TANSZÉKVEZETŐ

DIPLOMATERVEZÉSI FELADAT

Hanich Péter
Mérnökinformatikus hallgató részére

Sablonvezérelt weboldal-generáló SaaS platform fejlesztése

A szoftver mint szolgáltatás (SaaS) modellek egyre nagyobb teret hódítanak a webes alkalmazások fejlesztésében, mivel költséghatékony, skálázható és könnyen karbantartható megoldásokat kínálnak a felhasználóknak. Egy SaaS alapú weboldal-generáló rendszer lehetővé teszi, hogy a felhasználók technikai ismeretek nélkül is professzionális megjelenésű weboldalakat hozzanak létre és üzemeltessenek. Ezt tovább erősíti a sablonvezérelt megközelítés is, ami nemcsak a fejlesztési időt csökkenti, hanem egységes kinézetet és könnyebb karbantarthatóságot is biztosít.

A feladat célja egy ilyen sablonvezérelt weboldal-generáló SaaS platform fejlesztése, amely lehetőséget biztosít a felhasználók számára egyoldalas weblapok létrehozására, testreszabására és közzétételére. A fejlesztés során kiemelt figyelmet kell fordítani a felhasználói élményre, a skálázhatóságra és a biztonságra.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be a SaaS platformok alapvető felépítését, alkalmazásának sajátosságait és technológiai kihívásait.
- Tervezze meg a platform architektúráját, ide értve a frontend és a backend főbb elemeit is, és implementálja az egyes komponenseket.
- Igazolja a tesztekkel az egyes komponensek helyes működését, valamint mutassa be a platform használatának lépéseit egy konkrét példán keresztül.
- Készítsen javaslatot a portál éles környezetben történő üzemeltetésére, különös tekintettel a szerverinfrastruktúrára, skálázhatóságra és folyamatos integrációs mechanizmusokra.

Tanszéki konzulens: Knyihár Gábor, doktorandusz

Budapest, 2025. március 11.

Dr. Charaf Hassan
egyetemi tanár
tanszékvezető

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék
H-1117 Budapest, Magyar tudósok krt 2. Q.B.207
Telefon: 463-2870. Fax: 463-2871
www.aut.bme.hu

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Development of a template-driven website generator SaaS platform

Master's Thesis

Author
Péter Hanich

Advisor
Gábor Knyihár

December 18, 2025

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Hanich Péter*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2025. december 18.

Hanich Péter
hallgató

Abstract

This master's thesis presents the development of a template-driven website generator SaaS platform. Motivation stems from the observation that current website builders and e-commerce platforms, while powerful, are often feature-heavy, costly, and impose a steep learning curve. For users who only require basic informational sites (e.g., resumes, portfolios, local business pages), the complexity and recurring subscription costs of mainstream SaaS offerings present a practical barrier.

The implemented solution composes a Laravel backend for data and user management, a React-based frontend with a streamlined component based editor, and a lightweight Node.js proxy to programmatically deploy user sites to a containerized environment. Sites are modelled as collections of static pages generated from templates and user content, the platform automates publishing, routing, and basic security considerations. The implementation emphasizes modularity, maintainability, and testability.

Results include a complete prototype, a documented end-to-end usage example, and comprehensive testing validating core user flows. The prototype demonstrates that non-technical users can create and publish a static informational site within minutes, without complex configuration and technical knowledge required.

Production considerations are also discussed, covering containerization, CDN-backed asset delivery, managed Kubernetes, CI/CD workflows, secrets and backup strategies, observability, tenant isolation, and autoscaling.

In conclusion, the implemented SaaS can effectively serve many basic website needs. Future work includes additional components and customizability, revision history and rollback, stronger multi-tenancy, automated backups, richer observability, and evaluation of production-scale resilience and performance.

Kivonat

Jelen diplomamunka egy sablonvezérelt weboldal-generáló SaaS platform fejlesztését mutatja be. A motiváció abból ered, hogy a jelenlegi weboldal készítők és elektronikus kereskedelmi platformok, bár funkcióban gazdagok, gyakran túl sok funkciót kínálnak, költségesek, és meredek tanulási görbével járnak. Azoknak a felhasználóknak, akik csak egyszerű információs oldalakat igényelnek (például önéletrajz, portfólió vagy helyi vállalkozás bemutató oldala), az eljterjedt SaaS szolgáltatások összetettsége és ismétlődő előfizetési költségei gyakorlati akadályt jelentenek.

A megvalósított megoldás egy Laravel alapú backendből a felhasználó és adatkezeléshez, egy React alapú kliensből egy egyszerűsített komponensalapú szerkesztővel, valamint egy könnyű Node.js proxyból áll, amely programatikusan telepíti a felhasználói oldalakat a konténerizált környezetbe. A weboldalakat sablonokból és felhasználói tartalomból generált statikus oldalak gyűjteményeként modelleztem, a platform automatizálja a publikálást, a routingot és az alapvető biztonsági beállításokat. A megvalósítás során a modularitásra, karbantarthatóságra és tesztelhetőségre helyeztem a hangsúlyt.

Eredményként bemutatok egy működő prototípust, egy részletes és teljes példát a használatra, valamint átfogó teljeskörű teszteket, amelyek igazolják az alapvető felhasználói folyamatok helyes működését. A példa szemlélteti, hogy nem szakmai felhasználók percek alatt létrehozhatnak és publikálhatnak egy statikus információs oldalt, bonyolult konfiguráció és mély technikai ismeretek nélkül.

A dolgozat tárgyalja az üzemeltetési megfontolásokat is: konténerizációt, CDN-támogatott tartalomszolgáltatást, menedzselt Kubernetes megoldásokat, CI/CD munkafolyamatokat, titkok és mentések kezelését, megfigyelhetőséget, izolációt és automatikus skálázás szempontjait.

Összefoglalva, a megvalósított, szűkebb funkcionalitású SaaS hatékonyan képes kielégíteni számos alapvető weboldal igényeit. További fejlesztések közé tartozhatnak további komponensek és jobb testreszabhatóság, verziókezelés és visszaállítás, erősebb multitenancy, automatikus mentések, kiterjedtebb megfigyelhetőség, valamint az éles környezetben való üzemeltetés rendelkezésre állásának és teljesítmények vizsgálata.

Chapter 1

Introduction

Websites and web applications are part of everyday life: businesses, clubs, and people use them to share information, advertise, or show their work.

For many small organizations and individuals, for example someone who wants a simple resume site or a small local shop advertising its services, having a website should be quick, affordable, and easy to manage. In practice, creating and keeping a website running often requires time, money and technical knowledge. Hiring a developer or an agency can be expensive, building and maintaining a site yourself can be confusing, time-consuming and requires deep technical knowledge.

A software-as-a-service (SaaS) platform for websites tries to solve this problem by offering an all-in-one service: easy tools to build the site, hosting, so the site is always available, and simple ways to update content without requiring professional skills.

Many such platforms exist on the market, including popular solutions like Squarespace, Wix, GoDaddy, and Shopify. However, most of them are tailored to complex needs and come with a steep learning curve. They typically provide either no free tier or a very limited one, and as a site grows more complex, a paid subscription becomes unavoidable.

Consider Wix, one of the most popular website builders with millions of users worldwide. It offers an extensive drag-and-drop editor with hundreds of templates, e-commerce capabilities, advanced SEO (Search Engine Optimization) tools, and marketing integrations. While powerful, this breadth of features comes with complexity: users must navigate through numerous menus, settings, and configuration options. Wix's free tier displays prominent Wix branding and provides limited storage and bandwidth, making it unsuitable for professional use. To remove branding and access essential features, users must subscribe to paid plans, with full-featured e-commerce plans costing even more.

Similarly, Shopify targets e-commerce specifically, offering inventory management, payment processing, shipping integrations, and detailed analytics. Its comprehensive feature set makes it ideal for serious online retailers, but also means a steeper learning curve as users must understand product variants, shipping zones, tax settings, and more. Shopify requires a minimum subscription with no free tier, and transaction fees apply unless using Shopify Payments.

While these platforms excel at their intended use cases, complex websites and professional e-commerce, they are overcomplicated and cost-prohibitive for users who simply need a basic informational website. Someone wanting to display a resume, showcase a portfolio, or create a simple landing page for a local service does not need inventory management, payment gateways, or advanced marketing automation.

This thesis presents a free, lightweight SaaS platform designed specifically for these simpler use cases: websites made of static pages that display essential information. By focusing on basic functionality rather than comprehensive features, the platform offers a significantly reduced learning curve. Users can create and publish a site in minutes without navigating complex settings or paying subscription fees. The trade-off is intentional: while the platform cannot match the advanced capabilities of Wix or Shopify, it provides exactly what is needed for straightforward informational sites, nothing more, nothing less.

1.1 Overview

In Chapter 2, I review background material needed to understand the concepts I used later. This includes the idea of software-as-a-service (SaaS), an overview of the Laravel backend framework, a summary of React and client-side technologies, and an introduction to Kubernetes and key deployment concepts.

Chapter 3 defines the functional and non-functional requirements of the platform and presents the system-level architecture. I describe the purpose and general behaviour of each component and how they work together.

Chapter 4 begins the implementation, focusing on the backend: the data model, core services, security and identity handling, integrations with external services, and the logic that provides the platform's main features.

Continuing the development, Chapter 5 covers the frontend implementation, presenting the user interface, components, and typical user flows for site creation and management.

Finishing the main development areas, Chapter 6 addresses deployment and operations by describing the lightweight proxy application I used for deploying and serving user websites.

Moving on, Chapter 7 provides the testing strategy and results to demonstrate and validate correct behaviour, and offers a full end-to-end example showing how an end user would use the SaaS platform.

Chapter 8 discusses differences to consider when moving to production and practical considerations for operating the service.

Finally, Chapter 9 summarizes my work and suggests directions for future improvements.

Chapter 2

Literature review

2.1 Software as a Service

In order to develop a software as a service application suite we first have to explore what a SaaS entails. To do this, an overview and understanding of cloud computing and its related terminologies is necessary. ISO defines cloud computing as a "paradigm for enabling network access to a scalable and elastic pool of shareable physical or virtual resources with self-service provisioning and administration on-demand [1]." The National Institute of Standards and Technology (NIST) defines five essential characteristics, four deployment models and three service models for the cloud [2].

The essential characteristics are the following:

- On demand self-service: resources can be provisioned without additional human assistance.
- Broad network access: users can access capabilities through standardized methods over the network.
- Resource pooling: the provider manages their cloud resources in such a way that it can serve multiple customers flexibly.
- Rapid elasticity: capabilities are scalable and re-assignable on demand.
- Measured service: the cloud monitors resource usage, and it is controlled and automatically optimized.

The aforementioned deployment models are:

- Private cloud: a single organizational entity uses the cloud.
- Community cloud: a community composed of multiple organizations or users uses the infrastructure.
- Public cloud: the cloud is open for use to the public.
- Hybrid cloud: the cloud is a mix of the models mentioned before.

All the above deployment models may be either on- or off-premise and additionally owned or managed by the organization(s) using them, third parties or some combination of the former two.

Most importantly for the topic, the three service models are:

- Infrastructure as a Service (IaaS): the provided resources are fundamental to computing, such as network, storage, or processing power.
- Platform as a Service (PaaS): the provided resources allow the customer to deploy applications to the cloud provider's platform without configuring the underlying computing infrastructure.
- Software as a Service (SaaS): "The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings [2]."

There are many cloud providers available for use, most notably Amazon Web Services (AWS), Microsoft's Azure, and Google Cloud Platform (GCP), which boast numerous amounts of services belonging to the models described previously.

2.1.1 Aspects and challenges of SaaS

After establishing a definition of SaaS and its related terminologies, lets dive into the different aspects and challenges of developing, maintaining, and operating one.

"Software as a service (SaaS) is a software delivery model in which third-party providers provide software as a service rather than as a product over the Internet for multiple users [3]." SaaS applications hence are also web applications, and are built with web technologies and frameworks. However, not all web applications are SaaS due to the aforementioned delivery model. In a way SaaS is a specific subset of web applications.

There is not a fixed defined set of characteristics that a SaaS has to meet in order to be considered one, however there are some commonly recurring ones that most take into account, such as multi-tenancy, customization, and scalability [4].

Most SaaS applications support some form of multi-tenancy. Multi-tenancy is the ability to share the same (and single) hosted instance of the application between multiple tenants, instead of deploying the application separately for each tenant. A tenant is a user or group of users (e.g., an organization) with the same share of the service [5]. Commonly, people refer to multi-tenancy as the ability to have multiple organizations use the service with their own share and users, however, as we can see, this is not a strict requirement. There are many ways to achieve multi-tenancy with different amounts of complexity in concern areas like architecture, scaling, and security. The methods usually differ by how separated each tenant's share is on a database, data model and infrastructure level [6].

This goes hand-in hand with the next common characteristic, which is customization. SaaS applications must support a level of customization to meet all tenants' needs. This allows tenants to adapt the service more seamlessly into their workflows, processes, and requirements. The ability to change aspects also reduces repetitive tasks between users of the same tenants. Customization can range from interface level differences to full on tailoring of the tenant's share of the application down to the platform or even infrastructure level. Each added aspect of personalization, however, increases complexity, due to which a balance must be struck between customization and standardization [7].

Scalability is the ability of the service to handle increasing workloads and usage needs. Generally there are two methods of scaling: vertical scaling (also known as scale-up) and horizontal scaling (also known as scale-out). Vertical scaling involves giving more resources to the machine running the service, while horizontal scaling means distributing the application on multiple machines. In a cloud environment, consumers rely on a scale-out solution more by default, due to the available amount of resources, and also due to the fact that increasing a machine's resources is only feasible up to a level. However, most complex production systems usually combine the two forms in some way. Not only that, SaaS platforms commonly employ either a two-level or a generalized K-level scalability structure where each level is a separate dimension of the service capable of scaling independently. The platforms also make scaling solutions tenant-aware so that each tenant gets scaled for their respective needs [8].

Of course, plenty more aspects could be reviewed and explored regarding SaaS applications, however, the previously mentioned concerns and areas are sufficient to get a general understanding required for the topic.

2.2 Laravel

In this section I will be reviewing Laravel, and its most important features regarding the thesis. Laravel is a popular PHP web framework created by Taylor Otwell and first released in June 2011 [9]. He based it upon Symfony, which is another PHP web application framework first and foremost, but also contains a wide variety of PHP tools and libraries. It was also heavily inspired by Ruby on Rails.

"Laravel is, at its core, about equipping and enabling developers. Its goal is to provide clear, simple, and beautiful code and features that help developers quickly learn, start, and develop, and write code that's simple, clear, and lasting [9]." The tools and ecosystem of the framework allow developers to write more scalable and maintainable code.

Laravel uses the Model-View-Controller (MVC) architecture.

Models represent logical entities or resources in the application and are implemented by the Eloquent ORM (Object-Relational Mapping) of Laravel. Controllers provide methods to handle incoming requests and return an appropriate response or serve a view. Views display the aforementioned response from a controller method. Laravel provides the built-in Blade templating engine to return HTML (HyperText Markup Language) views to the users [10].

An example diagram visualizing this architecture can be seen in Figure 1.

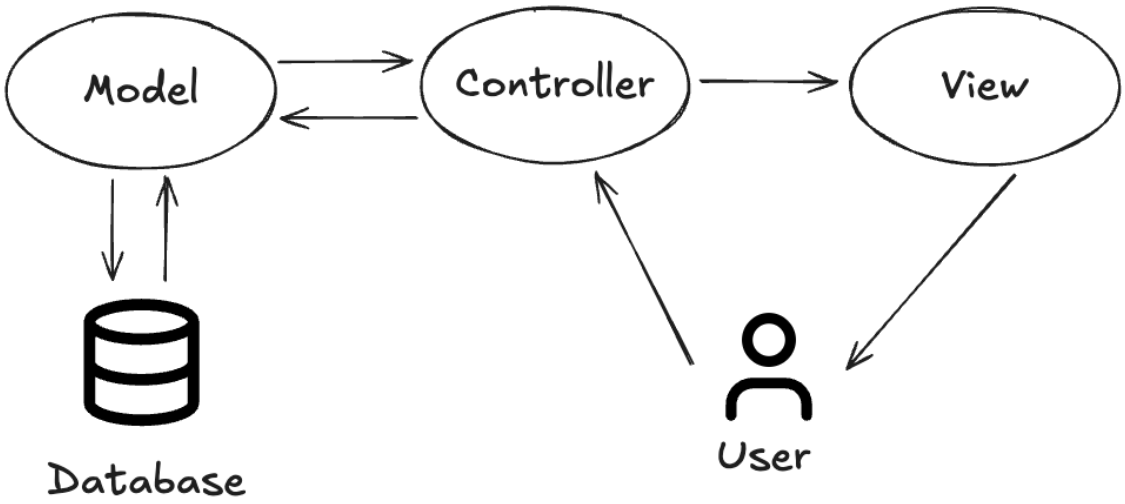


Figure 1: Model-View-Controller architecture

2.2.1 Request Lifecycle

Aside from understanding the general architecture of Laravel, getting a grasp of its inner workings through the request lifecycle allows one to better understand on a high-level how the framework truly functions. As with all web applications, a web server serves Laravel, most commonly Apache, Nginx, or nowadays FrankenPHP (built on Caddy). The starting point of the application is the `public/index.php` file, to which the web server directs all requests. From here, the file retrieves a Laravel application instance from the `bootstrap/app.php` file. Next, it sends the incoming request to the HTTP (HyperText Transfer Protocol) kernel, which bootstraps all necessary service providers, and executes application level middleware. These include the database, queue, validation, and routing components, among many others. Following this, the router will dispatch the incoming request to route-specific middleware, after which a route or controller will handle generating a response. Finally, the response will be sent out through the HTTP kernel and application instance [11].

A diagram of the complete request lifecycle and MVC can be seen in Figure 2

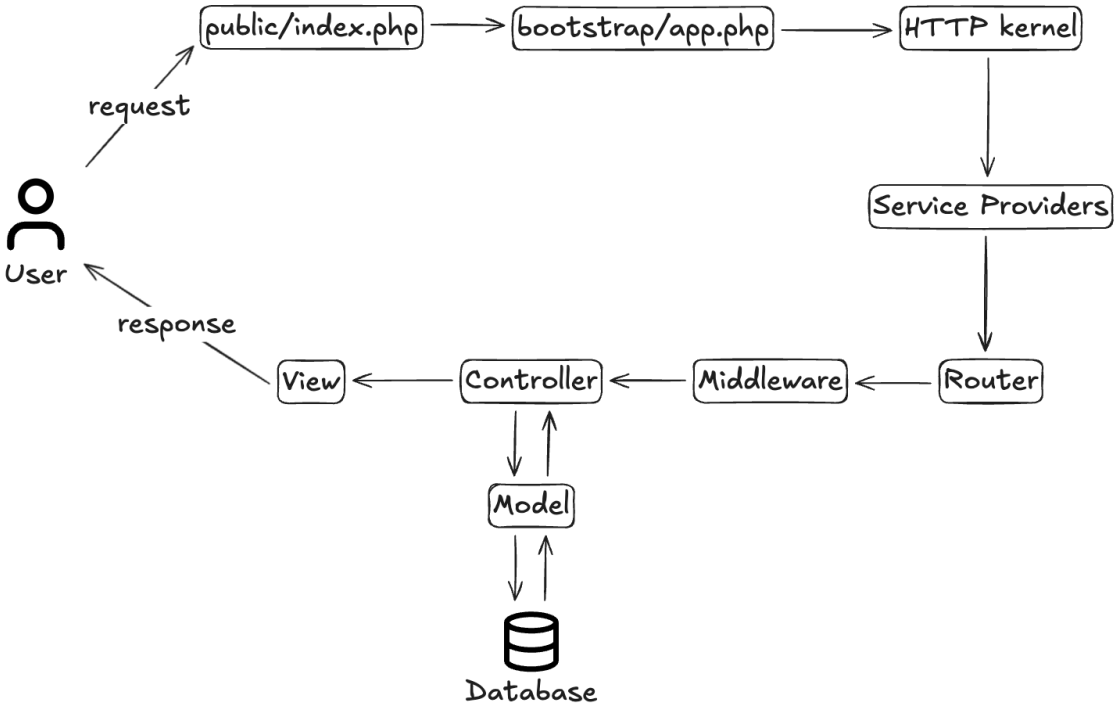


Figure 2: Laravel request lifecycle

2.2.2 Artisan

Laravel includes Artisan, a command-line interface that provides dozens of helpful commands for common development tasks. Artisan commands cover a wide range of functionality, including database migrations, seeding, queue management, scheduled task execution, and code generation for models, controllers, and other framework components. Developers can also create custom Artisan commands by extending the base command class, defining input arguments and options, and implementing the command logic. Commands can be invoked interactively or programmatically from within the application, making them useful for both manual development workflows and automated processes [12].

2.2.3 Ecosystem

Laravel provides many built-in functionalities for use, such as mailing, testing, queues, notifications, storage, authentication, and authorization, among many. All of these capabilities can be extended, or new ones can be added through the rich package ecosystem surrounding the framework [13]. The creators of Laravel also maintain numerous first party packages, but there are also many other third party big packages and package maintainers. Most notably Spatie [14], whose team maintains many packages and tools for the framework, and FilamentPHP, which is a plugin extendable admin panel and UI (user interface) framework [15].

2.3 React

This section provides a short overview of React and more importantly, the packages I'm going to use throughout the implementation. React is JavaScript library for building

user interfaces through reusable and composable components. It leverages a virtual DOM (Document Object Model) that decides whether the component has to be re-rendered or not, based on the state and properties (props) of the component and their changes [16].

Developers commonly use React to implement client-side rendered single page applications (SPA), in which the server sends only one HTML file and the bundled JavaScript code dynamically rebuilds the DOM, without needing to make additional requests to the server [17].

2.3.1 SPA routing: TanStack Router

Since SPA-s only serve one HTML file from the backend, they need some way to know which component to render for which web route. For this purpose, developers use one of the many available routing libraries. In React, the most popular routing library has been React Router for a long time [18]. However, recently React Router became part of the Remix framework, which was later rebranded as version 7 of React Router and introduced declarative, data, and framework modes to the router [19]. Due to these changes and the fact that I planned to use TanStack Query (formerly known as React Query), I opted to use TanStack Router instead, for easier use and better integration.

TanStack Router provides many features out of the box, including but not limited to file- and code-based routing, caching, prefetching and data loading, route contexts, search, URL parameter validation, native TypeScript support, and type safe navigation [20].

The thesis project uses file-based routing, which we should take a closer look at. When we create our React application, it creates a router instance with the default context (e.g., auth, theming) and settings like error boundaries, cache stale time, and view transitions. We also have to create a routes folder and a root route where we can configure application wide error, not found, and loading components. From here on out, all tsx files (except the ones with special naming) will be treated as routes and will be auto managed by the router. This means creating route stubs and auto updating them on changes. From the structure of the routes folder a route tree file will also be generated, which provides the type safe route checking features. However, files can use special naming conventions for given purposes. I demonstrate the ones relevant for this project in Table 1.

Table 1: TanStack Router naming convention examples

Symbol	Example	Function
___	___root.tsx	The root route
_ (prefix)	_layout.tsx	Layout route
- (prefix)	-button.tsx	Excluded, used for colocation
.	app.posts.tsx	Route nesting: /app/posts
$	posts/$postId.tsx	Route parameter

Folders can also be used for nesting instead of the dot notation with the prefixes outlined above. The main difference is that in folder-based grouping, the file serving exact path of the folder (e.g., `/posts`) must be named `index.tsx` while layout routes should be named `route.tsx`. Folders can also be used for logical grouping without adding additional routes segments by putting the folder name in parentheses (e.g., (auth)) [20].

2.3.2 TanStack Query

"TanStack Query (formerly known as React Query) is often described as the missing data-fetching library for web applications, but in more technical terms, it makes fetching, caching, synchronizing and updating server state in your web applications a breeze [21]."

The library integrates seamlessly with TanStack Router and helps replace boilerplate API (Application Programming Interface) data fetching codes relying on fetch, useEffect and state management. Moreover, the package provides intelligent and automatic caching, background updates, and request deduplication. TanStack Query has three fundamental concepts, queries which fetch data (GET), mutations that modify data (PUT, POST, DELETE), and the query client, which is the coordinator of the other two [22].

Under the hood, the library uses Stale-While-Revalidate (SWR) caching, which was put forward by RFC 5861 [23]. SWR is a client side caching strategy that works the following way. The client caches all queries upon fetching. The next time the user would need the result of that query the cache serves the **stale** data immediately, **while** triggering a re-fetch in the background. Once the server returns the fresh data, it is **revalidate**d and the application updates the UI without blocking. The client identifies queries by their query key, which is most often the path of the query and all or any associated URL parameters. The cache can become stale after a set timeout, be manually invalidated, or on window or page switches and network reconnections [24]. This can be seen illustrated in Figure 3.

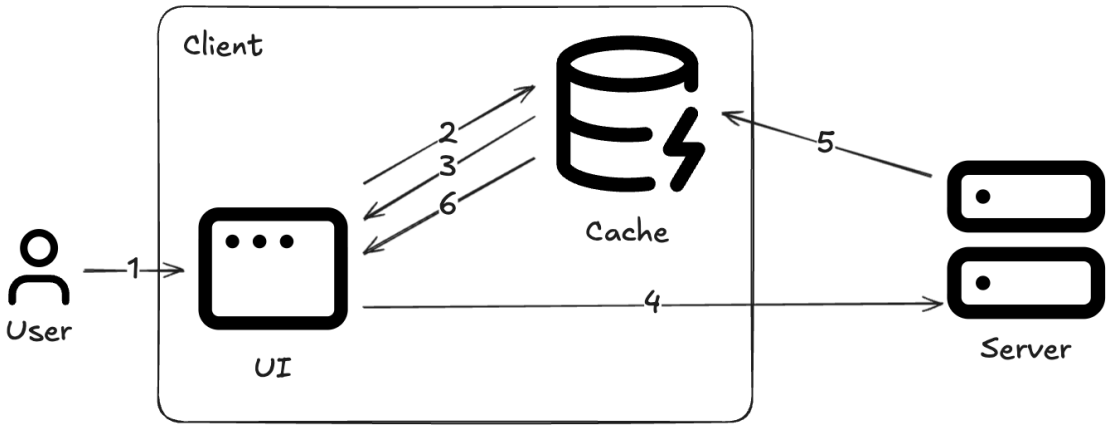


Figure 3: Stale-While-Revalidate caching

2.3.3 Editor libraries

Due to JavaScript's, and by extension React's rich package and library ecosystem, there exist multiple options to choose from regarding all needs. For the basis of the editor functionality I investigated two libraries, namely GrapesJS and CraftJS. They both offer capabilities to build one's own editor with drag-and-droppable UI elements [25]. Ultimately, I decided against GrapesJS as CraftJS is built more around React, while the former relies more heavily on pure JavaScript elements.

CraftJS is a React library for building extensible drag-and-drop page editors. Unlike traditional page builders that provide a fixed set of components and limited customization, CraftJS provides a flexible foundation that allows developers to define their own editable components, control rendering logic, and implement custom drag-and-drop behaviours. The library manages the editor state through a centralized store, handles serialization and

deserialization of the component tree, and provides hooks for accessing and manipulating editor content programmatically.

Components in CraftJS are defined by wrapping standard React components with editor-specific configurations that specify draggable regions, droppable areas, connection points, and editable properties. The library distinguishes between user-facing components (rendered in the canvas) and their corresponding settings panels (displayed in toolbars or sidebars), allowing complete separation of editing interface and output. CraftJS also supports constraints and rules, such as restricting which components can be dropped into certain containers or limiting nesting depth.

The library's modular architecture makes it well-suited for building domain-specific editors tailored to particular use cases [26].

2.4 Kubernetes

In this section I'm going to provide an overview of Kubernetes and its minimal subset of features required to understand this thesis, as Kubernetes is a vast and complex topic by itself. I'm assuming a basic understanding and familiarity of Docker, containerization, and virtualization from the reader.

"Kubernetes is an open source container orchestration system. It manages containerized applications across multiple hosts for deploying, monitoring, and scaling containers. Originally created by Google, in March of 2016 it was donated to the Cloud Native Computing Foundation (CNCF) [27]."

Kubernetes has multiple distributions, can be installed standalone, or used through the services of popular cloud platform providers. It is used declaratively to configure the desired state of a system or an application. Most often users interact with Kubernetes through its command line interface (CLI) with the `kubectl` command. This interacts with the underlying API of Kubernetes.

Pods are either a single or a group of containers with shared network and storage resources, and are the smallest configurable and manageable deployment unit in the system. A Pod can have any number of Init Containers. They must run successfully and in a set order, and can be used to do setup tasks for their respective Pod. Replicasets maintain a set number of replicas of the same pod, while Deployments provide a declarative way to update Replicasets or Pods. Finally, a Service can be used to expose a network application that is running a set of pods behind a single outward facing endpoint [28].

2.4.1 Ingress, Gateway API

Since Pods can be short-lived, often recreated, and have multiple replicas, they need a specialized service to access HTTP and HTTPS (HyperText Transfer Protocol Secure) routes from outside the cluster. This is provided by the Kubernetes Ingress or Gateway API.

"Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource [29]."

The Gateway API is, in a way, a successor to the Kubernetes Ingress, and it provides a more modern and scalable solution to route and traffic management [30].

Both the Ingress and Gateway API need a controller or provider to function. One such industry standard is Traefik, which can serve both functionalities, and is an open source application proxy [31].

A small illustration of the above concepts can be seen in Figure 4.

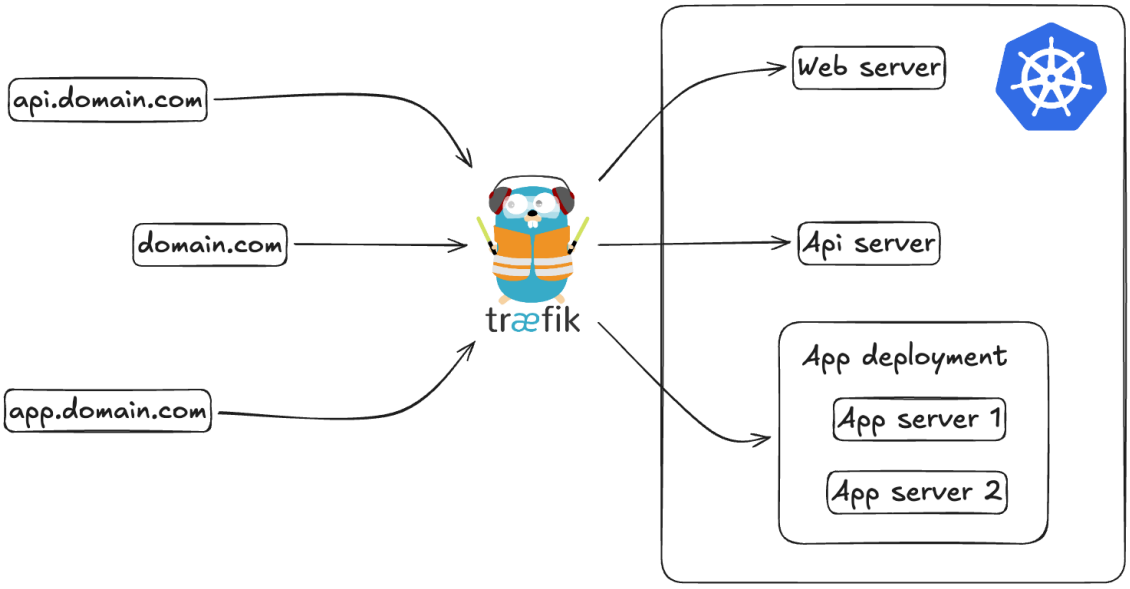


Figure 4: Traefik Gateway API

2.4.2 Client Libraries

Kubernetes provides official client libraries for several programming languages, including Go, Python, Java, JavaScript, and .NET, enabling programmatic interaction with the Kubernetes API. Additional community maintained libraries are also available for most common programming languages and environments.

These libraries enable developers to communicate with the cluster through code instead of calling the REST API directly. They provide typing support for requests, responses, and resources as well. Moreover, they handle common tasks, which include authentication against the cluster, using and discovering service accounts, and understanding and parsing the `kubeconfig` to read addresses and credentials [32].

Chapter 3

Architecture

The goal of the thesis is to develop a template-driven website generator SaaS platform that allows users to create, customize, and publish single-page websites. While the specification only requires the ability to generate single-page websites, I decided to implement the application in a way that allows users to create sites with multiple pages and navigation between them. This solution still allows for single-page websites, but offers more flexibility at the same time.

Special attention must be paid to scalability, security, and user experience (UX). The system needs to provide users with the tools to create and design static websites for unique use cases, publish them, and handle continuous integration and delivery of changes. All of this has to be implemented in a way that doesn't require users to have any knowledge of the underlying technical details. Thus, they should only have to be concerned about the content and appearance of their site.

A platform that can achieve this task from one end (the user) to the other (deployment) is inherently complex and requires the cooperation and integration of many technologies. To begin with, I will present the overarching high-level architecture of the complete system, introduce each component, then explain the ways they integrate and interact in detail. The architecture itself can be viewed in Figure 5.

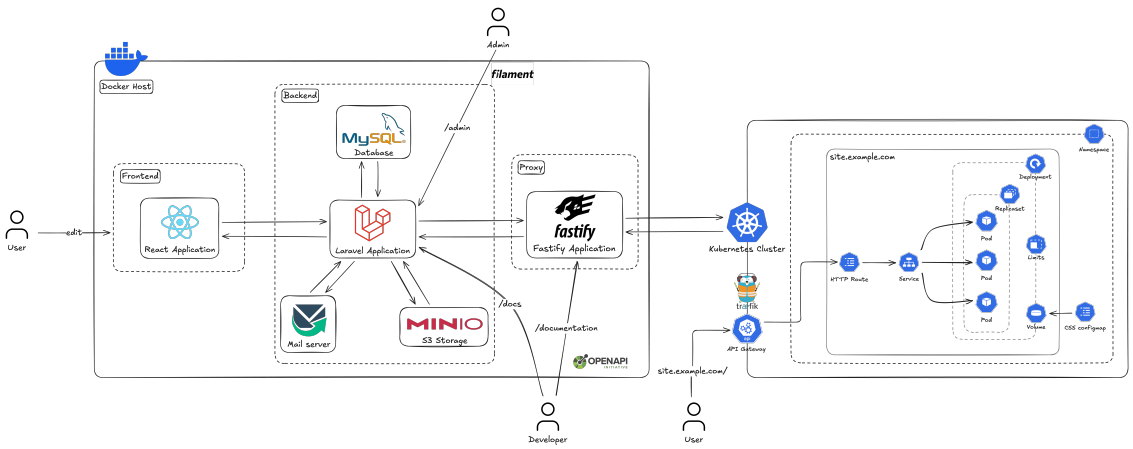


Figure 5: System-level architecture

The system I designed has two main components: a Docker host and a Kubernetes cluster. The Docker host contains the three main applications of the SaaS, each running inside containers and having their own Docker Compose configuration.

I will be building the frontend primarily using React with the Vite build and bundling tool. It is going to communicate with the backend through a REST (Representational State Transfer) API. The CraftJS library will provide the drag-and-droppable editor capabilities, which is going to allow users to design and compose their websites using predefined blocks and templates.

The backend relies mainly on Laravel, but is also going to use MySQL for the database, Mailpit, which is a mailing testing tool that acts as an SMTP server, and MinIO, an S3 compatible object store. The backend will serve the admin panel using FilamentPHP for the implementation, while I am going to use the Scribe package to generate the backend's API documentation using the Scalar documentation template. The backend will communicate with both the frontend and the proxy application also through REST.

The Fastify NodeJS web framework is going to provide the basis for the proxy application and the Kubernetes JavaScript library to connect to the running cluster. It will also serve its own OpenAPI specification using the native Fastify OpenAPI plugin. For the reason of this functionality having its own separate application, refer to Chapter 6.

The proxy is going to connect programmatically to the Kubernetes cluster to deploy the created websites using its official JS (JavaScript) client library. The websites are will all be deployed to a given namespace ('default' namespace if not changed) using a Kubernetes Deployment. The Deployments will have the common style CSS (Cascading Style Sheets) attached as a volume through a ConfigMap. Furthermore, each Deployment by default will have a revision limit of 10, default resource limits set, and a replica requirement of one. In production, these values could obviously be reconfigured and scaled up. Finally, the application will also create a Service for each website, as well as an HTTP Route to connect to the Traefik API Gateway of the cluster for routing.

A user thus will be able to create and design their website on the frontend, which will then be saved to the backend and through the proxy app deployed to the running Kubernetes cluster. Outside users will then be able to visit the deployed website through the Traefik API Gateway acting as a reverse proxy.

Chapter 4

Backend

4.1 Setup

I implemented the backend primarily using the Laravel PHP web framework as stated beforehand. As the first step, the application has to be installed and set up. This can be done with Laravel's own installer, where many application templates can be selected. A React template is available, however, that uses Inertia.JS by default, which is a package that enables the frontend to be server-side routed. I wished to opt out of this and develop a traditional API based SPA, so I decided to use the minimal setup. So Laravel will serve as an API backend, and the view layer of the MVC architecture will be entirely handled by the frontend.

After installation, I decided to install the Laravel Sail package, which provides a preconfigured development Docker Compose setup, and also provides options to install additional components into our environment. The extra containers in our case will be the MySQL database, the Mailpit SMTP test server, and the MinIO S3 compatible object storage. However, the Laravel creators published Sail as a package, which, by default would need a local PHP and Composer (PHP's package manager) installation. This would kind of defeat its purpose. Luckily, this can be avoided by using a temporary PHP container to quickly install initial dependencies and then use the Sail application for further required packages.

Afterwards, the provided `.env.example` file needs to be copied to `.env` and the appropriate environment variables need to be set, including the mail server, database, and object storage connections. In the MinIO object storage, a bucket to store files in must be created and set to public, so that the links generated to the files can be accessed by the other applications. Finally, the application key must be initialized using the Artisan CLI and the default database migrations have to be run.

4.2 Eloquent ORM

I begin the backend implementation from the foundational level, focusing first on the Eloquent ORM. The database schema of the backend including the most important tables can be seen in Figure 6.

Laravel has some other default tables, related to built-in features such as jobs and caching that I omitted from the diagram for better clarity as they will be unused in the application.

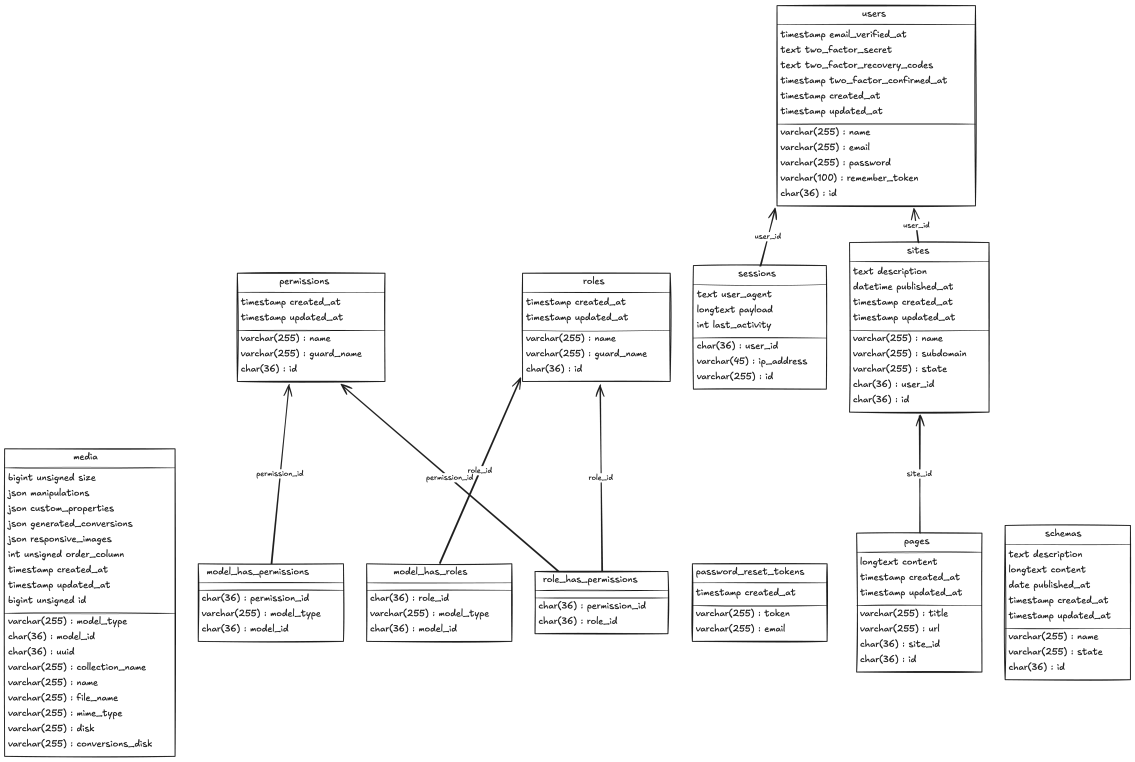


Figure 6: Database schema

All the tables have a corresponding migration file that defines a migration class with an up and down method. In these methods, using the provided schema class, we can define the blueprints of the tables with their fields and data types, and how to create and drop them. The Eloquent ORM will then take care of translating it to the query language of the chosen database management system when running the migrations.

The users table is auto generated, the only change I made is regarding the ID (Identifier) field. In all tables and models, I opted to switch from the default integer ID to UUIDv7 (Universally Unique Identifiers version 7), which are time-based sortable universally unique identifiers composed of 32 hexadecimal digits. The application uses the simplest form of tenancy where each user is a tenant, as that is sufficient for the current requirements.

All tables also have two timestamp fields by default: `created_at` and `updated_at`. Laravel automatically manages these and casts them by default to the included carbon class, which provides helper methods to manage times professionally.

The Spatie Laravel Permission package adds the role related tables, which I will talk about in more detail in Section 4.4. On the other hand, the Spatie Media Library provides the media table, which will be used to store profile pictures and the HTML files of the static pages. The application uses the password reset tokens and sessions table for authentication related features. The Sites table stores the websites of the users. Each site must have a unique subdomain that will be assigned to it upon creation, and which cannot be changed later.

Also worth mentioning, that each site also has a state that can be either draft or published, to track changes and whether they are visible to the end users. This is implemented using the Spatie Model States library, which adds an easy to configure state machine pattern support with helper functions, such as checking viable transitions. It uses an abstract state (`PublishingState`) class with actual states being the concrete implemen-

tations (`Published`, `Draft`). Allowed transitions and transition handlers are also easy to configure and customize.

A site can also have pages which need a corresponding URL segment (e.g., `/about`) for routing. The database stores the pages content in a zlib compressed and base64 encoded format for better transfer and storage efficiency, but can be cast to the uncompressed version if needed. The structure of the content will be further specified in Chapter 5, but in short, it is a JSON file encoding a tree structure. Pages also take advantage of the media library and S3 object storage to store the page's HTML in files. Last but not least, the schemas table is a mix of the previous two, having both a content identical to that of the pages table, and the same published and draft state as sites. Schemas allow administrators to create and publish page schemas that users can base their own pages off of.

The models paired with the tables contain most of the business logic that is independent of routes (heavy models). This includes defining all relationships and their inverses, like in the case of sites and pages a `hasMany` and its inverse `belongsTo` functions, defining fillable fields that can be assigned from requests, and any custom casts and attributes. The former include casting to the aforementioned state classes while the latter includes accessing the decompressed and decoded content and the storage and retrieval of the HTML content of pages. Models register media collections for storing files by setting the name, accepted MIME (Multipurpose Internet Mail Extensions) types and the amount of files they can hold.

Finally, for most of the mentioned models, I extended the special boot method with additional functionality. Inside this, the CRUD functionality can be extended with additional logic. Examples include ensuring that sites transfer to the draft state when one of their pages gets updated, ensuring that a home page (`/`) exists for all sites, and relationship and media clean up logic upon model deletion.

A feature related to models is factories, which allow the developer to create instances of models with set default attributes and persist them to the database. Another related feature is database seeding, which allows us to populate the database with initial data. Since I use the aforementioned capabilities mostly for testing, I will go into more detail about them in Section 7.1. For development purposes, however, I use the user model's factory to create a super-admin user and seed roles and permissions into the database. I will talk about the previously mentioned features more in Section 4.4.

The final component related to the Eloquent ORM I used and want to mention here is API resource classes. They provide a way to transform models into the JSON format that is returned to the application's frontend. It can be thought of as a Data Transfer Object (DTO) implementation, as resource classes allow fine granular control over the returned properties. Fields can be hidden from models, relationships can be included with select fields, and any custom or calculated field can be defined. Moreover, resource collection classes can be defined, which describe how to return a collection of the given model as opposed to returning a singular instance. Furthermore, resource classes can also be defined without them being connected to a given model.

For each model and endpoint, I defined a corresponding resource and resource collection class. In order to parse requests and responses from the proxy application, I implemented custom deployment resources as well. I organized all of these classes according to the API versioning, to which refer to Section 4.5.

4.3 Authentication

Moving on from the ORM features, next, I will describe the implementation of the authentication capabilities of the backend. The combination of two packages maintained by the Laravel team provide the functionality, the first being Laravel Sanctum, while the second is Laravel Fortify.

Laravel Sanctum provides API token based, SPA session based, and Mobile token based authentication methods and provides CORS (Cross-Origin Resource Sharing) and CSRF (Cross-Site Request Forgery) protections for better security. Out of these three, I only used the SPA session based authentication using cookies for the frontend. To begin with, the `SANCTUM_STATEFUL_DOMAINS` environment variable has to be set with the frontend and proxy URLs. Next, in the application bootstrapper the stateful API global middleware has to be applied. Lastly, the CORS configuration should be published, and the allowed paths have to be set.

In contrast, Laravel Fortify provides a frontend agnostic implementation of authentication features for the backend. In other words, it implements the routes and controllers for login, registration, password reset and confirmation, email verification, and two-factor authentication if needed. Each of these features can be enabled or disabled in its own configuration file. Additionally, Fortify can provide Blade templating based views for each of these features if needed, however, since I will use an SPA frontend, I disabled them. I also opted out from the two-factor authentication capabilities, the rest, however, are fully supported by the developed application suite.

One small caveat with using a separate React frontend is that the default password reset and email verification links that are generated by the backend navigate to itself instead of the frontend. I fixed this by overriding the link generator methods in the application service provider. To finalize the authentication implementation, the user model has to be extended from the `Authenticable` class, and has to implement the `MustVerifyEmail` interface.

4.4 Authorization

Strongly related to authentication is authorization. Laravel provides gates and policies built-in for this purpose. Gates are closures that define whether the application authorizes the given user to make a certain action.

However, I won't go into more detail about them as instead I will be using Spatie's Laravel Permission package. It allows developers to manage permissions and roles in a database. Since it provides native enum support, I implemented a role and a permission enum class.

The defined roles in the system are user, which grants basic permissions after registration, admin who can manage users, schemas, and access the admin panel, and lastly, the super admin. The last is the only role where I use Laravel's gates natively, to grant super admins implicitly all permissions in the application service provider before any other checks.

I defined permissions separately for each model and each CRUD method, as well as a unique permission for allowing access to the admin panel.

Since the package is storing authorization logic in the database, it needs to be available after application setup. The straightforward method to do this is to write a seeder class that initializes the default values and what maps the appropriate permissions to the corresponding roles.

Now, I only needed to set up the policy classes that use these roles. For each model, an appropriate policy class can be created that checks whether the given user can be allowed to make the specified action. Inside these functions we can check users against ownership of the roles defined before. A special use case of policies I used is inside the page's policy where I check against the page's URL and prevent overriding or deleting the homepage.

4.5 Core Logic

The first part of the core CRUD logic is defining the API routes in the routes folder. Here, routes can be defined for web, API, and console use cases. The web handles standard web requests if Laravel were to serve the frontend as well. Hence, the React SPA takes care of this functionality, the only route handler I defined here is a redirect to the frontend's index page. The console routes can be employed to define custom artisan commands, so they remained unused in this case.

I decided to use API versioning for a cleaner design and architecture, so I prefixed all of my routes with `v1`. If, in the future, a new API version would need to be released with breaking changes, this allows for a cleaner transition while keeping backwards compatibility.

I grouped all routes by their respective controller that handles the incoming request. There is a controller for each model, an avatar controller for handling profile pictures, a dashboard controller for returning the frontend dashboard's content, and a deployment controller for handling making deployment requests to the proxy application. This is worth talking about more in depth, which will be done in Section 4.6. For each group authentication, I applied the middleware provided by Laravel Sanctum. Apart from the user and dashboard controllers, the others also have the verified middleware applied onto them that ensures the user making the request must have verified their email.

The routes also have the appropriate policy methods called on them that I talked about in Section 4.4, to check against the given user's roles and capabilities. Route parameters specified with e.g., `{param}` also get automatically dependency injected into the controller method.

Next in line, I implemented Laravel's form request classes for the POST and PUT routes. They encapsulate the custom validation logic for each incoming request, which can be defined either using one of the many built-in validation rules or by creating a custom rule via a callback. A custom rule I implemented is checking that each path segment mapped to a page must be unique within the parent site. I used the predefined rules to validate against types, null checks, optional body parameters, sizes, and file MIME types among many. Custom auth can be implemented here, however, since it is already taken care of by the route policy and middleware, I only provided a simple check to make sure the user exists. I also defined a body parameters function in each form request to help the OpenAPI schema generation.

Due to the separation of concerns into the many capabilities, most controllers are incredibly thin. Frequently, they are only comprised of retrieving the validated form data, performing the CRUD operation, and returning the appropriate API resource instance. As an extra, I implemented pagination when returning collections of resources for smaller payloads and better UX on the frontend. I used, however, the Scribe third-party package for generating an OpenAPI schema and hosted API documentation. The package auto discovers routes and controller methods and tries to deduce the parameters, bodies, and responses, but it can have errors or shortcomings. One way it can be helped is by giving the controller methods annotations or doc comments. I chose the latter and provided

additional descriptions for the schema generation. It is also possible to document the routes and controllers provided by other packages, which in my case was Laravel Fortify. This can be achieved by writing a custom YAML (YAML Ain't Markup Language) file describing those endpoints.

I also modified the configuration of the package so that the application publishes the generated schema on the `/docs` endpoint, which can be seen in Figure 7.

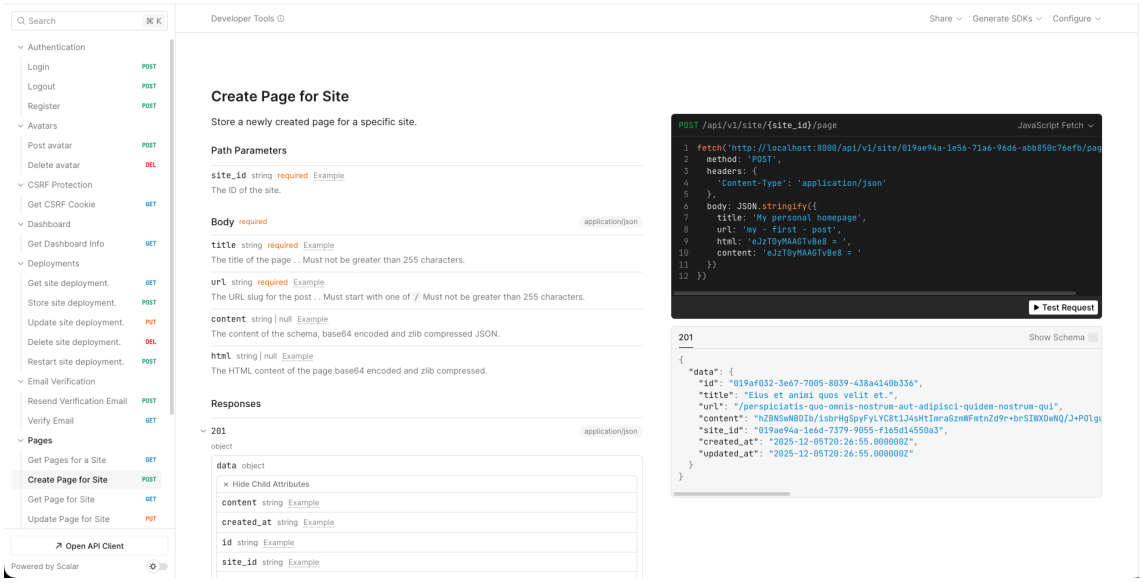


Figure 7: OpenAPI docs powered by Scribe

4.6 Deployment Logic

The deployment logic technically first begins at the page controller. When a page gets updated, the frontend sends both the content and generated static HTML's body in an encoded manner for more efficient network transfer. The HTML body then gets base64 decoded and zlib decompressed on the backend side to recover the raw HTML. Following this, the HTML gets sanitized to remove any unknown tags and malicious scripts, and to ensure no security concerns got added by any third-party. I implemented this by writing an HTML sanitizer service. This service uses Symfony's `HTMLSanitizer` class that I preconfigured with the allowed and rejected elements and attributes. Then, I registered this service as a singleton in the application provider, so that Laravel can automatically discover it and inject it into the Page controller constructor.

As a part of the sanitization process if it is successful, the service afterwards injects the body into a Blade template. This template provides the root HTML and head tags and a link to the stylesheet, which will be later injected by the proxy application. In the end, the complete HTML content gets returned and saved to the S3 storage and, of course, an API resource object gets sent to the frontend.

The MinIO S3 object storage showing the stored HTML files can be seen in Figure 8.

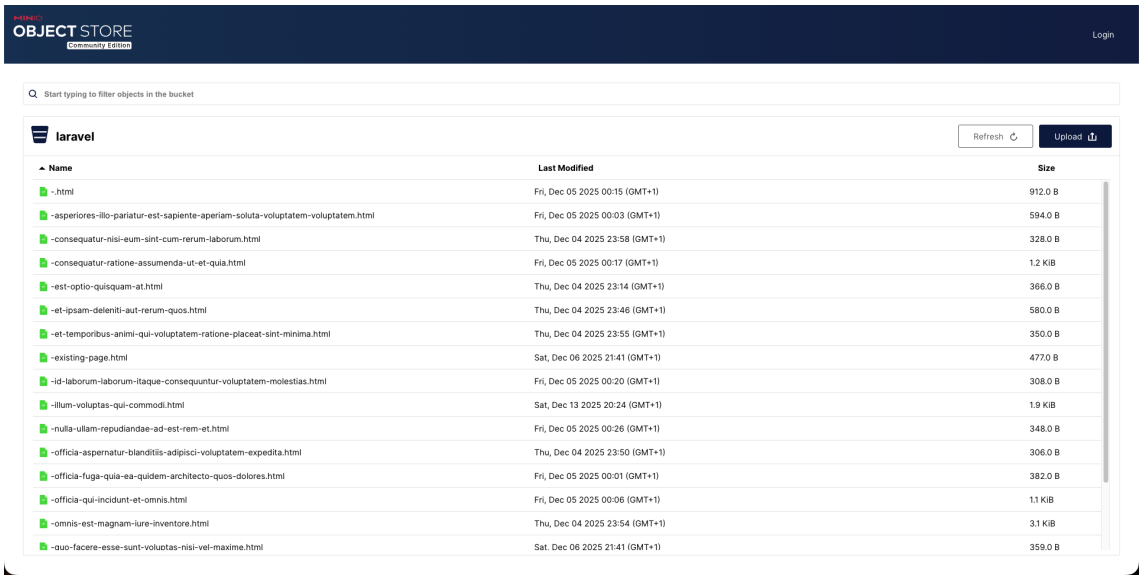


Figure 8: MinIO object storage

A sequence diagram displays this process to the fullest in Figure 9.

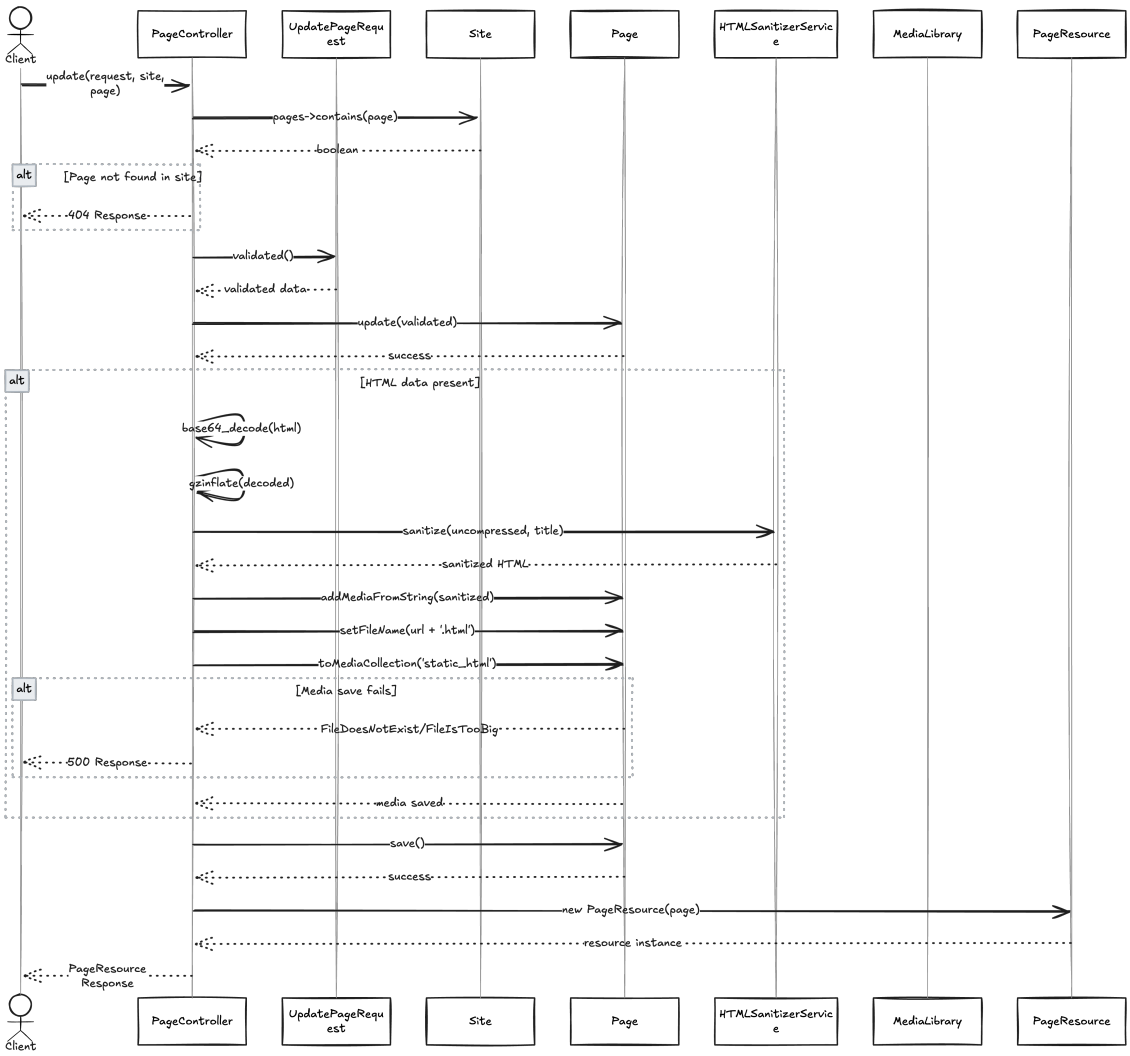


Figure 9: Sequence of saving the page content

Next comes the Deployment controller, which handles the actual site publishing related methods, which includes a full CRUD and the option to restart the site if it is needed. I have written and registered another singleton service handling deployments that the controller forwards the incoming request to. The deployment service then checks if the site can be transferred to the published state, builds the path and body of the request, which afterwards gets deferred to the proxy application. When the proxy returns with a response, the service forwards it to the controller, which finally forwards it to the frontend.

Another sequence diagram in Figure 10 shows this process.

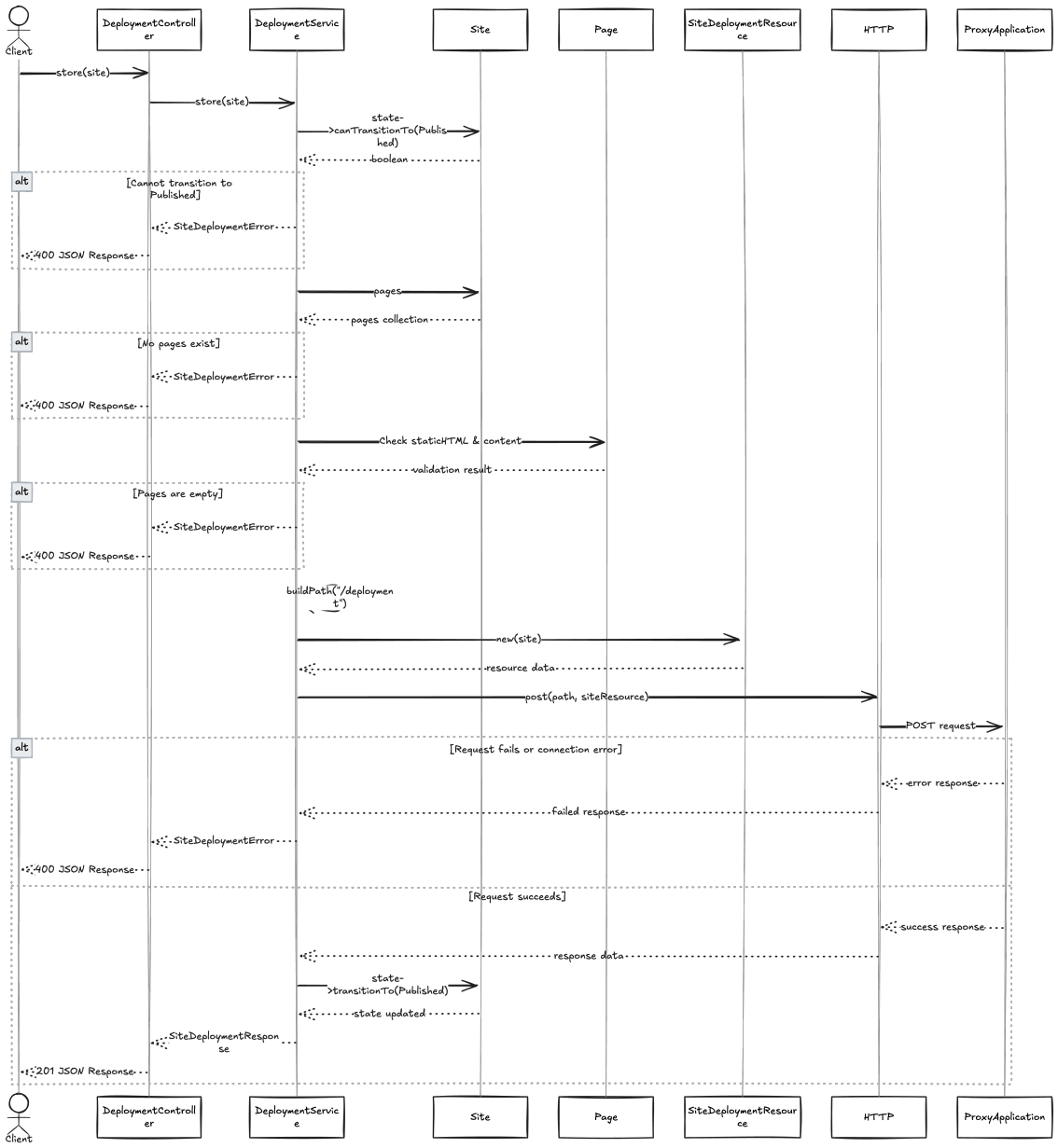


Figure 10: Sequence diagram of the deployment process

4.6.1 Admin Panel

The final feature of the backend I want to talk about in this chapter is the admin panel. It is provided by the Filament PHP package, which specializes in building primarily admin panels for Laravel powered applications, but can also be used as a standalone server-driven UI framework if required.

For the first step, the user model has to implement the `FilamentUser` interface. This interface requires the implementation of the `canAccessPanel` method, which has to return a boolean based on whether the user has access to the panel. Here is where I use the previously mentioned panel access permission. Afterwards, using the included artisan command by Filament, the admin panel needs to be initialized. If done, then it can be visited on the `/admin` route.

The admin site is empty at first and has to be populated. Filament defines resources, which encompass the forms, pages, and schemas related to each Laravel model class. Here, UI elements Filament provides can be used to build interactive data tables, forms, and views. The package provides filtering and sorting options by default, easily configurable searching, and relationships can be displayed through more descriptive fields as well. For example, instead of displaying the user's ID, their name can be retrieved for a better UX. Further customization is possible, but the defaults cover most use cases.

In Figure 11, a data table view of the sites can be seen.

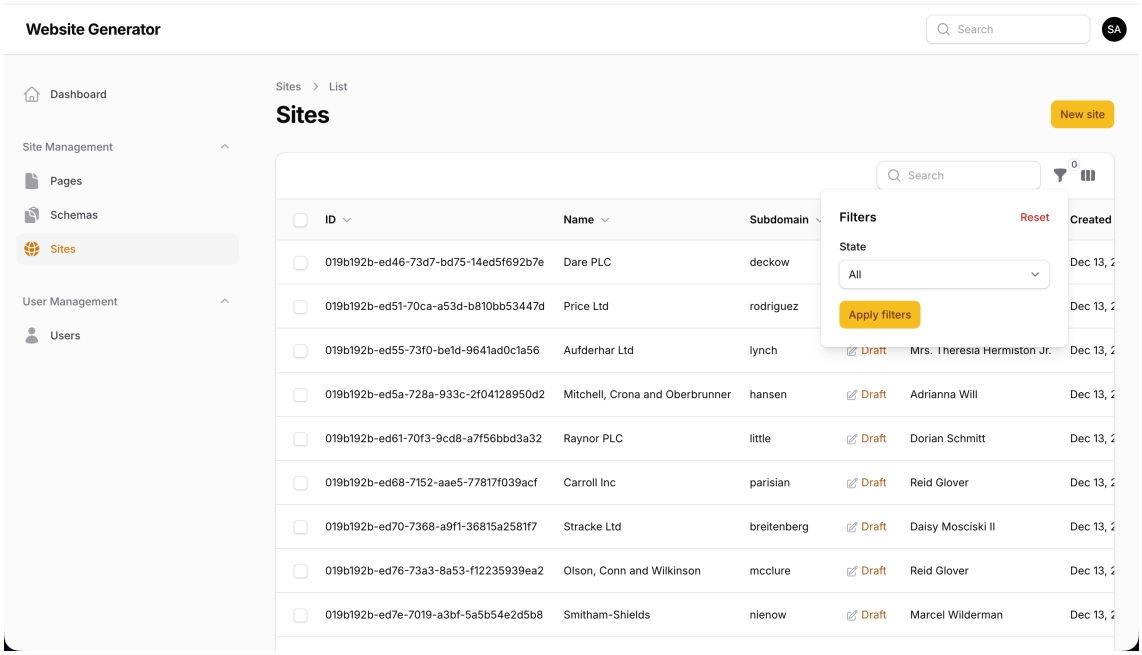


Figure 11: List view of sites on the admin panel

I extended the functionality of the admin panel with the Filament StateFusion plugin created by Assem Alwaseai, and the Spatie Media Library plugin maintained by the Filament team themselves. The former plugin integrates the Spatie Model States package into the admin panel and provides quick actions to transfer models from one state to another. It also checks for allowed states. The latter plugin provides a droppable file upload field for storing files in the media library's collections, and also handles displaying them. This is mainly used to handle the user avatars from the panel.

Furthermore, I implemented a Filament relation manager for sites and pages so that the pages of the sites could be controlled from the site view. Lastly, I created a custom action group to handle deployment related operations from the admin panel. These directly call the deployment service from the admin panel.

An overview of the previous two functionalities of a site's detail view can be seen in Figure 12.

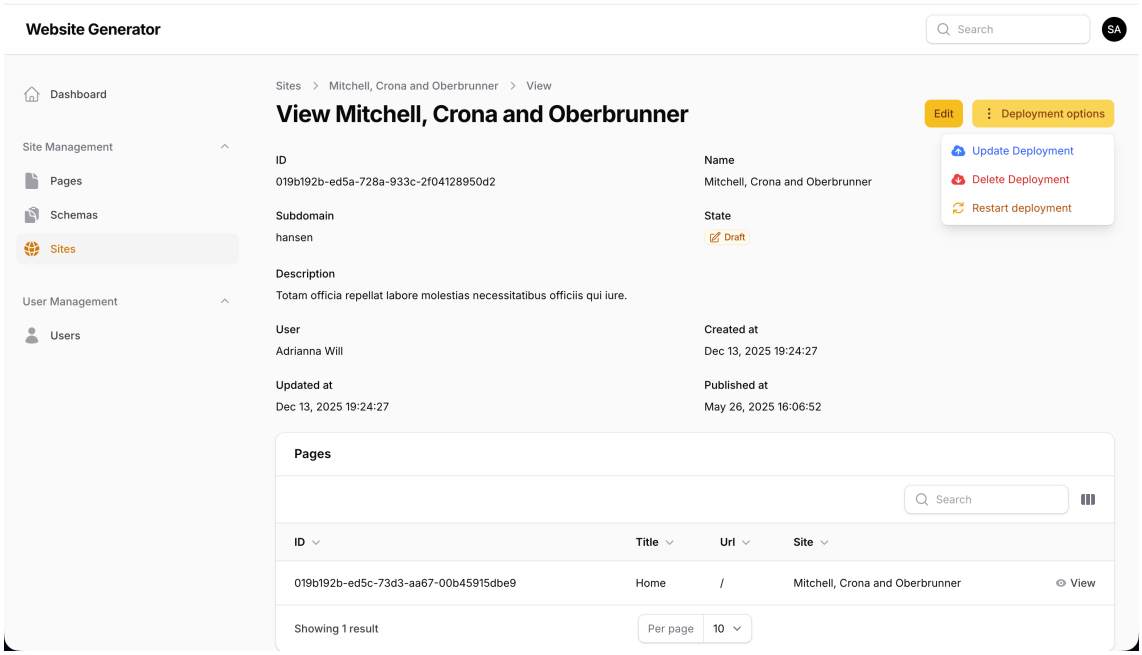


Figure 12: Site detail view on the admin panel

Chapter 5

Frontend

5.1 Setup and overview

I implemented the frontend using React first and foremost. The most important packages I used are TanStack Router for client side routing, TanStack Query for robust data fetching and caching, and CraftJS for the editor capabilities.

As the first step, the React app had to be installed. I scaffolded the application using the Vite bundling tool's CLI and chose a TypeScript based application. Then I set TypeScript to use strict type checking to disallow any types, and also installed TailwindCSS and its Vite plugin as I used the ShadCN UI components. ShadCN distributes its components and blocks in an open code manner through the npx CLI, which means that instead of installing an npm package, the source code of the block gets copied into the application. This has the advantage that the components can be completely customized if needed, and the downside that the repository size grows considerably.

I also used predesigned UI blocks form ShadCN.IO [33]. While I was building the application, all components from this site were open-sourced and free, however, as of writing this documentation, they introduced many new blocks that got locked behind a subscription. Previously free blocks remain as such, but require a free account now. This change doesn't affect the application as the source code got inserted into the project directly, so further use and modification is possible.

Next, I used Docker's init command to scaffold a Dockerfile and `docker-compose.yaml` for containerization, to which I only made minor changes, mostly regarding setting up environment variables. I also configured ESLint, which is a static code analyser, to ensure better code quality.

Since unlike Laravel, React is unopinionated about the structure of the application, let me talk about how I architected the frontend's folder structure. I was partly inspired by the Bulletproof React repository [34], however, I made a few changes to better fit my needs. The high-level overview of the layout can be seen in Figure 13.

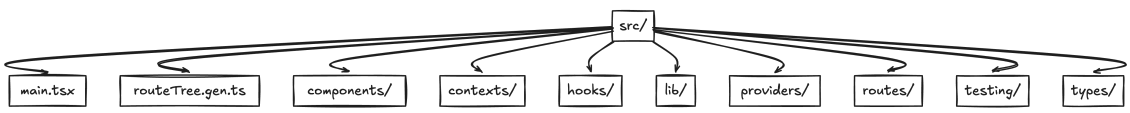


Figure 13: React application structure

The `main.tsx` is the setup file for the frontend application, about which I will go into more detail shortly. The `routeTree.gen.ts` file contains the routing tree that TanStack router generates. For more information about this, refer to Subsection 2.3.1. The `components` folder contains the UI elements added by the ShadCN library, and other component blocks I created using them that are global to the application. The `context` folder includes the globally defined application contexts, that I will talk about in Section 5.2. Inside `hooks` live the custom React hooks that I defined. The `lib` folder collects utility functions, global constants and the generated API client, which I will elaborate in Section 5.2. The `providers` folder houses the context providers that the aforementioned context objects return. The `routes` folder encompasses not only the route hierarchy but using the router's - notation it also colocates any route specific components and logic. The `testing` folder accommodates all test related files. Refer to Section 7.2 for a thorough explanation. Finally, `types` includes all TypeScript types that are either generated by the API client or Zod schemas.

Zod is a library used for schema validation primarily focusing on TypeScript. I used it in conjunction with React Hook Form, a hook based form library for React, to generate form schemas and provide client side validation. I also validated search parameters using it in a few routes. A great feature of Zod is that TypeScript types can be inferred from form schemas for further use.

Now, let me talk about setting up the application in the `main.tsx` file. Some details will be expanded upon later but are understandable at this point as well.

First, a router instance has to be constructed, to which the route tree needs to be provided. Additionally, the router context has to be configured with initial values. This has to include everything that routes need to have access to before even rendering their component. In my case, this contained TanStack query's query client, which is constructed here, the auth context, and a default callback that returns the given route's title. The latter will be overridden in each route and will be used for breadcrumb generation. The auth context is undefined at this point and will be updated later. Here, I also set up a default view transition that provides a smoother animation between route switching and scroll restoration. Moreover, I configured the loader stale time to 0 (never stale), as the query client will handle that instead of the router (both share this concept). Finally, I changed the default preload method to 'intent', which has the effect of launching loader preloads after 50 milliseconds when a user hovers on the link of the destination route.

Next, a component I called `InnerApp` has to be created. This component's only purpose is to have access to the useAuth hook and return the router provider with the router instance and a now existing valid authentication context. The router provider will take care of rendering child routes.

Moving on, I created an `App` component which wraps the previous `InnerApp` with the auth provider, the theme provider, and query client provider. The query client provider has access to the same query client as the router. Also, I included the `Toaster` component from the Sonner package created by Emil Kowalski alongside the `InnerApp`. This package is an easy-to-use and opinionated way of making toast notifications. I primarily use it to provide feedback to the user from background tasks such as API calls.

Last but not least, the `App` itself is surrounded with React's strict mode component that enables development-only extra behaviours with the aim of finding common bugs. In production, it has no effect and is turned off. At the end of all this, the component chain gets rendered by the root React DOM object, which finishes the setup process, as the `index.html` by default includes the `main.tsx` script.

One last thing I want to talk about here is the root route. It must be named as `__root.tsx` and is the root of the route tree. Here, the library's Outlet component has to be returned, which will take care of rendering all child routes. Additionally, I implemented application wide loading, not found, and error boundary components to provide a better UX for these scenarios.

For example Figure 14 shows the not found component that renders when a user navigates to a non-existent route.

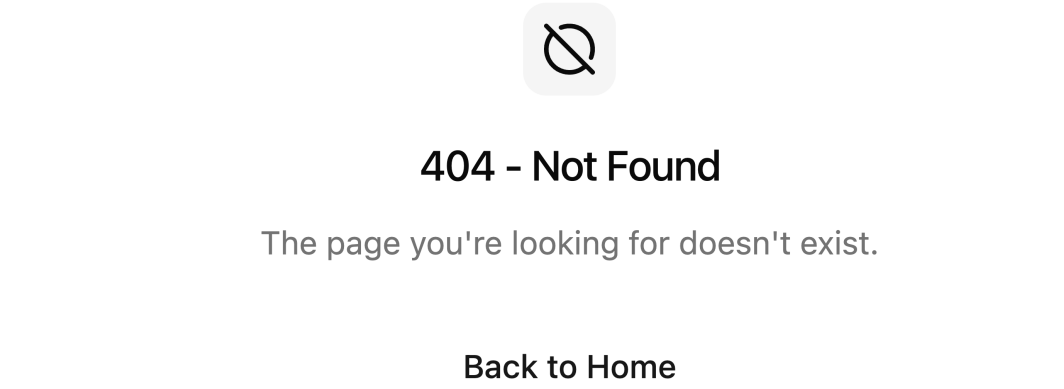


Figure 14: Not found component for missing routes

5.2 API client and context providers

After the setup process, I will move on to describing the features mainly in the `context`, `providers`, `hooks`, and `lib` folders.

Generating the OpenAPI schema of the backend has the great benefit of allowing me to take advantage of the many available libraries specializing in API client code generation. I investigated a couple of existing solutions, namely Orval, Kubb, and OpenAPI-TS. Orval and Kubb both feature an extensible approach where you can pick what you want to use. However, they both proved to be way more complicated than what I needed, so I opted to use OpenAPI-TS. OpenAPI-TS provides many smaller libraries that provide client-side code generation from OpenAPI schemas. For my use case, OpenAPI-React-Query was the one I needed and ended up using. It is a tiny, around 1 kb wrapper around TanStack Query, and it incorporates two other libraries by the same author, OpenAPI-Fetch and OpenAPI-TypeScript. It validates the available API routes inside the query calls, checks parameters and bodies, and can be used to generate TypeScript types for the requests and responses.

Next, I generated the API client with the library in the `lib/api` folder and also customized the fetch client used by it. Namely, I created a custom `APIError` class that handles error messages from the backend, configured the base URL from environment variables and the default headers. Additionally, since the backend uses CORS and CSRF protection, it requires requests to have a valid XRSF (Cross-Site Request Forgery) token. This can

be obtained by first making a request to the `/sanctum/csrf-cookie` endpoint. This was also configured in the fetch client to be handled automatically and re-fetch when the token expires.

Moving on, I implemented the custom theme and authentication contexts. Developers use React context to avoid the so-called prop drilling. Prop drilling is passing a component's prop (data) down multiple components in the component tree just to be used at the innermost child. This leads to many scalability and maintainability issues and is considered a code smell. Context solves this by making the prop available for direct use anywhere in the child component tree. There are also libraries such as Redux and Zustand that provide global state management, but I opted not to use these, as they themselves recommend against it if the application has few contexts. In some places, I also use prop drilling instead of defining a context, but it is never deeper than three layers and the intermediate layer also uses the passed prop in these cases.

For the theme provider, I was guided by the ShadCN documentation [35] on how to configure light, dark, and system themes using TailwindCSS's functionality. The provider stores the selected theme in and recovers it from the browser's local storage on subsequent visits. Toggling the themes is possible from the account page, and the app defaults to the system theme. Accessing the theme in other components is possible through a custom `useTheme` hook.

Afterwards, I created the authentication provider for auth state management. It provides methods through the `useAuth` hook for logging in, registering, logging out, re-fetching user context, and deleting the account. All these methods use the API client wrapping TanStack Query's mutations and query functions. The context encompasses two React states: one boolean for quick checks, and the other containing the logged-in user's information. I also store the state in the browser's local storage and recover it upon mounting by a `useEffect` call so that the user remains logged in on revisits.

5.3 Authentication, dashboard, and account routes

In this block, I will mainly be presenting the authentication and account related pages that I created on the frontend. To begin with, a use case diagram in Figure 15 shows in detail the authentication capabilities.

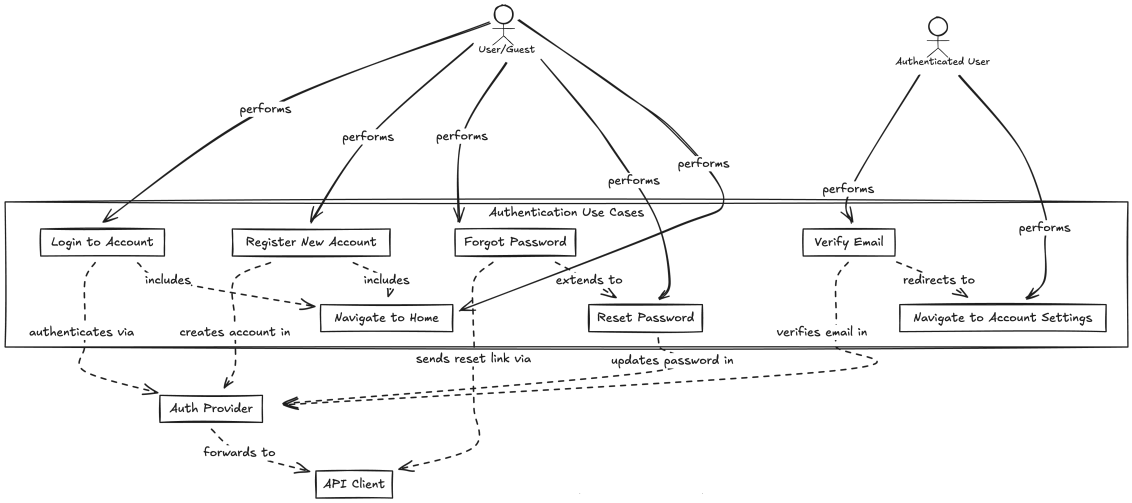


Figure 15: Authentication use cases

Guests or unauthenticated users can log in, register, and ask for a forgot password link, which redirects them to a form allowing them to reset their password. Authenticated users can verify their emails, which they automatically get an email for after registration, or can request a new one on the account page, which I will talk about later. Obviously, if the user clicks on the verification link in their email while being unauthenticated they will first be redirected to the login page.

When a user first visits the frontend, a simple index page greets them with directions to login, or register, or go to the dashboard if they are already logged in. The index page as shown for an unauthenticated user can be seen in Figure 16.

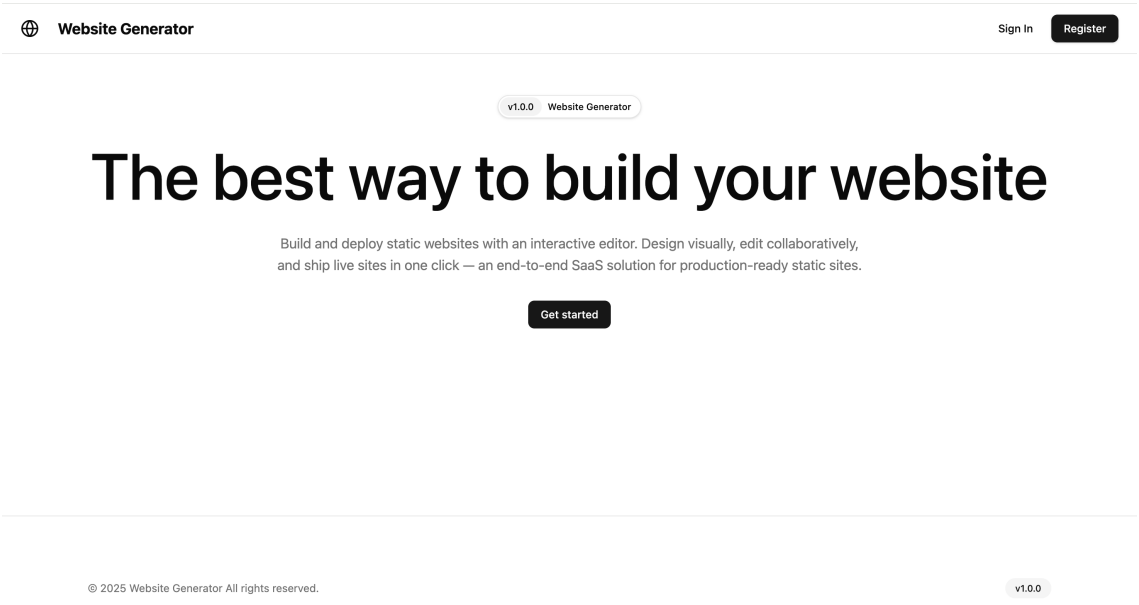


Figure 16: Index page for an unauthenticated user

From the landing page, the user can directly log in and/or register. As stated previously, I used React Hook Form with Zod for creating all my forms. This ensures that there is proper client side validation, and can be used to display meaningful error messages to the user. All forms also employ a loading state to prevent multiple submissions while a previous request is running. I also display success and error responses using the aforementioned toast notifications in the bottom right corner of the screen. The registration page with an example for validation errors can be seen in Figure 17.

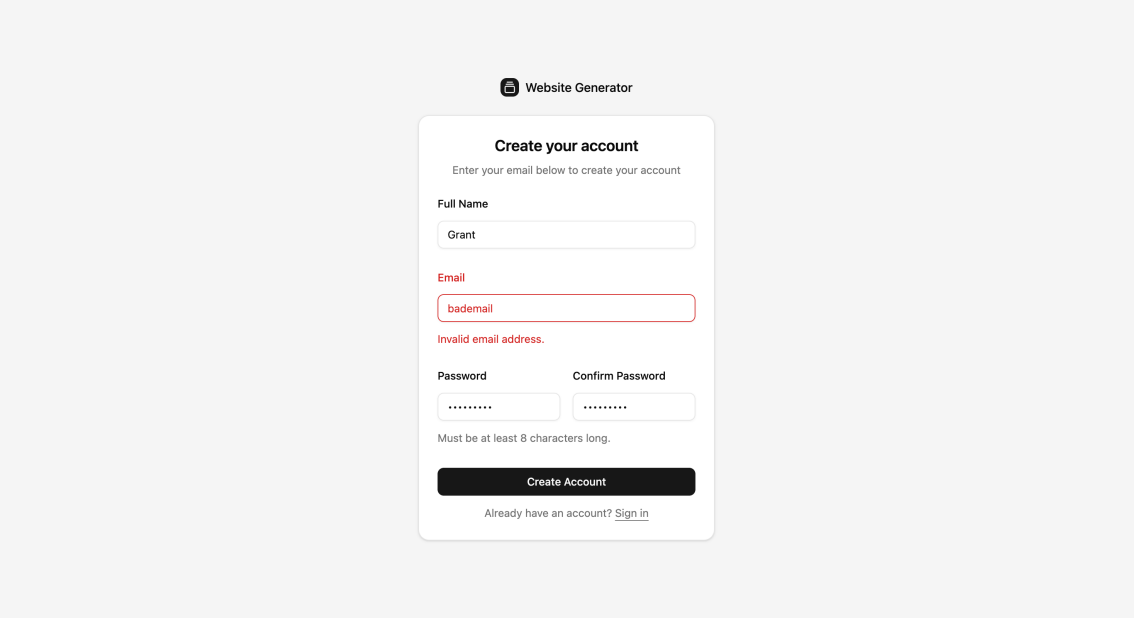


Figure 17: Register page with validation errors

The rest of the forms look almost identical to the registration one, so I won't show each of them one-by-one unnecessarily. One thing worth mentioning is that both the registration and forgot password links send out emails. Such an example email after registration captured by Mailpit can be seen in Figure 18. On the sidebar the incoming password reset links can be seen as well.

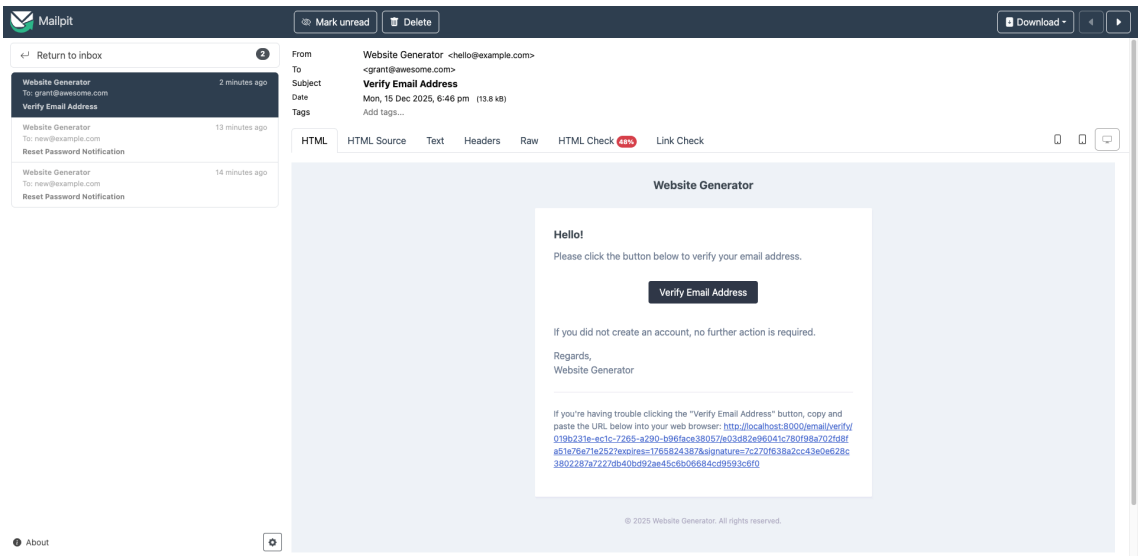


Figure 18: Registration email seen in Mailpit

After logging in, the user can access the main application routes grouped by the _app folder. They are all surrounded with a layout component that has two main responsibilities. First, check if the user has successfully logged in, if not, redirect them to the login route. Secondly, this is where the title callback I mentioned in Section 5.1 is used. The layout component collects the result of all title callbacks from the router state, which include the title names for all rendered child routes. Afterwards, it dynamically builds the breadcrumb trail displayed to the user.

The first such route is the dashboard, which displays a few statistics and quick actions. Figure 19 showcases it in detail.

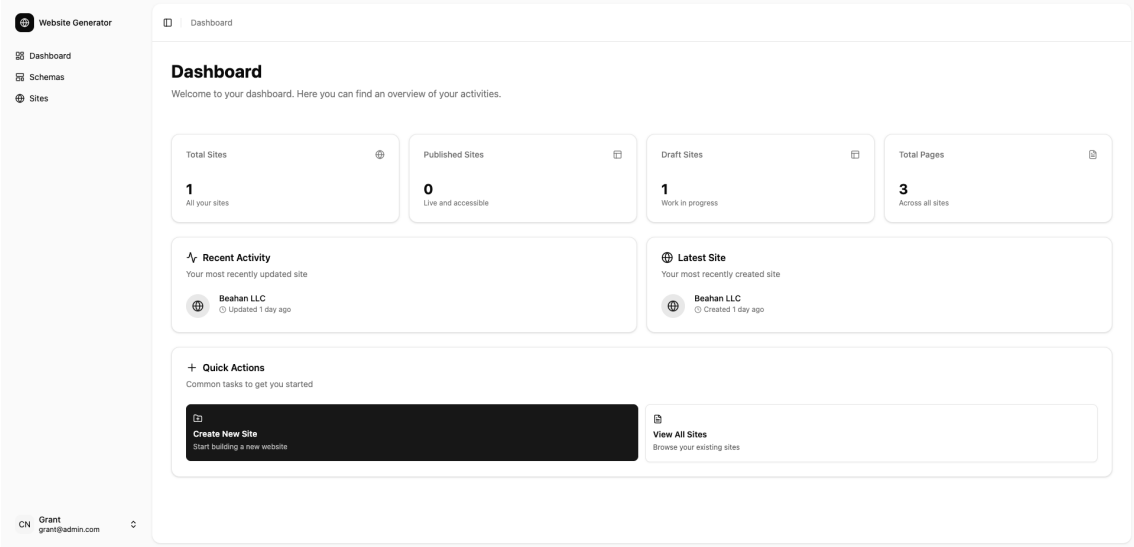


Figure 19: The dashboard page

Next, let me provide and overview of account settings, which can be accessed via a pop-over by clicking on the user section of the sidebar. Another use case diagram showcasing the account-related options can be seen in Figure 20.

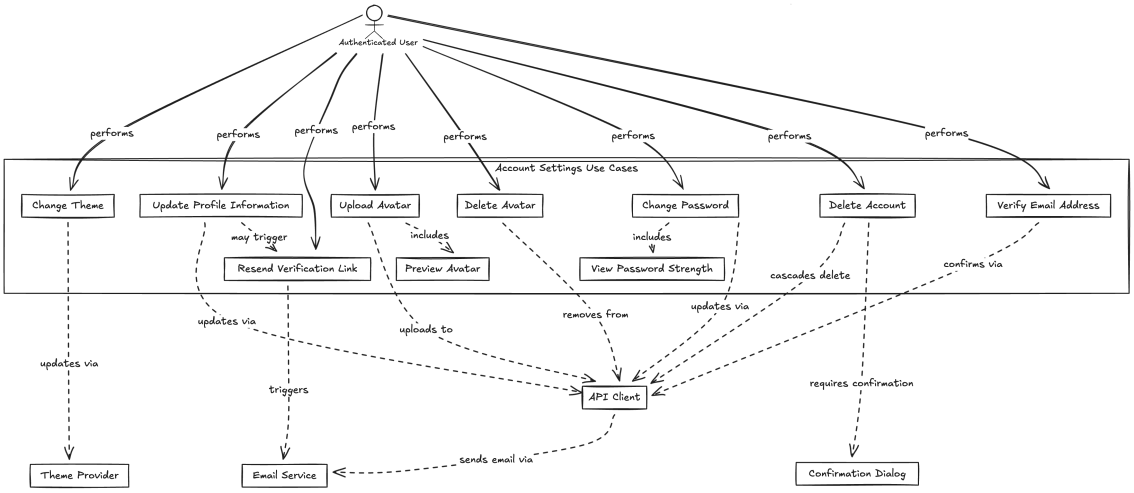
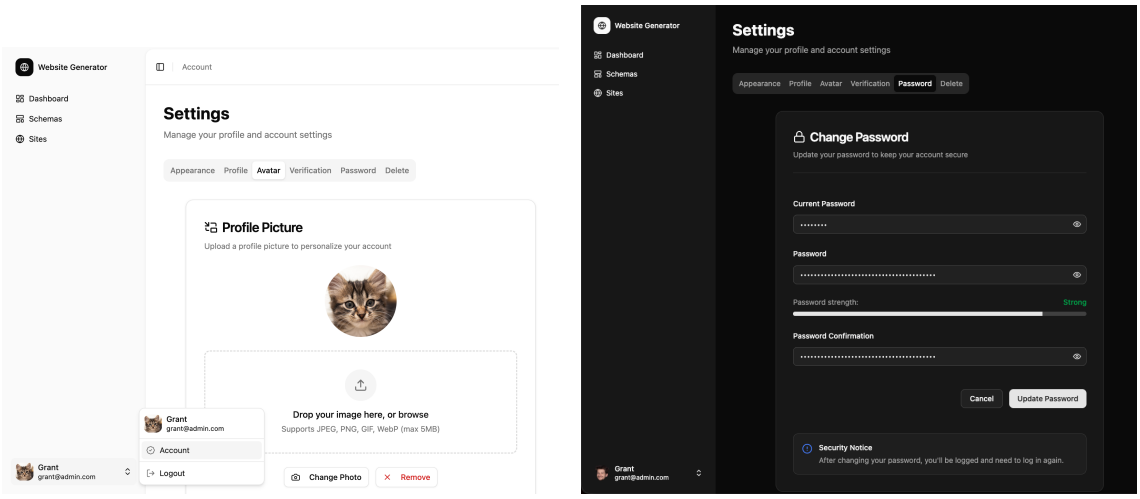


Figure 20: Account settings use cases

The user is able to switch the application between light, dark, and system themes. They can also update their profile information, which includes changing their display name and email. Changing the email automatically sends a verification email to the new address. They can upload an avatar via a drag and drop field or the system file picker, which can be previewed before saving. A tab, which shows the verification status is available, and if unverified, a new link can be sent from here. Passwords can be changed using a form that shows the new password's strength using simple metrics like length and contained special characters. After changing it, the user will be logged out automatically and prompted to

log in again. Lastly, the account can be deleted if the user wishes so. This will first show a confirmation dialog to avoid accidental deletion.

Two pictures comparing light and dark modes can be seen in Figure 21. The light mode picture on Subfigure 21a showcases the avatar upload tab, while the dark mode picture on Subfigure 21b presents the password change form. The light mode picture also displays how to get to the account settings page.



(a) Light mode avatar upload **(b)** Dark mode password change

Figure 21: Dark and light modes

5.4 Schemas, sites, and pages

Next up, let me talk about creating and managing schemas. As with the previous features, I present a use-case diagram of them in Figure 22.

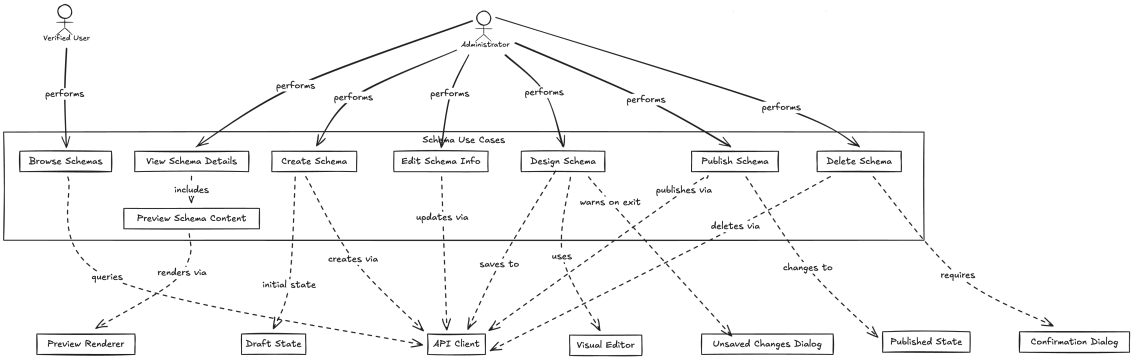


Figure 22: Schema use cases

Schemas can be used as templates for pages, which I will mention the details of later. Most of the use cases include basic CRUD operations. Users, however, have to have a verified email to access schemas (sites and pages too), and also they can only view them. They also only see published schemas. Only administrators are capable of editing, designing, and publishing schemas for users to reuse. I used similar page designs for related features across schemas, pages, and sites, so I won't show each page related to the schemas here. In Figure 23 a list view of schemas can be seen.

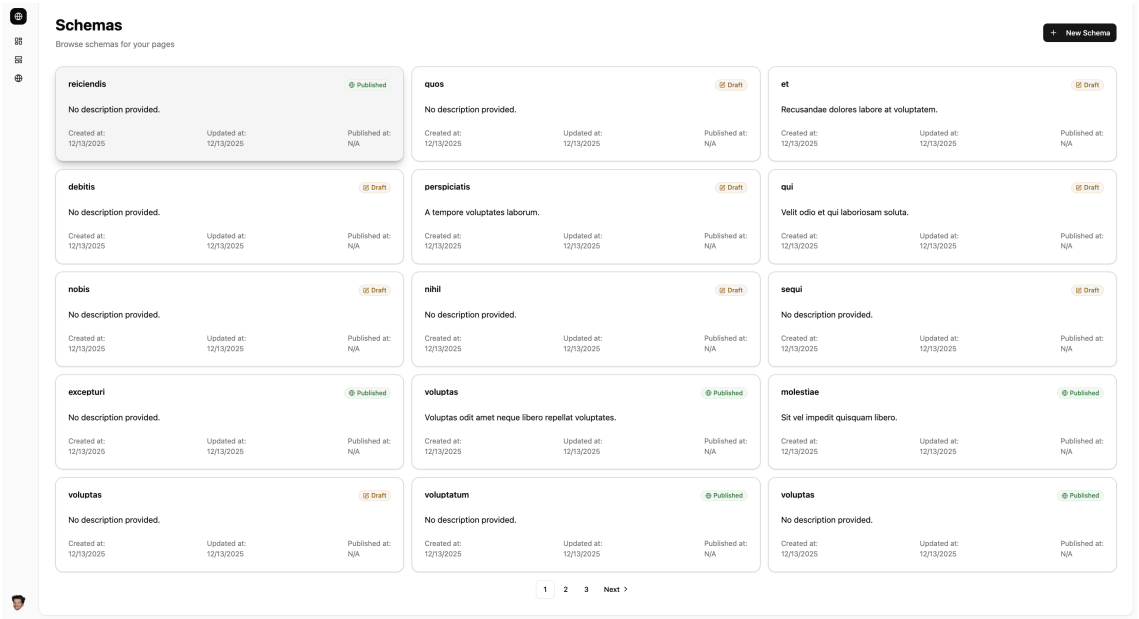


Figure 23: Card list of schemas

At the bottom of the screen, the pagination component is visible, and the hovered first card has a shadow and shading applied. The detailed view of schemas is almost identical to the pages. For an idea on how that looks, see Figure 30. The schema edit and create page is basically the same, and it is a simple form that can be viewed in Figure 24.

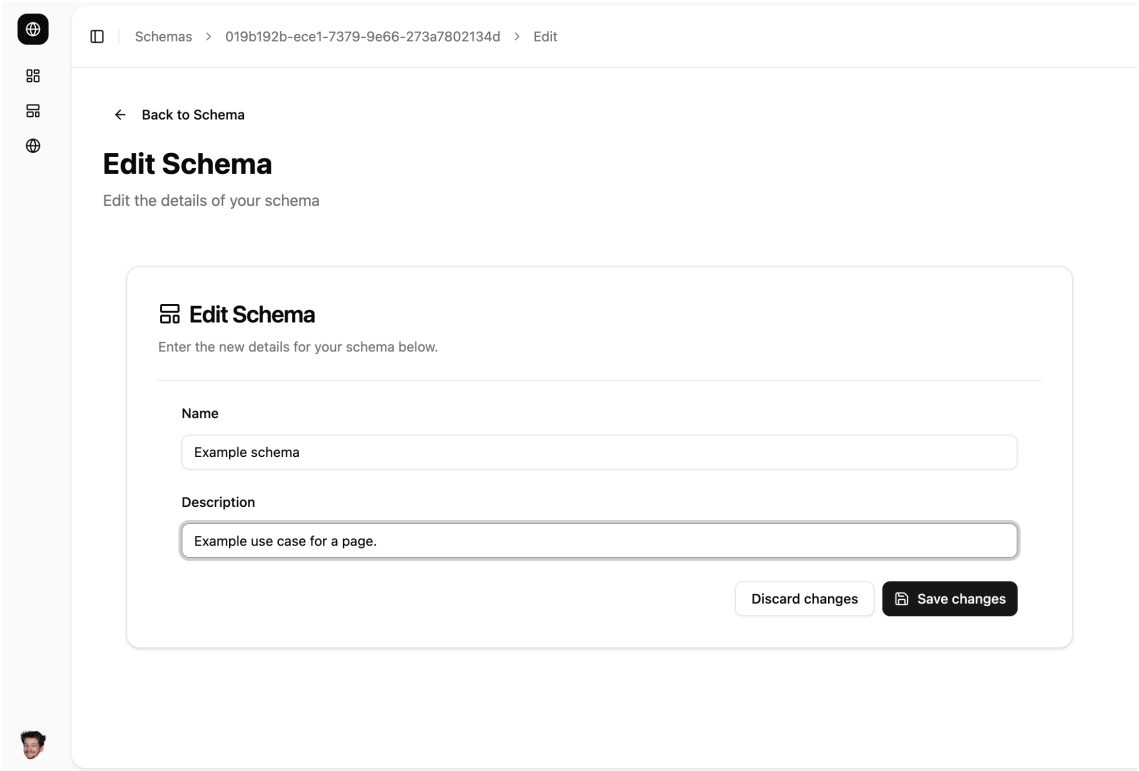


Figure 24: Schema edit form

The designer for schemas is the same as the one for the pages and is the main topic of Section 5.5.

The next resource on the frontend is the sites. Once again, a use case diagram of site related features is visible in Figure 25.

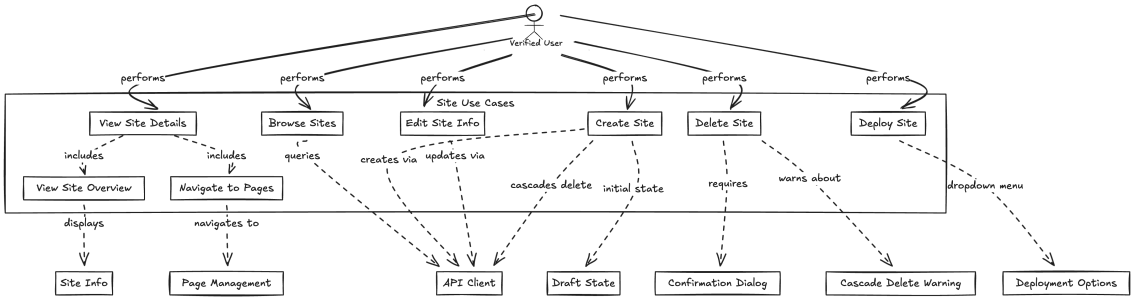
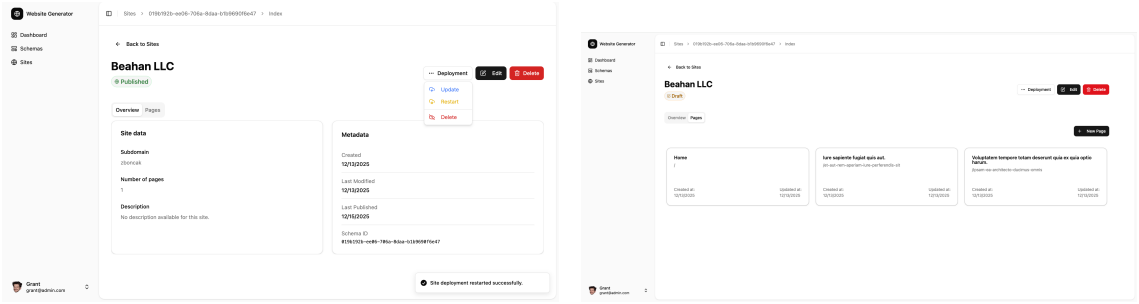


Figure 25: Site use cases

Sites encompass their pages. They have a two tabbed detail view, one showing general information, while the other shows their pages. It also contains controls for deploying the application if it has not been published yet. If it has, it is possible to restart, update and delete it from the dropdown. A confirmation dialog appears in all mentioned cases. The detail view of the site appears in Figure 26. Subfigure 26a shows the general information, deployment controls, buttons that take the user to the edit form and designer pages, and a delete button for deleting the site and all its resources. A toast notification indicating a successful restart is also visible. On the other hand, Subfigure 26b shows the pages of the site where their detail view can be visited by clicking on a card, or a new one can be created with the provided button.



(a) Overview tab **(b)** Pages tab

Figure 26: Site detail view

Regarding the deployment options, this and the schema publish is the only scenario in the frontend where I have to use manual query invalidation to update the displayed data in time. This is because we stay on the same view, instead of navigating away, which would normally trigger SWR's re-fetch mechanism. All three resources I talk about in this section also have an empty component that is displayed if the returned API answer is an empty collection. Figure 27 displays the one belonging to sites.

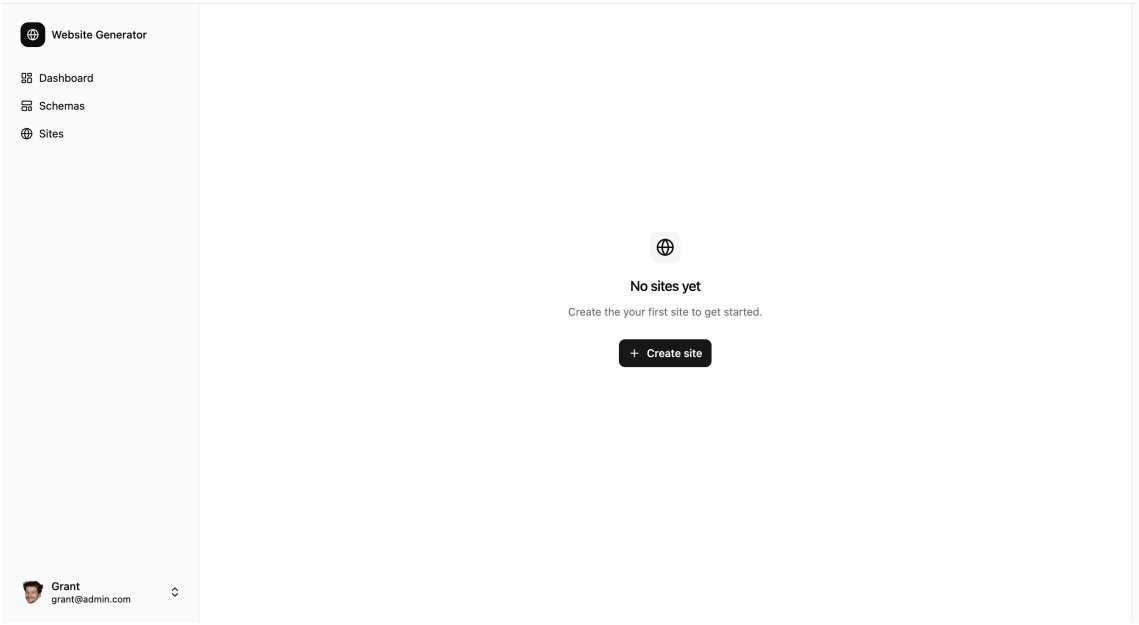


Figure 27: Site's empty component

Finally, let me talk about the pages. As with the previous two cases, Figure 28 shows a use case diagram for existing page related actions.

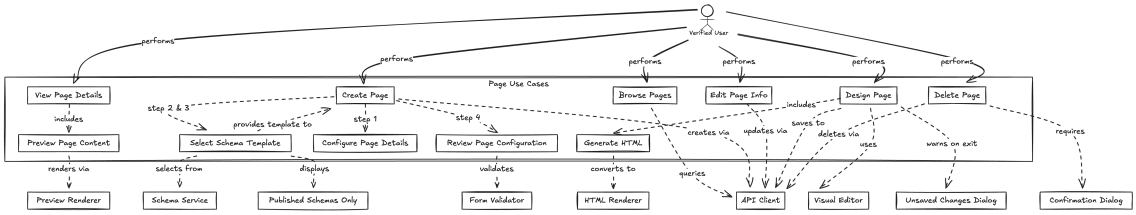


Figure 28: Page use cases

The editor related actions, such as generating the HTML content, play core parts in saving the page to the backend. Most other functions are identical to schemas and sites. A crucial difference is the creation form. Since pages can be based off of schemas, the creation form needs to be able to provide the ability to select a schema as the base content for a page. Later, this of course can be edited by the user without it having any effect on the original schema. The form basically copies its content into the page. To solve this problem, I created a multistep form using the Stepperize React library, which has built-in support for the ShadCN UI components [36]. Figure 29 showcases this form and the schema selection step. Schemas can be previewed here via a dialog.

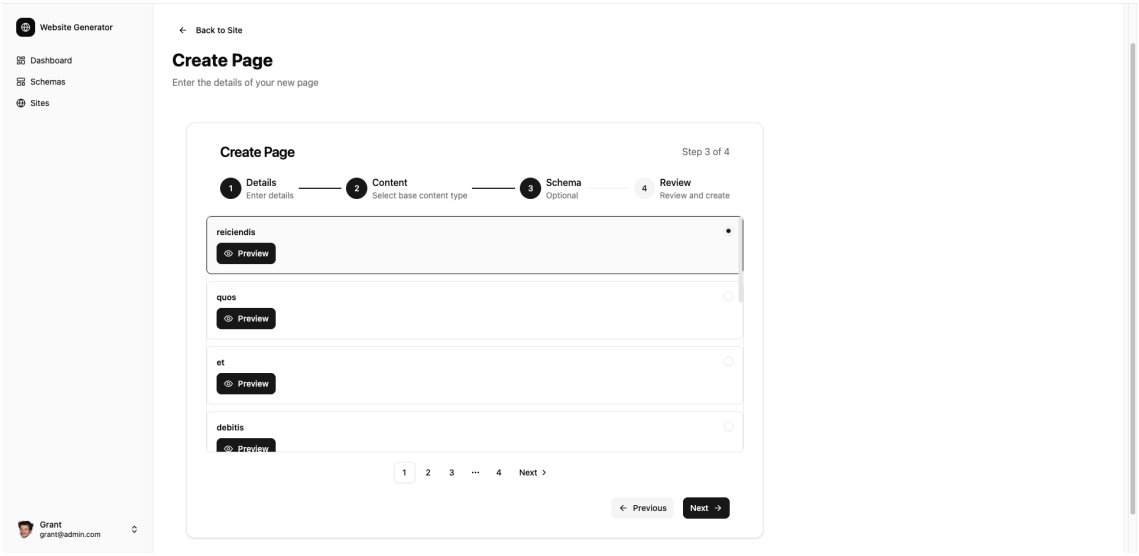


Figure 29: Page creation multistep form

At the end of the form I show the user a review tab, and the process can also be reset here if needed. The page edit form is similar to the schema edit form in Figure 24. The detail view of the pages also contains a preview of the content for easier identification. Figure 30 presents this functionality.

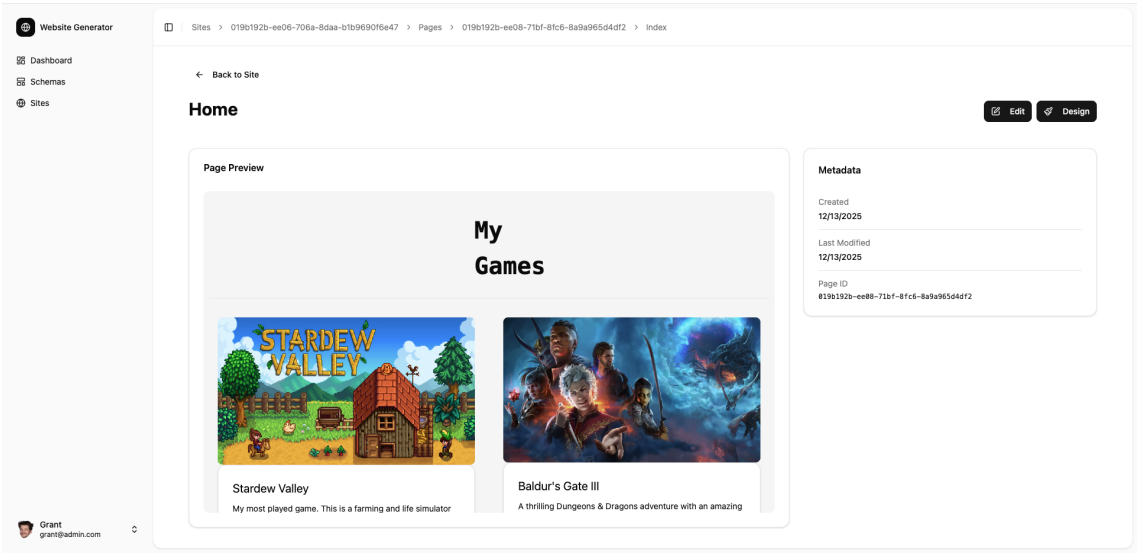


Figure 30: Page details with content preview

5.5 Editor

The editor or designer is undeniably the most complex and central component of the frontend. It uses the CraftJS library for its core functionalities. I reused this component in both schemas and pages, by passing a callback from the parent component that handles the saving functionality, which is the only difference it this case between the two.

CraftJS allows the developer to define editable components. It leverages this by providing custom React hooks. These define the editable props, what options can be set for these

props, whether the component is editable or not, and what regions can the user drop specific components to inside itself.

The full use cases of the developed designer can be viewed in Figure 31.

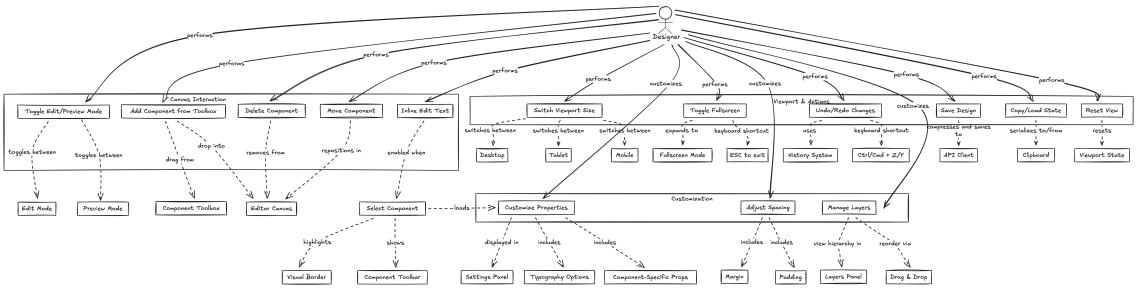


Figure 31: Use cases of the designer component

The editor with some example components inserted is displayed in Figure 32.

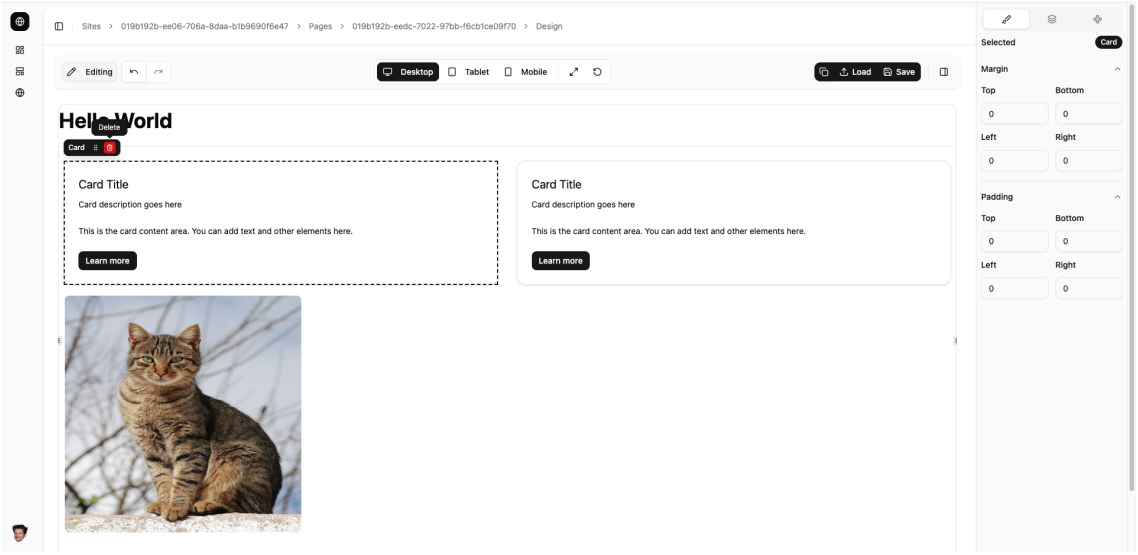


Figure 32: The editor filled with example content

I composed the designer of three main components. The first and most important is the editor area, which shows the design the user is creating. This is a resizeable canvas to which users can drag and drop the editor's components. On hovering over the components, an outline appears for the selection, showing the name of the component, a mover icon prompting that it can be moved, and a delete button to remove it from the page. When dragging components around, green and red lines show that the given component can or cannot be moved to the new desired location.

The second is the top bar, which contains the control buttons. On the left side is a toggle that enables or disables preview mode. The preview mode prevents any sort of edits and changes, and the sidebar of the editor, which is the third main component I will mention shortly, reflects this as well. Next to this is the buttons for undoing and redoing changes, for which I also implemented the usual keyboard shortcuts. CraftJS provides this functionality natively by itself, through a hook.

The middle section of the top bar has viewport controls that resize the editable area to see how the page would look like on desktop (default), tablet and mobile sized screens. There

is also a button to view the content in full screen, from which exiting is also possible by pressing the Escape key. Lastly, in this area I placed a reset button to exit full screen and or return the viewport to the desktop size.

Figure 33 showcases the designer in preview and full screen mode.

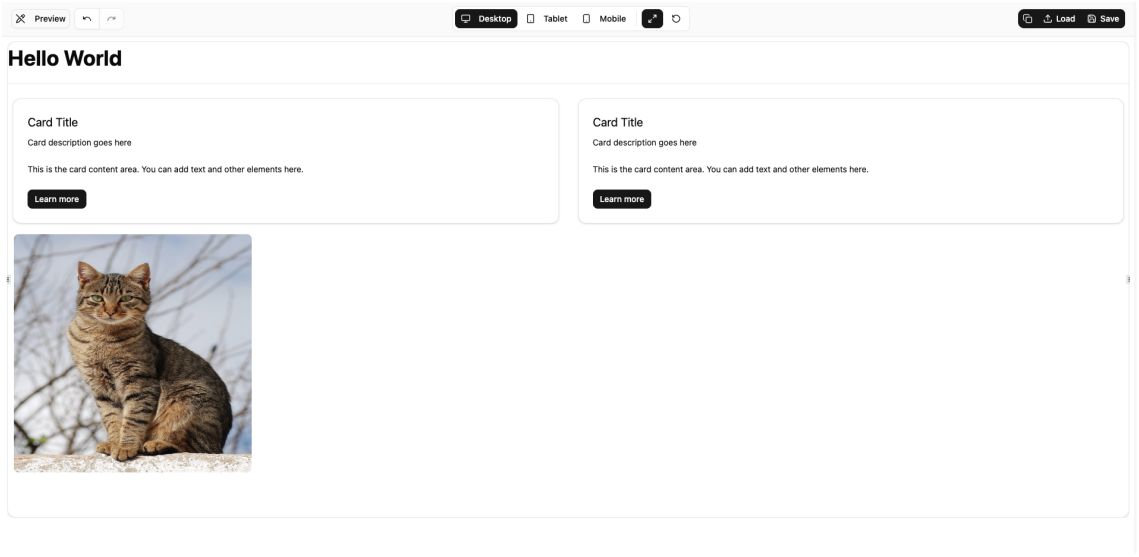


Figure 33: Editor full screen preview

The third section of the top bar houses a copy button for quick copying of the editor content, and a load button that allows rebuilding the page from the copied content through a dialog. Lastly, I located the save button for saving the changes to the database here. Related to saving, I also implemented navigation blocking for pages that use this component. This displays the built-in alert of the browser notifying the user that unsaved changes will be lost upon leaving the page.

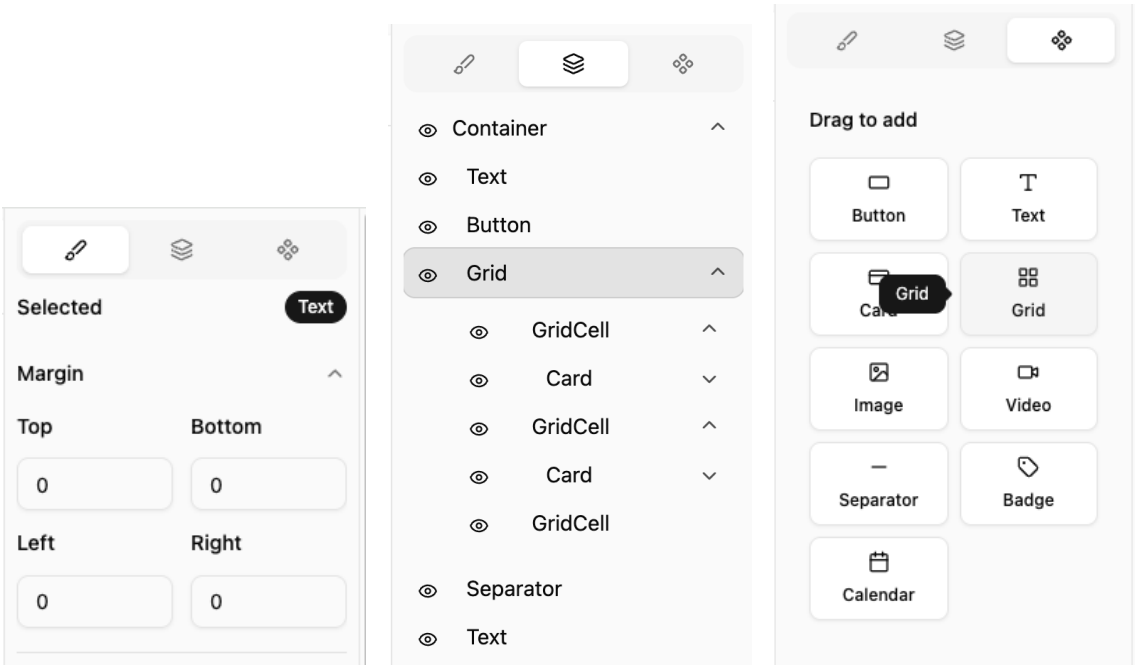
Before I describe the editor sidebar, I briefly want to talk about saving the content. CraftJS serializes the editor content into a JSON structure describing the component tree of the edited page. The root component is the canvas to which components can be dropped to. Each node of this tree also describes the props of the component and what values users passed to it. It is easy to see that this tree can grow large relatively quickly. Due to this, I used the Pako library providing a zlib port to JavaScript, to compress the JSON tree [37]. I then also base64 encoded this to further reduce its size for network transfer. The steps described above, however, only take care of the JSON, but the static HTML has to be somehow generated as well. To achieve this, I create an empty div to which I provide the editor in a disabled state with the filled content. Then, I call React's `flushSync` method on this, which returns the HTML that would be rendered by React. This is compressed the same way as the editor component tree and gets sent to the backend for processing.

The final part of the editor page is the aforementioned sidebar. It has three tabs, the first shows the common settings and specific settings of the selected component, organized by accordions. Common settings include setting the margin and padding of the element. The second displays a customized version CraftJS's layers companion library, which renders the component tree, allows reorganization and selection directly, and can show or hide specific components. This has a default styling by default, so I replaced that following the example provided by the author to use the ShadCN UI library to match my application.

The third tab shows the available components using small cards with icons, descriptions, and tooltips. From here, they can be dragged and dropped to the designer area.

I implemented nine components as a part of the thesis. One is a generic typography component for creating texts, which can use three different fonts (sans, serif, and mono), weights, italics, underlining and strike through, among many. There is a button component, which allows connecting the pages of a given site by using relative routes (e.g., `/about`) or by providing absolute links to external pages. The card component is similar to the card view of schemas and sites. Simple badge and separator components are also available. A calendar component can be used to display a specific date. There is an image component for external images, for which size and width can be either fixed or auto, or a combination of the two (e.g., fixed width and auto height). A video component is available for linking external videos, which also automatically converts YouTube links to their embedded versions. Finally, a grid component can be used for organization, which allows a row or column layout and setting the amount of cells. They can also be nested inside each other.

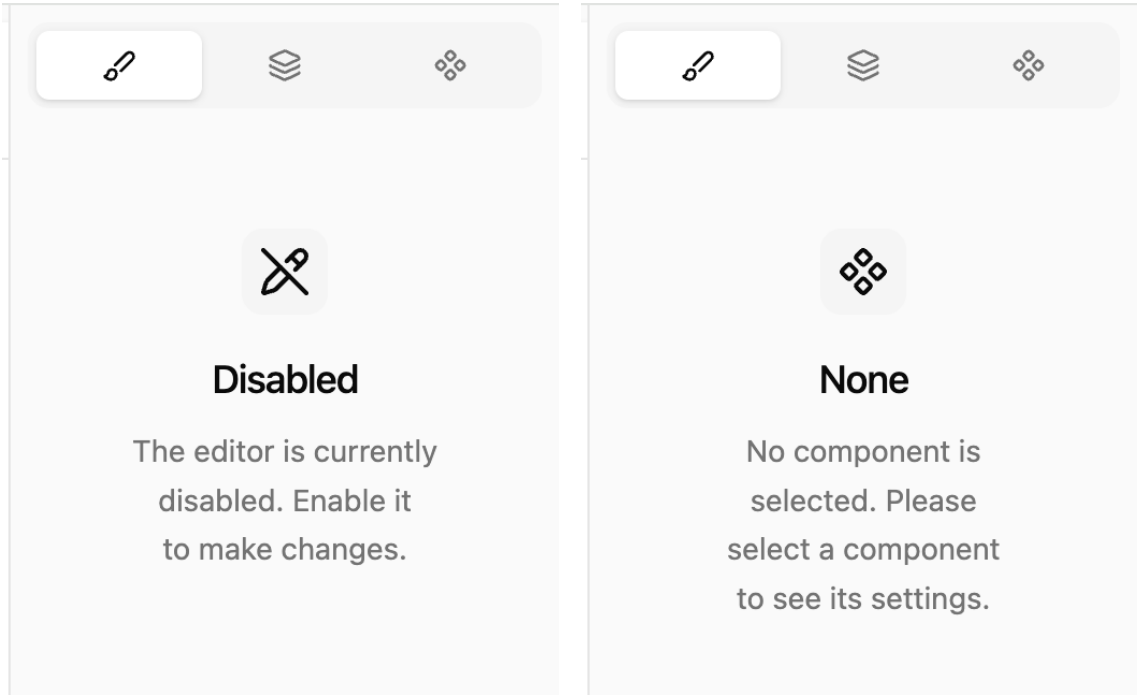
Figure 34 showcases these sidebar features in detail. Subfigure 34a showcases some customization options, Subfigure 34b presents the custom layers component, while Subfigure 34c displays the component toolbox.



(a) Customization tab **(b)** Layers tab **(c)** Components tab

Figure 34: Editor sidebar

To finish off, in Figure 35 the previously mentioned reaction to the disabled or unselected states can be viewed, in Subfigure 35a and Subfigure 35b respectively.



(a) Disabled hint **(b)** No selection hint

Figure 35: Sidebar reacting to different states

Chapter 6

Proxy application

The proxy application is a small NodeJS based application that forwards the backend's deployment request to the Kubernetes cluster. The first obvious question one may have is why this is a separate application instead of part of the backend. Unfortunately, there is no official PHP client library maintained by the Kubernetes team itself. There are many community packages, and most seemed promising, but regrettably all of them turned out to be outdated either from the perspective of the Kubernetes API versions or from major Laravel versions. Due to this reason, I decided on outsourcing this capability to a small proxy application, which is the application described in this chapter. This comes with a few benefits, since the deployment logic of the system is completely separate from the rest of the business logic. The proxy application, and by extension the orchestration of the platform could be scaled and updated independently of the other parts. Moreover, by abstracting away the implementation details of operation, the platform as a whole is more flexible. New ways of deployment could be added by extension or be replaced without friction and notice.

Before I begin with the development details of the app, I want to mention setting up the Kubernetes cluster itself for development. I used Docker desktop's built-in cluster while developing this project, but other distributions, such as minikube or k3d should suffice as well. Then, I installed Traefik as a Gateway API into the cluster, with the helm package manager for Kubernetes.

Now let me describe the implementation of the proxy application itself. I chose to use NodeJS due to it having an official client library and also due to the fact that experience with the language could be carried over from the frontend development. As with the other two applications, I set up a Docker init based Dockerfile and compose file for containerization purposes. I also configured TypeScript as the main language, set strict mode, provided an ESLint config, and created `.env` and `.env.example` files for environment variables. I decided to use Fastify as the server of this application as it provides request validation built-in, TypeScript support and is extendible over plugins. Furthermore, I decided to use the OpenAPI generator plugin to also provide hosted documentation. This is seen in Figure 36.

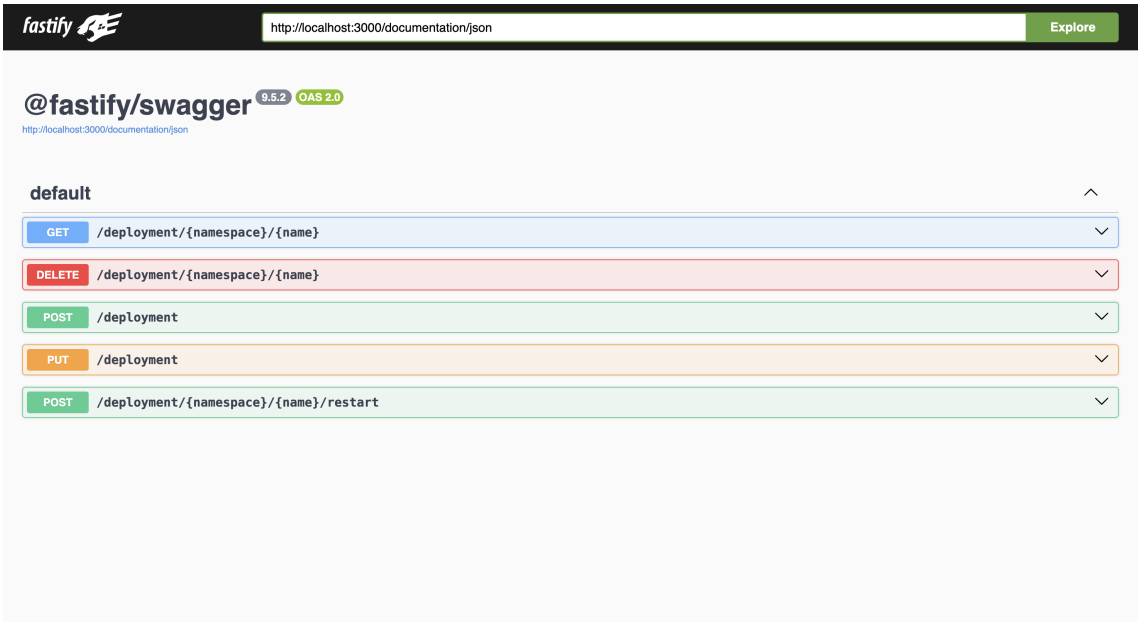


Figure 36: OpenAPI documentation of the proxy

Since Fastify is also unopinionated about the code structure, I decided to take inspiration from NestJS's module based structure, which provides a maintainable and extensible base for this proxy application. This structure can be seen in Figure 37.

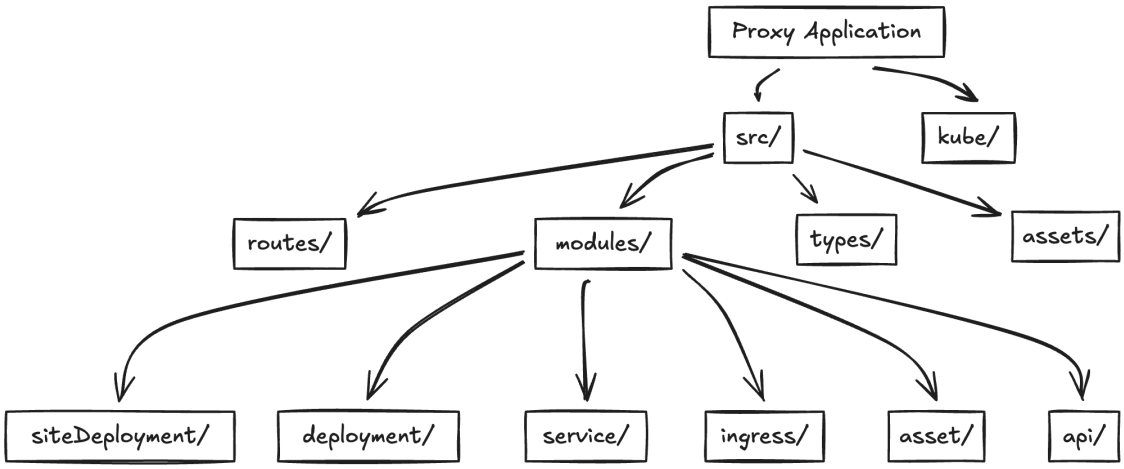


Figure 37: The structure of the proxy application

The `kube` folder contains an example configuration file for connecting to the cluster, which has to be copied and set accordingly. Inside `src`, the `routes` folder contains files, which list the routes of the application, in this case the CRUD and restart routes for deployments. The `assets` contains the `index.css` file, which provides the style of the websites, while `types` organizes global TypeScript types, as module specific ones live inside their respective folders. The `modules` folder contains colocated logic of each Kubernetes resource, which is mostly composed of requests, services, controllers, and types. Lastly, I set the server up in the `src/index.tsx` file where I registered the routes to the server instance.

The API module configures the Kubernetes client library and exports the core, apps, and objects APIs for other modules to use and create resources. The asset service has a method

that ensures that a config map containing the style CSS file is available within the cluster for mounting as a volume for deployments. It leverages this by querying for the object and if not found, assembles the specification and creates it through the library.

The ingress module, which in reality uses the GatewayAPI and HTTPRoutes, assembles the specification of HTTPRoutes for the Traefik Gateway. It specifies the namespace, label selector for the application, name of the route resource, and the host, which is composed of the domain of the SaaS and the subdomain given the website. All other Kubernetes resources get the same label selector defined to easily identify all cluster resources belonging to the same site.

The service module constructs the service for the website, which provides a single definition point.

The deployment module has the task of creating the deployment object for the website. This is composed of multiple parts. First, it mounts the aforementioned config map to access the style sheet. The HTML files already contain a link for this sheet. Then, it parses the incoming request. This contains the name of the site, which in this case is equal to the subdomain as it is a unique identifier for it, the array of its pages, and the namespace to deploy it to. The pages only encompass the route path and object storage URL of the HTML file. Afterwards, a one target replica and a revision history limit of 10 gets set. Next, I use an init container to fetch the HTML files from the provided URLs of the S3 storage and mount the result to the actual container inside the pods. The container itself is a simple NGINX web server serving the site and its pages statically. The pod also has limits set as 500 m for CPUs and 500 Mi for memory.

Finally, the `siteDeployment` module leverages the aforementioned modules to handle the full CRUD of the deployment strategy. The full sequence diagram for the creation of the website's deployment is in Figure 38.

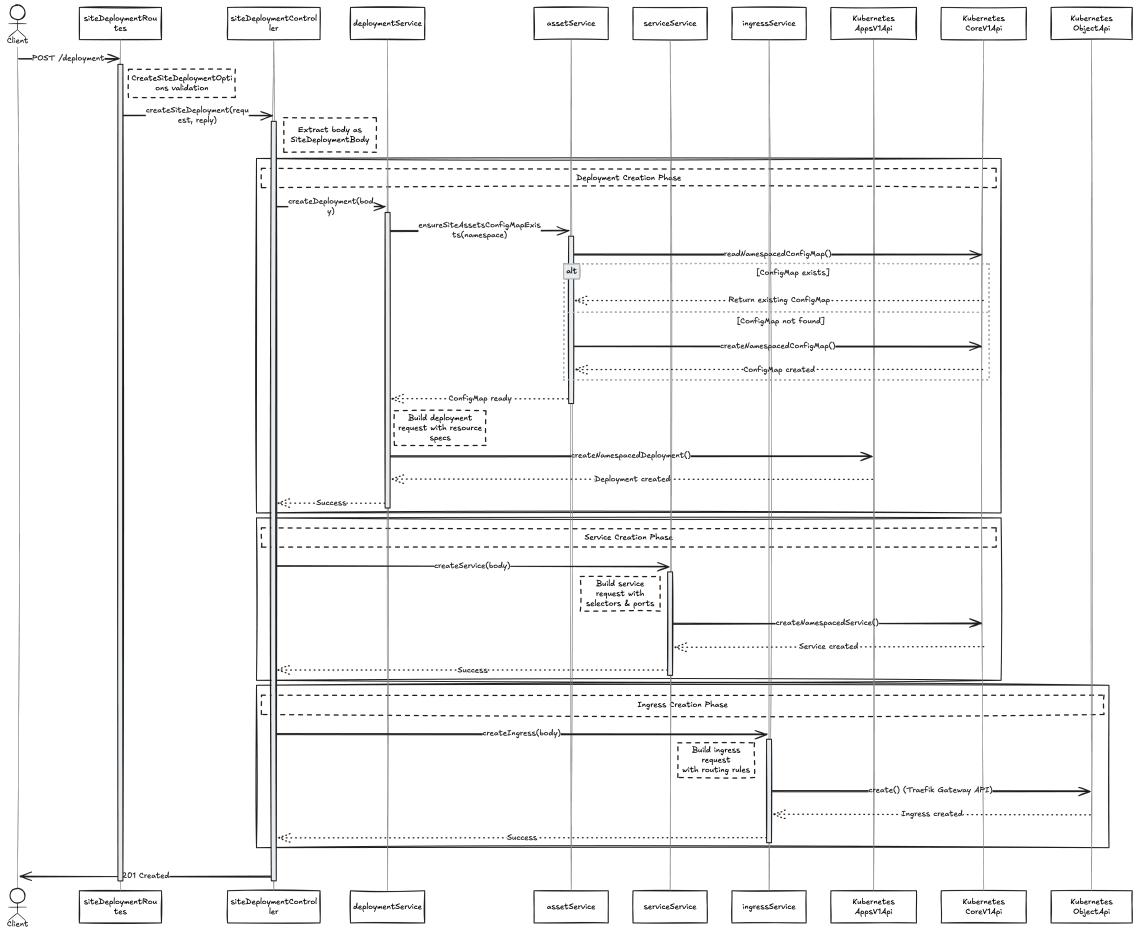


Figure 38: Full deployment sequence diagram

Chapter 7

Testing

Testing is a crucial task to ensure that the application functions now and in the future, according to the given requirements. Even though testing doesn't ensure bug free code, it reduces the number of (unnoticed) bugs, and the chance of errors. It also does not protect against logical errors or oversights. I tested both the frontend and backend applications exhaustively. The proxy application I opted not to test, as it is mostly a lightweight wrapper around the Kubernetes JavaScript client library, which itself is already well-tested by its creators and maintainers.

7.1 Backend testing

Let me start off with elaborating the testing procedure of the backend. First, I needed testing data for the test cases themselves. For this use case, Laravel provides the built-in factory and seeder classes already mentioned partly in Chapter 4.

The factory classes provide a method to create mocked instances of model classes. The mocked values can either be hard coded test values or generated by the FakerPHP package. The aforementioned package boasts a wide variety of methods to create organic feeling testing data. Apart from simple data types, such as numbers and text, complex data for everyday use cases can be generated too. This includes, but is not limited to various file types (images, HTML), emails, addresses, payment information, etc. I created a custom enum for testing the editor content with two cases, the first representing the encoded value, and the second containing the raw value. As such, I implemented factory classes for all of my application's models with fake data.

Seeders, on the other hand, fill the database with data, in this case conveniently created by factories. Inside seeders, while generating fake data, relationships can be handled and generated accordingly, too. I created seeder classes for all models, and a test seeder, which calls each of these to prepare the database for testing.

Laravel also incorporates a way to mock certain aspects of the application, such as event listeners and services. I used this feature to mock the deployment service and sanitizer service during tests. I also faked the HTTP client to intercept outgoing request to the proxy application and provide mocked responses.

Next, the testing environment has to be set up. Sail provides a separate database connection for testing out of the box, to avoid touching the main database. I then configured tests to refresh this test database between each test run and auto seed it with the test

seeder's data. This ensures that the results of previous test runs have no effect on one another, and they are isolated properly.

After this, the test cases themselves were written using the PHPUnit testing framework. It is fully supported by Laravel out of the box and has an object-oriented approach to writing tests. Tests are classes, where each test method has to be prefixed with the `test` word to be picked up by the framework during execution. I wrote unit tests for the two custom services, to ensure they handle external communication and edge cases correctly. I also tested the form request classes' validation logic for incoming data. Lastly, I wrote feature tests for all the controllers' logic, testing CRUD functionality.

These all accounted to a 64% line coverage, which covers 60% of functions and methods. I excluded from the coverage report the Filament and Fortify related functions. The authors of the packages thoroughly test them, and since I used them without modifying the inner workings, they should function as intended. Thus, the implemented tests ensure the backend works according to the requirements. Figure 39 shows the generated coverage report.

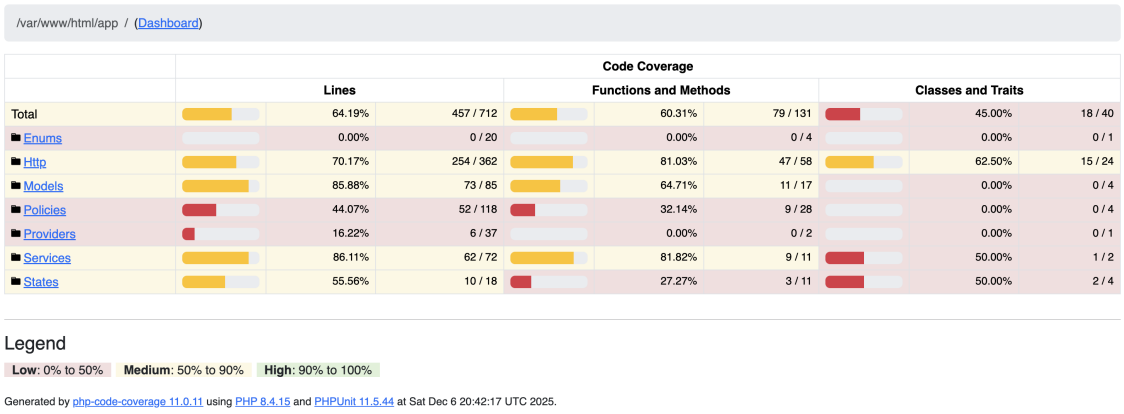


Figure 39: Backend test coverage report

7.2 Frontend testing

React is also unsurprisingly unopinionated about testing, and there are many robust packages that solve this issue. A distinct difference about the available libraries is whether they actually run a browser in which the frontend runs and is tested, or rather the DOM gets simulated, and the tests run in this mocked environment. The latter has the potential problem that the simulated environment is not a one-on-one match with the live one, and the DOM and CSS get handled a bit differently. Due to this, I opted to go with the former method as it ensures the tests interact with the exact same interface as users would.

I chose to use Vitest in browser mode out of the available solutions. Vitest provides comprehensive support for testing all Vite based applications. This means not just React, but Vue and Angular based web apps can be tested using it. I used the package in conjunction with Playwright, which provides the browser engine for the tests.

As the first step, I installed the necessary packages and configured tests in the Vite config file. This included setting the default timeouts, the Playwright provider with the Chromium engine for browser tests, and location of the output, setup, and coverage files.

Next, I created the setup file for Vitest. This included registering mocked hooks, specifying the `beforeAll`, `afterAll`, and `afterEach` methods. The first two handle starting and stopping the mocked worker server (more on this later), while the third resets the handlers of the server and clears the test query client.

Moving on, I created mocks for various aspects of the application. I used the Mock Service Worker (MSW) package, with the OpenAPI-MSW lightweight wrapper to intercept and mock the API calls to the backend and return fake responses. The wrapper ensures that all mocked paths are valid according to the OpenAPI schema, and mocked requests' and responses' structures are correct. A mock handler thus had to be created for each valid endpoint. Similarly to the backend, I used the FakerJS library to create factory methods to mock types and responses with fake but realistic data.

Afterwards, I implemented utilities to create a test query client for TanStack Query and a test router for TanStack Router. I exported a method for rendering routes with the test providers inside tests according to the recommendation of the TanStack Router documentation.

All that remained at this point was implementing the test cases. I implemented some basic component tests, which test that each page renders properly, navigation is possible between them, and the elements react as expected to user input. This mostly means checking that elements having certain texts, tags, and other properties exist at a given time or after an event.

Finally, I implemented two end-to-end tests for comprehensive feature testing. The first one tests the authentication flow by registering a new user, logging out and back in, changing user information and password, and finally logging back in with the new credentials after the automatic logout. The other end-to-end test checks the site management flow. It includes logging in, navigating to the sites view, creating a new site, viewing it in the detail view, opening its home page in the designer, re-saving the design, navigating back to the site detail view, and deploying it.

In Figure 40, the testing browser environment with the successfully executed tests can be seen.

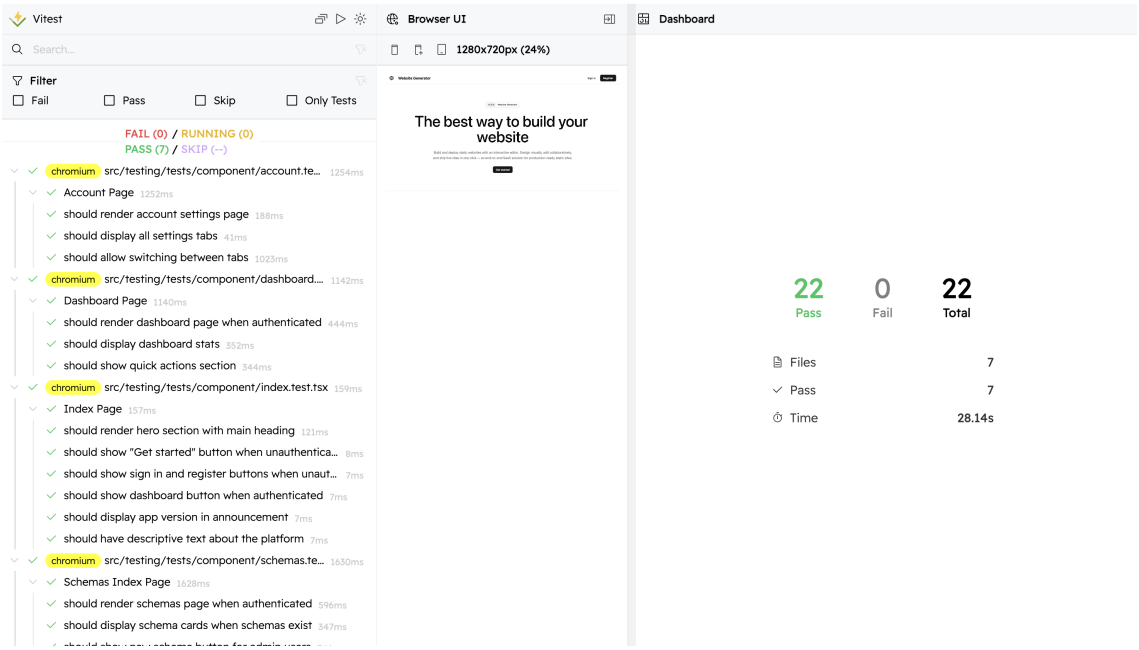
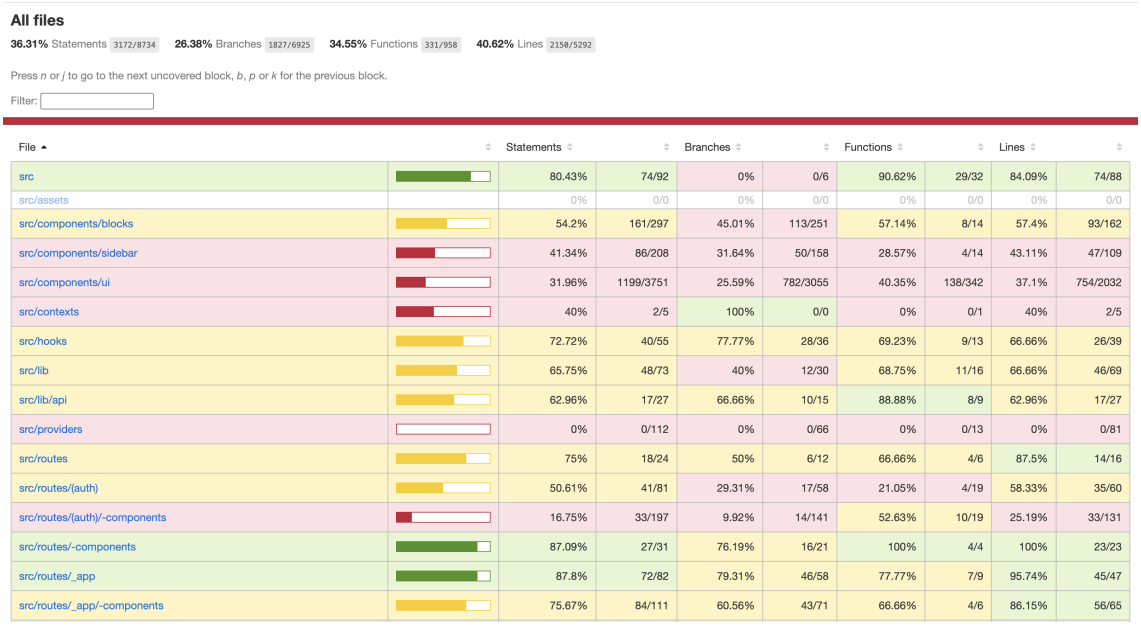


Figure 40: Frontend browser test environment

On the other hand, Figure 41 showcases the coverage report for the frontend. In total, the tests covered about 40% of the frontend.



File	Statements		Branches		Functions		Lines	
src	80.43%	74/92	0%	0/6	90.62%	29/32	84.09%	74/88
src/assets	0%	0/0	0%	0/0	0%	0/0	0%	0/0
src/components/blocks	54.2%	161/297	45.01%	113/251	57.14%	8/14	57.4%	93/162
src/components/sidebar	41.34%	86/208	31.64%	50/158	28.57%	4/14	43.11%	47/109
src/components/ui	31.96%	1199/3751	25.59%	782/3055	40.35%	138/342	37.1%	754/2032
src/contexts	40%	2/5	100%	0/0	0%	0/1	40%	2/5
src/hooks	72.72%	40/55	77.77%	28/36	69.23%	9/13	66.66%	26/39
src/lib	65.75%	48/73	40%	12/30	68.75%	11/16	66.66%	46/69
src/lib/api	62.96%	17/27	66.66%	10/15	88.88%	8/9	62.96%	17/27
src/providers	0%	0/112	0%	0/66	0%	0/13	0%	0/81
src/routes	75%	18/24	50%	6/12	66.66%	4/6	87.5%	14/16
src/routes/(auth)	50.61%	41/81	29.31%	17/58	21.05%	4/19	58.33%	35/60
src/routes/(auth)/-components	16.75%	33/197	9.92%	14/141	52.63%	10/19	25.19%	33/131
src/routes/-components	87.09%	27/31	76.19%	16/21	100%	4/4	100%	23/23
src/routes/_app	87.8%	72/82	79.31%	46/58	77.77%	7/9	95.74%	45/47
src/routes/_app/-components	75.67%	84/111	60.56%	43/71	66.66%	4/6	86.15%	56/65

Figure 41: Coverage report for the frontend

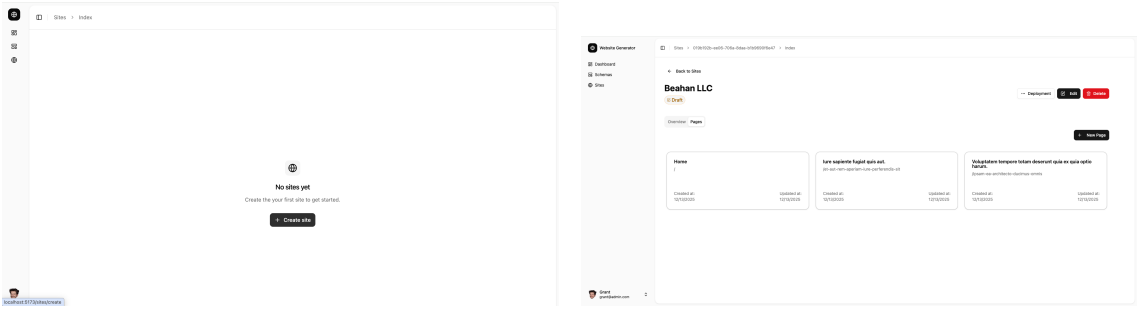
7.3 Usage example

In this section, I will provide a full example of creating, designing, and publishing a website in the developed SaaS system from a user's perspective.

First, let me quickly go through the prerequisites. I am going to assume the user already successfully obtained an account to the application. Not only that, I expect that they already managed to verify their email, which is required for managing the website.

First, the user navigates to the sites page where at first, there is only an empty prompt signifying the user has not yet created a site. Let's create a site, that for the example's sake, showcases the imaginary user's hobbies. The site's name will be 'My hobby site'.

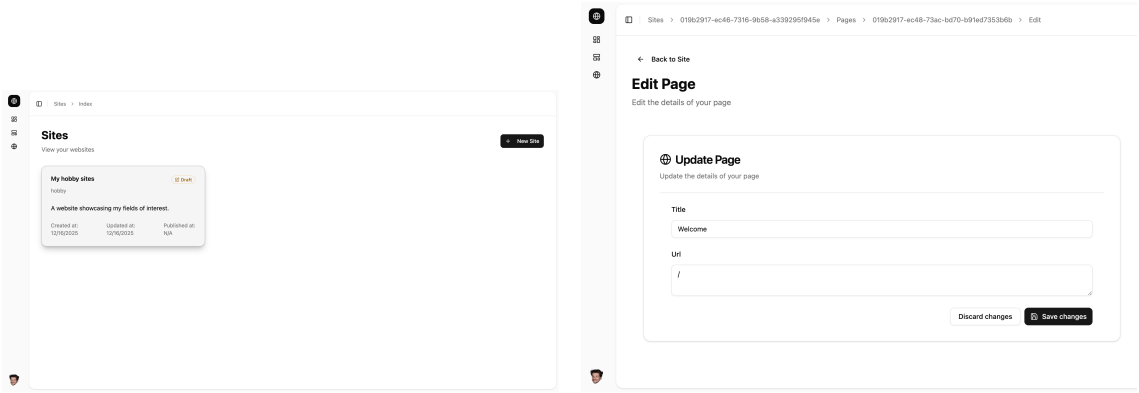
Figure 42 showcases these first two steps. Subfigure 42a shows the empty page, while Subfigure 42b displays the creation form.



(a) Empty sites view **(b)** Example site creation

Figure 42: Creating an example site

Next, the created page shows up in the sites list. Let's edit its autogenerated homepage to be instead called 'Welcome'. This is shown in Figure 43, in which Subfigure 43a shows the newly created site appearing in the list and Subfigure 43b shows the page update form.

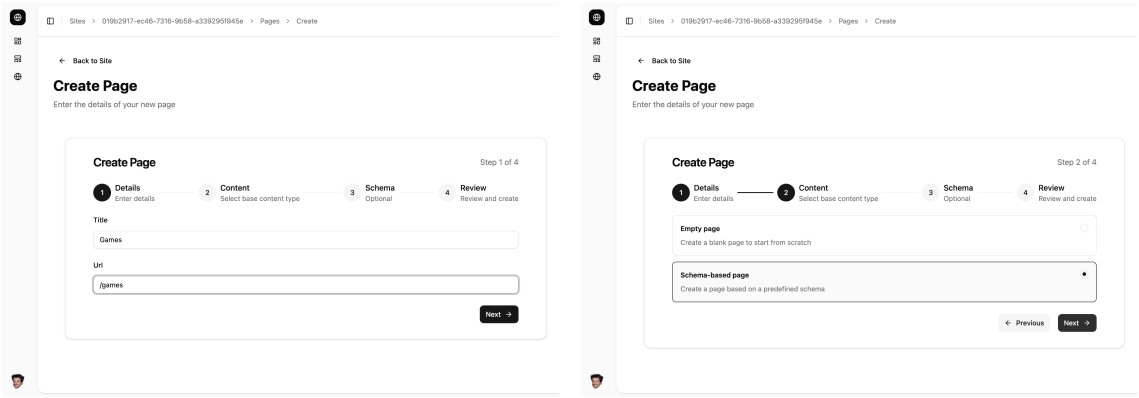


(a) Example site in the list **(b)** Page update

Figure 43: Updating a site's page

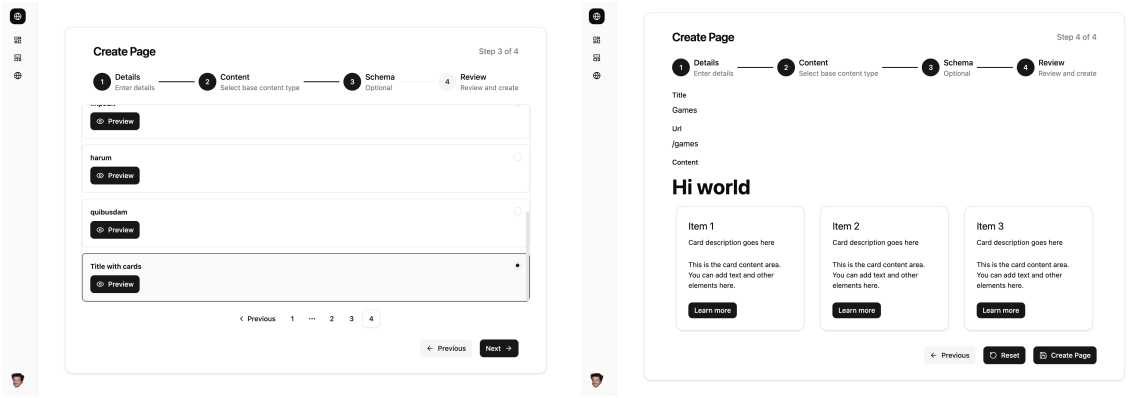
Continuing, let me create a new page for the hobby site. Filling the multi-step form, let's name it 'Games' and give it the path `/games`. Then, I chose to base it off of a schema and selected an example schema called 'Title with cards'. At the final step, I reviewed my selection and saved the page.

This process, with each step outlined is in Figure 44 and Figure 45. Subfigure 44a shows entering the detail of the new page, while Subfigure 44b shows choosing between an empty or schema based page. Subfigure 45a displays the list of schemas a user can choose from and Subfigure 42a shows the review of the process.



(a) Details **(b)** Schema as a base

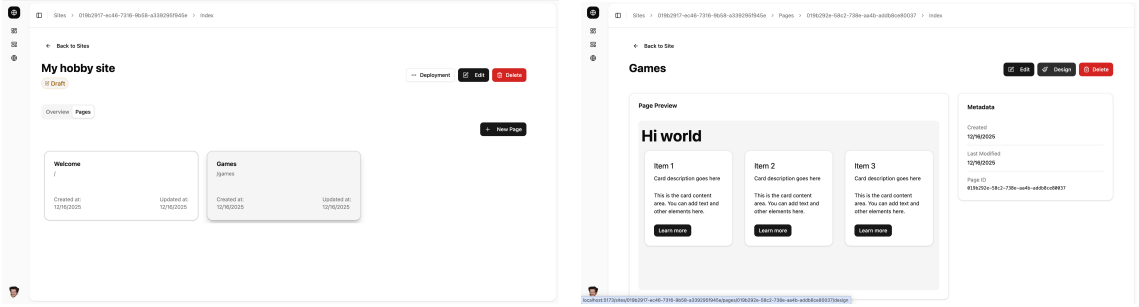
Figure 44: Creating a new page, first two steps



(a) Selecting a schema **(b)** Reviewing

Figure 45: Creating a new page, last two steps

Afterwards, the newly created page appears in the page list of the hobby website. Let's enter its details, which shows the preview of the page (the selected schema), details and controls. Figure 46 shows these steps. Subfigure 46a shows the page in the site's page list, while Subfigure 46b presents the page's detailed view.

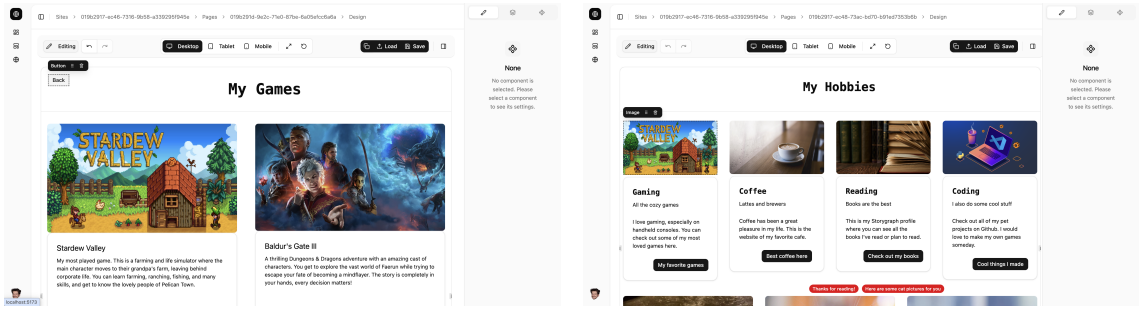


(a) Page list **(b)** Page detail

Figure 46: Newly created page's detail and list view

Let me suggest some possible designs for both pages, which are displayed in Figure 47. Subfigure 47a presents the design of the welcome page, containing cards for each interest

of the user and some cat pictures at the bottom. In contrast, Subfigure 47b presents some of their favourite games in a similar fashion.



(a) Welcome page **(b)** Games page

Figure 47: Example designs

Then, let's return to the site's detail view and deploy the page with the provided controls. Figure 48 presents the publish dialog asking for confirmation.

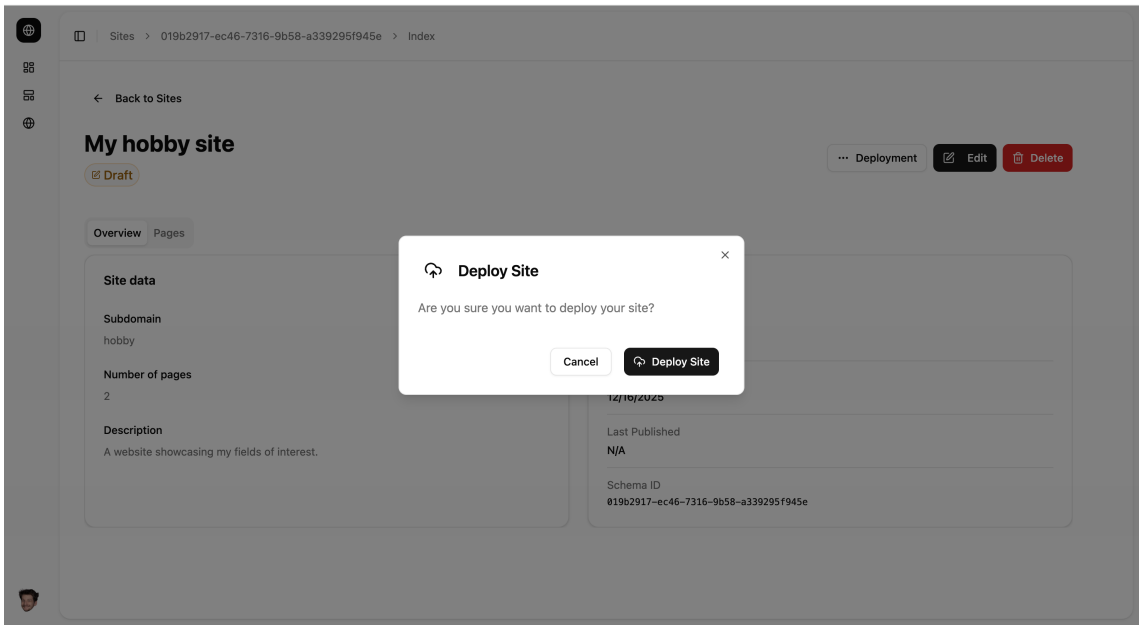
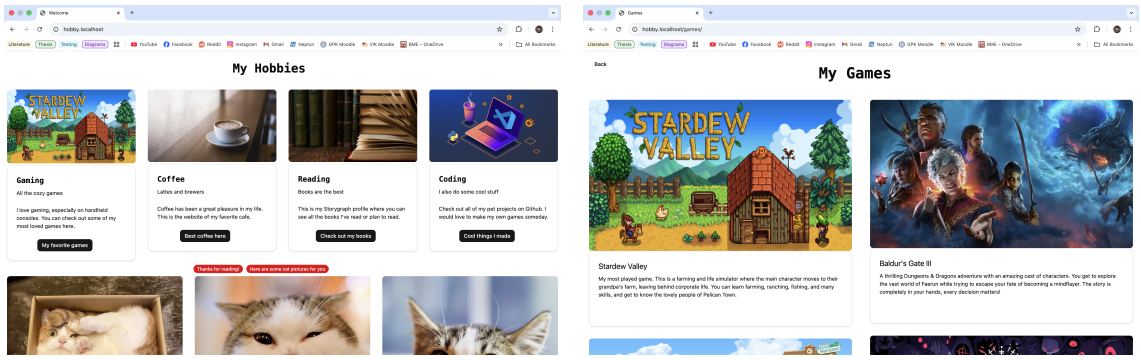


Figure 48: Deploying the example page

Finally, in my local development environment the published site can be visited. Figure 49 shows the two pages accessible on `hobby.localhost`. Subfigure 49a shows the welcome page, and Subfigure 49b presents the games page.



(a) Welcome deployed **(b)** Games deployed

Figure 49: Example site deployed

I will stop the example here, as it already showcases the main functionalities and capabilities of the application. However, based on this short demo, it is easy to see how an end user may use the SaaS platform. It can also be imagined that the user could easily follow these steps again, for example to edit this page further and redeploy the changes for others to see.

Chapter 8

Production considerations

In this chapter, I will outline some key considerations regarding running the SaaS system in a production environment, and its differences to the development environment.

Starting from the frontend, the React application needs to be built and bundled using Vite, and a production ready container configuration needs to be defined. On the Laravel backend's side, Sail is strictly intended as a development solution and the provided Docker container is quite heavy. A lighter base (e.g., Alpine Linux) should be used and configured with production ready environment options and live services. A production ready PHP server should also be employed, such as FrankenPHP. Moreover, Mailpit should be replaced with a proper SMTP (Simple Mail Transfer Protocol) server, and MinIO needs to be replaced with either its own enterprise solution or another provider's S3 compatible object storage solution. A database solution that is easy to scale and is enterprise ready is heavily recommended as well. Also, the proxy app's Docker configuration should be edited as well, and a production grade Kubernetes cluster should be used.

Many cloud-based services exist for the above steps that provide out of the box configuration or detailed help. Of course, the whole SaaS or parts of it could be self-hosted on an on-site infrastructure too if the resources are available. Nowadays, however, it is both cheaper and more convenient to rely on cloud providers, so I am going to focus on this solution. Most notable out of the providers is the big three, namely Azure, AWS, and Google Cloud Platform, which are all well regarded options. They also provide database, object storage and mailing services that can be used out of the box and scaled if needed. However, one should consider backups or replicas in other services as well. If the SaaS uses one provider exclusively, that can prove as a single point of failure, should the provider go offline for a while due to an error in the service.

I suggest the Kubernetes cluster to be based on a cloud provider's solution even if one chooses the main app to be on-site. Such concrete options are Azure Kubernetes Service (AKS), Amazon Elastic Kubernetes Service (Amazon EKS) and Google Kubernetes Engine (GKS), as they have vast amounts of computing resources available. In production, the revision history limit and replica set configurations may need to be reviewed and edited accordingly. Horizontal autoscaling is beneficial to set up as well, to handle increased loads to deployed websites. This can be considered for parts of or the whole of the main application as well. Many providers also have easy to set up and use options for monitoring the cluster, gathering usage and data analytics, and reporting problems or errors, which I highly recommended looking into. Lastly, a domain needs to be acquired, whose subdomains the main application can use to deploy the created sites.

For continuous integration and delivery of the SaaS system, a staging environment should be deployed as well for testing the new version. In a cloud-based production environment, the available resources should make this feasible. Additionally, a blue-green deployment strategy, where two identical production versions of the system, the new and old, run simultaneously could be used. This would allow for switching between the new and old version of the SaaS with zero downtime on updated. For the individual deployed sites, since users can choose when to update with their new designs, this is already solved.

Security is another aspect that needs special attention in a production environment, especially in a SaaS system as the single deployment model and cloud based architecture provide many attacking points. The data of users is highly confidential and should be stored with great care. Tenants' data and resources, such as websites need to be properly isolated, and should never be allowed to mix or affect each other. Additionally, frequent encrypted backups should be created and then stored on a separate remote location, to protect against data loss and local disasters. The dependencies and the environment should be frequently updated to avoid critical vulnerabilities. These exposures could also be mitigated by actively employing tools that scan the system for vulnerable components.

As a hypothetical full example, one might build the React frontend with Vite and publish static assets to a Contend Delivery Network (CDN) backed object store, for instance, S3 with CloudFront. Meanwhile, the Laravel backend could run in lightweight Alpine-based containers with FrankenPHP behind a TLS (Transport Layer Security) terminating reverse proxy. Containers could be deployed to a managed Kubernetes cluster such as EKS, AKS or GKS using Helm charts. A vault might keep secrets, like AWS Secrets Manager or HashiCorp Vault. A managed relational database, namely Amazon RDS (Relational Database Service), Aurora or Cloud SQL, could provide automated backups and read replicas. Mailpit could be swapped for a transactional mail provider, for example SendGrid or Mailgun, with uploads stored on S3-compatible object storage. CI/CD could be handled by GitHub Actions or GitLab CI to promote builds from staging to production using blue-green or canary strategies. Observability could be added with Prometheus, Grafana, and centralized logging, for instance Elastic Stack (ELK) or CloudWatch. TLS can be automated via cert-manager and Let's Encrypt, and tenant isolation can be enforced with network and ingress policies at the cluster level.

I believe the above considerations and the example should help one get started in deploying the developed SaaS into production and help consider various scenarios and use cases. Of course, every situation is unique, so a solution fitting the requirements should be used that fits those needs. Not only that, as the deployed system evolves, changes, and grows the deployment and operational strategy should be fitted to the emerging problems and needs.

Chapter 9

Summary

To summarize, I designed and developed a software as a service platform providing users with the capability to create, design, and publish personal websites without required professional knowledge. I began my work by researching the topic thoroughly and familiarizing myself with modern solutions used today in web application development and cloud based solutions.

Then, I described the requirements expected of the platform and designed the overall system-level architecture of it, detailing each component's responsibilities. Afterwards, I began the implementation of the backend using the modern Laravel framework that provides the core logic and functionality of the system, with an approach that focuses on security, extensibility, and maintainability. I continued the development with the frontend using React, which houses the complex, but easy to use drag and drop editor for designing websites. All the while I paid attention to providing a responsive and modern user experience on the client side. I finished off the implementation by developing a NodeJS proxy application to handle the deployment and orchestration of the created sites to the Kubernetes cluster programmatically.

Next, I provided unit, feature, and end-to-end tests for the implemented platform to ensure it meets the proposed requirements now and in the future. I also provided a full example in great detail, showcasing general usage of the system from a regular user's viewpoint. Finally, I presented some key considerations regarding the production use of the system, focusing on the aspects of continuous integration, continuous delivery, scalability, and redundancies.

Overall, I believe I managed to solve a complex software development task by creating the SaaS platform, and had the opportunity to learn many modern technologies and further my knowledge and skills in the process.

9.1 Further improvements

As with all software products, further refinements, iteration, and improvement is always possible. In this section, I would like to outline a few ideas that could enhance the capabilities of the platform further.

First, the editor could be extended with many more components to allow creating more complex websites. Extending the amount of options for each component is also considerable to allow for further customization.

Version control or a revision history for websites would also be highly beneficial to allow rollbacks in case of problems with new designs. Related to this, a soft deletion system could also be employed with restoration options, and automatic clean-up after a set amount of time.

Next, a higher level of multi tenancy could be implemented that supports multiple organizations, provides custom schemas, and ensures better separation of data.

Finally, data usage, traffic tracking and various other observability and engagement metrics could be integrated into the cluster, such as for example, Prometheus. The data gathered could be relayed to the user in the frontend, which could help in making decisions about further improving their websites. It could also be used for developers and admins alike to make further choices about improving the overall platform.

Acknowledgements

I would like to express my sincere gratitude to Gábor Knyihár, my thesis supervisor for his guidance, support, and valuable feedback throughout this thesis and the previous year's project laboratories.

Many thanks to my family and friends for their encouragement and continued support during the development of this project.

Finally, special thanks to Dóra Kálmánczhelyi for her non-stop encouragement, help with checking for spelling and grammatical errors, and taking the time to put together some example sites and provide valuable feedback.

Bibliography

[1] ISO/IEC22123-1. Information technology — Cloud computing Part 1: Vocabulary. Standard, International Organization for Standardization, Geneva, CH, February 2023.

[2] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[3] Saiqa Aleem, Rabia Batool, Faheem Ahmed, Asad Khatak, and Raja Muhammad Ubaid Ullah. Architecture guidelines for saas development process. In *Proceedings of the 2017 International Conference on Cloud and Big Data Computing*, pages 94–99, 2017.

[4] Javier Espadas, David Concha, and Arturo Molina. Application development over software-as-a-service platforms. In *2008 the third international conference on software engineering advances*, pages 97–104. IEEE, 2008.

[5] Rouven Krebs, Christof Momm, and Samuel Kounev. Architectural concerns in multi-tenant saas applications. *Closer*, 12:426–431, 2012.

[6] WeiTek Tsai, XiaoYing Bai, and Yu Huang. Software-as-a-service (saas): perspectives and challenges. *Science China Information Sciences*, 57(5):1–15, Mar 2014. DOI: https://doi.org/10.1007/s11432-013-5050-z. URL https://link.springer.com/article/10.1007/s11432-013-5050-z.

[7] Farhan Aslam. The benefits and challenges of customization within saas cloud solutions. *American Journal of Data, Information, and Knowledge Management*, 4(1): 14–22, 2023.

[8] Wei-Tek Tsai, Yu Huang, Xiaoying Bai, and Jerry Gao. Scalable architectures for saas. 04 2012. DOI: 10.1109/ISORCW.2012.44.

[9] Matt Stauffer. *Laravel: Up & running: A framework for building modern php apps.* " O'Reilly Media, Inc.", 2019.

[10] Zoltán Subecz. Web-development with laravel framework. *Gradus*, 8(1):211–218, 2021.

[11] Martin Bean. *Laravel 5 essentials*. Packt Publishing Ltd, 2015.

[12] Laravel 12 artisan console. https://laravel.com/docs/12.x/artisan, 2025. Accessed: 2025-12-08.

[13] Laravel 12 documentation. https://laravel.com/docs/12.x, 2025. Accessed: 2025-12-08.

[14] Spatie. https://spatie.be/, 2025. Accessed: 2025-12-08.

[15] Filament php. https://filamentphp.com/, 2025. Accessed: 2025-12-08.

[16] Arshad Javeed. Performance optimization techniques for reactjs. In *2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–5. IEEE, 2019.

[17] Madhuri A Jadhav, Balkrishna R Sawant, and Anushree Deshmukh. Single page application using angularjs. *International Journal of Computer Science and Information Technologies*, 6(3):2876–2879, 2015.

[18] Sagar Ganatra. *React Router Quick Start Guide: Routing in React Applications Made Easy*. Packt Publishing Ltd, 2018.

[19] Merging remix and react routers. https://remix.run/blog/merging-remix-and-react-router, 2024. Accessed: 2025-12-08.

[20] Tanstack router documentation. https://tanstack.com/router/latest/docs/framework/react/overview, 2025. Accessed: 2025-12-08.

[21] Tanstack query documentation. https://tanstack.com/query/latest/docs/framework/react/overview, 2025. Accessed: 2025-12-08.

[22] Tashi Garg. React query: Revolutionizing data fetching and modernizing user interfaces. *Authorea Preprints*, 2025.

[23] Mark Nottingham. Rfc 5861: Http cache-control extensions for stale content. https://datatracker.ietf.org/doc/html/rfc5861, 2025. Accessed: 2025-12-08.

[24] Dave Micheal. Leveraging stale-while-revalidate in react query for optimal ux. 2025.

[25] Mahesh Kumar Bagwani, Virendra Kumar Tiwari, and Ashish Jain. Implementing grapesjs in educational platforms for web development training on aws. *Int. J. Sci. Res. in Multidisciplinary Studies Vol*, 10(8), 2024.

[26] Craftjs documentation. https://craft.js.org/docs/overview, 2025. Accessed: 2025-12-08.

[27] Arun Gupta Alan Hohn. Getting started with kubernetes. *Kubernetes. Retrieved May 2019.*, 24:2019, 2019.

[28] Marko Luksa. *Kubernetes in action*. Simon and Schuster, 2017.

[29] Ingress. Kubernetes ingress. https://kubernetes.io/docs/concepts/services-networking/ingress/, Nov 2025. Accessed: 2025-12-08.

[30] Upendra Kanuru and Upendra Kumar Gurugubelli. Modern approach to kubernetes traffic management: Migrating from ingress to gatewayapi. *International Journal of Emerging Trends in Computer Science and Information Technology*, 6(3):1–11, Jul. 2025. DOI: 10.63282/3050-9246.IJETCSIT-V6I3P101. URL https://www.ijetcsit.org/index.php/ijetcsit/article/view/282.

[31] Rahul Sharma and Akshay Mathur. *Traefik API Gateway for Microservices*. Springer, 2020.

[32] Client Libraries. Client libraries. https://kubernetes.io/docs/reference/using-api/client-libraries/, Jan 2025. Accessed: 2025-12-08.

[33] shadcn.io. shadcn.io. https://www.shadcn.io/, 2025. Accessed: 2025-12-16.

[34] Bulletproof react. https://github.com/alan2207/bulletproof-react, 2025. Accessed: 2025-12-08.

[35] shadcn. Introduction. https://ui.shadcn.com/docs, 2025. Accessed: 2025-12-16.

[36] Stepperize. Stepperize react documentation. https://stepperize.vercel.app/docs/react, 2025. Accessed: 2025-12-16.

[37] Pako: zlib port to javascript. https://github.com/nodeca/pako, 2025. Accessed: 2025-12-16.