of these metrics for two functionally equivalent systems to evaluate how they change with time.

Also considering the gap between the reported metrics for both systems, it would be relevant to contrast the use of micro-frontend architecture with monolithic component-based frontend applications. It may be possible to achieve a similar effect of micro-frontends by using stronger constraints on the component-based systems.

REFERENCES

Ang, R. J. (2019). Use of content management systems to address nursing workflow. *International Journal of Nursing Sciences*, 6(4), 454-459.

Asleson, R., & Schutta, N. T. (2006). Foundations of AJAX (Vol. 2, pp. 50-52). Berkeley, CA: Apress.

Attardi, J. (2020). Introduction to Netlify CMS. In *Using Gatsby and Netlify CMS* (pp. 1-12). Apress, Berkeley, CA.

Barker, D. (2016). *Web content management: Systems, features, and best practises*. O'Reilly Media, Inc..

Barrow, S. (2019, July 25). Measure Your Software Architectural Health. Lattix Inc. Retrieved from https://www.lattix.com/measure-your-software-architectural-health/

Benevolo, C., & Negri, S. (2007). Evaluation of Content Management Systems (CMS): a Supply Analysis. *Electronic Journal of Information Systems Evaluation*, 10(1).

Bhowmik, D., Roy, M., Biswas, D., Roy, S., & Roy, S. (2019). Converting and Developing Live Web Site into a Web Content Management System. In *Advances in Communication, Cloud, and Big Data* (pp. 79-87). Springer, Singapore.

Bjuhr, O., Segeljakt, K., Addibpour, M., Heiser, F., & Lagerström, R. (2017). Software architecture decoupling at Ericsson. In 2017 *IEEE International Conference on Software Architecture Workshops (ICSAW)* (pp. 259-262). IEEE.

Boiko, B. (2004), *Content Management Bible*, Hungry Minds, New York, pp5-7,9,65,81-84,104,507-508,887

Botella, P., Burgués, X., Carvallo, J. P., Franch, X., Grau, G., Marco, J., & Quer, C. (2004). ISO/IEC 9126 in practice: what do we need to know. In *Software Measurement European Forum* (Vol. 2004).

Cabot, J. (2018). WordPress: A content management system to democratize publishing. *IEEE Software*, *35*(3), 89-92. doi: 10.1109/MS.2018.2141016.

Chen, L. (2018). Microservices: architecting for continuous delivery and DevOps. In 2018 *IEEE International conference on software architecture (ICSA)* (pp. 39-397). IEEE.

Colliander-Celik, J. R. (2021, August 8). Plutt: A tool for creating type-safe and version-safe microfrontends. Retrieved from https://www.diva-portal.org/smash/get/diva2:1463748/FULLTEXT01.pdf.

da Hora, D. N., Asrese, A. S., Christophides, V., Teixeira, R., & Rossi, D. (2018). Narrowing the gap between QoS metrics and Web QoE using Above-the-fold metrics. In *International Conference on Passive and Active Network Measurement* (pp. 31-43). Springer, Cham.

Da Silva, W. O., & Farah, P. R. (2018). Characteristics And Performance Assessment Of Approaches Pre-rendering and Isomorphic Javascript As A Complement To SPA

architecture. In Proceedings of the *VII Brazilian Symposium on Software Components, Architectures, and Reuse* (pp. 63-72).

Dobrojević, M. (2018). Acceleration of Web Content and Page Structure Management Using Shortcode. In Sinteza 2018 - *International Scientific Conference on Information Technology and Data Related Research* (pp. 61-67). Singidunum University.

Domański, A., Domańska, J., & Chmiel, S. (2014). JavaScript frameworks and Ajax applications. In *International Conference on Computer Networks* (pp. 57-68). Springer, Cham.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, 195-216.

Fagan, J. C. (2002). Server-side includes made simple. *The Electronic Library*.

Fontana, F. A., Ferme, V., & Spinelli, S. (2012). Investigating the impact of code smells debt on quality code evaluation. In 2012 *Third International Workshop on Managing Technical Debt (MTD)* (pp. 15-22). IEEE.

Fouh, E., Karavirta, V., Breakiron, D. A., Hamouda, S., Hall, S., Naps, T. L., & Shaffer, C. A. (2014). Design and architecture of an interactive eTextbook–The OpenDSA system. *Science of Computer Programming*, 88, (pp. 22-40).

Gao, Q., Dey, P., & Ahammad, P. (2017). Perceived performance of top retail webpages in the wild: Insights from large-scale crowdsourcing of above-the-fold qoe. In *Proceedings of the Workshop on QoE-based Analysis and Management of Data Communication Networks* (pp. 13-18).

Garousi, V., Borg, M., & Oivo, M. (2020). Practical relevance of software engineering research: synthesizing the community's voice. *Empirical Software Engineering*, 25(3), 1687-1754.

Garousi, V., Felderer, M., & Mäntylä, M. V. (2019). Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106, 101-121.

Geers, M. (2020). *Micro Frontends in Action*. Simon and Schuster.

George N. (2015) Django CMS Plugins. In: *Beginning Django CMS*. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-1669-9_6

Grassi, G., Teixeira, R., Barakat, C., & Crovella, M. (2021). Leveraging Website Popularity Differences to Identify Performance Anomalies. In *INFOCOM 2021-IEEE International Conference on Computer Communications*.

Hariprasad, T., Vidhyagaran, G., Seenu, K., & Thirumalai, C. (2017). Software complexity analysis using halstead metrics. In 2017 *International Conference on Trends in Electronics and Informatics (ICEI)* (pp. 1109-1113). IEEE.

Harms, H., Rogowski, C., & Lo Iacono, L. (2017). Guidelines for adopting frontend architectures and patterns in microservices-based systems. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (pp. 902-907).

Hoβfeld, T., Metzger, F., & Rossi, D. (2018). Speed index: Relating the industrial standard for user perceived web performance to web qoe. In 2018 *Tenth International Conference on Quality of Multimedia Experience (QoMEX)* (pp. 1-6). IEEE.

Jackson, C. (2019). Micro frontends. Retrieved from https://martinfowler. com/articles/micro-frontends. html.

Jackson, Z. (2020, March 8). Webpack 5 Federation. A Game-changer to Javascript architecture. Retrieved from https://indepth.dev/posts/1173/webpack-5-module-federation-a-game-changer-in-javascript-architecture

Kalske, M., Mäkitalo, N., & Mikkonen, T. (2017). Challenges when moving from monolith to microservice architecture. In *International Conference on Web Engineering* (pp. 32-47). Springer, Cham.

Kumar, S. (2019). A Review on Client-Server Based Applications and Research Opportunity. *International Journal of Scientific Research*. 10. 33857-33862. 10.24327/ijrsr.2019.1007.3768.

Laumer S., Maier, C. & Weitzel, T. (2017). Information quality, user satisfaction, and the manifestation of workarounds: a qualitative and quantitative study of enterprise content management system users, *European Journal of Information Systems*, 26:4, 333-360, DOI: 10.1057/s41303-016-0029-7.

Legunsen, O., Hariri, F., Shi, A., Lu, Y., Zhang, L., & Marinov, D. (2016). An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 583-594).

Liu, Y., & Su, Y. (2020). Implementation of Graphic Plugin Loading Platform Based on Python. In *Journal of Physics: Conference Series* (Vol. 1533, No. 3, p. 032071). IOP Publishing.

MacCaw, A. (2011). *JavaScript Web Applications: JQuery Developers' Guide to Moving State to the Client*. O'Reilly Media, Inc.

Masmali, O., & Badreddin, O. (2020). Code Complexity Metrics Derived from Software Design: A Framework and Theoretical Evaluation. In *Proceedings of the Future Technologies Conference* (pp. 326-340). Springer, Cham.

McDonough III, E. F. (2000). Investigation of factors contributing to the success of cross-functional teams. *Journal of Product Innovation Management: An International Publication of the Product Development & Management Association*, 17(3), 221-235.

Mesa, O., Vieira, R., Viana, M., Durelli, V. H., Cirilo, E., Kalinowski, M., & Lucena, C. (2018). Understanding vulnerabilities in plugin-based web systems: an exploratory study of wordpress. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1* (pp. 149-159).

Mozilla Developer Network (MDN) Web Docs (2021, August). Iframe. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe.

Mozilla Developer Network (MDN) Web Docs (2021, August). Web Components. Retrieved from https://developer.mozilla.org/en-US/docs/Web/Web_Components.

Neto, G. T. G., Santos, W. B., Endo, P. T., & Fagundes, R. A. (2019). Multivocal literature reviews in software engineering: Preliminary findings from a tertiary study. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1-6). IEEE.

Niknejad, N., Ismail, W., Ghani, I., Nazari, B., Bahari M., & Bin Che Hussin, AR. (2020). Understanding Service-Oriented Architecture (SOA): A systematic literature review and directions for further investigation. *Information Systems*. 91. 101491. 0306-4379. https://doi.org/10.1016/j.is.2020.101491.

Paiva, E., Barbosa, D., Lima, R., & Albuquerque, A. (2010). Factors that influence the productivity of software developers in a developer view. In *Innovations in computing sciences and software engineering* (pp. 99-104). Springer, Dordrecht.

Pan, W., & Chai, C. (2019). Measuring software stability based on complex networks in software. *Cluster Computing*, 22(2), 2589-2598.

Pavlenko, A., Askarbekuly, N., Megha, S., & Mazzara, M. (2020). Micro-frontends: application of microservices to web front-ends. *Journal of Internet Services and Information Security*, 10(2), 49-66.

Peltonen, S., Mezzalira, L., & Taibi, D. (2021). Motivations, Benefits, and Issues for Adopting Micro-Frontends: A Multivocal Literature Review. *Information and Software Technology*, 106571.

Perez De Rosso, S., Jackson, D., Archie, M., Lao, C., & McNamara III, B. A. (2019). Declarative assembly of web applications from predefined concepts. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (pp. 79-93).

Puskaric, H., Djordjevic, A., Stefanovic, M. & Zahar Đorđević, M. (2019). DEVELOPMENT OF WEB BASED APPLICATION USING SPA ARCHITECTURE. *Proceedings on Engineering Sciences*. 1. 457-464. 10.24874/PES01.02.044.

Quadri, S. A. (2011). Developing, managing and maintaining web applications with content management systems: Drupal and Joomla as case study.

Ramakrishnan, R., & Kaur, A. (2020). An empirical comparison of predictive models for web page performance. *Information and Software Technology*, 123, 106307.

Ramalingam, E. (2016). Research Paper on Content Management Systems (CMS): Problems in the Traditional Model and Advantages of CMS in Managing Corporate Websites.

Richardson, C. (2017). Microservice architecture pattern". Retrieved from microservices.io.

Robertson, J. (2004), Definition of information management terms, Step Two Design, CM Briefing 2004-04, Published on 5 February 2004, [online], available: http://www.steptwo.com.au/papers/cmb_definition/index.html.

Salama, M., & Bahsoon, R. (2017). Analysing and modelling runtime architectural stability for self-adaptive software. *Journal of Systems and Software*, 133, 95-112.

Salama, M., Bahsoon, R., & Lago, P. (2019). Stability in software engineering: Survey of the state-of-the-art and research directions. *IEEE Transactions on Software Engineering*.

Souer, J., Urlings, T., Helms, R., & Brinkkemper, S. (2011). Engineering web information systems: A content management system-based approach. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services* (pp. 329-332).

Stadnik, W., & Nowak, Z. (2017). The impact of web pages' load time on the conversion rate of an E-commerce platform. In *International Conference on Information Systems Architecture and Technology* (pp. 336-345). Springer, Cham.

Sturtevant, D. (2017). Modular architectures make you agile in the long run. *IEEE Software*, 35(1), 104-108.

Thrivani, J., Venugopal, K. R., & Thomas, B. (2017). An efficient cloud based architecture for integrating content management systems. In 2017 *International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)* (pp. 337-342). IEEE.

Tóth, Z. (2017). Applying and evaluating halstead's complexity metrics and maintainability index for RPG. In *International Conference on Computational Science and Its Applications* (pp. 575-590). Springer, Cham.

Vaidya, S. P., Kadam, V. J., Dabhade, S. S., Dange, P. V., & Gofankar, R. B. (2013). How to Choose a Website Content Management System. In *National Conference On Emerging Trends In Academic Library, On 18th–19th January*.

Villamizar, M., Garcés, O., Castro, H., Verano Merino, M., Salamanca, L., Casallas, R., & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. 10.1109/ColumbianCC.2015.7333476.

Vogt L., Baum R., Köhler C., Meid S., Quast B., Grobe P. (2019) Using Semantic Programming for Developing a Web Content Management System for Semantic Phenotype Data. In: Auer S., Vidal ME. (eds) *Data Integration in the Life Sciences*. DILS 2018. Lecture Notes in Computer Science, vol 11371. Springer, Cham. https://doi.org/10.1007/978-3-030-06016-9_19

W3Techs. (2021, April, 8th). Historical yearly trends in the usage statistics of content management systems, Retrieved from https://w3techs.com/technologies/history_overview/content_management/all/y

Wang, D., Yang, D.M., Zhou, H., Wang Y., Hong, D., Dong, Q. & Song, S. (2020). A Novel Application of Educational Management Information System based on Micro Frontends,

Procedia Computer Science, 176, 1567-1576, 1877-0509, https://doi.org/10.1016/j.procs.2020.09.168.

Yang C, Liu C, Su Z. (2019). Research and application of micro frontends, *IOP Conference Series: Materials Science and Engineering* 490. 062082.

APPENDIX A

SOURCE CODE

ROOT MICROFRONTEND - index.js

```
import "./App.css";
import Header from "./Header";
import ProductList from "./ProductList";
import { useEffect, useState } from "react";
import axios from "axios";

function App() {
 const [products, setProducts] = useState([]);
 const [itemsSelected, setItemsSelected] = useState([]);

 const selectProduct = (item) => {
   setItemsSelected((prevState) => {
     const existingIndex = prevState.findIndex(
       (product) => product.id === item.id
     );
     if (existingIndex === -1) {
       prevState.push({ ...item, quantity: 1 });
     } else {
       prevState[existingIndex].quantity++;
     }
     return prevState;
   });
 };

 useEffect(() => {
   async function fetchData() {
     const rawProducts = (
       await axios.get(
         "https://services.etin.space/notes/wp-json/wc/v3/products",
         {
           auth: {
             username: "USERNAME",
             password: "PASSWORD",
           },
         }
       )
     ).data;
     const products = rawProducts.map((product) => ({
       id: product.id,
       title: product.name,
       imageUrl: product.images?.[0]?.src ?? "https://via.placeholder.com/150",
       price: product.price,
     }));
     setProducts(products);
   }
   fetchData();
 }, []);
 return (
   <div className="App">
     <Header
       title="My Store"
       links={[
         {
           url: "/",
```

```
          text: "Home",
        },
      ]}
      cartItems={itemsSelected}
    />
    <div className="bg-black text-white flex items-center justify-center h-96">
      <div className="w-3/4">
        <h1 className="text-4xl">Welcome to my store</h1>
        <p>This store is built headlessly with WordPress</p>
      </div>
    </div>
    <ProductList onSelect={selectProduct} products={products} />
    <footer className="bg-black text-white text-center py-3">® My Headless Store
2021</footer>
  </div>
 );
}

export default App;
```

ROOT MICROFRONTEND - layout.js

```
const axios = require('axios')

module.exports.getLayout = () => {
 return axios.post('http://0.0.0.0:3031/batch', {
   header: {
     name: 'Header',
     data: {
       title: 'My Store',
       links: [
         {
           url: '/',
           text: 'Home'
         },
       ]
     }
   }
 })
   .then(({ data }) => {
     return data.results.header
   })
}
```

ROOT MICROFRONTEND - content.js

```
const axios = require("axios");

module.exports.getContent = async () => {
 try {
   const rawProducts = (await axios.get(
     "https://services.etin.space/notes/wp-json/wc/v3/products",
     {
       auth: {
         username: "USERNAME",
         password: "PASSWORD",
       },
     }
```

```
    )).data;
    const products = rawProducts.map((product) => ({
      id: product.id,
      title: product.name,
      imageUrl: product.images?.[0]?.src ?? "https://via.placeholder.com/150",
      price: product.price,
    }))
    return axios
      .post("http://0.0.0.0:3030/batch", {
        content: {
          name: "ProductList",
          data: {
            title: "Products",
            items: products,
          },
        },
      })
      .then(({ data }) => {
        return data.results.content;
      });
  } catch (err) {
    console.log(err);
  }
};
```

CART MICROFRONTEND - index.js

```
import express from 'express';
import path from 'path';
import hypernova from 'hypernova/server';
import { renderReact } from 'hypernova-react';

import Header from './components/Header';

hypernova({
 devMode: true,
 getComponent(name) {
   if (name === 'Header') {
     return renderReact(name, Header);
   }

   return null;
 },
 port: process.env.PORT || 3031,
 createApplication() {
   const app = express();

   app.use(express.static(path.join(process.cwd(), 'dist')));

   return app;
 },
});
```

CART MICROFRONTEND - client.js

```
import { renderReact } from 'hypernova-react';
```

```
import Header from './components/Header';

renderReact('Header', Header);
```

CART MICROFRONTEND - client.js

```
import { renderReact } from 'hypernova-react';

import Header from './components/Header';

renderReact('Header', Header);
```

CART MICROFRONTEND - Cart.jsx

```
import * as React from "react";

export default function Cart({ products, ...props }) {
 return (
   <div
     className="fixed top-14 right-0 bg-white border max-w-xs px-3 pt-3 pb-9 shadow-
sm rounded-sm"
     {...props}
   >
     <h3 className="text-xl">Cart</h3>
     {products.map((product, index) => (
       <div key={index} className="border-b mb-3">
         <div className="grid grid-cols-4 items-center py-3">
           <div className="col-span-1">
             <img
               className="h-12 block mx-auto"
               src={product.imageUrl}
               alt=""
             />
           </div>
           <div className="col-span-3">{product.title}</div>
         </div>
         <div className="flex justify-between">
           <span>₦{parseInt(product.price) * product.quantity}</span>
           <span>{product.quantity}</span>
         </div>
       </div>
     ))}
     <div className="font-semibold flex justify-between">
       <span>
         Total: ₦
         {products.reduce(
           (sum, product) => sum + parseInt(product.price) * product.quantity,
           0
         )}
       </span>
       <span>
         {products.reduce((sum, product) => sum + product.quantity, 0)}
       </span>
     </div>
   </div>
 );
}
```

CART MICROFRONTEND - Header.jsx

```
import React from "react";
import PropTypes from "prop-types";

import NavBar from "./NavBar";
import Cart from "./Cart";

class Header extends React.Component {
 constructor(props) {
   super(props);
   this.state = {
     itemsSelected: [],
     cartIsOpen: false,
   };
   this.itemSelected = this.itemSelected.bind(this);
   this.toggleCart = this.toggleCart.bind(this);
 }

 componentDidMount() {
   document.addEventListener("itemSelected", this.itemSelected);
 }

 componentWillUnmount() {
   document.removeEventListener("itemSelected", this.itemSelected);
 }

 itemSelected({ detail: item }) {
   this.setState((prevState) => {
     const existingIndex = prevState.itemsSelected.findIndex((product) => product.id
=== item.id);
     if (existingIndex === -1) {
       prevState.itemsSelected.push({...item, quantity: 1})
     } else {
       prevState.itemsSelected[existingIndex].quantity++;
     }
     return { ...prevState };
   });
 }

 toggleCart() {
   this.setState((prevState) => ({
     ...prevState,
     cartIsOpen: !prevState.cartIsOpen,
   }));
 }

 render() {
   const { title, links } = this.props;
   const { itemsSelected } = this.state;
   return (
     <React.Fragment>
       <header className="header w-full fixed top-0">
         <div className="header__brand">{title}</div>
         <span className="header__space" />
         <span onClick={this.toggleCart}>Cart: {itemsSelected.reduce((sum, item) =>
sum + item.quantity, 0)}</span>
         <NavBar links={links} />
       </header>
       {this.state.cartIsOpen && (
         <div className="relative ml-auto">
```

```
          <Cart products={itemsSelected} />
        </div>
      )}
    </React.Fragment>
  );
 }
}

Header.propTypes = {
 title: PropTypes.string.isRequired,
 links: NavBar.propTypes.links,
};

Header.defaultProps = {
 links: [],
};

export default Header;
```

CART MICROFRONTEND - NavBar.jsx

```
import React from 'react';
import PropTypes from 'prop-types';

const NavBar = ({ links }) => (
 <nav className="navbar">
   <ul>
     { links.map(({ url, text }) => <li key={url} className="navbar__item"><a
href={url}>{text}</a></li>) }
   </ul>
 </nav>
);

NavBar.propTypes = {
 links: PropTypes.arrayOf(PropTypes.shape({
   url: PropTypes.string.isRequired,
   text: PropTypes.string.isRequired,
 })),
};

NavBar.defaultProps = {
 links: [],
};

export default NavBar;
```

PRODUCT LIST MICROFRONTEND - index.js

```
import hypernova from 'hypernova/server'
import { renderVue, Vue } from 'hypernova-vue'
import express from 'express'
import path from 'path'
```

```
import ProductList from './components/ProductList.vue'

hypernova({
 devMode: true,
 getComponent (name, context) {
   if (name === 'ProductList') {
     return renderVue(name, Vue.extend(ProductList))
   }
 },
 port: process.env.PORT || 3030,

 createApplication () {
   const app = express()

   app.use(express.static(path.join(process.cwd(), 'dist')))

   return app
 }
})
```

PRODUCT LIST MICROFRONTEND - client.js

```
import { renderVue, Vue } from "hypernova-vue";
import ProductList from "./components/ProductList.vue";

renderVue("ProductList", Vue.extend(ProductList));
```

PRODUCT LIST MICROFRONTEND - ProductList.vue

```
<template>
 <div class="product-list">
   <h2 class="text-3xl font-semibold">{{ title }}</h2>
   <ul>
     <li
       v-for="(item, idx) in items"
       :key="idx"
       class="product-item flex flex-col items-center justify-center text-center"
     >
       <img :src="item.imageUrl" class="h-52 block mx-0" />
       <h4 class="text-xl font-semibold mb-3">{{ item.title }}</h4>
       <p class="text-base">₦{{ item.price }}</p>
       <button
         class="p-3 bg-yellow-200 font-semibold text-black"
         @click="select(item)"
       >
         Add To Cart
       </button>
     </li>
   </ul>
 </div>
</template>

<script>
export default {
 props: {
   title: {
     type: String,
     required: true
   },
```

```
    items: {
      type: Array,
      default: () => []
    }
  },
  methods: {
    select(item) {
      const event = new CustomEvent("itemSelected", { detail: item });
      document.dispatchEvent(event);
    }
  }
};
</script>
```

APPENDIX B

APPLICATION PROGRAMMING INTERFACE RESPONSES

WooCommerce All Products Response


```
[
  {
    "id": 615,
    "name": "Hair Moisturizer",
    "slug": "hair-moisturizer",
    "permalink": "https:\/\/services.etin.space\/notes\/product\/hair-
moisturizer\/",
    "date_created": "2021-08-25T12:50:15",
    "date_created_gmt": "2021-08-25T12:50:15",
    "date_modified": "2021-09-10T12:25:38",
    "date_modified_gmt": "2021-09-10T12:25:38",
    "type": "simple",
    "status": "publish",
    "featured": false,
    "catalog_visibility": "visible",
    "description": "",
    "short_description": "",
    "sku": "",
    "price": "300",
    "regular_price": "300",
    "sale_price": "",
    "date_on_sale_from": null,
    "date_on_sale_from_gmt": null,
    "date_on_sale_to": null,
    "date_on_sale_to_gmt": null,
    "on_sale": false,
    "purchasable": true,
    "total_sales": 0,
    "virtual": false,
    "downloadable": false,
    "downloads": [],
    "download_limit": -1,
    "download_expiry": -1,
    "external_url": "",
    "button_text": "",
    "tax_status": "taxable",
    "tax_class": "",
    "manage_stock": false,
    "stock_quantity": null,
    "backorders": "no",
    "backorders_allowed": false,
    "backordered": false,
    "low_stock_amount": null,
    "sold_individually": false,
    "weight": "",
    "dimensions": {
      "length": "",
      "width": "",
      "height": ""
    },
    "shipping_required": true,
    "shipping_taxable": true,
    "shipping_class": "",
    "shipping_class_id": 0,
    "reviews_allowed": true,
```

```
    "average_rating": "0.00",
    "rating_count": 0,
    "upsell_ids": [],
    "cross_sell_ids": [],
    "parent_id": 0,
    "purchase_note": "",
    "categories": [
      {
        "id": 20,
        "name": "Uncategorized",
        "slug": "uncategorized"
      }
    ],
    "tags": [],
    "images": [
      {
        "id": 616,
        "date_created": "2021-08-25T12:51:13",
        "date_created_gmt": "2021-08-25T12:51:13",
        "date_modified": "2021-08-25T12:51:13",
        "date_modified_gmt": "2021-08-25T12:51:13",
        "src": "https:\/\/services.etin.space\/notes\/wp-
content\/uploads\/2021\/08\/moisturizer-voor-droog-haar.jpeg",
        "name": "moisturizer-voor-droog-haar",
        "alt": ""
      }
    ],
    "attributes": [],
    "default_attributes": [],
    "variations": [],
    "grouped_products": [],
    "menu_order": 0,
    "price_html": "<span class=\"woocommerce-Price-amount amount\"><bdi><span
class=\"woocommerce-Price-currencySymbol\">&#36;<\/span>300.00<\/bdi><\/span>",
    "related_ids": [
      614,
      612,
      608,
      605,
      613
    ],
    "meta_data": [],
    "stock_status": "instock",
    "_links": {
      "self": [
        {
          "href": "https:\/\/services.etin.space\/notes\/wp-
json\/wc\/v3\/products\/615"
        }
      ],
      "collection": [
        {
          "href": "https:\/\/services.etin.space\/notes\/wp-json\/wc\/v3\/products"
        }
      ]
    }
  },
  {
    "id": 614,
    "name": "Ear Fibre",
    "slug": "ear-fibre",
    "permalink": "https:\/\/services.etin.space\/notes\/product\/ear-fibre\/",
    "date_created": "2021-08-25T12:47:21",
```

```
"date_created_gmt": "2021-08-25T12:47:21",
"date_modified": "2021-09-10T12:25:48",
"date_modified_gmt": "2021-09-10T12:25:48",
"type": "simple",
"status": "publish",
"featured": false,
"catalog_visibility": "visible",
"description": "",
"short_description": "",
"sku": "",
"price": "250",
"regular_price": "250",
"sale_price": "",
"date_on_sale_from": null,
"date_on_sale_from_gmt": null,
"date_on_sale_to": null,
"date_on_sale_to_gmt": null,
"on_sale": false,
"purchasable": true,
"total_sales": 0,
"virtual": false,
"downloadable": false,
"downloads": [],
"download_limit": -1,
"download_expiry": -1,
"external_url": "",
"button_text": "",
"tax_status": "taxable",
"tax_class": "",
"manage_stock": false,
"stock_quantity": null,
"backorders": "no",
"backorders_allowed": false,
"backordered": false,
"low_stock_amount": null,
"sold_individually": false,
"weight": "",
"dimensions": {
  "length": "",
  "width": "",
  "height": ""
},
"shipping_required": true,
"shipping_taxable": true,
"shipping_class": "",
"shipping_class_id": 0,
"reviews_allowed": true,
"average_rating": "0.00",
"rating_count": 0,
"upsell_ids": [],
"cross_sell_ids": [],
"parent_id": 0,
"purchase_note": "",
"categories": [
  {
    "id": 20,
    "name": "Uncategorized",
    "slug": "uncategorized"
  }
],
"tags": [],
"images": [
  {
```

```
        "id": 620,
        "date_created": "2021-08-25T12:51:19",
        "date_created_gmt": "2021-08-25T12:51:19",
        "date_modified": "2021-08-25T12:51:19",
        "date_modified_gmt": "2021-08-25T12:51:19",
        "src": "https:\/\/services.etin.space\/notes\/wp-
content\/uploads\/2021\/08\/snapshotimagehandler_339994122.jpeg",
        "name": "snapshotimagehandler_339994122",
        "alt": ""
      }
    ],
    "attributes": [],
    "default_attributes": [],
    "variations": [],
    "grouped_products": [],
    "menu_order": 0,
    "price_html": "<span class=\"woocommerce-Price-amount amount\"><bdi><span
class=\"woocommerce-Price-currencySymbol\">&#36;<\/span>250.00<\/bdi><\/span>",
    "related_ids": [
      605,
      615,
      608,
      613,
      612
    ],
    "meta_data": [],
    "stock_status": "instock",
    "_links": {
      "self": [
        {
          "href": "https:\/\/services.etin.space\/notes\/wp-
json\/wc\/v3\/products\/614"
        }
      ],
      "collection": [
        {
          "href": "https:\/\/services.etin.space\/notes\/wp-json\/wc\/v3\/products"
        }
      ]
    }
  },
  {
    "id": 613,
    "name": "Face Wash",
    "slug": "face-wash",
    "permalink": "https:\/\/services.etin.space\/notes\/product\/face-wash\/",
    "date_created": "2021-08-25T12:46:57",
    "date_created_gmt": "2021-08-25T12:46:57",
    "date_modified": "2021-09-10T12:26:15",
    "date_modified_gmt": "2021-09-10T12:26:15",
    "type": "simple",
    "status": "publish",
    "featured": false,
    "catalog_visibility": "visible",
    "description": "",
    "short_description": "",
    "sku": "",
    "price": "900",
    "regular_price": "900",
    "sale_price": "",
    "date_on_sale_from": null,
    "date_on_sale_from_gmt": null,
    "date_on_sale_to": null,
```

```
    "date_on_sale_to_gmt": null,
    "on_sale": false,
    "purchasable": true,
    "total_sales": 0,
    "virtual": false,
    "downloadable": false,
    "downloads": [],
    "download_limit": -1,
    "download_expiry": -1,
    "external_url": "",
    "button_text": "",
    "tax_status": "taxable",
    "tax_class": "",
    "manage_stock": false,
    "stock_quantity": null,
    "backorders": "no",
    "backorders_allowed": false,
    "backordered": false,
    "low_stock_amount": null,
    "sold_individually": false,
    "weight": "",
    "dimensions": {
      "length": "",
      "width": "",
      "height": ""
    },
    "shipping_required": true,
    "shipping_taxable": true,
    "shipping_class": "",
    "shipping_class_id": 0,
    "reviews_allowed": true,
    "average_rating": "0.00",
    "rating_count": 0,
    "upsell_ids": [],
    "cross_sell_ids": [],
    "parent_id": 0,
    "purchase_note": "",
    "categories": [
      {
        "id": 20,
        "name": "Uncategorized",
        "slug": "uncategorized"
      }
    ],
    "tags": [],
    "images": [
      {
        "id": 619,
        "date_created": "2021-08-25T12:51:17",
        "date_created_gmt": "2021-08-25T12:51:17",
        "date_modified": "2021-08-25T12:51:17",
        "date_modified_gmt": "2021-08-25T12:51:17",
        "src": "https:\/\/services.etin.space\/notes\/wp-
content\/uploads\/2021\/08\/download.jpeg",
        "name": "download",
        "alt": ""
      }
    ],
    "attributes": [],
    "default_attributes": [],
    "variations": [],
    "grouped_products": [],
    "menu_order": 0,
```

```
    "price_html": "<span class=\"woocommerce-Price-amount amount\">&lt;bdi&gt;<span
class=\"woocommerce-Price-currencySymbol\">&#36;<\/span>900.00<\/bdi><\/span>",
    "related_ids": [
      605,
      614,
      608,
      612,
      615
    ],
    "meta_data": [],
    "stock_status": "instock",
    "_links": {
      "self": [
        {
          "href": "https:\/\/services.etin.space\/notes\/wp-
json\/wc\/v3\/products\/613"
        }
      ],
      "collection": [
        {
          "href": "https:\/\/services.etin.space\/notes\/wp-json\/wc\/v3\/products"
        }
      ]
    }
  },
  {
    "id": 612,
    "name": "Hand Sanitizer",
    "slug": "hand-sanitizer",
    "permalink": "https:\/\/services.etin.space\/notes\/product\/hand-sanitizer\/",
    "date_created": "2021-08-25T12:45:54",
    "date_created_gmt": "2021-08-25T12:45:54",
    "date_modified": "2021-09-10T12:25:59",
    "date_modified_gmt": "2021-09-10T12:25:59",
    "type": "simple",
    "status": "publish",
    "featured": false,
    "catalog_visibility": "visible",
    "description": "",
    "short_description": "",
    "sku": "",
    "price": "650",
    "regular_price": "650",
    "sale_price": "",
    "date_on_sale_from": null,
    "date_on_sale_from_gmt": null,
    "date_on_sale_to": null,
    "date_on_sale_to_gmt": null,
    "on_sale": false,
    "purchasable": true,
    "total_sales": 0,
    "virtual": false,
    "downloadable": false,
    "downloads": [],
    "download_limit": -1,
    "download_expiry": -1,
    "external_url": "",
    "button_text": "",
    "tax_status": "taxable",
    "tax_class": "",
    "manage_stock": false,
    "stock_quantity": null,
    "backorders": "no",
```

```
    "backorders_allowed": false,
    "backordered": false,
    "low_stock_amount": null,
    "sold_individually": false,
    "weight": "",
    "dimensions": {
      "length": "",
      "width": "",
      "height": ""
    },
    "shipping_required": true,
    "shipping_taxable": true,
    "shipping_class": "",
    "shipping_class_id": 0,
    "reviews_allowed": true,
    "average_rating": "0.00",
    "rating_count": 0,
    "upsell_ids": [],
    "cross_sell_ids": [],
    "parent_id": 0,
    "purchase_note": "",
    "categories": [
      {
        "id": 20,
        "name": "Uncategorized",
        "slug": "uncategorized"
      }
    ],
    "tags": [],
    "images": [
      {
        "id": 618,
        "date_created": "2021-08-25T12:51:16",
        "date_created_gmt": "2021-08-25T12:51:16",
        "date_modified": "2021-08-25T12:51:16",
        "date_modified_gmt": "2021-08-25T12:51:16",
        "src": "https:\/\/services.etin.space\/notes\/wp-
content\/uploads\/2021\/08\/meraki-hand-sanitizer-gel-with-80-alcohol-490-ml-
309770005.jpeg",
        "name": "meraki-hand-sanitizer-gel-with-80-alcohol-490-ml-309770005",
        "alt": ""
      }
    ],
    "attributes": [],
    "default_attributes": [],
    "variations": [],
    "grouped_products": [],
    "menu_order": 0,
    "price_html": "<span class=\"woocommerce-Price-amount amount\"><bdi><span
class=\"woocommerce-Price-currencySymbol\">&#36;<\/span>650.00<\/bdi><\/span>",
    "related_ids": [
      615,
      605,
      608,
      614,
      613
    ],
    "meta_data": [],
    "stock_status": "instock",
    "_links": {
      "self": [
        {
          "href": "https:\/\/services.etin.space\/notes\/wp-
```

```
json\/wc\/v3\/products\/612"
        }
      ],
      "collection": [
        {
          "href": "https:\/\/services.etin.space\/notes\/wp-json\/wc\/v3\/products"
        }
      ]
    }
  },
  {
    "id": 608,
    "name": "Body Lotion",
    "slug": "body-lotion",
    "permalink": "https:\/\/services.etin.space\/notes\/product\/body-lotion\/",
    "date_created": "2021-07-12T12:02:46",
    "date_created_gmt": "2021-07-12T12:02:46",
    "date_modified": "2021-09-10T12:26:26",
    "date_modified_gmt": "2021-09-10T12:26:26",
    "type": "simple",
    "status": "publish",
    "featured": false,
    "catalog_visibility": "visible",
    "description": "",
    "short_description": "",
    "sku": "",
    "price": "720",
    "regular_price": "720",
    "sale_price": "",
    "date_on_sale_from": null,
    "date_on_sale_from_gmt": null,
    "date_on_sale_to": null,
    "date_on_sale_to_gmt": null,
    "on_sale": false,
    "purchasable": true,
    "total_sales": 0,
    "virtual": false,
    "downloadable": false,
    "downloads": [],
    "download_limit": -1,
    "download_expiry": -1,
    "external_url": "",
    "button_text": "",
    "tax_status": "taxable",
    "tax_class": "",
    "manage_stock": false,
    "stock_quantity": null,
    "backorders": "no",
    "backorders_allowed": false,
    "backordered": false,
    "low_stock_amount": null,
    "sold_individually": false,
    "weight": "",
    "dimensions": {
      "length": "",
      "width": "",
      "height": ""
    },
    "shipping_required": true,
    "shipping_taxable": true,
    "shipping_class": "",
    "shipping_class_id": 0,
    "reviews_allowed": true,
```

```
    "average_rating": "0.00",
    "rating_count": 0,
    "upsell_ids": [],
    "cross_sell_ids": [],
    "parent_id": 0,
    "purchase_note": "",
    "categories": [
      {
        "id": 20,
        "name": "Uncategorized",
        "slug": "uncategorized"
      }
    ],
    "tags": [],
    "images": [
      {
        "id": 617,
        "date_created": "2021-08-25T12:51:15",
        "date_created_gmt": "2021-08-25T12:51:15",
        "date_modified": "2021-08-25T12:51:15",
        "date_modified_gmt": "2021-08-25T12:51:15",
        "src": "https:\/\/services.etin.space\/notes\/wp-
content\/uploads\/2021\/08\/aveeno-daily-moisturising-body-lotion-lavender-300ml-
2.jpeg",
        "name": "aveeno-daily-moisturising-body-lotion-lavender-300ml-2",
        "alt": ""
      }
    ],
    "attributes": [],
    "default_attributes": [],
    "variations": [],
    "grouped_products": [],
    "menu_order": 0,
    "price_html": "<span class=\"woocommerce-Price-amount amount\"><bdi><span
class=\"woocommerce-Price-currencySymbol\">&#36;<\/span>720.00<\/bdi><\/span>",
    "related_ids": [
      614,
      615,
      605,
      613,
      612
    ],
    "meta_data": [],
    "stock_status": "instock",
    "_links": {
      "self": [
        {
          "href": "https:\/\/services.etin.space\/notes\/wp-
json\/wc\/v3\/products\/608"
        }
      ],
      "collection": [
        {
          "href": "https:\/\/services.etin.space\/notes\/wp-json\/wc\/v3\/products"
        }
      ]
    }
  },
  {
    "id": 605,
    "name": "Moist Towellettes (Pack of 6)",
    "slug": "moist-towellettes-pack-of-6",
    "permalink": "https:\/\/services.etin.space\/notes\/product\/moist-towellettes-
```

```
pack-of-6\/",
    "date_created": "2021-07-12T11:46:57",
    "date_created_gmt": "2021-07-12T11:46:57",
    "date_modified": "2021-09-10T12:26:38",
    "date_modified_gmt": "2021-09-10T12:26:38",
    "type": "simple",
    "status": "publish",
    "featured": false,
    "catalog_visibility": "visible",
    "description": "",
    "short_description": "",
    "sku": "",
    "price": "150",
    "regular_price": "150",
    "sale_price": "",
    "date_on_sale_from": null,
    "date_on_sale_from_gmt": null,
    "date_on_sale_to": null,
    "date_on_sale_to_gmt": null,
    "on_sale": false,
    "purchasable": true,
    "total_sales": 0,
    "virtual": false,
    "downloadable": false,
    "downloads": [],
    "download_limit": -1,
    "download_expiry": -1,
    "external_url": "",
    "button_text": "",
    "tax_status": "taxable",
    "tax_class": "",
    "manage_stock": false,
    "stock_quantity": null,
    "backorders": "no",
    "backorders_allowed": false,
    "backordered": false,
    "low_stock_amount": null,
    "sold_individually": false,
    "weight": "",
    "dimensions": {
      "length": "",
      "width": "",
      "height": ""
    },
    "shipping_required": true,
    "shipping_taxable": true,
    "shipping_class": "",
    "shipping_class_id": 0,
    "reviews_allowed": true,
    "average_rating": "0.00",
    "rating_count": 0,
    "upsell_ids": [],
    "cross_sell_ids": [],
    "parent_id": 0,
    "purchase_note": "",
    "categories": [
      {
        "id": 20,
        "name": "Uncategorized",
        "slug": "uncategorized"
      }
    ],
    "tags": [],
```

```
    "images": [
      {
        "id": 606,
        "date_created": "2021-07-12T11:46:44",
        "date_created_gmt": "2021-07-12T11:46:44",
        "date_modified": "2021-07-12T11:46:44",
        "date_modified_gmt": "2021-07-12T11:46:44",
        "src": "https:\/\/services.etin.space\/notes\/wp-
content\/uploads\/2021\/07\/56108_1604062874.jpeg",
        "name": "56108_1604062874",
        "alt": ""
      }
    ],
    "attributes": [],
    "default_attributes": [],
    "variations": [],
    "grouped_products": [],
    "menu_order": 0,
    "price_html": "<span class=\"woocommerce-Price-amount amount\"><bdi><span
class=\"woocommerce-Price-currencySymbol\">&#36;<\/span>150.00<\/bdi><\/span>",
    "related_ids": [
      614,
      615,
      612,
      608,
      613
    ],
    "meta_data": [],
    "stock_status": "instock",
    "_links": {
      "self": [
        {
          "href": "https:\/\/services.etin.space\/notes\/wp-
json\/wc\/v3\/products\/605"
        }
      ],
      "collection": [
        {
          "href": "https:\/\/services.etin.space\/notes\/wp-json\/wc\/v3\/products"
        }
      ]
    }
  }
]
```