UNIVERSITAT DE BARCELONA

Final degree project

DEGREE IN COMPUTER SCIENCE

Department of Mathematics and Computer Science
University of Barcelona

Implementation of a full-stack blog content management system with a specialized rich text editor

Author: Kamil Natan Malysa

Advisor:	Xavier Baró
Conducted at:	Faculty of Mathematics and Computer Science
Barcelona,	June 9, 2025

Abstract

Content management systems for publishing blog-like websites are widely used across the internet, however despite advancements in web development technologies, the existing solutions are reluctant to take advantage of them.

Established blog content management systems find themselves in environments with declining developer support, and content editing tools that have fallen behind standards set by new frameworks.

This project explores the process of implementing a blog-centric content management system using modern web technologies and a RESTful backend, employing emerging frameworks to build an alternative that takes advantage of the rich content editing user experience that these frameworks enable.

The content management system created in this project makes use of Nuxt and FastAPI to create a robust full-stack blogging solution, with an emphasis on the user experience of writing blog content through a custom rich text editor built upon the ProseMirror framework.

Keywords: REST API, Nuxt.js, Vue, web development, FastAPI, content management systems (CMS), blogging, rich text editing, MVVM (Model-view-viewmodel), SQL databases, ProseMirror

Resum

Sistemes de gestió de contingut per la publicació de llocs web de tipus blog són en ampli ús a l'internet, però malgrat avanços en tecnologies de desenvolupament web, les solucions existents mostren poc interès per aprofitar-les.

Els sistemes establerts de gestió de contingut de blogs es troben en entorns amb declivi de suport per part de desenvolupadors, i eines d'edició de contingut que no assoleixen els estàndards establerts per nous *frameworks*.

Aquest projecte explora el procés d'implementar un gestor de contingut orientat a blogs, usant tecnologies de web modernes i un *backend* d'arquitectura REST, usant *frameworks* emergents per construir un gestor alternatiu que aprofita les millores d'experiència d'usuari que aquestes eines permeten quant a l'edició de contingut.

El sistema de gestió de contingut creat en aquest projecte fa ús de Nuxt i FastAPI per construir una solució robusta per blogging, posant èmfasi en l'experiència d'usuari d'escriure contingut de blog amb un editor de text ric propi fet amb el *framework* de ProseMirror.

Paraules clau: REST API, Nuxt.js, Vue, desenvolupament web, FastAPI, sistemes de gestió de contingut, blogging, edició de text, MVVM (Model-view-viewmodel), bases de dades SQL, ProseMirror

Resumen

Sistemas de gestión de contenido para la publicación de sitios web de tipo blog tienen amplio uso en el internet, pero a pesar de avances de tecnologías de desarrollo web, las soluciones existentes no las aprovechan.

Los sistemas establecidos de gestión de contenido de blogs se encuentran en entornos en decadencia en cuanto a soporte de desarrolladores, y con herramientas de edición de contenido que no alcanzan los estándares establecidos por nuevos *frameworks*.

Este proyecto explora el proceso de implementar un gestor de contenido orientado a blogs, usando tecnologías de web modernas y un *backend* con arquitectura REST, usando *frameworks* emergentes para construir un gestor alternativo que aprovecha las mejoras de experiencia de usuario que estas herramientas permiten en cuanto a edición de contenido.

El sistema de gestión de contenido creado en este proyecto usa Nuxt y FastAPI para construir una solución robusta para blogging, con énfasis en la experiencia de usuario en cuanto a la escritura de contenido blog mediante un propio editor de texto rico hecho con el *framework* de ProseMirror.

Palabras clave: REST API, Nuxt.js, Vue, desarrollo web, FastAPI, sistemas de gestión de contenido, blogging, edición de contenido rico, MVVM (Model-view-viewmodel), bases de datos SQL, ProseMirror

Acknowledgements

I would like to express my gratitude to my tutor, Xavier Baró, for his guidance throughout this final degree project. His knowledge and advice were essential in the execution of the project, and I sincerely appreciate the time he has taken to mentor me.
I would also like to give special thanks to Matteo "Ameranth" T. and Davide "Elric" T. for being the highest inspiration for my software engineering values. Their meticulous design philosophies and support throughout the years have greatly shaped my ambitions as a programmer, and have influenced this project as well.

Contents

1 Introduction

This final degree project explores a full-stack implementation of a blog-oriented content management system using modern web development frameworks, with an emphasis on the user experience of writing blog content through a custom rich text editor with specialized features to cover emerging blogging needs.

Blogs are informational websites hosting text content, often managed by individuals or small groups, posting periodic articles on specific topics. Ever since their emergence in the late 1990s, blogs have remained one of the most common types of websites, with there being over 700 million blogs on the internet as of 2024, making up for over 40% of internet traffic[**12**]. Blogs typically employ specialized content management systems (CMS) to generate their websites, together with tools for the blog's owners to facilitate the creation of posts.

In spite of advances in web technologies and development frameworks, the majority of blogs today are powered by established solutions from the 2000s[**2**], with there being few attempts at building new blog content management systems that take advantage of emerging web frameworks and their potential to deliver better experiences for both users and developers.

Numerous leading solutions like WordPress[1] are built with languages experiencing a decline in usage such as PHP[2][3], creating concerns for longevity and extendability. Other solutions like Tumblr[3] are centralized, lacking self-hosting options, and have lagged behind the growing standards of content editing tools.

Newer web frameworks have enabled much more elaborate user experiences for editing rich content, yet the post editors for many of these blogging solutions fail to match the expectations for content editing that have been set by web applications of other sectors.
In a time when content creation is becoming ever more collaborative and users expect web content to be both informative and attractive, solutions like Tumblr or Medium[4] still support little more than plain text and images, and lack collaborative features to allow multiple individuals to work on the same blog effectively.

This project aims to fill the need for self-hostable blog options, using modern frameworks to ease extendability, and a custom rich text editor that addresses common shortcomings of the post editors of current blogging systems.

The motivation for the project comes from personal interest in how such blogging systems are implemented, as well as from frustration with the outdated features and low efficiency of the rich text editors that these systems typically employ for content creation. I was interested in exploring the development of such blogging systems from start to finish, together with tackling the challenges of dedicated content creation tools that users now expect for them.

Blogging was one of my first contacts with computer science, through Jekyll[5] and its HTML templating engine, sparking a later interest in web development.

[1] WordPress: web content management system, first developed for the creation of blog sites.
[2] PHP: multi-paradigm programming language oriented for web development.
[3] Tumblr: centralized blogging and social networking site active since 2007.
[4] Medium: blog publishing platform and social network.
[5] Jekyll: blog-oriented static site generator written in Ruby.

With a renewed interest in blogging and the skills I acquired throughout my studies, I was fascinated by the variety of challenges that come with implementing such a CMS and a specialized rich text editor, and wished to experience first-hand the complexity of their common features that are often taken for granted.

1.1 Objectives

The project intends to develop a full-stack, web-based content-management system (CMS) for self-hosted blog websites, for which the staff members can publish and manage text articles through a custom rich text editor specialized for blog content.

The project would allow editing and publishing articles, with collaborative features to support managing blogs where content is authored by a group of people, a use case that is neglected by many of the formerly mentioned solutions.
The functionality would be generic to support multiple blog-like use cases, including personal homepages and posting announcements for events.

The site staff would be able to manage content intuitively through a web-based admin control panel and the aforementioned rich text editor for writing articles. Site visitors would be able to browse and search articles, and interact with the owners of the site through comment sections and signing up for events.

The end result aims to be a CMS usable out-of-the-box as a generic self-hosted blog that does not require the user to edit files or source code directly, instead offering web interfaces to manage the content and write attractive articles with its integrated editor.

The custom rich text editor would be a key component of the project, aiming to provide a good blog content creation experience inspired by features and user experience patterns from other modern text editors.Developing it alongside the rest of the features of a blog CMS gives it a realistic context of usage, ensuring coherency in the editor's functionalities.

The project thus has **3 main objectives**:

1. Implement a **backend system with a REST API** for common blogging functionality, such as indexing articles, asset management, commenting, etc.

2. Design a **modern site for serving and managing the blog's content**, using modern frameworks to build an admin interface that integrates the API to manage the blog in a user-friendly manner.

3. Build a **custom rich text editor** with features specifically oriented towards blog content, with flexibility to be extended and reusable in other similar contexts.

The following section will outline the structure of the document and the various phases of the project's development.

1.2 Document structure

The document will detail the 4 months of the project's development, divided into the following major sections:

- **Project planning** (Section 2), which details the preliminary task of organizing the workload of the project, as well as techniques to track its development and stay on schedule.
- **Requirements gathering** (Section 3), where the functional and non-functional requirements of the project were analysed, together with artifacts that were used to guide development and design.
- **Technology** (Section 4), which details the frameworks and architecture chosen for the project based on its requirements.
- **Implementation and development** (Section 5), which covers the technical details and implementation choices of the project. This section is further divided into subsections for each major component of the project.
- **Deployment** (Section 6), which details how the application was launched in a production environment.
- **Evaluation** (Section 7), where the application was tested using usability testing and auditing tools to assess its results and issues, comparing it to existing solutions.

2 Project planning

The project's workload was organized using a Gantt chart(Fig. 1), defining feature implementation and documentation tasks, together with the intended time frames to complete them.

The tasks were based on the functional requirements gathered in Section 3, and ordered by the importance of the features they were part of, such that major features of the project were worked on earlier and functionality with less value was planned for later weeks.

Color legend: Frontend | Backend | Docs | Evaluation

Task	Completion notes	Start date	Days	End date
Plan user API routes	Done	2/17/25	2	2/18/25
Implement user tables & APIs	Done	2/17/25	4	2/20/25
Create UI mockups	Done	2/19/25	4	2/22/25
Add staff login page	Done	2/22/25	2	2/23/25
Add dockerfile	Done early	2/22/25	2	2/23/25
Add user control panels	Done early	2/24/25	4	2/27/25
Plan article APIs	Done	2/27/25	2	2/28/25
Add article tables & APIs	Done	3/2/25	4	3/5/25
Add content management panels	Done	3/5/25	3	3/7/25
Add article editor base functionality	Done early	3/6/25	5	3/10/25
Integrate content management and editor	Done	3/10/25	2	3/11/25
Add article and category pages	Done	3/10/25	4	3/13/25
Add article pagination	Done	3/13/25	2	3/14/25
Add article metadata forms	Done	3/15/25	2	3/16/25
Add semantic article editor blocks	Done	3/17/25	5	3/21/25
Add image handling	Done	3/24/25	3	3/26/25
Add article editor image support	Done	3/27/25	3	3/29/25

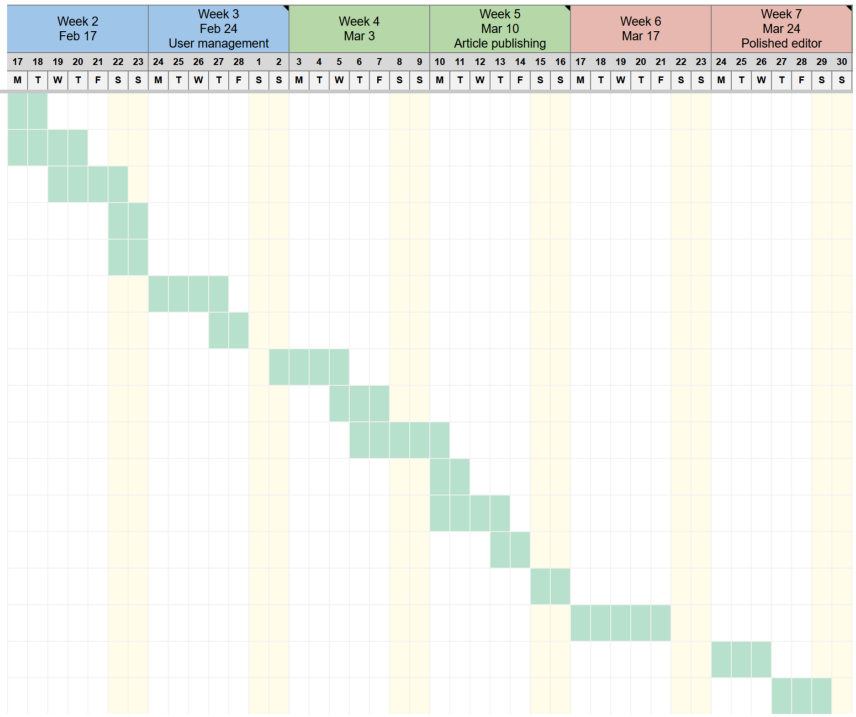


Figure 1: Fragment of the Gantt chart spreadsheet used for planning work on the major features.

The tasks were also grouped into 2-week milestones to give clear short term goals, with each milestone denoted by additions of key features. They acted as minimum viable products (MVPs[6]), helping keep the project on track and giving every week of work a clear objective. Some of the major milestones were:

1. User account system and management through a control panel site
2. Basic article editor functionality and publishing articles to the site
3. Navigation and search in the published site, comment system, and file management
4. Advanced article editor functionality and visual improvements

The first week of development was dedicated to implementing generic systems (mainly user management); these were worked on to ease becoming familiar with the frameworks chosen before attempting to add the main features, as well as due to them being a prerequisite for many core features.

The tasks of the Gantt chart spanned until the end of May, so as to allow for $\sim$2 weeks of deviation in execution. The Gantt chart was only used for scheduling these major features and milestones to give an overview of the expected timeline

[6]Minimum viable product: early version of a software with just enough features to be usable and evaluable.

of the project. Daily work was tracked using a kanban board, described in Section 2.1.

After reaching a minimum viable product milestone in late April, the planning of the two weeks of May was changed to focus on refactoring and visual polishing of the frontend, dropping various minor features that were originally intended to be added at that time. This decision was brought forth by an evaluation of the project up until that point, where the frontend's code architecture and styling were deemed to not meet personal standards of quality.
With the frontend experience acquired in the first half of the project, it was considered worthwhile to thoroughly refactor the article editor component of the frontend to be able to present a more elegant and maintainable solution.

Despite these few issues in the planning, it has overall served the project well; time estimates were accurate (with only a few overestimations noted in the chart), and milestones helped to structure work and deliver stable MVPs with valuable features. The full Gantt chart and its older version are available in Annex 10.7.

2.1 Work methodology

The project used Git for version control, with the repository being hosted on GitHub at `https://github.com/PinewoodPip/bloggy`. A mono-repo approach was used, with both the backend and frontend in the same repository, as it was the most efficient choice for a single-person project. A simple branching strategy was used with only a main branch for stable versions (coinciding with Gantt chart milestones) and a development branch for daily work, both covered by continuous integration through automated testing(*See annex* 10.6).

During development, the features from the Gantt chart were split into smaller, more granular implementation tasks that were tracked using the GitHub Projects service, using it as a kanban board(*See annex figure* 47).
This kanban board was also used to keep track of found bugs, possible improvements, and refactoring work, using labels to organize and filter tasks by their nature and the part of the project they concerned.

Daily work would focus on the corresponding features from the Gantt chart, paced using the Pomodoro technique by working on the project ∼5 hours daily in 30-minute intervals separated by short breaks. After finishing a feature, the remaining time was used for improvements and bug fixes, pulling work from the GitHub Projects board. This document would be worked on every weekend, noting down the results and decisions of the week's work, which also offered an opportunity to reflect on them.

The following sections will cover the initial work of the project's requirements gathering and the choices of technologies used.

3 Requirements gathering

The project's development started with an analysis of its requirements - the functionality that the finished product is expected to have. Even in the case of a single-person project, these are necessary to limit the project's scope so that its planning and design can be stable.

The requirements gathering started with identifying the types of users of the software (the **user roles**), with their motivations and goals defining the expected functionality. The following sections cover how these were deduced, together with the documentation artifacts that served to guide the design and implementation of the project.

3.1 User roles

Any CMS distinguishes at least two groups of users: the staff that creates and manages the content of the system and its consumers. In the case of a blogging-oriented CMS, the consumers would be the site visitors who come to read the articles, which are written by a team of editors.

These two groups need to be divided further to get a more complete view of the user roles, as there are functional needs to distinguish between logged-in visitors and "guests", as well as distinguishing the staff based on authority over the system. With this in mind, the project identifies the following user roles:

- **Reader**: a guest visitor of the site who has read-only access to the content; they can only read published articles and search them.
- **Authenticated reader**: a visitor that is logged-in to the site and can perform actions that need to be tied to an identity, such as commenting or subscribing to events, in addition to the abilities of the reader role.
- **Editor**: a logged-in user who is authorized to manage the content on the website (create and edit articles and categories). The content they have authority over would be limited by a permissions system.
- **Admin**: a logged-in superuser who can create editor accounts and manage their permissions, as well as edit site-wide configuration.
- **Staff member**: alias that refers to an editor or admin role.

The admin and editor roles are both staff roles that manage the site, differing in the authority they have.

The functional requirements as well as use cases were then derived from the needs and expected capabilities of those user roles, creating artifacts to guide the project's development.

3.2 Artifacts

The project's requirements gathering was done using multiple types of design artifacts, as none of them are comprehensive when used alone. The process started with artifacts focused on end-user desires, followed by more technical artifacts that focus on system behaviour and flow. In order, the artifacts created were:

- **User stories** to capture the features end-users expect and their motivation.
- **Functional requirements** to extract more detailed and technical requirements from the user stories.
- **Use cases** to define interaction flows of the various features from start to finish.

These are briefly covered over the next sections, with complete lists of all the artifacts being available in Annex 10.10.

3.2.1 User stories

User stories describe the functionality each user role expects in a one-line sentence using the following structure:
As a ***{user role}*** *I want to* ***{action}*** *so that* ***{value statement}***.

They provide a user-centric overview of expected features, useful for the value statement - the motivation behind why users want the functionality described[5]. The user stories thus helped justify the features and were a good starting point for deriving other artifacts.

The project deduced user stories from informally interviewing users and analysing existing solutions, grouping the stories into 6 "epics" to organize them by overarching theme:

- **User management**: administrative actions over staff accounts.
- **Content management**: CMS operations such as creating articles or editing their metadata.
- **Article writing**: functionality of the custom article editor.
- **Content exploration**: usage of the published site (navigation, search, etc.).
- **Event creation**: user stories related to scheduling events.
- **Site management**: user stories related to site-wide configuration needs.

Usually, acceptance criteria would be written for each user story to define when they're considered completed; in the project's case, the functional requirements and use case artifacts were used to fulfill this need instead.

The list of user stories is as follows(Fig. 2), presented in a table format where the columns correspond to the parts of the user story format (role, action, value statement) to list each story in a compacted manner. The table also denotes which user stories had their functional requirements implemented by the end of development, as well as reasons for the ones that were ultimately not worked on due to being low-value or being dropped due to planning changes.

Epic	As a...	I want to...	So that...	Implemented	Reason for being unimplemented
User management	Admin	create an editor account	another staff member can manage the site	☑	
	Staff member	log in using a username and password	I can access the site's control panel	☑	
	Editor	edit my profile's name, avatar and biography	it reflects my identity better to readers	☑	
	Admin	set which categories an editor can manage articles in	I can ensure they cannot change content they don't work on	☐	Was deemed low-value; not included in planning.
	Admin	delete an editor account	I can prune accounts of people no longer working with the site	☑	
	Staff member	change my account's password	I can safeguard my account if I suspect my credentials were leaked	☑	
	Staff member	enable 2 factor authentication for my account	I can keep my account more secure	☐	Dropped due to time constraints to focus on other features.
Content management	Editor	create a category to group articles	I can organize articles thematically in a folder-like structure	☑	
	Editor	create an article under a category	I can make a new post to the site	☑	
	Editor	set an article be published at a specific time	it can be posted to the site at a planned time automatically	☑	
	Editor	add text tags to an article	I can label articles by their topics and themes	☑	
	Editor	save an article as a draft	I can work on it later without posting it to the site yet	☑	
	Editor	set whether comments can be posted on an article	I can allow or disallow readers to discuss it	☑	
	Editor	mark other editors as co-authors	they can be credited for helping with the article	☑	
	Editor	set whether the article displays metadata (its authors, post date)	I can hide it in articles where it's not relevant	☑	
	Editor	set a summary for an article to be shown in pagination	I can make the article's preview more attractive	☑	
	Editor	set a featured image for an article to be shown as a thumbnail	I can make it more visually attractive in pagination	☑	
	Editor	set a category to display articles in a chronological list	I can present articles in the order they were posted	☑	
	Editor	set a category to display articles in a card grid	I can present articles more attractively	☑	
	Editor	to move an article to another category	I can reorganize it to a more appropriate category	☑	
	Editor	upload an image or file to the CMS	I can link it in articles I write	☑	
	Editor	set an article to span multiple pages based on its headings	a long article can be presented in a more digestible manner	☐	Was deemed low-value; not included in planning.
	Editor	to edit the order in which articles appear in a category	I can display articles in a custom order	☑	
Article writing	Editor	edit existing articles	I can amend their content	☑	
	Editor	format text within articles as bold, italics or underline	I can emphasize certain parts of the text	☑	
	Editor	insert images by uploading them	I can show images I had on my device	☑	
	Editor	insert images that were already uploaded to the site	I can embed images that I already used before in another article	☑	
	Editor	insert images by hotlinking them	I can embed images from external sites	☑	
	Editor	insert code blocks with syntax highlighting in articles	I can showcase code snippets attractively	☑	
	Editor	insert section headings in articles	I can organize content into sections	☑	
	Editor	set an element's alignment in the article editor	I can center or uncenter paragraphs, images and other figures	☑	
	Editor	copy/cut & paste content within an article	I can reorder content easily	☑	
	Editor	add links in articles	I can reference other articles or sites	☑	
	Editor	add YouTube embeds to articles	I can include a related video	☑	
	Editor	add tables with text cells to articles	I can show tabular information	☐	
	Editor	add MathJax blocks to articles	I can display math formulas	☐	Dropped due to time constraints to focus on polishing the other editor features.
	Editor	insert math symbols from a searchable list into an article	I can write math formulas more easily	☐	
	Editor	add a card with my biography at the end of an article	readers can learn more about me after reading the article	☑	
	Editor	add footnotes to articles	I can add details or clarifications about specific parts	☑	
	Editor	add notes/callouts to articles with different colors	I can call attention to warnings or important details	☑	
	Editor	add nestable bullet or numbered lists to articles	I can itemize or enumerate content	☑	
	Editor	add quote blocks to articles	I can denote text written by someone else	☑	
	Editor	set whether an article is visible on the site	I can hide an article temporarily without deleting it	☑	
	Editor	delete an article	I can remove it from the site	☐	Was deemed low-value; articles may be hidden as a workaround/alternative.
	Editor	undo an operation in the article editor	I can recover from a mistake	☑	
	Editor	redo an operation in the article editor	I can recover an undone operation if I change my mind	☑	
	Editor	insert emojis into an article from a searchable list	I can express myself easily on devices that don't have emoji keyboards	☑	
	Editor	insert content using Markdown syntax	I write articles more efficiently with markup syntax I'm familiar with	☑	
	Editor	customize keyboard shortcuts for article editor operations	I can use keybinds that I find comfortable and efficient	☑	
	Editor	customize which editor operations are visible in a toolbar	I can hide operations that are not relevant to my kind of articles	☑	
	Editor	reorder editor operations in the toolbar	I can customize the toolbar layout to my liking	☐	Was deemed low-value; not included in planning.
	Editor	have unsaved article changes be cached on my device	I can restore them if I leave the page without saving	☐	
	Editor	create article templates to reuse when creating new articles	I can write multiple articles with a common structure more quickly	☐	Dropped to focus on polishing the other editor features.
	Editor	create macros to insert content to articles repeatably	I can insert repeating structures more easily	☐	
	Editor	simultaneously edit an article with other editors	I can work on an article together with other team members	☐	Was a stretch goal, not included in planning.
	Editor	add annotations to articles that only other editors can see	I can leave notes on comments to ease collaboration	☑	
	Editor	set a description for a category	readers can quickly know what its articles are about	☑	
Content exploration	Reader	browse articles of a category	I can find articles of a specific topic	☑	
	Reader	search articles by their text content and name	I can find specific articles I know keywords of	☑	
	Reader	browse articles by their tags	I can find other articles about a specific theme	☑	
	Reader	browse articles by their author	I can find other articles written by an author I like	☑	
	Reader	log-in using a third-party platform SSO	I don't need to create an account just for the site	☑	
	Authenticated reader	to post a comment on an article	I can tell the author my opinion or ask questions	☑	
	Editor	to reply to a comment on an article	I can respond to feedback or questions	☑	
	Editor	to delete a comment on an article	I can remove inappropriate comments	☑	
	Authenticated reader	to delete my own comment on an article	I can retract a comment I no longer want others to see	☑	
	Reader	to subscribe to an RSS feed of the website	I can receive notifications when new articles are posted	☐	Dropped due to time constraints.
	Reader	to share articles to common social media platforms	I can quickly send an interesting article to my followers	☑	
	Editor	to add text patterns to filter out undesirable comments	I don't need to manually moderate every comment	☐	Was deemed low-value due to commenting itself also being a minor feature.
	Editor	to approve filtered-out comments to be shown publicly	I can whitelist comments that were wrongly filtered out	☐	Was deemed low-value; not included in planning.
Event creation	Editor	to create events under a category	I can announce it to readers	☐	Dropped due to being low-value; would not have required much additional work, but the time was thought better spent on polishing the core features.
	Editor	to set a date, time & duration of an event	I can inform readers of when the event occurs	☐	
	Reader	to view upcoming events in a chronological manner	I can be informed of upcoming events	☐	
	Authenticated reader	to register interest in an event using my email	I can be reminded when it starts	☐	
	Authenticated reader	to unsubscribe from e-mail reminders about events	I can stop receiving notifications about events	☐	
	Editor	to set a featured external link for an event	readers can clearly know where it takes place	☐	
	Editor	to view who registered interest in an event	I can see how much interest there is in it	☐	
Site management	Admin	to set the logo image of the site	I can set the site's branding image	☑	
	Admin	to set the name of the site	web crawlers can index the site better	☑	
	Admin	to set the favicon of the site	I can add flair to the site	☑	
	Admin	to view a log of editor and reader activity	I can stay up to date with staff activity and comments posted	☐	Was deemed low-value; not included in planning.
	Admin	customize the site's navigation bar	I can organize navigation to categories and articles	☑	
	Admin	to add external links to the navigation bar	readers can easily navigate to other related websites	☑	
	Editor	to rename categories	I can set a more fitting name	☑	
	Admin	add a sidebar on the home page	I can insert a description of the site or related links	☑	
	Admin	to insert a Twitter/BlueSky feed in the sidebar	I can feature social media accounts related to the site	☐	Was deemed low-value; not included in planning.
	Admin	to set the language of the static interfaces and labels of the site	the static text on the site is consistent with the language of articles	☐	Was deemed low-value; not included in planning.
	Admin	to set which social media platforms articles can be shared to	I can offer a more coherent experience for the site's demographic	☑	

Figure 2: Table of user stories grouped by epic, along with which ones were fulfilled by the end of the project.

3.2.2 Functional requirements

The functional requirements define what the system should do in terms of features and behaviour in more depth than user stories, specifying technical details to make it clearer what the implementation requirements are.

These were derived from the user stories and follow the common "**{User role}** should be able to..." format. A single user story can correspond to multiple requirements, especially in cases where the user's objective from the story can be accomplished in multiple ways that demand separate implementations.

The complete list of functional requirements is available in Annex 10.10.1.

3.2.3 Non-functional requirements

Non-functional requirements define qualities of the system that are desirable but not strictly tied to its functionality; they concern factors such as performance, interoperability or technology constraints[1]. They tend to affect implementation decisions throughout the whole project, and thus are important to keep in mind during development to work towards a robust product.

The following non-functional requirements were set for the project:

- Article pages should be crawlable and parsable by search engines for SEO optimization.
- Pages and interfaces should be navigable using the keyboard.
- Article content should be saved in a plain text format for interoperability with external editing tools.
- The interfaces of the control panel and pre-set text on the site should be localizable.
- The control panel and APIs should employ proven security measures for access control.
- The CMS should be designed with extendability in mind, to make it easy to add additional features in the future.
- The CMS should support automated testing.

Ultimately, not all of these were met; some were deemed very time-consuming and low-value, and thus were left as future work and not included in the planning, as will be detailed in Section 9 ≪Future work≫.

3.3 Use cases

Use cases are the third artifact the project employed during planning, listing the specific actions of users to accomplish goals from the functional requirements.

The relations between user roles, use cases and their flows were visualized with a use case diagram, providing a high-level overview of the program's functionality(Fig. 3):

Figure 3: The project's use case diagram.

Intuitively, one might consider making the admin role inherit from editor, to allow admins to also manage content itself (such as the articles). Although conceptually valid, it was decided not to allow it under the consideration that "super-admin" roles should not have any physical presence on the site - the readers should not know which accounts are admins, and their accounts should be used exclusively for site management to nudge users towards better account practices and separation of concerns.

Since most actions are stand-alone/independent of each other, there aren't many use cases with an *"extends"* or *"includes"* relation. The "edit article" use case is an exception, as the process of writing articles can involve many optional flows elaborate enough to warrant their own use cases - such as the addition/edition of the various semantic blocks in the article (images, code blocks, footnotes, etc.).

After the creation of the diagram, formal scenario scripts were written for the complex use cases, modelling the interaction flows between the users and the system to guide the implementation and understand the functional requirements in their context.

These use case scenarios were written using a systematic structure, defining their actors (user roles), their goal, system preconditions, the trigger/motivation that begins it and a script of the scenario. Care was taken to minimize mentions of the user interface so as not to condition its design at this point.

A list of the use case scenarios and additional considerations is available in Annex 10.10.2.

3.4 Database design

The project requires persistence to store the articles, user accounts and site configuration, making a database a necessity.

The structure requirements of the database were designed using an entity-relation (ER) diagram to organize the data into entities with relationships between them, giving an SQL-oriented overview of the data, which would later be used as a reference for the implementation of the database in Section 5.2. The entities and relationships were derived from the requirements, producing the following diagram(Fig. 4):

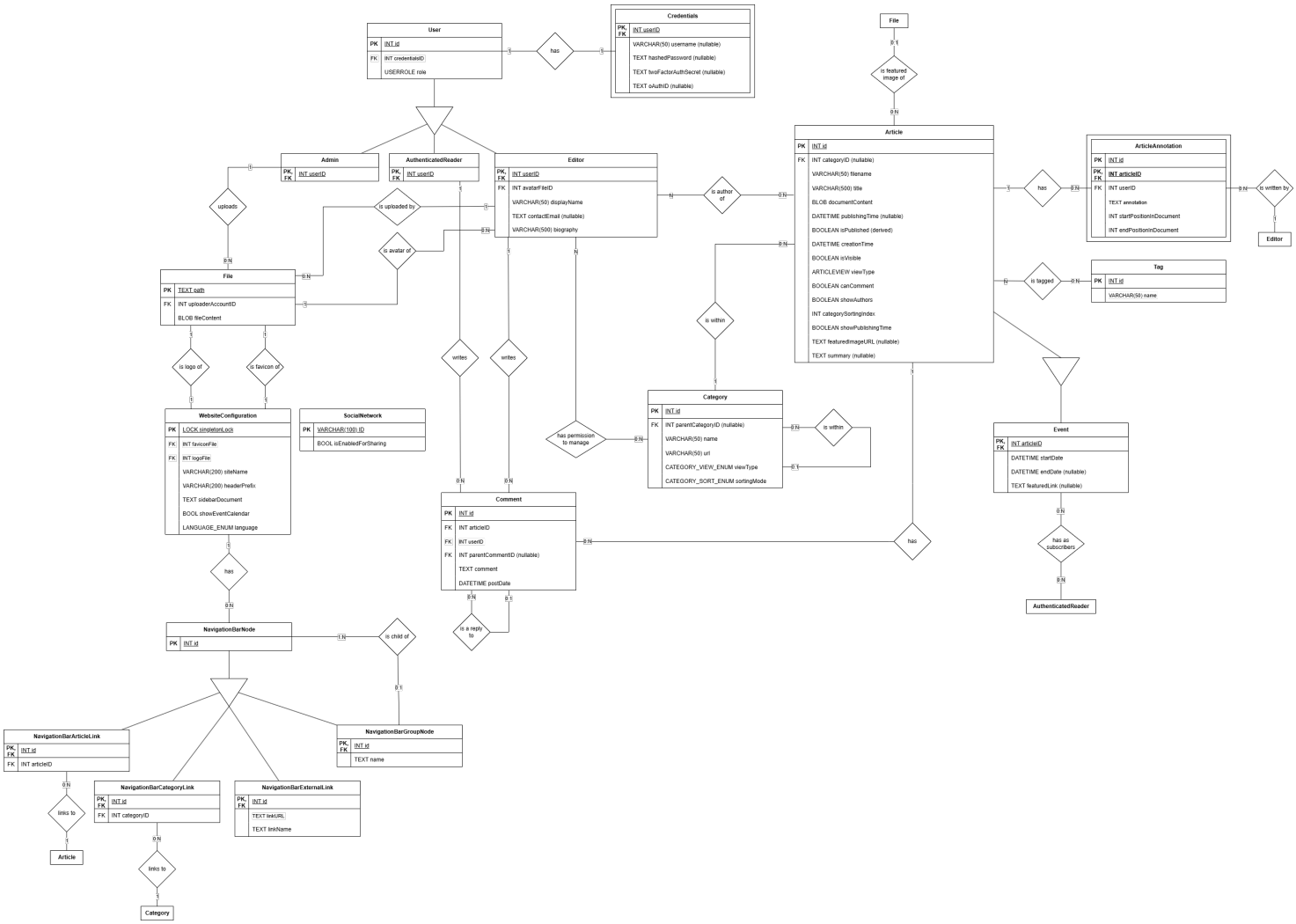


Figure 4: Entity-relation diagram of the project's database. *Certain entities were duplicated and shown in a collapsed manner to improve the readability of remote relationships.*

Three types of user accounts exist, with two corresponding to site staff user roles (*Admin* & *Editor*), and the *AuthenticatedReader* entity corresponding to accounts of readers that have logged into the site. Accounts are modelled via inheritance to be able to speak of an account in the general sense, simplifying relationships and operations that can involve any account type.

It was chosen not to store any personal information for the *AuthenticatedReader* entity, so as to avoid dealing with data privacy concerns for the time being. Although admin and authenticated reader accounts have no additional fields, a separate entity for them was still created to simplify database extensions, should the need to store new fields specific to them appear in the future.

The user credentials are stored as a separate entity with a 1 — 1 relation as they are a separate concern from the regular user data, and seldom relevant when dealing with user-related queries. This entity's fields accommodate the various authentication methods: username and password, 2FA, or third-party SSO. An account may support multiple of these log-in schemes; the fields for the ones not in use by an account would be `null`.

Articles are the largest entity in the diagram, though mostly consisting of metadata fields derived from the requirements. The article entity would also store the text content of the article itself or some form of resolving it; the details of this were decided during implementation instead, as the purpose of the diagram was only to outline the data needs.

Articles always pertain to categories, which are meant to be used to create nestable directory-like structures for the site's article content; as such, a category may have a parent category that it belongs to. This is modelled using a self-referential relationship (*"Category is within Category"*), by giving the entity a foreign key referencing another category; a `null` value would represent a category at the root of the tree structure. Comments on articles, which may be replies to one another, are modelled similarly.

Articles may also have tags, which present a many-to-many relationship, as a tag may be present in multiple articles, each with multiple tags. Authorship of articles is another many-to-many relationship case, as an article may be written by multiple editors.

One of the requirements includes the ability for editors to annotate parts of articles to leave internal notes for collaboration; this is modelled with the *Annotation* entity, which aside from storing the note text and its author would also need to be able to denote the range of tokens it references within the article text.

Weak entities are denoted by an additional rectangle around their table; these are entities that shouldn't conceptually exist independently without other entities they have relationships to; a *Credentials* entity for example cannot exist without a corresponding *User*.

As stated in the requirements, the app should allow changing some site-wide settings and metadata in a user-friendly manner - meaning these changes should be possible through the application and not just by editing the code itself. This creates a need to store these settings externally; it was decided to centralize all data in the app within the database rather than relying on external configuration files, using the *WebsiteConfiguration* entity, of which there should only ever be one entry

in the database.

One of the configurability requirements is being able to customize the navigation bar of the site. A typical website navigation bar contains links to pages/content of a site, sometimes grouping them to display multiple related links together in dropdowns or similar UI elements. At the database level this can be modelled as another tree, where the nodes correspond to links to categories, external URLs or specific articles. The diagram uses a hierarchy to differentiate these different nodes, with a *NavigationBarGroupNode* entity to represent groups of links.

The *File* entity is intended to be used for images and files linked from articles, as well as branding assets (logo, favicon) of the site configuration, thus its entity has relations with the files involved.

Additional commentary on the database design and the diagram are available in Annex 10.4.

4 Technology

The project inherently involves web development due to the nature of blogs being hosted on the internet and accessed through browsers.

Given that the project requires users to be able to create content directly through the application and persist configuration options, it cannot be merely a static website[7]; a service architecture was required.

Simple web applications typically employ a multi-tier architecture, an extension of the client-server model where there is a distinction between the server application and its data persistence[22], such that there are at least 3 components:

- The **presentation layer**, which corresponds to the user interface. In web development, this tier is called the **frontend** and serves the HTML and assets to browser clients.
- The **logic layer**, which handles business logic and manages the application's data. Referred to as the **backend** in web development.
- The **data layer**, which is responsible for persistence and access to the application data. This is typically a **database**.

This approach simplifies maintenance, and allows flexibility in choosing different technologies for the layers based on the application's specific needs. Alternative architectures, such microservices[8], are more intended for large-scale applications and would've been an over-engineered choice for the project.

The following sections will cover the project's chosen technologies for each layer: the frontend, backend and database, based on which technologies were thought to best fit the requirements and objectives of the project.

[7] Website that serves HTML files directly, with no dynamic responses.

[8] Web application architecture where the application is divided into multiple services interacting with each other[24].

4.1 Backend

The backend service is the stack that manipulates the data of the application, abstracting this process from the user and frontend by offering an API as an entry point to its services.

Given that the vast majority of the requirements can be implemented with one-off requests and do not need the client to maintain a constant connection to the backend, a REST[9] API was ideal for the project.
REST is one of the most common and reliable patterns for web application APIs as it is simple to design and implement due to being stateless, consisting of sending individual HTTP requests to fetch or modify data on the backend[11].
It also allows taking advantage of HTTP caching mechanisms to reduce latency and server processing time, which is particularly useful in the case of a read-heavy application such as a blog, where most data rarely changes and thus can be cached.

The following table gives an overview of the technology used for the project's backend(Fig. 1), which will be detailed in the coming paragraphs.

Concern	Solution chosen	Motives
API endpoints	FastAPI	REST-oriented, concise syntax
Data validation	Pydantic	Simplifies API error handling
Object-relational mapping	SQLAlchemy	Well-established, supports complex queries
Authentication	JSON Web Tokens	Stateless, easy to implement
Article search & indexing	Elasticsearch	Flexible search queries, prior familiarity

Table 1: Overview of backend technologies chosen.

REST APIs are typically implemented using scriptable web servers (SGIs). The **FastAPI**[10] Python framework was chosen for the implementation of the API, being a robust RESTful framework with emphasis on reducing boilerplate code and verbosity. FastAPI is built on top of the highly performant asynchronous Starlette[11] HTTP middleware, and integrates numerous libraries to make REST development convenient, such as schema validation, dependency injection and automatic documentation generation.

Pytest[12] was used as the testing framework for the backend for its simplicity. Tests with it are quick to write, with minimal boilerplate and suitable for integration testing. Plugins for it also support measuring test coverage[13], a necessity for verifying reliability of any backend application.

[9] Representational State Transfer: software architecture pattern for stateless web-based applications.

[10] **FastAPI**: Python framework for building ASGI applications and APIs.

[11] Starlette: lightweight ASGI server.

[12] **Pytest**: functional testing framework for Python.

[13] Test coverage: metric for the amount of executable code lines that are ran by automated testing.

4.2 Frontend

The frontend is the service that generates the HTML view for the user's browser and handles interactions with it by making requests to the backend's API.

One of the project's main objectives is to leverage advancements in web development to improve the user experience of blogging, namely by taking advantage of the complex rich content editing user experience (UX) that emerging frameworks made possible, which existing blogging solutions are neglecting to adopt.

Another important requirement is having an admin interface for site staff to manage the site content, which would also benefit from being a dynamic web page. Such interfaces are now often implemented as Single-Page Applications (SPAs), which use these JavaScript frameworks to update the view in response to user interaction rather than redirecting to new pages[9].

JavaScript-based web frameworks are thus ideal for the project's needs of implementing an admin interface and a highly-dynamic rich text editor, as they allow dynamic pages and "web apps" to be implemented more efficiently than with template-based web servers or regular JavaScript.

The following table shows an overview of the frontend technologies chosen for the project(Fig. 2), which will be detailed afterwards.

Concern	Framework/library	Motives
User interface	Vue	MVVM-oriented, component-based
Development framework	Nuxt	Hybrid rendering (both client-side & server-side)
Styling	daisyUI	Semantic CSS classes, customizable theming
Scripting language	TypeScript	Strict typing, strong IDE support
Form UIs	Nuxt UI	Good accessibility out of the box
Rich text editing	ProseMirror	Barebones framework, highly extendable
Markdown parsing	MarkdownIt	Plugin system, support for common syntax extensions
API querying & caching	tanstack-query	Convenient error-handling and response caching

Table 2: Overview of frontend technologies and main reasons for them being chosen.

Vue[14] was chosen as the JavaScript framework for the project, as it is oriented towards a clear MVVM pattern[15], organizing UI elements into stateful *components* that can be reused and made dynamic with JavaScript. Though it's a relatively new framework, Vue already has a mature ecosystem and was among the 3 most-used web UI frameworks among developers in 2024[18].

Nuxt[16] was chosen as the framework for working with Vue. Nuxt is a comprehensive Vue framework that offers vast workflow and quality-of-life improvements to Vue development, handling many repetitive tasks and reducing boilerplate code automatically during the build process, such as auto-importing used components and libraries. Aside from a build pipeline, it offers numerous additional libraries of

[14] **Vue**: JavaScript user interface framework

[15] Model-view-viewmodel: architecture for graphical applications that relies on the observer pattern to decouple UI from business logic.

[16] **Nuxt**: Vue framework and build toolchain supporting hybrid rendering.

its own to speed up UI development, such as robust and flexible Vue implementations of common form elements (input fields, dropdowns, modals, etc.) with their Nuxt UI[17] library.

The most relevant advantage of Nuxt for the project, however, is its **hybrid rendering** option that allows easily mixing client-side and server-side rendering for the app, combining the benefits of the performance of server-rendered sites and the interactivity of SPAs[18][**21**].

This is an important benefit, as the different requirements of the project are best implemented using different web app approaches. The published blog site where visitors come to read articles would be best implemented using server-side rendering to have content load as soon as possible and allow it to be indexed by web crawlers for SEO[19], which is unreliable in SPA due to requiring additional requests to load a page's content[**25**].
Usually these SEO shortcomings are not a concern for the typical usage of SPA as they're not used for content pages, however, in the case of a blog, it is essential for the blog content to be parsable by search engines so that it may appear correctly in search results.

On the other hand, the control panel requires high interactivity, which is best achieved using the SPA approach(*See annex* 10.1.8).
Nuxt allows both approaches to be used in a single project instead of needing to implement these two parts of the project using two separate frameworks, which greatly simplifies the development of the project's frontend.

Regarding style and theming, the daisyUI[20] library was chosen as it has clean, material-design styling for many common web elements, saving a lot of development time that would otherwise be spent on visual design aspects. It also provides multiple decent themes and colour palettes along with easy creation of custom ones, which is helpful for the requirement of the site's theming being customizable. The site's visual design was complemented using icons from Icones[21], a repository of Apache and MIT-licensed icon libraries.

The project also uses TypeScript[22], a superset of JavaScript that implements strict typing to reduce mistakes during development and improve the maintainability of the codebase. The major advantage of TypeScript comes from IDE integrations; auto-completion, linting and diagnostics help detect coding mistakes as soon as they happen rather than once the app is tested, improving development efficiency substantially, as well as code readability, as there are no ambiguities on the types of parameters or fields of an object.

[17] Nuxt UI: Vue form & UI component library
[18] Single-page application: web app architecture where an entire site is accessed from a single HTML page, using JavaScript to update it in lieu of traditional navigation.
[19] Search engine optimization: practices that improve a site's ranking in search engines.
[20] daisyUI: Tailwind-based design system
[21] Icones: repository of freely-usable icons for user interfaces.
[22] TypeScript: superset of JavaScript with strict typing.

For the implementation of the WYSIWYG[23] article editor, the **ProseMirror**[24] library was used, which offers a framework for creating rich text editors.
ProseMirror handles parsing text documents into semantic node trees and offers APIs and callbacks to implement custom document editing operations and HTML rendering, making it very flexible. By itself it offers little to no functionality, only the foundations to build any kind of document editor; it is thus very suitable for the variety of semantic blocks the project wants to support within articles, and its ambitions to build a blogging editor with specialized features not present in existing solutions or generic text editor libraries.

Additional notes on technology decisions are available in Annex 10.3.

4.3 Database

For persistence of the site's content and configuration, **PostgreSQL**[25] was chosen as the database as it is a highly reliable and well-established open-source SQL database.

NoSQL databases were initially considered, however during the database design phase in Section 3.4 it was quickly noted the data is a lot more referential than was initially thought, and that the strong integrity checks of SQL databases would be more desirable.

Given that the entities are few and structured without many variants or foreseen need to add new attributes often, the advantages of loosely-structured document-based NoSQL databases were thought not very relevant to the project, and the second-class relation support they have would've hindered development and performance.

Considering the project is intended to be self-hosted and not a centralized platform, horizontal scaling[26], a major advantage of most NoSQL databases, was also not a strict necessity.

Additionally, there are no entities in the database that would be modified very frequently, as write operations are mostly done by the small group of site staff; as such, it was not necessary to add additional layers of data storage/caching such as a Redis database to have faster read/write access to specific resources. For this same reason, an event queue to decouple database writes from the rest of the application was also deemed unnecessary.

The project does however use Elasticsearch[27] to index content of articles and allow them to be searched by text or metadata like tags.

During development, some of these decisions turned out to have shortcomings, which are detailed in the project's conclusions in Section 8.

[23] "What you see is what you get": content editing software paradigm where the content being edited is already displayed in a manner very similar to its appearance when finished/published for consumption. An example would be Microsoft Word.

[24] **ProseMirror**: framework for building web-based rich text editors.

[25] **PostgreSQL**: Open-source relational database.

[26] Instantiating multiple servers and databases to handle mass requests to a single system.

[27] Elasticsearch: data store solution usable as an external index and document search engine.

5 Implementation and development

Development began on the second week of the semester after the requirements gathering and selection of frameworks and tools.

For any feature, the order of work to implement it was generally in order of dependencies:

1. Adding/updating database tables.
2. Implementing backend CRUD functions and API routes.
3. Adding API tests.
4. Implementing the corresponding UI elements in the frontend.
5. Integrating the frontend UI with the API.

Though it would've been possible to implement the parts in a different order by mocking dependencies, that practice is mainly useful in teams where there's simultaneous work on frontend and backend; for a 1-person project it was considered time-inefficient.

The following sections go into detail on the design and implementation of the project's services, as well as the patterns and conventions used.

5.1 Backend

The backend was implemented using **FastAPI** and **SQLAlchemy**[28], handling the API for the project as well as database management. The goal of the design of the API was to make an idiomatic REST API - one that doesn't just follow the functional principles, but is also intuitive and convenient for the clients (the frontend) to use.

This meant planning and designing the API with the frontend's usage of it in mind, so that during the frontend's development it wouldn't be necessary to go back and make adjustments to the API, saving development time.

The backend is split into 2 main systems:

- **CRUD**[29] **function libraries**, which manipulate database objects.
- The **API routes**, which define the REST endpoints and handle requests by invoking the CRUD functions, as well as application-level logic like authentication.

The folder structure organizes scripts by their purpose and "layer" within the application. In terms of dependencies, the package diagram looks as following(Fig. 5):

[28] **SQLAlchemy**: Python Object-Relational Mapping library.

[29] "Create, Read, Update, Delete": functions that manage objects for persistent storage applications like databases.

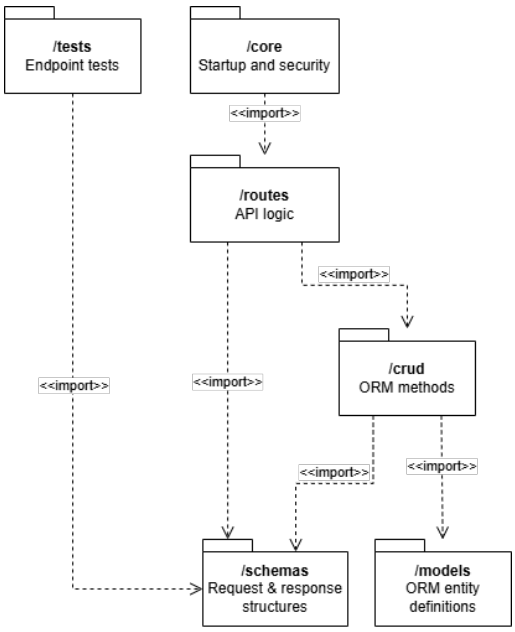


Figure 5: Backend package diagram, denoting dependencies between the modules.

The project makes extensive use of the built-in Python type annotations to provide auto-completion and type-checking within IDEs, to speed up development and catch mistakes when they occur.

Additional information on the project's structure and general code style is available in Annex 10.8. The following sections cover the backend's modules and how they achieve the API's goals and separation of concerns between them.

5.1.1 Routes

The routes are the URLs that the server accepts for API requests. The project follows REST conventions, where the URLs are intuitively named after the database entities they work with, accepting the common HTTP methods to perform operations with the entities (creating, reading, updating, deleting). For example, user accounts can be accessed by making requests to `/users/`, and a specific article may be accessed at `/articles/{category}/{article filename}`.

The following diagram shows an outline of the API's routes, categorized by their function(Fig. 6):

Figure 6: Diagram of the backend's API routes, indicating the endpoints and methods they accept. *A lock icon indicates endpoints that require authentication.*

The API was carefully designed to consider its usage in the frontend, so as to minimize oversights that would lead to needing to make adjustments to it later. These considerations include:

- String identifiers for resources are favoured over numeric ones, as they tend to be more intuitive and easier to map to URLs on the frontend (such as article pages).
- Routes that return multiple entities (such as fetching the articles of a category) support "limit" and "skip" query parameters, a standard REST pattern which

allows clients to implement pagination(*See annex 10.8.2*), optimizing bandwidth and server CPU time.

- Exceptions while handling requests are turned into appropriate HTTP exception codes, with descriptive messages.

The project makes use of Swagger[30] to offer a web interface to view the documentation of all endpoints, generated from the *pydoc*[31] strings and type annotations in the source code, which was useful as a reference while developing the frontend. This documentation is only present while running the project in a development environment.

5.1.2 Schemas

Endpoints that receive data from the client (to create or update entities) as well as responses sent by the API require the data to be in JSON with a specific structure, defined by schemas.

The project uses the Pydantic[32] library to define these schemas as Python classes, which handle validating JSON into them, ensuring requests and responses always adhere to the expected format and removing the need to do this explicitly.

This integrates well with FastAPI; requests to create or update entities whose data does not adhere to input schemas immediately receive the `HTTP 422 Validation error` response, which includes an automatically-generated detail message specifying to the client which fields failed validation and why, making it easy to debug mistakes when using the API.

For every entity usable in the API there are multiple schema classes each corresponding to the different operations (create, read, update)(*See annex 10.8.8*). The schema classes are pure data structures and do not depend on any classes/types or functions from the application, allowing them to also be used in the CRUD libraries described in the following section.

5.1.3 CRUD

The API endpoints work by invoking functions from the CRUD libraries, which manage the database tables. A separate library exists for each major table/entity in the database.

The CRUD functions perform only database operations and their parameter validation only considers database-level coherence checks, such as preventing creating multiple users with the same username. Any application-level validations such as permission and authority-based restrictions are applied in the API route logic instead to keep the CRUD methods flexible.

The CRUD methods make sure to only commit transactions to the database at the end of all operations. In the case of any exception, the transaction is rolled back, and the exception is propagated to the original caller (the API)(*See annex 10.8.4*).

[30] Swagger: documentation generator for REST APIs.
[31] Python's built-in type annotation system.
[32] Pydantic: Python data validation library.

CRUD operations always return the database ORM objects for ease of further operations. The API however needs to convert these objects into the corresponding Pydantic response schema when "returning" them to the request client. To ease this process, the CRUD scripts have utility methods to convert ORM entities to their schema form using reflection-like methods from SQLAlchemy and Pydantic to map ORM fields to the corresponding schema ones(*See annex 10.8.5*).

Creation and update request schemas are also used as parameters of the CRUD create & update methods respectively to reduce the amount of parameters these functions take. This makes it convenient to expand them with new fields, as function calls do not need to be refactored to add/remove parameters to the signature, they only need to be adjusted to use the new fields within.

The following sections will cover the project-specific functionality of the API that these CRUD libraries and routes cover.

5.1.4 Article management

Articles and categories are managed via 2 endpoints: `/categories/` and `/articles/`, which allow site staff to create and modify them to manage the content of the site and organize it.

Since both entities store a path that is meant to map them to URLs on the published site, the API routes also intuitively identify them by this path using FastAPI catch-all routes(*See annex 10.8.9*); for example, an article named "Cake" within a "Cooking recipes" category may be retrieved with `GET /articles/cooking_recipes/cake`.

Normally, resolving such paths to their entity would involve traversing the "tree" relationships using each part of the URL and validating it. To optimize retrieving articles and categories by these paths, their database entities cache this full path in a column that uses a hash index to allow resolving them with minimal overhead.

Articles may have multiple text tags to further organize them thematically. Tags in the API are referred by name but are tracked by a numeric ID in the database, which allows for renaming tags without needing to remove & reapply them to every article that uses them.

The `/comments/` endpoint allows fetching and posting comments to articles. This endpoint was made separate from the articles, as comments are not always relevant to requests relating to articles, and would result in a lot of irrelevant data being sent if they were present in all article requests. The post dates of comments are set on the server to prevent spoofing them.

5.1.5 Searching

The `/search/` endpoint allows searching articles by their text content and metadata like tags and authors. This is implemented using Elasticsearch, by indexing the relevant properties of the articles into it.

The main search endpoint supports 3 kinds of searches via query parameters, which may be combined:

- **text**: searches content, titles and summaries of articles using Elasticsearch's "phrase-prefix" search type to allow partial matches of keywords.
- **author**: filters results to articles written by a specific editor.
- **tag**: filters results to articles that contain the tag (as an exact keyword match)

The indexes are configured to tokenize and standardize article content to make searching more efficient and robust; the text is split into keywords with stop words removed, and the keywords are also stubbed/stemmed to disregard verbal tenses, plurality, etc.

The article API routes are responsible for indexing and updating an article's document in Elasticsearch when requests to create/patch articles are received. Since this is a slow process and the API queries that trigger it don't need it to be resolved to return their response, these updates are implemented using FastAPI's "background tasks", running asynchronously from the request.

In addition, a "latest articles" endpoint exists, which aggregates and sorts all publicly-visible articles across all categories, intended to be used for the site's homepage.

5.1.6 File and configuration management

The backend supports uploading arbitrary files to a `/files/` route, primarily intended to be used to embed images in articles or add download links to other files. Users give files a custom URL to identify them, and the account that uploaded them is tracked for administrative purposes.

Since the file data must be sent inside a regular JSON request along metadata like filename, it must be encoded to base 64 first to ensure there are no problematic characters in the JSON.

For efficiency, fetching files is split into two endpoints: one to retrieve the file metadata (such as the uploader's info) with a usual JSON response, and another that returns the raw file data to avoid the overhead of encoding & decoding its binary data to & from JSON.

An endpoint to retrieve all files exists, but is limited to site staff to prevent leaking files that are not meant to be public, such as images/attachments used in articles that haven't been published yet.

The `/config/` routes are used to manage and retrieve site-wide configuration settings, such as its branding assets, metadata and navigation bar scheme. Only admins can modify the configuration.

The site's logo and favicon assets are retrievable through a special endpoint to allow them to be cached separately from the rest of the configuration.

Though the enabled social networks are stored in a separate table, the API simply allows passing a list of network IDs to toggle them for convenience, rather than having an entity-specific endpoint for them. There is no API support to register more social networks, as there wouldn't be any benefit to being able to do so from the API, since adding proper support for more networks would require the frontend code to be updated with the art & integration work either way.

5.1.7 Authentication and security

The project's user roles define authority requirements; only the admin and editor roles are supposed to be able to manage the site's content, as such, for both functional and security reasons it's necessary to limit the access of certain endpoints based on the role of the user making the request.

The endpoints a user has access to is determined by the role of the account they're logged into, which is done via a POST request to a login endpoint providing the username and password. Passwords in the database are hashed using the bcrypt algorithm[6] for security in case of leaks.

The project uses JSON Web Tokens for authentication, consisting of an HS256[33]-signed JSON document that the client passes in every request to prove their identity. This is a common authentication scheme which fits into REST & HTTP being stateless, as it doesn't require implementing a concept of a session on the backend[20].

The token is generated upon sending a `POST /login/` request. JWT tokens are signed such that to verify the identity of their owner it's only necessary to decrypt them and verify the integrity (expiration date and issuer); there is no need to persist them in the database.

The client stores the token as an expirable cookie and includes it in all subsequent requests to authenticate themselves. Explicitly logging out (via a logout endpoint) invalidates previous tokens by setting a timestamp in the user's database entry, which is checked to reject tokens issued before that date.

The project uses the payload part of the JWT to store the username of the account associated with the token as well as its expiration date; this is convenient for the client/frontend to know which account it is logged into to fetch its data more conveniently, as well as to proactively know when the token will expire.

At the implementation level, the project uses FastAPI's dependency injection features to automatically resolve JWT tokens in requested headers to their database user, making it easily available in any endpoint. Endpoints that require authority (such as creating or modifying articles) first check if the token is present and valid; if it isn't, a `401 Unauthorized` response is sent back to the client immediately.

Some endpoints to fetch data also support optional authentication to have them include otherwise private data; for example, the `GET /categories/` endpoint only returns published articles by default, but will include hidden ones as well if the request is made by a staff account.

The implementation and alternatives considered are covered in detail in Annex 10.8.6.

[33] HS256: common JWT hashing algorithm which uses SHA-2 hashing function and RSA signature.

5.1.8 Tests

The pytest[34] framework is used for testing, where tests are written as functions that call the API methods and assert the responses received.

Tests exist for each API endpoint, checking both regular usage and invalid inputs and edge cases, asserting the HTTP response status codes and content of the JSONs received.

Pytest's fixtures system is used to set up starting scenarios before each test, which is used to pre-create the test's dependencies, such as populating the database with users and articles that the tests need. The tests thus follow the standard "arrange-act-assert" pattern:

- **Arrange**: setup test preconditions, creating database entities that are needed. Mostly handled by fixtures, as the preconditions tend to be similar across the tests of each endpoint.
- **Act**: perform requests to the API.
- **Assert**: validate responses received.

To keep tests self-contained, another fixture exists that runs after any test and deletes all database entities created during it. This ensures each test starts with a clean slate and does not cause any side effects that may cause other tests to fail or complicate their design. For example, asserting the amount of created entities in a test becomes verbose if you cannot assume the database was previously empty. The tests are also configured to use a different database, so that running the tests locally does not override the database used for local development.

Requests in tests are sent using a wrapper class for an HTTP client that also accepts instances of the schema classes (from Section 5.1.2) and automatically converts/parses them to JSON. This makes the "act" and "arrange" steps more convenient to write, as well as easier to maintain, as IDE refactoring operations on the schema classes will also update their uses in tests. Tests that intentionally send malformed data are of course an exception to this and use raw JSON instead.

It was chosen not to write tests for the CRUD methods specifically, as the API methods already cover their execution and correct API responses depend on the CRUD methods working properly. Since CRUD exceptions and their messages are propagated by the API, the flows that lead to them can also be tested.

[34] pytest: functional testing framework for Python.

5.2 Database

Configuring the PostgreSQL database and creating all the entities was done via SQLAlchemy's declarative APIs from within the backend code.

SQLAlchemy models tables as Python classes, using special types to declare columns and their properties (such as nullability, primary/foreign key status, etc.) in a database-agnostic manner. Upon initializing the backend, these table classes are parsed, and SQLAlchemy handles generating the SQL statements to create the tables based on the database used.

Creating the database this way has the minor advantage of making it easier to migrate the program to a different SQL database if it were necessary, as there would be no need to adjust the table declarations to consider differences in their SQL dialects.

The tables themselves were designed using the ER diagram shown previously in Section 4 as reference.

Entity inheritance (such as the *User* hierarchy) was modelled by creating tables for both the parent and child entities, and creating a one-to-one "parent-child" relationship between them. This approach removes concerns of duplicating data across the various "derived" entities and integrates better with ORM libraries, however it does incur a small overhead of needing JOINs to traverse the hierarchy. These child-parent relations are configured to automatically delete the "child" entities when their parent is deleted (via the `delete-orphan` cascade), simplifying delete operations and ensuring no orphan entities linger. The responsibility of maintaining integrity of the tree-like relations (such as avoiding circular relationships) falls to the backend code.

Multiple relationships in the ER diagram such as *"Editor is author of Article"* had N:N cardinality. This required creating a separate table for the relationship itself, whose rows are pairs of foreign keys of the two entities to handle the multiplicity of the relationship.

The *Configuration* entity, which holds site-wide settings, was implemented using a singleton table pattern by having the CRUD methods forbid creating additional rows, as there should only ever be 1 configuration entry.

This configuration table also stores the configuration of the site's navigation bar. The navigation bar as a whole is very much a semi-structured entity; it can be seen as a tree of different kinds of nodes linking articles, categories, arbitrary URLs, etc. This kind of structure is tedious to model and manage using SQL due to requiring many joins to reconstruct the whole tree, which is ultimately what the client (frontend) will want.

For this reason, it was decided to store the navigation bar using the Postgres JSON column type instead, using nested JSON objects to represent the parent-child relationships of the link nodes.

Although using a JSON goes against SQL practices, the benefit of being able to fetch the whole navigation bar structure in a single query was thought to outweigh the disadvantages. This JSON refers to articles & categories by their numeric IDs; these never change, thus it's only necessary to revalidate the document when articles or categories are deleted, which is an uncommon occurrence.

Configuring the social networks that sharing is enabled for is done via a `SocialNetwork` table with a simple boolean to toggle the feature for each network registered by the backend at startup.

For ease of prototyping the rest of the application, it was chosen to store the article documents and files directly in the database using the BLOB/binary type. In a serious usage scenario, it would be ideal to store these files externally using a cloud storage solution such as Amazon S3[35]; these BLOB columns would then be replaced by the uploaded file's identifier. Storing the files in the database is still preferrable over disk storage as it avoids dealing with OS-specific filesystem quirks, and centralizes the site's data to simplify backups.

Additional implementation notes are available in Annex 10.5.

5.3 Frontend

The implementation of the frontend intends to leverage Nuxt and the chosen libraries to create a good user experience that's consistent and coherent, while keeping its code concise and readable. Design-wise, this translates to minimizing boilerplate code, maintaining a clean MVVM separation of concerns, and reusing UI code as much as possible to have the site look and behave consistently.

The folder structure of the frontend source code is as follows(Fig. 7). Many of these root folders are dictated by Nuxt's framework, as they have special treatment during the build process.

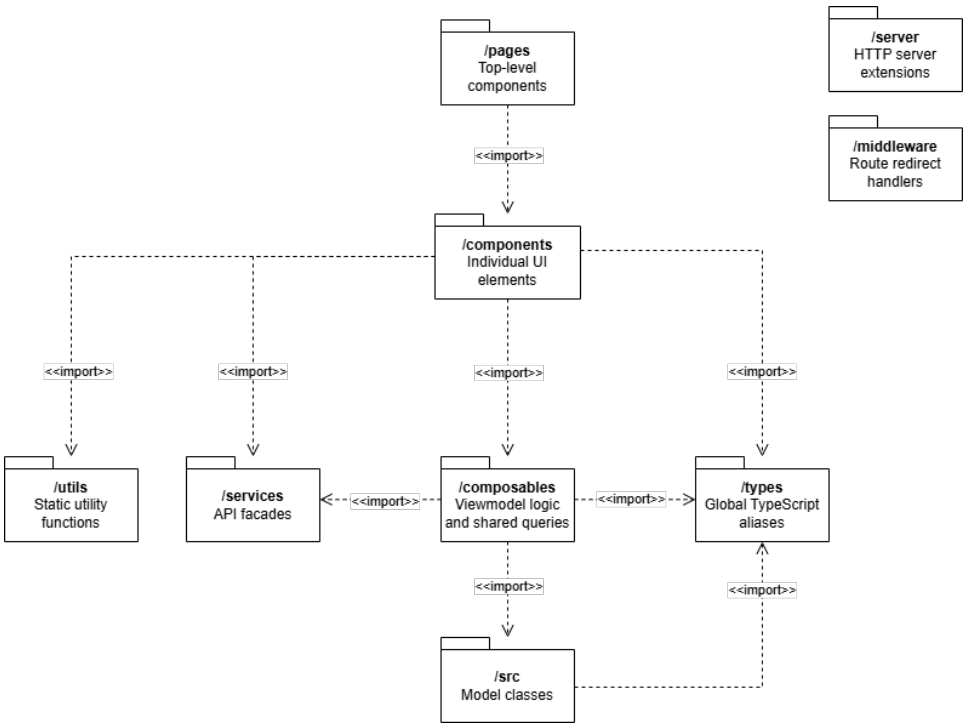


Figure 7: Package diagram of the frontend, denoting dependencies between the modules.

[35] Amazon S3: cloud storage solution, part of Amazon Web Services.

The following sections cover the various aspects of the frontend's design, implementation and the features that it integrates from the backend API. The article editor, though part of the frontend, will be covered in a separate section due to its complexity.

5.3.1 Visual design and sitemap

The layout of pages and important widgets was first designed using wireframe mock-ups with Balsamiq[36] to define their visual hierarchy, covering UI design aspects such as element order and positioning.

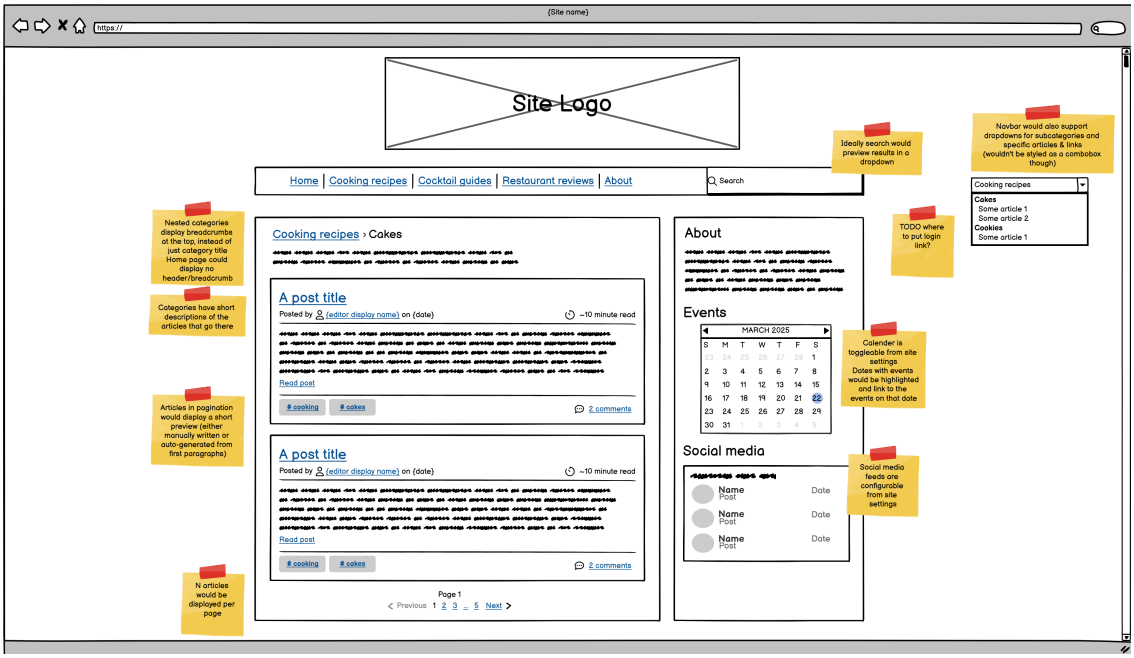


Figure 8: Mock-up of a category page and its article listings. Design decisions are annotated with "sticky notes".

These mockups were highly useful to implement the frontend efficiently, as having the layout already designed meant there was less trial-and-error needed in regard to structuring the site HTML.

The finished site only slightly deviates from the mock-ups, in situations where minor UX issues with the original design were identified after an element's initial implementation. A full list of the mock-ups produced is available in Annex 10.1.2.

The frontend site is split into two isolated parts: the user-facing **"published site"** where articles may be read, and the staff-only **"control panel"** where the site's content and configuration is managed.

The following sitemap diagram displays the major pages of the site and their navigation(Fig. 9). The actual sitemap of the published site is determined by the categories and articles created, and as such it may be structured freely by the site staff.

[36] Balsamiq: low-fidelity UI wireframing tool.

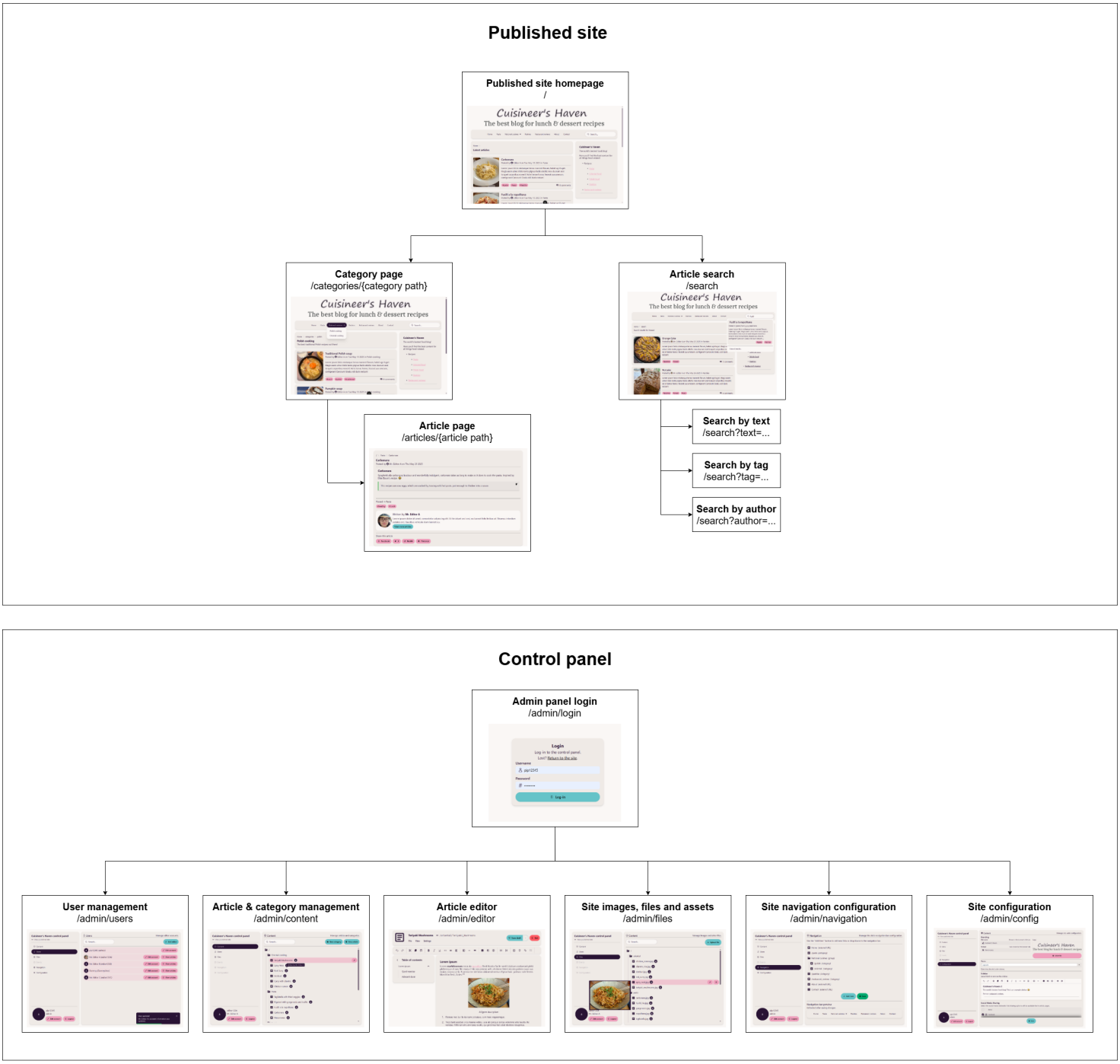


Figure 9: Sitemap diagram of the frontend site.

The following sections will cover the finished pages of the site and their implementation details, starting with the pages of the "published site".

It's worth noting that due to time constraints, it was decided early on to only target desktop devices - making all interfaces responsive was left as future work.

5.3.2 Published site

The "published site" is the part of the frontend that corresponds to the site that visitors (the "reader" role) see, which displays the articles in a blog-like website(Fig. 10). The screenshots shown use a set of mock data for demonstration purposes(*See annex* 10.8.1), showing an example of a cooking-themed blog.

Figure 10: Front page of an example published site, displaying the latest articles posted.

The homepage displays the latest articles from across all categories in the site; the user can browse articles from a specific topic through links to categories in a navigation bar. A sidebar is also present which may be used to add information about the site, links, or images.

The navigation bar also features a search bar for finding articles, which uses a "search-as-you-type" UX pattern(*See annex* 10.1.9): the top search results are previewed in the page as the user types in the field, featuring the article's summary and tags. Complete search results can be accessed in a dedicated page by submitting the search(*See annex figure* 39).

A category's articles can be browsed at the `/categories/{category path}` route, offering a paginated view where articles are displayed in a vertical layout using a "card" design pattern(*See annex figure* 36). The article cards display details to entice readers, showing the article's title, date & author, a summary of the article, a featured image, as well as its tags and comment count. The user may read the full article through the link in the title, or navigate to its comments section directly, useful to quickly read up on new comments of articles they already read in the past. The article card also displays the article's tags in a "badge" format, which may be clicked to search for articles with that same tag - convenient for exploring more

articles covering the same topics. Semantic tags are applied to parts of the card to make them more digestible to assistive technologies; `<figure>` is used for the cover image, `<summary>` for the summary, etc.

Each article has a corresponding page on the published site(Fig. 11), which displays its content, metadata and comments section. The articles themselves support various forms of formatting and rich content, which is covered in detail in the article editor's implementation in Section 5.4.

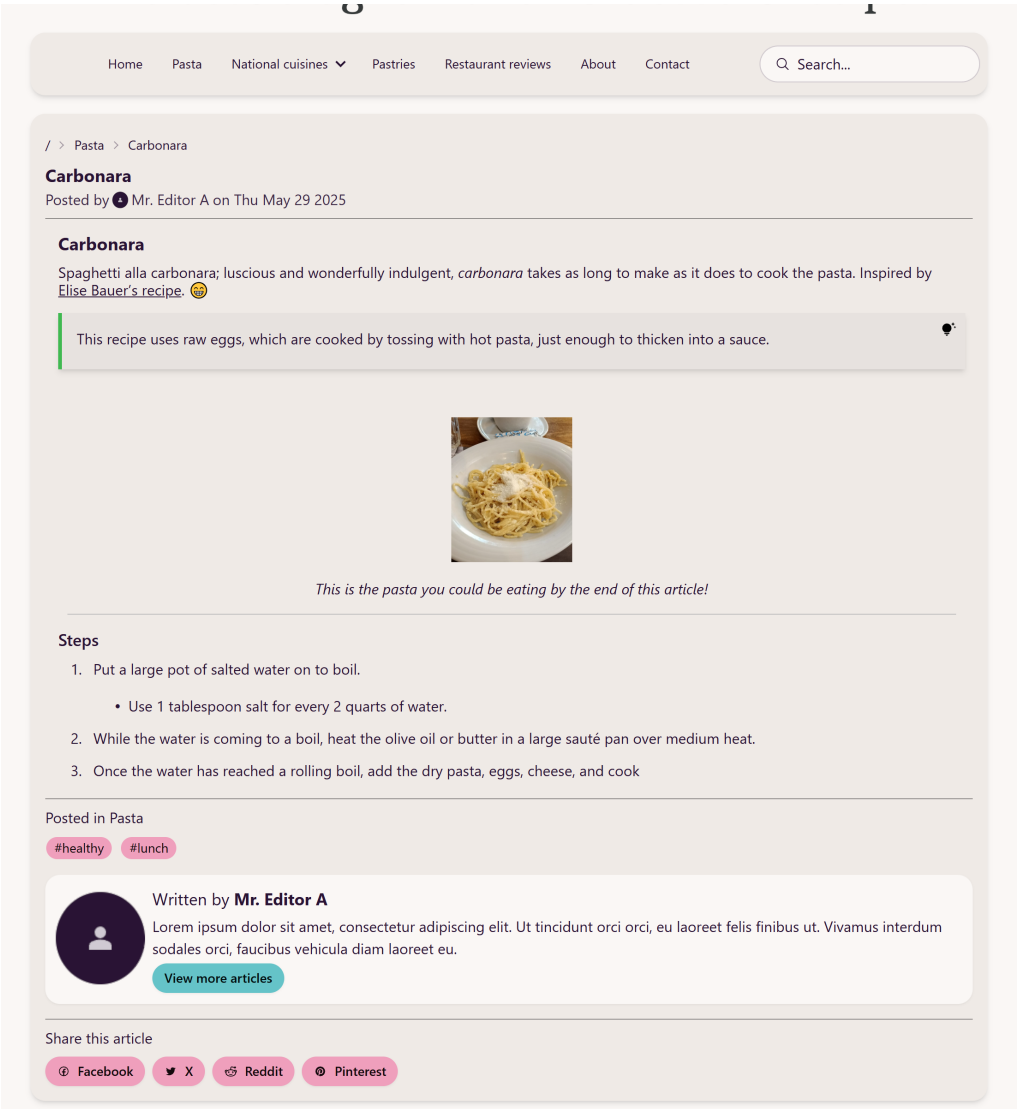


Figure 11: Page for a published article.

A card with the author's biography and buttons to share the article to social media networks are present at the end of articles; these make use of the vue3-social[37] library to handle opening a pop-up browser window pre-configured to share the article link to any of the social media networks that are enabled on the site.

[37] vue3-social: Vue library for invoking user flows to share content to common social media platforms.

Articles also support comments(*See annex figure 38*); site visitors may log in using their Google account to post comments on articles, using a reduced version of the project's text editor with only the relevant features (such as quoting parts of the article).

Comments may be replies of another, intended to be used by site staff to respond to questions or feedback, with the replies displaying as a collapsible tree. Comments of reader accounts are anonymous and display no personal information, and neither does the system store any user information to avoid dealing with data protection concerns(*See annex 10.1.13*).

5.3.2.1 Implementation details

The published site makes use of Nuxt's hybrid rendering to make content load faster for visitors while also offering the smooth navigation experience of an SPA. The main layout components of the site as well as the article themselves are rendered by the web server, so their HTML can be included in the very first request a visitor makes to the site. This also ensures that search engine crawlers can index the site's content, as it will be included in the HTML from the very start, instead of requiring JavaScript and extra requests to load.

Rendering an article page requires the frontend server to make an API call to fetch the article document and metadata from the backend; since articles are not modified frequently, the frontend caches each article fetched using tanstack-query to reduce API calls and render subsequent accesses to the pages faster.

Some navigation, such as switching between pages of search results, is instead done using client-side component updates, having the client call the API to fetch the next batch of articles. This improves page responsiveness as the user doesn't see the whole page reload, and it also reduces requests to the web server.

Since URLs for articles and categories are dynamic, the site's routing[38] must also be dynamic. Instead of pre-defining mappings of article & category URLs to the page components that display them, the project uses Nuxt catch-all routes to do so implicitly. The trailing components of the URL are used as parameters to have the components fetch the corresponding article(*See annex 10.1.11*).

Most pages have a similar basic layout (with a logo, navigation bar, sidebar, footer, etc.), as such these common elements are extracted to a single component to avoid code repetition, which uses Vue `slots`[16] to have specific pages only define the elements that do differ between them.

The components of the published site make use of the semantic sectioning HTML tags, such as `<nav>` or `<article>`. These tags serve to denote the nature of their content to assistive tools such as screen readers or navigation shortcuts.

The navigation bar is rendered based on the JSON navigation schema fetched from the API. As mentioned previously, the navigation schema supports different kinds of "nodes" to link to different kinds of content (categories, specific articles, external URLs). To support these variants cleanly, the special `<component>` tag is used to map node types to their components to avoid lengthy `if-else` chains in the navigation bar component.

[38] Vue routing: mapping of URLs to the root component that displays their page.

5.3.3 Control panel

The management of the site is done through an admin interface at the `/admin/` route, requiring the user to authenticate first with a login form(*See annex figure 25*). The panel allows viewing and editing the site's articles & categories, files, user accounts, and configuration(Fig. 12) through different tabs accessed from a sidebar.

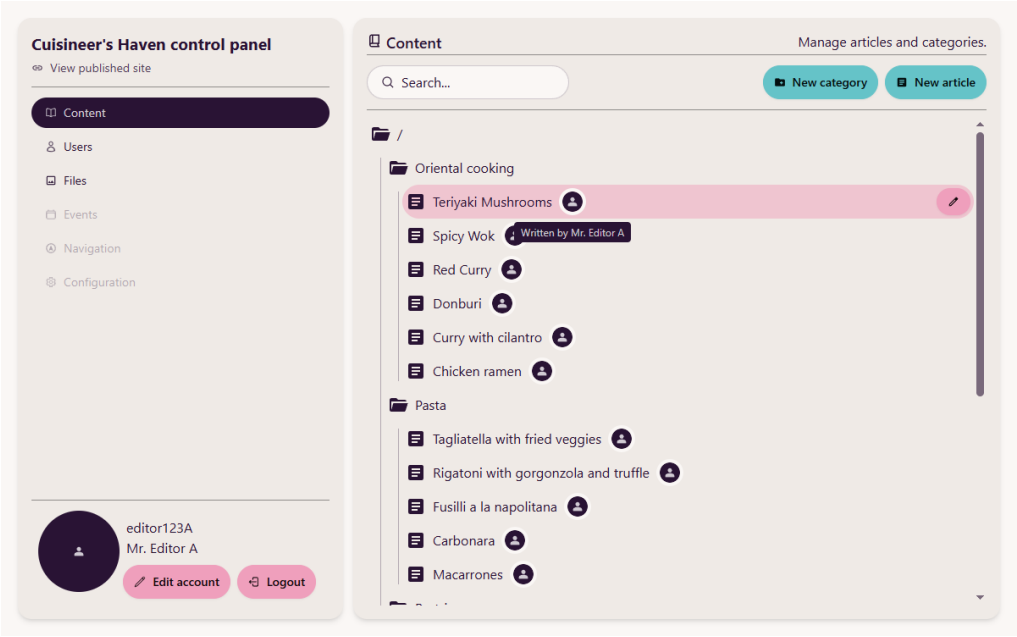


Figure 12: Article & category management page within the control panel.

The admin interface is an SPA, using only client-side rendering to make navigation smoother, as it is highly dynamic and initial load time is not a concern for its use case.

Categories and articles are displayed using a directory-like tree view, and may be searched to find specific items by name. This tree is rendered recursively using a flexible, general-purpose component to render such trees(*See annex 10.1.12*). Searching/filtering is performed client-side for better responsiveness, using only simple string matching.

Creation and editing operations are done through modal forms with a standardized layout(*See annex figure 31*). Requests to create or edit resources are handled using "mutations" from the tanstack-query[39] library to create *reactives*[40] for the state of the request. This allows the forms to easily display feedback like loading spinners while the requests are being resolved. Creating or editing resources re-fetches relevant data from the API to update the view; the data is otherwise cached when navigating between the tabs of the control panel.

[39] tanstack-query: cross-framework asynchronous state management library.
[40] Vue's system for observable variables[**15**].

5.3.3.1 Files tab

The Files tab allows managing uploaded files(*See annex figure 24*), displaying them in a tree view using the project's general-purpose tree component. These files are used for numerous purposes throughout the site, such as image embeds in articles, editor account avatars, or site branding assets.

Hovering over a file previews it in a tooltip and clicking it opens it in a new browser tab, and it's also possible to re-upload/replace files.

As mentioned in the backend implementation section, files are stored as blobs in the database. To make them viewable in the frontend, the project uses the Nuxt "server routes" functionality to extend its HTTP server; when URLs at `/files/` are accessed from the frontend, the web server fetches the corresponding file from the backend and sends back a plain HTTP response with the file data instead of rendering any Vue component.

5.3.3.2 Configuration tab

The configuration tab concerns global site settings, such as its branding assets, colour theme, and which social networks readers can share articles to(Fig. 13).

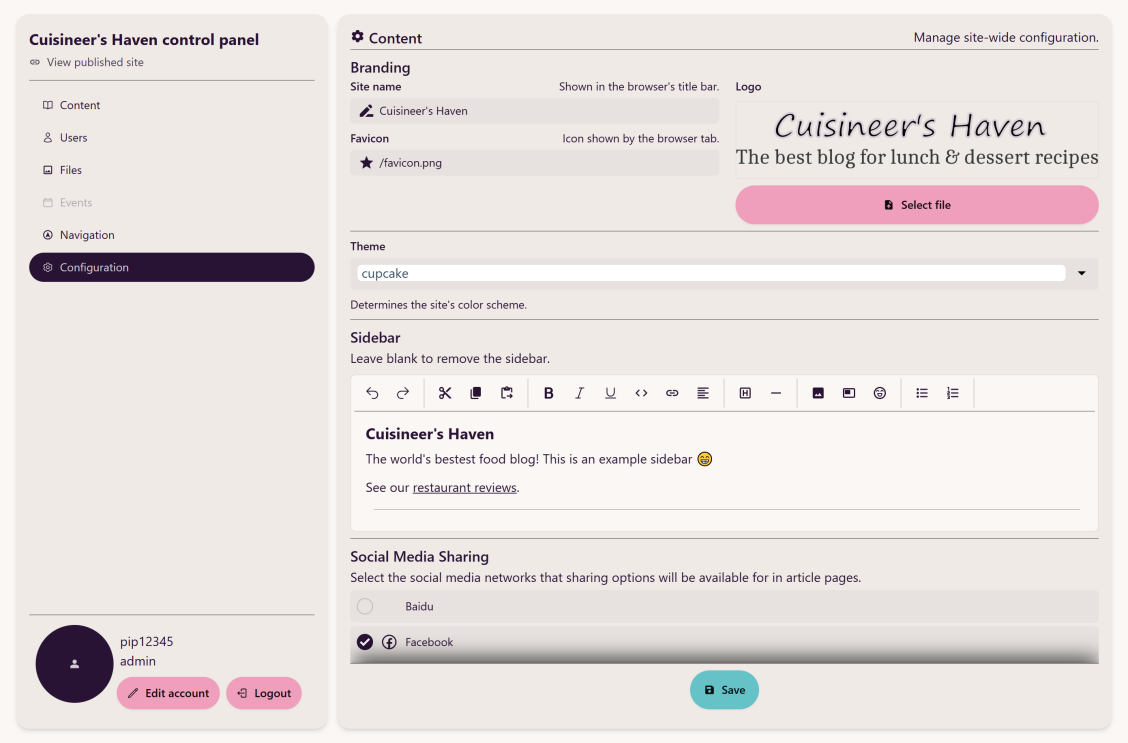


Figure 13: Site settings tab within the control panel.

The logo and favicon images are set by selecting a file from the site's file storage, through modals that resemble traditional file pickers(*See annex figure 29*). The site's sidebar may also be customized using a rich text editor; this re-uses functionality from the article editor, which will be covered in Section 5.4.

The control panel also allows customizing the site's navigation bar, which is also modelled as a tree(*See annex figure 26*). The root nodes of this tree correspond to the top-most navigation options, which may be links to categories, articles, external sites or groups with child links that are displayed as dropdowns. A preview of the resulting navigation bar is displayed for convenience.

The site also supports Google Analytics for tracking traffic across the pages, though this must be configured via environment variables instead, as it's necessary for the analytics ID to be present at build time.

The control panel is only accessible to site staff. If the user accesses any page of the panel while not logged in or with expired credentials, they are redirected back to the login page(*See annex figure 25*).

5.3.4 API integration

The frontend interacts with the backend's API through classes in the `services` folder, which are wrappers around the Axios[41] HTTP library.

A separate service class exists for each major endpoint in the API (ArticleService for `/articles/`, UserService for `/users/`, etc.) with methods that correspond to the API endpoints to encapsulate requests and receive responses asynchronously, using parameter/return types that mirror the API schemas.

The service classes inherit from a base Service class, which provides protected methods that wrap Axios's HTTP methods. To reduce code repetition, these wrappers automatically pass common request headers such as the authentication token, if its cookies are present.

Since these classes are stateless libraries, only a single instance of each is created, when they are used for the first time by a component. Components access them via a Vue composable(*See annex 10.1.5*) to remove the need to explicitly import them.

The components use the tanstack-query library to automatically perform calls to the services when they need data from the API. The responses are cached to be reused across different components to reduce API calls during a session(*See annex 10.1.7*). Shared queries are extracted to composables, which in some specific cases also transform the API responses into more convenient data structures based on the needs of the components.

These queries also make it convenient to track whether API requests are still pending, which components use to show feedback to the user while fetching/posting data (via spinners or other widgets in place of the content being loaded). Requests that fail also show user-friendly versions of the HTTP exceptions and error messages, using toast notifications from the Nuxt UI library.

5.3.5 Component design

As was mentioned in the technology overview, Vue is a component-oriented UI framework, where pages are created by aggregating components that manage subtrees of the page's DOM[42] elements.

[41] Axios: promise-based HTTP library for JavaScript.

[42] Document Object Model: in-memory representation of a website's HTML, intended to be interacted with by JavaScript[8]

The project's implementation of the Vue components follows certain patterns to improve their ease of use, reusability, and maintainability.

Business logic is generally present in top-level components rather than scattered through their children, so as to centralize it and make interaction flows easier to follow. Components make use of custom events to simplify this parent-child communication, converting input events to contextual ones that are propagated to the parents, which act as the viewmodels.

The injection pattern is used to provide major viewmodel variables from top-most components to all their children in a convenient manner, rather than needing to explicitly pass them along the hierarchy(*See annex* 10.1.14). This is used mainly by recursive components such as file trees to avoid the "prop-drilling" anti-pattern, where all components along a hierarchy need to explicitly define and pass the prop variables, creating code repetition[**14**].

Pages with similar layouts, such as the various pages of the published site, use a single component that renders the shared elements, and uses Vue slots[**16**] to allow inserting the elements that do differ between those pages. This avoids code repetition from needing to insert common elements such as the site logo or navigation bar.

Modals are used very frequently throughout the control panel to display pop-up forms for editing and creating resources such as categories or user accounts. For this reason, their common layout (header, body, and footer with close & confirm buttons) and behaviour is extracted to a component that provides slots to customize the content within, and handles common behaviour such as confirming/closing a modal with keyboard shortcuts.

To make these modals reusable across different pages, their logic is also self-contained; for example, the modal to edit an article's metadata manages not just the form elements, but also the API calls to update the article. The parent component simply calls an exposed method to open them when needed, and is notified of when its flow completes (ex. when the article is edited through the form) via an event. This allows these modals to be re-used in different contexts; for example, the modal to upload a file to the CMS is present both in the dedicated files tab in the control panel, and in the article editor when choosing to insert an image by upload.

These considerations have made the frontend's components significantly more maintainable and flexible, allowing them to be reused efficiently throughout the user interfaces during the project's development. Many of these patterns also applied to the article editor's implementation, which is detailed in the next section.

5.4 Article editor

The article editor is a cornerstone page of the application, where the site staff write content for the site in a WYSIWYG editor. The editor may be accessed by clicking on an article in the content panel of the control panel, described previously in Section 5.3.3.

The editor is complex enough to warrant separate sections for its end-user features and their underlying implementation; this first section will introduce the features of the editor, with the implementation being detailed in Section 5.4.3.

The main view of the editor(Fig. 14) consists of a standard menu bar for global settings, followed by a toolbar and the area where the article content is written.

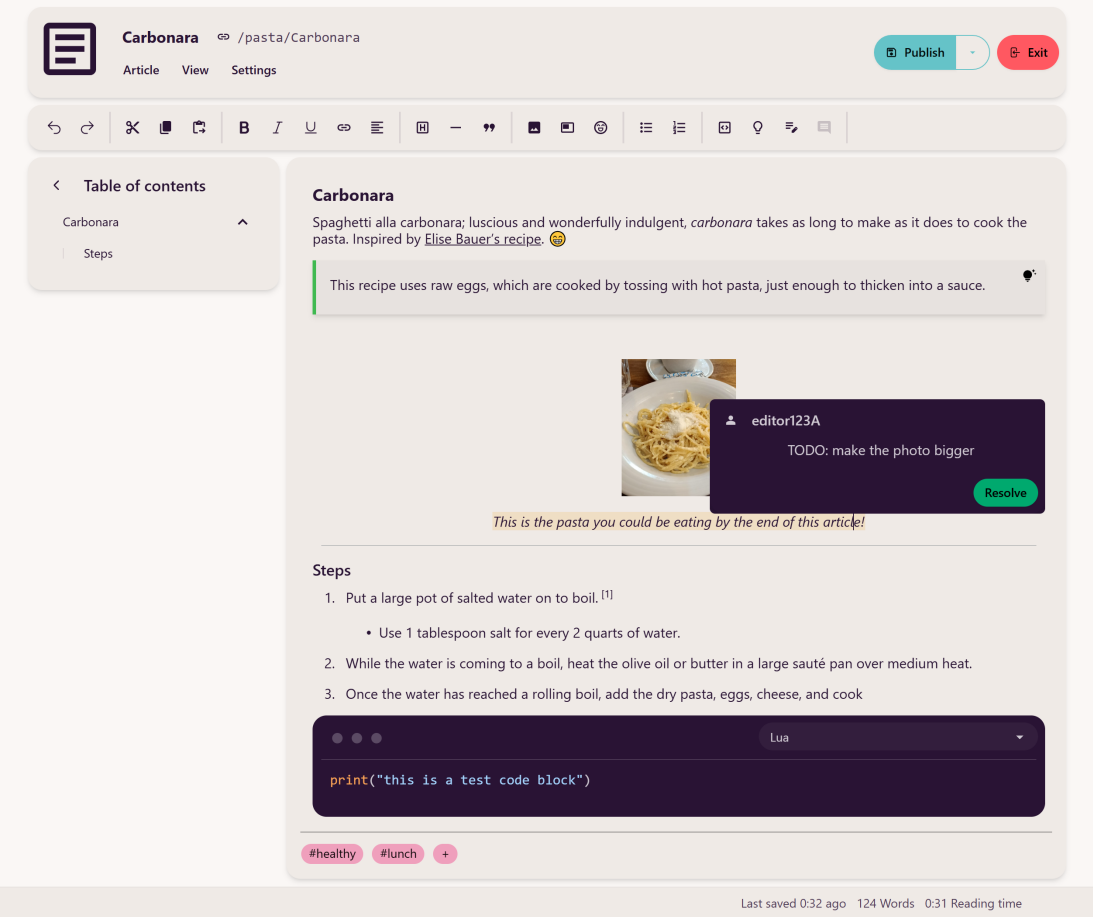


Figure 14: Article editor page in use, showing various rich content elements that it supports.

The editor supports the following rich content:

- Formatting text as bold, italics, underline, or headings.
- Links and images, supporting alt text.
- Embedding media from other sites, such as YouTube videos.
- Emojis, inserted from a filterable & searchable widget.
- Code snippets, with configurable syntax highlighting.
- Footnotes to include additional information, displayed in tooltips.
- Coloured notes/admonitions, useful to highlight warnings or important information, for example in tutorial-style blog content.
- Annotations that are only visible to other site staff, which allow leaving internal notes for collaboration or "To-dos".

Undo/redo is supported, as well as copying & pasting content, which works not just for text but also the rich content blocks(*See annex* 10.2.5). It is also possible to select and drag-and-drop content to reorder it quickly.

To achieve flexibility of use, the editor offers multiple ways of inserting content or executing other editing actions. They may be run with keyboard shortcuts, from a context menu (with options based on the element at the cursor/selection), or through buttons on a toolbar that sticks to the top of the viewport. The properties of some elements, like the URL of an image, may be edited through a context menu or by double-clicking them, which opens contextual modals to edit the elements(*See annex figure* 33).

The editor supports multiple ways of inserting images, through different options accessed from an "Insert image" dropdown in the toolbar:

1. Uploading them to the site's CMS, which is equivalent to doing so from the "Files" control panel in Section 5.3.3.1, using the same form but without needing to exit the editor.

2. Hot linking an image (ie. setting the image URL manually).

3. Selecting an existing image from the site's CMS, from a file-picker modal with a tree similar to the one in the files control panel(*See annex figure* 32).

Pasting an image URL will automatically embed it by doing string pattern matching on the clipboard content; this is also supported for media embeds, such as pasting YouTube links.

Long articles may be navigated using a "table of contents" sidebar that displays the headings in the document as a tree; clicking a heading will jump to that section. A status bar at the bottom of the viewport displays statistics, such as word count, estimated reading time and the last time the article was saved to the CMS. Saving can be done through a button in the top-right, which also supports saving changes as a "draft", which will not affect the published article, allowing the user to work on edits to an already-published article across multiple sessions.

5.4.1 Customization and shortcuts

A settings menu allows customizing the functionality of the editor(*See annex figure* 34). The keyboard shortcuts for tools can be rebound, and options the user doesn't use can be hidden from the toolbar to reduce visual clutter, as some functionality may not be relevant to the user based on their blog's topics.

To accommodate users that are used to writing documents in plain Markdown or other markup languages, the editor supports "shortcuts" to insert some rich context by inputting its corresponding Markdown syntax. For example, typing ** (2 asterisks) will toggle italics, typing ">" at the start of a paragraph will insert a quote, and ctrl + enter will toggle a bullet list(*See annex* 10.2.2).

5.4.2 Metadata editor

An article's metadata, such as title, publish time, authors, whether it admits comments, etc. can be modified from a form accessed from the "File" menu(Fig. 15).

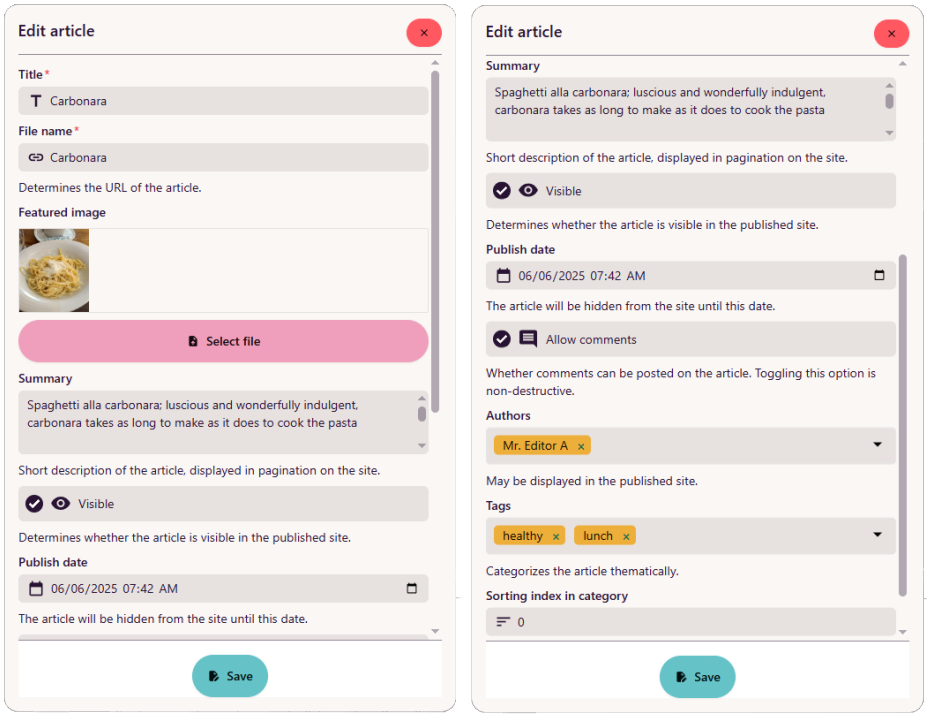


Figure 15: Form modal for editing an article's metadata.

Articles may be hidden to temporarily remove them from the site without deleting them, and they may also be scheduled to be posted at a specific time in the future. They may also be given "summary" text, which is displayed when previewing the article in paginated pages on the site. If not set manually, it is auto-generated from the first paragraphs found in the text.

An article can be credited to multiple authors through a multi-select combo-box. Since the site might have many authors, this field supports searching by name to narrow down the selection.

A similar multi-select combo box is used for adding tags. Tags that are already used by other articles appear as suggestions while typing, to avoid users from accidentally spelling a tag differently across articles (ex. using a "cooking recipe" tag in one article, but "cooking recipes" in another).

The following subsection will cover the technical details of the editor's implementation.

5.4.3 Implementation

The implementation of the article editor consists of a Vue wrapper for the ProseMirror framework, applying an MVVM architecture and custom models as facades into what is otherwise a pure JavaScript library.

In rich text editors, the document is not viewed as a stream of characters, but is instead typically structured into semantic blocks, such as paragraphs or images. ProseMirror models documents as a tree-like structure of ordered *nodes* defined by a custom schema. Nodes represent "block-level" content (document elements that

are typically stacked vertically) and may also contain text, *marks* to format the text, and metadata *attributes*.

The project's schema defines nodes for the various rich elements it supports (images, code blocks, footnotes, etc.) along with their attributes, nesting rules and how they're rendered to HTML. Node attributes are used to store properties such as the URL of an image node or the alignment of a paragraph node, whereas text formatting such as bold, italics, and links are implemented using marks.
Simple nodes like text paragraphs are rendered as plain HTML for best performance, while Vue components are used for complex or interactive nodes like code blocks, using the prosemirror-adapter[43] library.

The ProseMirror structures are immutable; editing the document is done by applying *transactions* that specify the operations, such as inserting nodes or modifying their text and attributes.

The project's rich text editor aims to create extra layers of abstraction around ProseMirror to make implementing the required features convenient, with flexibility to add more in the future.

The implementation of the editor follows an **MVVM**[44] pattern(Fig. 16) to decouple the model/state from the view, a necessity for the codebase to be flexible and open for extensions. The elements of the implementation are as follows:

- Vue components act as the **view**, displaying the document elements, toolbars, and offering various widgets for user interaction, with only presentational logic.
- Vue *composables* act as the **viewmodels**, offering reactive bindings of model data to the UI as well as callbacks to perform editing actions on the model document.
- Plain TypeScript **model** classes handle state mutations on the ProseMirror document, as well as storing user preferences.

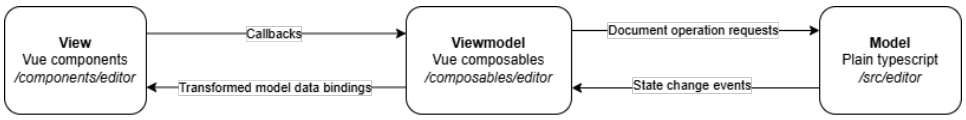


Figure 16: The editor's MVVM pattern diagram.

5.4.3.1 Model classes

The `Editor` class is the main **model** class which wraps the ProseMirror document state class, responsible for applying transactions to edit the document and storing user preferences.

[43] prosemirror-adapter: library for integrating ProseMirror's node renderers with modern web frameworks.

[44] Model-view-viewmodel: architecture for graphical applications relies on the observer pattern to decouple UI from business logic.

Editing operations, such as changing the formatting of text or inserting nodes, are implemented using an `Action` interface that acts as a wrapper around ProseMirror's transaction system. Each operation is a derived class that overloads an `execute` method that creates a transaction which performs its effects on the document state.

These actions are designed to be flexible and parameterizable; for example, formatting text can be done with the `ToggleMark` action, whose constructor takes a *mark* type from the document schema and the resulting `Action` instance applies/removes that mark to the selected text when it is executed.
This allows the action to be re-used for all the markup the editor supports (bold, italics, code, etc.). Most action classes are thus general-purpose and can be reused or extended to handle additional node / mark types. Their implementation is built on top of a custom library of utility functions to manage the ProseMirror document model(*See annex* 10.2.3).

The class diagram for the editor model classes(Fig. 17) is as follows:

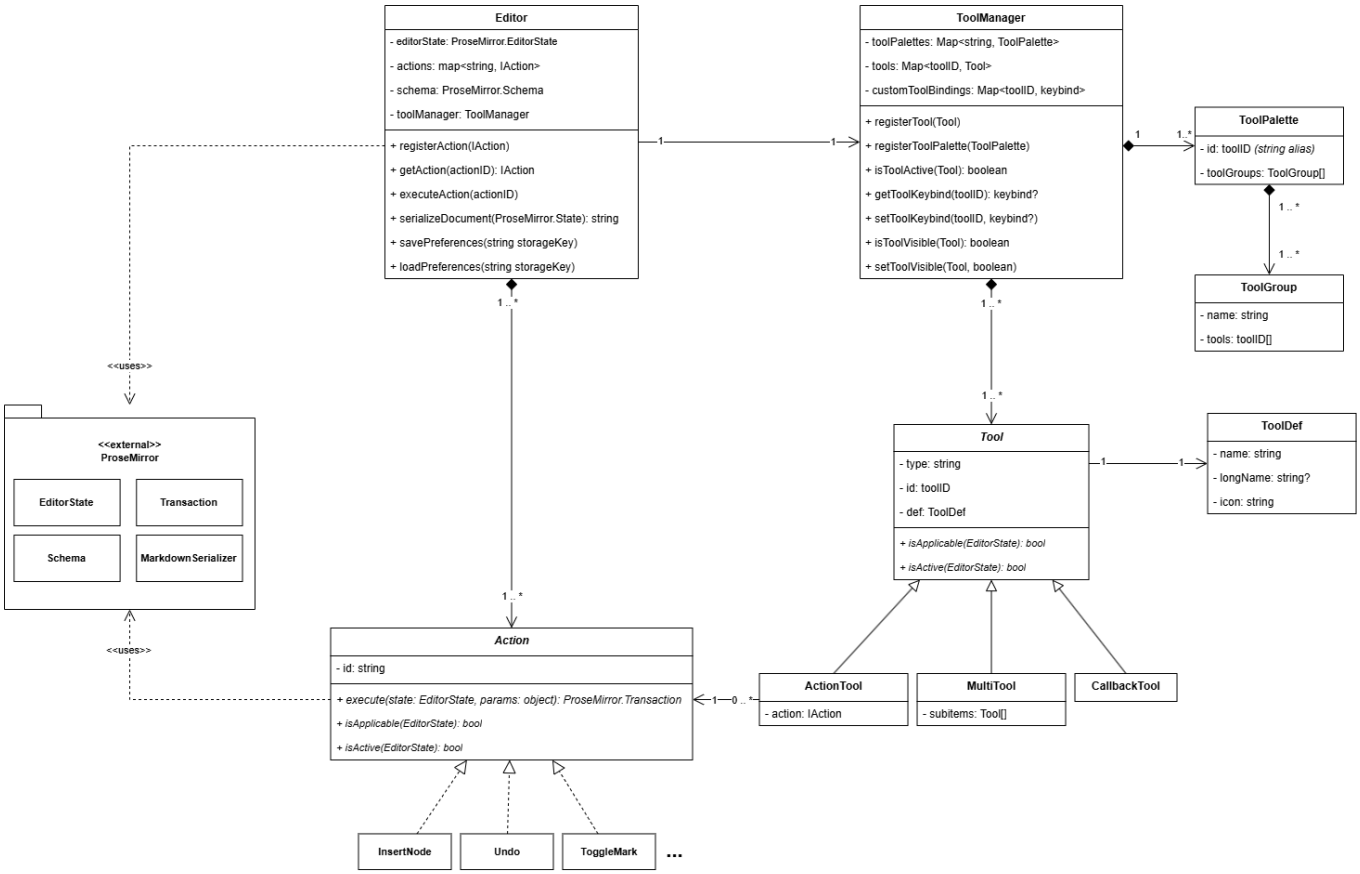


Figure 17: Class diagram of the editor's model. *Getters were omitted for readability.*

The `ToolManager` class manages the editor interactions that should be available to the user, such as the aforementioned editing actions, structuring and ordering them as well as storing the user's preferences of their key bindings. Its purpose is to group interactions of similar nature (via the `ToolPalette` class), which allows the UI to then present them in an organized manner (in a toolbar, context menu,

menu bar, etc.).

These interactions are defined through the `Tool` abstract class, which contains methods that dictate if the tool is contextually valid for the document state (mainly checking what content is currently selected or underneath the cursor), and provides metadata such as name and icon intended to be displayed by the UI.

Multiple subclasses of `Tool` exist for different interaction needs:

- **ActionTool**: executes an editing action at the current cursor position in the document. Intended for simple editing operations that do not need any extra user interaction flows, such as formatting or copy/paste.
- **CallbackTool**: sends a simple event back to the viewmodel. Intended to request application-level operations that do not affect the document content, such as saving the document or starting some secondary interaction flow (ex. opening the article editor's modal to upload an image).
- **MultiTool**: aggregates "variants" of a tool. Intended to group different ways of achieving a same end result, to present those tools in some compacted manner in UIs such as a dropdown. An example of this is the "Insert image" group in the article editor, which contains the various methods of inserting images (by upload, from URL, or from the site).

For flexibility, these items are not limited to the ones needed for the article editor; additional derived `Action` and `Tool` classes can be registered to the Editor class to extend the editor's functionality and solve open-closed concerns when reusing the editor in a different context, which the project does for the comment editor and site sidebar customization.

Tools support configurable keyboard shortcuts, as well as choosing which items should be hidden from the UI, which are persisted using localstorage[45] with simple map and set structures respectively by storing the tool identifiers.

5.4.3.2 Viewmodels

The **viewmodel** is implemented with Vue composables, which are functions that encapsulate stateful logic (instancing reactive variables, configuring event listeners, etc.) for components, intended to allow complex logic to be built by calling multiple simpler composables[**13**].

The project's viewmodel uses composables to extract presentational logic and data bindings that would normally be in a Vue component into functions with reactive dependency injections of the model. These are written using the functional-programming paradigm to allow more complex viewmodels to be created by wrapping simpler ones.

The viewmodel is the authority over issuing editing actions to the model and is split into multiple composables, each oriented towards developing different parts of the UI based on the data and callbacks they offer. An overview of them is shown in the following table(Fig. 18).

[45] A standard persistence mechanism for web applications[**10**].

Layer	Composable	Purpose
Application-level viewmodels	useArticleEditor()	Factory composable for instanciating an Editor model configured with the schema & tools for writing articles for the site
	useArticleEditorQueries()	Creates reactive API queries for fetching and mutating articles in the CMS to & from the editor's content
	useCommentEditor()	Factory composable for instanciating an Editor model configured with a minimal schema & tools for writing comments on articles
General-purpose Viewmodels	useEditorContextMenu()	Creates bindings for tools that should be shown based on the cursor position and selection in the document
	useEditorToolbarItems()	Creates bindings for tools that should be shown in a "main toolbar", filtered by user preferences
	useEditorSettings()	Exposes toolbar user preferences, configuration options and callbacks to set them
	useEditorMultiTool()	Creates bindings for the sub-tools of a "Multi-tool"-type tool, filtered by user preferences
Basic model bindings	useEditorTool()	Creates reactive bindings for the state of a specific tool (whether it's currently applicable, in use, its keybind, etc.)
	useEditorTools()	Exposes callbacks to run the effects of a tool
	useEditorToolCallback()	Registers a callback to run when tools are used
Dependency injections	useEditorProvides()	Provides the Editor model class dependency to all child components
	useEditorInjects()	Accesses the Editor model dependency injection from useEditorProvides()
	useEditorSchema()	Accesses the document schema of the Editor model dependency injection

Figure 18: Table of viewmodel composables grouped by layer/responsibility. Composables of higher categories depend on those of lower ones.

For example, the `useEditorContextMenu()` composable creates reactive bindings for options that should be shown in a context menu, filtering the toolbar items registered in the model based on which ones are valid for the document content underneath the cursor, and providing methods to execute their actions on the model.

This decouples the logic of a context menu from the UI that would display it, allowing different views to be implemented for it, or extending it to create more complex viewmodels without code repetition. This gives the editor high flexibility, and allows it to be re-used in other contexts (such as the comment editor) or projects while having full control over its features and look of the UI.

5.4.3.3 View

The project itself implements multiple **view** components for the different text editors that appear throughout the site: the article editor, the comment editor on articles pages, and the editor for the site's sidebar, each using a subset of the viewmodel's composables based on the features needed.

The article editor, being the most complex feature-wise, is composed of a multitude of components to display article metadata and the interactions with the document. The following figure is an overview of its component hierarchy(Fig. 19).

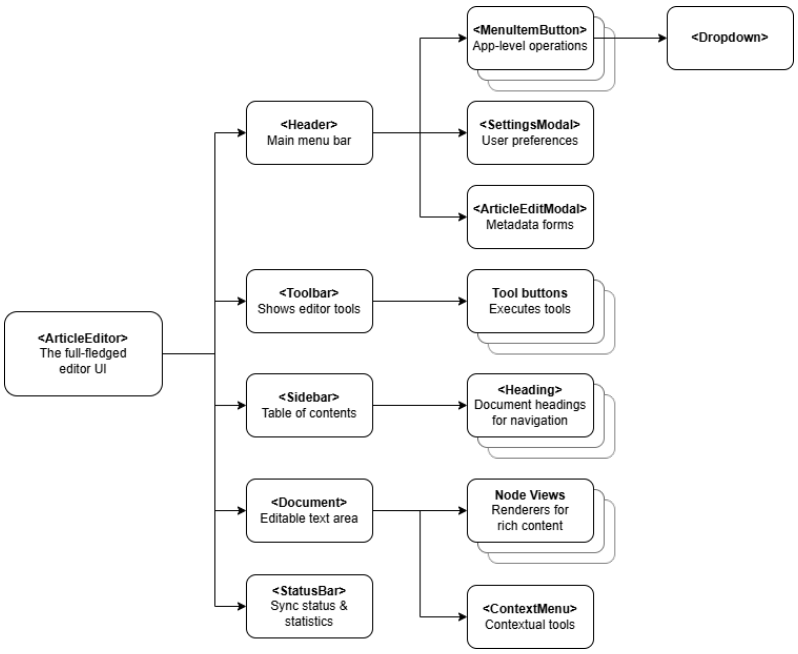


Figure 19: Diagram of hierarchy of the major Vue components used by the article editor's view.

The editor features various widgets and modals for contextual interaction flows, such as the context menu or the settings modal, invoked through event listeners when corresponding UI elements or tools are used. These interaction flows eventually dispatch requests to the viewmodel to run actions to edit the document model(*See annex* 10.2.6).

Additional MVVM implementation notes and design iterations are described in Annex 10.2.7.

5.4.3.4 CMS integration

Since reusability was one of the concerns for the editor's implementation, its inner workings are uncoupled from the blog CMS. Integration with the CMS is handled by additional viewmodels and components in the view layer.

Articles on the backend are stored into text files with Markdown-like syntax for portability reasons, to allow the site's content to be easily migrated, viewed or edited using other applications if necessary. As such, the ProseMirror document, which is an in-memory node tree format, must be serializable to text. Similarly, for an article to be loaded in the editor, it must be parsed from Markdown into the project's ProseMirror tree schema first.

The project uses the **MarkdownIt**[46] library for serializing & parsing the nodes to & from Markdown, using either the CommonMark[47] specification or popular non-standard syntax extensions to ease interoperability with other Markdown editors. More exotic nodes such as footnotes needed custom serialization & parsing, which was implemented as MarkdownIt plugins(*See annex* 10.2.4).

[46] **MarkdownIt**: library for parsing Markdown to & from HTML.
[47] CommonMark: a strongly-defined Markdown syntax specification.

Given that the article editor is a WYSIWYG editor, it is also expected that the content of the articles within it display the same way as they'd do on the published site. To achieve this, article pages are rendered using a reduced, read-only ProseMirror instance that parses the document and renders it using the same node renderers as the editor itself. This ensures the look of elements is consistent and avoids needing to implement a separate Markdown-to-HTML renderer. Node renderers check if the editor is in this read-only mode to hide interactive widgets, such as the language selector for code blocks.

As a whole, the project's text editor turned out quite feature-rich and well-integrated with the CMS, while also leaving room for easy additions thanks to its well-crafted architecture.

5.5 Containerization

The project uses **Docker**[48] to containerize the frontend and backend services to make their deployment easily replicable, using separate *dockerfiles* to simplify their use in deployment and avoiding environment conflicts between them.

The frontend container uses **Nitro**[49] as the web server, which is Nuxt's custom Node.js server, needed for the server-side rendering functionality. The backend container runs the API using **uvicorn**[50], an ASGI server with high vertical scalability.

To optimize build times, the `--link` Dockerfile option is used when copying the source code into the container; this option allows caching operations done with the files copied, namely the entire build process of the frontend. This allows the lengthy build step to be skipped if the files haven't changed, which greatly speeds up container start-up when the container needs to be restarted (ex. after server maintenance).

Docker Compose is used to coordinate launching all services for local development and testing. Aside from the frontend and backend services, the compose file launches a PostgreSQL container, which the backend depends on. The compose file also includes a PGAdmin[51] service, a web-based PostgreSQL administration interface useful for inspecting the database.

It's worth noting that the frontend server also communicates with the backend, as the pages that use server-side rendering require the frontend server to make API calls itself to pre-render HTML to send to the client. The server is configured to use the internal Docker network to speed up the API calls, as opposed to the public API URL the browser clients would use.

A "tests" service also exists, which executes pytest in the backend container. This service is disabled by default, being used only by continuous integration GitHub actions to run tests after every push.

The backend and frontend container images were uploaded to Docker Hub, which was used for deployment as per the section that now follows. Detailed instructions on running the project are also provided in the *readme* file.

[48] **Docker**: software containerization engine.
[49] **Nitro**: web server toolkit from the creators of Nuxt.
[50] **uvicorn**: Python-based ASGI web server
[51] PGAdmin: PostgreSQL database management tool.

6 Deployment

A deployment of the project was tested on the Render[52] cloud application platform.

The deployment consisted of provisioning a PostgreSQL database and two of their "web service" instances, on which the backend and frontend were deployed separately using their container images that were uploaded to Docker Hub, as per Section 5.5 «Containerization».

The deployment test used the following architecture(Fig. 20), barring the Elasticsearch instance to save costs(*See annex* 10.8.3).

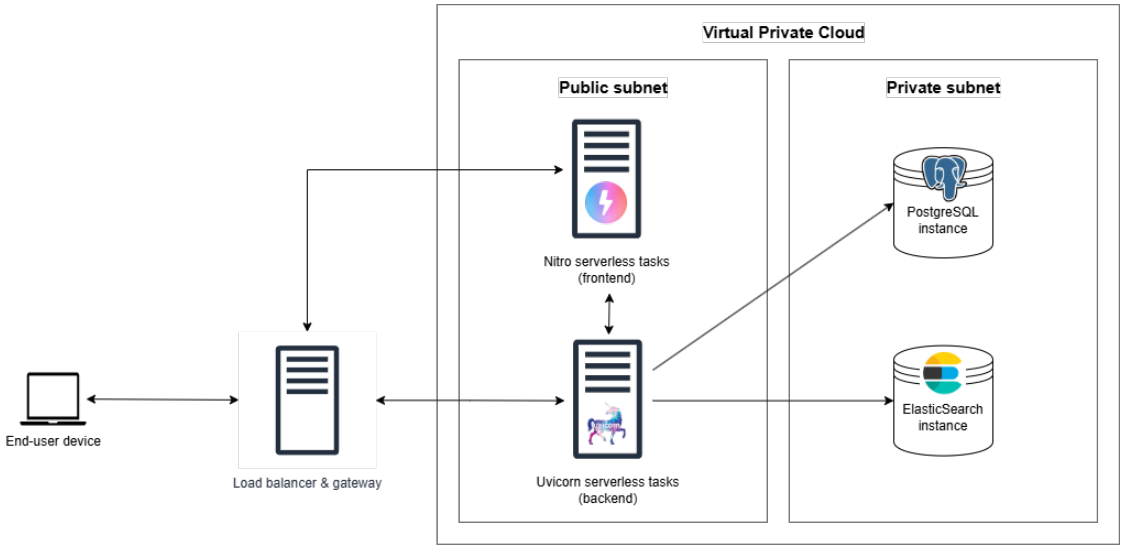


Figure 20: Deployment architecture diagram.

Both the backend and frontend were deployed as "serverless" services, a cloud service type where instances of the service are launched as they are needed rather than provisioning a dedicated server in advance[4]. This kind of deployment is effective and highly cost-efficient for services that don't receive constant traffic, such as small-scale blogs that the project targets.

The entry point for the service is a load balancer[53] that spins up instances of the frontend & backend as necessary and distributes traffic between them, allowing for horizontal scaling in case of a spike in traffic. Both of the container images perform very few tasks on start-up, such that there is little delay in handling responses even if the service is accessed when there are no instances running.

The only hurdle specific to the project was configuring the environment variables to have the frontend & backend use the correct URL for the API and database connection respectively within the private virtual cloud (PVC) subnet. The database was configured to only accept connections within the platform's private network for security, preventing external access.

[52] Render: cloud deployment PaaS (platform-as-a-service) by Render, Inc.

[53] Auxiliary server that acts as an entry point and distributes traffic across an application's servers to avoid computational or network overload.

SSH was used to connect to the backend to run additional scripts to populate the database with mock/test data, corresponding to the example cooking blog shown in prior figures of this document(*See annex* 10.8.1).

A similar deployment would be possible on Amazon Web Services or Google Cloud, using their equivalent serverless services (AWS Fargate[54] and Google Cloud Run[55] respectively), though aspects like the load balancer, subnetting and HTTPS would need to be configured manually.

It's worth noting that the frontend is missing a cookie consent form and a clean way of including a privacy policy. These would need to be added when hosting a blog with the project that serves users in countries that require these forms by law.

7 Evaluation

The project's reliability was evaluated through usability testing and web auditing tools to determine if the project's features and UX were adequate and properly executed, as well as to detect issues and possible improvements.

7.1 Usability tests

Usability testing is an evaluation technique for products focused on user interactions, which involves having user with no prior exposure to the system test it in a controlled environment [19]. Usability testing applied to software allows gaining insights on the intuitiveness of its design, and was chosen for the project to validate its UX and visual design decisions.

Usability testing requires a structured plan, detailing the information that will be gathered, its purpose, and a systematic plan for the execution of the tests, which will be covered in the following subsection.

7.1.1 Methodology

Due to the project's proof-of-concept state, it was decided to design the usability tests to focus more on qualitative tests, which is best suited for discovering UX problems[19], as benchmarking with quantitative metrics is mainly useful for applications in production.

Following guides on user testing with low resources[23], the following considerations were taken to ensure the efficacy and correctness of the tests:

- Anonymized data was collected using a spreadsheet to organize it per-user and per-task.
- A within-subjects approach was chosen due to a low participant account; all participants took on the same test tasks to maximize feedback gathered.
- All participants signed an informed consent form and no personal data was kept.

[54] AWS Fargate: serverless cloud computing service, part of Amazon Web Services.
[55] Google Cloud Run: serverless cloud computing service, part of Google Cloud.

- The tests were done in-person using a device specifically prepared for the test, running the project locally with the database pre-populated with data designed for the test's tasks to avoid unexpected issues.
- The tests were overseen by a moderator that helped record data and observations during the tests.
- The test was designed to last 40 minutes at most to avoid participant fatigue.
- A "pilot test" was first performed to detect and amend any issues with the test plan itself.

The participants were informed of the nature of the test prior to attending. The test itself started with an introduction to the project and the motive of the test (finding issues with the project), assuring the subject that they themselves were not being judged. The participants then signed the informed consent form and proceeded to perform the tasks, which were explained to them one at a time, and the users were asked to confirm they understood them.

The test consisted of 5 tasks on the site designed to evaluate the major features and user flows. These tasks had users first assume the role of a reader/site visitor, followed by tasks where they switched to a staff role to test the content management tools of the control panel. These five tasks were:

1. Finding an article of a specific topic on the published site, which tested the various navigation options (search bar, pagination, category pages).
2. Posting a comment on an article, which tested the log-in flow and the comment editor functionality.
3. Signing in to the control panel to create a new article for the site; this more free-form task served to test the first impressions and intuitiveness of the article editor.
4. Editing a specially-prepared draft article and finishing it; the users were told they had to finish an article that was started by another staff member, who left annotations on work that was left to do on the article. This tested the collaborative annotations, as well as specific features of the article editor like editing bullet lists and adding tags.
5. Changing the site's configuration, such as logo and the navigation bar, which tested various features from the control panel.

The tests had a qualitative focus, with its value coming from mainly from noting down user errors and points of frustration of the users, as well as explicit feedback gathered from each task. The users were asked whether they found it easy to perform the tasks, as well as for suggestions on improvements they would've liked to see for the systems.

A few metrics were still tracked such as time taken, task success (completed, partially completed or uncompleted), amount of user errors (mis-clicks, misunderstood

UI elements, etc.) and facial expressions (positive and negative). These helped with identifying parts of the site with subpar usability, even in absence of direct feedback from participants. The script of the tests is available in Annex 10.9.1.

7.1.2 Execution and insights gathered

The tests were performed with 4 participants with a certain technological background and experience with similar blog systems. The results of the tests showed that the features of the project were generally coherent and intuitive, but with particular aspects dragging the experience down. An example of the data collected for the 4th task is shown below(Fig. 21).

				Expressions and comments		
User	Success	Time	Errors made	Positive	Negative	Observations
1	Yes	6:13	3	0	2	Tried opening the links added in editor, and thought the tooltip field was the link text. Found it difficult to add tags, was expecting them to be at the bottom of the article
2	Partial	6:40	3	0	1	Was not able to find the article metadata form to add tags; tried doing so by writing "#tag" in the article and looking at toolbar options. Also misunderstood one of the tasks (reordering sections) Did not notice annotation popups at first, thought the yellow highlight was just formatting
3	Partial	3:40	2	2	1	Tried adding tags with a "#tag" text and tried opening inserted links. Used lists, copy+paste, alignment and resizing well. Was not initially aware that the yellow highlighted indicated annotations. Commented that the resize cursor should not just show a Y-axis resize. Commented they found the editor to be easy to use overall.
4	Yes	4:39	2	2	0	Initially opened the "edit metadata" form instead of editing the article content - figured it out shortly later. Tried dragging the image to center it. Did not resolve annotations, only deleted the highlighted text. Completed most of the tasks very efficiently (copy+paste, select + toggle list, etc.) Resized the image using the form instead of the handle - commented later that it was hard to see. Struggled to find the tag options (also wrote #tag), but then saw them when inspecting another article and did figure it out shortly later.
Average		5:18				
Deviation		1:23				

Figure 21: Observations and metrics from the 4th task of the usability testing.

In particular, the fourth and fifth tasks saw a lower completion rate and a higher error rate. In the fourth task, users struggled with an objective that required them to add metadata like tags to the drafted article, failing to find the option in the menu bar that allowed them to do so. The feedback indicated the issue stemmed from the view in the article editor not being completely coherent with how the article is shown on the site, and that the users expected better consistency given that it's a WYSIWYG editor.
The fifth task was also found to have been a bit poorly written, in the sense that some sub-tasks seemed unrelated to each other, which caused some confusion among participants.

The task that required users to change site settings showed the most usability issues and direct complaints. Users reported the control panel's interface as being too visually dense and thought that the preview functionality for some changes, such as editing the navigation bar, was not up to expectations(*See annex figure 59*).
The results indicate that the control panel's interface is too technical, with multiple complaints of being overwhelming in terms of elements shown at once.

On the positive side, participants overall found the article editor to be intuitive and satisfactory; they were quick to find the tools and functionalities required, and also made correct use of additional features not strictly required by the tests.
In particular, two users praised the Markdown-like shortcuts and drag-and-drop functionality, which they commented they thought convenient to bridge an efficiency gap between markup editors and WYSIWYG ones in the context of blogging.

The tests related to the published site (tasks 1 & 2) showed no major issues; the users found the site navigation and commenting intuitive.

The results of the test were taken into account for future work decisions in Section 9, and some minor improvements were implemented shortly after the tests in response. The sheets used to collect the results and note down the observations and feedback are available in Annex 10.9.2.

7.2 Auditing and automated testing

Aside from user testing, the coverage of the backend's tests was measured, and some standard audits were performed on the frontend.

The backend tests achieve a coverage of 87% of lines of executable source code(Fig. 3). The tests focused covering the functional requirements, but there are two systems that they did not cover: Google SSO and searching. Tests for Google SSO would've required a way to mock OAuth flows, which was deemed too complex to set up for how little functionality it's used for. Tests for the search endpoints were omitted due to requiring an Elasticsearch instance, which would've slowed down test runs and their development significantly.

Source file	Lines executed	Lines missed	Coverage
crud\article.py	175	47	73%
crud\category.py	127	15	88%
crud\comment.py	42	4	90%
crud\file.py	73	27	63%
crud\site.py	102	15	85%
crud\user.py	170	43	75%
crud\utils.py	31	4	87%
routes\article.py	85	21	75%
routes\category.py	35	4	89%
routes\comment.py	41	5	88%
routes\file.py	72	33	54%
routes\search.py	25	9	64%
routes\site.py	55	18	67%
routes\user.py	74	5	93%
schemas\article.py	91	1	99%
schemas\user.py	64	2	97%
TOTAL	1262	253	87%

Table 3: Coverage report of the backend tests.

The frontend achieves an average score of ~80 across Google's Lighthouse[56] web auditing tool(*See annex* 10.1.3). The site scores well in SEO evaluations, but fails some accessibility checks, leading to some elements like the navigation bar being difficult to use with screen readers or smaller touch screens.

[56] Lighthouse: web quality auditing tool that evaluates a site's performance, accessibility and SEO factors.

Some elements also suffer from low text & background colour contrast. The severity of the contrast issues varies based on the colour theme the site is configured to use; given that the themes come from a well-established external library, it's possible its CSS classes were misused in these circumstances.

Performance-wise, the frontend scores well in regard to cumulative layout shift[57] thanks to the server-side rendering, whereas SPA tend to be prone to high CLS due to lazy-loading of content.
The LCP[58] metric could not be evaluated correctly on the published site as the mock data assets used on the test site were largely unoptimized in regard to file size, leading to the LCP metric being inflated. Pages with no images, such as the control panel, score decently for performance (87/100) with recursive elements like the tree items having the largest impact on rendering time, showing room for improvement.

7.3 Comparison with other blog systems

Though the project by no means can compete with the leading solutions like WordPress, it still has some advantages over other blogging systems.

The project supports having multiple individuals manage a single blog, which services like Tumblr or Medium don't support directly, being limited to blogs ran by individuals.
The project also has some limited asynchronous collaboration features like annotations in the article editor, which other CMS lack or only support through paid plugins such as Multicollab[59] for WordPress.

The Markdown-like keyboard shortcuts are another notable feature of the project that makes writing content more efficient for users coming from markup languages, and which the alternatives lack. Codeblocks and footnotes are other desirable features for modern-day blogging, which some platforms like Tumblr do not support, leading to a subpar experience for programming blogs.

A downside of the project is that the visual customization of the site is limited to just colour theme changes; changing the layout requires modifying the source code, as there is no drag-and-drop editor like in WordPress or Ghost[60]. However, the usage of Vue and a permissive license of the source code makes deeper customization viable for developers.

[57] Cumulative Layout Shift: measure of how much elements move in the site layout while it is loading. A high CLS leads to poor user experience and possible misclicks on the moving elements[7].
[58] Largest Contentful Paint, a metric for the highest time spent rendering the content of the site.
[59] Multicollab: Multicollab: collaborative plugin for WordPress.
[60] Ghost: Open-source blog CMS with a React frontend.

8 Conclusions

The main objective of the project was to design and implement a content management system from start to finish, going through planning, to backend & frontend implementation, deployment and evaluation, while also covering specialized needs of blog content creation with a custom rich text editor.

Looking back at the **3 major objectives** set in Section 1.1, the project was able to fulfill all three to a satisfactory degree. The project achieves the majority of the functional requirements planned, offering a fairly feature-complete blogging system with a convenient admin interface and a robust custom article editor.

In regard to **objective #1**, the backend and API cover the important features from the requirements gathering and are validated by tests; articles can be managed and searched, file management is present, and the API and account system factor in security concerns.
However, some smaller features, such as scheduling event announcements, RSS feeds and per-account editing permissions, were dropped to focus on polishing the rest of the project.

The frontend for **objective #2** is highly functional, integrating all of the API's features, and making good use of the chosen framework and libraries to serve the blog's content efficiently with server-side rendering and good SEO practices, although evaluation showed room for improvements in regard to accessibility and visual design.

The rich text editor from **objective #3** for writing articles was received positively in user testing, with a decent set of specialized features for efficiently writing blog-style content, as well as good integration with the rest of the CMS and some advantages over the built-in editors of other blogging systems.

On the technical side, the implementation of the editor turned out very satisfactory, with a robust MVVM pattern that would allow further features to be added easily.
The editor also proved to be very flexible, as it was able to be used not just for editing articles, but also for other text editing use cases within the project, such as the comments box and the site's sidebar.

Its correct implementation however was considerably more time-consuming than expected due to a lack of prior experience with creating rich text editing software, requiring numerous attempts and refactorings to achieve a clean and extendable codebase for it. Some minor features of the CMS had to be cut as a result.

In regard to the work methodology, the project would've benefitted from using more agile methodologies beyond the kanban board, even if they seem more oriented for teams. The planning was adhered to very rigidly, and it led to some time being spent on features that were ultimately deemed lower value than what was thought during the initial planning. Adopting some agile rituals such as sprints and planning the project in bi-weekly intervals would've led to better priorities, especially in the second half of the project's development.

From an academic point of view, the project was a great opportunity to put into practice many of the concepts and principles learnt throughout the degree. The full-stack nature of the project required highly transversal skills, from database structuring, to API & user interface design, deployment and user-centric evaluation. The rich text editor was a particularly interesting part, as it required applying those skills to a highly-specific software problem.
The opportunity to freely choose the technologies used was also a good exercise in critical thinking for making better framework and architectural decisions in the future.

Not all of the features identified during requirements gathering could be implemented during the project's time frame, and a few had to be cut due to priority changes, as mentioned in Section 2.
The following section details the possible further work on the project, including changes to the architecture and external libraries to solve maintainability concerns identified during development.

9 Future work

As mentioned previously, various features from the requirements gathering were either not included in the project's planning or dropped later due to time constraints; these excluded features were noted in the list of user stories in Figure 2.

Out of these, the most important and interesting one to undertake would be synchronous collaborative article editing, which would benefit the project in use cases where the site content is managed by multiple users.
ProseMirror, the framework used for the article editor, was designed with some concurrency in mind[17], such that real-time collaboration would be possible using a server that acts as a central authority for rebasing concurrent document changes. The remaining unimplemented editor "blocks" (tables and math formulas) would only involve creating additional node types, renderers and serializers for the editor's schema, which the project handles well for existing features.

The features from the unimplemented "event creation" epic could all be implemented as extensions of the article system; their only major difference in terms of functionality is needing to keep track of "subscribed" users and send notifications to them by e-mail. However, this would imply storing personal data, with all the privacy regulation concerns that this involves.

On the technical side, certain technical debt[61] was identified in both stacks, which would need to be addressed if development were to continue to keep the project maintainable.

On the frontend side, several dependencies turned out to be deficient. Nuxt UI, the library the frontend used for form elements, turned out to be a poor choice due to the elements being very inflexible in terms of usage and styling, requiring awkward workarounds to tailor its components to the project. In hindsight, it would've been wiser to spend some time trying multiple component libraries before

[61] Implied cost (either temporal or monetary) of future work on a software caused by earlier design choices not being adequately robust for a project.

committing to a single one. Both Nuxt UI and daisyUI released new major versions during the project's development that solved some of the major issues that plagued the project's development, however the migration work was too large to undertake at the time.

In regards to non-functional requirements, all the strings in the frontend would have to be extracted to support localization, which would've been more efficient to support right from the start of the project, especially in regard to strings with formatting.

Additionally, after finalizing the layout of the frontend's interfaces and making them responsive, it would be ideal to implement end-to-end testing[62] to be able to detect regressions in any future development.

On the other hand, **the backend** would've benefitted from using a documental database (such as MongoDB[63]) for semi-structured entities like articles and the navigation bar configuration, which were cumbersome to implement in SQL and poorly extendable. A mixed architecture that uses SQL for integrity-critical entities like user accounts and a documental database for the CMS content would've simplified development and offered a more extendable design.

Additionally, given how seldom resources like articles are changed, it would've been beneficial to implement caching at the backend level as well, using an in-memory database like Redis[64] to cache API responses and reduce workload on the backend server.

As was mentioned in the database implementation section, for performance and scalability it would be ideal to rework file storage to use an external CDN such as Amazon S3.

Finally, by the end of the project's development, some security issues with the Nuxt and FastAPI frameworks were published on CVE[65]. Although none appeared exploitable through the project's usage of these frameworks, it would be best practice to update the dependencies, while ensuring any breaking changes in newer versions are accounted for.

[62] Automated testing that verifies functionality from a user's perspective, involving both the user interface and backend systems.

[63] MongoDB: NoSQL documental database.

[64] Redis: in-memory key-value database.

[65] CVE: Common Vulnerabilities & Exposures, a database of publicly-disclosed software vulnerability issues.

References

[1] Scott Ambler. *Quality of Service (QoS) Requirements: An Agile Introduction*. URL: https://agilemodeling.com/artifacts/technicalRequirement.htm (visited on 02/21/2025).

[2] BuiltWith. *Blog Usage Distribution in the Top 1 Million Sites*. URL: https://trends.builtwith.com/cms/blog (visited on 05/25/2025).

[3] BuiltWith. *PHP Usage Statistics*. URL: https://trends.builtwith.com/framework/PHP?utm_source=the+new+stack (visited on 03/04/2025).

[4] Cloudflare. *What is serverless?* URL: https://www.cloudflare.com/learning/serverless/what-is-serverless/ (visited on 05/22/2025).

[5] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley, 2004. ISBN: 0321205685.

[6] Provos N. Maziéres D. "A Future-Adaptable Password Scheme". In: *The USENIX Association* (1999).

[7] MDN Web Docs. *Cumulative Layout Shift (CLS)*. URL: https://developer.mozilla.org/en-US/docs/Glossary/CLS (visited on 06/02/2025).

[8] MDN Web Docs. *Document Object Model (DOM)*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model (visited on 06/02/2025).

[9] MDN Web Docs. *SPA (Single-page application)*. URL: https://developer.mozilla.org/en-US/docs/Glossary/SPA (visited on 06/01/2025).

[10] MDN Web Docs. *Window: localStorage property*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage (visited on 05/24/2025).

[11] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (visited on 03/02/2025).

[12] Matthew Giannelis. *Top Global Blogging Statistics*. URL: https://www.techbusinessnews.com.au/blog/how-many-blogs-are-there-in-the-world-blogging-statistics-2023/ (visited on 03/04/2025).

[13] Vue.js User Guide. *Composables*. URL: https://vuejs.org/guide/reusability/composables (visited on 03/03/2025).

[14] Vue.js User Guide. *Provide / Inject*. URL: https://vuejs.org/guide/components/provide-inject (visited on 04/10/2025).

[15] Vue.js User Guide. *Reactivity Fundamentals*. URL: https://vuejs.org/guide/essentials/reactivity-fundamentals.html (visited on 05/28/2025).

[16] Vue.js User Guide. *Slots*. URL: https://vuejs.org/guide/components/slots.html (visited on 04/10/2025).

[17] Marijn Haverbeke. *Collaborative Editing in ProseMirror*. July 28, 2015. URL: https://marijnhaverbeke.nl/blog/collaborative-editing.html (visited on 04/27/2025).

[18] Stack Exchange Inc. *2024 Stack Overflow Developer Survey*. 2024. URL: https://survey.stackoverflow.co/2024/technology (visited on 03/03/2025).

[19] Dennis G. Jerz. *Usability Testing: What Is It?* July 19, 2000. URL: https://jerz.setonhill.edu/writing/technical-writing/usability-testing/ (visited on 05/22/2025).

[20] Michael B. Jones, John Bradley, and Nat Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015. DOI: 10.17487/RFC7519. URL: https://www.rfc-editor.org/info/rfc7519.

[21] Nuxt. *Rendering Modes - Nuxt Concepts*. URL: https://nuxt.com/docs/guide/concepts/rendering (visited on 03/03/2025).

[22] Ariel Ortiz Ramirez. *Three-Tier Architecture*. July 1, 2000. URL: https://www.linuxjournal.com/article/3508 (visited on 03/12/2025).

[23] M. Ribera. *Planificació d'un test d'usuaris. Cas d'estudi resolt. 1a. ed.* Fundació per a la Universitat Oberta de Catalunya (FUOC)., Sept. 2023.

[24] Chris Richardson. *Microservice Architecture pattern*. URL: https://microservices.io/patterns/microservices.html (visited on 06/03/2025).

[25] Ralf van Veen. *Single-Page Applications (SPAs) in SEO*. 2024. URL: https://ralfvanveen.com/en/technical-seo/my-strategy-for-seo-for-single-page-applications-spas/ (visited on 02/20/2025).

10 Annex

10.1 Frontend

10.1.1 Site screenshots

This annex contains screenshots of the pages and user interfaces of the frontend, for both the published site (from Section 5.3.2) and the control panel (from Section 5.3.3).

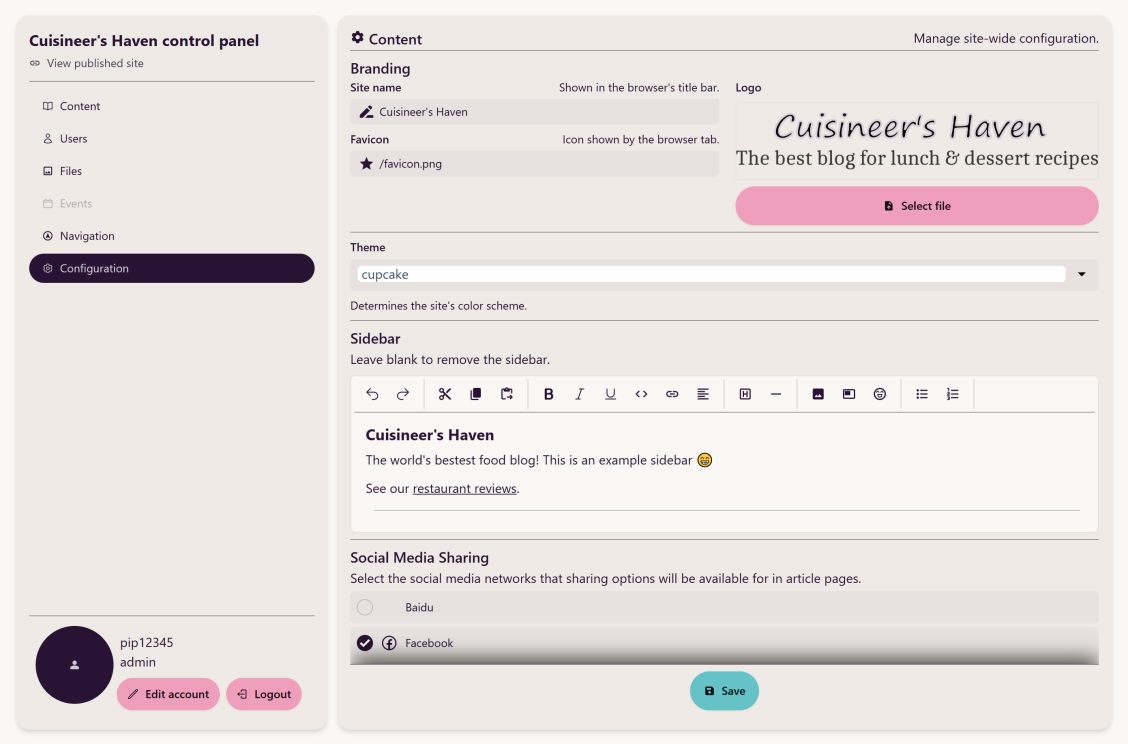


Figure 22: Site settings within the control panel.

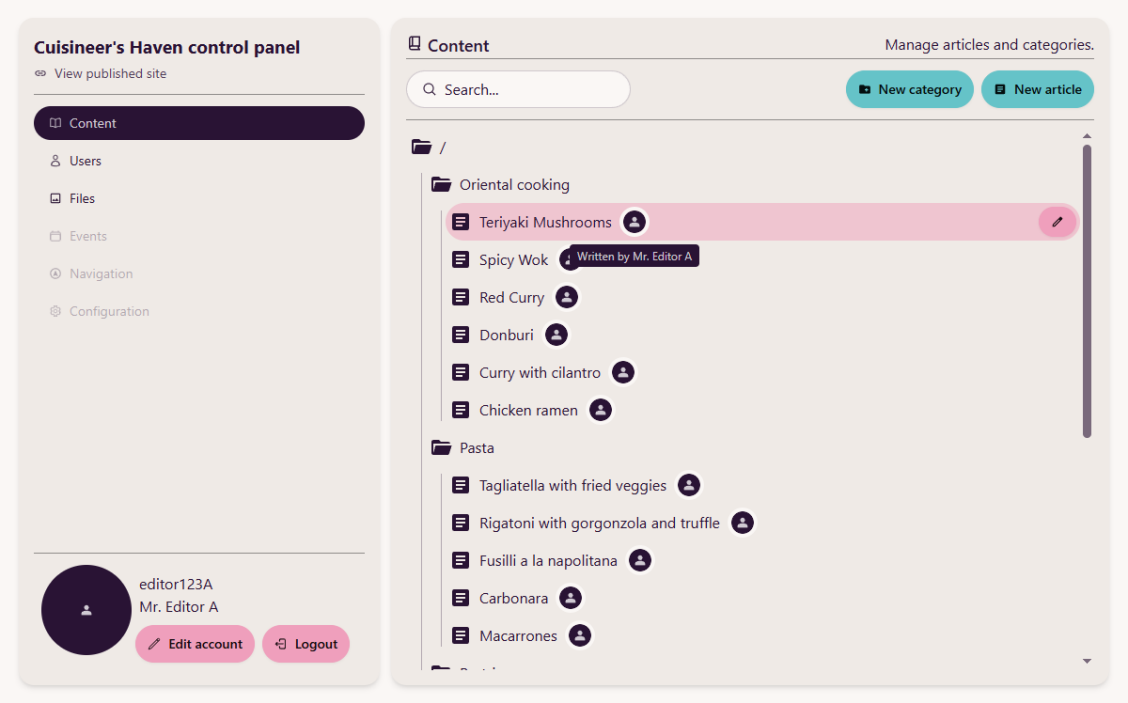


Figure 23: Article & categories management panel.

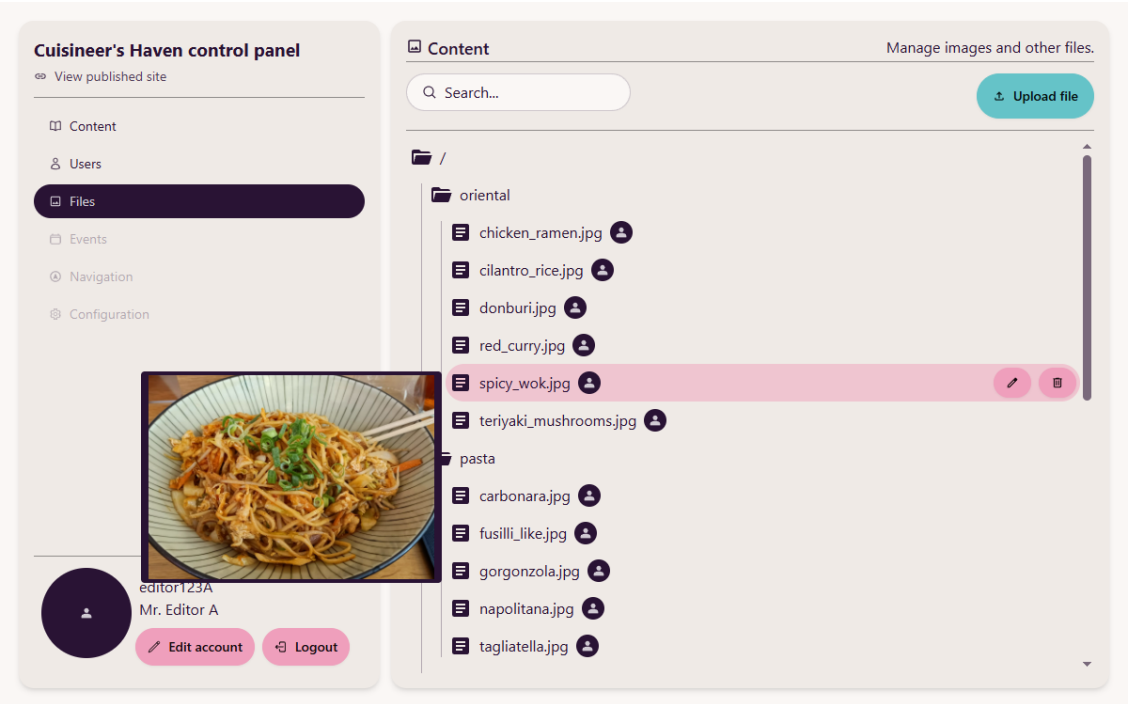


Figure 24: File management panel.

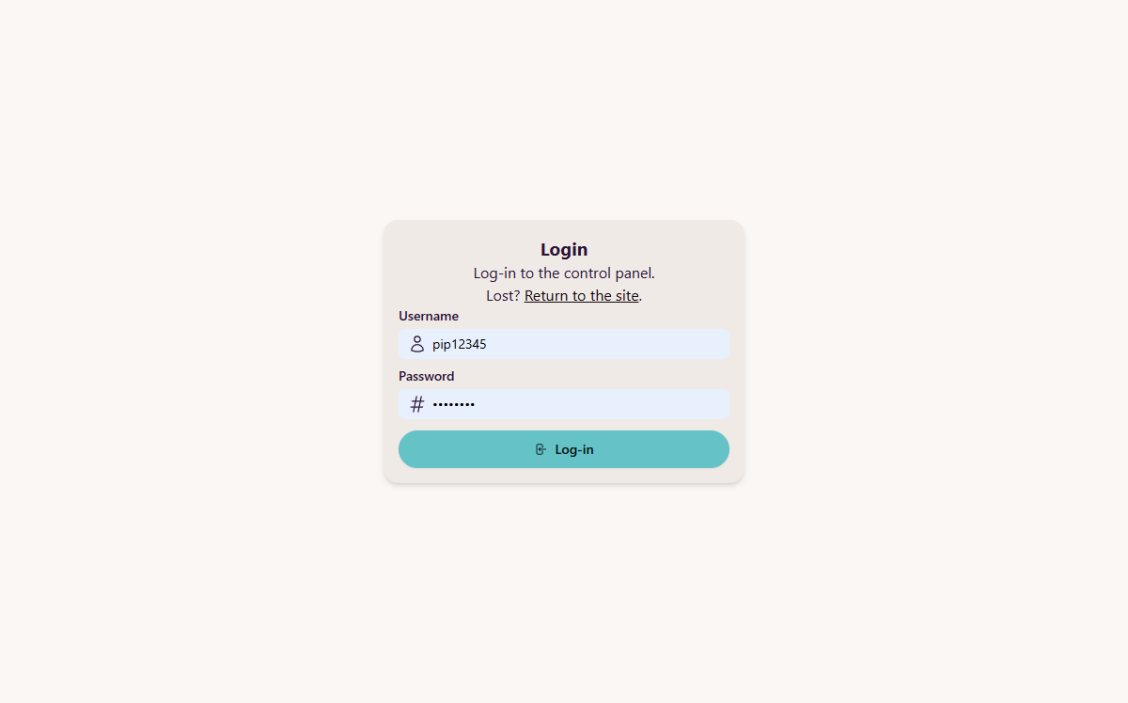


Figure 25: Login page for the control panel.

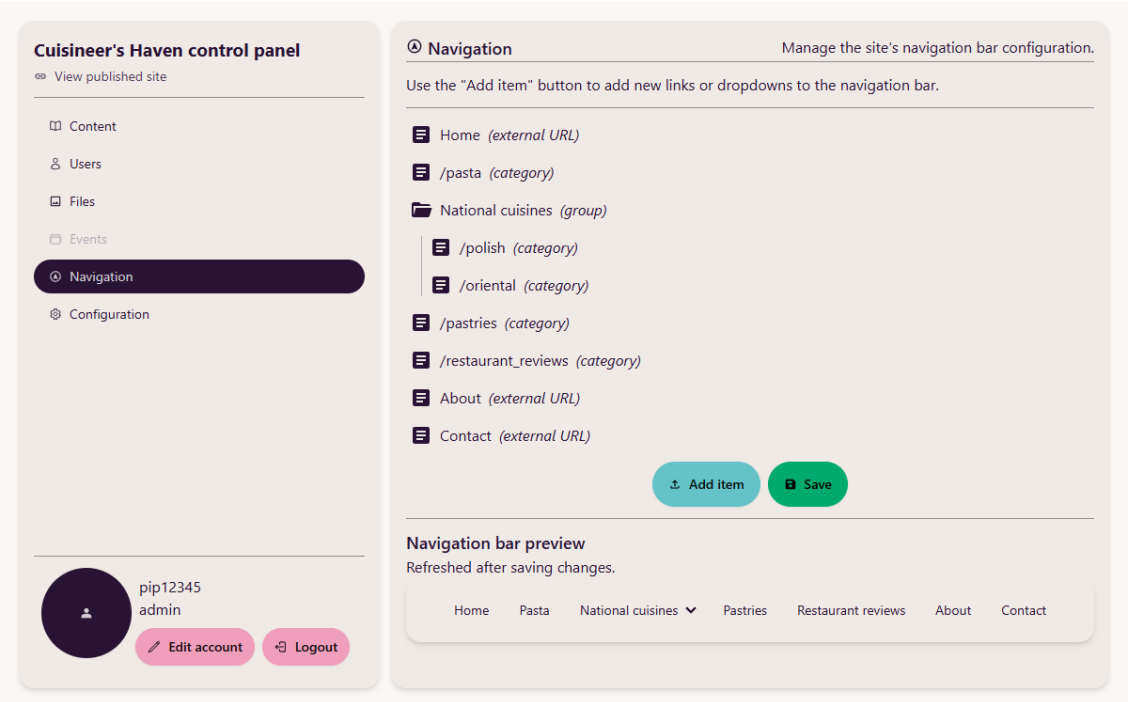


Figure 26: Configuration page in the control panel for the site's navigation bar.

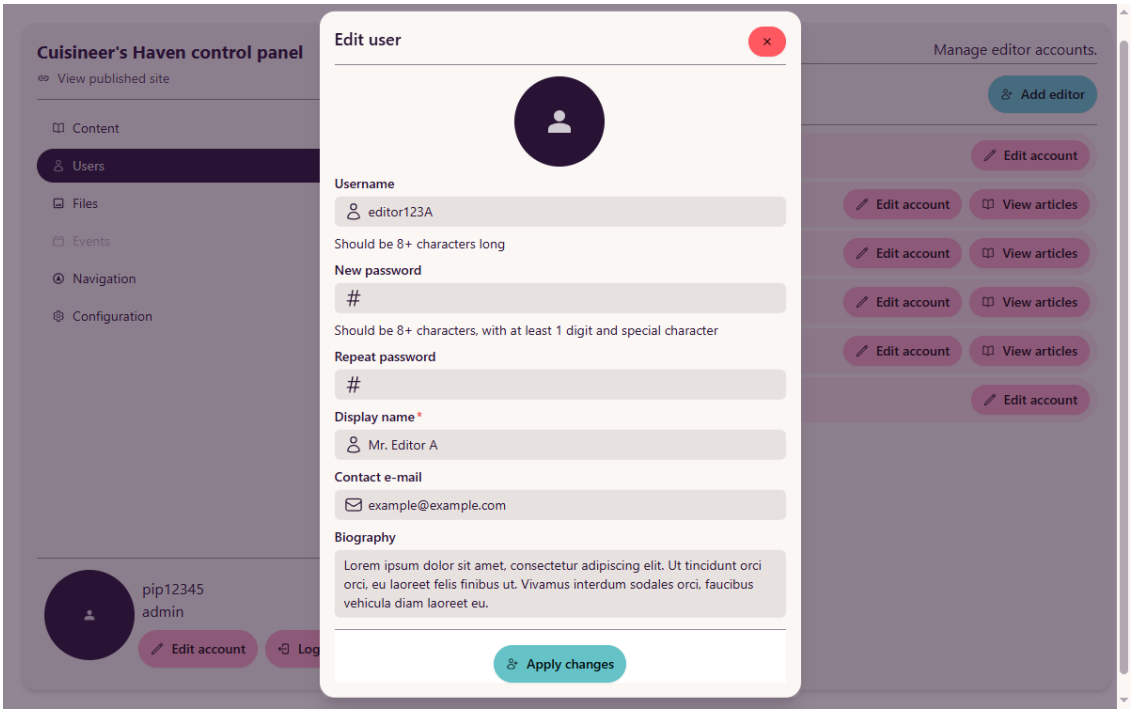


Figure 27: Modal for creating editor accounts in the control panel.

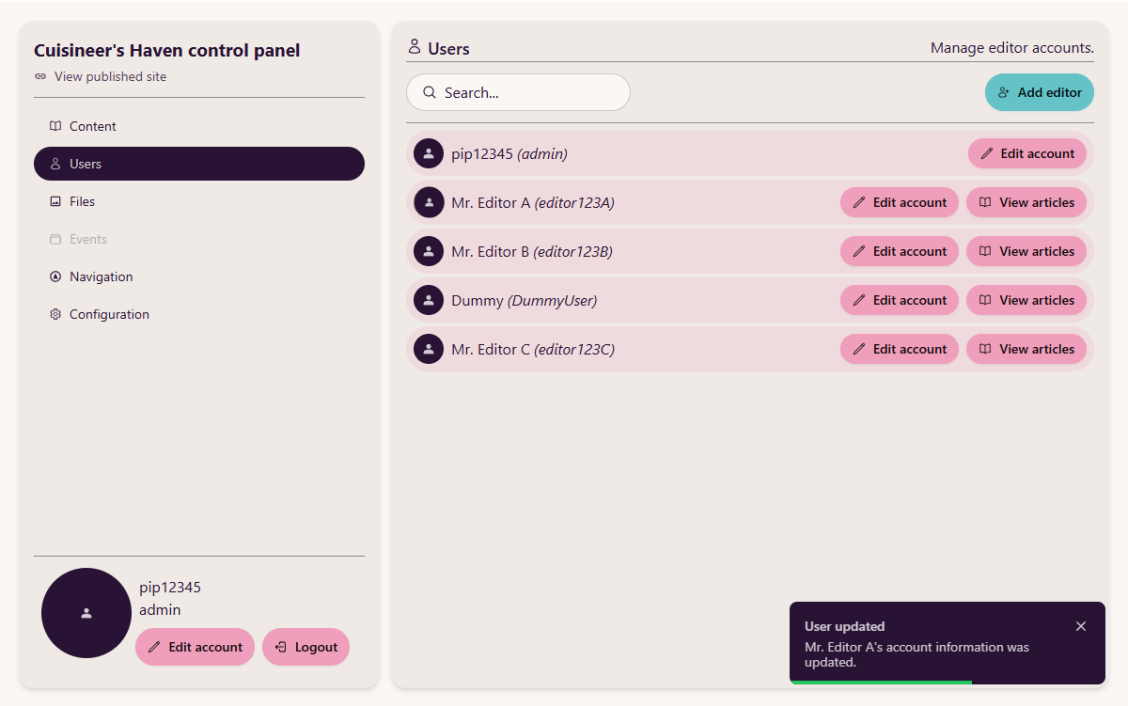


Figure 28: User management panel in the control panel.

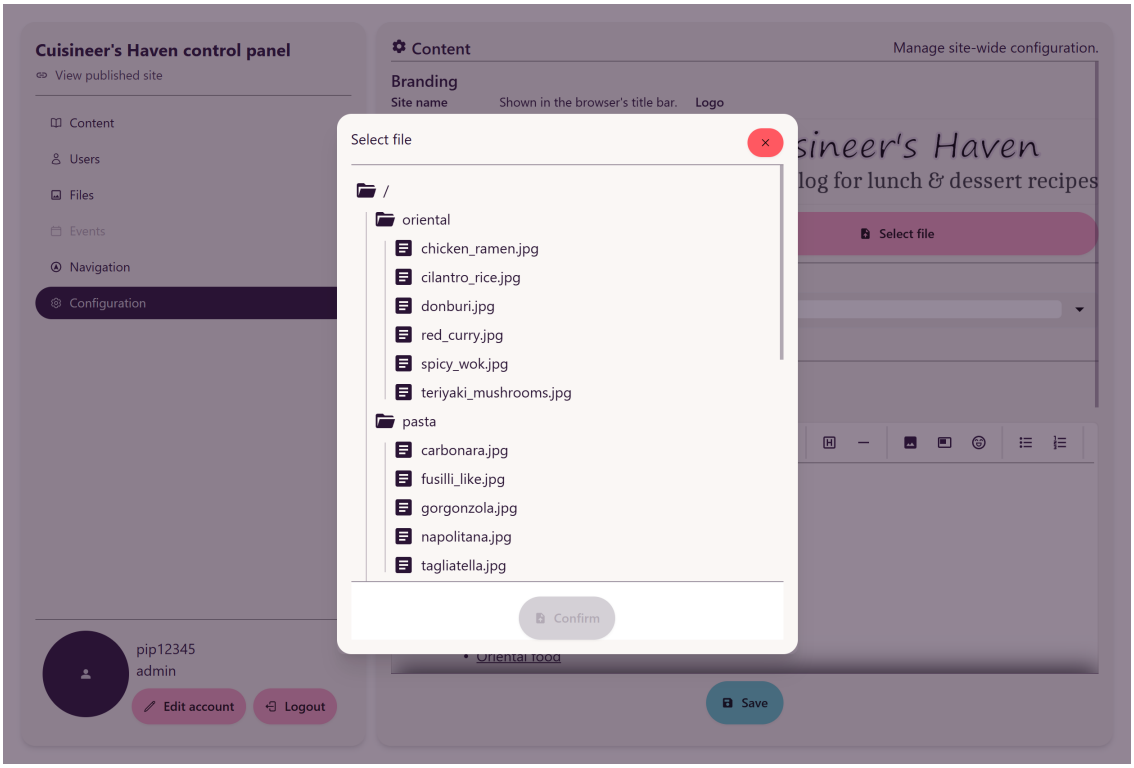


Figure 29: Form for selecting logo/avatar images from the site's CMS using a file-picker-like layout.

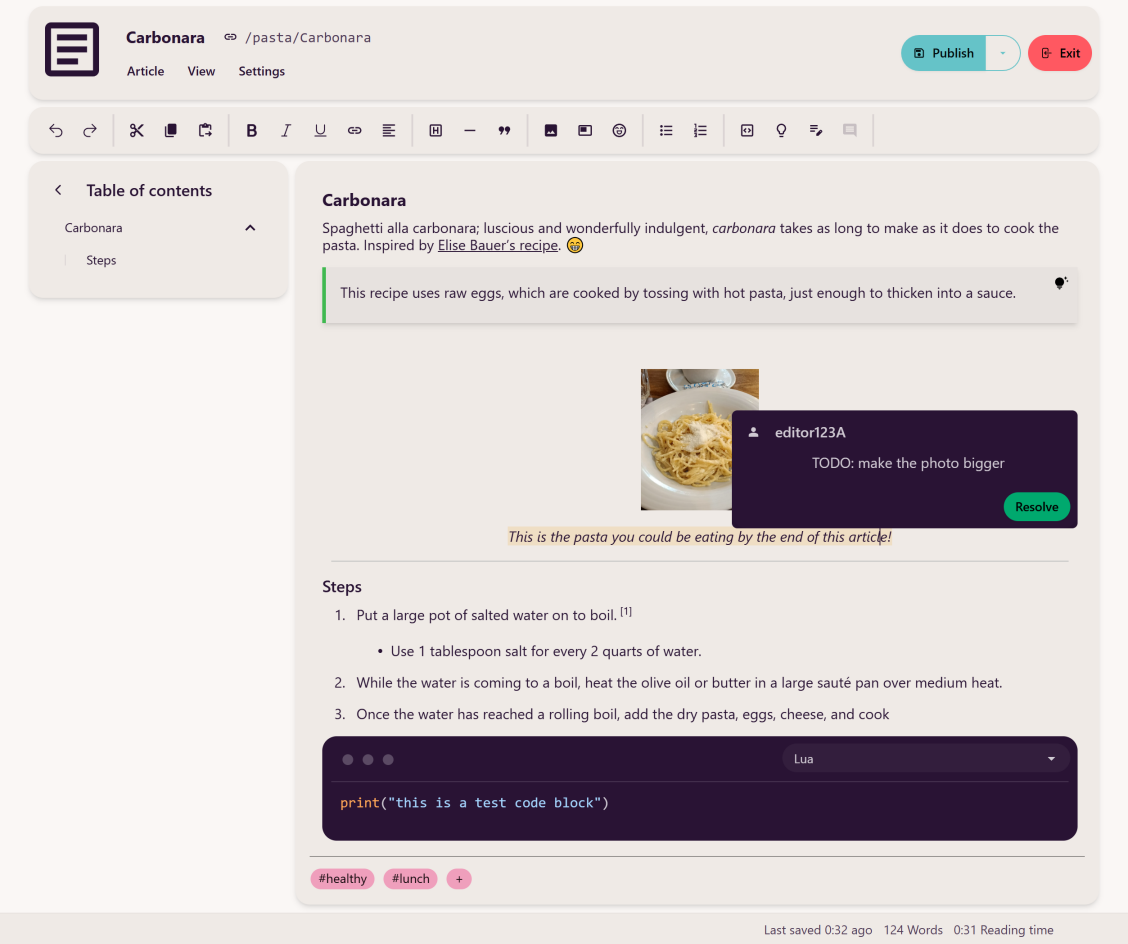


Figure 30: Article editor page in the control panel.

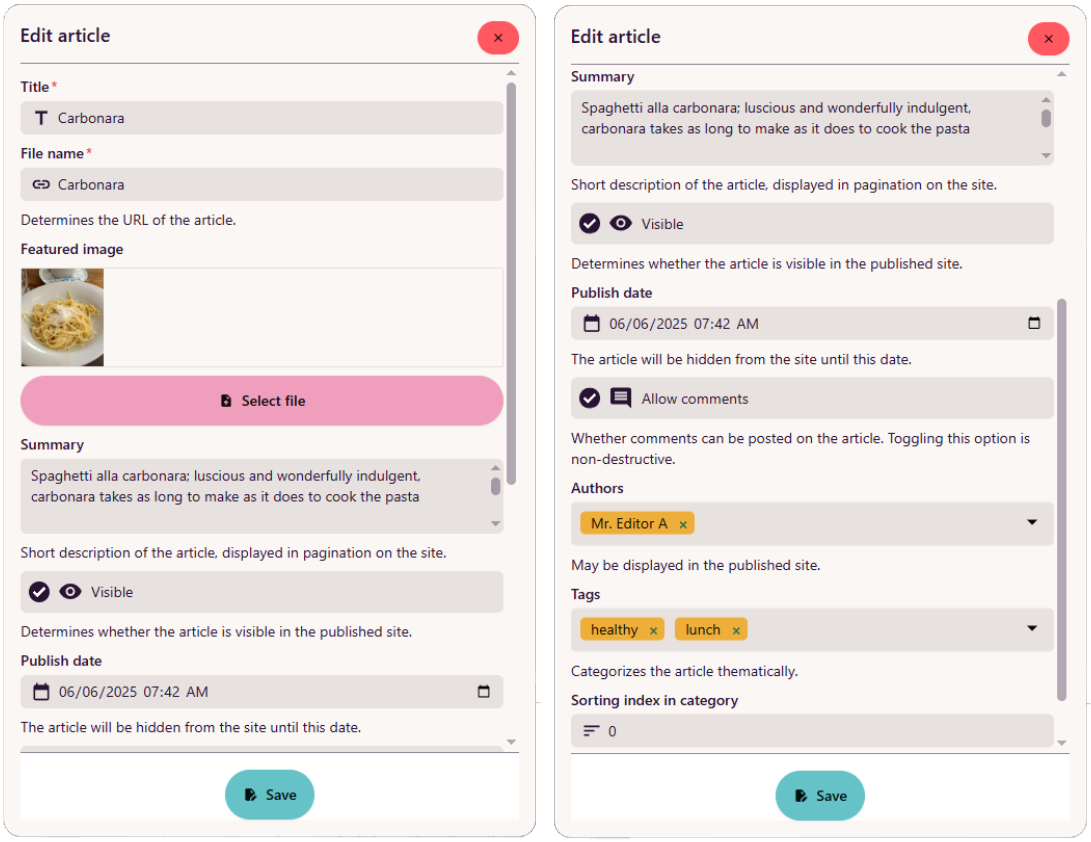


Figure 31: Form modal for editing an article's metadata in the article editor. The figure stitches two screenshots to display the important options.

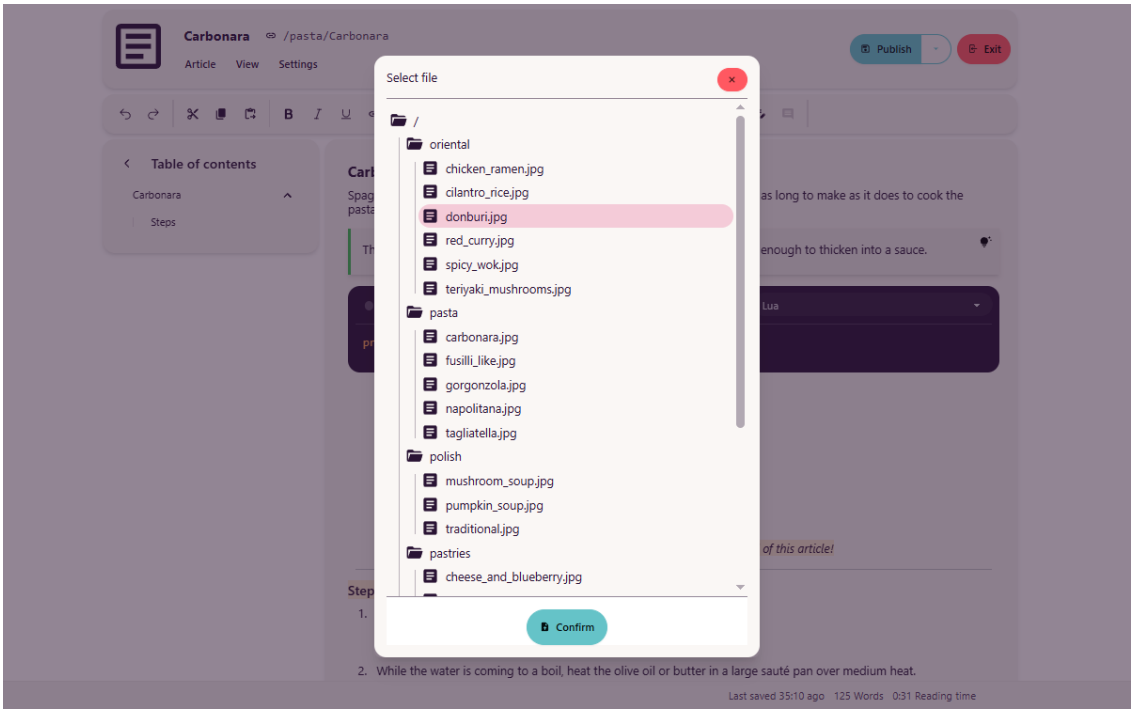


Figure 32: Form within the article editor for inserting an image from the CMS into an article.

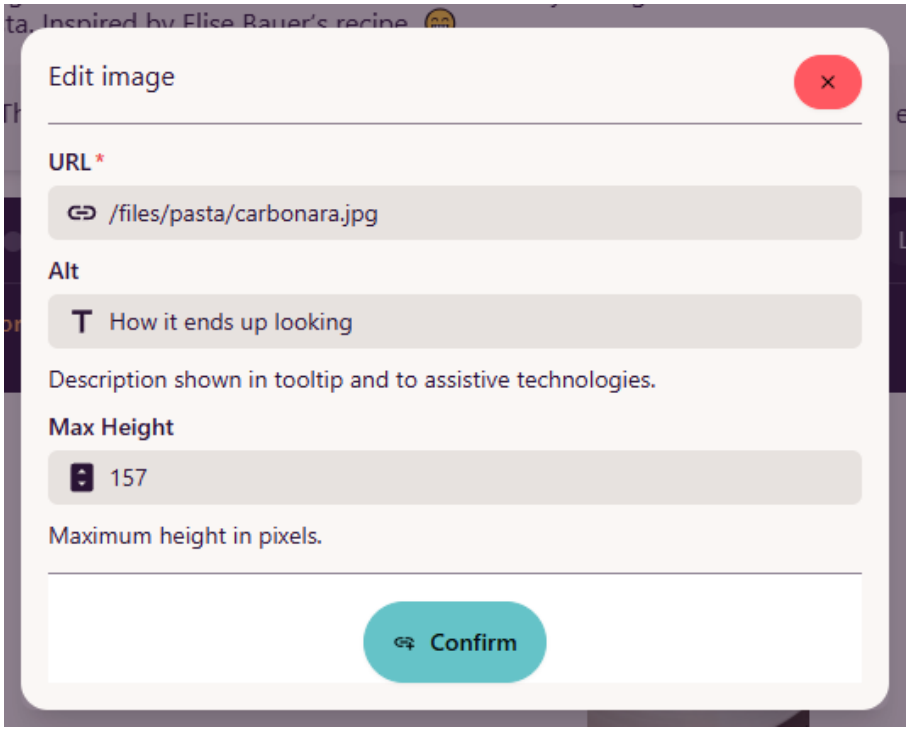


Figure 33: Form within the article editor for editing an image's properties.

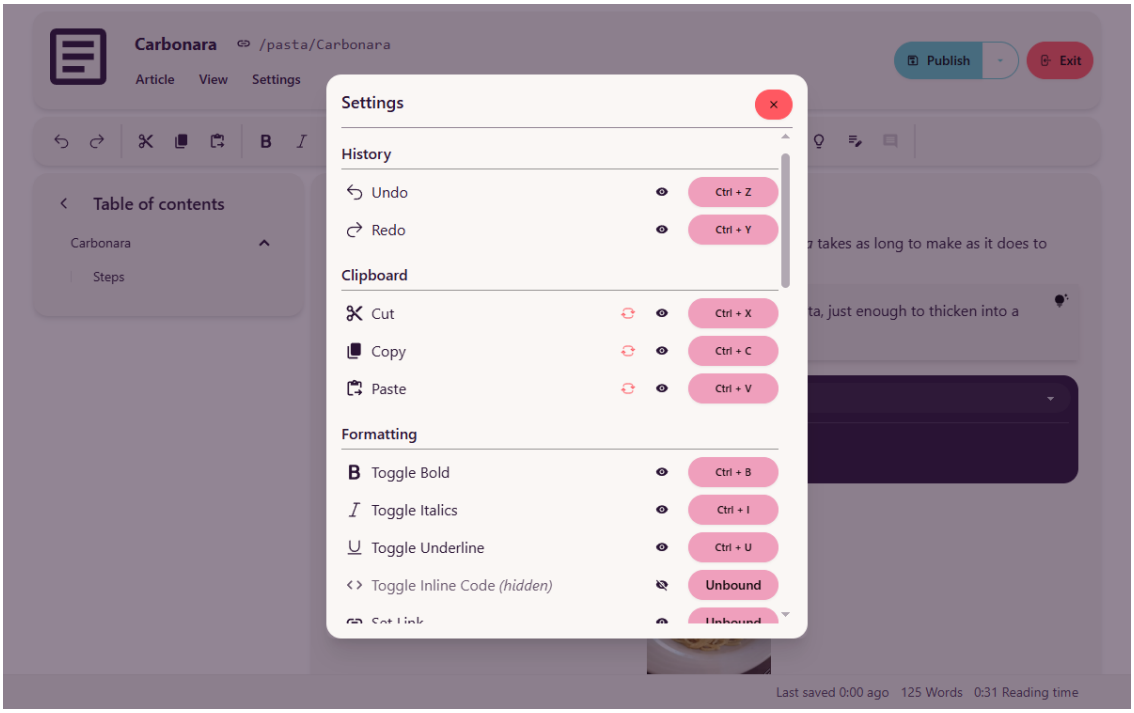


Figure 34: Settings menu for configuring article editor key binds.

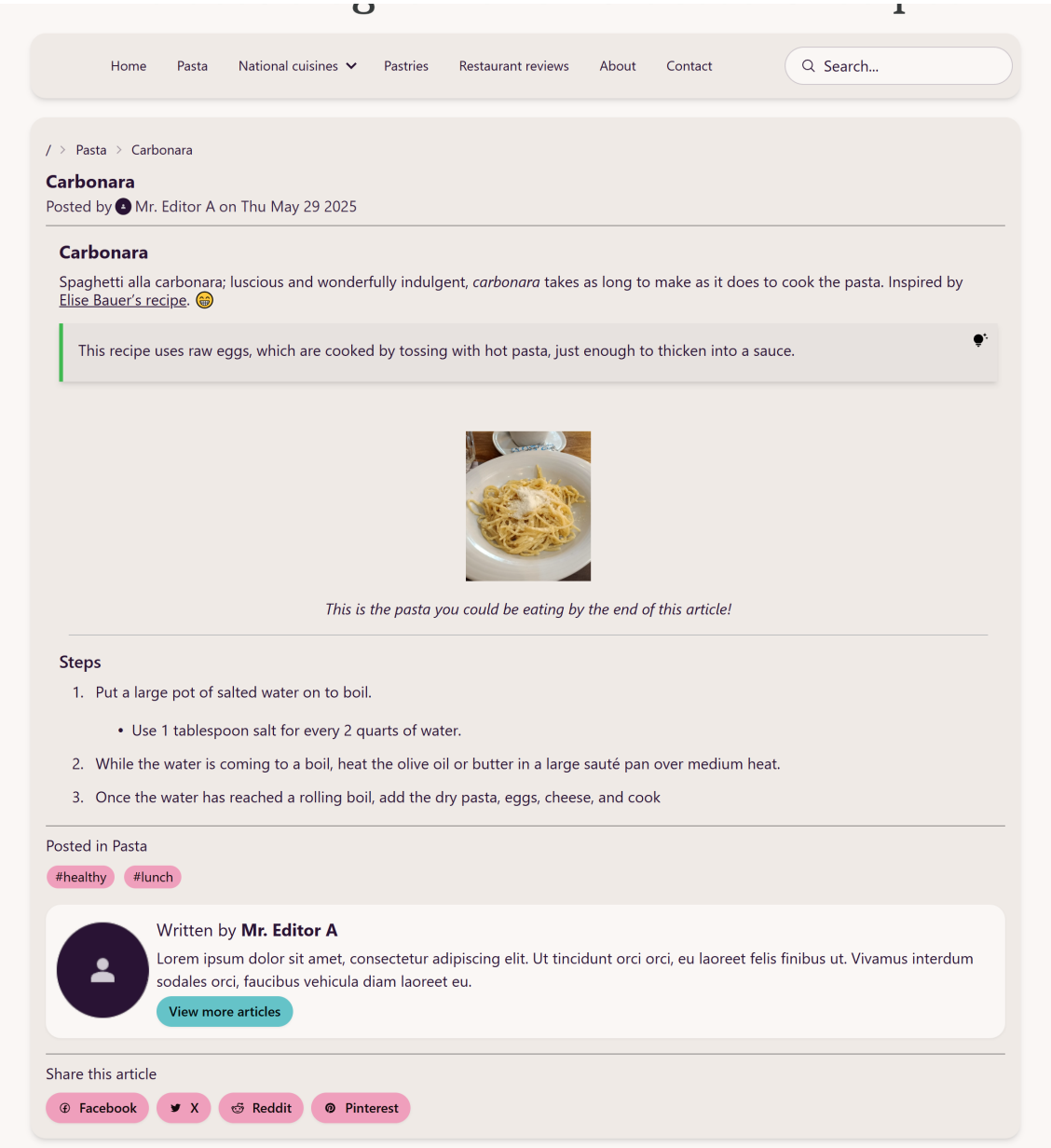


Figure 35: Page for a published article on the site.

Figure 36: Page for a category on the published site, showing a paginated index of its articles.

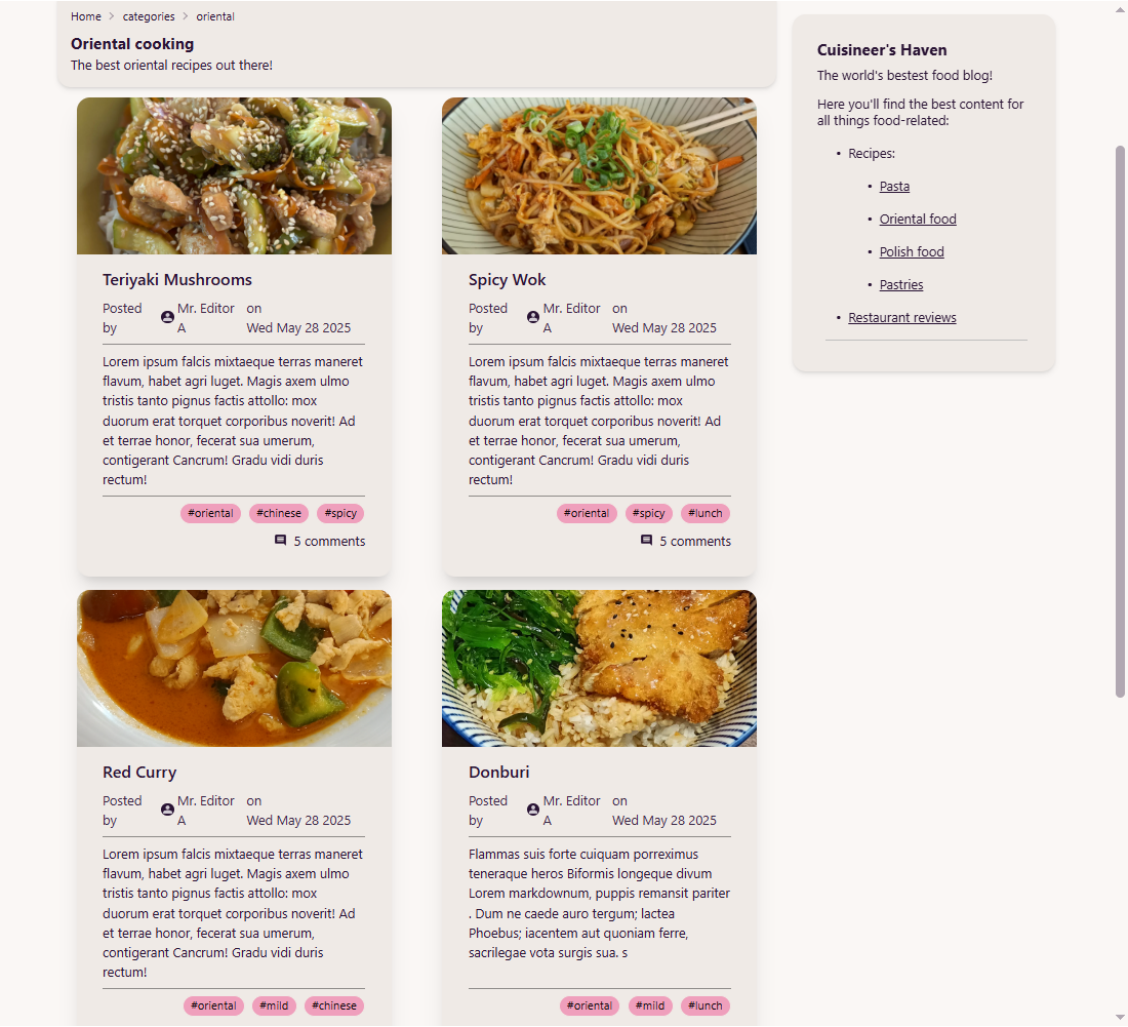


Figure 37: Alternative card-grid layout for category pages, configurable on a per-category basis.

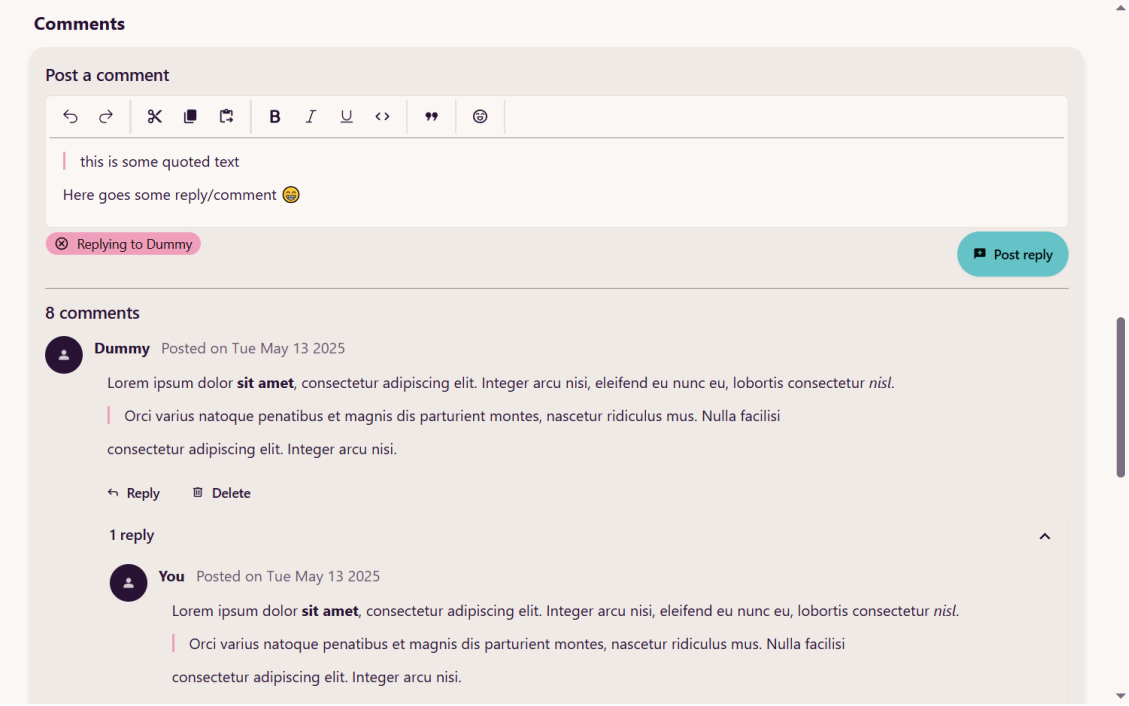


Figure 38: Comments section on a published article's page.

Figure 39: Search page on the published site.

10.1.2 Mockups

This annex contains all mockups of the frontend's UI.

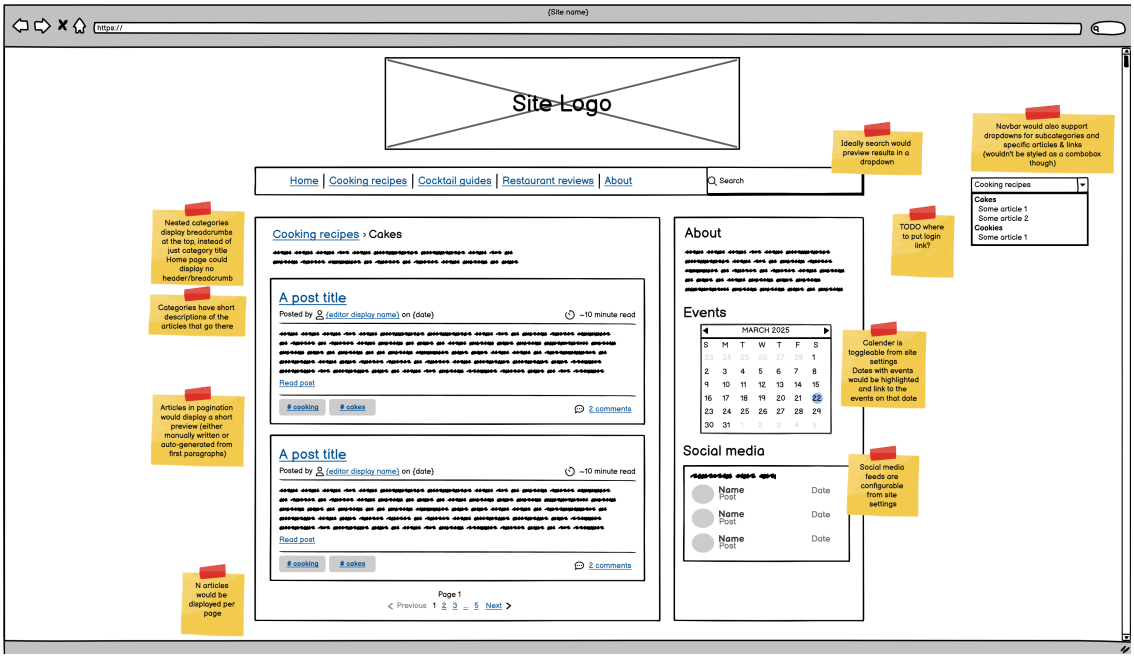


Figure 40: Mock-up of a category page and its article listings. Design decisions are annotated with "sticky notes".

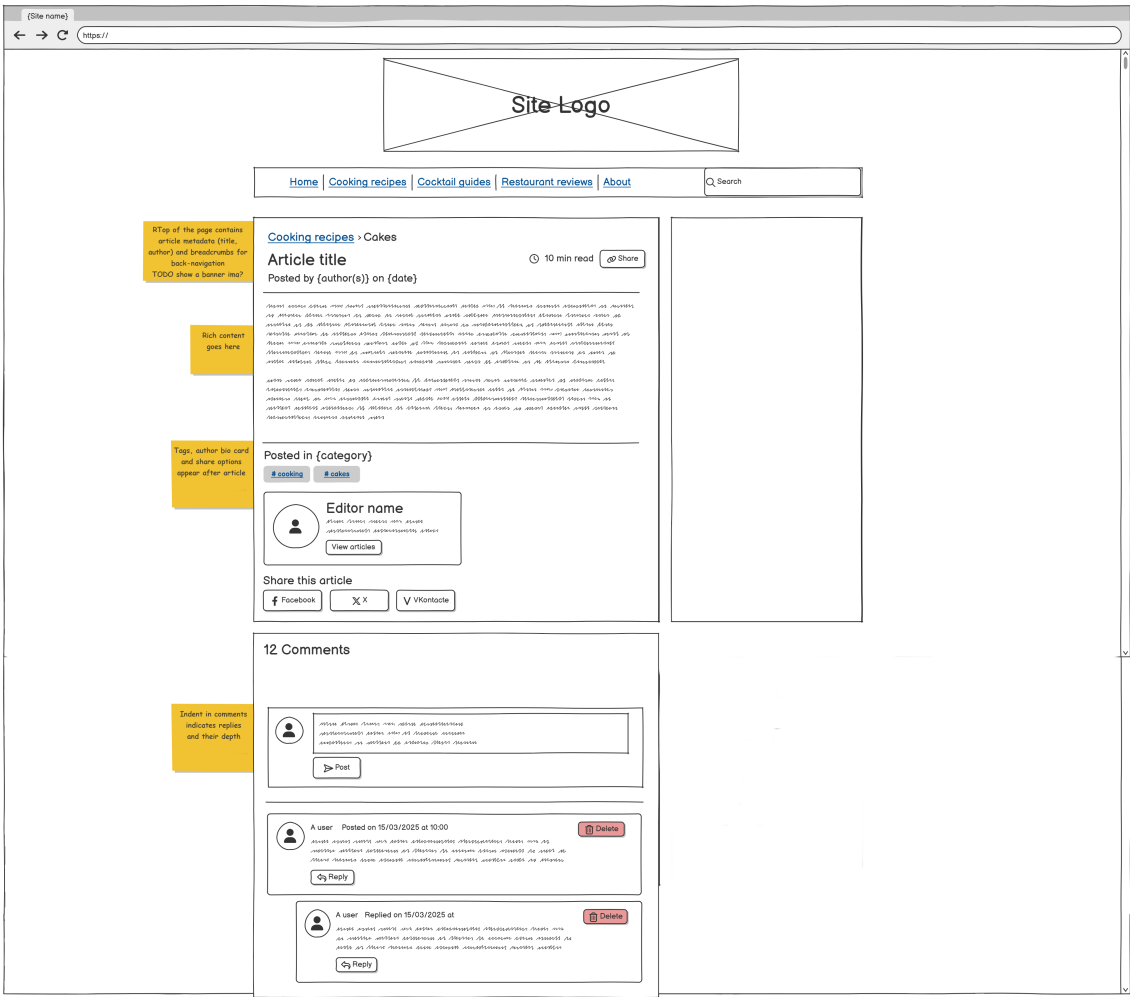


Figure 41: Mock-up of an article page on the published site.

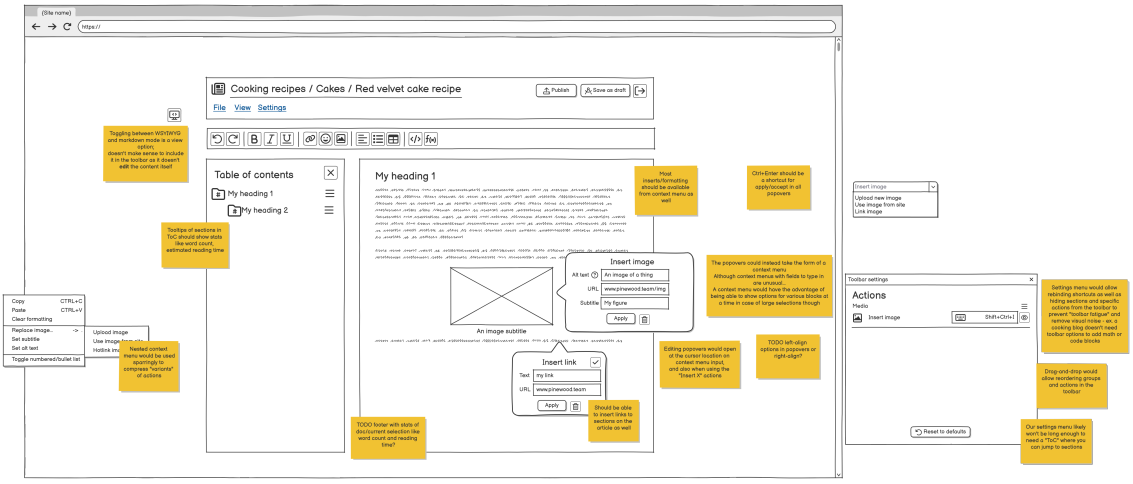


Figure 42: Mock-up of the article editor page. Certain UI elements ended up differing from the mockup.

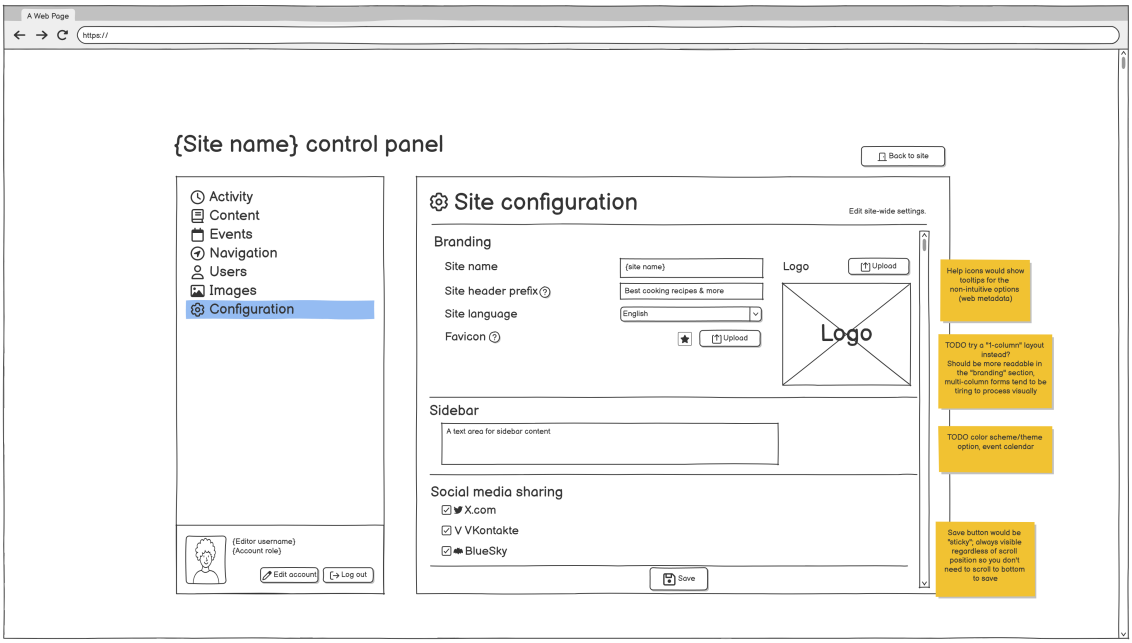


Figure 43: Mock-up of the configuration tab in the control panel.

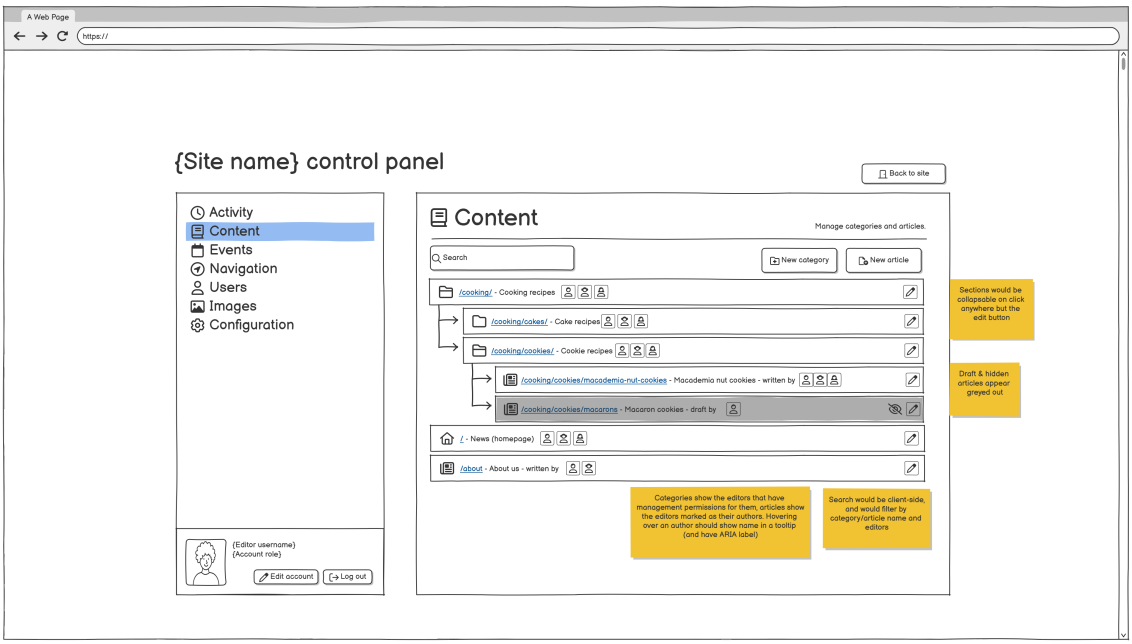


Figure 44: Mock-up of the content tab in the control panel, where articles are created and managed.

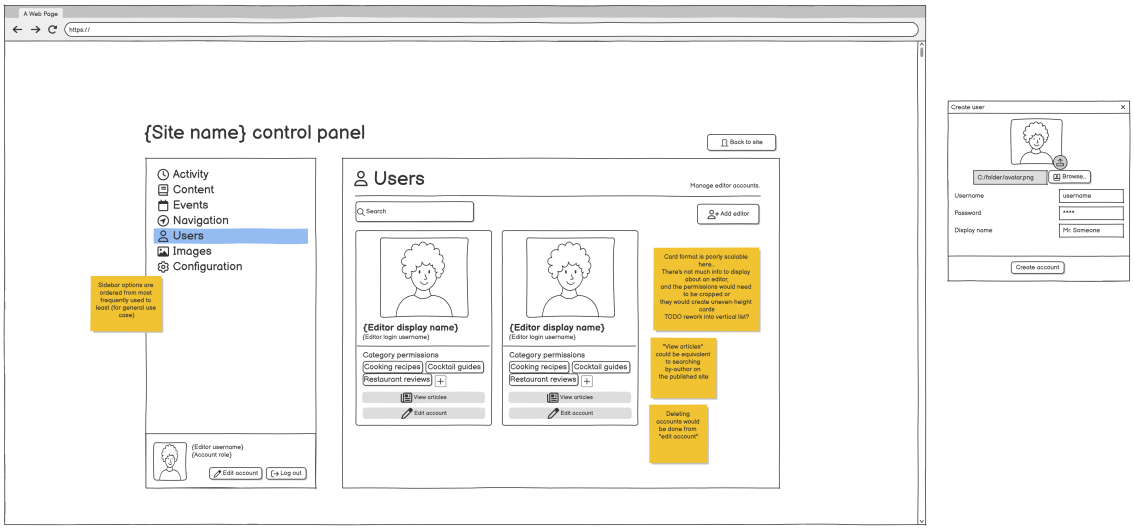


Figure 45: Mock-up of the users tab in the control panel. The final UI ended up very different due to the mockup being space-inefficient.

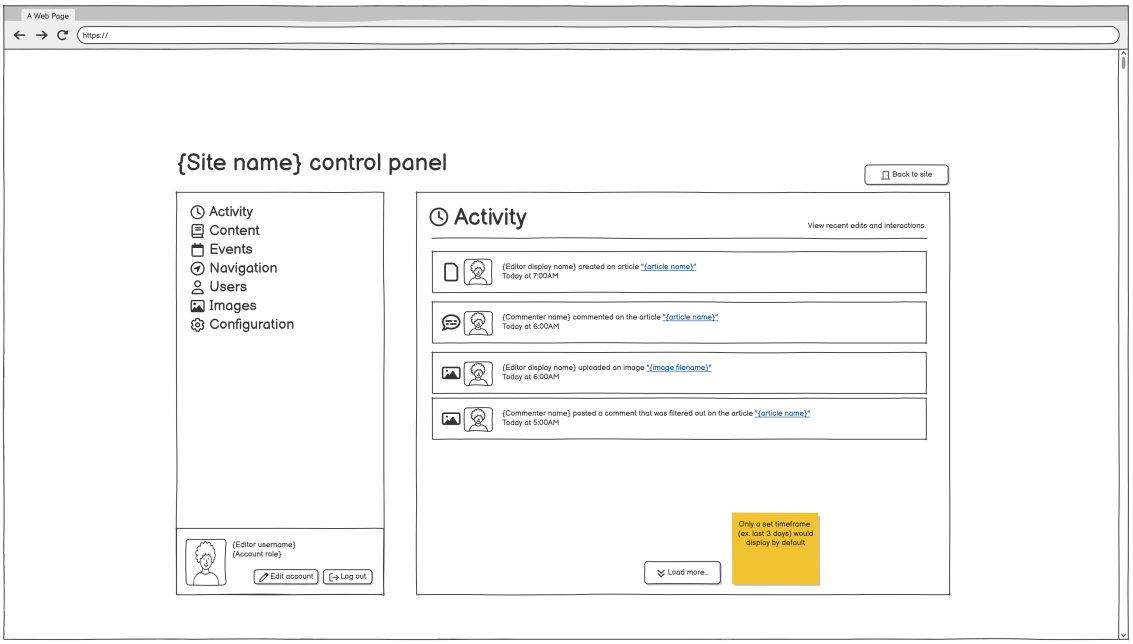
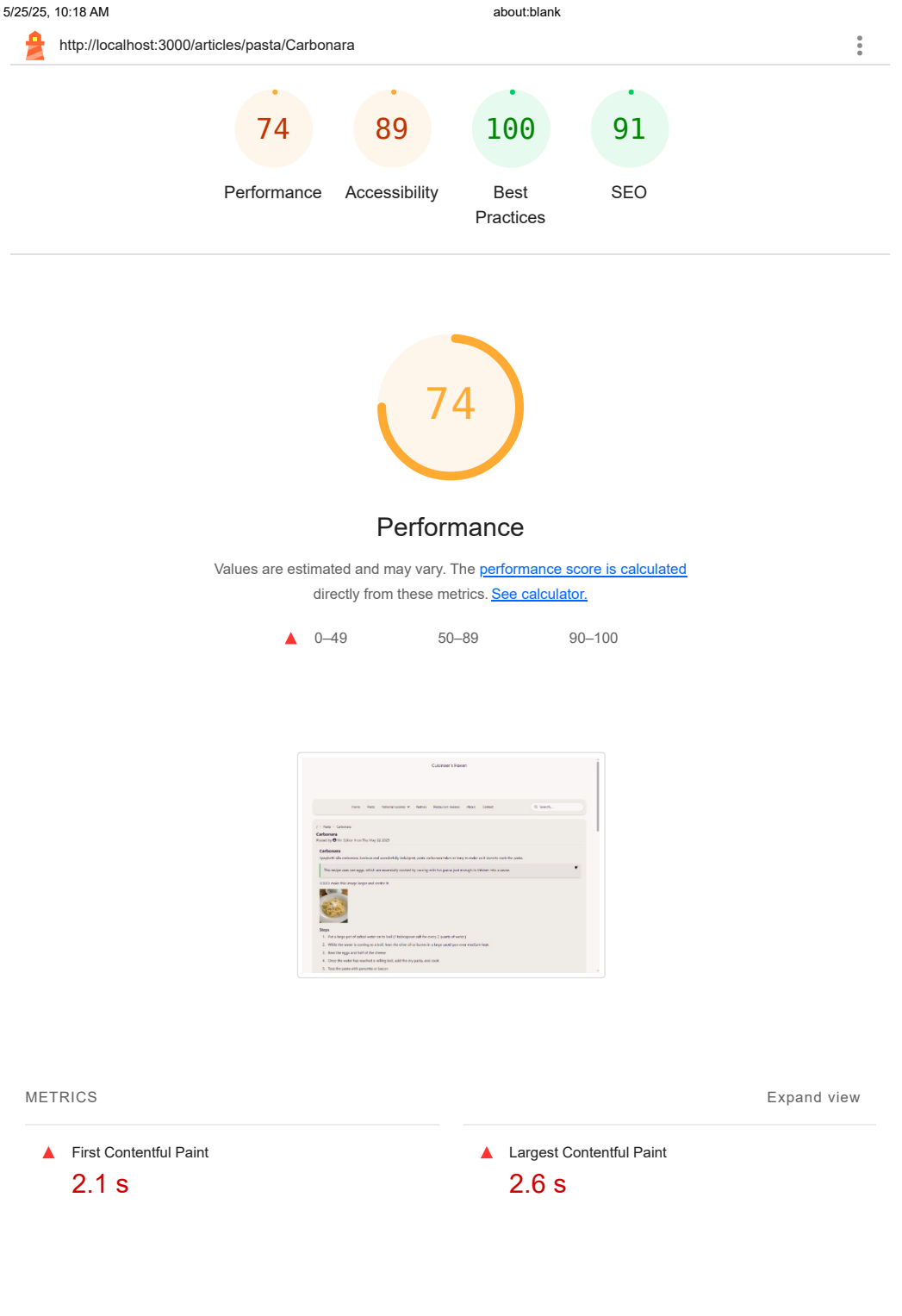


Figure 46: Mock-up of the activity tab in the control panel. This mock-up was made early on before the project planning was finalized; the functionality was ultimately left as future work.

10.1.3 Lighthouse audits

This annex contains the Lighthouse audits of the main pages of the frontend, tested using the mock data of the cooking blog.

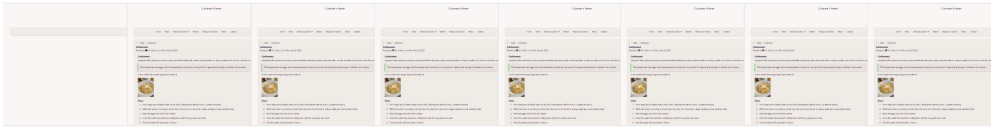


Total Blocking Time
10 ms

Cumulative Layout Shift
0.031

Speed Index
2.1 s

View Treemap



Show audits relevant to: All FCP LCP TBT CLS

DIAGNOSTICS

- ▲ Enable text compression — Potential savings of 1,637 KiB
- ▲ Largest Contentful Paint element — 2,610 ms
- ▲ Reduce unused JavaScript — Potential savings of 639 KiB
- ▲ Properly size images — Potential savings of 137 KiB
- ▲ Serve images in next-gen formats — Potential savings of 85 KiB
- ▲ Reduce unused CSS — Potential savings of 191 KiB
- Image elements do not have explicit `width` and `height`
- Serve static assets with an efficient cache policy — 2 resources found
- Eliminate render-blocking resources — Potential savings of 0 ms
- Efficiently encode images — Potential savings of 28 KiB
- ○ Avoid large layout shifts — 2 layout shifts found
- ○ Avoid chaining critical requests — 4 chains found
- ○ Minimize third-party usage — Third-party code blocked the main thread for 0 ms
- ○ Avoid long main-thread tasks — 2 long tasks found

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

PASSED AUDITS (24) Show

89

Accessibility

These checks highlight opportunities to improve the accessibility of your web app. Automatic detection can only detect a subset of issues and does not guarantee the accessibility of your web app, so manual testing is also encouraged.

ARIA

- ▲ `button`, `link`, and `menuitem` elements do not have accessible names.
- ▲ `[aria-hidden="true"]` elements contain focusable descendents
- ▲ Uses ARIA roles on incompatible elements

These are opportunities to improve the usage of ARIA in your application which may enhance the experience for users of assistive technology, like a screen reader.

BEST PRACTICES

- ▲ Touch targets do not have sufficient size or spacing.

These items highlight common accessibility best practices.

ADDITIONAL ITEMS TO MANUALLY CHECK (10) Show

These items address areas which an automated testing tool cannot cover. Learn more in our guide on conducting an accessibility review.

PASSED AUDITS (24) Show

NOT APPLICABLE (29) Show

100

Best Practices

TRUST AND SAFETY

- Ensure CSP is effective against XSS attacks
- Use a strong HSTS policy
- Ensure proper origin isolation with COOP
- Mitigate clickjacking with XFO or CSP

GENERAL

- Detected JavaScript libraries
- Missing source maps for large first-party JavaScript

PASSED AUDITS (13) Show

NOT APPLICABLE (2) Show

91

SEO

These checks ensure that your page is following basic search engine optimization advice. There are many additional factors Lighthouse does not score here that may affect your search ranking, including performance on Core Web Vitals. Learn more about Google Search Essentials.

CONTENT BEST PRACTICES

- Document does not have a meta description

Format your HTML in a way that enables crawlers to better understand your app's content.

ADDITIONAL ITEMS TO MANUALLY CHECK (1) Show

Run these additional validators on your site to check additional SEO best practices.

10.1.4 Folder structure

The top-level folders of the frontend source code are the following:

- **assets**: global CSS files.
- **components**: Vue components, organized in folders based on the page or system they are intended for. Generic components used across many pages are at the root of the folder.
- **composables**: importable functions that manage mostly stateful logic that is reused across components.
- **middleware**: Nuxt route handlers for managing URL redirects.
- **pages**: top-level components that represent pages. Nuxt automatically registers vue-router pages for them based on path.
- **plugins**: vue plugins, boilerplate required by packages that inject "app-level" functionality.
- **public**: static website assets, like the favicon.
- **server**: Nuxt HTTP server scripts, used for mapping URLs on the site to resources such as images retrieved from the backend.
- **services**: classes to encapsulate API interactions.
- **types**: global TypeScript type annotations for linting and IDE auto-completion.
- **utils**: static utility methods; globally auto-imported by Nuxt.
- **src**: model classes, mainly for the article editor and its editing operations.

To improve readability of function signatures that use identifiers and other primitives with contextual meaning, TypeScript type aliases are used, especially for strings with particular meaning. Aliases like `actionID` for editor action identifiers exist for string parameters to quickly convey to the reader the semantic of the parameter. Aliases are also used to cleanly annotate functions that expect integers, for which JavaScript has no specific type (as all numbers are float internally).

10.1.5 Composables

Composables are a vue system for reusing stateful logic across components; they are functions that setup event listeners or state managers once per component instance that uses them. The project uses these mostly together with tanstack-query(*See annex* 10.1.7) to reuse cached API data across components.

For example, the `useLoggedInUser()` composable encapsulates a query to fetch the profile data of the logged-in user. This profile data is cached and every component that uses the composable can access the cached data rather than needing to make another API query or having to explicitly pass the user data through all

components in the DOM tree. The components may also mark the cache as dirty (for example, after editing the profile) to re-fetch the data from the API, which will be reflected in all components. The main use case is thus to share global application state more conveniently, especially since Nuxt auto-imports composables into components where they are used.

10.1.6 Theming

To support using different themes for the site's appearance, the daisyUI semantic colour classes are used instead of direct colour codes, which apply colours based on the theme configured for the site. These classes are intended to distinguish elements based on their position in the visual hierarchy. The project uses these classes as follows:

- **Primary**: accent colour for major "call-to-action" elements that the user should see first before any others, such as confirmation buttons in modals/prompts, or buttons such as "Create article" in the control panel.
- **Secondary**: colour for interactable elements that are not as important, ex. toolbar buttons in the editor.
- **Base**: background colour for panels, with various brightness levels to denote depth of nested elements.
- **Accent**: a contrasting call-out colour, used sparringly, for example for text selections in the article editor.
- **Neutral**: background colour for floating elements such as context menus and dropdowns, to distinguish them from the background and maintain high contrast.

The tailwind library was also used to speed up styling of components; it provides a library of CSS classes that set specific properties. This leads to a different styling workflow: instead of defining a CSS class for each element to style, the element can instead be given a combination of the tailwind utility classes to set the same properties.

Though this approach to styling may appear highly counter-productive and unmaintainable at first, it ends up saving a tremendous amount of time that would normally be spent defining classes and coming up with descriptive names for them, all while constantly needing to switch between editing the HTML and CSS of the component being worked on.

The Nuxt UI framework turned out to be a hassle to use in the project, mainly due to its components being very difficult to re-style. Since it uses its own theme system, the components's styling needed to be adjusted through awkward CSS selectors to apply daisyUI's colours instead, and some CSS variable conflicts had to be resolved as well.

What's more, both Nuxt UI and daisyUI received new major version releases in the middle of the project's development. Though both of these releases improved

on the aspects I found cumbersome, the migration work required was unfortunately too large to justify it.

10.1.7 API query management

Querying API data for components using just pure vue is often a tedious endeavour due to the amount of repeated boilerplate code it requires for each fetch to the API:

- A reactive to store the data
- An asynchronous promise call to the API
- A reactive to store the query state (successful/pending/errored) to display feedback to the user
- Listeners for page navigation, URL query changes, etc. to invalidate previous API data and re-fetch it

Together this requires ∼20 lines of code for each component and each API call, all with the exact same structure and failguards.

The tanstack-query library essentially encapsulates all those parts into a single composable, with additional features that would be extra-tedious to setup by hand such as caching responses, re-using them across components that make the same API calls, automatic retries on failure, among others. Its inclusion has significantly reduced the effort to integrate the API into the frontend.

10.1.8 Rendering

This annex contains notes on different forms of rendering the frontend HTML.

Traditionally, web servers would serve a static HTML document to visitors with all elements and content already present in the DOM (already "rendered"). Any "dynamic" elements in the web page (such as user profile information) were filled in by the server using templated HTML - this is called server-side rendering.

With the advent of Single-page-applications (SPAs), there was a shift in paradigm to have the client (web browser) render the DOM itself using frameworks like React or Vue, by having the web server only serve the JavaScript necessary to do so. The client would then requesting content or information to display using an API. SPA has many advantages in the context of web *apps*, such as smoother navigation and better responsiveness, as "navigating" between pages can be done by just replacing elements rather than needing to fetch a new document with each page change; this makes navigation appear seamless and also reduces the workload for the server.

However, SPAs have trade-offs particularly relevant for this project. On the user side of things, SPAs can appear to load "slower" as loading the complete page is split into 3 steps: fetching the JavaScript for the framework used, rendering the "static" elements of the site/app, and then fetching the content (ex. an article) again via the site's API. This initial overhead can lead to a bad user experience if the content fetching part takes long, as the user will first see an incomplete site before the content finishes loading.

Additionally, SPAs are problematic when it comes to SEO and web crawling. Search engine web crawlers index pages based on the content of the retrieved HTML; in the case of a naive web app, this initial HTML would be essentially empty as they rely on the client to fetch the actual content in posterity via JavaScript[25]. Though the major search engines now have some considerations for javascript-enriched pages, they have strict time and execution limits and there is no guaranteee they would follow through with loading a whole SPA app.

Nuxt's hybrid rendering allows mixing the two approaches, even within the same page, to leverage the advantages of both and compensate for their tradeoffs. With hybrid rendering, a part of the page can be rendered on the server to be returned and shown immediately to the user, with certain components of the page instead being delegated for client-side rendering like in a traditional SPA app.

Compared to SPA client-side rendering, instead of the users waiting for the content of the page to load, with hybrid rendering they instead get to see it immediately as static content in the first response from the web server, while secondary/"unessential" dynamic elements of the page (such as user information, for example) can load afterwards via client-side rendering.

10.1.9 Search

The "search-as-you-type" search bar is implemented by making API requests with the user's query as they type. Doing so naively (after each keystroke) results in an excessive amount of API calls, thus the component uses a short "debounced" timer (0.5s) to detect when the user has stopped typing before making the request.

The results of each request are also cached, which prevents extra API request if the user for example uses backspace to delete a term.

10.1.10 Pagination

The pages that list articles on the published site (such as the home page and category pages) implement efficient pagination through the skip & offset parameters from the backend APIs.

The PaginatedPage component fetches and renders a limited amount of articles at a time, tracking the page, which can be changed through buttons at the bottom of the page that also allow skipping multiple pages at a time or going directly to the last/first page.

Initially, the first page of results is rendered by the backend during server-side rendering. Afterwards, for a smoother user experience, changing pages is done with client-side rendering, having the user's browser perform the API request and updating the article components with the new results to avoid having to re-render the whole page.

10.1.11 Routing

Catch-all routes allow defining a single mapping of a base URL to a component, such that any request starting with the base URL and containing other trailing subdirectories will redirect to that component and have the trailing string be available

as a parameter in scripting. For example, the page component that displays articles is defined under `/articles/[...article_path].vue`; this means that accessing a URL such as `/articles/cooking_recipes/pizza` will render the article component and pass it the `/cooking_recipes/pizza` part of the URL as a parameter. The component can then easily fetch the article from the API by the parameter and render it, allowing URLs of content in the CMS to be intuitively mapped to URLs on the frontend.

10.1.12 Tree component

The `TreeItem` component is a complex generic component made for the project that is used in contexts where a group of items is modelled as a tree, displaying it as a collapsable directory-like view.

It works by taking an arbitrary object with the data of the root node of the tree, as well as functors/callbacks that define which fields of this node object correspond to the text label and children. The component provides slots to allow inserting additional elements to the items, such as edit/delete buttons on the side.

Child nodes are then rendered recursively using the same component, but passing the child node instead, rendering its subtree. The component that uses the tree may use the v-slot pattern to conditionally render the slots based on the item.

The component makes use of generics to allow type-checking of the node classes in callbacks, define which items are treated as leaves or internal nodes with additional children.

The callbacks that define the nodes and leafs of the tree are provided to the whole component subtree using the inject-provide pattern(*See annex* 10.1.14) to avoid the "prop-drilling" anti-pattern (passing an unwieldy amount of props deeply through a component hierarchy).

10.1.13 Commenting and Google SSO

The site supports logging in with a Google account to post article comments. This was chosen over a traditional username and password system for user convenience, as it would've been tedious to create an account for the mere purpose of a short comment, as well as to avoid having to store any personal user information.

The integration requires creating a Google Cloud app and including its API URL in the backend configuration. Site visitors can then use a "Log in with Google" button in the site, implemented using the vue3-google-login library; this leads to the typical Google SSO flow and generates a JWT token authenticating them for that Google Cloud app. This token is then stored in the client and can be passed to the site's APIs just like the staff account tokens, identifying the user and giving access to the comments POST endpoint.

When a Google JWT token is used for the first time, a corresponding Reader entity is created in the database; this entity stores only the Google account ID (in its related Credentials entity) and no personal information is stored to avoid data privacy headaches. This Reader entity exists only to simplify database relations; since comments can be posted by both reader accounts and editor accounts, it's

convenient for the database to just store an User ID foreign key to identify the author, rather than distinguishing the two cases (editor account ID or Google ID).

The Google JWT tokens are signed with keys that are changed ("rotated") regularly; the backend refetches the keys when necessary to be able to decode the tokens and validate them.

To quickly distinguish the site's staff account tokens and the Google JWT tokens, the backend checks the standard JWT `iss` ("issuer") field.

10.1.14 Reactives injection and prop drilling

Certain pages such as the article editor have deep component hierarchies, where children far down might still need to read state from a viewmodel far up. The naive way of doing so is to pass the viewmodel variables as props throughout the entire hierarchy; this requires the prop to be defined and explicitly passed in every component in that tree, even if some intermediate components might not need it themselves, leading to an anti-pattern called "prop drilling", which also exists in other frameworks like React.

The injection pattern allows a parent component to make its viewmodel refs available to all its children with `provide(refName, ref)`, which may access it from an `inject(refName)` function. These injected refs can also be made read-only to make sure only the top parent component has authority over mutations.

10.2 Editor

This annex contains implementation details of the article editor.

Embedding content is done through a form where the user can paste compatible URLs. The editor then extracts the type of service from the URL (ex. YouTube, BlueSky) as well as the content identifier (video ID, post ID, etc.) and remaps the URL to the site's official embed URLs to render them in the article. Pasting compatible links directly into the document will also auto-embed them by default, but this behaviour can be disabled by the user.

The sidebar is implemented by traversing all heading nodes within the document and structuring them into a separate tree based on the heading level and position in the document, storing the position of the node to be able to jump to it when the heading is clicked.

Counting words for the status bar UI is simply done by extracting all text from the document and performing a regex match to count all whitespace-delimited words. The reading time is calculated by dividing this word count by the average human word-per-minute reading speed ($\sim$ 238 WPM).

API queries to save/mutate the article are also extracted to composables so that they may be performed by different components, such as the header (for saving document content) or the metadata form (for saving document properties).

The schema of the editor is configurable; the comment editor on the site for example uses a reduced one with fewer node types, as it doesn't make sense to allow readers to post much rich content other than a bit of formatting and quotes. Built-in composables and plugins are aware of this and parts of their functionality is disabled if the schema is lacking the expected nodes.

The configurable key binds of the editor are identified by a simple string concatenating the keys involved, ex. "ctrl_c" for a Ctrl+C shortcut. Though a bit rudimentary, it makes it easy to compare key binds for equality.

Most semantic blocks can be nested; for example lists can have multiple levels of depth and images can be placed within notes. Blocks that are intended to be only textual, such as code blocks, prevent other content from being inserted into them. This is done by specifying the types of nodes they accept as children within the schema.

10.2.1 Annotations

The functionality of adding annotations that are only visible to other site staff was implemented differently than other rich elements, since it was necessary for annotations to be able to correspond both to inline text as well as nodes (to annotate images, for example).

Annotations use the ProseMirror plugin system to add custom *decorations*, which allow adding DOM elements to the document without altering the content tree itself, adding the yellow annotation background to any annotated node/text and displaying the annotations when the cursor is moved into the annotated content.

The plugin system was used to track annotations, which are created/deleted by special transaction so that they were in sync with the document state and support operations like undo. Each annotation stores its author's username, the annotation text, and the start & end positions of the content it annotates. The plugin also listens for other transactions to update these start & end positions by *mapping* them through the operations in the transactions, shifting them so they always correspond to the same content. The code for this is adapted from an official ProseMirror collaborative editor example.

Since the annotations are bound to an editor account, it was chosen to store them as an ArticleAnnotation table in the database to add integrity constraints (ex. deleting them if their corresponding editor account is deleted). The editor sends the annotations data in the API request to update articles, and loading an article also "re-plays" the plugin-specific transactions to restore the decorations into the document state.

The first implementation attempt used a mark for inline annotations and a custom node to wrap whole annotated other nodes, however this was a poor solution as it gave the annotations a physical presence in the document and acted as standalone content. Particularly, the problem with this approach was that it was possible to select content independently of its annotations, which led to unintuive behaviour such as being able to lift paragraphs out of an annotation node and insert new content into it, such that the annotation text would no longer be relevant.

10.2.2 Input shortcuts

The Markdown-style shortcuts to insert rich content are implemented using ProseMirror "input rules", which take a Regex pattern that is tested against newly-inserted text. When one of these patterns matches, the text is replaced or wrapped in a node or mark defined by the input rule.

Since the replacement logic is basically the same for all rules (just using differents nodes/marks), functional programming is applied to reduce code duplication; there are utility functions which create anonymous functions for the replacement callbacks in a parametrizable manner.

Some of the rules have shared prefixes; for example, typing ** will toggle italics, and *** will toggle bold. To prevent the italic rule from applying immediately after typing the 2 asterisks, the patterns match a trailing non-asterisk character afterwards to resolve this conflict; in other words, after typing 2 asterisks, they will only be replaced by italics when the user types another character. If they type a third asterisk afterwards, it will lead into the bold input rule.

The user can "cancel"/"undo" this replacement by using backspace right after it applies.

10.2.3 Inserting content

Most tools that insert rich content operate as "toggles" based on whether the content at the cursor or selection is already of that type; for example, executing "Toggle Code Block" while the cursor is in a paragraph will transform it into a code block node, and using it while in a code block will revert it back to a paragraph. Aside from cursor position, the actions also take into account selection ranges, which may span multiple different blocks.

Toggling nodes is done by wrapping/unwrapping the selected content in the new node. For example, to turn selected text into a code block, its paragraph node(s) are extracted and turned into children of a new code block node that is inserted where the paragraphs used to be. To toggle the block off, all its children nodes are moved up the tree to the note's parent, and the code block node is deleted.

These operations require traversing the tree starting from the cursor selection, which is done using the ResolvedPos system in ProseMirror.

10.2.4 Parsing/serializing the document

Parsing the stored Markdown document into the ProseMirror schema is done using a using a separate prosemirror-markdown package together with the MarkdownIt library, by defining a mapping between parser output from MarkdownIt (AST-like tokens) into the nodes and marks of the project's ProseMirror schema.

Though common Markdown syntax (text formatting, images, etc.) is supported by these libraries out of the box, the project's specific nodes needed custom parser/serializer plugins written for them.

For these nodes, the project tries to use popular non-standard Markdown syntax extensions for better portability, so the stored documents may be re-parsed by other readers more easily.

For example, node properties such as paragraph alignment are stored using a popular "extra attributes" syntax, while custom block-level nodes such as alerts use another popular "container" syntax, wrapping the node content in a fence specifying the block's node type.

Encoding the node tree back into a Markdown document is done using the same stack, but mapping ProseMirror nodes into MarkdownIt tokens instead, which can

then be converted back into a Markdown text document to send to the backend.

10.2.5 Copy/paste

Copy & paste actions have special handling to make them work not just with the selected text, but also the nodes that contain it, allowing the user to also copy & paste semantic blocks such as images or tables.

This is done using the standardized browser clipboard APIs and ProseMirror's selection callbacks. When selecting content with the cursor, the editor keeps track of the nodes it spans within the tree, which can then be encoded to JSON and stored on the clipboard.

Upon pasting, this JSON is parsed back into nodes and re-inserted into the document at the cursor's position.

To prevent the user from pasting this JSON outside the editor context, the clipboard API calls mark it with the `web application/json` text type, which makes it be ignored outside of JavaScript access.

It's worth noting that selections don't need to span entire nodes, for example if the user selects only half of a paragraph's text. Pasting such content by inserting new nodes naively produces unintuitive behaviour, as it will for example create new paragraphs rather than inserting the node's content into the existing one the cursor is on. This issue is solved by converting these "open" ends of the selection into *sliced nodes*, which upon being pasted back will attempt to be merged into the nodes before & after the cursor by ProseMirror.

10.2.6 Editor widgets

Editing certain nodes (for example, changing the URL of an image) is done through modals with form-like fields. These are opened either through UI interactions of events from the viewmodel when a corresponding tool is used. For example, the modal to edit an image's URL(*See annex figure 33*) can be opened either by double-clicking the image, or with the "Edit image" tool in the context menu, both of which send events that are consumed by the modal to start its interaction flow.

To make these more maintainable and modular, each is a separate component that manages its state and issues operations to make to the document. For example, the "Upload image" action invokes a modal which handles uploading the file to the CMS, extracting the URL and then issuing the ProseMirror transaction to insert an image node into the document at the cursor position, making the whole flow self-contained in a single component.

The many modals the UI uses are aggregated in a renderless WidgetsManager component, which listens for editor callback events to open the relevant modals and performs the corresponding operation on the document when the user finishes using the modal and closes it.

Internally all ways of inserting images correspond executing the same action to insert the image node of the ProseMirror schema, which is serialized to the standard Markdown image link syntax. The different ways of inserting images only differ in the form the user fills out to determine the URL.

10.2.7 MVVM pattern

Changes to the editor model are made observable by wrapping the model class instances in a Vue ref, which makes it deeply reactive. These bindings are created by the viewmodel composables and cause the components that depend on them to be re-rendered when there are changes to the model's fields, such as the user preferences.

Structures that are immutable, such as the document's schema, are excluded from this to avoid the overhead of unnecessary refs.

Some composables take parameters; for example, the `useToolbarActionMenu` composable creates a viewmodel for a specific action menu. Since the components that use this are likely to receive the menu as props, the bindings that the composable creates also need to be reactive. For this, the ref normalization syntax is used to efficiently create separate reactives for specific prop items rather than passing the entire props object itself to the composable, which might include additional data it doesn't need.

The EditorDocument component is the authority over the ProseMirror state mutations. The initial implementation of the editor attempted to have only the root component be the authority over state mutations, with the motivation that it would make the flow of interactions and model mutations clearer; child components contained only presentational logic and interactions with them would only propagate events to this root component, which would then apply the corresponding operations on the model.

This approach proved to be very poorly scalable and tedious to maintain due to the amount of event listeners and dispatchers (Vue emits) required.

The entire editor was refactored in the middle of the project's development to apply the final MVVM architecture; this change of philosophy in regards to separation of concerns led to a much cleaner and flexible implementation.

10.3 Technology choices

Other web frameworks such as React would've been perfectly viable for the project, with only minor inconveniences; the decision was skewed towards Vue due to personal desire to learn it at a deeper level and be able to make better comparisons & decisions between the frameworks in future projects.

Both Vue and React are based around the idea of rendering the site through re-usable and self-contained components that combine HTML templates with scripting, but the implementation and developer experience of this concept differs greatly between them.

In React, the HTML of components is defined using a JSX, a syntax expression of JavaScript; this makes it very easy to implement highly-dynamic HTML rendering using all conditional and flow structures JavaScript offers, however this often leads to rendering and model logic being coupled together, difficulting maintainability as structural (and even stylistic) HTML changes generally also imply having to modify the logic that generates it. In a way, it can be seen as a programmatic UI framework. Though very powerful, the project's UIs weren't as dynamic so as to greatly benefit from this.

The syntax of components in Vue on the other hand maintains the traditional web separation of HTML, JS and CSS, making it easy to maintain them as a whole as the three concerns (document structure, logic and appearance respectively) are not all scattered throughout a single component's file, and instead have dedicated regions to define them within the component file.

React is also purely a view library; other essential aspects of web development such as state management are delegated to other libraries and do not have first-class support.

10.4 Database design

This annex contains some additional comments on the database's design, referencing the entity-relation diagram(Fig. 3.4).

Admin accounts do not have any biographic information like editors do (such as the display name field) since they are not intended to have a physical presence on the published site, unlike editors authoring the articles.

Whether an article is published is a boolean that is a derived attribute; this means its value can be inferred, as we can consider an article to be published if the `publishingTime` field is not null. Derived attributes need not be stored physically in the database implementation since they're redundant information, but are useful in ER diagrams to model more natural/intuitive properties of an entity.

The requirements for site configuration include the ability to toggle which social media platforms articles can be shared to; this is simply modelled by the Social-Network entity with a boolean field to enable/disable sharing to that network.

The File entity would be used mainly for images uploaded to be used in articles, as well as attachments linked. It has a relation to store the Editor who uploaded the image; though this is not necessary given the app's requirements, it was decided to track the uploader for future-proofing in case a need to identify them arises later, similar to the reason for the existance of an Admin entity.

Typically a configuration file (a JSON, YML or similar) is used for global app configuration, instead of a database entity the project used. Though it would've been possible for the backend to write/update such file from requests, it would require storing it separately from the database. This would've created a concern of the app's persistent data being stored in multiple different manners, making it less convenient to make backups or to restore from them.

10.5 Database implementation

This annex contains notes on the implementation of the database.

The ER diagram of the project presents entity inheritance, mainly for the User hierarchy to distinguish roles. Multiple ways of modelling inheritance in SQL exist:

- Creating tables only for the derived entity, and giving them columns corresponding to fields of both the derived and parent entity in the diagram.
- Creating tables for both the parent and child entities, and creating a one-to-one "parent-child" relationship between them.

The first approach has the downside of duplicating attributes across tables, which makes further modifications to the database inconvenient as all child tables need to be modified to add/change/delete columns to/from the "parent" entity. In addition, ORM libraries tend to handle this implementation poorly as the tables are seen as independent classes, which tends to add verbosity when writing code that works on only the attributes of the parent entity (switch-cases, casting, etc.).

The second approach solves the duplicity and extendability concerns, at the performance cost of needing to perform JOINs to traverse the hierarchy and get all information of an entity including fields from the parent/child. In this particular case only 1 JOIN is required to query all information of a user, thus the advantages of this approach were considered to outweigh the minor performance hit. The Admin and Editor classes have the username attribute as both primary keys and foreign keys relating them to their corresponding entry in the User table. With this implementation, Admin and Editor are weak entities - they cannot exist without a corresponding User entry.

Staff accounts are identified by a username, which was chosen to be the primary key of the User table as clients accessing the database (the backend, and by extension, the frontend) prefer to use friendly string identifiers over arbitrary integers in queries whenever possible. Using an integer as primary key would also slow down queries by username unless an additional index were created for that field.

The diagram uses the convention that the cardinality N corresponds to "one or more"; for relationships like "Editor is author of Article", the cardinality "N - 0:N" is used as an Editor might have no articles (for example, if it's a newly-created account) but an Article must always be credited to at least 1 author.

Storing the CMS's files, such as the article documents and uploaded images, has multiple possible approaches:

1. Storing the document as a regular file, and storing its relative path in a text column in the database.

2. Using the BLOB or FILESTREAM types in SQL to store the document contents directly in the database.

3. Storing the document in a cloud storage solution (ex. Amazon S3), and storing its identifier in a text column in the database.

The first solution is perhaps the most intuitive but also the most naive - its main issue is fragmenting the sources of data for the app. For the backend, this creates the need to handle additional exceptions such as the file having been deleted from the stored path externally (outside the app) or the file/directory not being writable due to OS shenanigans - it's a low-integrity solution that creates many design problems.

BLOB and FILESTREAM types in SQL allow storing binary data. Text documents, even long blog articles, tend to be small (below 256KB) and as such storing them as blobs would work and maintain data integrity, avoiding most of the issues of dealing with files directly - accessing the data would be slower, but this is not a large concern.

The most robust and scalable option is to rely on a cloud storage solution, although this couples the app to such a service and makes it more difficult and inconvenient for a user to even locally try out the app, as they would need to setup and pay for the service even before committing to deploying it for production.

The navigation bar configuration is also stored in a special way. The navigation bar, as described in the requirements, can be modelled as a tree where internal nodes are categories, and leaf nodes are links to articles or external links. Though it would've been possible to model this via tables and relations, it was deemed a needlessly convoluted implementation.

Ultimately any client will want a complete view of the navigation tree in a structured format; storing it via tables would mean needing dozens of joins to reconstruct the whole tree whenever it is needed. As such, it was decided to store the navigation bar configuration as a JSON document that already has the complete structure, consisting of nodes that reference categories and articles by their identifiers only. Since the frontend will need to fetch the navigation bar structure for each visit to the site, storing it in an immediately-usable format lowers server load for these requests - they are reduced to a single database query.

10.6 Version control and git workflow

The project used Git for version control with a mono-repo approach - there is only repository that holds the source code for both the backend and frontend. Since there are only 2 services, using a separate repository for each was deemed to be unnecessary and inefficient for a one-person project.

The mono-repo pattern has a minor inconvenience as it may not be obvious to readers which service each commit concerns, yet it's important for commits to be organized and clear in that regard. The project follows some personal patterns that worked well in prior projects.

Commits always only contained changes to 1 service, never mixing changes to both. This constraint is helpful in situations where a particular change to a service needs to be cherry-picked to another branch or reverted, as it allows doing so without picking/reverting changes to the other service which may not be necessary.

Commit messages generally follow a format of "`<Service> <Module>: <Commit message>`", with the fields corresponding to:

- **Service**: either frontend or backend, for the reader to quickly be able to identify which part of the project the commit concerns.
- **Module**: the system or feature the commit concerns. May also be the class/-file name for commits that edit only 1 file.
- **Commit message**: describes the commit's changes, starting with a verb that indicates the nature of the commit ("add" for new features/systems, "fix" for bug fixes, etc.)

A consistent commit message schema like this makes the version history easier to read, but most importantly makes it easy to find commits that alter specific systems

by using a regular expression search for the service and module parts (using `git log --grep`). This is helpful in finding commits responsible for regressions discovered later. Together with the restriction of committing backend and frontend changes separately, this also allows filtering the commit history by service, lessening the disadvantage that the mono-repo approach has in this regard.

The repository was hosted on GitHub at https://github.com/PinewoodPip/bloggy, under the name "Bloggy" as it is self-descriptive, and the diminutive communicates its relatively small scope. A readme is provided with an overview of the project's features as well as instructions to configure and build it, listing environment variables, launch commands and development environment tips. The project's code was licensed under the GPL license, allowing the projects or parts of it, such as the editor, to be freely reused and modified while ensuring derivative work stays similarly-licensed.

Given that the project only had one developer, it wasn't worthwhile to have a complex branching strategy. The project used only two branches: the `main` branch with stable snapshots, and a `develop` branch for development, which was periodically merged into main after major milestones.

The project uses GitHub actions for continuous integration, executing the backend tests after every push to ensure regressions were never overlooked. These tests are run using Docker Compose, as was detailed in Section 5.5 «Containerization». In addition to this, when implementing major features or refactoring, the tests were also executed locally proactively to try to keep the branches healthy.

GitHub Projects was used to track daily work(Fig. 47), as described in Section 2.1.

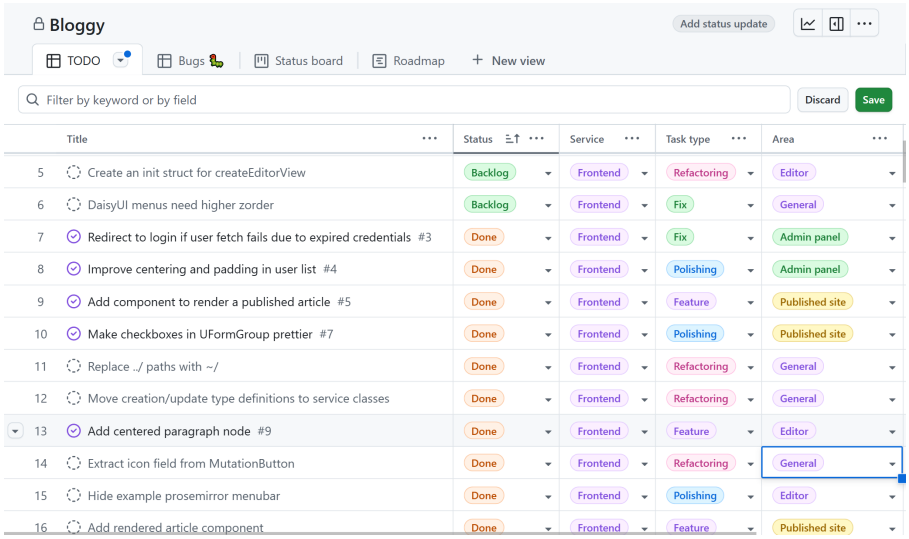


Figure 47: Screenshot of the GitHub Projects board used for tracking daily work.

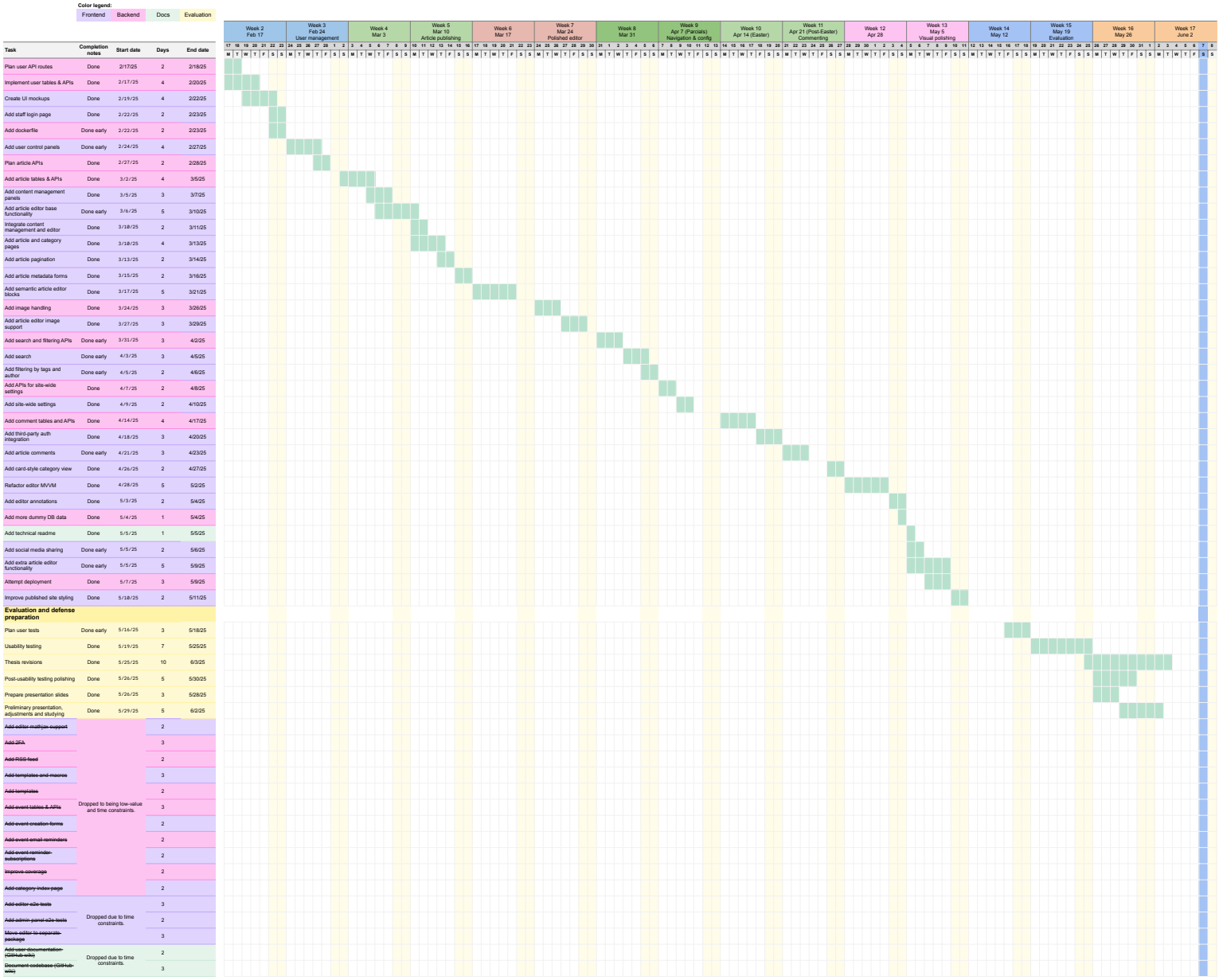


Figure 48: Final Gantt chart of the project.

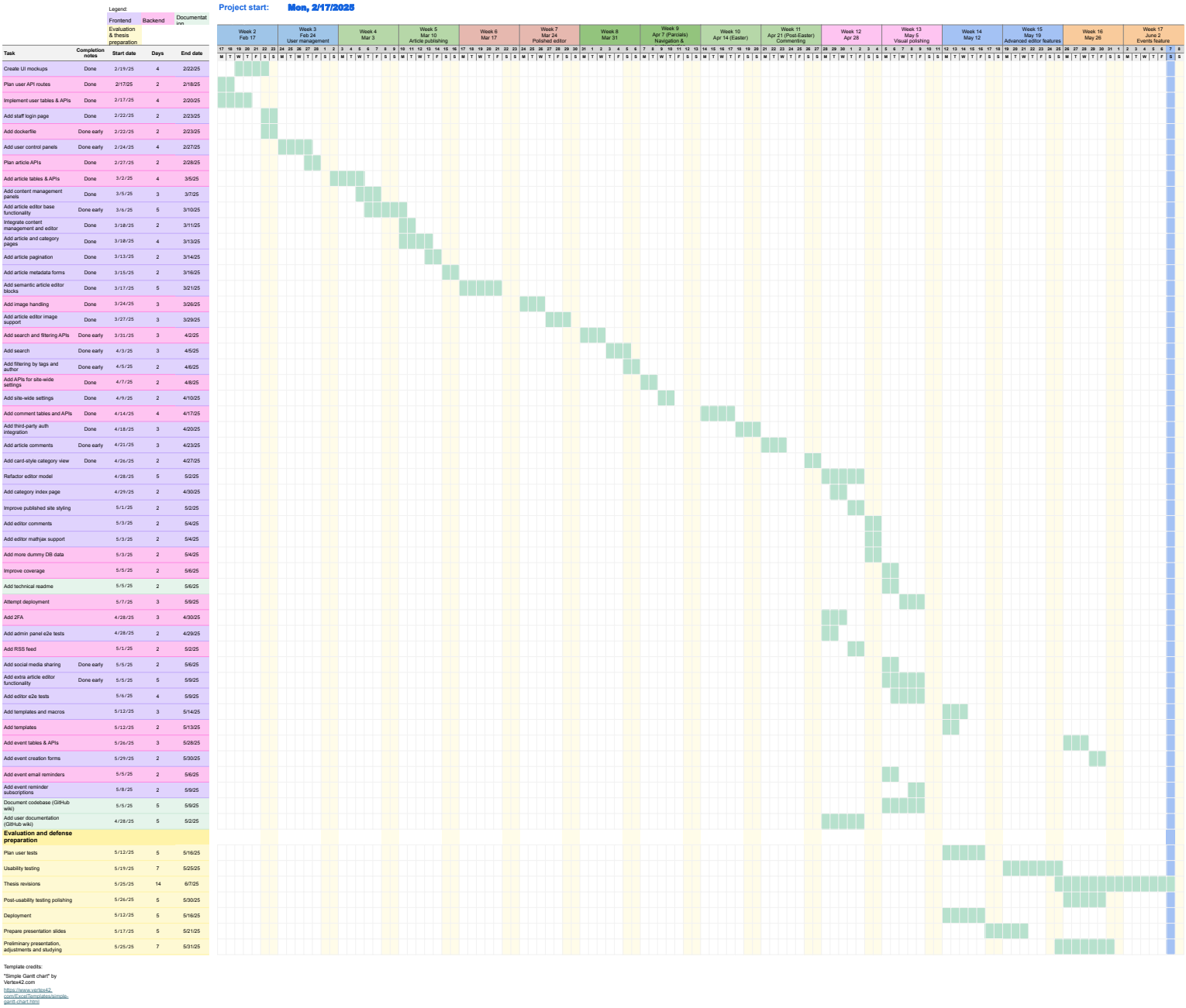


Figure 49: Initial Gantt chart of the project, used until late April.

10.8 Backend

The backend service uses the following folder structure:

- **core**: contains configuration structs, security utility methods (for password hashing) and lifespan manager (database initialization routines).
- **crud**: contains CRUD utility functions for all database entities, with separate files for each major "family" of entities (users, articles, categories, etc.).
- **models**: ORM table definitions for the database.
- **routes**: FastAPI endpoint routes; separated into one file per major root route (ex. `/articles/`, `/users/`).
- **schemas**: Pydantic data validation models for the API, most corresponding to database entities.
- **tests**: pytest test files, organized mainly by the API routes they cover.
- **main.py**: entry point for the application; creates the FastAPI application and registers all endpoint routes.

All function parameter and return values are explicitly type-annotated to improve IDE support, except in special cases where common IDE integrations can infer them, such as pytest fixtures.

Files that import model classes or schemas of other entities do so with a prefix (using the "`from X import * as Y`" syntax) to avoid confusion between database model classes and schemas.

10.8.1 Mock data

For demonstration purposes, the repository also contains a folder with mock data in the `backend/mock_data` folder, together with a Python script that populates the database with that data by scripting the corresponding API calls.

This mock data adds users, articles, categories and images to create an example usage scenario of a cooking-themed blog. Most article text is lorem ipsum with some formatting to preview the editor's features, which was generated using https://jaspervdj.be/lorem-markdownum/.

10.8.2 API pagination

Output schemas that contain lists of objects, such as a category's articles, may result in needlessly large responses to clients that may only need a slice of that data - for example, if the frontend only needs to display N articles of a category per page, it would be redundant (and wasteful on CPU time and bandwidth) to send it all articles at once in a single request, as there may be hundreds of them.

To address this, the relevant API routes accept the "limit" and "skip" parameters in requests, which allows limiting the length of the returned object list and

offsetting the items from the query respectively. This allows API clients to implement pagination, by fetching N items at a time, incrementing the skip parameter with each query to access latter items. The responses always included the total amount of items that exist to allow clients to deduce the amount of "pages".

10.8.3 Elasticsearch toggle

The backend supports disabling the Elasticsearch integration by leaving its URL environment variable unset, which is handy when not working on search features as the Elasticsearch server is very demanding on resources. If search is disabled, the corresponding API endpoints return the "503 Service unavailable" response.

10.8.4 Exception handling

The backend uses exceptions at both the API and CRUD level to notify the caller of any issues with requests/calls. Care is taken to use the appropriate exception types in each module.

The exceptions at the CRUD level use the basic, generic Python exception types (mainly `ValueError` for invalid parameters, `RuntimeError` for internal failures). These exceptions are caught by the API route functions and translated into the appropriate HTTPException type based on the nature of the CRUD exception to stay coherent with the HTTP protocol and give clear feedback to the client about request failures.

For example, fetching an article that doesn't exist would result in a ValueError at the CRUD level that is translated to the `404 Not Found` HTTP response. On the other hand, trying to create an article with a filename that is already used would instead result in a ValueError that translates to `400 Bad Request` response instead.

The exceptions at the CRUD level are generally focused on database coherency, whereas the API may also throw additional exceptions related to the permission system, for example if an editor account makes a request to an endpoint that only admins can use.

10.8.5 Mapping ORM models to schemas

When the API wishes to return an object from the database in a request, it must convert the ORM model instance to a corresponding schema class. Since the names & types of fields in the schema class can differ wildely from the fields internally used by the database objects (for example, having fields that refer to other objects from a relationship), this mapping cannot be done directly.

Normally this would involve instantiating the schema class and explicitly passing the model-schema mapping of each attribute explicitly in the constructor. To reduce verbosity and simplify expanding the models and schemas with new fields, utility methods exist to create these schemas automatically using reflection-like methods that SQLAlchemy and Pydantic provide. These methods accept an ORM object as well as the schema to create from it, using generics for type annotations to infer return types from the parameter types.

Similar utility methods exist to update an ORM model with values from an *Update* schema, reducing verbosity in the CRUD methods.

These utility methods are in the `app/crud/utils.py` script.

10.8.6 Authentication

Other solutions for authentication and tracking sessions were considered, but deemed inappropriate for the project:

- URL encoding: after logging in, a string session identifier is attributed to the user and they are redirected to a page that contains the session ID as a URL parameter (ex. www.example.com?sessionID="someID"). All links on the pages must contain this ID, which will be sent back to the server with each request to identify the user as they navigate the site. This is fairly annoying to implement, and clunky as the user will see the session identifier in their browser's address bar, and also insecure as anyone who sees that address can navigate to it to gain access to that session.
- The IP of the request cannot be used for this purpose as they do not identify a single device in the case of NAT (Network Access Translation), which is the case for basically all networks.

The project has utilities to make authentication checks convenient across all endpoints; endpoints that require auth are defined with a `current_user` parameter that defaults to a FastAPI "Depends" functor, which parses the header and returns the corresponding User entity if a valid token was passed, or throws an HTTPException otherwise. These "Depends" functors are called automatically by FastAPI when invoking an endpoint, making them ideal to avoid code repetition for checks like the authentication requirement.

Though multiple client-side storage mecanisms exist, cookies were most appropriate for security as they can only be read by scripts from the same site.

10.8.7 API methods

The API endpoints accept the following HTTP methods:

- GET: for retrieving the data of an entity.
- POST: for creating new entities.
- PATCH: for *partially* modifying the data of an entity.
- DELETE: for deleting entities.

The PUT method, which is intended to entirely replace the data of an object, is not used much by the API as it is made largely redundant by the more-flexible PATCH method, which may be used to both partially and fully replace an object's data, and often results in smaller request sizes. The exception to this is the `/files/` endpoint, where it only makes sense to completely replace all data when re-uploading a file - it's not possibly to partially "patch" a file.

The endpoint to fetch all files in the CMS is intended to only be used in the context of an admin panel; it requires auth to avoid it from being abused to leak files that are not yet intended to be public, for example to prevent being able to see images/attachment used in articles that have not yet been published.

10.8.8 Schemas organization

The project suffixes the schema class names based on their purpose within the API:

- Input (ex. "UserInput", "ArticleInput"): schemas for entity creation (POST requests)
- Output: schemas for result responses (GET requests, as well as to confirm the result of POST and PATCH requests) that contain a complete view of an entity's data.
- Update: schemas for modifying entities (PATCH requests). The PATCH HTTP method is functionally distinguished from PUT as it allows modifying parts of an object rather than replacing it entirely. This translates to requests needing to only provide the fields whose values are to be changed; the Pydantic schemas thus have all fields annotated as `Optional`
- Preview: smaller schemas that contain only essential metadata of entities (ex. their ID, user-friendly name) without all the details that would normally be in an Output schema. These are used as fields in schemas that reference other entities as part of their Output, with the intention of reducing sizes of responses by excluding details that are not essential to the request. For example, the response schema for a category lists the articles of the category; since data such as the entirety of the article content is not necessary for requests that fetch a category, the ArticlePreview schema is used instead, which provides only the name of the article and ID to be able to fetch its complete information in another request if necessary.

10.8.9 Catch-all routes

Since both categories and articles are meant to map to URLs on the published site, the API also identifies them by URL. For example, if an article "Red velvet" is in a "Cakes" category that is within a "Cooking recipes" parent category, then it may be retrieved with a `GET /articles/cooking_recipes/cakes/red_velvet` request.

In FastAPI this is implemented using "catch-all" routes; only a single route is defined for each entity *type* (/categories/ and /articles/) but are configured to accept trailing URL components that the endpoint handler receives as an additional parameter. As such, a request like `GET /categories/cooking_recipes/cakes` is handled by the `/categories/` endpoint and receives `/cooking_recipes/cakes` as a parameter.

As mentioned in the database implementation, the category entity uses a self-referential relationship to represent the parent-child dynamic and allow "nesting" categories. This allows traversing them like a tree.

Since the API uses slash-delimited URL paths as inputs, to validate a path it's necessary to decompose it into its parts and traverse the categories tree by fetching the subcategories. This is necessary for category creation to validate the parent category to place a new category under; since there may be multiple categories with the same directory name but under different parents, it is necessary to traverse all categories along the path to validate they exist (and have the correct parent) and to find the parent category of the new category being created.

This traversal is necessary for validating paths, but is unnecessarily slow for simply resolving a URL to a category. For this, categories also cache their complete URL so they may be retrieved using a single SELECT query. This cached URL is calculated when creating the category, as well as whenever the category's (or its parents's) directory name changes. Changing the directory name requires this cached field to be recalculated for all the subcategories, however this is an infrequent operation. To make resolving a URL to its category fast, the URL field is indexed using the "hash" index option in PostgreSQL. This creates a hash-table structure that allows looking up categories by this field in constant time; with enough categories in the system this is faster than a regular alphabetical index.

Articles may be moved to a different category via the PATCH /articles/ endpoint as opposed to the categories one. Although both endpoints make sense conceptually, the complexity of implementation (and complexity of use for the client) differs greatly:

- Moving an article using the article endpoints requires the user only to specify the new category for the article.
- Moving an article by patching a category would require the client to provide a new list of **all** articles for both the old category (with the article(s) being moved out) and new category (with the article(s) moved in). This implies not only 2 requests to the API, but also that the article(s) being moved would, at least temporarily, be in an orphaned state where they don't belong to a category. This is an awkward object state that would create a lot of hassle to deal with.

Though this appeared very clean and convenient at first, it poses a small problem with implementing nested resources, such as routes for comments of an article. Usually REST APIs use numerical IDs, thus there would not be issues with creating a route such as `GET /articles/<ID>/comments`. When using string identifiers for resources that in themselves are also paths, a possibility for ambiguity is introduced; it's unclear whether a request to GET /articles/SomeID/comments refers to the comments of an article with the path "SomeID", or if it refers to an article whose path is "SomeID/comments".

The project worked around this by using a separate top-level route (`/comments/`) rather than nesting resources.

10.8.10 Navigation bar schema

The site configuration database table stores the model of the site's navigation bar, as the requirements needed it to be configurable. Though it would've been possible

to automatically generate the sitemap based on the categories tree, it was thought to be more desirable to give the user control over it, especially to allow it to link to external sites.

The navigation bar is stored as a JSON document with a tree schema, where the root nodes are intended to correspond to the top-level navigation links, while child elements would correspond to secondary links the frontend would display in dropdowns. This schema supports different nodes to allow linking different content and structuring the navigation bar:

- **Article**: links to a specific article on the site. Intended to create links to articles that act as special standalone pages, such as the typical "About" page.
- **Category**: links to a category's index.
- **External URL**: arbitrary links to external sites that may be related to the blog.
- **Group**: groups multiple kinds of navigation nodes under a parent node, intended to create dropdowns in the navigation bar.

The JSON identifies articles & categories by their numeric IDs instead of their paths to make it robust against articles/categories being renamed or moved. The tree does need to be re-validated when any article/document is deleted, as the lack of formal relationships makes this impossible to detect, however this was considered not problematic as deletions aren't done frequently.

The API parses this document into a response schema that includes the metadata of the articles/categories so they may be displayed easily by the client without needing to make additional requests to fetch them.

The API to update the navigation bar uses the user-friendly path identifiers as the rest of the API. The schema classes make use the Pydantic Discriminator feature to be able to properly validate the differents kinds of nodes based on a "type" field.

As noted previously, using JSON for this is an unusual choice, but one justified by how cumbersome it would be to model all these nodes using separate tables. The navigation schema is semi-structured, and thus would've been much more convenient to implement if the project were using a documental database.

10.9 Usability tests

10.9.1 Methodology and planning

The following pages contain the script with the tasks given to the participants after the introduction and consent form:

Task 1

The first task tests the navigation of the published site by having the user find an article of a specific theme, which may be done by navigating to a specific category, or through the search bar or tag filters.

Script for the participant:

> You're looking for curry recipes to cook for lunch, and stumbled upon a general cooking blog that looks alright.
>
> Browse the site and look for a recipe for some red curry.
>
> The task finishes once you have found instructions on how to cook some red curry.

Starting page: http://localhost:3000/

Figure 50: Script for task 1 of the usability test.

Task 2

The second task tests posting comments on articles, which involves logging in via SSO.

Script for the participant:

> You have tried following a recipe from the site, but the taste turned out quite bad, and you suspect it's because one of the instructions in the recipe is wrong. You decide to try to let the author of the recipe know.
>
> Post a comment on the article to tell the writer that some part of the recipe is wrong. Include a quote from any text in the article and an emoji to sound more friendly.
>
> The task ends once you see the comment was posted.

Starting page: http://localhost:3000/articles/oriental/Red_Curry

Figure 51: Script for task 2 of the usability test.

Task 3

The third task covers creating an article. The user assumes the role of a site staff member.

This task involves the article editor and is intentionally a bit open-ended to observe the user's first impressions and experience with the editor.

Script for the participant:

> After contacting the site author to tell them about the mistake in that previous recipe, you were hired to work on the site, and tasked with writing a new pasta recipe.
>
> Your task is to access the site's management panel, create a new article for a pasta recipe, and write some step-by-step instructions on how to cook it.
> You're free to write the article how you want, but the boss requires you to add at least 1 image.
>
> The task ends once you have published the article and verified that it's visible on the site.

Starting page: http://localhost:3000/admin/login

- **Username**: editor123B
- **Password**: asdasd1!

Figure 52: Script for task 3 of the usability test.

Task 4

This task has the participant use specific editor tools to complete an article where another editor user has left "TO-DO" notes using the annotations feature, which also serves to gauge the participant's response to collaborative features.

Script for the participant:

> Some other member of the site started writing a carbonara pasta recipe, but they couldn't finish the article and left some problems for you to fix. He said he left notes on what's left to do.
>
> Your task is open the editor for the carbonara article, and to fix every issue that he noted within.
>
> The task ends once you have addressed every issue, published the article and verified it looks alright in the site.

Starting page: http://localhost:3000/admin/content

- **Username**: editor123B
- **Password**: asdasd1!

Figure 53: Script for task 4 of the usability test.

Task 5

The final task has the user become the site's manager and perform some site-wide configuration changes. It tests the "Configuration" tab of the admin panel specifically, as well as the navigation bar editor.

Script for the participant:

> After becoming an experienced writer for the site, you got promoted to the boss and now have full control over it. You decide to give it a rebranding.
>
> Your task is to change the site's name, upload a new logo, as well as to add a link to your pasta article to the navigation bar so it's easier to get to.
> (You have a nice new logo saved on the computer already.)
>
> The task ends once you confirm that the logo and navigation bar was changed on the site.

Starting page: http://localhost:3000/admin/login

- **Username**: pip12345
- **Password**: pip1234$

Figure 54: Script for task 5 of the usability test.

10.9.2 Usability test results

This section contains the spreadsheets used to note down the results and observations of the usability tests(Fig. 7.1).

				Expressions and comments		
User	Success	Time	Errors made	Positive	Negative	Observations
1	Yes	0:38	0	1	1	Tried clicking on the summary in the search results, which shows pointer cursor but doesn't work as a link
2	Yes	0:25	0	1	0	Used search bar, clicked the header in the search results
3	Yes	0:40	0	1	0	Used search bar, but clicked "all results" instead of the previewed ones (which included the goal article)
4	Yes	0:18	0	1	0	Like user 1, tried clicking summary, but then clicked the proper link shortly after. Didn't comment on this issue later.
Average		0:30				
Deviation		0:10				

Figure 55: Results and observations of task 1 of the usability testing.

				Expressions and comments		
User	Success	Time	Errors made	Positive	Negative	Observations
1	Yes	1:30	1	1	1	Used the editor toolbar features, but accidentally made a double-quote expecting it to toggle off the quote
2	Yes	0:58	0	0	0	Commented on copy/paste buttons on toolbar being redundant
3	Yes	1:00	0	0	0	Hesitated to log in. Commented that it's not normal to have to log in first before the comment box appears.
4	Yes	0:56	1	1	0	Accidentally created a nested quote, couldn't figure out how to remove it - had to rewrite the whole comment
Average		1:06				
Deviation		0:16				

Figure 56: Results and observations of task 2 of the usability testing.

				Expressions and comments		
User	Success	Time	Errors made	Positive	Negative	Observations
1	Yes	3:10	2	1	2	Tried clicking a category to create an article, and also closed the table of contents thinking its close button was a button to go back to the control panel Used the editor features well, and found the markdown shortcuts to be intuitive Note toggling was bugged
2	Yes	4:13	0	0	1	Inserted an image from a link. Commented about the control panel looking "too professional", later clarified to mean it looked too technical. Suggested using a cards-like layout
3	Yes	2:05	0	1	0	Immediately found the "+" button, understood the table of contents and editor controls (bullet list, image uploads) Suggested that the placeholder text of the editor (after creating a new article) should disappear after focus.
4	Yes	1:12	0	1	1	Was overall very quick to find all the relevant UI elements, including the create button. Was mildly annoyed by the placeholder text on the editor not clearing after focusing the field.
Average		2:40				
Deviation		1:18				

Figure 57: Results and observations of task 3 of the usability testing.

				Expressions and comments		
User	Success	Time	Errors made	Positive	Negative	Observations
1	Yes	6:13	3	0	2	Tried opening the links added in editor, and thought the tooltip field was the link text. Found it difficult to add tags, was expecting them to be at the bottom of the article
2	Partial	6:40	3	0	1	Was not able to find the article metadata form to add tags; tried doing so by writing "#tag" in the article and looking at toolbar options. Also misunderstood one of the tasks (reordering sections) Did not notice annotation popups at first, thought the yellow highlight was just formatting
3	Partial	3:40	2	2	1	Tried adding tags with a "#tag" text and tried opening inserted links. Used lists, copy+paste, alignment and resizing well. Was not initially aware that the yellow highlighted indicated annotations. Commented that the resize cursor should not just show a Y-axis resize. Commented they found the editor to be easy to use overall.
4	Yes	4:39	2	2	0	Initially opened the "edit metadata" form instead of editing the article content - figured it out shortly later. Tried dragging the image to center it. Did not resolve annotations, only deleted the highlighted text. Completed most of the tasks very efficiently (copy+paste, select + toggle list, etc.) Resized the image using the form instead of the handle - commented later that it was hard to see. Struggled to find the tag options (also wrote #tag), but then saw them when inspecting another article and did figure it out shortly later.
Average		5:18				
Deviation		1:23				

Figure 58: Results and observations of task 4 of the usability testing.

				Expressions and comments		
User	Success	Time	Errors made	Positive	Negative	Observations
1	Partial	8:25	2	0	1	Confused the sidebar with the navigation bar. Tried opening the links in the sidebar editor like in task 4
2	Yes	5:04	3	1	1	First exited the config panel without saving. Navigation modals were short and they struggled to find the correct option to link an article. Confused logo & favicon. Tried to ctrl+z in the navigation config
3	No	5:34	4	0	0	Was overall very confused about the task. Saw the config tab but didn't click on it. Messed around the article editor for a long time, seemingly trying to figure out the URL scheme, and then edited the metadata to change its thumbnail and name - later confirmed to have confused the "logo" and "site name" with those fields. This is largely a fault of the task being a bit poorly written, as well as having forgotten to ask the participant to recite it back to confirm they understood it correctly
4	Yes	5:25	2	1	1	Found config tab quickly, but struggled to find the upload button - expected it to be in the popup modal. Visited the files tab a few times but it took them 3 visits until they noticed the upload button. Didn't save settings before leaving the tab the first time; they noticed this afterwards Possibly thought that the file browser in the UI was the PC file browser? Changed navbar very quickly and correctly. Commented the task was easy except for the logo part.
Average		6:07				
Deviation		1:32				

Figure 59: Results and observations of task 5 of the usability testing.

	1 = Strongly disagree		5 = Strongly agree	
Question	User 1	User 2	User 3	User 4
1. I think that I would like to use Bloggy frequently.	3	3	4	4
2. I found Bloggy unnecessarily complex.	3	3	3	2
3. I thought Bloggy was easy to use.	3	2	4	4
4. I think that I would need the support of a technical person to be able to use Bloggy.	4	2	2	1
5. I found the various functions in Bloggy were well integrated.	5	3	5	4
6. I thought there was too much inconsistency in Bloggy.	4	2	1	1
7. I would imagine that most people would learn to use Bloggy very quickly	3	3	5	5
8. I found Bloggy very cumbersome to use.	3	2	3	1
9. I felt very confident using Bloggy.	3	4	5	4
10. I needed to learn a lot of things before I could get going with Bloggy.	2	1	1	1
	52.5	62.5	82.5	87.5
Average	71.25			
Standard deviation	16.52018967			

Figure 60: Results of the system usability scale questionnaire.

User 1	User 2	User 3	User 4
Wanted the ability to select text in the article and to quote it in the comments box directly, instead of needing to copy-paste it manually	Wanted a direct quote feature while commenting, just like user 1	Would've wanted the staff tasks like creating & editing articles to be available directly from the published site, if you're logged in.	Overall found the site to be well-integrated. Had little feedback beyond the explicit complaints about the issues found during the tasks (mentioned in the task sheets)
Would like a preview feature in the article editor, that would also show elements like the subtitle and the footer cards	Would've liked a way to upload files directly when ex. changing logo	Generally would've preferred the site & admin panel to not be entirely separate interfaces, have them be more integrated with each other	Would greatly appreciate a more direct way of replacing assets like the logo, directly from the config tab
Would've liked a way to view all annotations on an article, and a historial		Would like tags to be added using "#tag" text in the article, similar to how they work in social media	
Would've liked more previews for config options like the sidebar		A large reason why they'd like to see the control panel & site integrated is to make it easier to confirm that edits were made correctly, in the 4th task they were annoyed they had to go to the published site exlicitly to confirm so as the admin interface does not make previewing some changes clear	

Figure 61: General feedback, comments and suggestions from the participants.

10.10 Design artifacts

The following sections contain complete lists of the design artifacts created during the requirements gathering phase in Section 3.

10.10.1 Functional requirements

This annex contains the complete list of functional requirements, grouped by their corresponding user story epic.

10.10.1.1 User management

The user management requirements revolve around administration of the site staff's accounts. The site only supports traditional accounts for the staff that manage the site and its content - regular site visitors (the "reader" role) may "log-in" using third-party authentication systems instead, described later in the "Content exploration and interaction" requirements list.

1.1 Admins should be able to create editor accounts with a username & password

1.2 Editors should be able to edit their profile (name, avatar, biography)

1.3 Editors and admins should be able to change their password

1.4 Admins should be able to set which under which categories an editor can create & edit articles and subcategories

1.5 Editors and admins should be able to log in using username and password

1.6 Admins should be able to delete editor accounts

1.7 Admins and editors should be able to opt into 2-factor authentication for their accounts

10.10.1.2 Article creation and management

The article creation requirements describe the options the editors have for setting up new content articles for the site. Articles are organized into categories, which model file directories and would define the sitemap (URL structures).

2.1 Editors should be able to create named categories to organize articles thematically

2.2 Editors should be able to create an article under a category

2.3 Editors should be able to set an article to be published (made visible to readers) after a set date & time

2.4 Editors should be able to add text tags to articles

2.5 Editors should be able to save unfinished articles as "drafts" which are not visible to readers

2.6 Editors should be able to set whether comments can be posted on an article

2.7 Editors should be able to mark other editors as co-authors of an article

2.8 Editors should be able to set whether an article displays metadata (creation date, author) to readers or only the article content itself

2.9 Editors should be able to set a summary/short description of the article to be displayed in contexts where the article is previewed

2.10 The system should auto-generate summaries/short descriptions of articles that do not have these set manually

2.11 Editors should be able to set how articles are displayed while browsing a category; in a vertical paginated layout, or using a grid card-like view

2.12 Editors should be able to set how articles within a category are sorted; chronologically by publishing date or manually-ordered

2.13 Editors should be able to set a featured image for an article, to be displayed in paginated views

These operations would mainly be accessible from a site control panel accessible only to site staff.

10.10.1.3 Article editing

The article edition requirements mainly concern the functionality of the rich-text article editor and the semantic blocks the user can insert into them.

Though not specified explicitly, the "add X" requirements should also allow the user edit the result in posterity or to remove it from the article.

3.1 Editors should be able to format text within articles (bold, italics, underline, mono-spaced/code)

3.2 Editors should be able to add images to articles by choosing a file to upload and setting its alt text

3.3 Editors should be able to add images to articles by choosing an image that was already uploaded to the site previously

3.4 Editors should be able to add images to articles by hotlinking them

3.5 Editors should be able to edit existing articles

3.6 Editors should be able to add code blocks to articles and set their language for syntax highlighting

3.7 Editors should be able to add section headers to articles

3.8 Editors should be able to reorder sections within an article

3.9 Editors should be able to add tables to articles, consisting of text cells

3.10 Editors should be able to add links to articles, specifying URL and its name/-text

3.11 Editors should be able to add MathJax blocks to articles

3.12 Editors should be able to add YouTube embeds to articles

3.13 Editors should be able to add admonition blocks to articles, with different styles for semantics (note/info, warning, danger)

3.14 Editors should be able to add nestable bullet-point or ordered lists to articles

3.15 Editors should be able to add footnotes to articles, referenced from text within the article

3.16 Editors should be able to add a card with their profile details (name, avatar, short biography) at the bottom of articles

3.17 Editors should be able to set whether an article is visible to readers

3.18 Editors should be able to delete articles

3.19 Editors should be able to undo operations within the article editor

3.20 Editors should be able to redo operations within the article editor

3.21 Editors should be able to insert emojis to articles from a searchable list

3.22 Editors should be able to customize keyboard shortcuts for operations

3.23 Editors should be able to customize which operations are visible in the toolbar

3.24 Editors should be able to reorder operation shortcuts shown in the toolbar

3.25 Editors should be able to add a subtitle / figure description to images

3.26 Editors should be able to insert common mathematical symbols from a searchable widget

3.27 Editors should be able to restore cached article edits if they exit the article editor without saving

3.28 Editors should be able to save article templates to reuse in new articles

3.29 Editors should be able to create macros to insert custom article structures or content

3.30 Editors should be able to add notes within articles that are only visible while editing the article

3.31 Editors should be able to simultaneously edit an article in real-time

10.10.1.4 Content exploration and interaction

The content exploration set of requirements concerns the ways that visitors (the "reader" role) will interact with the site's content articles.

4.1 Readers should be able to browse articles by their category

4.2 Readers should be able to search articles by their text content and name

4.3 Readers should be able to browse articles by their tags

4.4 Readers should be able to browse articles by their author

4.5 Readers should be able authenticate themselves using a third-party platform

4.6 Identified readers should be able to post plain text comments on articles

4.7 Editors should be able to reply to comments on articles

4.8 Editors should be able to delete comments on articles

4.9 Readers should be able to delete their own comments on articles

4.10 Readers should be able to subscribe to an RSS feed of the website

4.11 Readers should be able to share articles to common social media platforms (ex. Facebook, Twitter, BlueSky, Pinterest) directly

4.12 Editors should be able to add text patterns to filter out undesirable comments

4.13 The system should hide comments that are filtered out by the set text patterns so they aren't visible to other visitors of the site

4.14 Editors should be able to approve filtered-out comments to be shown publicly

The "categories" would define the sitemap; the published site would have pages corresponding to them, accessed through a navigation bar, which would display a paginated list of article previews linking to the articles's dedicated page where they may be read in full.

10.10.1.5 Event creation

As described previously, the site should also support being used for posting generic announcements for events that take place outside the site. Conceptually, such event announcements can be viewed as a subtype of article with additional functionality:

5.1 Editors should be able to create events under a category

5.2 Editors should be able to set a date, time & duration of an event

5.3 Readers should be able to view upcoming events in a chronological manner

5.4 Readers should be able to register interest in an event using their mail

5.5 The backend should be able to send e-mail reminders to readers who registered interest in an event

5.6 Readers should be able to unsubscribe from e-mail reminders about events

5.7 Editors should be able to set a featured external link for an event

5.8 Editors should be able to view who registered interest in an event

10.10.1.6 Site management

The site management requirements concern site-wide operations and configuration options which would allow the site owner(s) to tailor the presentation of the site to their needs and demographic.

6.1 Admins should be able to set the logo image of the site

6.2 Admins should be able to set the name of the site in the navigation bar

6.3 Admins should be able to set which categories and articles are accessible from the navigation bar

6.4 Editors should be able to rename categories

6.5 Admins should be able to edit the text shown on a sidebar on the home page

6.6 Admins should be able to set which category's articles are shown on the home page

6.7 Admins should be able to choose whether to insert a Twitter/BlueSky feed in the sidebar

6.8 Admins should be able to choose whether to insert a contact form in the site, with subject, reply e-mail and message fields, usable by readers

6.9 Admins should be able to set the language of the static interfaces and labels of the site

6.10 Admins should be able to set which social media platforms articles can be shared to

- Desirable as some supported platforms might not be relevant to the site's demographic; for example, an english-speaking blog wouldn't benefit from a button to share to vk (a russian-speaking social media network)

6.11 Editors should be able to edit the order in which articles appear in a section

6.12 Editors should be able to move an article to another section

6.13 Admins should be able to add external links to the navigation bar

6.14 Admins and editors should be able to view a log of editor and user activity (an audit log; would show new comments, article additions/edits)

6.15 Admins should be able to set the favicon of the site

10.10.2 Use cases

The structure the project used for use case scenarios is the following:

- **Actor(s)**: the user roles involved. This may also include technical actors such as the backend or database, when their behaviour/response is relevant to specify.
- **Goal**: what the actors intend to achieve with the use case, from a user perspective.
- **Preconditions**: the state the system must be in for the use case to be able to happen. Gives context for the use case.
- **Trigger**: the motivation for the actors to begin the use case. If the actor is a user, this helps see the value that the use case has for them.
- **Scenario**: the script of the use case, with the user & system interactions. Divided into 2 parts:
  - **Basic flow**: the script of the use case in an ideal scenario where all actors behave as expected (users enter valid data, the system doesn't error, there's no network issues, etc.)
  - **Alternative flows**: deviations in the basic flow that occur when actors act unexpectedly (users enter invalid data, the system runs into some issue, network problems, etc.). These are helpful to determine how the system should behave in case of such errors.

- **Notes**: additional details and technical requirements to watch out for during implementation, or clarifications.

A single use case may cover multiple functional requirements if they're strongly related. Any cases of "extends" or "includes" are indicated by referencing the relevant UC at the moment they could occur within the basic flow.

Though similar in execution, the use cases for browsing articles by category, tags or author do not use inheritance in the use case diagram as the triggers for them are different - they are initiated differently from each other.

Aside from user roles, the use cases also reference a "system" role to denote important automated tasks to be done by the system - in the project's case, the backend server.

The following sections list the use cases for which complete scenario scripts were written.

10.10.2.1 1. User management

UC 1.1 - Create editor account
Actor(s): admin, backend
Goal: register an account for a content editor on the site
Preconditions: the admin user is logged in
Trigger: the admin navigates to the user control panel and selects the option to create a new editor user
Basic flow: the admin is redirected to a form requesting the information of the new account: username (the login identifier), initial password, display name, and optionally sets their avatar and biography text. Upon confirming, the backend creates the user entry and reports back the success.

Alternative flows:

- If the username is already in use by another account or the username/password don't meet the requirements described in notes, an error message is displayed and the account is not created. The admin can then retry submission without data loss.

Notes:

- The username should be at least 8 characters long.
- The password should be at least 8 characters long, with at least 1 digit and one non-alpha-numeric character to make it less prone to dictionary attacks.
- The UI should inform the user about the aforementioned field requirements.

UC 1.2 - Log-in
Actor(s): editor or admin, backend
Goal: to gain access to the site and content management functions
Preconditions: an editor or admin account exists for the user
Trigger: the user navigates to the "log-in" page

Basic flow: the user is redirected to a page where they are prompted to enter their username and password. Upon confirming, the backend responds with a temporary authentication token and the user is redirected to the control panel.

Postconditions:

- The user gains access to the functionality of the editor or admin role, based on the account type.

Alternative flows:

- If the user enters invalid credentials, an error is shown and the user remains on the page and can resubmit the form without data loss.

Notes:

- The authentication token should expire in a few days.
- The client should persist the authentication token for reuse in later sessions.
- The client should discard the authentication token after its expiration date or if a request made it with it fails due to the token no longer being valid.
- The log-in page should make it clear that log-in functionality is only intended for site staff.
- An admin account should exist in the system by default.

UC 1.3 - Edit profile
Actor(s): editor, backend
Goal: to change the displayed information of an article writer
Preconditions: the editor is logged in
Trigger: the editor navigates to the control panel and selects the "edit profile" option
Basic flow: the editor is redirected to a form where they can set their display name, short biography, and (re)upload an avatar, with the form fields defaulting to their current information. The editor can then press "confirm" to have the backend update their information.

UC 1.4 - Edit permissions
Actor(s): admin, backend
Goal: to change which categories an editor can create & edit articles under to prevent possible abuse
Preconditions: the admin is logged in and the relevant category exists
Trigger: the admin navigates to the "content management" section of the control panel and selects a "configure permissions" option under the category they want to edit permissions for
Basic flow: the admin is presented with a multi-choice widget where they can select which editors have write access to the category. After selecting the editors, the admin presses "confirm" to have the backend update the permissions.

Notes:

- The editor should be able to crop the avatar image and preview it in the UI before uploading.
- If any field was changed, the user should be warned if they attempt to exit the page without confirming the changes.
- Permission to manage a category also extends recursively to all its subcategories. The admin should be warned of this to prevent user error/confusion.
- If a category has no editors with permission to manage it, a warning should be shown beside it in the control panel to prevent user error/confusion.

10.10.2.2 2. Article creation and management

UC 2.1 - Create category
Actor(s): editor, backend
Goal: creating a category under which articles or further subcategories can be organized
Preconditions: the editor account has permission to create categories
Trigger: the editor navigates to the "content management" section of the control panel and selects the "create category" option, either at the root of the content tree or under an existing category
Basic flow: the editor is prompted to provide the user-friendly name of the category as well as its URL identifier and submits it; the backend then registers the category and the content tree view is refreshed so the editor can see the result.

Alternative flow:

- If the category name or URL path is already used by another category under the same parent category, the system shows a warning and prevents submission.
- If while creating a category an admin removes the editor's permissions to manage the parent category, the editor is shown an error message and is prevented from submitting.

Notes:

- The category name should accept unicode.
- The category URL/path should only accept characters that do not need to be URL-encoded, to ensure category URLs are readable.
- In case of error while submitting, the editor should be able to resubmit the form without data loss.

UC 2.2 - Create article
Actor(s): editor, backend
Goal: creating a new content article
Preconditions: a category relevant to the article exists (see UC 2.1)

Trigger: the editor navigates to the "content management" section of the control panel, selects the category they wish to create the article under and select the "new article" option
Basic flow: the editor is redirected to a rich text editor UI where they enter the article's content (see UC 3.1) and hit "submit" once done. Prior to sending the article to the backend, the editor is prompted to configure its metadata (see UC 2.3). Once the editor confirms to post the article, the backend stores the article and the editor is redirected to the URL corresponding to the article so they can verify it was posted and looks as intended.

Notes:

- If the editor must leave the page before finishing the article, they can choose a "save as draft" option to have the system save it without the article being visible to readers on the site. The editor is then redirected back to the content management panel where the draft article will be visibly marked as a draft and available for further editing.
- If an error occurs while submitting the article or saving it as a draft, the editor stays on the same page, is shown an error message and can retry the submission without data loss.
- If while saving the article or draft an admin removes the editor's permissions to manage the category the article is under, the editor is shown an error message and is prevented from submitting.

UC 2.3 - Edit article settings
Actor(s): editor, backend
Goal: changing the presentation and metadata of an article
Preconditions: an article is being edited
Trigger: the editor selects an "edit metadata" option in the article editor or the content management panel.
Basic flow: the editor is shown a prompt that includes settings such as whether comments are enabled on the article, the date & time the articles should be published at (defaulting to current time), whether the article should display this metadata to readers. In this prompt the editor may also mark other editors as co-authors of the article by selecting them from a multi-choice widget, choose whether their short biographies should be displayed under the article, as well as adding multiple text tags to denote the themes the article covers.

Postconditions:

- Pages containing the article are updated to reflect the new settings.

Notes

- When editing the tags of an article, the editor should be shown previously-used tags (from other tags) as suggestions to prevent tag fragmentation (ex. using a "programming" tag in one article, and "programs" tag in another)

10.10.2.3 3. Article editing

UC 3.2 - Insert image into article
Actor(s): editor, system
Goal: add an image to an article being edited
Preconditions: the editor is editing an article
Trigger: the editor selects the "insert image" option from the toolbar
Basic flow: the editor is shown a widget with the different options to insert an image based on its source (see derived use cases; UC 3.2.1 - UC 3.2.3). The editor then selects "confirm" and the image is inserted onto the rich text editor at the previous cursor position.

Notes:

- In all cases the editor is also prompted to enter alt text for the image.
- The alt text of images should be displayed as tooltips when hovering over the inserted image content, both in the article editor and in the published article.

UC 3.2.1 - Upload image
Actor(s): editor, backend
Goal: upload a new image to add to an article
Preconditions: the editor is adding an image
Trigger: the editor selects the "insert image" option from the toolbar UI and chooses "upload image"
Basic flow: derives from UC 2.5. The editor is prompted to select a file from their device, in addition to entering a path to store it under in the backend.

Postconditions:

- The image is previewed onto the rich text editor UI.
- The uploaded image is stored in the backend.

Alternative flows:

- If the user enters a path that is already used by an existing image in the backend, they are shown a warning that uploading the image will replace the existing one in all places where it was previously used.
- If the user chooses to upload a file of an unsupported type, a warning is shown and they are not able to submit it.

UC 3.2.2 - Insert existing image
Actor(s): editor, backend
Goal: add an image already in the site to an article
Preconditions: the editor is adding an image
Trigger: the editor selects the "add image from site" option from the "insert image" widget
Basic flow: derives from UC 2.5. The editor is shown a tree-structured view of

the images that were previously uploaded to the site. The editor then selects a file and clicks “confirm”.

UC 3.2.3 - Insert hotlinked image
Actor(s): editor
Goal: add an image from an external site to an article
Preconditions: the editor is adding an image
Trigger: The editor selects the “add image from URL” option from the “insert image” widget
Basic flow: derives from UC 2.5. The editor is then prompted to enter the URL of the image, after which they can click the confirm option.

UC 3.3 - Add code block to article
Actor(s): editor
Goal: inserting formatted code to an article
Preconditions: the editor is editing an article
Trigger: the editor selects the “insert code” option from the toolbar UI
Basic flow: a widget is inserted at the last cursor position within the article. The editor can then write or paste code into the widget and set the code language through a dropdown besides the widget. Upon changing the language of the code block, syntax highlighting corresponding to the language is applied to the text within the widget.

Notes:

- The selected language should default to a “plain text” option with no syntax highlighting.
- The language selection dropdown should support search if the list of supported languages is long.
- The language selection dropdown should only display when editing the article (and not when viewing it after publishing).
- The widget should display a “copy to clipboard” option when viewed in a published article.

UC 3.4 - Reorder sections
Actor(s): editor
Goal: reordering semantic sections of content in an article
Preconditions: an article is being edited
Trigger: the editor opens the “table of contents” within the article editor
Basic flow: the rich text editor UI displays a table of contents as a tree of headings within the article. The editor can select and drag-and-drop sections to move all its section’s content to be before or after another. After doing so, the text field area of the article editor is updated to move the content to the new corresponding position within the article content.

Notes:

- The content of a section is considered to start at its heading (included) and ends before the next heading.

UC 3.5 - Add table to article
Actor(s): editor
Goal: insert a table content block to an article
Preconditions: an article is being edited
Trigger: the editor selects the "insert table" option in the toolbar UI
Basic flow: the editor is prompted to enter the amount of rows and columns of the table and whether to insert a row of column headers. Upon confirming, a table of row x columns cells is added to the article editor area at the previous cursor position. The editor can then select cells to insert text content within them. The editor can also choose to add/remove rows and columns to the table after its insertion, which will add empty cells in the corresponding axis or remove cells.

Notes:

- The row of column headers should be visually distinct to denote its semantic meaning

UC 3.6 - Add link to article
Actor(s): editor
Goal: inserting a URL link in an article
Preconditions: an article is being edited
Trigger: the editor selects the "insert link" option within the toolbar UI
Basic flow: the editor is shown a widget that prompts them to enter a URL and the text of the link. Upon confirming, a link is inserted at the previous cursor position within the article editor. The editor can click on the link within the article editor to preview its URL in a popover, and can also edit the destination from it.

Notes

- While the link's text is selected, typing will edit the link's text, and deleting all its text removes the link.
- Within the article editor the link is displayed as an HTML link, but does not function as such to prevent accidentally leaving the page.

UC 3.7 - Add footnote to article
Actor(s): editor
Goal: inserting a footnote that can be referenced from earlier parts of an article
Preconditions: an article is being edited
Trigger: the editor selects the "insert footnote" option in the toolbar UI
Basic flow: the editor is prompted to reference an existing footnote from an ordered list or to create a new one. If a new footnote is created, the editor is prompted to enter the footnote text. In both cases, a "[i]" marker referencing the footnote is added to the previous cursor position in the article, with "i" corresponding to the index of the footnote. The editor may delete references by pressing backspace

while the cursor is in front of them. If all references of a footnote are deleted, the footnote itself is also deleted.

Alternative flows:

- If no footnotes exist in the article yet, the toolbar option immediately inserts a new footnote without opening the prompt to reference an existing one or create a new one.

Notes:

- Selecting a footnote with a cursor should preview its footprint in a tooltip (both while editing and in a published post)
- Deleting a footnote should automatically delete all references to it in the text and adjust the indices of the remaining footnotes to be sequential.

10.10.2.4 4. Content exploration and interaction

UC 4.1 - Post comment to article
Actor(s): identified reader, backend
Goal: add a comment to an article
Preconditions: the reader is on an article page that has comments enabled
Trigger: the reader scrolls down to the end of the article where a comment section is shown
Basic flow: the reader is presented with a text field where they can write their comment in plain text, with a character limit. After writing the comment, the reader can press "submit" and the backend will save the comment.

Postconditions:

- The page view is updated to show the written comment.

Alternative flows:

- If the comment fails to be posted to the backend, the reader is shown an error message and can retry submission without data loss.

Notes:

- The text length limit should be visible to the reader. The reader should not be stopped from writing beyond the limit - only stopped from submitting in that case.

10.10.2.5 5. Event creation

UC 5.1 - Create event
Actor(s): editor, backend
Goal: add an event announcement to the site
Preconditions: a relevant category to post the event under exists

Trigger: the editor selects a "create event" option in the content management panel under an existing category
Basic flow: the editor is redirected to a page with a form to fill in the details of the event, date (in the editor's local time), and optionally time of day and duration (in minutes, hours and days). The editor must also add information about the event in the form of an article using the article editor (see UC 2.2, 2.3 and 3.1)

Notes

- The event-specific details may also be edited when editing the corresponding article (as in UC 2.3) with no restrictions on changing the date (to allow fixing user errors).
- If the event date is changed to a past date, readers who registered interest should not receive any notification.
- When viewing the published event, the date and time should be converted to the reader's timezone, and the reader should be able to see the timezone to avoid confusion.