



VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Xuan-An Cao

HEADLESS CMS AND QWIK FRAMEWORK

and their practicalities in the future of application development

School of Technology
2023

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to everybody who has helped me in the process of writing this thesis.

First and foremost, I would like to thank my advisor, Mr. Kenneth Norrgård for his dedicated guidance as well as support and feedback throughout the entire research and development phase. This thesis could not have been completed without his crucial direction and invaluable expertise.

I also want to extend my thanks to my leaders and colleagues at BraveBits Vietnam where I have spent nearly a year working as a Full Stack developer. Their insightful contribution and constructive criticism have broadened my knowledge and challenged me to be a more well-rounded software engineer.

Last but not least, I cannot express how thankful I am to have my family and my high school friends to have always been there by my side, through all the ups and downs, all the trials and tribulations. Their unconditional love, encouragement and support have been the greatest motivation for me to overcome all the obstacles that occurred.

This thesis is the collective achievement of all these individuals and I am grateful for their contribution, intellectually or mentally.

Xuan-An Cao,

Hanoi, Vietnam, 2023

ABSTRACT

Author	Xuan-An Cao
Title	Headless CMS and Qwik Framework
Year	2023
Language	English
Pages	123
Supervisor	Kenneth Norrgård

Web development is an ever-changing field and the need for more flexible, scalable and future-proof methodologies is increasing rapidly. This thesis focuses on the two recently developed approaches on web development, Headless CMS and Qwik framework, and discusses how a shift towards these two technologies addresses the drawbacks and limitations of current traditional web building techniques, and the roles they will play in the future of this field.

The thesis consists of two phases: research and development. The research phase aims to point out the flaws of current web architecture and determine the roles of Headless CMS and Qwik in solving the issue. It does that by means of reviewing and analyzing released documents, such as previous studies, official documentations and developer interviews, as well as studying real-world cases of applying the two emerging technologies. The development phase demonstrates an actual implementation of a web application utilizing both Headless CMS and Qwik, presenting all the vital steps of building an application, from the base architecture to deployment and performance testing, to practically prove the indication made in the research part.

The findings manage to comprehensively highlight the advantages of Headless CMS and Qwik, as they provide the utmost flexibility by enabling faster web development and better content management, which is crucial to omnichannel strategies and user experiences. However, the study also identifies several drawbacks such as compatibility issues, limited community and documentation, as well as a difficult learning curve. The product implemented in the development phase achieves its aims by providing a well-functional web application with high performance score on the Lighthouse measurement tool.

Keywords	Headless, Qwik, web applications, content and framework
----------	---

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

ABSTRACT

TABLE OF CONTENTS.....	4
LIST OF FIGURES.....	7
LIST OF TABLES.....	10
LIST OF CODE SNIPPETS	11
LIST OF ABBREVIATIONS	13
1. INTRODUCTION.....	15
1.1. Background and Context	15
1.2. Research Questions and Objectives	15
1.3. Methodologies	16
1.4. Significance and Contribution	17
2. THEORETICAL BACKGROUND	19
2.1. Headless CMS	19
2.1.1. The Early Era of World Wide Web	19
2.1.2. The Rise and Fall of Monolithic CMS	21
2.1.3. The Emergence of Headless CMS.....	25
2.1.4. Case Study: Contentful.....	30
2.2. Qwik Framework	33
2.2.1. The Early Jistory of JavaScript	34
2.2.2. MV* Architecture and SPA Frameworks.....	37
2.2.3. Metaframeworks and the Problem of Hydrations.....	41
2.2.4. The Qwik Framework	43

2.2.5.	Case Study: Builder.io.	47
3.	FILMMASH APPLICATION OVERVIEW	49
3.1.	Background, Motivation, and Objectives	49
3.2.	Requirement Analysis.....	51
3.3.	Core Functionality and User Flows.....	52
3.3.1.	Elo's Rating Algorithm.....	52
3.3.2.	User Flow	54
3.3.3.	Adding and Retrieving Content from Contentful.....	56
3.4.	System Infrastructure and Relevant Technologies	61
4.	FILMMASH APPLICATION IMPLEMENTATION	65
4.1.	Database Design	65
4.2.	Backend Implementation	67
4.2.1.	Backend Code Structure	67
4.2.2.	Data Models	70
4.2.3.	Routers and Controllers: Gallery Model	73
4.2.4.	Routers and Controllers: Film Model.....	78
4.2.5.	Routers and Controllers: User Model	82
4.2.6.	Utilities	86
4.2.7.	Backend Conclusion	88
4.3.	Frontend Implementation	90
4.3.1.	Frontend Code Structure	90
4.3.2.	Root, Global styling, and Entry Giles.....	92
4.3.3.	Routes	93
4.3.4.	Initialized Components: Homepage and Header	95

4.3.5.	Form Components: Login, Sign Up and Create Gallery	97
4.3.6.	Gallery Components: Overview, Mash and Ranking	100
4.3.7.	Styled Components	104
5.	PERFORMANCE EVALUATION	105
5.1.	Contentful-based Management System	105
5.2.	Google Lighthouse Performance Metrics	107
6.	CONCLUSION	111
6.1.	Thesis Summary	111
6.2.	Key Findings.....	112
6.3.	Future Applicability	113
	REFERENCES	115
	Articles	115
	Books and Research Papers	122

LIST OF FIGURES

Figure 1. The share of the population and the total number of people using the internet (Roser, 2018).....	21
Figure 2. The functioning diagram of monolithic CMS as a coupled system (Yermolenko & Golchevskiy, 2021).....	23
Figure 3. A comparison of desktop and mobile market share worldwide from January 2009 to January 2023 (StatCounter, 2023)	24
Figure 4. The evolution of omnichannel (Nguyen, 2018)	25
Figure 5. The architectures of Monolithic and Headless CMS (Butti, n.d.)	26
Figure 6. Examples of structured content (Ottervig, 2023)	29
Figure 7. The evolution in CMS structural, from traditional CMS to Headless CMS and Content Platform (“Headless CMS Explained”, n.d.)	30
Figure 8. The infrastructure of Contentful (“Separate content”, n.d.).....	31
Figure 9. Contentful’s domain model structure (“Domain model”, n.d.).....	32
Figure 10. The traditional model for web applications compared to the AJAX model (Garrett, 2005)	36
Figure 11. MVC architecture (Envall, 2022)	38
Figure 12. MVP architecture (Envall, 2022)	38
Figure 13. MVVM architecture (Envall, 2022)	38
Figure 14. Survey on the most popular frameworks (“2022 Developer Survey”, 2022)	39
Figure 15. Hydration process and why it is overhead (Hevery, 2022b).....	43
Figure 16. Comparing the application initial load of hydration and resumability (Hevery, 2022c)	44

Figure 17. Example of a QRL as an HTML attribute - it points to the location of the JavaScript event handler (Hevery, 2022c)	45
Figure 18. Examples of how Qwik Optimizer works ("Optimizer", n.d.)	46
Figure 19. Google Lighthouse's measurement for Builder.io homepage	47
Figure 20. Google Lighthouse's measurement of performance metrics	48
Figure 21. User flow for ranking films in a gallery feature	55
Figure 22. User flow for creating new gallery	56
Figure 23. Creating a new content type on Contentful	57
Figure 24. The two content types required for a gallery's data retrieving in Filmmash	58
Figure 25. The sample content type fields for gallery's information	58
Figure 26. The sample content type fields for a gallery's film entry	59
Figure 27. Retrieving the necessary API keys for connecting Contentful and Filmmash	60
Figure 28. Filmmash's system infrastructure	61
Figure 29. Filmmash entity relationship diagram	66
Figure 30. Filmmash backend code structure	67
Figure 31. Sequence diagram for creating new gallery	73
Figure 32. Sequence diagram for getting multiple galleries	75
Figure 33. Sequence diagram for getting one gallery	77
Figure 34. Sequence diagram for getting all films in a gallery	78
Figure 35. Sequence diagram for getting one specific film	80
Figure 36. Sequence diagram for updating a film's points	82
Figure 37. Sequence diagram for getting a user	83
Figure 38. Sequence diagram for logging in and signing up	84

Figure 39. Frontend code structure	91
Figure 40. Homepage and Header components	95
Figure 41. Login, Sign up and Create gallery form	99
Figure 42. Overview component.....	101
Figure 43. Mashing functionality	103
Figure 44. Ranking component	104
Figure 45. Filmmash's overview section, viewed on laptop screen (1440px in width).....	106
Figure 46. Filmmash's overview section, viewed on laptop screen (425px in width).....	106
Figure 47. Google Lighthouse's measurement of Filmmash	108
Figure 48. Filmmash's performance metric scores.....	108
Figure 49. Filmmash's passed audits according to Google Lighthouse	109
Figure 50. Google Lighthouse's measurement of Next.js documentation	110
Figure 51. Google Lighthouse's measurement of Nuxt documentation	110

LIST OF TABLES

Table 1. The comparison between Monolithic CMS and Headless CMS (“Headless CMS Explained”, n.d.).....	28
Table 2. Requirement analysis of Filmmash	52

LIST OF CODE SNIPPETS

Code Snippet 1. Backend's package.json file	68
Code Snippet 2. Backend's tsconfig.json file	69
Code Snippet 3. Backend's example of an .env file	69
Code Snippet 4. Gallery data model	71
Code Snippet 5. Film data model	71
Code Snippet 6. User data model	72
Code Snippet 7. Backend code for creating new gallery	74
Code Snippet 8. Backend code for getting multiple galleries.....	76
Code Snippet 9. Backend code for getting one gallery.....	77
Code Snippet 10. Backend code for getting all films in one gallery	79
Code Snippet 11. Backend code for getting a specific film.....	81
Code Snippet 12. Backend code for updating a film's points	82
Code Snippet 13. Backend code for getting a user.....	83
Code Snippet 14. Backend code for sign up operation.....	85
Code Snippet 15. Backend code for login operation	86
Code Snippet 16. Backend code for getting authorized user	87
Code Snippet 17. Backend code for retrieving data from Contentful	88
Code Snippet 18. Backend Express application initialized.....	89
Code Snippet 19. Backend `index.ts` file	90
Code Snippet 20. Frontend `root.tsx` file	92
Code Snippet 21. Frontend router's root component.....	93
Code Snippet 22. Frontend's routing layout.....	94
Code Snippet 23. Homepage component code	96

Code Snippet 24. HomeGalleries component code.....	96
Code Snippet 25. Header component code.....	97
Code Snippet 26. Login component code	98
Code Snippet 27. Gallery component's root file.....	100
Code Snippet 28. Gallery's overview code	101
Code Snippet 29. Gallery's mashing state management.....	102
Code Snippet 30. Gallery's mashing `useResource` hooks	102
Code Snippet 31. Elo's algorithm in code	103

LIST OF ABBREVIATIONS

Abbreviation	Definition
CMS	Content Management System
HTML	HyperText Markup Language
SPA	Single Page Application
FTP	File Transfer Protocol
CSS	Cascading Style Sheets
SSI	Server Side Includes
DOM	Document Object Model
XML	Extensible Markup Language
AJAX	Asynchronous JavaScript and XML
CMA	Content Management Application
CDA	Content Delivery Application
API	Application Programming Interface
REST	Representational State Transfer
WYSIWYG	What You See is What You Get
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
HTTP	Hypertext Transfer Protocol
XSLT	Extensible Stylesheet Language Transformation
MV*	Model – View – Whatever
MVC	Model – View – Controller
MVP	Model – View – Presenter

MVVM	Model – View – ViewModel
UI/UX	User Interface/User Experience
JSX	JavaScript Syntax Extension
CSR	Client-side Rendering
SSR	Server-side Rendering
SSG	Static Site Generation
TTI	Time-to-interactive
QRL	Qwik URL
URL	Uniform Resource Locator
MEN	MongoDB – Express – Node.js
MERN	MongoDB – Express – React – Node.js
MEAN	MongoDB – Express – Angular – Node.js
CRUD	Create – Read – Update – Delete
JWT	JSON Web Token
CLI	Command Line Interface

1. INTRODUCTION

1.1. Background and Context

Ever since Tim Berners Lee invented the World Wide Web in the early 1990s, the field of web development has seen radical changes throughout the years to match with the increasing needs for more versatile and adaptable websites and applications. From the first static HTML websites that were only for displaying hand-coded contents, to the emergence of scripting languages, libraries and frameworks and the introduction of SPA concepts, web application has always been evolving with a view to provide the utmost interactive and responsive users' experience.

However, the status of web development is currently reaching a deadlock. A more interactive and responsive website requires a much higher amount of JavaScript code to be downloaded and executed, which inadvertently slows down the web bootup process, resulting in negative users' experience (Hevery, 2022a). The rise of multi-device and Internet of Things as well as the mobile-first approach also poses threats to the traditional content management approach since it is a monolithic system and is not capable to maintain different platforms. (Holmes, 2022).

Hence, the necessity for a drastic reformation in web application development and content management systems is increasing rapidly amidst the advancement of technologies and the rise of users' expectations.

1.2. Research Questions and Objectives

To address the rising needs for an overhaul of web development, this thesis examines the two modern solutions: Headless CMS and Qwik framework.

Headless CMS is a relatively new approach in backend content management, created to adapt with the invention of smartphones in the late 2000s and the acceleration of technological innovations. It separates where the content and the presentation layer are stored, making it possible for a set of content to be displayed on multiple screen devices, such as mobile phone, smartwatches, or car infotainment system (Maelver, 2023).

Qwik is a recently introduced frontend framework by developers at Builder.io, deemed to be the fastest loading framework with $O(1)$ loading time (Fu, 2022). Although having only reached the beta version in September 2022, Qwik has gained an increasing interest in the developers' community, with a weekly 10,000 installations, according to 2023 statistics by Node Package Manager. Qwik focuses on reducing the loading time of the initialization process without affecting the interactivity or responsiveness of the web application.

The thesis therefore centers around these two topics to answer the following questions:

- What are the problems with current web frameworks and content management systems and how do Headless CMS and Qwik attempt to address these issues?
- What are the real-world study cases of Headless CMS and Qwik
- What are the possible limitations of Headless CMS and Qwik?
- How to apply Headless CMS and Qwik into a full-stack application?

The answers to these questions will provide valuable insights whether Headless CMS and Qwik framework are possible approaches towards a revolution of web application and the future of web development.

1.3. Methodologies

This thesis is a research and development study on Headless CMS and Qwik framework.

The first part of the thesis studies the released materials on the research topics, such as previous publications, official documentations, developers' insights and interviews, to discuss the drawbacks of state-of-the-art technologies and the advantages of the use of Headless and Qwik. The part is also accompanied with the real-world study cases of Contentful, one of the first and leading Headless CMS platforms, and Builder.io, the landing page builder product where Qwik is first invented and put into production.

The second part focuses on the implementation of Filmmash, a social media web application based on the idea of the infamous Facemash by Mark Zuckerberg in 2004. This part presses on all the vital steps of building an application, from base architecture to deployment and performance testing, especially on the integration of Headless CMS and Qwik, to determine their effectiveness and practicality.

1.4. Significance and Contribution

The significance and contribution of this thesis are noteworthy in several ways. The study sheds light on the importance and benefits of Headless CMS and Qwik, the technologies which have the potential to be the future of web development. It provides in-depth analysis on how they can enhance the current state of web applications, which is crucial for businesses to stay competitive in today's digital landscape.

The thesis also demonstrates a practical implementation of Headless CMS and Qwik to showcase their utilities in real-world scenarios. The working prototype proves the feasibility and effectiveness of using the technologies in web development.

The thesis contributes to an existing body of works dedicated to this topic. However, due to the novelty of the technologies, there are only a limited number of studies that have been published. This study not only delves into both

matters with detailed research, but also provides the connection between the two approaches and real-world study cases.

Overall, the findings of this study can help web developers and businesses make informed decisions on the adoption of Headless CMS and Qwik frameworks in their projects and grasp the next wave of web application revolution.

2. THEORETICAL BACKGROUND

This section presents a synthesis and analysis of the existing literature on Headless CMS and Qwik framework. It aims to extensively examine the evolution and historical landscape of web content management and framework development and determine a turning point where old approaches became legacy amidst the rising market demands, hence proving a practical need for modern technologies. Next, the chapter explores the underlying technologies of Headless and Qwik, conducting a comparative analysis with traditional systems to evaluate their benefits and advancement in performance, along with potential drawbacks and challenges. Overall, the section provides a comprehensive overview on the research topics and lays the foundations for subsequent chapters in the thesis.

2.1. Headless CMS

2.1.1. The Early Era of World Wide Web

Berckmans (2022) proposes the definition of content management system as a platform which “provides the framework that content publishers will then only have to dress up with the help of texts, images, videos, contents”. The statement aligns with the first ideation of the World Wide Web, as in 1989, computer scientist Tim Berners-Lee submitted a memorandum regarding information management to derive a solution for the difficulties in finding data stored on different computers (May, 2019). The proposal discusses the idea of an information system for better content management in a large organisation and used the terms “web” and “hypertext” for the first time (Berners-Lee, 1990). Hence, it can be inferred that the history of content management systems started with the invention of the World Wide Web and the creation of the very first websites.

In the early days of the web, websites were built using static HTML files, with all content being manually coded into. To make the files available on the web, they

had to be uploaded via an FTP program to a directory on a running web server. The advancement of technologies in the following years resulted in websites becoming more visually appealing and multifunctional: the supporting of images in a Mosaic browser, the creation of CSS for styling websites and the introduction of SSI for enhancing content management of a websites by dividing sections into smaller manageable portions (Ottervig, 2022). At that stage content management was limited to the building and maintenance of static information only; however, as the Internet expanded, there was an increasing demand for content being dynamically updated and regulated. The inception of dynamic content can be traced back to 1997 when the Document Object Model (DOM) was first introduced. It was a revolutionary system enabling website creators to manage parts of a document as well as to obtain full control over HTML elements, both in terms of style and content. Coupled with the invention of Asynchronous JavaScript and XML (AJAX) two years later, which allows developers to send and receive updated data without reloading the page, it laid the technical foundation for content management systems.

This era also witnessed a rapid upsurge in the number of computer usages, as Roser (2018) pointed out that by the year of 2000 “almost almost half of the population in the US was accessing information through the internet”. The United States Department of Labour (1999) also measured a sharp increase in households owning computers by different levels of education from 1990 to 1997, with “the amount spent by the average household on computers and associated hardware more than tripled”. Heslop (2022) argues that the rise started what is now known as Web 2.0, where websites were becoming more sociable and dynamic, and the need for fresh, relevant, and daily-updated content as a collaborative effort grew significantly. With the rapid technological advancement and the widespread accessibility to computers and the Internet, it is reasonable to presume that the need for an information management system that allows individuals to control and publish their data was on the rise, hence, the acceleration of content management systems.

publish content to the Internet. Its monolithic architecture is composed of three main components:

- A database that stores the content tree, both programmatically and stylistically
- A content management application (CMA) that allows users to create, manage, and edit content in a CMS. It typically provides a user-friendly interface that enables users to create and publish content without requiring technical knowledge. A CMA allows content creators to upload and manage digital assets such as images, videos, and documents, and organize them into folders or categories.
- A content delivery application (CDA) is responsible for delivering the content to end-users. It retrieves and delivers the content stored in the database to the appropriate device or platform, using a template or layout to present the content to the end-user, with possible provided features such as caching, authentication, and security.

This first movement of content management systems integrates both the backend administration and the frontend presentation layer into controlling a website, thus creating a coupled and all-in-one solution to content creators. The approach has its apparent advantages, as it brings down the technological barriers as the CMA provides people a non-coding solution for creating, uploading, presenting content. The monolithic system also provides non-developers with the ability to manage content at ease by tightly packing the client-side and server-side in one unit, hence remarkably reducing maintenance and deployment cost. With the then-revolutionary system, the monolithic CMS dominated the Web 2.0 era, with multiple enterprises revolving around content management began to emerge, including FileNet, Vignette, Interwoven or EpiServer (Ottervig, 2022). The early 2000s also witnessed a surge in website building platforms, with WordPress being the most popular, with its open-source

policy to let third-party developers contribute different customizations and extensions (Dineley, 2008).

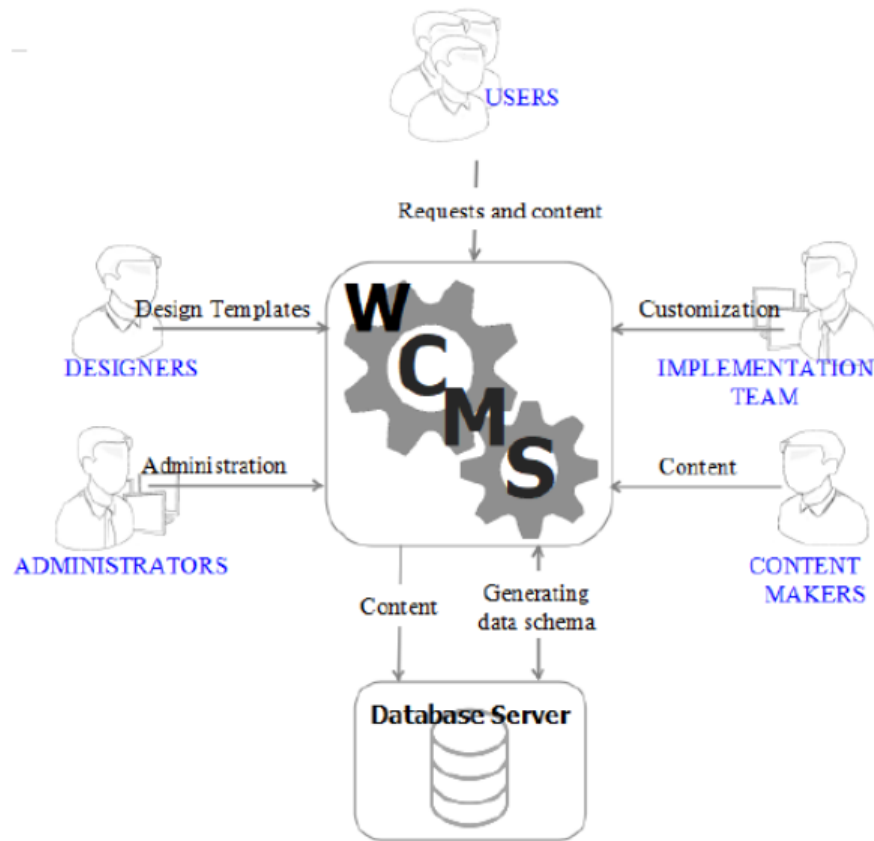


Figure 2. The functioning diagram of monolithic CMS as a coupled system
(Yermolenko & Golchevskiy, 2021)

Alongside with the positive impacts, however, the monolithic approach to content management also contains huge problems in its architecture. The conventional CMS makes it challenging to expand web applications because of its bundled, comprehensive structure, which may cause conflicts between the server and client sides once being scaled up. It also poses a problem for developers who maintain the site as they are constantly presented with new frontend frameworks and tools to apply for the application, however, monolithic CMS with its inflexible integration of templates can hinder developers from adopting a more suitable approach to improve the user experience. And the problems worsened as the world saw another technological revolution: the

invention of smartphones and the rise of omnichannel. The use of smartphones as a channel of content viewing rocketed after the introduction of the first iPhone in 2007 and the first Android smartphone in 2008. StatCounter (2023) estimated that in a span of only seven years from 2009 (the first year that data was collected) to 2016, a sharp rise in the usage of smartphones overtook personal computers as the main source of accessing the web and has continued to rise ever since.

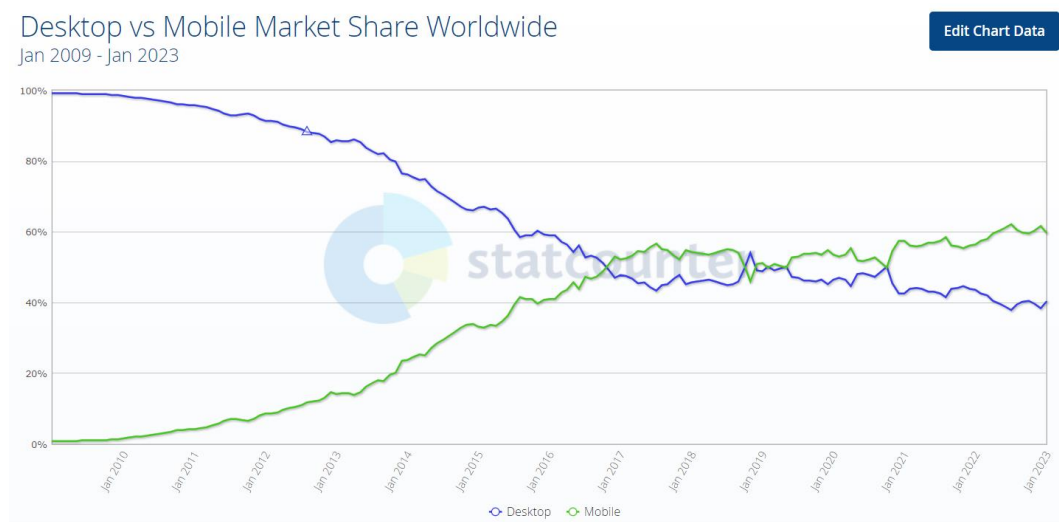


Figure 3. A comparison of desktop and mobile market share worldwide from January 2009 to January 2023 (StatCounter, 2023)

This shift to mobile devices as the main web content provider posed detrimental effects for monolithic CMS, as it was “explicitly created for delivering Web content to desktops and laptops” (Heslop, 2022), therefore it was merely impossible to contribute content to both platform with respect to the style or the structure. Temporary solutions have been drawn to address these problems, the most notable being creating a subdomain with a “m dot” in the beginning to signify whether a website is made for mobile devices. However, it also meant that developers would have to maintain two unrelated sides in terms of address, with the similar content, and managing two sides as well as performing possible analytics measurements would be a burden (“What is a”, 2019).

Furthermore, with the proliferation of innovative technologies of which smartphone was a highlight, businesses faced with opportunities and challenges to adapt to in-store technological solutions across multiple devices, thus creating the term “omnichannel” to provide a smooth customer experience regardless of the channel (Piotrowicz & Cuthbertson, 2014). Omnichannel in its literal terms means “every channel”, and it was the focus of technological innovation in the 2010s, with the introduction of many revolutionary devices, such as tablets, smartwatches, gaming consoles or even voice-activated devices.

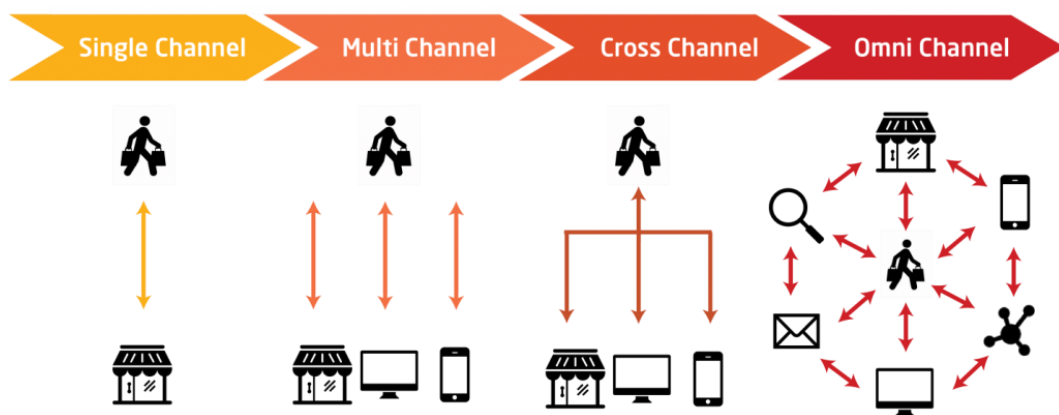


Figure 4. The evolution of omnichannel (Nguyen, 2019)

Hence, the maintenance and management of content for every single existed channel was impossible, and the field demanded a shift in CMS paradigm, where content can be detached from the presentation layer thus being compatible across multiple devices.

2.1.3. The Emergence of Headless CMS

The switch to omnichannel delivered a new solution for content management called Headless CMS. A Headless CMS is a backend content management system where the content repository – the body – is detached from the presentation layer – the head and managed individually, hence the term “head-less”. This separation allows content creators to control content in a single location and distribute it to any desired device. In the age of omnichannel, this approach plays

a crucial role as users are given the capabilities to integrate content into any software, device, or content carrier system and without having to worry about conflicting layers.

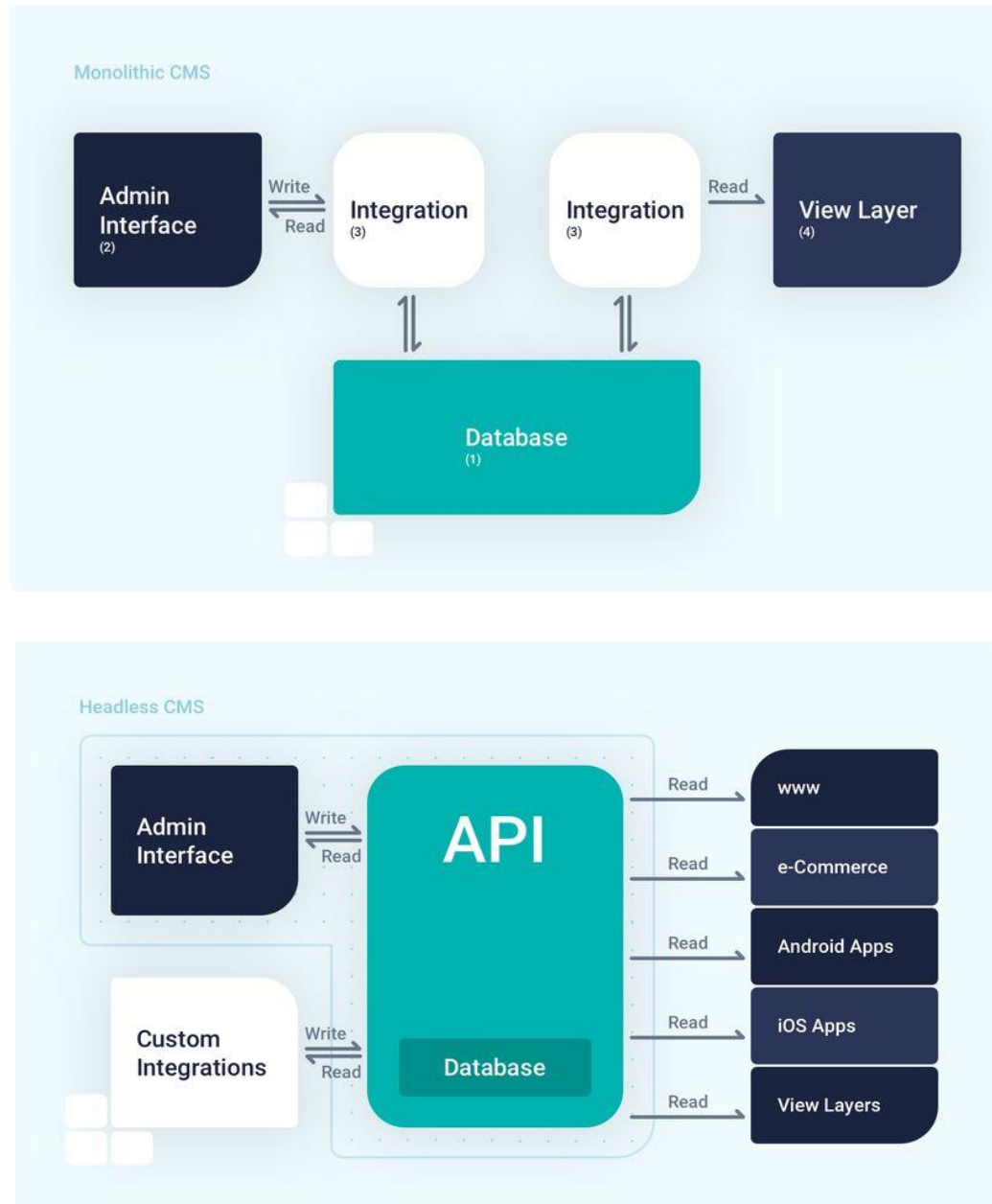


Figure 5. The architectures of Monolithic and Headless CMS (Butti, n.d.)

The center of a headless CMS revolves around the communication of data from the content provider to the multiple device endpoints, through an application programming interface (API). The concept of API was first introduced in the 1960s as a data connector method for different computer systems (Cotton &

Greatorrex, 1968), before being introduced to the field of databases by Date and Codd (1975). With the advance of the Internet in the 1990s and the introduction of the REST architecture (Fielding, 2000), the API has prevailed and has been used by a wider range of developers and businesses. In the omnichannel's era API is an essential component of modern software development and in the discussed case, the Headless CMS, allowing developers to integrate various services and data sources into their applications. The headless system only manages the content input by creators and transmits the pieces of information through API to the platforms that present the content to application users.

By detaching the process and let the individual system manage its own respective task, the headless CMS has shown enormous improvement from the monolithic version and has gradually become the defined content management system in recent years. While traditional CMS relies on a single system to work on all fields, including controlling content in the backend and rendering the view in the frontend, the Headless CMS provides a substantial impact by leaving the rendering for the browser and the client-side used framework. This is beneficial to not only content creators, but also application developers and users, as it enhances the editorial experiences, the flexibility and scalability in the choice of technology stack, and the improved rendering mechanism. Tighter security is also a result of this new approach, as the monolithic system poses a larger vulnerability to attacks for its all-in-one approach.

	Monolithic CMS	Headless CMS
Hosting & delivery	In-house	In the cloud
Development mindset	Project-focused	Product-focused
Content model	Built for a single page	Built for many products
Supported devices	Limited	Limitless
Reach	One-to-one	One-to-many
Workflow	Waterfall	Agile
Updates	Scheduled	Continuous
Backend system	Monolithic, all-in-one	Microservice, best-in-class

Investment	Large up-front cost	Quick proof of concept
------------	---------------------	------------------------

Table 1. The comparison between Monolithic CMS and Headless CMS (“Headless CMS Explained”, n.d.)

However, being a relatively new approach, Headless CMS also contains many issues. Its complex nature demands knowledge in both frontend and backend development, and the steep learning curve to understand thoroughly its workflow may serve as an obstacle for content builders. While the detachment, as previously discussed, would provide freedom for developers in choosing their own technology stacks, the lack of built-in functionalities such as forms, navigations, and search bars that the monolithic approach provides also negatively contribute to the creation of web applications through Headless CMS.

As the Contentful website (n.d.) suggests, the idea of Headless CMS still has not resolved the problems of unstructured content. Unstructured content cannot be broken down to individual component as it mixes the content of an application into a single entity of information, data, and code, thus providing immense difficult to update a small part of content as the whole entity also requires republishing. This is the approach that is mainly used with the WYSIWYG editor, and while it can create a functional website, the content is stuck to the format and “cannot be easily reused across different platforms and channels or

repurposed for new projects” (“Headless CMS Explained”, n.d.).

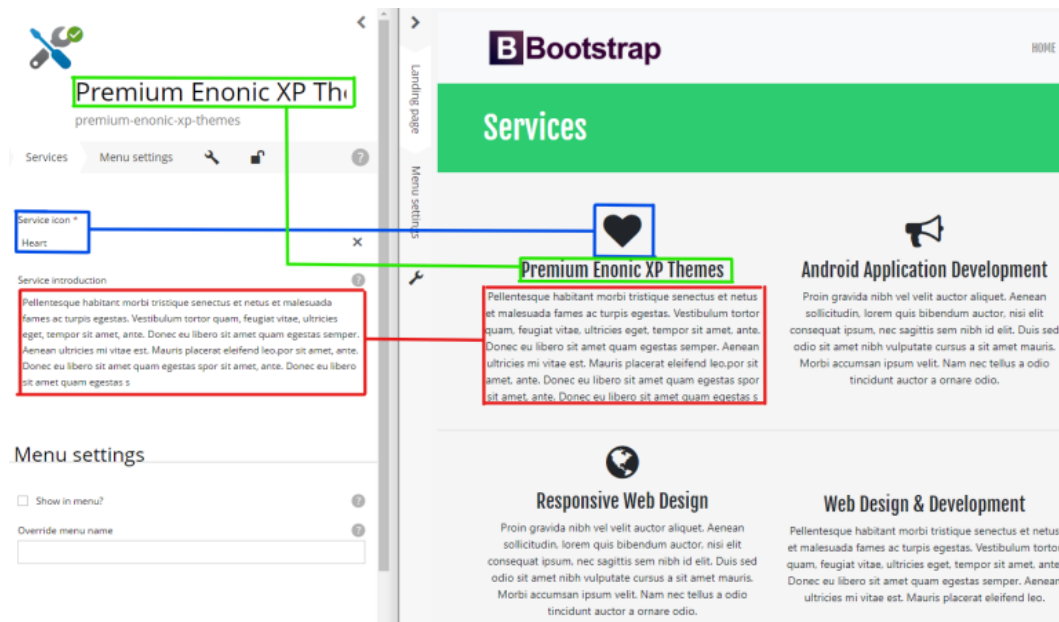


Figure 6. Examples of structured content (Ottervig, 2023)

Structured content, in contrast, is information divided into chunks of meaningful data in an organized and categorized way. These pieces of data are classified into the right structural content type, thus can be managed predictably regardless of representation layers. Hence, the goal of application developers in the modern era is to create a content management system that combines the adaptability of Headless CMS and the organized structured content model, or to define by a term, composable content platform.

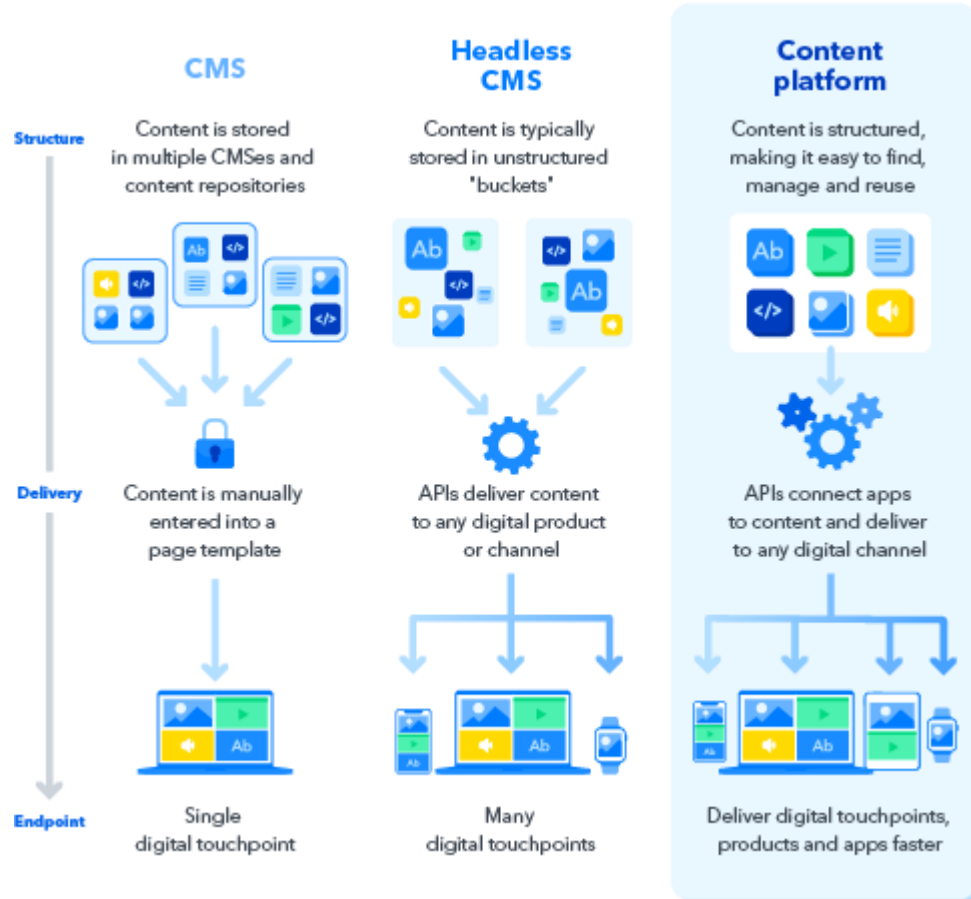


Figure 7. The evolution in CMS structural, from traditional CMS to Headless CMS and Content Platform ("Headless CMS Explained", n.d.)

2.1.4. Case Study: Contentful

Contentful is a composable content platform founded in Berlin in 2013 by Sascha Konietzke and Paolo Negri, with its customers are from a wide range of fields, from Spotify, Nike to Lyft (Lardinois, 2018). Although the company does not publicly advertise their content management system as headless, it should be

regarded as one due to its main mechanism of delivering content through API.

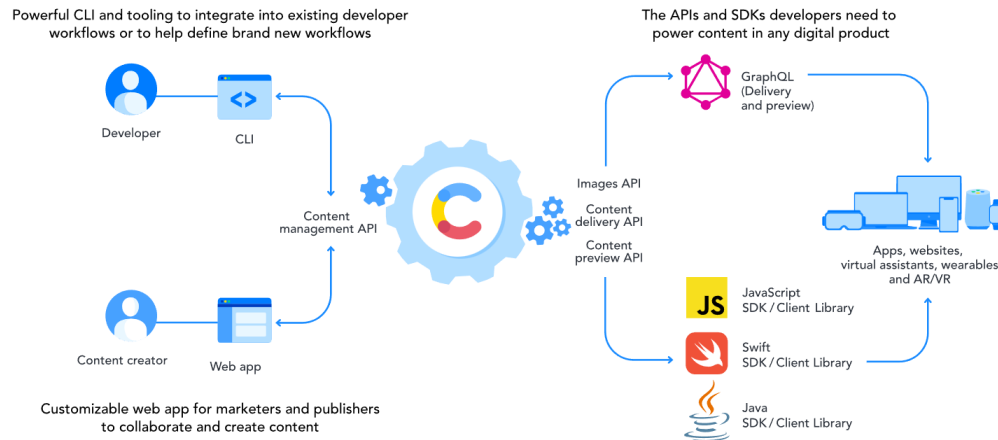


Figure 8. The infrastructure of Contentful (“Separate content”, n.d.)

Lying at the heart of Contentful are its three main concepts: domain model, content model and API. A domain model is a conceptual model where behavior and data are incorporated into objects and entities (Fowler, 2002). According to its documentation, Contentful’s domain model includes four entities:

- User: refers to Contentful’s account holder and is provided with authentication method and a personal access token, which would be necessary to create content on Contentful.
- Organization: is a group of users, taking care of account-related matters such as subscriptions or billings.
- Space: is the container of content that allows separation of data to match with each project’s structure.
- Environment: Environments refer to entities that exist within a particular space and enable the creation and management of contents and setup specific to that space, providing the ability to modify them independently from one another.

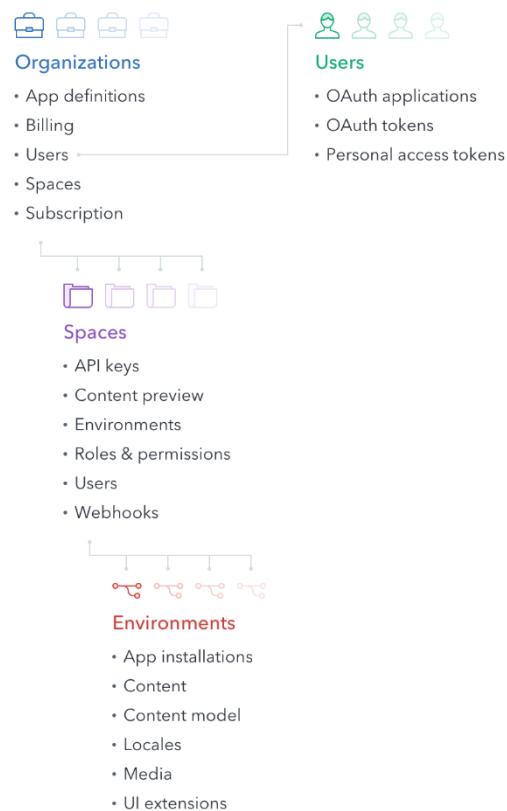


Figure 9. Contentful’s domain model structure (“Domain model”, n.d.)

Content model is the core of a content composable platform such as Contentful. As previously mentioned, a Headless CMS with structured content type is the eventual goal of content management systems, and content model is Contentful’s approach to reach the target. Content model works the same way as entities in a database do, in terms of content structuring, breaking it down into individual elements, providing a comprehensive description of each element, and illustrating their interconnections as well as mapping out their relations (Pope, 2021). The content model comprises of content types, and their attributes, which in Contentful’s case includes fields, entries, references and media assets, and this approach enhances the digital content management experience as it enables user to categorize and classify their contents to distribute on many platforms.

Contentful provides different APIs for the creation, management, and distribution of contents (“Contentful contents API”, n.d.):

- Content Delivery API is the read-only API used for delivering content, in the form JSON data or media and file assets, stored at Contentful to multiple application endpoints. The API can be described as the connector between the backend, which is Contentful, and the frontend of any chosen presentation layers.
- Content Management API provides the ability to contribute content to Contentful as a plugin with other backend systems, and it is highly effective with applications that need custom editing experiences.
- Content Preview API is relatively the same as Content Delivery API, however, it is to preview the content retrieved from Contentful before publishing to the desired application for public consumption.
- Images API provides the ability for images editing without the loss of CDN caching.
- GraphQL Content API provides an alternative method of retrieving data other than the REST service.

With the wide range of functionalities provided, Contentful serves as a modern solution in creating, managing, and distributing content in respect to the composable content platform. The application built for this thesis uses Contentful for backend management as a practical demonstration.

2.2. Qwik Framework

Since its inception in the mid-1990s, JavaScript has been the driving force behind the development of web applications, with the developer community continuously pushing the boundaries of the language, introducing new concepts and frameworks for greater efficiency and versatility, thus contributing to its ever-changing ecosystem. However, as the solution of a problem may well be the creation of another, the growth of JavaScript also results in unintended

consequences. This part of the thesis analyzes the evolution of JavaScript and its frameworks, acknowledging the innovations as well as the problems of current web technologies and the attempts to solve them, and explores how the newly introduced Qwik framework is leading the next generation of Javascript-based applications.

2.2.1. The Early History of JavaScript

As mentioned in the previous section, in the early days of web development, popular web browsers like Mosaic and Netscape Navigator were static-only, and the desire for a dynamic web page where users can interact with, or input data into was higher than ever. In 1995, tasked with the dynamization of browsers, Netscape collaborated with Sun Microsystem to embed its Java language while also implementing Scheme language into its flagship browser, the Navigator, which raised questions and debates on whether it was necessary for web browsers to be implemented by two different languages. Netscape management eventually decided on inventing a new scripting language with syntax resembling Java's, and the first prototype, called Mocha and then LiveScript, was created by software developer Brendan Eich in just ten days. By the end of the year, with the success proven by its beta release and an attempt to capitalize on the popularity of the Java language, its name was then changed to the famous JavaScript developers have come to know since (Rauschmayer, 2014). It should also be noted that despite having similar names, Java and JavaScript are fundamentally different programming languages. Java is an object-oriented language that is compiled to bytecode, which can run on any platform that has a Java Virtual Machine (JVM) installed, while though JavaScript shares the similar syntax and programming concepts, it is an interpreted language that runs on a browser's JavaScript engine and is used to create dynamic and interactive web pages, hence these similarities are superficial and do not imply that the two languages can be used interchangeably.

The invention of JavaScript was effectively the beginning of the dynamic web pages era, where web pages were equipped with the client-side scripting methods that could respond to users' actions, interact with information and data input, and perform customizable events. However, the browser war between Netscape and Microsoft, who debuted the Internet Explorer browser in 1995 by reverse-engineering JavaScript into their own language called JScript, halted the evolution of the language as well as dynamic web applications (Rauschmayer, 2014). While JScript was greatly influenced by JavaScript, its approach was noticeably different from that of the predecessor, consequently leading to the problem of incompatible platforms. Developers then faced a conundrum of polishing their code to ensure that they worked for both browsers (Champeon, 2001). Only until the standardization of JavaScript by the ECMA International in 1997 with the introduction of the term "ECMAScript" as a compromise between involved parties, and the domination of Microsoft's Internet Explorer browser did the innovation of JavaScript begin (Envall, 2022).

The turning point of JavaScript history was when developers discovered its utilities in handling the web pages reloading issue. As dynamic websites replaced the static ones, it was noticeable that each time users interacted with the web pages, such as searching for an address or submitting a form, the websites would need a full page reload, thus developers were often required to embed a waiting page for even the users' simplest requests. This annoying characteristic of web browsers made websites inferior to the smoothly executed desktop applications. The task was to find a way to send HTTP requests with client-side scripts in the background, so that the update of data would not interfere with users' browsing experience. In 2005, Jesse James Garrett introduced AJAX (Asynchronous JavaScript and XML) to deal with this problem. His attempt was influenced by non-reloading websites such as Google Maps and Google Suggest, utilizing "data interchange and manipulation using XML and XSLT" as well as "asynchronous data retrieval using XMLHttpRequest", with JavaScript being the binding engine

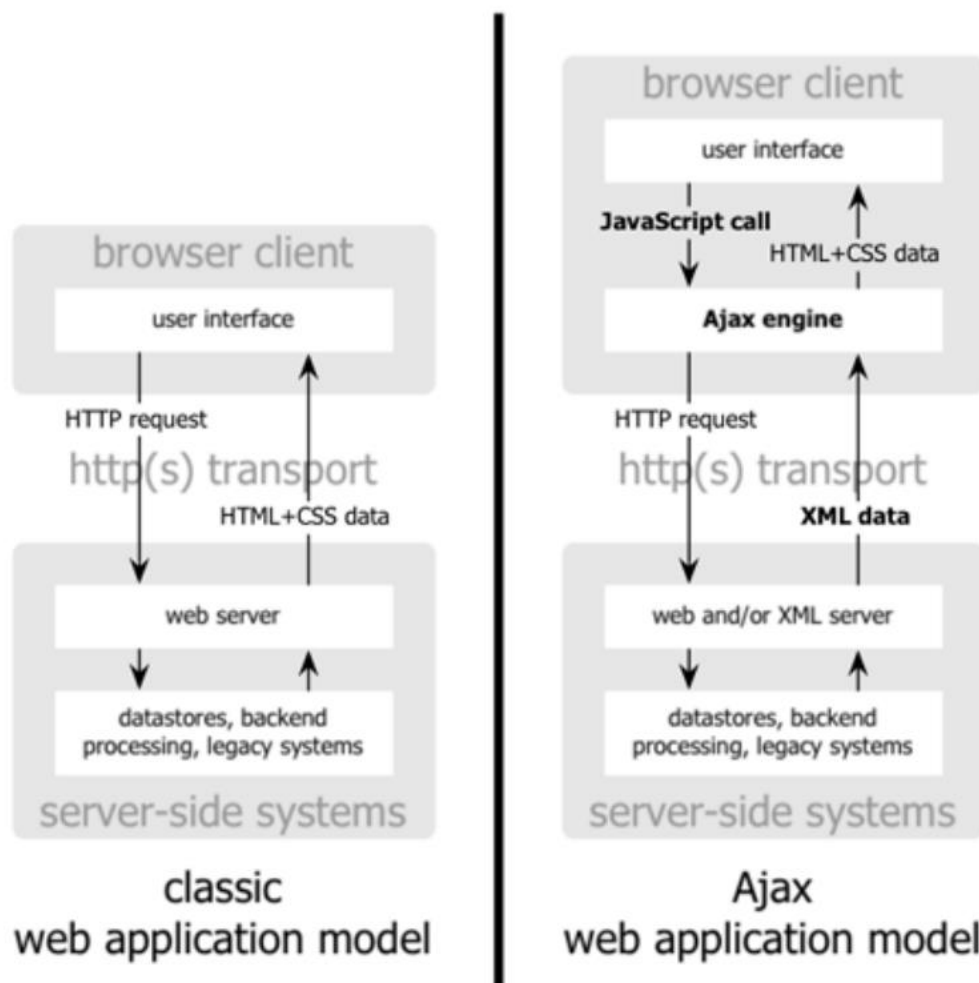


Figure 10. The traditional model for web applications compared to the AJAX model (Garrett, 2005)

The introduction of AJAX was the backbone of dynamic web applications nowadays and it strengthened JavaScript's importance on pushing the limitations of web development, as in a short span of time, frameworks based on the AJAX model were released daily (Mahemoff, 2005). However, it could be argued that the "AJAX-framework boom" also prompted programmers into developing more complex features without a standard structure, hence negatively affecting codes' efficiency and maintainability. As JavaScript code's size had been increasing rapidly and becoming an indispensable part of the browser, it can be concluded

that the new era of web development demanded a standardized and scalable set of JavaScript frameworks to regulate, maintain and improve web applications.

2.2.2. MV* Architecture and SPA Frameworks

The acronym MV*, or MVW, represents “Model – View – Whatever”, which is a set of architectural patterns designed as a standard approach for JavaScript frameworks. The concept incorporates the Separation of Concerns (SoC) principle and attempts to provide modularity and structure for projects by separating blocks of code by their specific purposes (Dewhirst, 2022). SoC was a tremendous improvement towards web development, as categorizing code in an organizing way provided developers the ability to modify parts of the application without altering unrelated sections. This made changes to the code faster and easier, reduced bugs while also made unit testing much more feasible.

The three main design patterns of MV* concepts are Model-View-Controller (MVC), Model-View-Presenter (MVP) and Model-View-ViewModel (MVVM). Laying at the core of these patterns are Model and View:

- Model is the data storage layer of MV*. It interacts with databases and networks while representing the application logic of how data should be managed and processed.
- View is the user interface (UI) layer that visually presents the rendered Model's data and provides interactions.

The roles of Model and View are particularly similar in these patterns; however, their mechanisms differ on how to connect and transfer data from Model to View and vice versa.

- The Controller in an MVC pattern functions as a connector between the Model and View, it handles user action from the view and updates the model accordingly. Designed in the 1970s, the MVC pattern is one of the oldest software architectures, and it sets the basis for future patterns.

However, the circular relationship between layers in MVC makes them tightly coupled to each other, and the view layer has no information about the Controller, thus making it harder to scale up the applications.

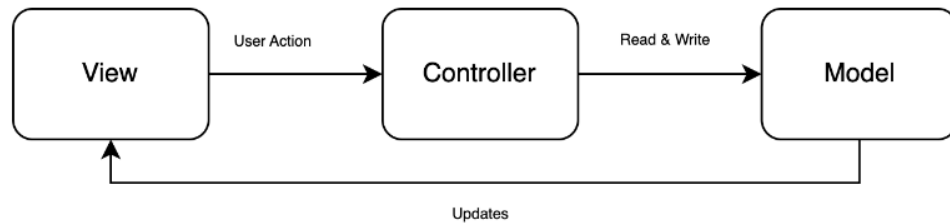


Figure 11. MVC architecture (Envall, 2022)

- The Presenter in an MVP pattern resolves the problems of MVC pattern, as not only does it update the model based on user action from the view but also acknowledges the model's changes and updates the UI.

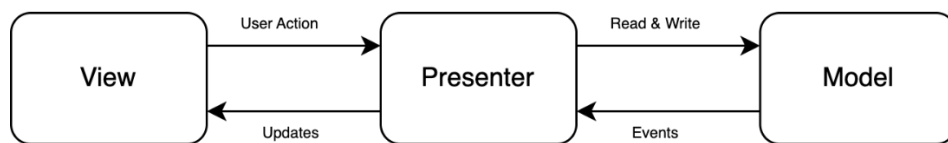


Figure 12. MVP architecture (Envall, 2022)

- The ViewModel in an MVVM pattern is the modern approach of MVP, as it uses a technique called data binding to connect the Model and the ViewModel, which reduces the data flow and enhances flexibility and testability.

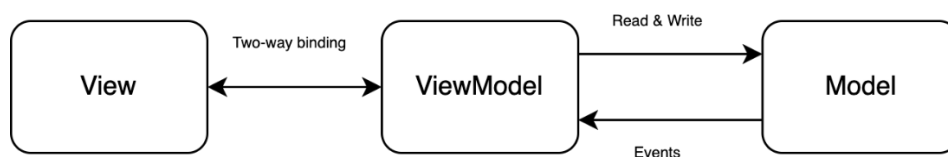


Figure 13. MVVM architecture (Envall, 2022)

These MV* design patterns created a structural approach for software developers to build JavaScript frameworks and libraries based on, and they

fueled the desire of creating more interactive and native web applications, thus being the backbone of the introduction of single page applications (SPA). SPA can be understood as a web application that loads an initial HTML document and dynamically updates its contents based on users' interaction. Its highlighted innovation is to ignore the default method of a full page reload whenever users interact with the application, but instead communicating with the server using APIs and AJAX to fetch data when needed. This approach means applications utilizing SPA greatly enhance user experience and result in a faster, more responsive, and more engaging interactivity, especially for the heavily interactable web application. Maheshwari (2021) pointed out that many large technology corporations have been utilizing SPA in parts of their products, including Facebook (at the time this thesis written whose name has been changed to Meta), Google, LinkedIn, Twitter or Netflix, and he stated that "revolutionized the basic thinking in the industry to design modern web applications resulting in the invention of modern JavaScript-based frameworks with wide-open support from the community." This thesis discusses the most widely used, React, while briefly introduces Vue and Angular which are also popular among developers, and all based on the MV* patterns.



Figure 14. Survey on the most popular frameworks (FullStackOverflow, 2022)

React was created by Facebook (now Meta) in 2013 by Jordan Walke to counter their rapid expansion of user base and create a more responsive and interactive UI/UX. Because of this, React is often regarded as the V in a MVC pattern, as it only improves the view mechanism and leaves the rest for the developers' desired backend implementation. It is also worth noting that React is not a framework but a library. While the framework lays out the structure and foundation that developers write their codes on, libraries provide programmers with predefined sets of functions and methods to integrate into their application at their own ease. React breaks down parts of the view into multiple components holding their own states and props, and they are written based on an extension of JavaScript called JSX, to embed JavaScript code inside HTML, providing great efficiency for component rendering. Whenever there is an event incoming from a user, the view signals to the component's state with an action for an update, which then triggers a re-render of the view to reflect the state change. While this unidirectional data flow proves a significant improvement since it handles events individually, it can also be expensive due to its rendering of any small actions input by users. Therefore, to tackle this problem, React also introduced the concept of virtual DOM. The virtual DOM is a cached structure re-created whenever an event is fired, computes the differences with its old snapshot, and determines to render only not the whole page but only the altered components in the real DOM. This algorithm is known as reconciliation, and it boosts React performance since no extra cost is spent for the non-altered components.

Another SPA framework gaining popularity in recent years is Vue. Developed by Evan You in 2014, Vue is a MVVM frontend framework, and its core foundation lay on the ViewModel and the two-way binding mechanism. Vue components, like React, also use virtual DOM and a computed watcher (the alternative in React is the `useEffect` hook) to execute re-rendering based on user activities. Vue is largely influenced by the AngularJS framework that You worked on in his time as a Google software engineer, however, he felt that it was too heavy for its use

case and set out to build a lightweight framework that adapted all the strengths of AngularJS while introducing the modern concept learnt from frameworks like React (Cromwell, 2017). The result is an open-source framework with an easy learning curve and provides high efficiency.

In 2016, Google also released Angular. It was the successor of the AngularJS framework initially released by Misko Hevery in 2010, first introducing the groundbreaking concepts of directives and two-way binding. Angular revolutionized its predecessor by moving to ES6, incorporating TypeScript as default, and utilizing components instead of scope and controllers as a method to stay relevant as React and Vue's stance was increasing rapidly. However, to deal with component updates, Angular used a concept called incremental DOM), which is a series of instructions that creates DOM trees and directly mutates them whenever changes happen.

2.2.3. Metaframeworks and the Problem of Hydrations

SPA frameworks, however, have their downsides. To match with its desire of fetching, loading, and rendering components natively, most single page applications utilize Client-side Rendering (CSR) as the default render system, which means that pages and their components are directly rendered to the view by JavaScript. One detrimental effect of CSR is for SEO, which is a critical factor for businesses to thrive in a digitalization era. SEO uses web crawlers which consume HTML and tends to ignore JavaScript, which is the engine of CSR. Because of this, frameworks which allow the use of SPA but implores techniques that are more SEO-friendly began to appear in the second half of 2010s. These are called metaframeworks as they were built on top of the open-source SPA frameworks, such as Next.js for React or Nuxt.js for Vue, and supports the use of Server-side Rendering (SSR) and Static-site Generation (SSG), which generates HTML on the server and only send it to the browser on request.

However, this shift in the rendering system only solves half of the big picture. As applications grow significantly in complexity, the codebase of JavaScript is also enlarged proportionally, hence, there must be a solution to connect the JavaScript with the HTML to send as a pack within SSR. Hydration was the initial solution for this problem. Hydration is a technique that aims to attach event listeners to the rendered HTML so that a web application receives full interactivity. To achieve this goal, the hydration process needs to go through the following steps:

- Downloading every JavaScript file from the server.
- Parsing and executing JavaScript files to determine the specific task of each event handler.
- Looping through all the DOM nodes to find which exact DOM node that an event handler belongs to.

Hydration does solve the connection problem between JavaScript and the DOM nodes as it neatly binds the script to its desired node. However, developers now have been raising an argument that hydration is expensive regarding executing time, and as Carniato (2022) claimed that it “increases JavaScript payload and may have even longer times until the application is interactive than client-rendering only”. Hevery (2022b) broke down the pieces to further prove the statement, as he argued that for a large application, the technique of downloading and parsing JavaScript is very slow, and the fact that hydration needs to traverse through all the DOM nodes to attach those parsed event listeners is a re-execution of components that the SSR/SSG process has already finished. This duplication in the rendering system is the solid proof for the flaw in hydration’s mechanism. This eventually greatly affects the application’s time-to-interactive, as users would need to wait for all the steps to be finished, including downloading, parsing, and binding JavaScript event listeners into every DOM node.

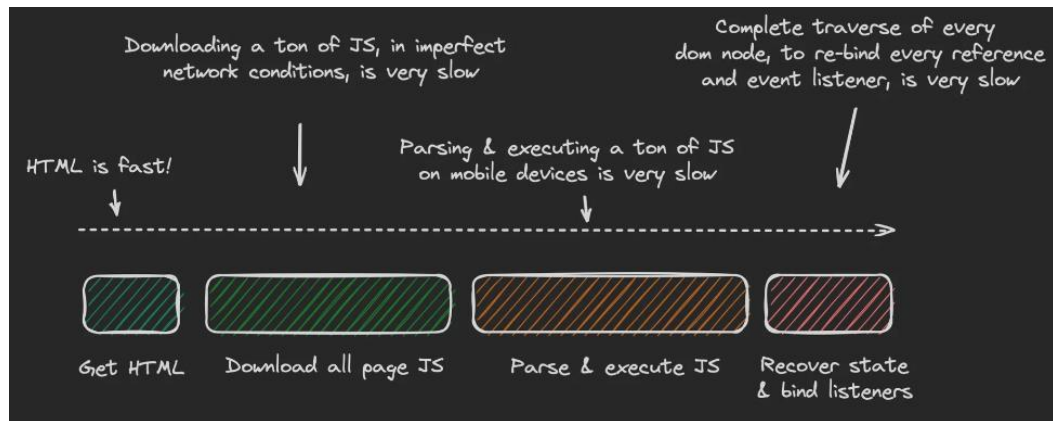


Figure 15. Hydration process and why it is overhead (Hevery, 2022b)

Astro's islands architecture is a possible workaround for hydration, as instead of starting the full hydration process at the initial page load, it only hydrates when users interact with a component, and only the interacted components are hydrated. This approach proves to be useful since frameworks nowadays tend to divide the applications into blocks of components, or "islands", and the cost for the hydration process is greatly reduced for its distribution to components if and only when there is interaction. However, this method is only considered as a workaround, not a replacement of hydration, as the bigger the application gets, the larger the islands are, and there occurs the problem of inter-islands communication. For example, if there is a component that links to many components in the application, the hydrating that component triggers all the hydrations for the attached components, which reverts to the original problems of hydration.

Hence, it is feasible to say that hydration is not a viable solution, and the next generation of framework needs to explore a set of new techniques to resolve these issues of current state-of-the-art.

2.2.4. The Qwik Framework

As the previous section concludes, the new generation of JavaScript frameworks needs to resolve the problems of slow time-to-interactive (TTI). And in 2021, the

Builder.io team, led by CTO Misko Hevery, attempted to solve this issue by releasing the Qwik framework. It is worth to noted that Misko Hevery was the person in charge of the original AngularJS project that helped shape the future of JavaScript MV* frameworks in the early 2010s, and therefore he could be regarded as one of the forefront revolutionists of modern web development. His latest introduction, Qwik, differs from the existing frameworks by introducing the concepts of resumability and progression.

Resumability is a no-hydration approach, meaning that instead of eagerly downloading, executing, and attaching JavaScript event listeners to DOM nodes, it pauses the server execution and resumes it on the client side. By this method, Qwik web applications are loaded almost instantly as it only retrieves the HTML in its initial load, and since pure-HTML is fast, it greatly reduces the TTI.

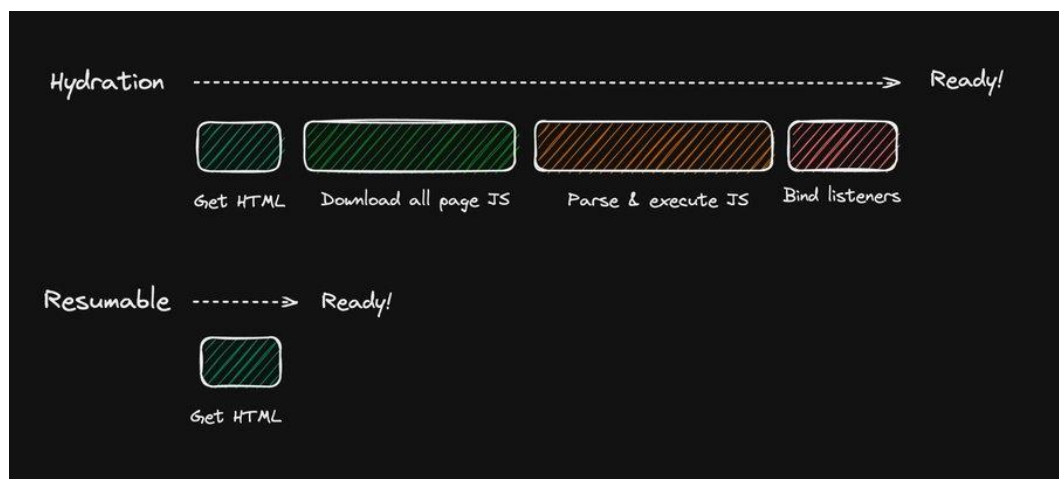


Figure 16. Comparing the application initial load of hydration and resumability (Hevery, 2022c)

Hevery (2022c) argues that “a page may easily have hundreds of event listeners, but the vast majority of them will never execute”, thus concluding that the hydration process that binds listeners to DOM is a waste of resources. Hence, for the client-side interactivity, Qwik’s resumability concept delays the JavaScript load by utilizing a method called serialization. In the initial rendering, the SSR acknowledges the event listeners, however it neither binds nor ignores these

JavaScript codes, instead, it serializes them as HTML attributes, known as QRL. QRL can be viewed as an URL pointing to the specific JavaScript chunk location to load whenever users execute an event listener. All these QRLs

```
<button on:click="MyComponent_click">Click me!</button>
```

Figure 17. Example of a QRL as an HTML attribute - it points to the location of the JavaScript event handler (Hevery, 2022c)

are stored in a tiny 1KB JavaScript file called the Qwikloader that is loaded in the initialization, and compiles the event listeners into one global handler, thus, whenever users interact with a listener, the bubbling process distributes it until the global handler receives and executes it. This approach is revolutionary, as it solves the core problem of hydration, since there are no heavy codes, states, or templates to be downloaded prematurely, and it is not necessary to bind every event handler with its node before users interact with it (Hevery, 2022c). Moreover, the Qwikloader, which is the only JavaScript bundle downloaded from the server in the bootstrap process, is only 1KB in size, and it is independent from the application complexity, which strengthen Fu's (2022) and Hevery's (2022d) statements that Qwik is an $O(1)$ – a linear, scalable, and high-efficient framework.

If the resumability concept is powerful for reducing initial cost for event handlers, Qwik's progression concept achieves similar efficiency for components. As hydration proves to be impractical for components, Qwik attempts to solve this issue with an optimizer that splits the components into chunks with indicators to be lazy loaded when necessary. It is argued that fine-grain lazy loading is the goal of next-generation frameworks (Hevery, 2022e), and Qwik's optimizer attempts to fulfill this. It searches the code for an indicator, the \$ symbol, to understand at which point it should break the components into smaller files. The documentation provides the following example.

```
export const Counter = component$(() => {
  const count = useSignal(0);

  return <button onClick$={() => count.value++}>{count.value}</button>;
});
```

```
const Counter = component(qrl('./chunk-a.js', 'Counter_onMount'));
```

chunk-a.js:

```
export const Counter_onMount = () => {
  const count = useSignal(0);
  return <button onClick$={qrl('./chunk-b.js', 'Counter_onClick', [count])}>
};
```

chunk-b.js:

```
const Counter_onClick = () => {
  const [store] = useLexicalScope();
  return store.count++;
};
```

Figure 18. Examples of how Qwik Optimizer works (“Optimizer”, n.d.)

In this example, it can be understood that the optimizer locates the \$ symbol on the components and the button’s click event and breaks them down to smaller pieces of code that can be lazy loaded whenever users request them. This approach builds on the technology of island architecture, however, coupling with Qwik’s own serialization discussed, it makes the code retrieving process much faster and efficient.

While Qwik is a promising and innovative framework, it is still considered in beta mode as of March 15, 2023. Hence, there are still some technical limitations and challenges associated with its use. Firstly, since it is a new framework, there may be a lack of support and documentation compared to more established

frameworks. Additionally, its focus on speed and simplicity may mean that it is not as feature-rich or customizable as other frameworks. However, as Qwik continues to evolve and improve, these technical limitations may be addressed over time. It is also worth mentioning that the development of Qwik is still ongoing by the time this thesis is written, so changes may occur to the information stated in this thesis.

2.2.5. Case Study: Builder.io.

Builder.io, founded by Brent Locks and Steve Sidwell in 2019, is one of the leading businesses in the e-commerce industry, with its recent funding of \$14 million from Greylock and other investors (Hall, 2021). It develops a page builder application that utilizes multiple features, including drag-and-drop, integration with third-party applications like Shopify Storefront and especially Headless CMS. To enhance user experience and developers' expectation, in 2021, Builder.io developed Qwik to become the application's main framework, and has been using it for the homepage since 2022.

This section analyzes the page load of the Builder.io homepage website to determine the practical effect of using Qwik. Google Lighthouse is being used to measure the performance of this application. It grades the web page on a scale of 100, on four criterias: performance, accessibility, best practices, and SEO. The results of Builder.io website, built by Qwik framework, is represented as followed:

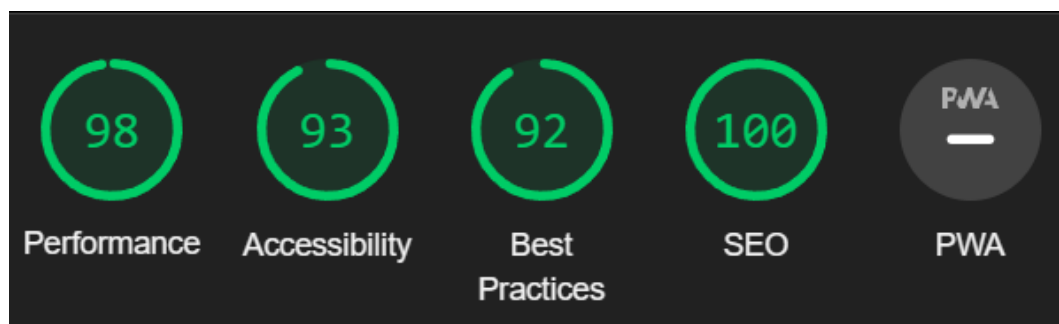


Figure 19. Google Lighthouse's measurement for Builder.io homepage

All four criterias of the Builder.io homepage achieve 90+ scores, which Google indicates as “good” (“Lighthouse performance”, 2019). Lighthouse also provides a detailed insight on smaller-scale performance metrics, including:

- First Contentful Paint
- Speed Index
- Largest Contentful Paint
- Time to Interactive
- Total Blocking Time
- Cumulative Layout Shift

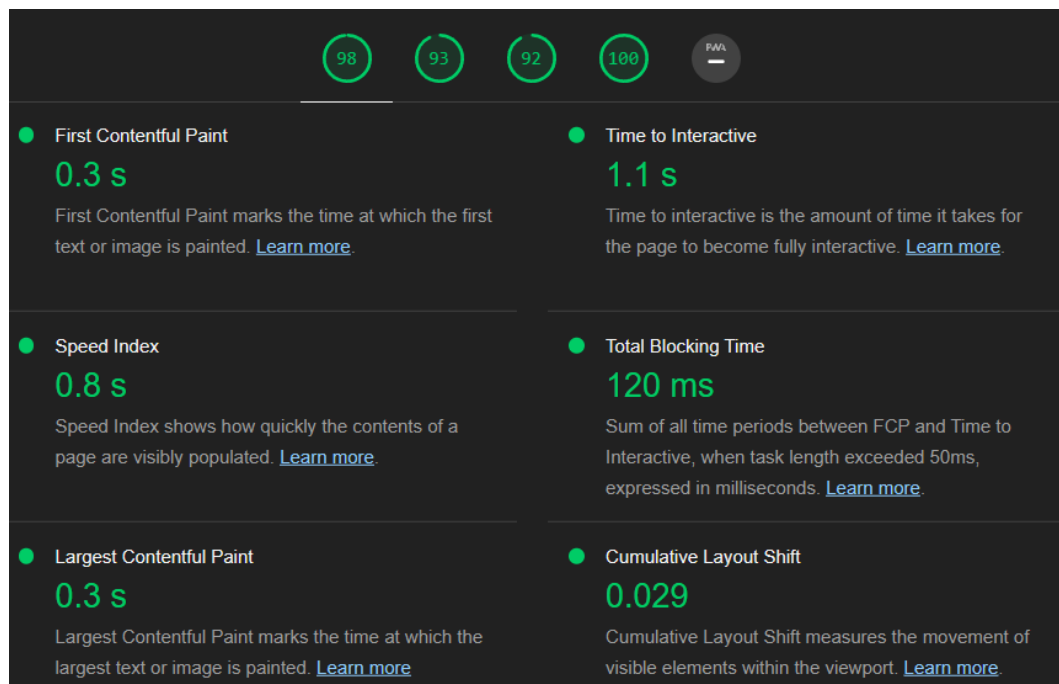


Figure 20. Google Lighthouse’s measurement of performance metrics

It can be concluded from these results that Qwik greatly enhances the Builder.io initial loading page’s efficiency and speed, further proves this thesis statement in the previous section.

3. FILMMASH APPLICATION OVERVIEW

The implementation of an application is a critical phase in the software development lifecycle, where ideas and theories are transformed into a functional product. The following chapters of this thesis document the implementation process of Filmmash - a social web application that allows users to vote on their favorite films, utilizing the previously attained knowledge of Headless CMS and Qwik framework. These sections present all the vital steps in a web application development phase, from pre-analysis, and database and system architecture design to backend and frontend implementation. The sections focus especially on the two discussed topics of this thesis, Headless CMS and Qwik framework, to determine the practical advantages and limitations of these technologies and reinforce what has been studied in previous chapters, by means of performance testing and post-analysis.

This chapter discusses the application overview description, including background and objectives, functionalities, and user flows as well as relevant architecture and technical stacks, and serves as the basis for further implementation.

3.1. Background, Motivation, and Objectives

This thesis has always been dealing with the matter of improving and transforming the current state of technology, therefore, it is only natural that the base idea of this project also came from a pioneer of social web application: Mark Zuckerberg's infamous Facemash website.

In late October 2003, Mark Zuckerberg, then a Computer Science sophomore at Harvard University came up with the idea and single-handedly created Facemash, a "hot-or-not" version of the university, with the compared individuals being the school's female students. Zuckerberg hacked into the online facebook of nine Harvard Houses, illegally retrieved images of students and displayed each two of them side by side, prompting users to decide who the

more attractive person is, and ranked all the students based on the input ratings. He documented his thoughts simultaneously while writing the codes, at one point declaring that he wanted to “put some of these faces next to pictures of farm animals and have people vote on which is more attractive”, before confirming his intention by saying he liked “the idea of comparing two people together”. The website was an immediate success, receiving an impressive 22,000 page views within its first few hours and spreading rapidly across the Harvard campus. Despite the unethical approach, the notoriety of Facemash ultimately proved to be a positive turning point for Zuckerberg. It solidified his reputation as a noteworthy figure among the great minds of the university and paved the way for the future of social media. He utilized the simple yet ingenious concept behind Facemash to create a small social networking site which became a household name – Facebook (Hoffman, 2010).

This thesis’s FIlmmash application, inspired by Facemash, aims to recreate, and improve upon the notorious website, with the compared subject being films due to the author’s personal preferences. There are several reasons for this ideation:

- Technological feasibility: The idea is not complex and can be implemented with the available technology as well as time resources. More importantly, the idea is flexible to integrate the technology of Headless CMS and Qwik framework, which are the main focuses of this thesis.
- Broad audience: Psychologically, it has been scientifically determined that people have a tendency to enjoy evaluating and ranking subjects (Dooley, 2014), and decision makers are greatly affected by ranks of things (Chun & Larrick, 2022). Therefore, it can be concluded that the idea has a guaranteed group of customers. Film rankings, more specifically, has been always generating interest with the example being the decennial Sight and Sound poll (Ebert, 2012).

- Personal interest: The idea is of the author's preferences, and the passion about a particular topic can make it easier to stay motivated and committed to the project.

The main objective of this project is to investigate the practical applications of two specific technologies: Headless CMS and Qwik framework. The purpose is to determine their viability in real-life settings, and to provide empirical evidence to support the theoretical benefits and drawbacks of these technologies. Additionally, the project aims to showcase the detailed process involved in building a web product using these technologies, by providing a step-by-step demonstration of the development process from start to finish. By accomplishing these objectives, this project aims to provide valuable insights into the application of Headless CMS and Qwik framework in web development, and to contribute to the existing knowledge in this area.

3.2. Requirement Analysis

The MoSCoW strategy is used to determine the requirement for Filmmash application. It is a prioritization technique used in project management to help teams determine which requirements or features are essential to a project's success. The term MoSCoW is an acronym for the four categories of prioritization: Must-haves, Should-haves, Could-haves, and Will-not-haves (yet). It is defined as followed ("MoSCoW analysis", 2009):

- Must-haves refers to requirements that are crucial for the final solution to be deemed successful.
- Should-haves refers to requirements that are considered high-priorities and should be incorporated into the solution if feasible.
- Could-haves refers to requirements that are considered desirable but not essential to the success of the project. If time and resources permit, these requirements are usually included in the solution.

- Will-not-haves (yet) refers to requirements that have been collectively decided will not be included in a particular release but may be considered for future implementation.

<p>Must-have</p> <ul style="list-style-type: none"> • Functional web application which calculates users input and ranks choices accordingly • Integration with Contentful and retrieving data using Contentful Content Delivery API • Implementation using Qwik as frontend framework 	<p>Should-have</p> <ul style="list-style-type: none"> • Responsive application to showcase Headless CMS's integration into multiple devices • Login and logout implementation for users
<p>Could-have</p> <ul style="list-style-type: none"> • Enhanced UX/UI design • Specific guidelines on retrieving and uploading content 	<p>Will-not-have-yet</p> <ul style="list-style-type: none"> • Integration with Contentful's Content Management API (to post content directly through application) • Specific endpoint for each film • Delete operation

Table 2. Requirement analysis of Filmmash

3.3. Core Functionality and User Flows

3.3.1. Elo's Rating Algorithm

The application revolves around the main concept of comparing two films and ranking them according to users' inputs. Each turn, it randomly selects two films from a set of pre-made categories, lets users choose their preferred one and calculates the points for each film based on the Elo's rating algorithm.

The Elo rating system is a mathematical method used to calculate the relative skill levels of players in two-player games such as chess. It was invented by Arpad Elo, a Hungarian-born American physics professor and chess player, to replace the previously Harkness system used in chess (Elo, 1967), and has since been widely adopted by many sports governing bodies, board games and even dating applications such as Tinder. It works by calculating the winning probabilities based on players' current ratings, and updating that rating based on the

outcomes of their head-to-head matches, or in simplified terms, one player's loss of points reflects in another player's gain. If a player wins a match against an opponent with a higher rating, the winner's rating will increase more than if they had won against a lower-rated opponent; similarly, if a player loses to a higher-rated opponent, their rating will not decrease as much as if they had lost to a lower-rated opponent. Elo indicated that a player's performance in each game is a random variable that conforms to a probability distribution in the shape of a bell curve over time, meaning that a player's true skill is represented by the average of their random performance variable in the Elo ratings system (Veisdal, 2019).

The Elo's rating algorithm can be divided into two formulas: calculating the expected score and updating the real rating based on performances.

$$E_a = \frac{1}{\left(1 + 10^{\frac{R_b - R_a}{400}}\right)}$$

$$E_b = \frac{1}{\left(1 + 10^{\frac{R_a - R_b}{400}}\right)}$$

E_a and E_b denote the winning probabilities, or the expected change in rating outcome, for player A and player B in a head-to-head match, while R_a and R_b reflect the current ratings of A and B. Without loss of generality, it is assumable that player A wins over player B, in which case the new rating for A and B can be calculated as:

$$\begin{cases} R'_a = R_a + K \times (S - E_a) \\ S = 1 \end{cases}$$

$$\begin{cases} R'_b = R_b + K \times (S - E_b) \\ S = 0 \end{cases}$$

R'_a and R'_b are the new ratings after calculation of the two players, and are calculated by S , denoting the match results in binary terms, 1 for the winner and

0 for the loser. Under normal circumstances, a draw is reflected in S being 0.5 for both players, however, this application refrains from dealing with that situation since the feature requires a decisive choice from the users. The K-factor is a numerical value used in the Elo rating system to determine the amount of rating points gained or lost by a player after a game. The K-factor represents the volatility of a player's rating and is used to adjust the rating change based on the perceived uncertainty of a player's true skill level. There are debates on which K factor is the most accurate. Sonas (2002) (4) indicates that K should be fixed at 24, while the USCF (the original implementer of the Elo's rating) and the FIDE base K on both a player's points and his number of matches previously played. For the sake of simplicity, the application utilizes the original K factor that the USCF used prior to 2013, as followed:

$$K = \begin{cases} 32 & (R < 2100) \\ 24 & (2100 \leq R < 2400) \\ 16 & (R \geq 2400) \end{cases}$$

3.3.2. User Flow

Filmmash provides two main features: ranking the films in a premade category (will be referred to as 'gallery'), by using the discussed Elo's algorithm for public users, and creating galleries by connecting to Contentful content management system for authenticated users.

When first accessing the homepage, users are provided with a list of galleries to choose from based on their personal interests. Users' choice subsequently leads to the gallery's specific page, where users can access three panels:

- The gallery's information panel consists of data regarding the chosen gallery, such as the creator, description, or links.
- The gallery's mash panel is where the core function takes place. Users are presented with two films to choose from, and each turn the users' choice

are recorded, recalculated, and saved into the database, before the application re-randomizes for a new set of films.

- The gallery's rankings panel displays the accumulated ranking results according to users' votes.

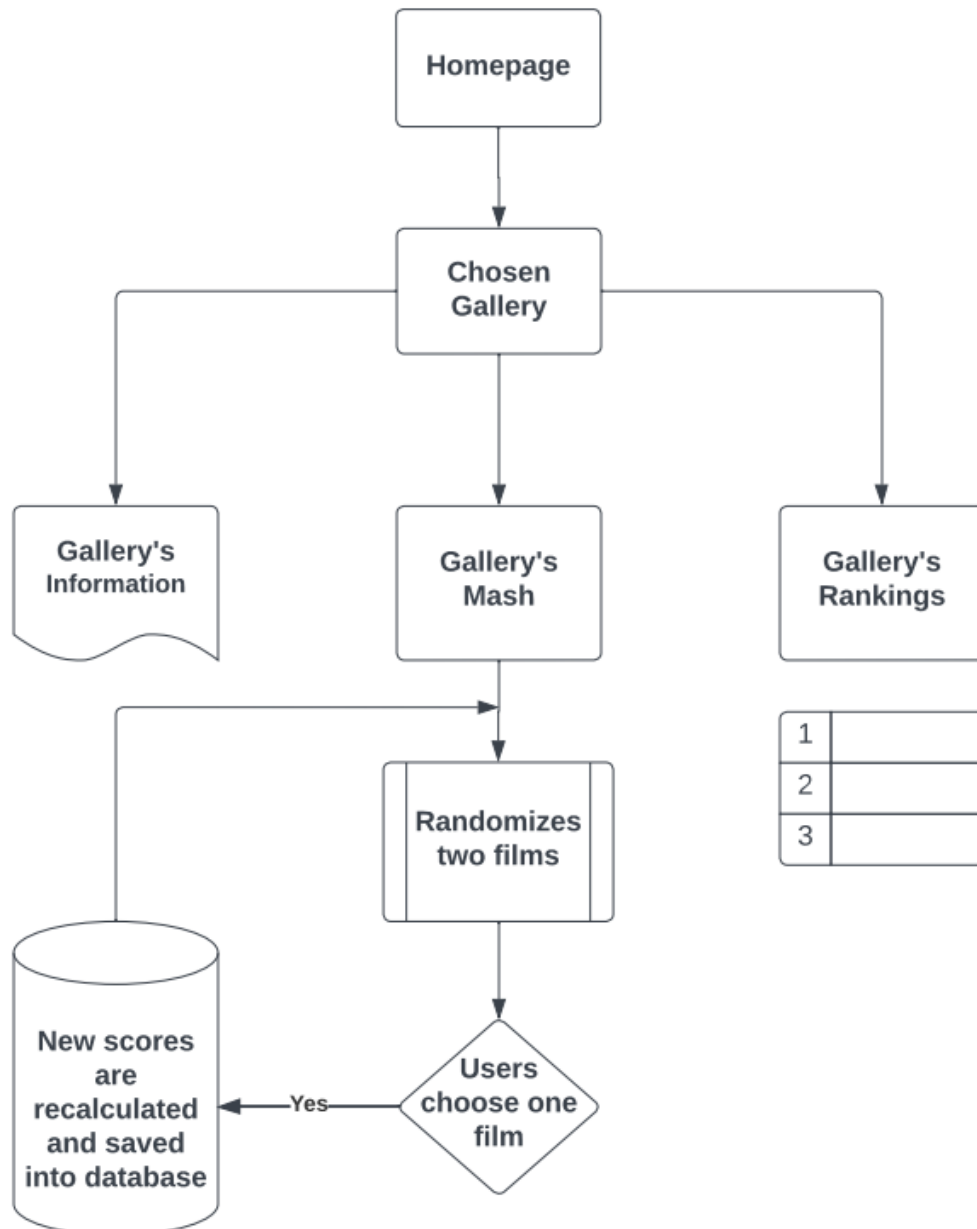


Figure 21. User flow for ranking films in a gallery feature

If users want to create their own galleries of films, users must create a Contentful account and provide the gallery's contents to Contentful's content type for management. After making sure that users are authenticated and authorized for creating galleries, the application redirects to the "Create New Gallery" form. At this stage, users are required to provide various Contentful's key and ID to retrieve contents saved in Contentful to populate the application's own database with.

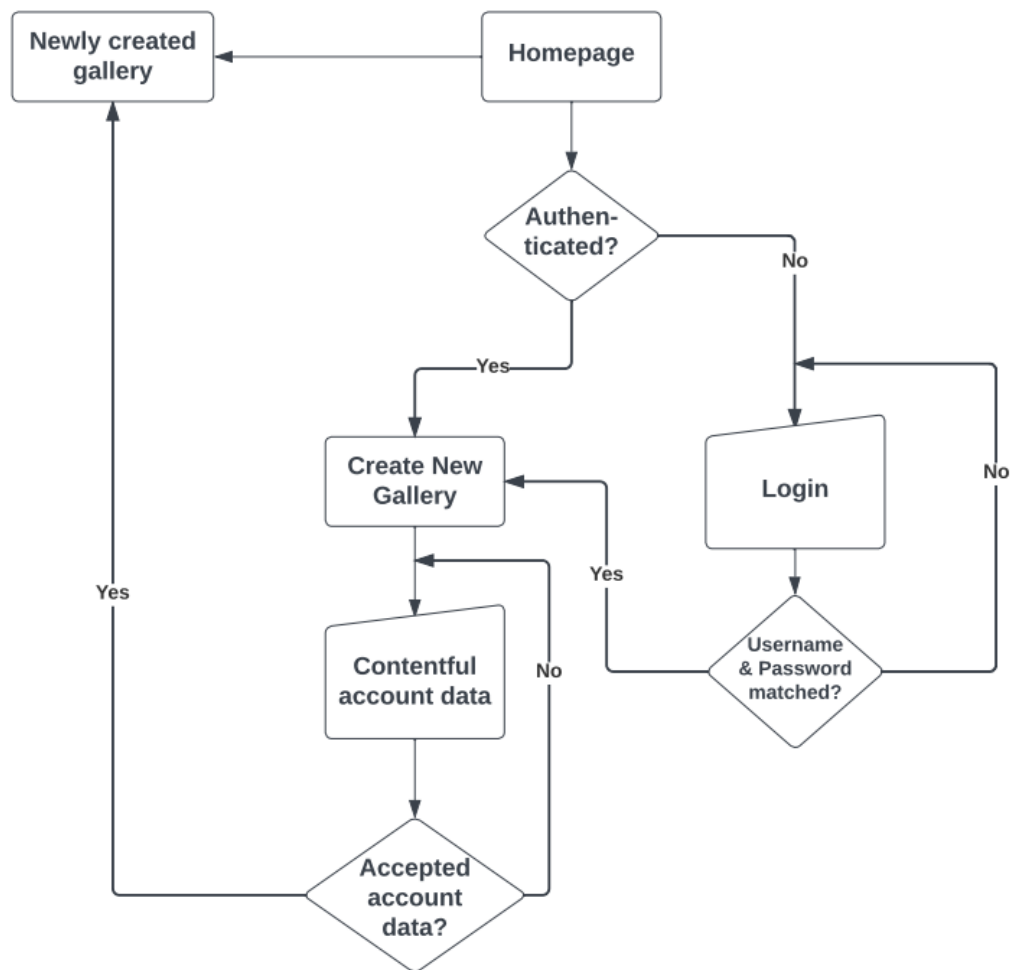


Figure 22. User flow for creating new gallery

3.3.3. Adding and Retrieving Content from Contentful

As one main target of this thesis is to explore the integration of Headless CMS into a real-world case, the application utilizes Contentful, the previously

introduced content management system, as the main method of film data regulation provided by the users. This subsection details the steps to create, upload content to Contentful as well as to retrieve content from the system to plug into Filmmash.

After creating the account, users can create their content types to determine the specific models to integrate into the system. There are two types of content model that Filmmash required from Contentful:

- Gallery's overview information: This content type determines the necessary attributes for a Filmmash's gallery.
- Gallery's film entry: This content type denotes the necessary attributes for a film entry.

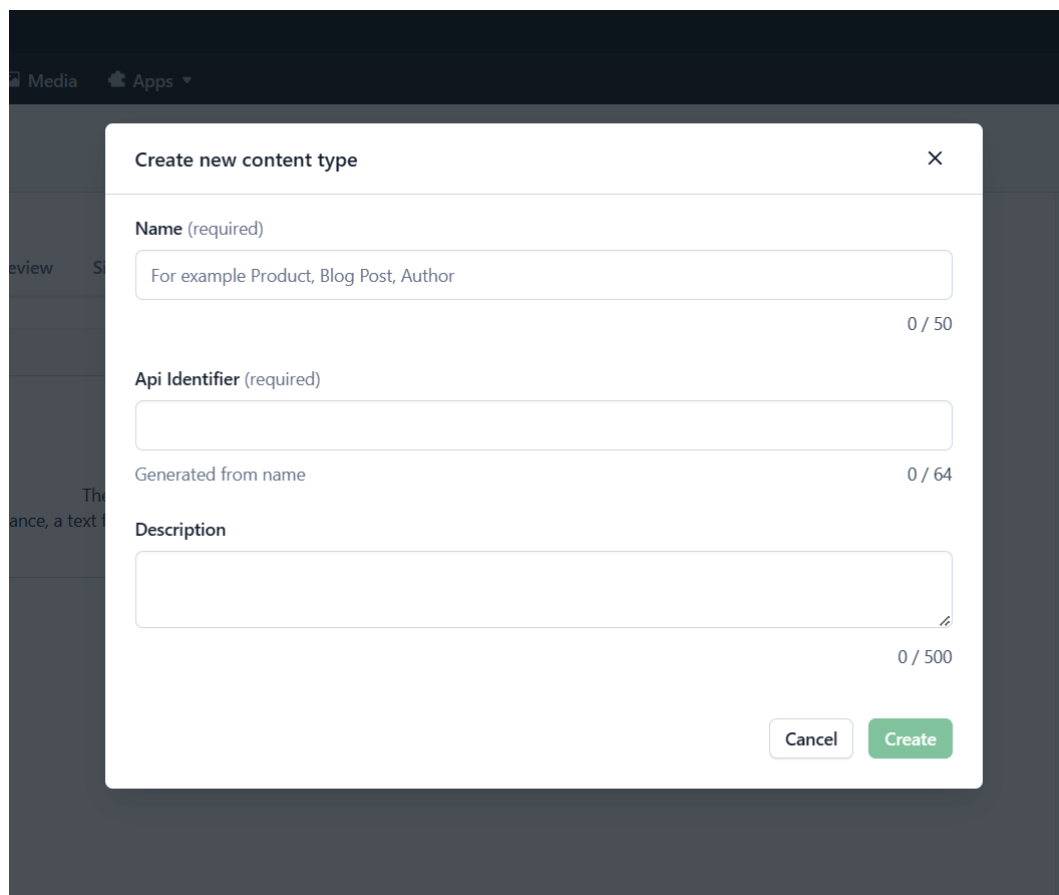

The image shows a 'Create new content type' modal window from the Contentful interface. The modal has a dark header bar with 'Media' and 'Apps' tabs. The main content area is white and contains three input fields: 'Name (required)' with a placeholder 'For example Product, Blog Post, Author' and a character count of '0 / 50'; 'Api Identifier (required)' with a character count of '0 / 64' and a note 'Generated from name'; and 'Description' with a character count of '0 / 500'. At the bottom right, there are 'Cancel' and 'Create' buttons.

Figure 23. Creating a new content type on Contentful

Name ▾	Description	Fields	Updated	By	Status
Greatest Films of All Time		5	a month ago	Me	Active
Greatest Films of All Time - Entries		8	a month ago	Me	Active

Figure 24. The two content types required for a gallery's data retrieving in Filmmash

Then, for each Contentful's model, users can define the specific attributes and their data types, which would be thoroughly detailed in the next chapter. Users subsequently create the contents for each content model, which match the fields already denoted in the respective model.


Greatest Films of All Time

Fields (5)
JSON preview
Sidebar
Entry editors

⋮

Ab name Short text

Entry title

Settings

⋮

⋮

bannerImg Media

Settings

⋮

⋮

avatarImg Media

Settings

⋮

⋮

description Long text

Settings

⋮

⋮

summary Long text

Settings

⋮

Figure 25. The sample content type fields for gallery's information

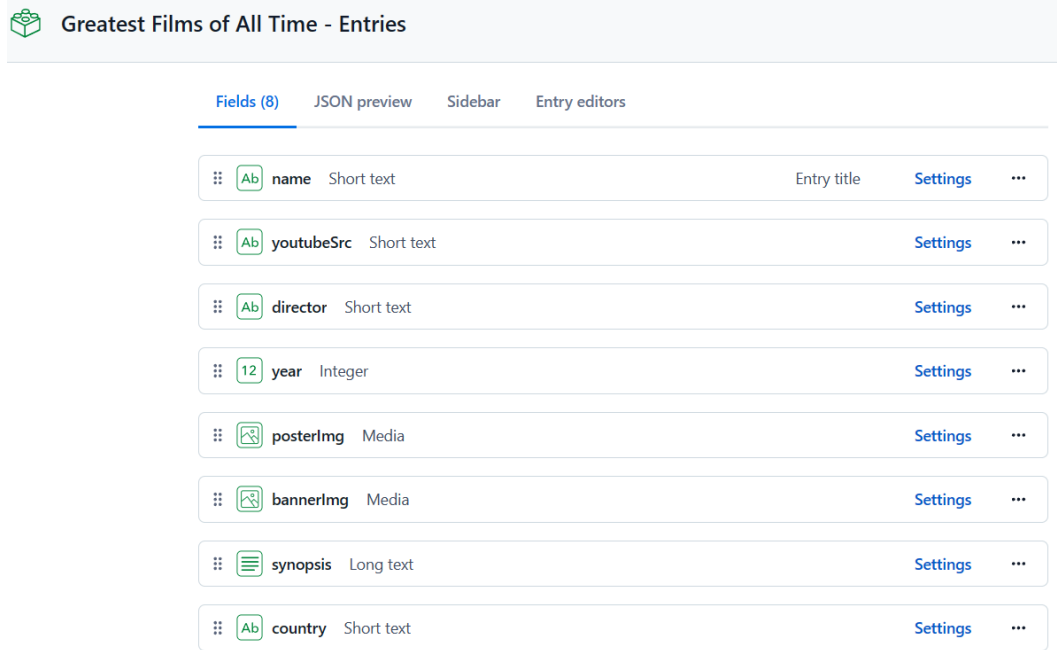


Figure 26. The sample content type fields for a gallery's film entry

The content successfully created in the Contentful system can be plugged into use in Filmmash through the utilization of the following keys:

- Space ID: The specific ID that denotes the users' space where the content models are created
- Environment ID: The specific ID that denotes the users' environment of the space. If users have not specified it from the beginning, the default key "master" will be used.
- Content Delivery API key: The key serves as the connection point for Filmmash to access the Content Delivery API, which is the main
- Gallery's information content type ID: The specific ID for the information model of the gallery. This ID is created by the users when first creating the content type
- Gallery's information content type ID: The specific ID for the film entry model. This ID is also initialized by users upon model creation.

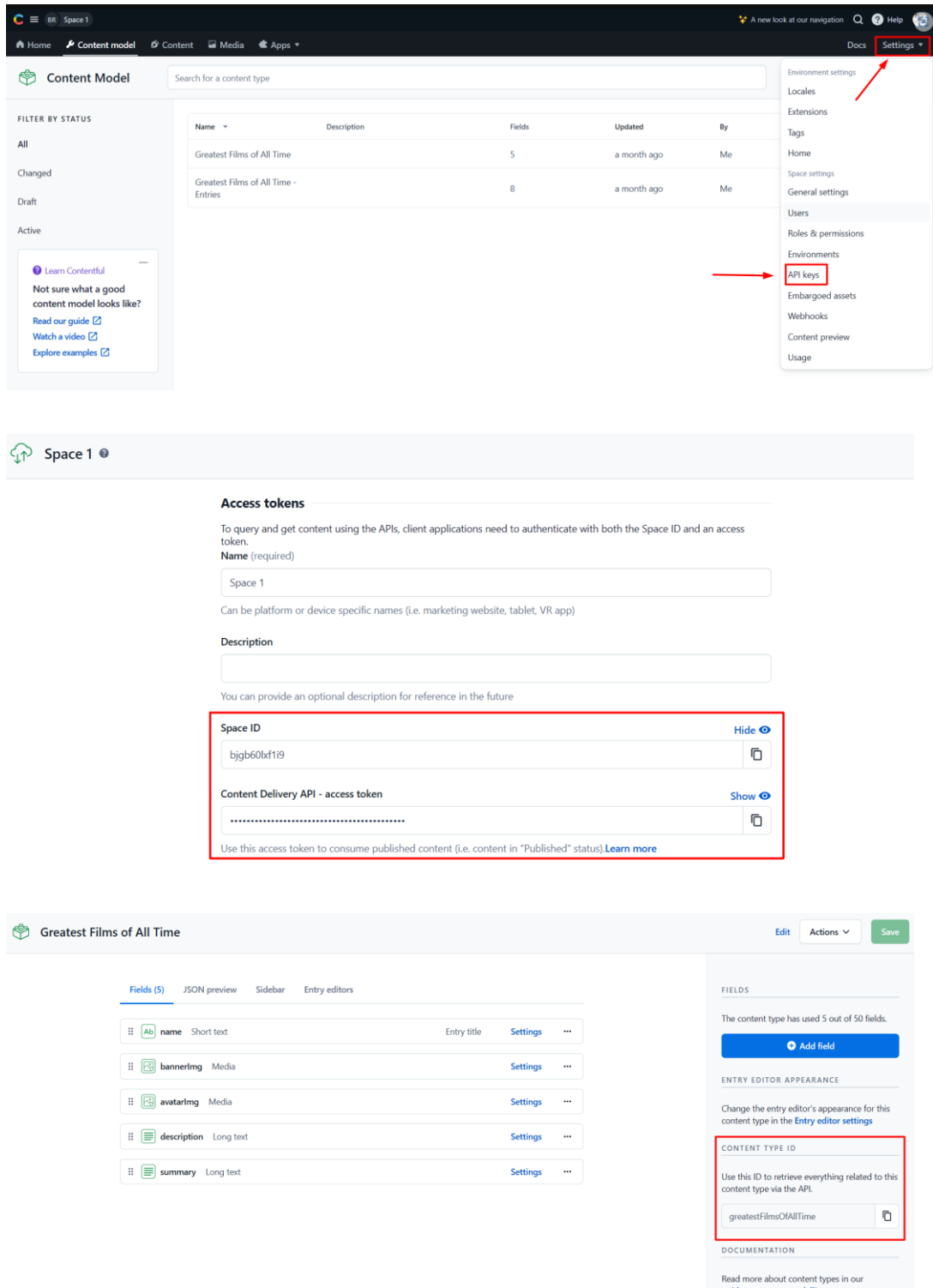


Figure 27. Retrieving the necessary API keys for connecting Contentful and Filmmash

3.4. System Infrastructure and Relevant Technologies

The system infrastructure of a web application is a crucial component that underpins its operation. It consists of all the hardware, software, and network components that are necessary for the web application to function effectively and efficiently. A well-designed system infrastructure ensures that the web application can handle heavy user traffic and data processing without compromising its performance and reliability. It also makes it easier to maintain and upgrade the web application over time, ensuring that it stays up to date with the latest technologies and features (Muzammil, 2021). Therefore, having a robust system infrastructure is essential for the successful operation of any web application.

Filmmash applies a three-layer structure to support the application's functionality. Its infrastructure is detailed as followed:

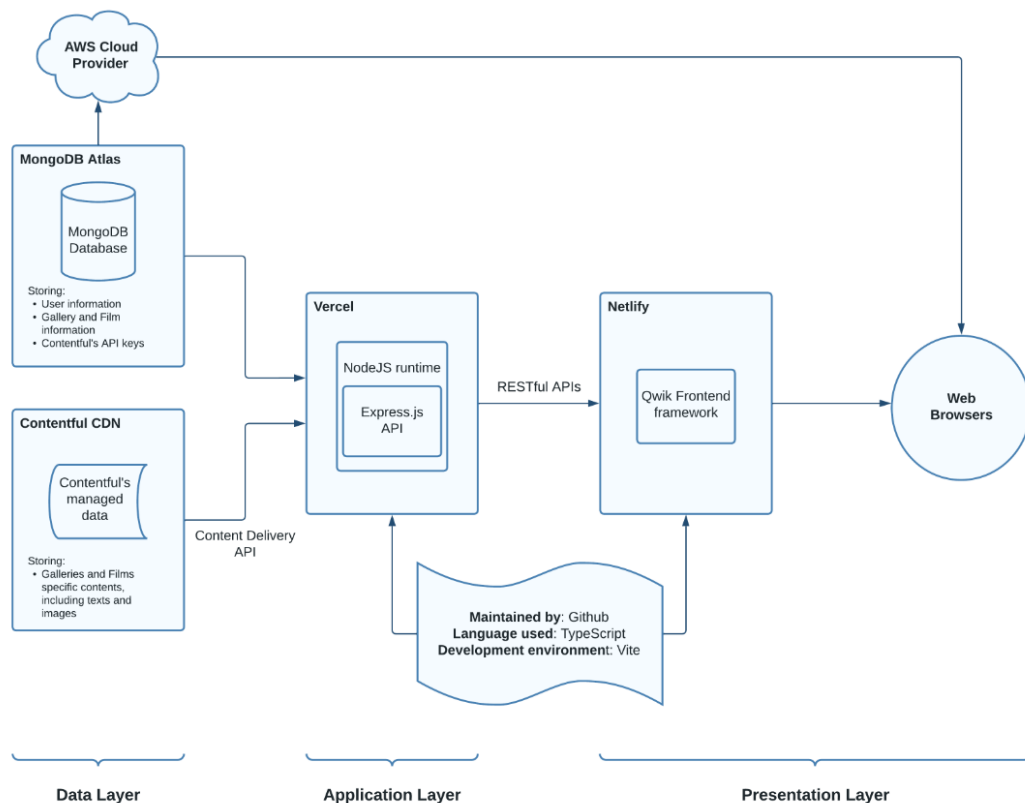


Figure 28. Filmmash's system infrastructure

The data layer, which stores the user login information, as well as holds the basic information and the ratings for every film, uses MongoDB as the main database system. MongoDB is a popular open-source NoSQL document-oriented database that allows developers to store and manage data in a flexible and scalable way. MongoDB is different from traditional relational databases, as it stores data in JSON-like documents with dynamic schemas, which means that each document can have a unique set of fields and data types. Filmmash uses MongoDB for its ability to handle unstructured data, since the application is heavily-content based and supports a variety sets of data types which makes it difficult to utilize relational alternatives. MongoDB is also used with a view for future scaling since it supports sharding, a technique that allows data to be distributed across multiple servers for increased scalability and performance, which is of great benefits with a larger amount of input data (Papiernik, 2021). However, instead of self-deploying databases on MongoDB, Filmmash uses MongoDB Atlas as a multi-cloud database service that provides easier management and deployment, as well as security methods for MongoDB applications. Filmmash uses the free cluster tier of MongoDB Atlas, which is 500 megabytes in size, as its database only stores the most basic collections' entries. There are several cloud platforms that MongoDB provides the integration to deploy data to, such as Google Cloud, Azure, or AWS, the latter of which is decided to be used by Filmmash. As mentioned, while the basic application's information is saved to MongoDB, the contents created by users are regulated and managed by Contentful and plugged to the backend system through Contentful's Content Delivery API.

Filmmash's server is built on Node and Express.js. Node.js is a powerful and popular open-source runtime environment for building server-side applications. Its most innovative aspect comes from the fact that it allows users to build the server using fully JavaScript, so that there is no conundrum of having to learn two languages to write an application. As Filmmash handles a considerable amount of traffic and requests while also aims for the highest performance, Node.js proves to be the most versatile with its minimal overhead cost, as well as

a rich ecosystem of modules, packages, and libraries to be integrated within, including Express. Express.js is a minimalist web framework for Node.js, commonly regarded as the standard Node.js framework (Serby, 2012), that provides a set of features for building web applications and APIs. It simplifies the process of building server-side applications by providing a robust set of HTTP utility methods and middleware functions that make it easy to handle requests and responses. MongoDB, Node.js and Express, together form the MEN backend stack, which plays a crucial role in nowadays web application, as it is an integral part of most popular tech stacks such as MERN or MEAN once connected to the frontend. As one of the main objectives of Filmmash is to investigate the practical use of Qwik, this framework is the utilized technology for the frontend.

The application is deployed on Vercel for the server side, and Netlify for the client side. Vercel is a cloud-based platform that specializes in serverless deployment of web applications. Its features, including generated SSL certificates, edge caching, and a global CDN for fast and reliable performance makes it the perfect candidate for hosting the server of Filmmash, which requires an automatic process to simplify the procedure. Netlify, on the other hand, is a cloud-based platform that specializes in static website hosting and deployment. As Qwik is encouraged to connect to Netlify for the increasing productivity the platform provides (Postma, 2022), it is the chosen deploy companion for the client side.

Instead of choosing pure JavaScript, Filmmash chooses TypeScript as the main language to be written on. Developed by Microsoft in 2012, it can be viewed as a superset of JavaScript, or “JavaScript with types”, and was created to resolve the problems of using JavaScript in a business scale, particularly dynamic typing. Dynamic typing, while allowing for more flexibility, can often result in bugs that impede the progress of programmers and can lead to increased costs for adding new code. The lack of features like types and compile-time error checks in JavaScript can be problematic for server-side code in larger companies and

codebases. Barr et al. (2017) indicate that TypeScript can detect up to 15% of JavaScript bugs, which can be considered a huge number regarding all the possibilities there are.

4. FILMMASH APPLICATION IMPLEMENTATION

This chapter is dedicated to describing the comprehensive process of implementing Filmmash. It is divided into several subsections which walk through the various development stages, including database design, server-side and client-side implementation as well as deployment. The section aims to achieve its goals by examining architectural diagrams, written code snippets as well as user interface examples that help illustrate the development process, consequently providing an in-depth understanding of the technical aspects involved in the development of Filmmash and how each component was implemented to achieve the final product.

4.1. Database Design

The application consists of three main models: Gallery, Film and User. While all the information regarding users is stored in MongoDB, the metadata of galleries and films are stored by Contentful management system, whose specific keys are stored in the respective models in MongoDB for content retrieval purpose. This method, as discussed, helps Filmmash to explore the possible advantages of using Headless CMS in a real-life production. The following figure describes the entity relationship diagram of the Filmmash application.

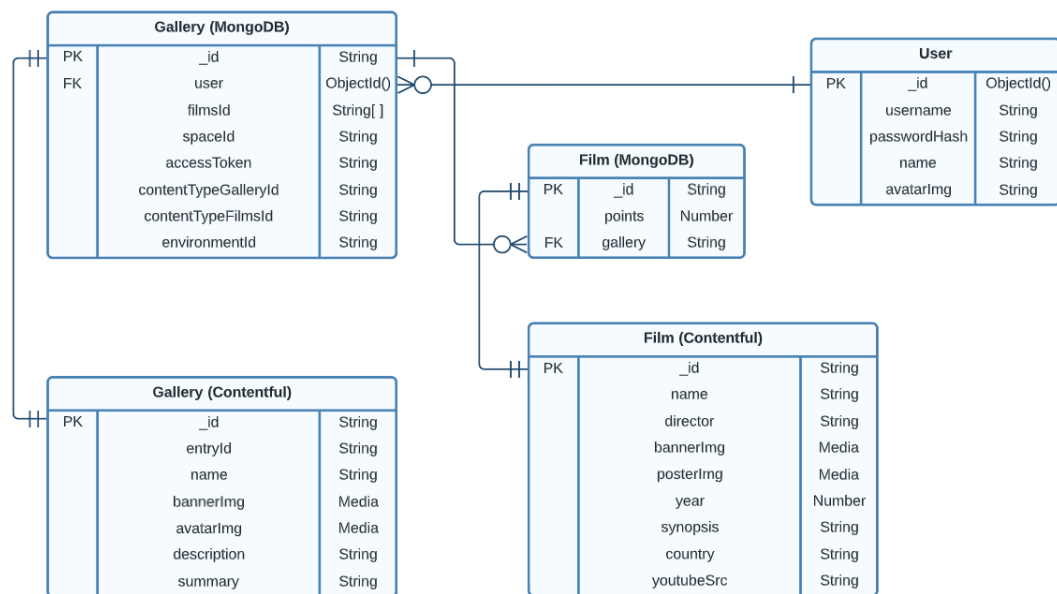


Figure 29. Filmmash entity relationship diagram

The Gallery model in Contentful consists of attributes that provide an informative overview about each gallery of films. This includes its name, summary, and description, as well as the avatar and banner images for styling purposes, and is linked with the Gallery model stored in MongoDB through its unique ID, hence making the relationship one-and-only-one. The Gallery model in MongoDB stores the discussed keys and configurations to retrieve content from the Contentful model.

The Film models of Contentful and MongoDB function in a relatively similar way, with meta-content of a film (name, directors, images and such...) is saved in Contentful, while the points of each film, based on users' votes, are saved in MongoDB for better server execution and adaption. Since every film input by users belongs to one gallery, and one gallery can contain many films, it is vital that this many-to-one relationship is reflected by a foreign key in the Film model that connects each film to its respective gallery.

Filmmash also provides authentication for users, since only authenticated and authorized users are given the rights to submit new galleries. Its model in

MongoDB consists of basic user information such as name, username, avatar image and a hashed password. Each user can create many galleries, hence reflecting a one-to-many relationship, with the Gallery model holding a foreign key that directs to its creator.

4.2. Backend Implementation

Filmmash uses Node.js and Express as the underlying framework for server-side development. This subsection explores the structure of the backend code and dives deep into explaining the process of constructing it.

4.2.1. Backend Code Structure

The following figure describes the Filmmash backend code structure.

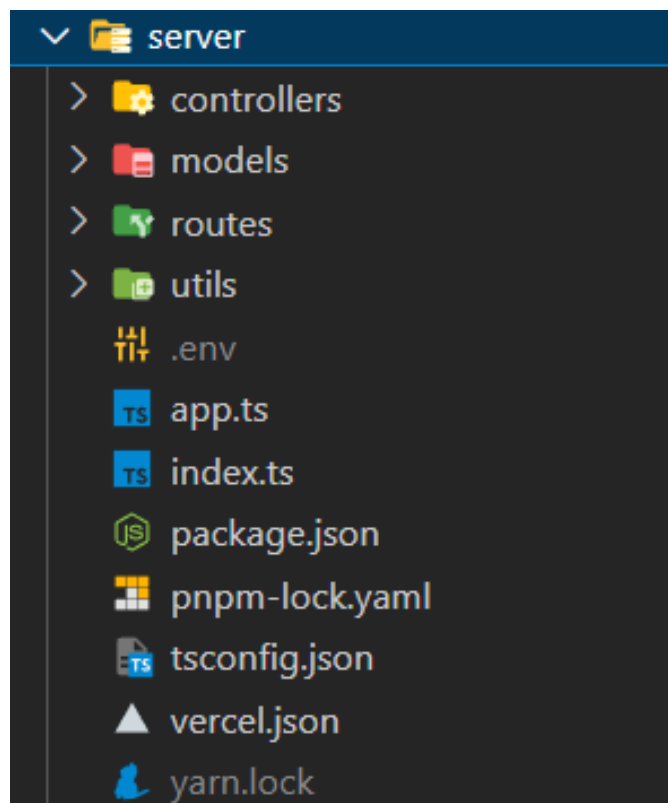


Figure 30. Filmmash backend code structure

The code can be divided into two parts: the configurations and the main functionalities. The backbone of `package.json`, which is created whenever a

Node.js project is initialized. It is a metadata file used to describe various aspects of the project, including its name, version, dependencies, and other relevant information. As it is described in the following figure, Filmmash `package.json` file contains the information of libraries and packages that help build the foundation of its backend, notably Express, Mongoose and Contentful. The function of each package will be further elaborated on its use.

```
/** package.json */  
  
{  
  "name": "filmmash",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.ts",  
  "scripts": {  
    "start": "node index.ts",  
    "dev": "nodemon index.ts",  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "keywords": [],  
  "author": "Xuan-An Cao",  
  "license": "ISC",  
  "devDependencies": {  
    "@types/node": "^18.8.3",  
    "nodemon": "^2.0.20"  
  },  
  "dependencies": {  
    "bcrypt": "^5.1.0",  
    "contentful": "^9.3.1",  
    "cookie-parser": "^1.4.6",  
    "cors": "^2.8.5",  
    "dotenv": "^16.0.3",  
    "express": "^4.18.2",  
    "express-async-errors": "^3.1.1",  
    "jsdom": "^20.0.1",  
    "jsonwebtoken": "^9.0.0",  
    "mongoose": "^6.6.5",  
    "mongoose-unique-validator": "^3.1.0",  
    "node-fetch": "^2.6.1",  
    "ts-node": "^10.9.1",  
    "typescript": "^4.8.4",  
    "vercel": "^28.4.14"  
  }  
}
```

Code Snippet 1. Backend's package.json file

Since TypeScript is the main language used for the project, a `tsconfig.json` file is required to specify compiler options and settings to determine how to compile TypeScript to JavaScript. The configuration part also contains an `.env` file, which is of utmost importance since it stores configuration variables and sensitive information that should not be committed to version control. The file typically contains key-value pairs, with each pair representing a configuration variable and helps developers to keep sensitive information out of the codebase and prevent

it from being accidentally exposed or leaked. In Filmmash, `.env` is used to store information regarding MongoDB Atlas client's URI and used server port.

```
/** tsconfig.json */  
  
{  
  "compilerOptions": {  
    "types": ["node"],  
    "typeRoots": ["node_modules/@types"],  
    "esModuleInterop": true,  
    "jsx": "react",  
    "lib": ["es2021", "DOM"]  
  },  
  "exclude": [  
    "node_modules"  
  ]  
}
```

Code Snippet 2. Backend's tsconfig.json file

```
/** env_example */  
  
MONGODB_URI='mongodb+srv://<username>:<password>@<clustername>.ja8pid0.mongodb.net/?  
retryWrites=true&w=majority'  
PORT=PORT_NUMBER  
SECRET=SECRET_KEY
```

Code Snippet 3. Backend's example of an .env file

As for the main functional part, as defined in the `package.json` file, the starting point for the backend is the `index.ts` file. When running a Node-based application, Node.js searches for a main file to run as configured in `package.json`, and in the case of Filmmash the default file is `index.ts`. This file typically contains the initialization code for the application, including setting up the server, defining routes, applying middlewares, and connecting to a database. However, for clarification, this file is divided into two, with `app.ts` being the definitive file for creating the Filmmash backend application, while `index.ts` imports the application created and connects it to the server.

The code of `index.ts` and `app.ts` is based on several repositories which are constructed to represent a MVC structure: models, controllers, and routers. The `models` folder contains the data models, which define the structure and behavior of the application's data, which is previously discussed in the database

design subsection. The ``controllers`` folder consists of the application's main logical behavior, which acts as an intermediary between the models and the application interface by handling requests from clients, processing data and returning responses which reflect users' interactions. These handlers are regarded as the CRUD operations, which is an acronym of the four main operations for implementing a backend application: create, read, update, and delete. These controllers' functionalities are then connected to their respective routes which are called in the ``routers`` directory, which defines the application's API endpoints and the HTTP methods that can be used to access those endpoints. The application also contains a ``utils`` directory for denoting the basic utilities that Filmmash needs which consists of reusable functions such as loggers, error handlers, middlewares or codes to represent separately specific tasks, including connecting to Contentful or JWT.

These directories will be explained thoroughly and separately in the next subsections, before combining with the elaboration of the ``app.ts`` and ``index.ts`` files to portray a vivid representation of Filmmash backend application.

4.2.2. Data Models

Filmmash data models, as reflected from the entity relationship diagram, consists of Gallery, Film and User.

```

/** models/gallery.ts */

import mongoose, { Schema, Types } from "mongoose";

export interface IGallery extends Document {
  _id: string;
  user: Types.ObjectId;
  filmsId: string[];
  spaceId: string;
  accessToken: string;
  contentTypeGalleryId: string;
  contentTypeFilmsId: string;
  environmentId: string;
}

const gallerySchema = new mongoose.Schema({
  _id: String,
  user: { type: Schema.Types.ObjectId, ref: "User" },
  filmsId: [String],
  spaceId: String,
  accessToken: String,
  contentTypeGalleryId: String,
  contentTypeFilmsId: String,
  environmentId: String,
});

const Gallery = mongoose.model<IGallery>("Gallery", gallerySchema);

export default Gallery;

```

Code Snippet 4. Gallery data model

```

/** models/film.ts */

import mongoose from "mongoose";

export interface IFilm extends Document {
  _id: string;
  points: number;
  gallery: string;
}

const filmSchema = new mongoose.Schema({
  _id: String,
  points: Number,
  gallery: { type: String, ref: "Gallery" },
});

const Film = mongoose.model<IFilm>("Film", filmSchema);

export default Film;

```

Code Snippet 5. Film data model

```

/** models/user.ts */
import mongoose from "mongoose";
import uniqueValidator from "mongoose-unique-validator";

export interface IUser extends Document {
  name: string;
  username: { type: string; required: true; unique: true };
  passwordHash: string;
  avatarImg?: string;
}

const userSchema = new mongoose.Schema({
  username: String,
  name: { type: String, required: true, unique: true },
  passwordHash: String,
  avatarImg: String,
});

userSchema.plugin(uniqueValidator);

userSchema.set("toJSON", {
  transform: (document, returnedObject) => {
    // the passwordHash should not be revealed
    delete returnedObject.passwordHash;
  },
});

const User = mongoose.model("User", userSchema);

export default User;

```

Code Snippet 6. User data model

The models are created using Mongoose, a library that connects MongoDB and Node.js to initialize and configure data schemas. Each file consists of a defined TypeScript interface and uses the constructor function `mongoose.Schema()` to define the structure, behavior, and functionalities of MongoDB documents. The fields and their properties strictly follow the entity relationship diagram, with foreign keys represented by references to the connected models. It should be noted in the User model that the schema uses a plugin called `mongoose-unique-validator` to determine the singularity of each entity, since clients' usernames cannot match with each other's. The users' passwords stored in MongoDB database are the hashed versions of the originals, a concept which is further discussed in the later section and are removed from the JSON output. This is done since the inclusion of hashed credentials in JSON files can lead to a false authorization, meaning that if an untrusted client successfully retrieves the content of these files, this client is able to obtain fraudulent access to the account linked to the subsequent passwords. By configuring the JSON file with

the option `toJSON`, Filmmash ensures that the schema is always serialized to JSON in the desired format.

4.2.3. Routers and Controllers: Gallery Model

The Gallery model supports three functionalities: create a new gallery, read all the Contentful data of all galleries, and read the data of a single gallery.

The following sequence diagram and code snippets describe the steps of creating

a new gallery:

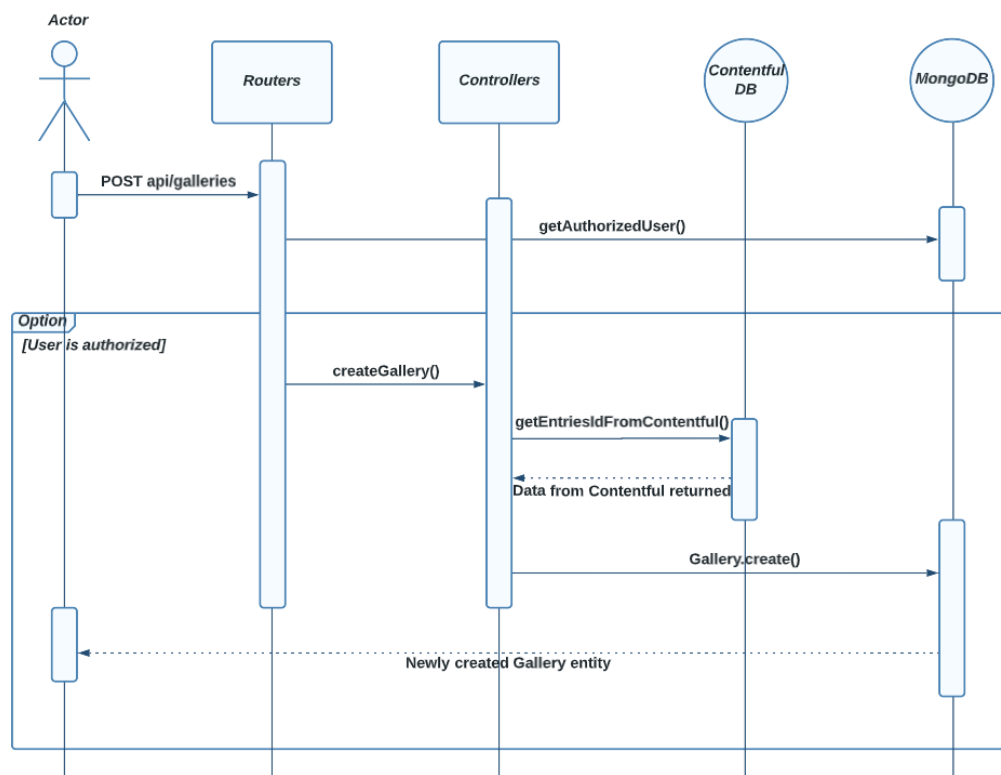


Figure 31. Sequence diagram for creating new gallery

```

/** routes/galleries.ts */

galleriesRouter.post(
  "/",
  async (req: express.Request, res: express.Response) => {
    const user = await getAuthorizedUser(req, res);

    const {
      spaceId,
      accessToken,
      contentTypeGalleryId,
      contentTypeFilmsId,
      environmentId,
    } = req.body;

    const gallery = await createGallery(
      {
        spaceId,
        accessToken,
        contentTypeGalleryId,
        contentTypeFilmsId,
        environmentId,
      },
      user._id.toString()
    );
    res.status(201).json(gallery);
  }
);

```

```

/** controllers/galleries.ts */

const createGallery = async (
  {
    spaceId,
    accessToken,
    contentTypeGalleryId,
    contentTypeFilmsId,
    environmentId,
  }: {
    spaceId: string;
    accessToken: string;
    contentTypeGalleryId: string;
    contentTypeFilmsId: string;
    environmentId: string;
  },
  userId: string
) => {
  const galleryFilms = await getEntriesIdFromContentful(
    spaceId,
    accessToken,
    contentTypeFilmsId,
    environmentId
  );

  const gallery = await Gallery.create({
    spaceId,
    accessToken,
    contentTypeGalleryId,
    contentTypeFilmsId,
    environmentId,
    user: userId,
    filmsId: galleryFilms,
    _id: `${spaceId}-${contentTypeGalleryId}`,
  });

  return gallery;
};

```

Code Snippet 7. Backend code for creating new gallery

When the browser receives a POST request to `/api/galleries/` endpoint, it understands that a user is trying to create a new gallery. As previously stated, only authenticated users are granted the permission to create one, therefore, the first step is to check whether the user is authorized for such actions. The router subsequently calls the `createGallery()` function to the controllers, using the credentials input by the users, which are in turn used to retrieve the necessary Contentful metadata before Mongoose finalizes the creation with its `create()` method.

There are two endpoints for retrieving multiple gallery documents based on the purpose, with the methods being relatively similar to each other. To read all galleries' metadata, users send a GET request to `/api/galleries/`, and if one requires all the galleries' data of a specific user, a GET request to `/api/galleries/users/:userId` is sent. For both endpoints, the router fires a `getAllGalleries()` function to the controller, which searches the database for the requested galleries' credentials. These credentials are returned and subsequently used to get all the galleries' metadata from Contentful.

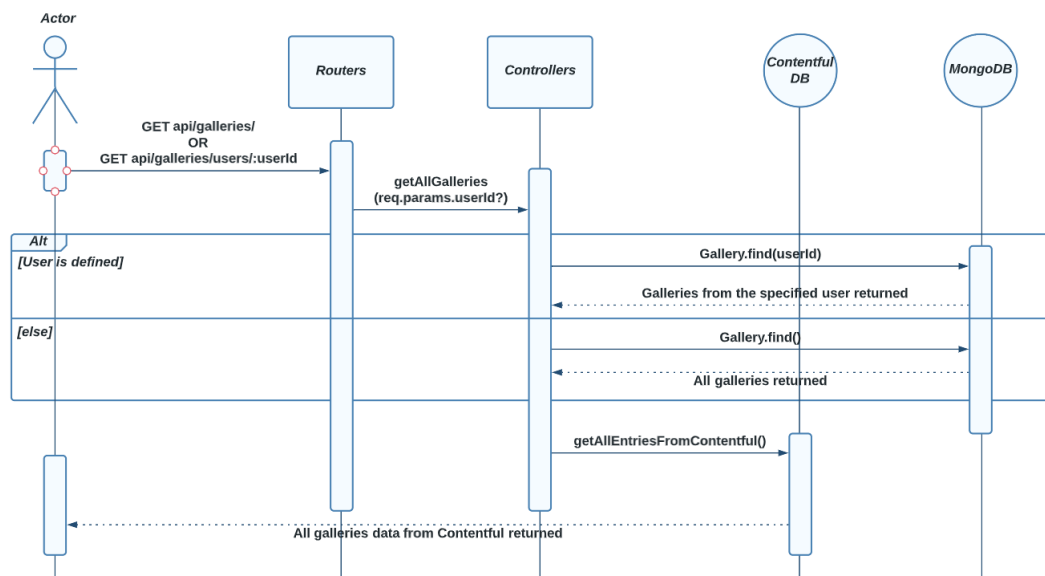


Figure 32. Sequence diagram for getting multiple galleries

```

/** routes/galleries.ts */

galleriesRouter.get(
  "/",
  async (req: express.Request, res: express.Response) => {
    const galleries = await getAllGalleries();
    res.json(galleries);
  }
);

galleriesRouter.get(
  "/user/:userId",
  async (req: express.Request, res: express.Response) => {
    const galleries = await getAllGalleries(req.params.userId);
    res.json(galleries);
  }
);

/** controllers/galleries.ts */

const getAllGalleries = async (user?: string) => {
  const galleries = await Gallery.find(user ? { user } : {});
  const galleriesData = galleries.map(async (gallery) => {
    const { spaceId, accessToken, contentTypeGalleryId, environmentId } =
      gallery;
    const galleryData = await getAllEntriesFromContentful(
      spaceId,
      accessToken,
      contentTypeGalleryId,
      environmentId
    );
    return { _id: gallery._id, user: gallery.user, ...galleryData[0] };
  });
  const resolvedGalleriesData = await Promise.all(galleriesData);
  return resolvedGalleriesData;
};

```

Code Snippet 8. Backend code for getting multiple galleries

Lastly, the method for reading a specific gallery document also bears resemblance to the discussed process, with an endpoint directed towards the galleryID being used. It is noted that the function used to get one gallery in this case is still the `getAllEntriesFromContentful()`, since as explained, each Gallery content model in Contentful should contain one and only one entity of gallery, hence, the function can be reused with only the first index (index 0) returned.

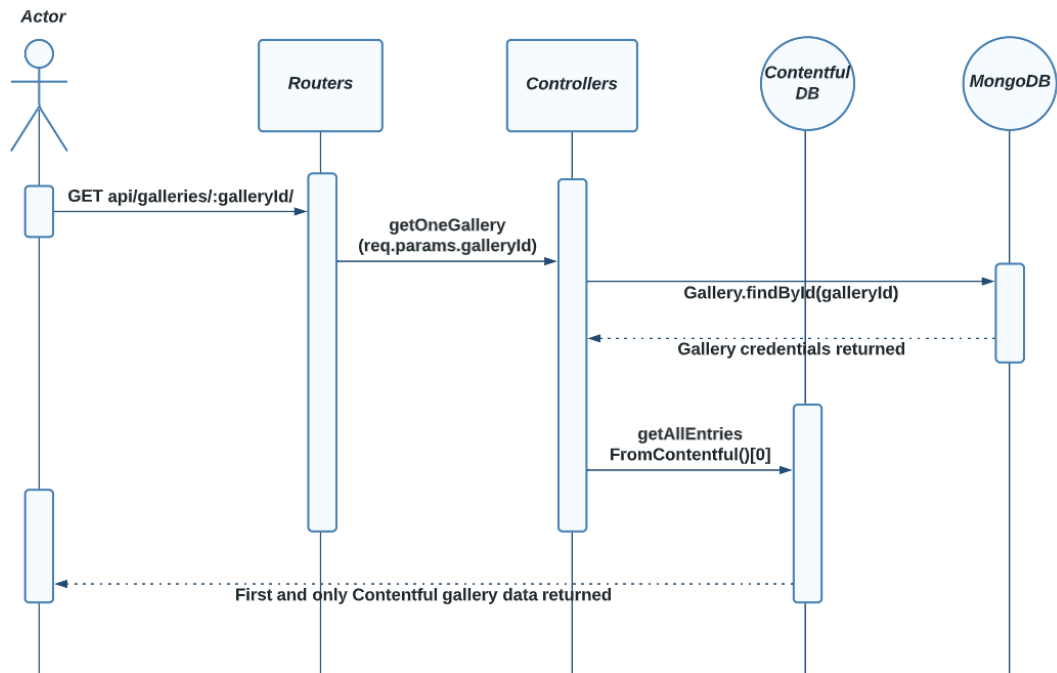


Figure 33. Sequence diagram for getting one gallery

```

/** routes/galleries.ts */

galleriesRouter.get(
  "/:galleryId",
  async (req: express.Request, res: express.Response) => {
    const gallery = await getOneGallery(req.params.galleryId);
    res.json(gallery);
  }
);

/** controllers/galleries.ts */

const getOneGallery = async (id: string) => {
  const gallery = await Gallery.findById(id);
  const { spaceId, accessToken, contentTypeGalleryId, environmentId } = gallery;
  const galleryContentfulData = await getAllEntriesFromContentful(
    spaceId,
    accessToken,
    contentTypeGalleryId,
    environmentId
  );
  const galleryData = {
    _id: gallery._id,
    user: gallery.user,
    filmsId: gallery.filmsId,
    ...galleryContentfulData[0],
  };
  return galleryData;
};

```

Code Snippet 9. Backend code for getting one gallery

4.2.4. Routers and Controllers: Film Model

The Film model also contains three functionalities, albeit with a few differences from the Gallery model: getting all the films of a gallery, getting a specific film, and updating the points of a film. Since all the film metadata are stored in Contentful, the create and update process are left for the content management system and only the reading and updating points process are managed in the backend.

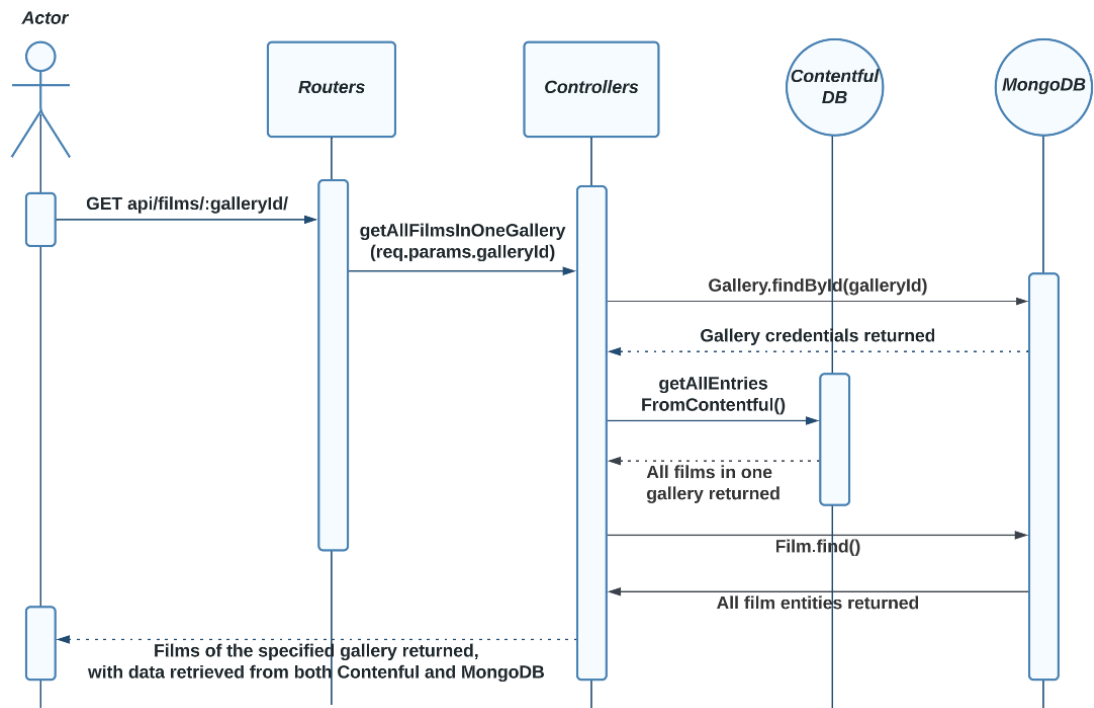


Figure 34. Sequence diagram for getting all films in a gallery

```

/** routes/films.ts */

filmsRouter.get(
  "galleryId",
  async (req: express.Request, res: express.Response) => {
    const films = await getAllFilmsInOneGallery(req.params.galleryId);
    res.json(films);
  }
);

/** controllers/films.ts */

const getAllFilmsInOneGallery = async (galleryId: string) => {
  const gallery = await Gallery.findById(galleryId);
  const { spaceId, accessToken, contentTypeFilmsId, environmentId } = gallery;

  const rawFilms = await Film.find({});

  const existedFilms = await getAllEntriesFromContentful(
    spaceId,
    accessToken,
    contentTypeFilmsId,
    environmentId
  );

  const allFilms = existedFilms.map((film) => {
    const matchedFilm = rawFilms.find(
      (rawFilm) => film.entryId === rawFilm._id
    );
    const points = matchedFilm?.points || 1400;
    return { ...film, points, _id: film.entryId };
  });

  return allFilms;
};

```

Code Snippet 10. Backend code for getting all films in one gallery

For usage in the client side, films retrieved from a gallery must combine both data from Contentful and MongoDB. Therefore, after determining the specific gallery using the endpoint galleryID, the controllers get all the films from the respective Gallery model in Contentful as well as those from MongoDB. The returned results are bundled so that all film documents hold data from both databases.

Retrieving a specific film is also the same as retrieving a specific gallery, with the endpoint directed towards the filmID.

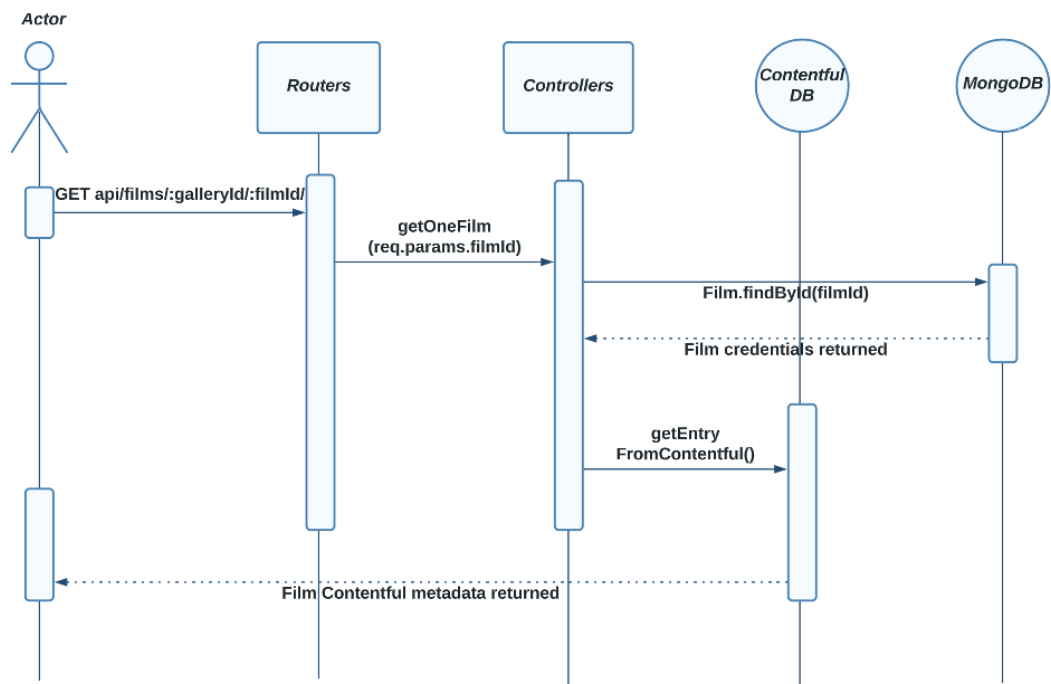


Figure 35. Sequence diagram for getting one specific film

Instead of calling directly to the gallery containing the searched film using its ID, Mongoose's `populate()` method is used, which can expand reference fields in a document with actual objects from other collections. By using this method, the controllers get the necessary Contentful credentials to retrieve the film data.


```

/** routes/films.ts */

filmsRouter.get(
  "/:galleryId/:filmId",
  async (req: express.Request, res: express.Response) => {
    const film = await getOneFilm(req.params.filmId);
    res.json(film);
  }
);

/** controllers/films.ts */

const getOneFilm = async (filmId: string) => {
  const film = await Film.findById(filmId).populate<{
    gallery: IGallery;
  }>("gallery");
  const { spaceId, accessToken, environmentId } = film.gallery;

  const filmData = await getEntryFromContentful(
    spaceId,
    accessToken,
    filmId,
    environmentId
  );

  return {
    ...filmData,
    points: film.points,
    _id: filmId,
    gallery: film.gallery,
  };
};

```

Code Snippet 11. Backend code for getting a specific film

The point updating process is straightforward, with the `findByIdAndUpdate()` method used. It is noted that there is a possibility of unsynchronization between the film data in Contentful and MongoDB, hence, there are films whose IDs may appear in the endpoints but have not been added to the MongoDB database. In this case, the `upsert` property of the `findByIdAndUpdate()` method is used, meaning that if the film has not yet been included in the database prior to its update, then the update process will also create a new document with the updated data.

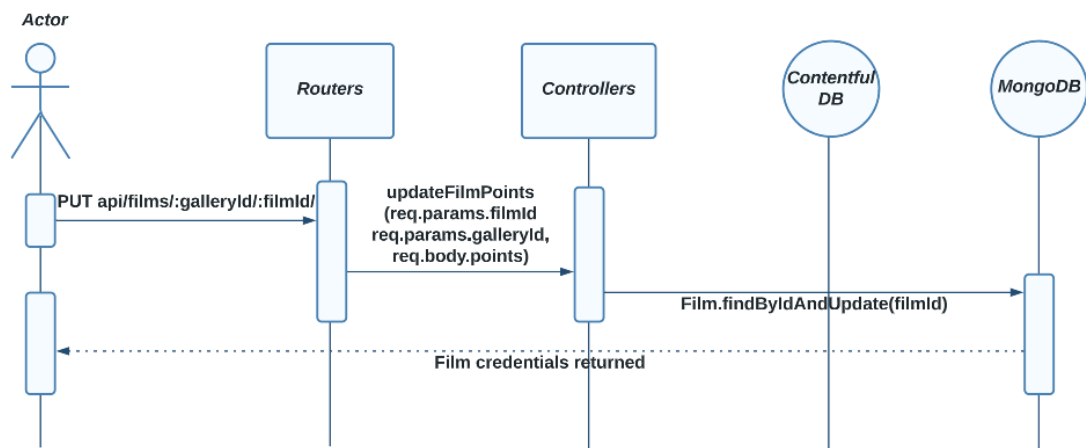


Figure 36. Sequence diagram for updating a film's points

```

/** routes/films.ts */

filmsRouter.put(
  "("/:galleryId/:filmId",
  async (req: express.Request, res: express.Response) => {
    const film = await updateFilmPoints(
      req.params.filmId,
      req.params.galleryId,
      req.body.points
    );
    res.json(film);
  }
)

/** controllers/films.ts */

const updateFilmPoints = async (
  filmId: string,
  galleryId: string,
  points: number
) => {
  const film = await Film.findByIdAndUpdate(
    filmId,
    { points, gallery: galleryId },
    { upsert: true }
  );
  return film;
};

```

Code Snippet 12. Backend code for updating a film's points

4.2.5. Routers and Controllers: User Model

The user model consists of three functionalities: getting a user, logging in and signing up, the latter two being closely related to each other. Reading a user's data is a simple process with only the username of the user required in the endpoint, being passed down to the MongoDB database for retrieval.

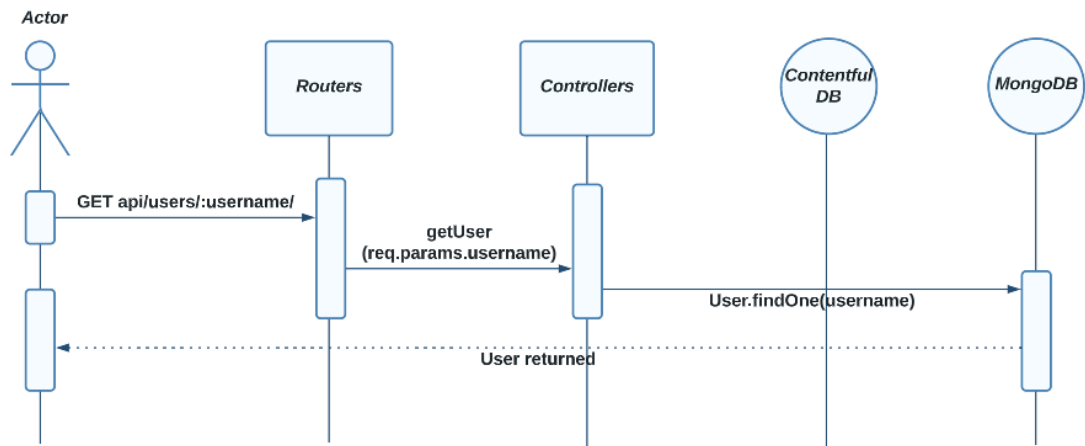


Figure 37. Sequence diagram for getting a user

```

/** routes/users.ts */

usersRouter.get(
  "/:username",
  async (req: express.Request, res: express.Response) => {
    const users = await getUser(req.params.username);
    res.json(users);
  }
);

/** controllers/users.ts */

const getUser = async (username: string) => {
  const user = await User.findOne({ username });
  return user;
};
  
```

Code Snippet 13. Backend code for getting a user

Filmmash supports user authentication and authorization, which means routes for logging in and signing up are fully provided. Signing up process requires the user's input data, whose password is subsequently hashed using the `bcrypt` library. It is based on the homonymous hashing function introduced by Provos and Mazières (1999), as an optimization of `crypt` hashing technique which was deemed not adaptable to the fast-paced evolution of computer hardware. It was thought to be unable to withstand a dictionary attack, in which hackers decipher a hashed key by brute-forcing through thousands or millions of possibilities. Bcrypt evolves on this technique by implementing the Blowfish cipher. It requires strenuous preprocessing for each time a key is changed, therefore increasing the workload and duration of hash calculations, which in turn be an impossible task

for brute force attacks (Arias, 2021a). Moreover, `bcrypt` also uses salting technique, where salt is a randomly generated value that is used in combination with a password or other data input to create a unique hash. The purpose of a salt is to add additional entropy to the input, making it more difficult for attackers to guess the original input or to use precomputed hash tables for attacks (Arias, 2021b). Based on this, the Filmmash's sign-up operation also provided a random number of salt rounds to strengthen the security of the hashed password.

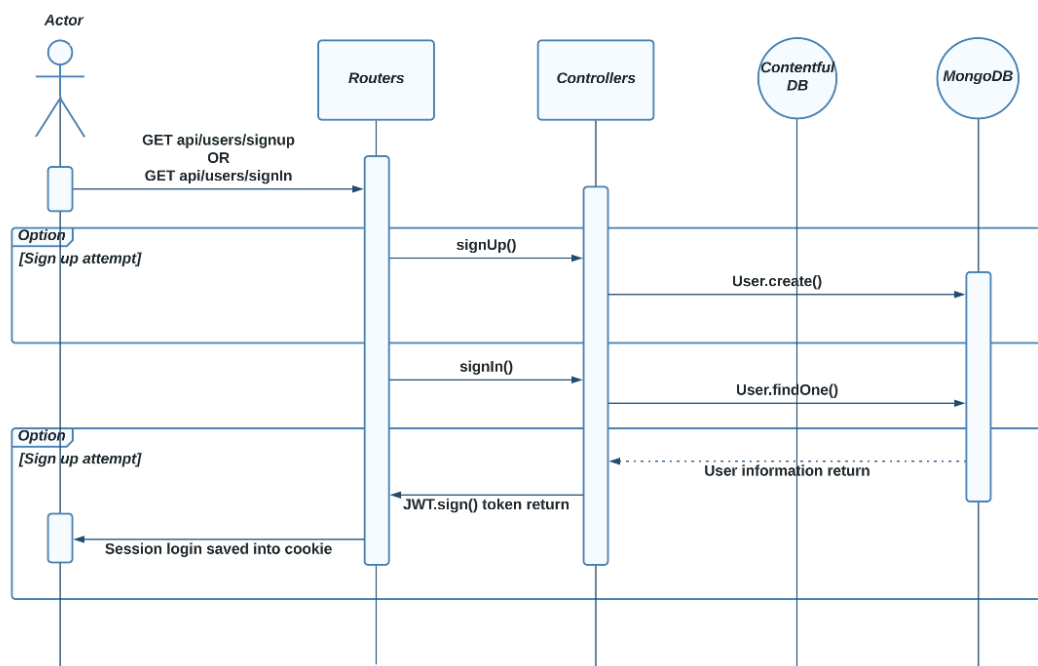


Figure 38. Sequence diagram for logging in and signing up

```

/** routes/users.ts */

usersRouter.post(
  "/signup",
  async (req: express.Request, res: express.Response) => {
    const { name, username, password, avatarImg } = req.body;
    const _createdUser = await signUp({ name, username, password, avatarImg });
    const [token, _user] = await signIn({ username, password });
    res.status(200).send({ token, username, name, avatarImg });
  }
);

/** controllers/users.ts */

const signUp = async ({
  name,
  username,
  password,
  avatarImg,
}: {
  name: string;
  username: string;
  password: string;
  avatarImg?: string;
}) => {
  const saltRounds = 10;
  const passwordHash = await bcrypt.hash(password, saltRounds);

  const user = await User.create({
    username,
    name,
    passwordHash,
    avatarImg: avatarImg || "",
  });

  return user;
};

```

Code Snippet 14. Backend code for sign up operation

The sign-up operation links directly to a log in. A log-in attempt finds the account linked to the unique input username in MongoDB collections. The data found is subsequently used to verify the password input by the user, and if a match is recorded, a session token is then created using the JWT library, whose `sign` method receives information about user data, the token expiry date (after which time user must redo the authentication process), and the secret key provided in the `.env` file.

```

/** routes/users.ts */

usersRouter.post(
  "/signin",
  async (req: express.Request, res: express.Response) => {
    const { username, password } = req.body;
    const [token, user] = await signIn({ username, password });
    if (!token) {
      return res.status(401).json({
        error: "invalid username or password",
      });
    }
    res.status(200).send({
      token,
      username: user.username,
      name: user.username,
      avatarImg: user.avatarImg,
    });
  }
);
/** controllers/users.ts */

const signIn = async ({ username, password }: {
  username: string;
  password: string;
}) => {
  const user = await User.findOne({ username });
  const passwordCorrect = !user
    ? false
    : await bcrypt.compare(password, user.passwordHash);

  if (!(user && passwordCorrect)) {
    return null;
  }

  const userForToken = {
    username: user.username,
    id: user._id,
  };

  const token = jwt.sign(userForToken, process.env.SECRET, {
    expiresIn: 60 * 60 * 24 * 7,
  });

  return [token, user];
};

```

Code Snippet 15. Backend code for login operation

4.2.6. Utilities

The `utils` repository contains reusable segments of code required for the functionalities of routers and controllers and is divided into four categories based on their respective actions. The two most important files are `token.ts` and `contentful.ts`

The `token.ts` file verifies the authorization of a user by retrieving a decoded token. This is done using the same JWT library previously mentioned, which also provides a `verify` method, which accepts two parameters, the token saved in the request's header, and the secret key saved in the `.env` file. If the decoded

token matches a user in a database, that user is therefore returned, otherwise it signals an error.

```
/** utils/token.ts */  
  
import jwt from "jsonwebtoken";  
import { User } from "../models";  
  
export const getAuthorizedUser = async (request: any, response: any) => {  
  const authorization = request.get("authorization");  
  const token =  
    authorization && authorization.toLowerCase().startsWith("bearer ")  
      ? authorization.substring(7)  
      : "";  
  
  const decodedToken = jwt.verify(token, process.env.SECRET);  
  
  if (!token || !decodedToken.id) {  
    return response.status(401).json({ error: "token missing or invalid" });  
  }  
  
  const user = await User.findById(decodedToken.id);  
  return user;  
};
```

Code Snippet 16. Backend code for getting authorized user

Since Contentful is being used as the bridge between controllers and MongoDB data objects, Filmmash utilizes its library to retrieve the application's metadata stored. While the backend provides three functions for different data collecting methods, the code backbone is to use the `createClient()` function by Contentful, with all tokens and credentials be adequately input. The response's data is cleaned accordingly based on the purpose of the retrieval.

```

/** utils/contentful.ts */
export const getAllEntriesFromContentful = async (
  spaceId: string,
  accessToken: string,
  contentType?: string,
  environmentId?: string
) => {
  const client = createClient({
    space: spaceId,
    environment: environmentId ? environmentId : "master",
    accessToken: accessToken,
  });

  const response = contentType
    ? await client.getEntries({
        content_type: contentType,
      })
    : await client.getEntries();

  //response data is cleaned
  //return cleanResponseData
};

```

Code Snippet 17. Backend code for retrieving data from Contentful

4.2.7. Backend Conclusion

All the code discussed in the previous subsections are combined in the two foundation files: `app.ts` and `index.ts`. This subsection elaborates on the functionalities of these two files and concludes Filmmash backend implementation. `app.ts` imports all the necessary libraries as well as the written code to maintain a well-functioned Express application. It initializes an Express app as well as connects to MongoDB using Mongoose, and then initializes the middleware functions using the `use()` method on the app instance. The `cors()` middleware is used to allow cross-origin resource sharing with credentials, while `cookieParser()` middleware is used to parse cookies sent from the client. The `express.json()` middleware is used to parse JSON requests from the client, while the `requestLogger`, as discussed, logs incoming requests. Lastly and most importantly, the Express application connects to the three defined routers: users, galleries and films, enabling handling of HTTP requests to the specific endpoints of these routers.


```

/** app.ts */

import express from "express";
import mongoose from "mongoose";
import "express-async-errors";
import cors from "cors";
import cookieParser from "cookie-parser";

//middlewares
import { MONGODB_URI } from "../utils/config";
import {
  requestLogger,
  unknownEndpoint,
  errorHandler,
} from "../utils/middleware";

//routers
import { usersRouter, galleriesRouter, filmsRouter } from "../routes";

```

```

/** app.ts */

const app = express();

mongoose
  .connect(MONGODB_URI)
  .then(() => {
    console.log("connected to MongoDB");
  })
  .catch((er) => {
    console.error("error connecting to MongoDB", er.message);
  });

app.use(cors({ credentials: true }));
app.use(cookieParser());
app.use(express.json());
app.use(requestLogger);

```

```

/** app.ts */

app.use("/api/users", usersRouter);
app.use("/api/galleries", galleriesRouter);
app.use("/api/films", filmsRouter);

app.use(unknownEndpoint);
app.use(errorHandler);


export default app;

```

Code Snippet 18. Backend Express application initialized

At this stage, the application backend is now fully formed and can be connected to the server for development and production. This step is finalized in the

`index.ts` file, which is the entry point of Filmmash backend application.



```
/** index.ts */  
  
import http from 'http';  
import app from './app';  
import { PORT } from './utils/config';  
  
// connect to server  
const server = http.createServer(app);  
  
const port = PORT || 8080  
  
server.listen(port, () => {  
  console.log(`Server running on port ${port}`);  
});
```

Code Snippet 19. Backend `index.ts` file

4.3. Frontend Implementation

Filmmash frontend utilizes the Qwik framework for development. This subsection explores the structure and content of the code to further prove the practicability of Qwik in a real-life project.

4.3.1. Frontend Code Structure

The following figure describes the structure of Filmmash frontend code:

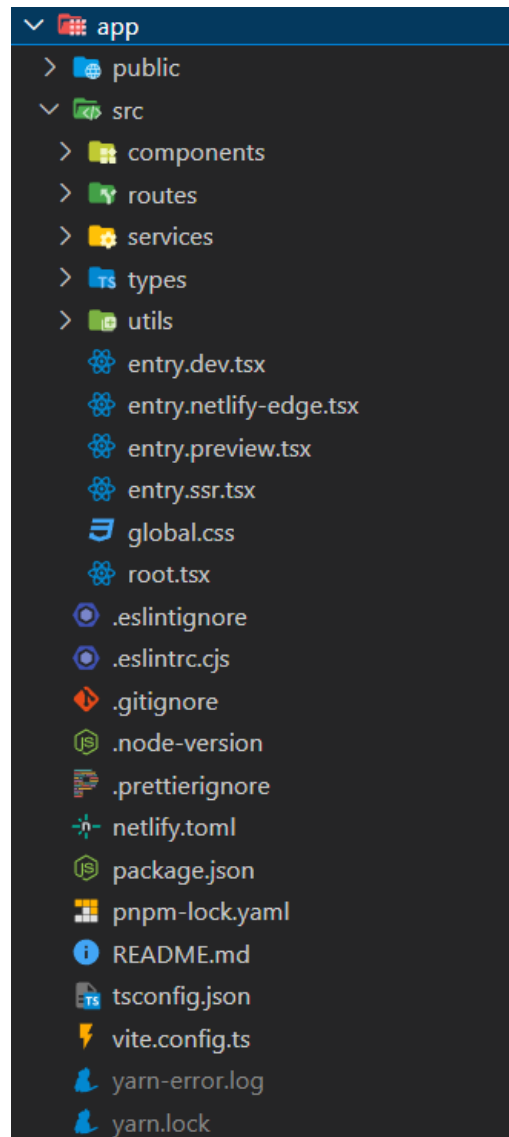


Figure 39. Frontend code structure

The structure of the code is based on Qwik's convention when initializing a project using Qwik CLI. It mirrors the structure of the backend with clear division between the application part (which is located under the `src` directory) and the configuration part. A noting point in the configuration part is the appearance of `vite.config.ts` file, which indicates the use of Vite, a developing environment that Qwik uses to build its application, which is a relatively new tool yet it has gained its stature in the developers' community thanks to its fast performance as well as its use of hot module replacement.

This section focuses primarily on the application part by exploring the files as well as the crucial directories, especially `routes` and `components`, to vividly portray the core functionality of a Qwik-based frontend application. Since the codebase is large, the section aims to explain in detail the most crucial factors while providing examples for the remaining.

4.3.2. Root, Global Styling, and Entry Files

The files described in this section are the entry points for Filmmash, which are generated on creating a Qwik application. The `root.tsx` file contains the root of the application tree and is wrapped in a `QwikCityProvider` component, which works as a meta-framework that provides routing capabilities for a Qwik application. The root component is practically a HTML document structure with the head includes the necessary `meta` tags for enhancing web details and performance.

```
/** src/root.tsx */

import { component$ } from "@builder.io/qwik";
import { QwikCityProvider, RouterOutlet, ServiceWorkerRegister } from '@builder.io/qwik-city';

import "./global.css";

export default component$(() => {
  /**
   * The root of a QwikCity site always start with the <QwikCity> component,
   * immediately followed by the document's <head> and <body>.
   *
   * Dont remove the `<head>` and `<body>` elements.
   */
  return (
    <QwikCityProvider>
      <head>
        <meta charset="utf-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
        <meta name="description" content="Put your description here."/>
        <title>FILMMASH</title>
      </head>
      <body lang="en">
        <RouterOutlet />
        <ServiceWorkerRegister />
      </body>
    </QwikCityProvider>
  );
});
```

Code Snippet 20. Frontend `root.tsx` file

For this specific reason, Filmmash's global styling `global.css` is also imported into this file. Unlike component-specific styles, which are defined within the

scope of a specific component and only apply to that component and its children, global styles apply to all components and elements on the page, which makes them useful for defining styles that need to be consistent across the entire application, such as basic typography or layout styles. It should be noted that `global.css` is the only native stylesheet file used in Filmmash, since the application uses a technique called CSS-in-JS which is later discussed. It is derived from the structure that Filmmash contains four entry files, each supporting HTML rendering in a specific environment. The two most crucial files are `entry.ssr.tsx` and `entry.preview.tsx`, with the former sets up SSR and renders the initial HTML for the page and the latter serves the application in production mode.

4.3.3. Routes

As mentioned, Qwik is supported by Qwik City, which provides a directory-based routing system for Qwik applications. This routing system is implemented through the routes folder, which contains an index.tsx file in the root directory as well as in each subdirectory. Essentially, the index.tsx file acts as the entry point for each directory in the routing system. When a user visits a particular route in the application, the index.tsx file associated with that route is responsible for handling the request and rendering the appropriate components.

```
/** src/routes/index.tsx */  
  
import { component$ } from "@builder.io/qwik";  
import type { DocumentHead } from "@builder.io/qwik-city";  
import Home from "~/components/homepage";  
  
export default component$(() => {  
  return (  
    <>  
      <Home />  
    </>  
  );  
});  
  
export const head: DocumentHead = {  
  title: "Welcome to Qwik",  
};
```

Code Snippet 21. The root component of the Frontend router

For example, the file ``src/routes/login/index.tsx`` which contains the Login component will be accordingly mapped to its correct URL path `"https://example.com/login"`. The directory-based routing system in Qwik City is significant since it offers a systematic and orderly approach to handling application routes, providing a high degree of flexibility in defining routes and handling requests, as well as facilitating the creation of intricate and versatile web applications, which is comparable to its counterparts such as Next.js or Remix.run. Another feature Qwik City supports is the use of dynamic route segments, which enables developers to create more customized and personalized user experiences. In Filmmash, parentheses and brackets are used to define the dynamic routes, as indicated in the ``galleries`` and ``users`` routes. While parentheses are used in cases where the routes' names should be omitted from the URL, the brackets indicate the path can be dynamically changed based on IDs or inputs.

Apart from the main structure, layout is also a vital concept in Qwik City's routing. Layout is typically used to define a reusable component for multiple pages in the application, such as a header, footer, or navigation menu. Filmmash uses the ``layout.tsx`` file at the root of the ``routes`` directory, which reuses the Header component and renders the nested routes under the Slot component.

```
/** src/routes/layout.tsx */  
  
import { component$, Slot } from "@builder.io/qwik";  
import Header from "~/components/header";  
  
export default component$(() => {  
  return (  
    <>  
      <Header />  
      <main>  
        <section>  
          <Slot />  
        </section>  
      </main>  
      <footer></footer>  
    </>  
  );  
});
```

Code Snippet 22. Frontend's routing layout

4.3.4. Initialized Components: Homepage and Header

The subsection onward discusses the implementation of core components based on the user flow described in Figure 21 and Figure 22, including Homepage, Header, Forms and Gallery components.

Homepage and Header are the first two rendered components when a client requests Filmmash's address, as displayed in the following figure.

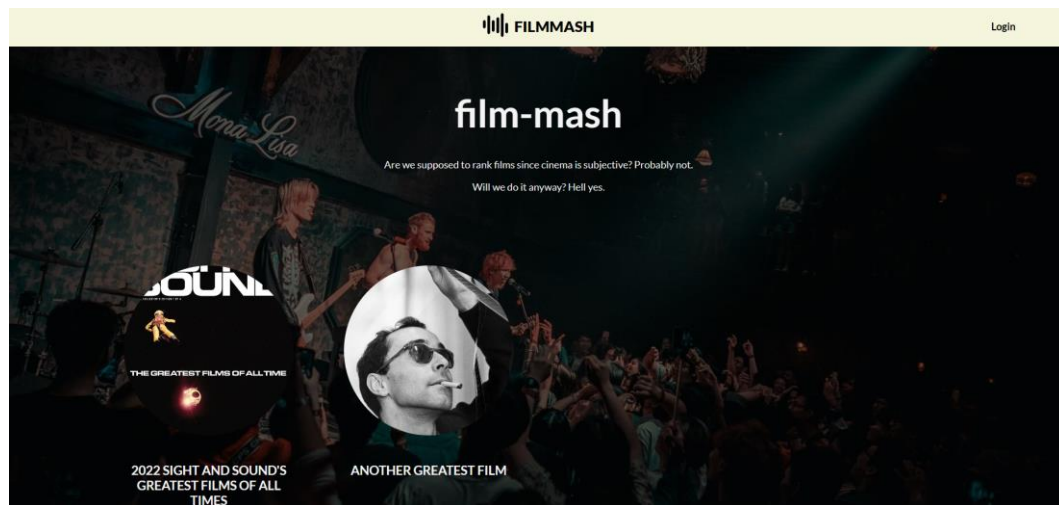


Figure 40. Homepage and Header components

The Homepage component returns a basic title and subtitle, as well as renders a list of available galleries. This can be achieved the Qwik's `useStore` hook, which initialize a state that holds the values of galleries, as well as the `useTask\$` hook to retrieve the list of all galleries.

```

/** src/components/homepage/index.tsx */
const Home = component$(() => {
  const store: GalleriesGridStore = useStore(
    {
      galleries: [],
    },
    { deep: true }
  );

  useTask$(async () => {
    store.galleries = await getAllGalleries();
  });

  return (
    <StyledHome>
      <Banner />
      <HomeGalleries galleries={store.galleries} />
    </StyledHome>
  );
});
export default Home;

```

Code Snippet 23. Homepage component code

The child component HomeGalleries, as described, receives the gallery's data retrieved by the `getAllGalleries()` method defined in the `services` folder, before being mapped accordingly for the rendering mechanism.

```

/** src/components/homepage/HomeGalleries.tsx */
const HomeGalleries = component$(({ galleries }: HomeGalleriesProps) => {
  return (
    <StyledGalleries>
      {galleries.map((gallery) => (
        <HomeGallery key={gallery._id} gallery={gallery} />
      ))}
    </StyledGalleries>
  );
});

```

Code Snippet 24. HomeGalleries component code

The Header component is relatively simple with the name of the web application displayed in the middle and a login link located in the right corner. It initially checks if the session is authenticated and accesses the login user information which is stored in the local storage of the browser upon signing in, replacing the "login" link with a profile image of the authenticated user.


```

/** src/components/header/index.tsx */

const Header = component$(() => {
  const store: any = useStore(
    {
      user: {},
    },
    { deep: true }
  );

  useVisibleTask$(() => {
    const loggedInUserJSON = window.localStorage.getItem("loggedInUser");
    if (loggedInUserJSON) {
      const user = JSON.parse(loggedInUserJSON);
      store.user = user;
    }
  });

  return (
    <StyledHeader>
      <div></div>
      <HeaderLink href="/">
        <HeaderIcon />
        <HeaderName> FILMMASH </HeaderName>
      </HeaderLink>
      {!store.user.username ? (
        <HeaderLogin href="/login">Login</HeaderLogin>
      ) : (
        <HeaderAvatar href={` /user/${store.user.username}`}>
          <AvatarLog src={` ${store.user.avatarImg}`} />
        </HeaderAvatar>
      )}
    </StyledHeader>
  );
});

```

Code Snippet 25. Header component code

4.3.5. Form Components: Login, Sign Up and Create Gallery

By the necessity of their features, there are three components required the use of form components: Login, Sign up and Gallery creation, the first two are in the `auth` subdirectory when the latter belongs to the `user` counterpart. Since the backbone of these components resemble each other, this subsection mainly discusses functionalities which compose each of them, with code snippets and results provided. In each component, a three-part code structure is presented which consists of state management, submit handler and form rendering. The use of `useNavigate()` hook is notable as it creates a redirection to the desired

location

post-submission.

```
/** src/components/auth/login.tsx */

const Login = component$(() => {
  const store: { username: string; password: string } = useStore({
    username: "",
    password: "",
  });
  const nav = useNavigate()

  const handleChange = $(e: any) => {
    const cloneStore = {
      ...store,
      [e.target.name]: e.target.value,
    };

    store.username = cloneStore.username
    store.password = cloneStore.password
  });

  const handleSubmit = $(async () => {
    const {username, password} = store
    const user = await signIn(username, password)
    window.localStorage.setItem(
      'loggedUser', JSON.stringify(user)
    )
    nav('/')
  })

  //render form...
})
```

Code Snippet 26. Login component code

The rendered components are displayed as followed:

LOGIN

Does not have an account yet? Sign up [here](#)

Username

Password

SIGN UP

Have an account? Log in [here](#)

Name

Username

Password

Avatar URL

Sign up

CREATE GALLERY

Space ID

Access Token

Environment ID

Gallery ID

Films ID

Create Gallery

Figure 41. Login, Sign up and Create gallery form

4.3.6. Gallery Components: Overview, Mash and Ranking

The Gallery components serve the main functionalities of the application, including letting users compare films one-to-one and ranking them accordingly. This subsection follows the user flow by first exploring the gallery overview, then explains in detail the mashing functions as well as the ranking system. The Gallery component includes a root `index.tsx` file that handles and manages gallery data retrieval using the `galleryService` in `services` directory, while also providing a header bar at the top of its child components for navigation purposes.

```
/** src/components/gallery/index.tsx */  
  
const GalleryPage = component$(() => {  
  const location = useLocation();  
  
  const store: GalleryPageStore = useStore(  
    { gallery: { _id: "", user: "" }, toggle: "overview" },  
    { deep: true }  
  );  
  
  useTask$(async () => {  
    store.gallery = await getOneGallery(location.params.gallery);  
  });  
  
  //...
```

Code Snippet 27. Gallery component's root file

The Overview child component involves only retrieving and displaying a gallery metadata stored in Contentful, which is straightforward given the data passed down as props from the Gallery's root.

```

/** src/components/gallery/overview/index.tsx */

import { component$ } from "@builder.io/qwik";
import { marked } from "marked";
import sanitizeHtml from "sanitize-html";
import {
  StyledOverview,
  GalleryAvatar,
  AvatarImage,
  GalleryDescription,
  GalleryName,
  GallerySummary,
} from "~/components/styled/overview.css";

const GalleryOverview = component$(({ gallery }: { gallery: Gallery }) => {
  return (
    <StyledOverview>
      <GalleryAvatar>
        <AvatarImage src={gallery.avatarImg} alt="" loading="lazy" />
      </GalleryAvatar>
      <GalleryName>{gallery.name?.toUpperCase()}</GalleryName>
      <GallerySummary
        dangerouslySetInnerHTML={`${sanitizeHtml(
          marked.parse(gallery.summary || "")
        )}`}
      />
      <GalleryDescription
        dangerouslySetInnerHTML={`${sanitizeHtml(
          marked.parse(gallery.description || "")
        )}`}
      />
    </>
  </StyledOverview>
  );
});

export default GalleryOverview;

```

Code Snippet 28. Gallery's overview code

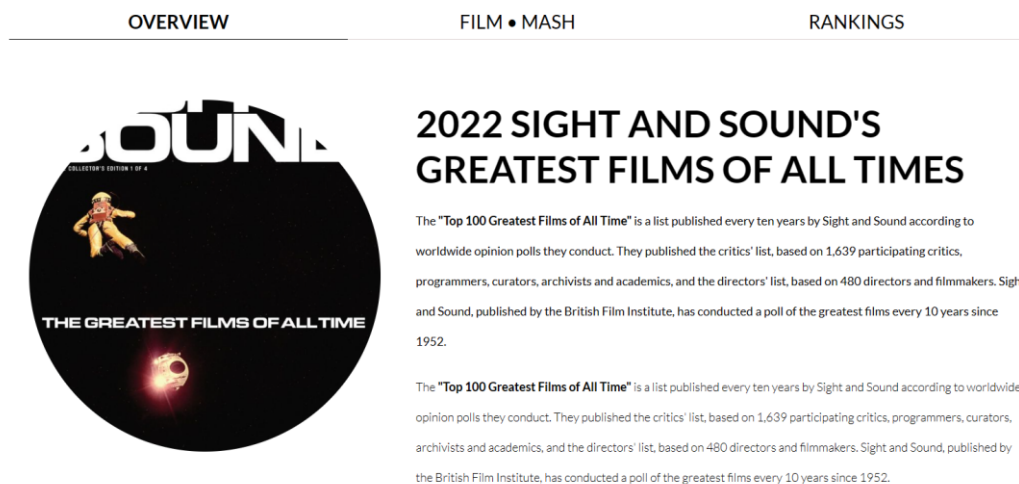


Figure 42. Overview component

The implementation of the mashing function for Filmmash is quite complex. The functionality requires a state which includes the two indexes that point to two

films in an array respectively, as well as the films' previous points.

```
/** src/components/gallery/mash/index.tsx */  
  
const store: FilmmashStore = createStore(  
  {  
    indexLeft: 0,  
    indexRight: 0,  
    filmLeft: { points: 0, _id: "" },  
    filmRight: { points: 0, _id: "" },  
    galleryFilms: [],  
    isLoading: true,  
  },  
  { deep: true }  
);
```

Code Snippet 29. Gallery's mashing state management

When users first access the page, the `useResource\$` hook provided by Qwik retrieves an array of all films in the specific gallery, before randomizing two integers that fall within the range of the array's length. With each round of randomization, Qwik's `track` attribute detects a behavioral change and updates the according states of the two films, hence resulting in a display of two films each in different rounds.

```
/** src/components/gallery/mash/index.tsx */  
  
useResource$(async () => {  
  store.galleryFilms = await getAllFilmsInOneGallery(gallery._id);  
  [store.indexLeft, store.indexRight] = randomizeFilm(  
    store.galleryFilms.length  
  );  
  store.isLoading = false;  
});  
  
useResource$(async ({ track }) => {  
  const indexLeft = track(() => store.indexLeft);  
  const indexRight = track(() => store.indexRight);  
  store.filmLeft = store.galleryFilms[indexLeft];  
  store.filmRight = store.galleryFilms[indexRight];  
});
```

Code Snippet 30. Gallery's mashing `useResource` hooks

The next step is to update the points of each film based on the user's decision, with a `calcEloRating()` function written to represent the Elo's Algorithm. The points returned from the function are used to update the films' points accordingly.

```

/** src/utils/elo-algorithm.ts */

export const calcEloRating = (
  ratingLeft: number,
  ratingRight: number,
  winner: string
) => {
  const kFactorLeft = ratingLeft > 2400 ? 16 : ratingLeft < 2100 ? 32 : 24;
  const kFactorRight = ratingRight > 2400 ? 16 : ratingRight < 2100 ? 32 : 24;

  const expLeft = 1 / (1 + 10 ** ((ratingRight - ratingLeft) / 400));
  const expRight = 1 / (1 + 10 ** ((ratingLeft - ratingRight) / 400));

  const newRatingLeft =
    winner === "left"
      ? ratingLeft + kFactorLeft * (1 - expLeft)
      : ratingLeft + kFactorLeft * (0 - expLeft);
  const newRatingRight =
    winner === "right"
      ? ratingRight + kFactorRight * (1 - expRight)
      : ratingRight + kFactorRight * (0 - expRight);

  return [Math.round(newRatingLeft), Math.round(newRatingRight)];
};

```

Code Snippet 31. Elo's algorithm in code

The result of the mashing functionality is displayed as followed:

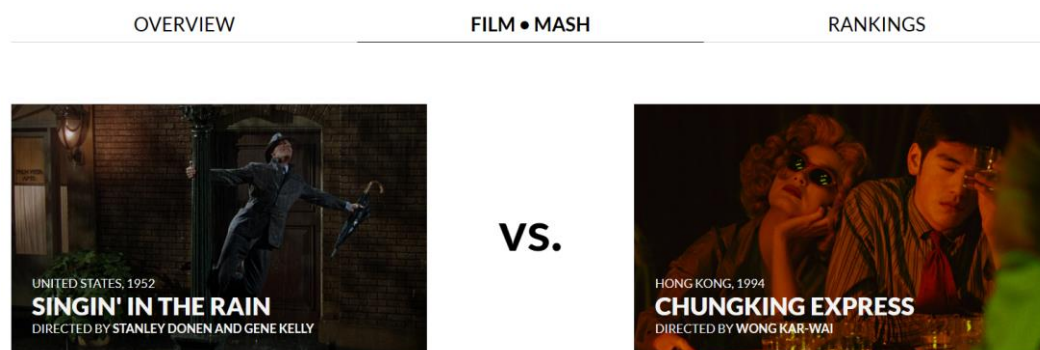



Figure 43. Mashing functionality

Lastly, the development of the ranking system is also fulfilled with sorting feature, and each row can be expanded if users choose to view the film's full metadata as input in Contentful.

#	Film	Director(s)	Year	Country	Points ▼
1	Breathless	Jean-Luc Godard	1960	France	1421



BREATHLESS

Directed by **Jean-Luc Godard**
France, 1960

A small-time thief steals a car and impulsively murders a motorcycle policeman. Wanted by the authorities, he attempts to persuade a girl to run away to Italy with him.

2	Chungking Express	Wong Kar-wai	1994	Hong Kong	1408
3	Battleship Potemkin	Sergei Eisenstein	1925	Soviet Union	1348
4	The Third Man	Carol Reed	1949	United Kingdom	1321
5	Vertigo	Alfred Hitchcock	1958	United States	1298
6	Singin' in the Rain	Stanley Doren and Gene Kelly	1952	United States	1295
7	Seven Samurai	Akira Kurosawa	1954	Japan	1293

Figure 44. Ranking component

4.3.7. Styled Components

As mentioned, Filmmash uses a technique called CSS-in-JS, which means styling components directly in a JavaScript/TypeScript file. This approach has several benefits, the first of which is that it allows for more efficient rendering of CSS styles, since it enables dynamic generation of CSS styles at runtime. Styled components also make it easier to create reusable components by defining styles and functionalities in a single place and recalling them across the application. All in all, CSS-in-JS enables more flexible and dynamic styling options, such as conditionally applying styles based on user interactions or data input, which results in a more interactive and responsive user experience. Since it aligns with Filmmash's intention, CSS-in-JS is achieved by using the Styled Vanilla library, which Qwik internally supports. The components are bundled with their respective styling under the `components/styled` subdirectory.

5. PERFORMANCE EVALUATION

The primary focus of this chapter is to assess how well the Filmmash application performs, specifically by analyzing the adaptability of its utilized technologies, Headless CMS and Qwik framework. The chapter aims to examine the practical usage of these technologies in Filmmash and to determine whether they perform as well as claimed in the theoretical part of the work, therefore providing evidence to support the claims made about these technologies and their effectiveness in this specific application. By examining the practical usage of these technologies, the chapter aims to provide a more complete understanding of how well they work in real-world situations.

5.1. Contentful-based Management System

As stated in the theoretical part, a crucial benefit of a Headless system is its ability to provide content across devices with utmost responsiveness and flexibility, as it detaches the content with its design and transfers information using API. In the case of Filmmash, which uses Contentful as the Headless-based content management, the content is rightfully delivered through Contentful's Content Delivery API to then coupled with the styling defined separately in the client side. This results in a responsive and scalable web application, where users' contents are distributed with respect to the omnichannel approach. The following figures describe an example of responsiveness of Filmmash, regardless of the content populated from Contentful.

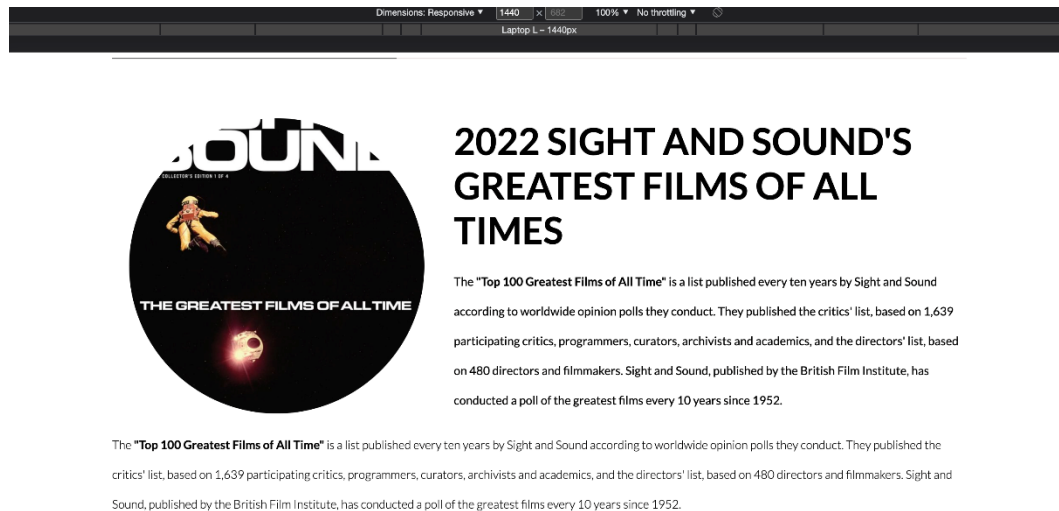


Figure 45. FilmMash's overview section, viewed on laptop screen (1440px in width)

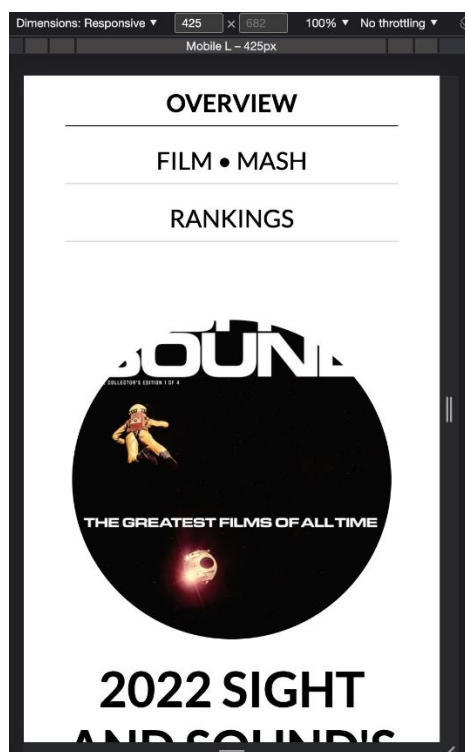


Figure 46. FilmMash's overview section, viewed on laptop screen (425px in width)

One problem raised in the theoretical section, unstructured content, is also addressed with the use of Contentful. While unstructured content prevents the

classification of individual components, Contentful separates its content into distinct content models with labeled fields of specific types. Users therefore can freely modify the one component's content or length without affecting the view layer of others, thus turning Filmmash into a composable content platform.

As mentioned, the Filmmash application presents a challenge of complexity and a steep learning curve. To create a new gallery and add content, users are required to have a fundamental understanding of the Headless CMS, as they need to create distinct data models in Contentful. However, this process is not simple and straightforward. One possible solution is to implement a customized form in Filmmash that mimics the behavior of Contentful and utilizes its Content Management API to input users' answers into Contentful's system. Such tasks are deemed as not applicable given the short time and small application's scope, as discussed in the requirement analysis section.

5.2. Google Lighthouse Performance Metrics

The theoretical section of the thesis makes claims about the innovative features of the Qwik framework. To support these claims, the performance of the Filmmash application, which uses Qwik for its frontend, is evaluated to gather empirical evidence and to demonstrate the effectiveness of the Qwik framework as a modern, innovative technology for developing frontend applications. This analysis is likely to involve testing and benchmarking Filmmash against other applications built using different frontend technologies to draw comparisons and conclusions about Qwik's performance. Google Lighthouse is the measurement tool used in this part, with the scores calculated as indicated in the following figure.

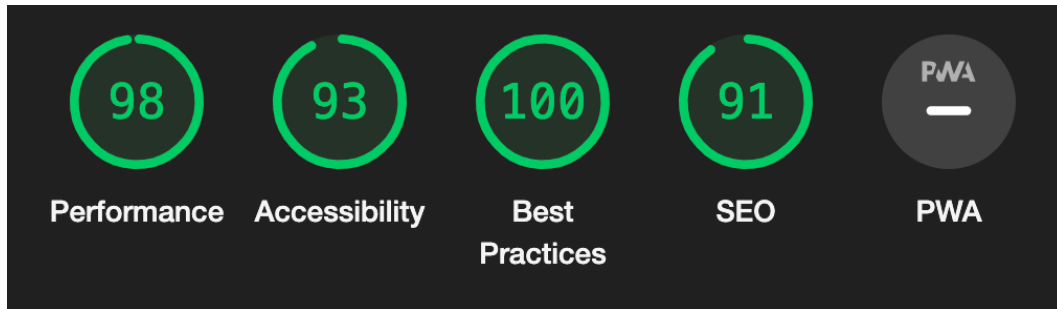


Figure 47. Google Lighthouse's measurement of Filmmash

It is noticeable that the four measured metrics of Filmmash exceed the 90-point threshold, which can be deemed as “good” in Google Lighthouse’s term. This score can be further analysed by looking into the Performance metrics, in which the total blocking time, or the sum of all time periods between the first contentful paint and the TTI, is an incredible 0 milliseconds, meaning that once loaded, the whole content of Filmmash displays in the viewport immediately without any delay for loading script.

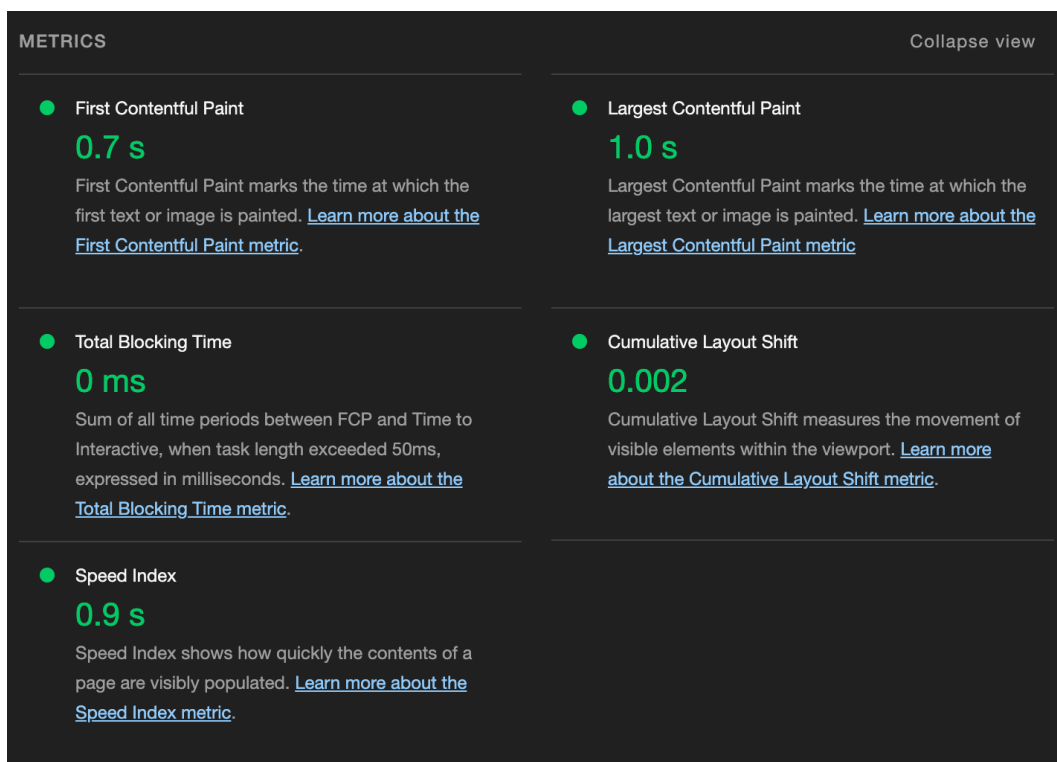


Figure 48. Filmmash's performance metric scores

This is also demonstrated in the “Passed audits” section, where Lighthouse acknowledges Filmmash for having minified the amount of JavaScript used.

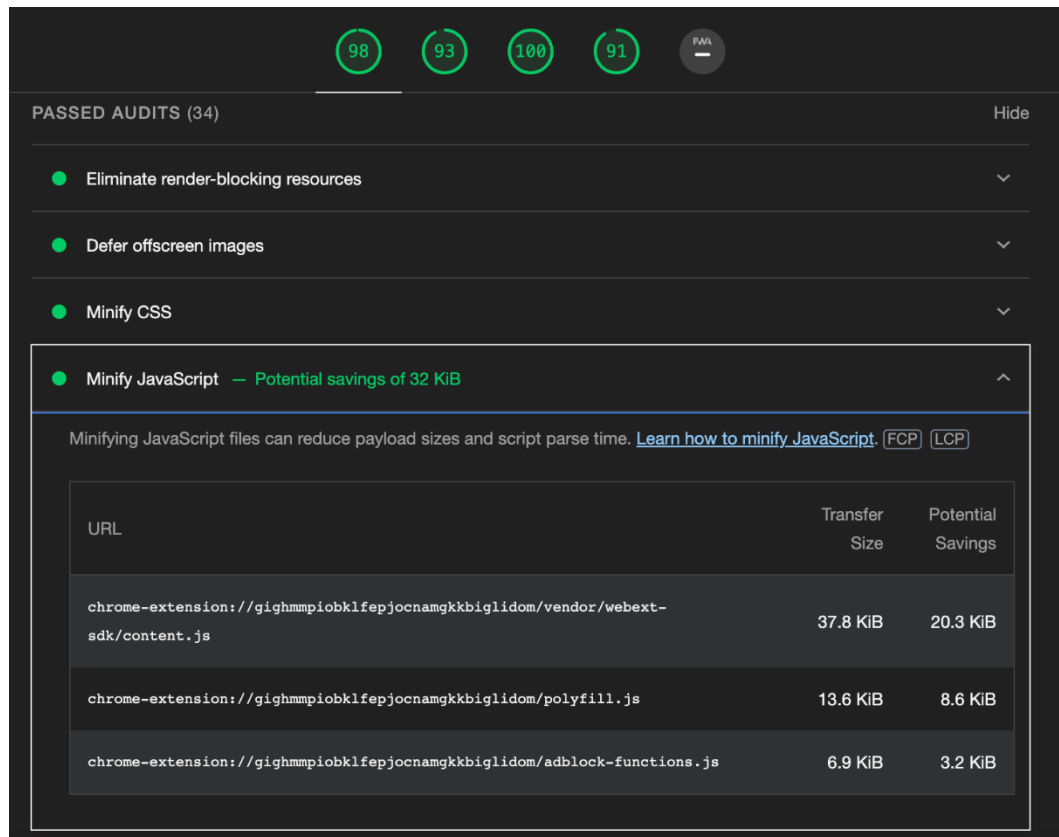


Figure 49. Filmmash's passed audits according to Google Lighthouse

The “Network” tab shows a deeper view of Filmmash’s initialization process, with the main document tree being loaded instantly, followed by images and stylesheet files, whereas the scripts are only loaded afterwards based on users’ behavior. This evidence proves the main innovation of Qwik in theory, which loads JavaScript if and only if users begin to interact with the application.

To further assess the practicality of Qwik, this part compares its performance with two web applications written in state-of-the-art metaframeworks, Next.js (meta-framework of React) and Nuxt (of Vue).

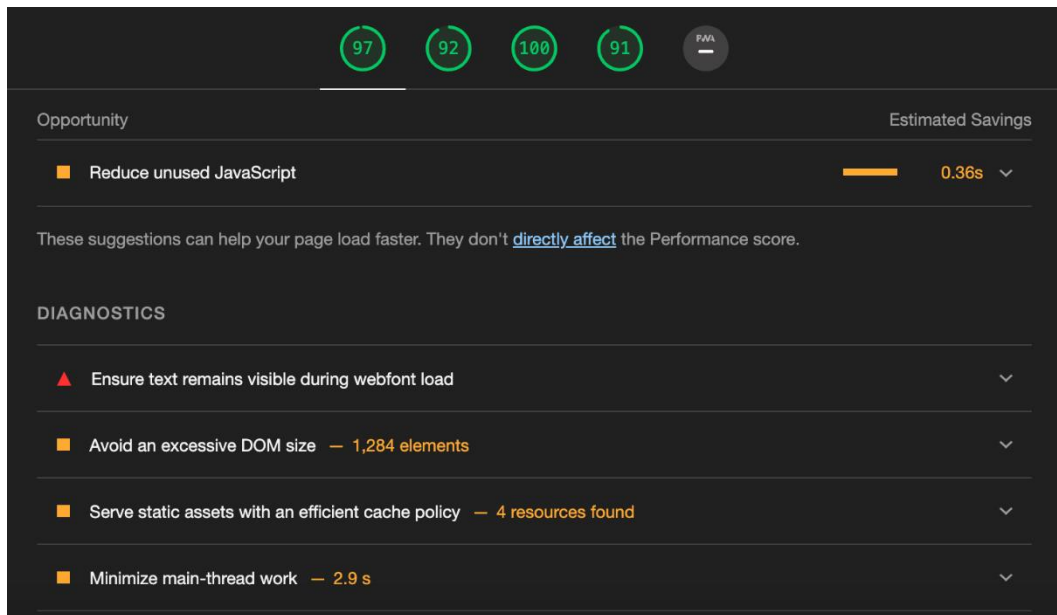


Figure 50. Google Lighthouse's measurement of Next.js documentation

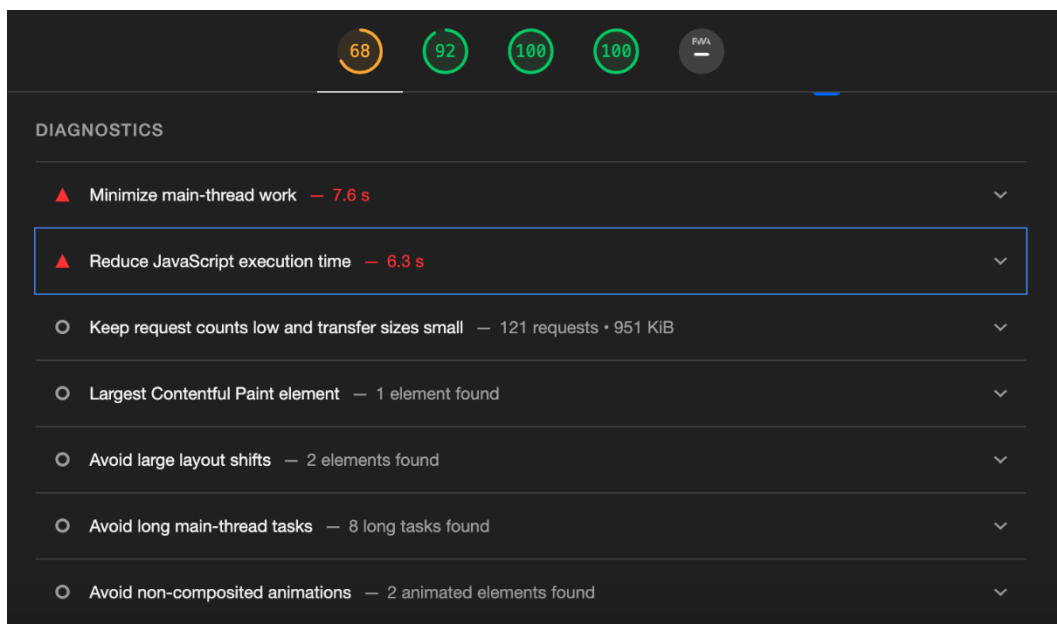


Figure 51. Google Lighthouse's measurement of Nuxt documentation

As it is suggested by the Google Lighthouse results, the two frameworks, which undoubtedly have its own advantages, do not resolve the JavaScript premature load issues, which affects the TTI and the initializing performance. The previously claimed features of Qwik are therefore proven.

6. CONCLUSIONS

6.1. Thesis Summary

The primary focus of this thesis is to investigate the capabilities of Headless CMS and Qwik framework in addressing the current issues with web frameworks and content management systems. To achieve this, several research questions have been formulated, including identifying the problems associated with current content management systems and web frameworks and analyzing how a shift towards Headless CMS and Qwik helps address these issues. The thesis studies real-world cases of Headless CMS and Qwik and identifies the potential benefits and limitations of the technologies, as well as understanding how Headless CMS and Qwik can be applied to a full-stack application. Throughout the research, a detailed exploration of the functionalities of Headless CMS and Qwik was conducted by means of reviewing existing research materials, analyzing real-world cases of Contentful and Builder.io to provide a practical understanding of the technologies' applications in various industries.

The thesis also demonstrates the implementation of Filmmash, an application utilizing the researched technology. It places emphasis on the essential stages of web application development, covering everything from the initial architectural design performance testing. A key focus of this section is on the integration of Headless CMS and Qwik, with the aim of evaluating the practicality and effectiveness of these technologies in the context of web application development.

In summary, this thesis contributes to the body of knowledge surrounding Headless CMS and Qwik, providing valuable insights into the capabilities of these technologies. By addressing the limitations and exploring the practical applications of these technologies, this research offers a comprehensive analysis of their potential impact on web application development.

6.2. Key Findings

The thesis examines the history of content management systems and determines that the introduction of omnichannel applications marks the turning point for the CMS industry, with Headless CMS emerging as a popular alternative to traditional CMS solutions. The primary advantage of Headless CMS is its ability to separate the content management functionality from the presentation layer of a website or application. This decoupling of the frontend and backend allows developers to build more flexible and scalable applications, as the content can be easily consumed by multiple channels such as mobile apps, websites, or IoT devices, regardless of their styling, formatting, or presentation. Additionally, Headless CMS solutions are often API-driven and cloud-based, which means that developers can access content programmatically, enabling more customization and automation of the content creation and delivery process in comparison to the traditional bundled and in-house content.

Despite the potential benefits of Headless CMS, there are several challenges and limitations associated with this relatively new approach. One of the main issues is its complexity, which demands expertise in both frontend and backend development, a steep learning curve that may pose a barrier to content builders who lack technical expertise. While the decoupling of content and presentation can provide more freedom for developers in choosing technology stacks, the lack of built-in functionalities that are typically provided by monolithic CMS solutions can make it more difficult to create web applications using Headless CMS. Headless CMS is also not regarded as the ultimate solution to the content management industry, as it has not fully resolved the problem of unstructured content. It can be concluded that the aim of contemporary application developers is to develop a content management system that merges the flexibility of Headless CMS with an organized and structured content model, with Contentful composable content platform being a study case. The Filmmash

application, which utilizing Contentful as the main content management provider, further proves its effectiveness in a real-world application.

The thesis also explores the evolution of web frameworks from its early days to the introduction of metaframeworks and figures out the problems of CSR and increased JavaScript initialized size which greatly affects users' experience and web application's SEO. While solutions such as hydration and island architectures are concluded as workarounds only, the thesis discusses the relatively new Qwik framework which aims to fully tackle the root of this issue. Qwik introduces the concept of resumability and progression, which involve pausing the server execution and resuming it in the client side. This approach enables Qwik web applications to load almost instantly and avoids the impracticality of hydration, as it only retrieves the HTML during its initial load, leading to a seamless user experience. The use of Qwik as a frontend framework is demonstrated in the implementation of Filmmash, which produces significant results based on evaluated metrics.

6.3. Future Applicability

Headless CMS and Qwik framework are innovative technologies equipped with advanced features that can be implemented in a wide range of web products, indicating their potential significance in the future. One likely use case for the applicability of these technologies is to develop customizable page builders. As the demand for easy-to-use page builders increases, Headless CMS can provide the backend infrastructure for creating a more flexible and customizable solution. The separation of content and presentation offered by Headless CMS allows for more streamlined, scalable, and efficient content management, saving up more time for developers to focus on building front-end interfaces. Incorporating Qwik framework into page builder development can also bring significant benefits. The framework's resumability and progression concepts can enhance page builder performance, allowing for faster load times and more efficient component loading.

Builder.io is the first to implement its page builder integrating both technologies, with other businesses likely to follow. The company at which the writer of this thesis is currently working, whose flagship product being the PageFly Landing Page Builder, is also actively researching and conducting tests on the applicability of Headless CMS and Qwik framework, with a view to integrate these technologies into the application. It is hoped that the findings of this research will serve as a useful reference for future studies and applications of Headless CMS and Qwik in web development.

REFERENCES

Articles

Arias, D. (2021a). *Hashing in Action: Understanding bcrypt*. Auth0. Accessed April 14, 2023. Retrieved from <https://auth0.com/blog/ hashing-in-action-understanding-bcrypt>.

Arias, D. (2021b). *Adding Salt to Hashing: A Better Way to Store Passwords*. Auth0. Accessed April 14, 2023. Retrieved from <https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords>.

Berners-Lee, T. (1989). *Information Management: A Proposal*. World Wide Web Consortium. Accessed February 27, 2023. Retrieved from <https://www.w3.org/History/1989/proposal-msw.html>.

Butti (n.d.) *Headless CMS explained in 5 minutes*. Storyblok. Accessed February 28, 2023. Retrieved from <https://www.storyblok.com/tp/headless-cms-explained>.

Carniato, R. (2022). *Why Efficient Hydration in JavaScript Frameworks is so Challenging*. Dev.to. Accessed March 12, 2023. Retrieved from <https://dev.to/this-is-learning/why-efficient-hydration-in-javascript-frameworks-is-so-challenging-1ca3>.

Champeon, S. (2001). *JavaScript, How Did We Get Here?*. O'Reilly Media, Inc. Archived July 19, 2016. Accessed March 10, 2023. Retrieved from https://web.archive.org/web/20160719020828/https://archive.oreilly.com/pub/a/javascript/2001/04/06/js_history.html.

Chrome Developers (2019). *Lighthouse performance scoring*. Accessed March 13, 2023. Retrieved from <https://developer.chrome.com/en/docs/lighthouse/performance/performance-scoring>.

Contentful (n.d. a). *Contentful content APIs*. Accessed March 1, 2023. Retrieved from <https://www.contentful.com/developers/docs/concepts/apis>.

Contentful (n.d. b). *Domain model*. Accessed March 1, 2023. Retrieved from <https://www.contentful.com/developers/docs/concepts/domain-model>.

Contentful (n.d. c). *Headless CMS explained in 1 minute*. Accessed February 28, 2023. Retrieved from <https://www.contentful.com/r/knowledgebase/what-is-headless-cms>.

Contentful (n.d. d). *Separate content from code: Build faster using our Free plan*. Accessed February 28, 2023. Retrieved from <https://www.contentful.com/free-plan>.

Cromwell, V. (2017). *Between the Wires: An interview with Vue.js creator Evan You*. Medium. Accessed March 12, 2023. Retrieved from <https://medium.com/free-code-camp/between-the-wires-an-interview-with-vue-js-creator-evan-you-e383cbf57cc4>.

Dewhurst, A. (2022). *Model View Controller: How to Use the MVC Architecture to Achieve Separation of Concerns*. Medium. Accessed March 11, 2023. Retrieved from <https://medium.com/@andrew.dewhurst8/model-view-controller-how-to-use-the-mvc-architecture-to-achieve-separation-of-concerns-1042c093f51d>.

Dineley, D. (2008). *Best of open source software awards: Collaboration*. InfoWorld. Accessed February 27, 2023. Retrieved from <https://www.infoworld.com/article/2638571/best-of-open-source-software-awards--collaboration.html>.

Dooley, R. (2014). *Power of Ten: The Weird Psychology of Rankings*. Straylight. Accessed March 20, 2023. Retrieved from <https://www.straylight.se/power-of-ten-the-weird-psychology-of-rankings>.

Ebert, R. (2012). *The best damned film list of them all*. RogerEbert.com. Accessed March 20, 2023. Retrieved from <https://www.rogerebert.com/roger-ebert/the-best-damned-film-list-of-them-all>.

Elo, A. E. (1967). *The Proposed USCF Rating System: Its Development, Theory, and Applications*. United States Chess Federation. Accessed March 20, 2023. Retrieved from http://uscf1-nyc1.aodhosting.com/CL-AND-CR-ALL/CL-ALL/1967/1967_08.pdf#page=26.

Envall, N. (2022). *History of JavaScript Framework*. Programming Soup. Accessed March 10, 2023. Retrieved from <https://programmingsoup.com/history-of-javascript-frameworks#birth-of-javascript-frameworks>.

Fu, J. (2022). *Introducing Qwik — The JavaScript Framework With $O(1)$ Load Time*. Better Programming. Accessed February 25, 2023. Retrieved from <https://betterprogramming.pub/qwik-the-javascript-framework-with-o-1-load-time-222f30613361>.

FullStackOverflow (2022). *2022 Developer Survey*. Accessed March 11, 2023. Retrieved from <https://survey.stackoverflow.co/2022/#most-popular-technologies-webframe>.

Garrett, J. J. (2005). *Ajax: A New Approach to Web Applications*. Adaptive Path. Accessed March 11, 2023. Retrieved from https://designftw.mit.edu/lectures/apis/ajax_adaptive_path.pdf.

Hall, C. (2021). *Builder.io aims to make developers happy with its no-code approach to digital storefronts*. TechCrunch. Accessed March 13, 2023. Retrieved from <https://techcrunch.com/2021/10/18/builder-io-aims-to-make-developers-happy-with-its-no-code-approach-to-digital-storefronts>.

Hevery, M. (2022a). *Interview with Misko Hevery, Chief Technology Officer at Builder.io*. Devmio. Accessed February 25, 2023. Retrieved from <https://devm.io/javascript/qwik-javascript-hevery>.

Hevery, M. (2022b). *Hydration is Pure Overhead*. Builder.io. Accessed March 12, 2023. Retrieved from <https://www.builder.io/blog/hydration-is-pure-overhead>.

Hevery, M. (2022c). *Death by Closure (and how Qwik solves it)*. Dev.to. Accessed March 13, 2023. Retrieved from <https://dev.to/builderio/death-by-closure-and-how-qwik-solves-it-44jj>.

Hevery, M. (2022d). *Our Current Frameworks are $O(n)$; We Need $O(1)$* . Builder.io. Accessed March 13, 2023. Retrieved from <https://www.builder.io/blog/our-current-frameworks-are-on-we-need-o1>.

Hevery, M. (2022e). *Qwik: the answer to optimal fine-grained lazy loading*. Dev.to. Accessed March 13, 2023. Retrieved from <https://dev.to/builderio/qwik-the-answer-to-optimal-fine-grained-lazy-loading-2hdp>.

Heslop, B. (2022). *History of Content Management Systems and Rise of Headless CMS*. Contentstack. Accessed February 27, 2023. Retrieved from <https://www.contentstack.com/blog/all-about-headless/content-management-systems-history-and-headless-cms>.

Hoffman, C. (2010). *The Battle For Facebook*. Rolling Stone. Accessed March 20, 2023. Retrieved from <https://www.rollingstone.com/culture/culture-news/the-battle-for-facebook-242989>.

Holmes, J. (2022). *Headless CMS vs. Traditional CMS*. Sanity.io. Accessed February 25, 2023. Retrieved from <https://www.sanity.io/headless-cms/headless-vs-traditional-cms>.

Lardinois F. (2018). *Contentful raises \$33.5M for its headless CMS platform*. TechCrunch. Accessed February 28, 2023. Retrieved from <https://techcrunch.com/2018/12/06/contentful-raises-33-5m-for-its-headless-cms-platform>.

Mahemoff, M. (2005). *Ajax Frameworks*. AjaxPatterns.org. Archived January 12, 2006. Accessed March 11, 2023. Retrieved from https://web.archive.org/web/20060112050544/http://ajaxpatterns.org/Ajax_Frameworks.

Maheshwari, D. (2021). *Why Single Page Application (SPA) architecture is so popular?* Medium. Accessed March 11, 2023. Retrieved from <https://medium.com/nerd-for-tech/why-single-page-application-spa-architecture-is-so-popular-141b85400204>.

May, A. (2019). *Happy 30th birthday, World Wide Web. Inventor outlines plan to combat hacking, hate speech*. USA Today. Accessed February 27, 2023. Retrieved from <https://www.usatoday.com/story/tech/news/2019/03/12/world-wide-web-turns-30-berners-lee-contract-thoughts-internet/3137726002>.

Melvaer, K. (2023). *Headless CMS explained in 1 minute*. Sanity.io. Accessed February 25, 2023. Retrieved from <https://www.sanity.io/headless-cms>.

Muzammil, K. (2021). *Importance of Web Application Architecture*. Aalpha. Accessed March 24, 2023. Retrieved from <https://www.aalpha.net/articles/importance-of-web-application-architecture>.

Nguyen, K. (2018). *How to Manage the Business Better in the Omnichannel World*. Magestore. Accessed April 26, 2023. Retrieved from <https://blog.magestore.com/omnichannel-management>.

Node Package Manager (2023). *@builder.io/qwik*. Accessed February 25, 2023. Retrieved from <https://www.npmjs.com/package/@builder.io/qwik>.

Ottervig, V. (2022). *The Essential History of CMS*. Enonic. Accessed February 27, 2023. Retrieved from <https://enonic.com/blog/the-history-of-cms--what-has-happened>.

Ottervig, V. (2023). *What is structured content?*. Enonic. Accessed February 28, 2023. Retrieved from <https://enonic.com/blog/what-is-structured-content>.

Papiernik, M. (2021). *How To Use Sharding in MongoDB*. DigitalOcean. Accessed March 24, 2023. Retrieved from <https://www.digitalocean.com/community/tutorials/how-to-use-sharding-in-mongodb>.

Pope, L. (2021). *Content model: Why you need one and how to make your colleagues take notice*. GatherContent. Accessed March 1, 2023. <https://gathercontent.com/blog/why-you-need-a-content-model-how-to-make-colleagues-take-notice>.

Postma, B. (2022). *How to Deploy the Qwik JavaScript Framework*. Netlify. Accessed March 24, 2023. Retrieved from <https://www.netlify.com/blog/how-to-deploy-the-qwik-javascript-framework>.

Qwik (n.d.). *Optimizer*. Accessed March 13, 2023. Retrieved from <https://qwik.builder.io/docs/advanced/optimizer>.

Roser, M. (2018). *The Internet's history has just begun*. Our World in Data. Accessed February 27, 2023. Retrieved from <https://ourworldindata.org/internet-history-just-begun>.

Serby, P. (2012). *Case study: How & why to build a consumer app with Node.js*. VentureBeat. Accessed March 24, 2023. Retrieved from <https://venturebeat.com/dev/building-consumer-apps-with-node>.

Singh, S. (2022). *React and Next.js is DEAD — Something New is (Finally) Replacing It (For Good)*. PlainEnglish.io. Accessed March 13, 2023. Retrieved from <https://plainenglish.io/blog/react-and-next-js-is-dead-something-new-is-finally-replacing-it-for-good-c792c48806f6>.

Sonas, J. (2002). *The Sonas Rating Formula – Better than Elo?*. ChessBased. Accessed March 20, 2023. Retrieved from <https://en.chessbase.com/post/the-sonas-rating-formula-better-than-elo>.

StatCounter (2023). *Desktop vs Mobile Market Share Worldwide - February 2023*. Accessed February 28, 2023. Retrieved from <https://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide/#monthly-200901-202301>.

Team of Horses Website Management (2019). *What is a m-dot site*. Accessed February 28, 2023. Retrieved from <https://www.tohwebmasters.com/mobile-website-design-what-is-an-m-dot-site>.

United States Chess Federation (2013). *K-Factor Change - May 2013*. Accessed March 20, 2023. Retrieved from <https://www.uschess.org/index.php/Announcements/K-Factor-Change-May-2013.html>.

U.S. Department of Labor (1999). *Issues in Labor Statistics*. Accessed February 27, 2023. Retrieved from <https://www.bls.gov/opub/btn/archive/computer-ownership-up-sharply-in-the-1990s.pdf>.

Veisdal, J. (2019). *The Mathematics of Elo Ratings*. Cantor's Paradise. Accessed March 20, 2023. Retrieved from <https://www.cantorsparadise.com/the-mathematics-of-elo-ratings-b6bfc9ca1dba>.

Wemanity (2022). *A Brief History of Content Management Systems*. Accessed February 27, 2023. Retrieved from <https://weblog.wemanity.com/en/a-brief-history-of-content-management-systems>.

Books and Research Papers

Chun, J. S., & Larrick, R. P. (2022). The power of rank information. *Journal of Personality and Social Psychology*, (pp. 983–1003). <https://doi.org/10.1037/pspa0000289>.

Cotton, I. W. & Grestorex, F. S. (1968). Data structures and techniques for remote computer graphics. In AFIPS (Ed.), *AFIPS '68 (Fall, part I): Proceedings of the December 9-11, 1968, fall joint computer conference, part I* (pp. 533-544). New York, NY: Association for Computing Machinery. <https://dl.acm.org/doi/10.1145/1476589.1476661>.

Date, C. J. & Codd, E. F. (1975). The relational and network approaches: comparison of the application programming interfaces. In SIGFIDET (Ed.), *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control: Data models: Data-structure-set versus relational* (pp. 83-113). New York, NY: Association for Computing Machinery. <https://doi.org/10.1145/800297.811532>.

Fielding, R. T. (2000), *Architectural styles and the design of network-based software architectures* (pp. 76-147). University of California, Irvine. <https://dl.acm.org/doi/10.5555/932295>.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture* (pp. 116). Addison-Wesley Professional.

Gao, Z., Bird, C. & Barr, E. T. (2017). "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript". *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, (pp. 758-769). <http://dx.doi.org/10.1109/ICSE.2017.75>.

MoSCoW Analysis (6.1.5.2) (2009). *A Guide to the Business Analysis Body of Knowledge (2 ed.)*. International Institute of Business Analysis. ISBN 978-0-9811292-1-1.

Piotrowicz W. & Cuthbertson R. W. (2014). *Introduction to the Special Issue Information Technology in Retail: Toward Omnichannel Retailing* (pp. 5-16). International Journal of Electronic Commerce. <https://www.tandfonline.com/doi/abs/10.2753/JEC1086-4415180400>.

Provos, N. & Mazieres, D. (1999). A Future-Adaptable Password Scheme. In USENIX (ed.). *Proceedings of 1999 USENIX Annual Technical Conference* (pp. 81-92). <https://dl.acm.org/doi/proceedings/10.5555/1268708>.

Rauschmayer, A. (2014). *Speaking Javascript*. O'Reilly Media, Inc. <https://dl.acm.org/doi/book/10.5555/2614440>.

Yermolenko A. & Golchevskiy Y. (2021). *Developing Web Content Management Systems – from the Past to the Future* (pp. 5). SHS Web of Conferences. <http://dx.doi.org/10.1051/shsconf/202111005007>.