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 13, 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 13.

Hanich Péter

Hanich Péter
hallgató

Abstract

Chapter 1

Introduction

Chapter 2

Literature review

2.1 Software as a Service

In order to develop a software as a service (SaaS) 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 NIST (National Institute of Standards and Technology) 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 some 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.

Most importantly for the topic the three service models:

- 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 providers 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 amount 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 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 tenants 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 tenants 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 machines resources is only feasible up to a level. However, most complex production system usually combine the two forms in some form. 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.

Nowadays, many popular applications operate using a SaaS model such as Microsoft's Office suite, Slack, or Notion. In regard to the thesis topic there are also many similar existing solutions such as Squarespace, Wix, GoDaddy, or Shopify. However, they almost always require a subscription or one time payment upfront and or only offer a limited trial. They also frequently have a steep learning curve due to their complexity, and specialize in one given business area.

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 is 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 views to the users [10].

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

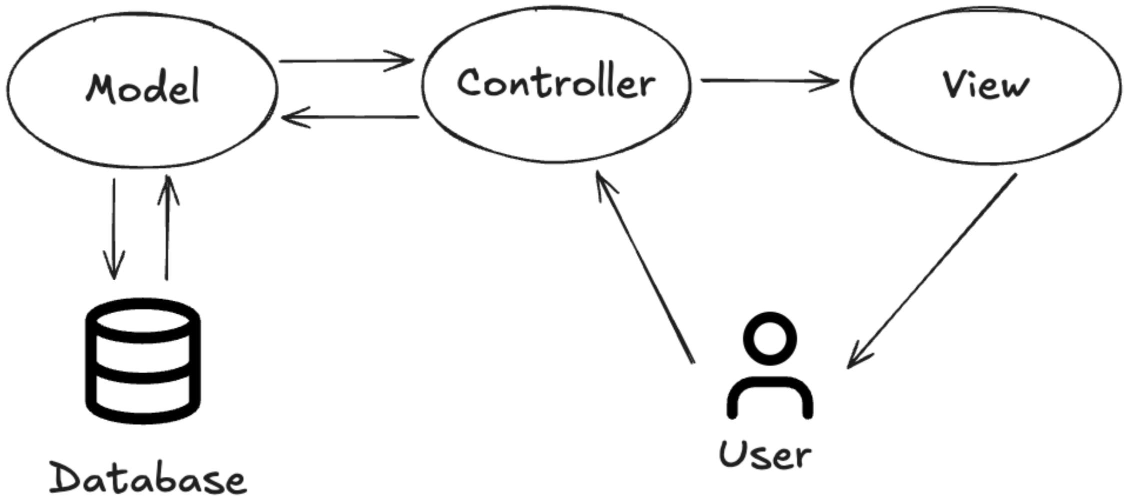

Figure 1: Model-View-Controller architecture

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 kernel which bootstraps all necessary service providers, and executes application level middleware. These include the database, queue, validation, and routing components among many. 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 on Figure 2

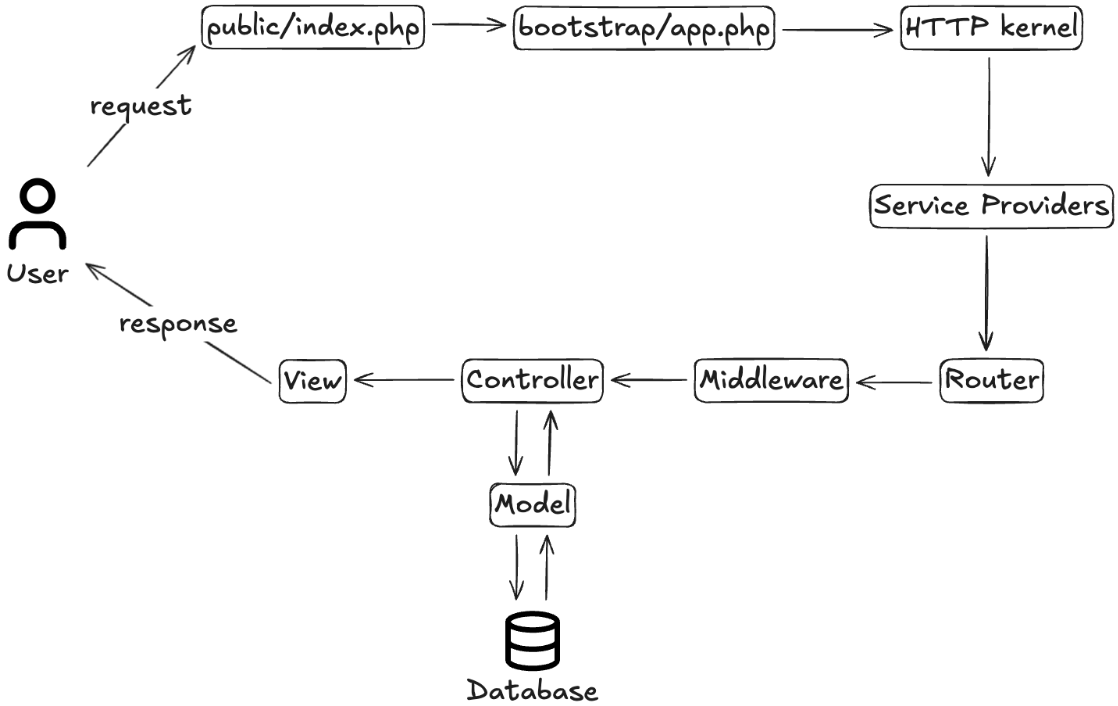

Figure 2: Laravel request lifecycle

2.2.2 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 [12]. 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 [13] whose team maintain many packages and tools for the framework and FilamentPHP, which is a plugin extendable admin panel and UI (user interface) framework [14].

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 (UI) 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 [15].

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 [16].

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 [17]. 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 [18]. 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, and URL parameter validation and native TypeScript support and type safe navigation [19].

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 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)) [19].

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 [20]."

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 providers 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 [21].

Under the hood the library uses Stale-While-Revalidate (SWR) caching which was put forward by RFC 5861 [22]. 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 [23]. This can be seen illustrated on Figure 3.

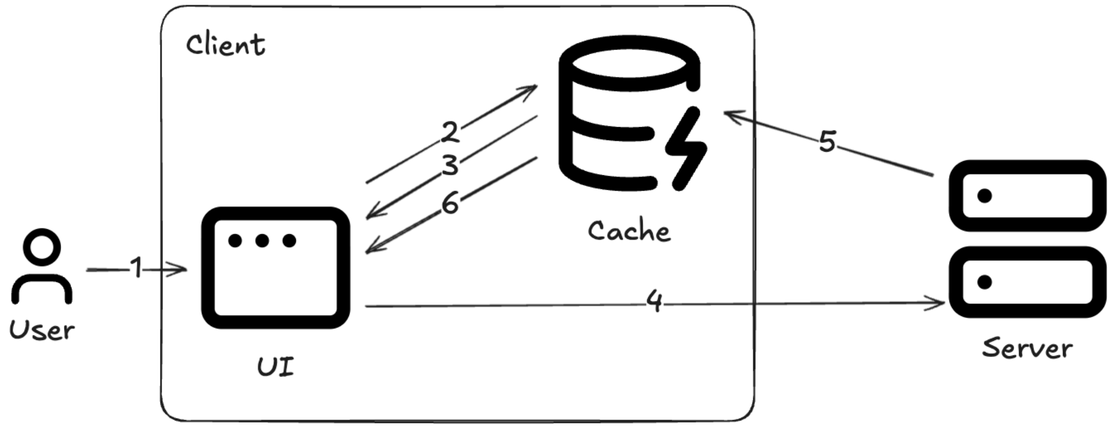

Figure 3: Stale-While-Revalidate caching

2.3.3 Editor libraries

Due to JavaScript's and by extension Reacts 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 ones own editor with drag and droppable UI elements [24]. Ultimately I decided against GrapesJS as the CraftJS is built more around React, while the former relies more heavily on pure JavaScript elements. CraftJS modularizes the editor an allows one to fully customize it according to their own needs [25].

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) [26]."

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 system or 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 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 its 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 [27].

2.4.1 Ingress, Gateway API

Since Pods can be short-lived, often recreated and have multiple replicas, a specialized service is needed to access HTTP routes from outside to 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 [28].”

The Gateway API is in a way is a successor to the Kubernetes Ingress and provides a more modern and scalable solution to route and traffic management [29].

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 [30].

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

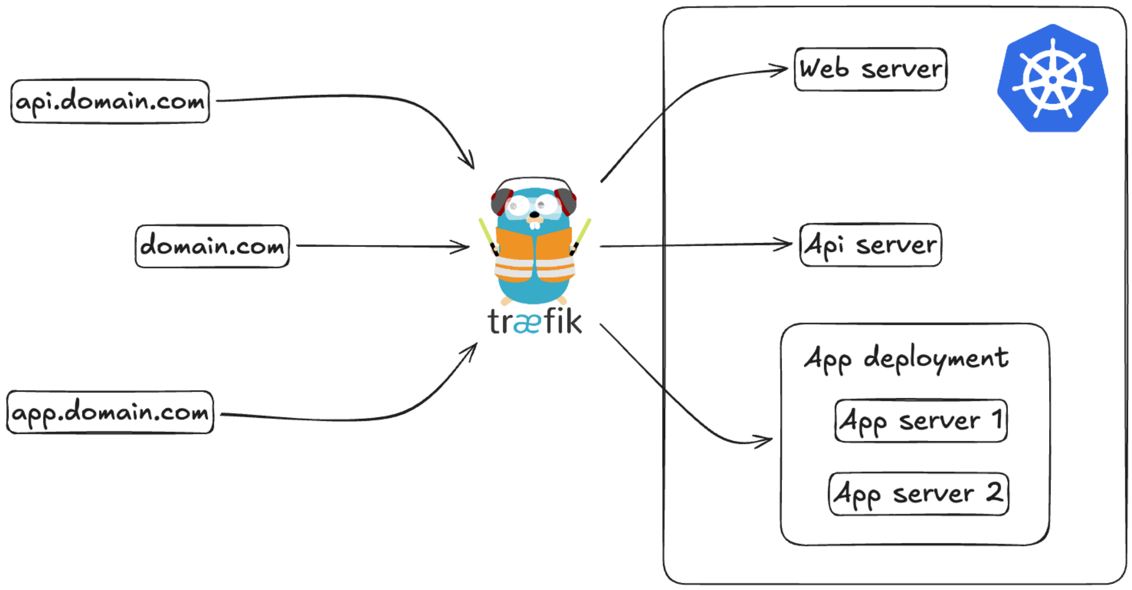

Figure 4: Traefik Gateway API

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. Special attention must be paid to scalability, security, and user experience (UX). 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 on Figure 5.

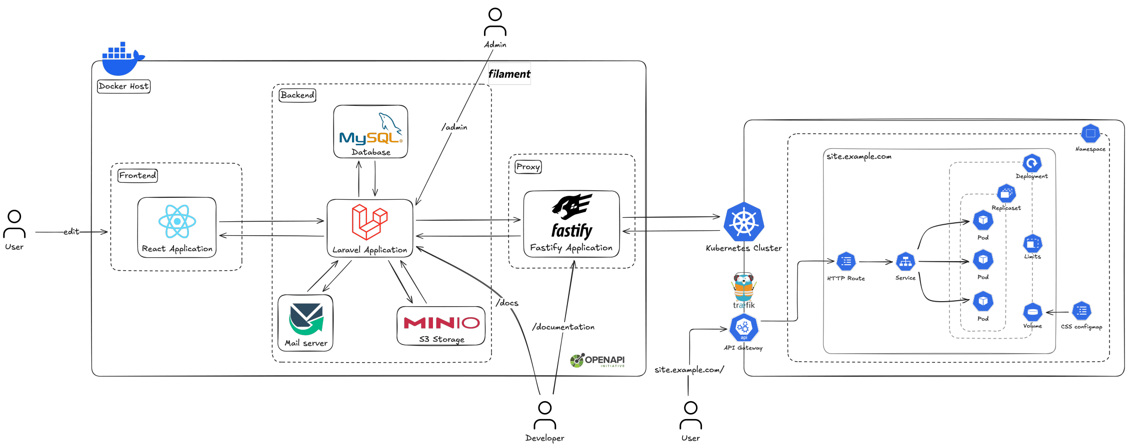

Figure 5: System level architecture

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 built the frontend primarily using React with the Vite build and bundling tool. It communicates with the backend through a REST API.

The backend relies mainly on Laravel, but also uses 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 serves the admin panel using FilamentPHP for the implementation, while Scribe is used to generate the backends API documentation using the Scalar API documentation platform. The backend communicates with both the frontend and the proxy application also through REST.

The Fastify NodeJS web framework provides the basis for the proxy application and the Kubernetes JavaScript library to connect to the running cluster. It also serves its own OpenAPI specification using the native Fastify OpenAPI plugin. For the reason of this functionality having its own separate application refer to section 4.3.

The proxy connects programmatically to the Kubernetes cluster to deploy the created websites using its official JS client library. The websites are all deployed to a given namespace ('default' namespace if not changed) using a Kubernetes Deployment. The Deployments have the common style CSS (Cascading Style Sheets) attached as a volume through a ConfigMap. Furthermore, each Deployment by default has a 10 revision limit, default resource limits set, and a one replica requirement. In production these values could obviously be reconfigured and scaled up. Finally, the application also creates 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 is able to create and design their website on the frontend which is then saved to the backend and through the proxy app deployed to the running Kubernetes cluster. Outside users can then visit the deployed website through the Traefik API Gateway acting as a reverse proxy.

Chapter 4

Development

4.1 Backend

4.1.1 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. Finally, the application key must be initialized using the Artisan CLI and the default database migrations have to be run.

4.1.2 Eloquent ORM

Let me start building up the backend's feature set from the bottom, starting with the parts of the Eloquent ORM. The database schema of the backend including the most important tables can be seen on Figure 6.

Note: 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.

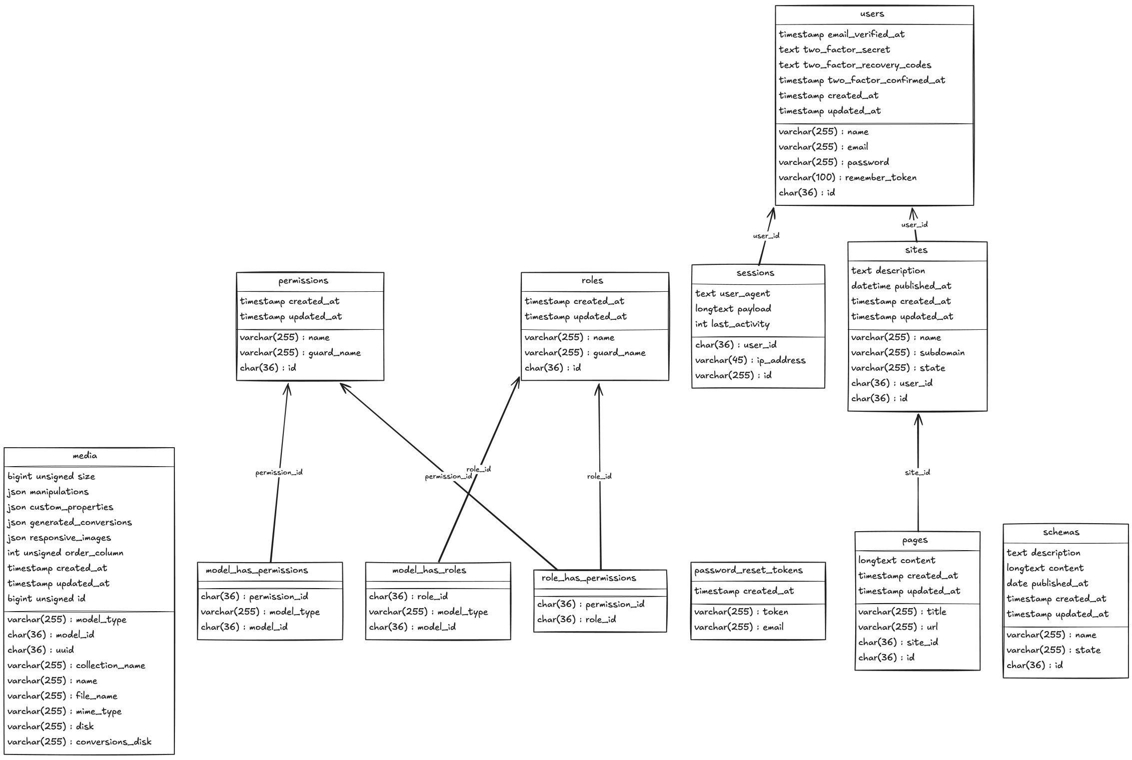

Figure 6: Database schema

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.

The Spatie Laravel Permission package adds the role related tables, which I will talk about in more detail in the authorization Subsection 4.1.4.

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 store 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 implementations (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 Section 4.2. 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 it's 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. Models register media collections for storing files by setting its 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.

4.1.3 Authentication

4.1.4 Authorization

4.1.5 Core CRUD

4.1.6 Deployment Logic

4.1.7 Admin Panel

4.2 Frontend

4.3 Proxy application

Chapter 5

Testing

5.1 Backend testing

5.2 Frontend testing

5.3 Usage example

Chapter 6

Production considerations

Chapter 7

Summary

Acknowledgements

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 documentation. `https://laravel.com/docs/12.x`, 2025. Accessed: 2025-12-08.

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

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

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

[16] 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.

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

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

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

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

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

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

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

[24] 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.

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

[26] T Kubernetes. Kubernetes. *Kubernetes. Retrieved May*, 24:2019, 2019.

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

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

[29] 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`.

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