Headless CMS for Designing Connected Content (Jamstack)

Patrick Berchtold, Nico Gabriel, Martin Rabensteiner, and Marcel Zisser

706.041 Information Architecture and Web Usability 3VU WS 2024/2025
Graz University of Technology

13 Dec 2024

Abstract

One of the core concepts of the Jamstack architecture is the decoupling of the content management system (CMS) from the presentation of the content (frontend). This offers flexibility for web developers by letting them choose a CMS that fits their individual needs, but the freedom of choice can also be overwhelming. In this survey, we provide a comparison of two popular headless CMS platforms, WordPress Headless and Strapi, by building a web site that can be run with either CMS. Furthermore, we compare different API systems by using REST API and GraphQL for WordPress and Strapi, respectively. We want to highlight the difference in effort during the development phase, as well as the performance difference for a live application. Through this dual comparison, we aim to ease the decision between different headless CMS architectures and API strategies in the context of a Jamstack-driven environment.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

In this survey we explore different content management systems (CMSs), as well as different API architectures. We do this by building a simple conference web site using the Jamstack architecture, which can connect to different CMSs using different APIs. We selected Strapi and Wordpress Headless for the CMS comaprison, and REST and GraphQL for the API comparison. For our frontend we selected Astro as a static site generator.

1.1 Jamstack

Jamstack is a web development architecture that decouples the frontend from the backend. It relies on pre-rendering static assets and delivering them through a Content Delivery Network (CDN), while dynamic functionalities are handled via APIs and serverless services. A common setup involves a frontend, which connects to the CMS using an API, clearly separating the frontend from the CMS. In contrast, traditional web applications are "monoliths", where the frontend and the CMS is bundled and tightly connected, as can be seen in Figure 1.1. Using a headless CMS in a Jamstack architecture shifts focus to pre-built, static outputs that can be efficiently cached and served globally, offering potential advantages in performance and security compared to monolithic systems.

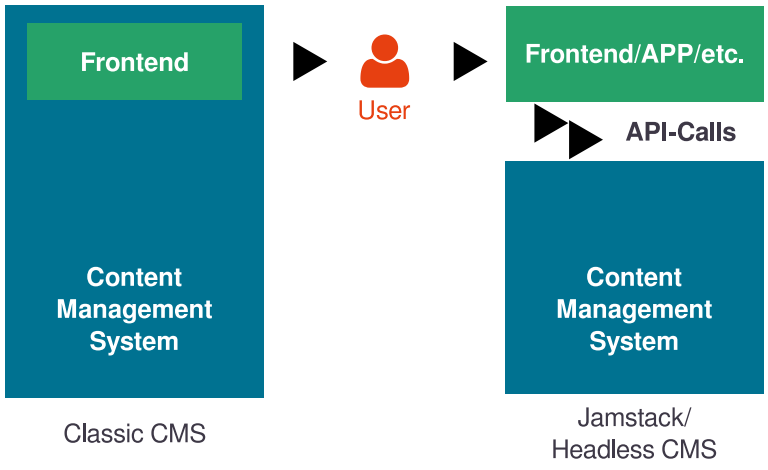

Figure 1.1: The Jamstack model compared to the traditional CMS architecture.

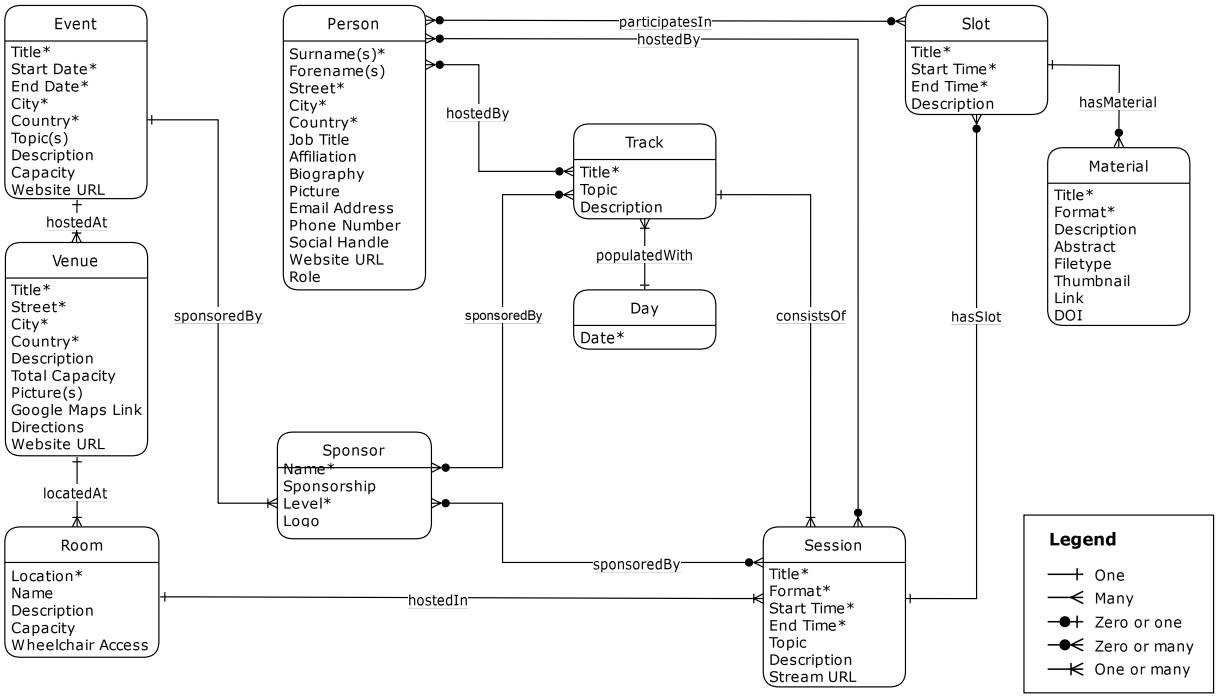

Figure 1.2: The content model for the web site, showing the used objects with their properties and relations to other objects. Adapted by the authors from a content model by Yannik Rauter, under the terms of a CC 4.0 licence.

1.2 Content Model

The diagram in Figure 1.2 shows all the objects (for example Event) of the content model, including all the object's properties, where an asterisk denotes a mandatory property (for example Title of an Event). The relations are also part of the content model, for example there is an m-to-n relationship between Person and Slot. The content model is based on a content model from Yannik Rauter which originally had the Role of a person as its own object, with a relation to both Person and Event. This would make sense to represent for example a chairman of an event, but we only include the speakers in the web site, so this was not needed for this use case.

Chapter 2

Headless Content Management Systems

A Content Management System (CMS) stores content for applications such as web sites. A CMS typically provides an authoring interface for an administrator to define a content model for the content to be stored, as well as an authoring interface for authors to then input and maintain content. Traditional CMSs provide a built-in user interface for end users to access and view the content, as well as one or more APIs for programmatic access to the content. A *headless* CMS focuses on programmatic access to the content through APIs, and may not provide a built-in user interface for end users at all.

2.1 Wordpress

Wordpress is one of the most widely used content management systems (CMS) worldwide [Wordpress 2024b]. Originally started as a blogging platform, Wordpress reached high popularity by having a simple functionality that can be adapted in many ways. With the open source concept, adaptions become easily possible, and around Wordpress a large community with many plugins and themes grew. This advantage can also become a disadvantage in specific use cases where many requirements are given, for example as a headless content management system with a given, sophisticated content model.

2.1.1 Installation and Requirements

The requirements of Wordpress are simple and have been standard on the web for many years. The current version requires at least PHP 7.4 and either MySQL 8.0 or MariaDB 10.5 or greater. The installation and configuration of Wordpress is user-friendly and webserver providers also support it with installation wizards.

2.1.2 Custom Fields

For standard page entries in a Wordpress instance, the CMS offers a very simple custom field functionality. It consists of simple key-value-pairs which can be set on each page individually. This is already the disadvantage, because templates for custom fields are not possible and therefore it cannot be ensured that specific fields are reliably available on different pages.

2.1.3 Secure Custom Fields

Due to the very limited out-of-the-box custom field functionality, third-party plugins are needed for higher requirements on custom fields. The Secure Custom Fields (SCF) plug-in makes it possible to model a more sophisticated content model within Wordpress, and is maintained by the Wordpress foundation itself. It was recently forked from the former Advanced Custom Fields (ACF) plugin, and the old name and

abbreviation are often still used [Wordpress 2024a]. SCF provides three types of customizations which are integrated into the backend, such that they give the same user experience as predefined Wordpress features.

2.1.3.1 Post Types

In the default Wordpress configuration, there are three fixed post types: post, page, and media. With the plugin, it is possible to define custom post types to reflect the content model. A post type can then be for example a person or conference venue.

2.1.3.2 Field Groups

A field group is the heart of the custom field functionality for a content model. For the person post type, the attributes like the name or the contact data are then defined as fields in a field group. The types of fields vary from simple text or number boxes to bi-directional page links, making sure that information is updated correctly and the relations stay in a correct manner. Within the field group, also the link to a post type takes place. This makes it possible to easily reuse one field group for multiple post types, but on the other hand to also provide different field groups for one post type.

2.1.3.3 Taxonomies

Taxonomies can be used to categorize custom post types without having the categories as another post type. With the given content model, it was not necessary to use taxonomies.

2.1.4 API

Wordpress has an integrated REST API (see Section 3.1) which is activated by default and makes the default Wordpress pages and posts available there, as well as custom-defined post types from the SCF plugin. An example for an API call would be the path `/wp-json/wp/v2/person/1`, where `person` is the custom-defined post type and `1` is the unique post id. To make sure that the fields of a field group are available in the API endpoint, the option “Show in REST API” has to be enabled in the field group configuration. The custom fields are then listed in the `acf` array at the API endpoint.

2.1.5 Adaptions for Headless Operation

To improve the operation of the CMS for a headless concept, the default Wordpress frontend should be deactivated and rerouted to the server where the actual webpage is delivered. This is not possible in a simple setting. Instead, a third-party theme or a short self-coded snippet to display an empty page or forward to the actual web page is necessary. The Wordpress API is already out-of-the-box ready to be used for a headless operation mode or for other distributed services.

2.1.6 Summary

Wordpress has a large community, many expanding possibilities and is well established. However, its main purpose is to be a general all-rounder CMS and for a smooth headless operation mode, plugins and adaptions are needed.

2.2 Strapi

Strapi is an open-source headless content management system (CMS), released under the MIT License [Strapi 2024]. It is designed to be flexible and scalable. It is built on Node.js and can therefore manage requests asynchronously. Its headless design allows decoupling of the frontend from the backend. Strapi offers customizable data models and a range of plugins. It supports both REST and GraphQL APIs and includes role-based access controls by default. There are multiple ways to install the latest version of Strapi (Version 5).

2.2.1 Installation from CLI

One option is self-hosted installation using the command line interface (CLI), which requires a web server that supports Node.js version 18 or above. Strapi is compatible with the following databases for storing the content and configuration:

- SQLite (default)
- MariaDB
- PostgreSQL
- MySQL

Installing Strapi using the CLI can be done using a single command:

```
npx create-strapi@latest my-strapi-project
```

After entering this command the user is guided through the following six steps of the installation process:

- Do you want to log in or sign up to Strapi-Cloud?
- Do you want to use the default database (sqlite)?
- Start with an example structure & data?
- Start with TypeScript
- Install dependencies with npm?
- Initialize a git repository?

At the and of this process, the user is prompted to generate an admin account. Strapi is set up and started in development mode after account creation.

2.2.2 Hosting on Strapi Cloud

Another option is to use Strapi Cloud, a premium hosting solution provided by Strapi. This is a subscription-based service that allows users to host their Strapi CMS without needing to manage their own infrastructure.

2.2.3 Self-Hosted Docker Container

Strapi, according to its official documentation, can be dockerized to run within containerized environments. While Strapi does not provide official container images, the documentation offers a short guidance on creating a custom Docker image.

2.2.4 Content-Type Builder

The Content-Type Builder in Strapi is a feature that allows developers to define and manage three content types: Single Types, Collection Types, and Components. It provides an interface for creating and customizing content structures by defining fields, configuring relationships, and specifying validation rules. The tool eliminates the need for manual coding in structuring data models and supports field types such as text, numbers, dates, and media. It also enables the creation of one-to-one, one-to-many, and many-to-many relationships between content types. Changes made in the Content-Type Builder are automatically applied to the Strapi API, ensuring data structure consistency throughout the application.

Strapi's Content-Type Builder is only available in development mode and is automatically disabled in production mode. This restriction is designed to prevent unauthorized or unintended modifications to content models while the application is in production, ensuring integrity of the deployed application.

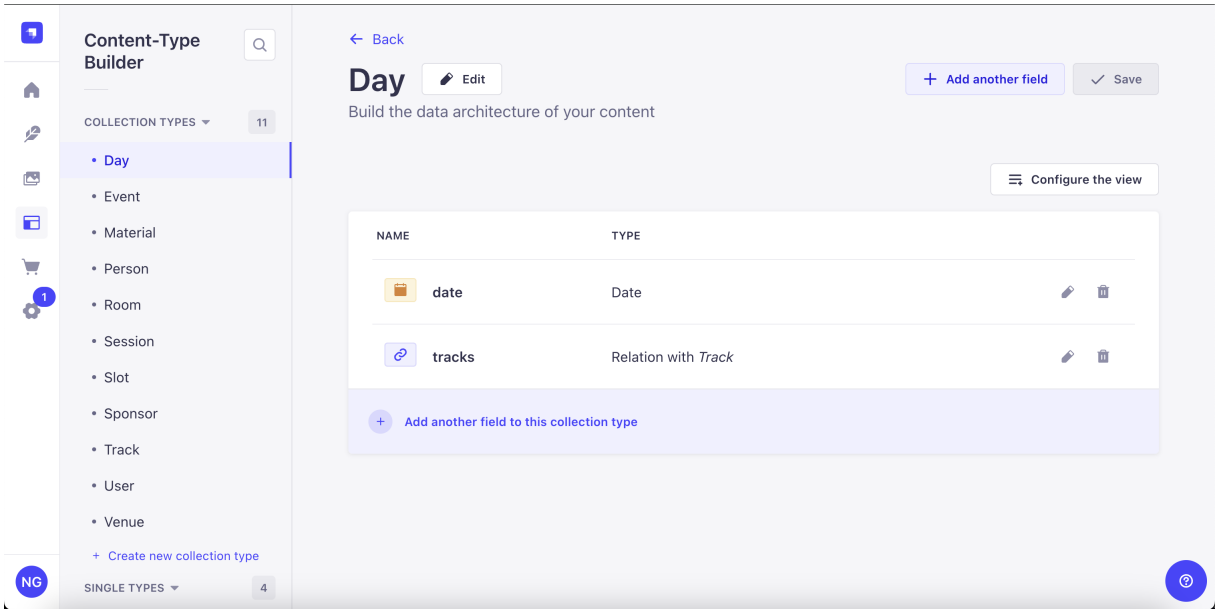

Figure 2.1: The Content-Type Builder interface in Strapi.

2.2.5 Components

Strapi Components are reusable combinations designed to group and organize fields that are commonly used together. They allow developers to create consistent content structures, making it easier to manage repetitive data. Components can include various types of fields, such as text, numbers, media, and relationships and can be used across multiple single or Collection Types. Components can also be nested inside other Components.

2.2.6 Single Types

In Strapi, Single Types are a type of content structure, which is used to manage unique pieces of content that do not require multiple entries. They were built for static pages such as a homepage, about page, or license page. Single Types consist out of multiple fields and Components, which can be added and customized as needed.

2.2.7 Collection Types

Collection Types are a content structure designed to manage multiple entries of the same type. They are ideal for dynamic content that needs to be repeated or listed, such as events, products, user profiles, or locations. Fields and Components within the collection type can be configured to define its structure and relationships to other Collection Types.

2.2.8 Content Manager

The content models created in the Content-Type Builder can be utilized in the Content Manager, allowing users to add modify or delete entries for both Single and Collection Types. The Content Manager includes features such as filtering, sorting, and searching through entries. It also offers the ability to set content in draft or published mode, enabling users to manage content visibility. Additionally, it supports role-based access controls, allowing administrators to define permissions and restrict content management actions based on user roles.

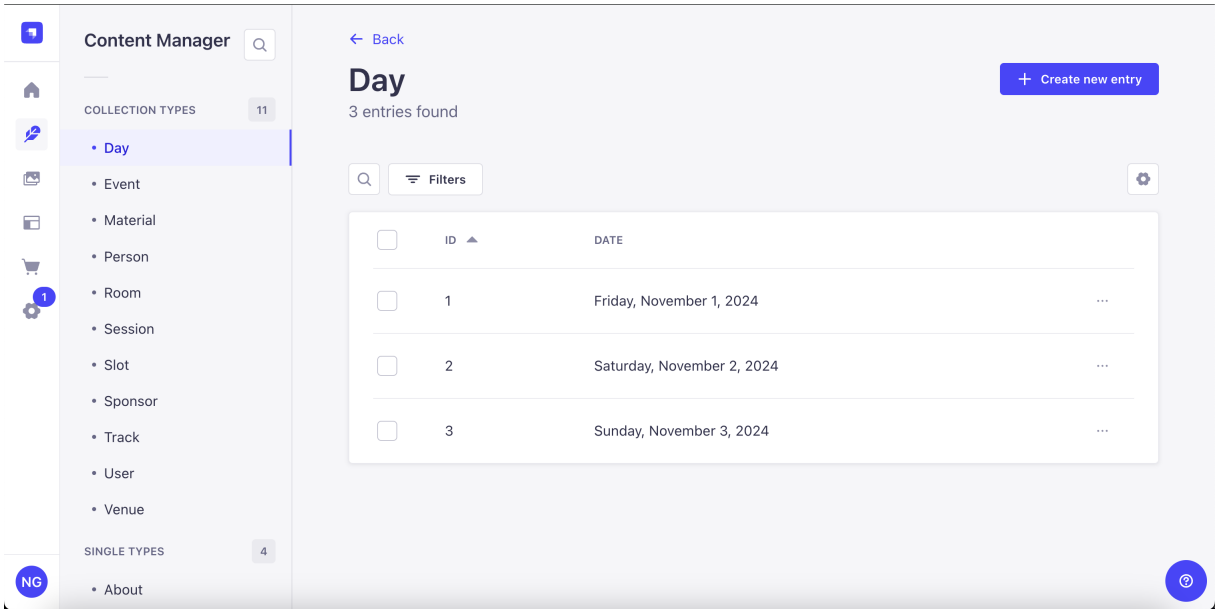

Figure 2.2: The Content Manager interface in Strapi.

2.2.9 Media Library

The Media Library is the centralized feature for organizing all media assets used within the CMS. It supports multiple file types, including images, videos, documents, and audio files. The library provides options for categorizing assets, adding metadata, and managing files in folder. External storage providers like AWS S3 and Google Cloud are also supported.

2.2.10 API

Strapi's API endpoints are automatically generated based on the content types defined in the Content-Type Builder, supporting both REST and GraphQL APIs. This dynamic generation ensures alignment with the content models, requiring no additional configuration from developers. The API comes with role-based access control, allowing administrators to define permissions for different user roles.

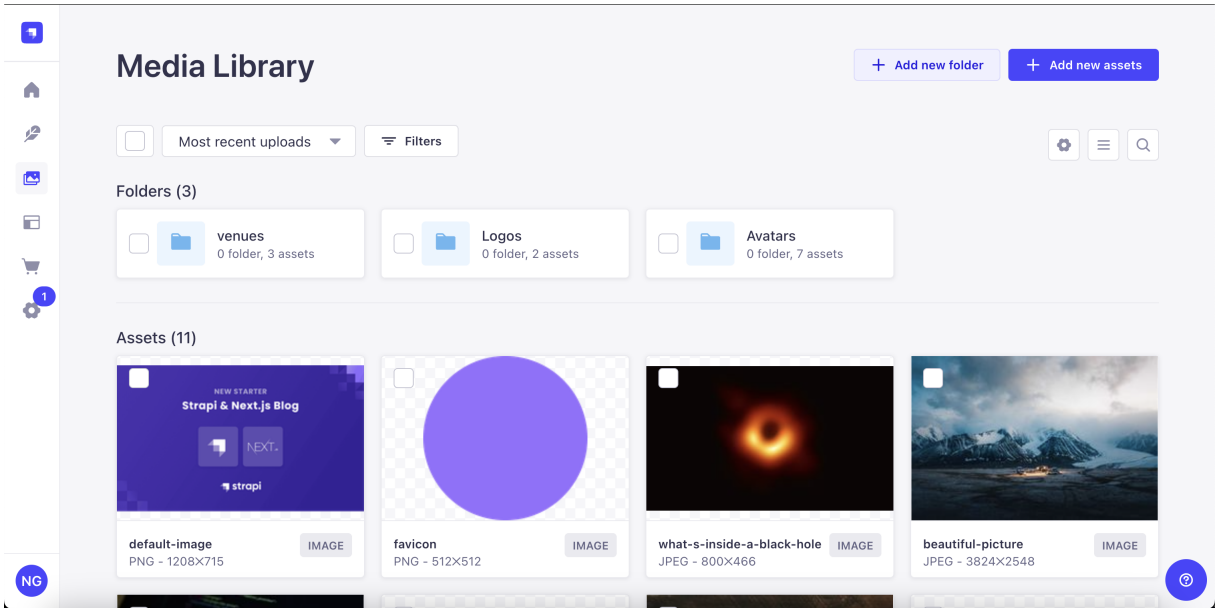

Figure 2.3: The Media Library interface in Strapi.

Chapter 3

APIs

In order to connect frontend and backend systems, different architectures have emerged, each with different strengths and weaknesses. In this chapter two prominent API architectures are discussed and compared: REST, which is simple and traditionally very widely used, and GraphQL, which has a more complex structure, but aims to solve some common problems that occur with REST.

3.1 REST

REST (Representational State Transfer) is a commonly used architecture introduced by Fielding and Taylor [2002], which leverages methods from the HTTP standard to interact with resources, for example GET to get a resource, or POST to create a resource. There is typically exactly one endpoint per resource, which can lead to under- or overfetching of data. Underfetching occurs when the client needs multiple resources, which requires the client to make multiple requests. This can negatively impact performance, since each request causes a certain overhead. Conversely, overfetching happens when a request fetches more data than is actually needed, for example when the client only needs part of a resource, wasting bandwidth with each request.

3.2 GraphQL

GraphQL is a query-based API developed by Facebook and open-sourced in 2015, formally specified in GraphQL [2024]. Unlike REST, GraphQL allows clients to precisely specify the data they need, preventing overfetching and underfetching. A key concept in GraphQL is the schema, which defines the types and relationships between resources. Clients can construct queries to fetch specific fields from multiple resources in a single request, optimizing network traffic and improving performance. For instance, a client could query for a user's name, email, and a list of their posts in a single request. This reduces the number of API calls, reducing the overhead that comes with each request. To update resources, GraphQL uses mutations, allowing clients to create, update, or delete data.

3.3 API Comparison

In this section we will discuss the key differences between the two API architectures, both from the developer's point of view, and performance wise.

3.3.1 Structure

In order to get a resource, a simple `GET` call is required for REST API. To show this in an example, the call to get a `person` with an id of `123`, a typical call would look like this:

```
GET /rest/person/123 HTTP/1.1
```

```
POST /graphql HTTP/1.1
Content-Type: application/json
Content-Length: 84

{
  "query":  "query Person {
    person(id: 123) {
      name
    }
  }"
}
```

Listing 3.1: Example GraphQL query to retrieve a person with ID `123`.

This would return the whole person object. To achieve a similar result using GraphQL, a `POST` call with a query is required. In such a query, all fields need to be explicitly listed (and defined in the schema beforehand), for example if the client only wants to know the event's name, a query would look like Listing 3.1.

Note that, for ease of reading, line breaks and indentation have been added to the payload, so the payload is not valid JSON. Line breaks would need to be substituted by control characters like `\n`. In order to fetch multiple resources, e.g. a person plus five slots that they participate in, one API call per resource is required in REST, for example six calls in total. In GraphQL, this can be achieved with a single call, if the `person-slots` relation and the behaviour of a `limit` parameter is defined properly in the schema. This can be seen in Listing 3.2.

To create a resource using REST, it is typical to just call the same endpoint with a `POST` request, and to send the resource data in the request body as in Listing 3.3. To create a resource using GraphQL, a query with the `mutation` type is used as in Listing 3.4.

3.3.2 Performance

REST and GraphQL operate on different principles, making each better for a different kind of environment. GraphQL's performance advantages are vital for large scale applications, where every performance optimization can have a huge impact on large user base. This comes at the cost of increased complexity (increasing maintenance efforts for developers), and increases setup time, since every schema and query needs to be defined explicitly. Due to this, REST may be sufficient for smaller projects, quick prototypes, or minimal viable products.

```
POST /graphql HTTP/1.1
Content-Type: application/json
Content-Length: 84

{
  "query":  "query Person {
    person(id: 123) {
      name
      slots(limit: 5) {
        title
      }
    }
  }"
}
```

Listing 3.2: Example GraphQL query with relation and limit parameter.

```
POST /rest/person HTTP/1.1
Content-Type: application/json
Content-Length: 23

{
  "name": "John Doe"
}
```

Listing 3.3: REST POST request to create a person named “John Doe”.

```
POST / HTTP/1.1
Content-Type: application/json
Content-Length: 114

{
  "query": "mutation CreatePerson {
    createPerson(input: {
      name: \"John Doe\"
    })
  }"
}
```

Listing 3.4: GraphQL POST request to create a person named “John Doe”.

Chapter 4

Frontend

To complete the Jamstack [Netlify 2024], a frontend is needed. Specifically, Jamstack typically utilizes a Static Site Generator (SSG). For this survey paper, the design of the frontend is supposed to imitate a modern conference web site. It provides general information about the conference such as information about the speakers, venues, and schedule. Based on an evaluation by Netlify [2023], Astro [Astro 2024c] best fitted the frontend implementation.

4.1 Static Site Generators

Static Site Generators (SSG) take templates and data and generate a set of static web pages that can be hosted on a web server and served to a web browser [Hawksworth 2020]. The important difference between classical server-side rendering and SSGs is that an SSG-generated page is ready to serve immediately upon a client's request, without any further intermediate processing. All of a site's web pages are already pre-rendered on the server, waiting to be served to a client. SSGs have many advantages over the traditional, on-demand approach for a web site stack, including, but not limited to, better security, better scalability, and better performance.

4.1.1 Security

An SSG generates static assets, which are ready to be served to clients. This also implies, that a server does not need any advanced business logic implemented to fulfill this task, since no database accesses or complex business logic are executed on the server. This trivializes the work of the server and minimizes the threat vector a potential attack has.

4.1.2 Scalability

Since page generation has been shifted to build time rather than request time, it is very easy and fast to deliver the pre-rendered pages to the client. There is no need to wait for expensive computations or database requests when the client accesses the web site, which means handling heavy loads of requests is also less computationally stressful for the server. Content delivery networks (CDNs) can be used to deliver static sites. With completely pre-rendered pages, it is possible to cache each page in the CDN, which makes this a scalable architecture by design.

4.1.3 Performance

The time to service a request is impacted by multiple factors. Among those factors are the amount of systems it has to interact with, as well as the amount of work done by these systems. With SSGs, those factors become irrelevant, because it already happened at build time. Any interaction with the underlying systems is completely avoided by simply delivering a pre-rendered view. Another important factor to consider is the reduced dependency on the underlying infrastructure. Spikes in requests will not have a major impact on the server load at all, since nothing is computed at request time.

4.1.4 Drawbacks

Of course, it cannot all be perfect. SSGs also have a number of drawbacks. The most obvious is that static web sites are not going to be very dynamic, which means they often have less functionality [Petersen 2016]. For example, displaying custom user data is not possible, since the pages are pre-rendered and cannot be dynamically populated with user data. Furthermore, SSGs are a relatively new trend and the tooling is not yet fully mature. Over the past few years, multiple different SSGs were very popular with the community, but it is possible that they will be replaced by a new trending SSG shortly.

4.2 Astro

According to Netlify [2023], Astro was the fastest-rising framework for web development in both usage and user satisfaction in the year 2023. The comparison includes more than 20 different frameworks, which were observed for 12 months. During this period Astro almost doubled its usage.

Astro is an SSG with some support for server-side rendering (SSR) [Astro 2024c]. Astro's recent popularity stems from multiple factors. The most important factor is that Astro is very easy to use. They provide a great CLI, good templates, and very good documentation for both beginners and advanced developers. Furthermore, Astro has a large and very helpful community of over 10.000 active developers on the community Discord server. Last, but not least, Astro is "UI-agnostic", which is touted as "Bring Your Own UI Framework" by the Astro team. This feature makes Astro compatible with components from multiple different JavaScript frameworks, including React, Svelte, and Vue. Astro became popular enough for some of the largest companies worldwide to adopt it and use it. The most notable companies are Google, Microsoft, The Guardian, and Porsche.

4.2.1 Islands

One of the core concepts of Astro are so-called Islands [Astro 2024a]. The general idea of Islands is to have placeholders in a pre-rendered and static HTML page, which can be injected with highly dynamic content. This process is also called partial or selective hydration. This allows for very selective injection of JavaScript and strips out any unnecessary JavaScript, which would cause the page to slow down.

An Island can be seen as an independent widget, dynamically inserted into the otherwise static HTML. A page can have multiple Islands, but the Islands run in isolation from each other. Even though they run in different contexts, Islands can still share a state and communicate with each other.

An example page is sketched out in Figure 4.1, where a mostly static page has two Islands, which are injected dynamically. The isolated hydration of the islands also provides boosts in performance, since they are loaded independently and do not need to wait on each other. The dynamic sidebar component does not need to wait for the low-priority image gallery to load. Astro also provides so-called client directives, which can be used to define when exactly an Island should be loaded. The combination of Islands and Astro's UI-agnostic approach makes it possible to mix components from multiple different frameworks on a single page, which makes Astro very flexible and adaptable.

4.2.2 Routing

In comparison to other frameworks, like Angular [Google 2024] where routes are defined manually by the developer, routes in Astro are generated automatically [Astro 2024b]. Astro distinguishes between static and dynamic routes, where the former refers to page components defined in the page directory within the source code, while the latter refers to multiple similar pages created based on a provided template and filled dynamically at build time.

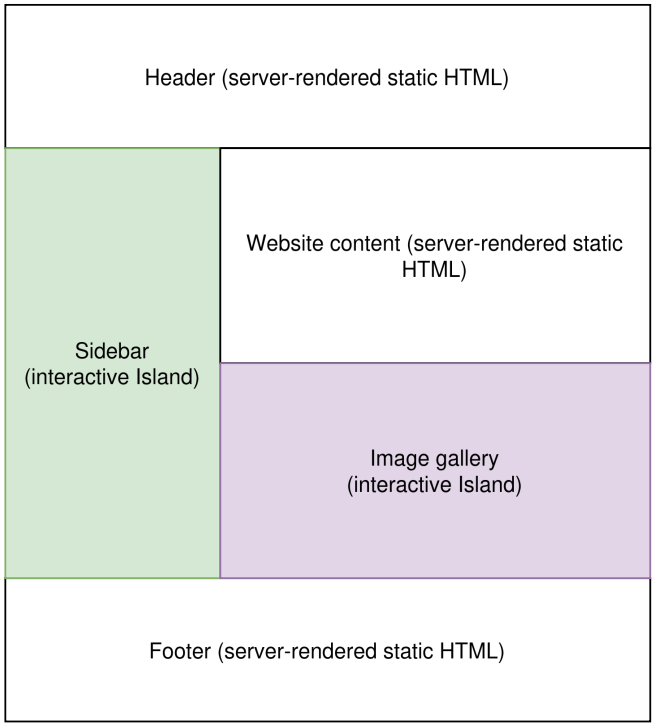

Figure 4.1: An Astro web site having mostly static content, but also two dynamically loaded Islands.

Chapter 5

Results and Recommendations

This survey project looked at using two different Headless CMSs in a Jamstack approach to build a conference web site.

5.1 Choice of Headless CMS

Strapi and WordPress both offer advantages depending on the use case. Strapi is a modern, developer-focused CMS that simplified our workflows with an intuitive interface. It also has no need for plugins. WordPress, by contrast, is widely recognized for its large community and extensive plugin ecosystem. While it requires plugins for advanced customization and frontend deactivation, it remains a practical option for projects. The choice between the two Content Management Systems depends on the specific project requirements and the user's level of expertise.

5.2 Choice of API

REST and GraphQL also both offer distinct advantages depending on the project's needs. We found REST to be straightforward and well-suited for smaller projects, since it is built into most systems and requires minimal setup. However, it can become less efficient and scalable as the complexity and volume of data grow. GraphQL, on the other hand, excelled in handling large amounts of data by minimizing fetching overhead and allowing clients to request only the data they need. While it provides significant performance benefits for larger projects, it can be unnecessarily complex for simpler use cases. The choice between REST and GraphQL should align with the scale and complexity of the project.

5.3 Choice of Frontend SSG

One output of the survey is an Astro web site for a conference, which utilizes the advantages of SSGs, Astro's Islands, and dynamic routing and is independent of the used API technology. Figure 5.1 shows how the web site displays the venues of the conference on a pre-rendered, static page. Every venue is clickable and will open the page of the respective venue. Astro's dynamic routing is used here, which means there is no need to change anything in the source code if new venues are added to the CMS.

Not everything can be fully static on a modern web site and sometimes client-side JavaScript is necessary. The conference schedule should be easy to use and not overwhelming. This can be a tricky task if a conference has multiple days and tracks. Therefore, we decided to utilize accordion and tab components to properly structure the conference schedule view. Figure 5.2 shows the structure of our conference schedule. Since accordions and tabs are components, which expect user interaction in the form of clicks, some client-side JavaScript was needed to make it work. That is why the accordion containing the schedule information is a dynamic Island written in React. While everything around the accordion is statically generated by Astro, the schedule accordion is dynamically hydrated.

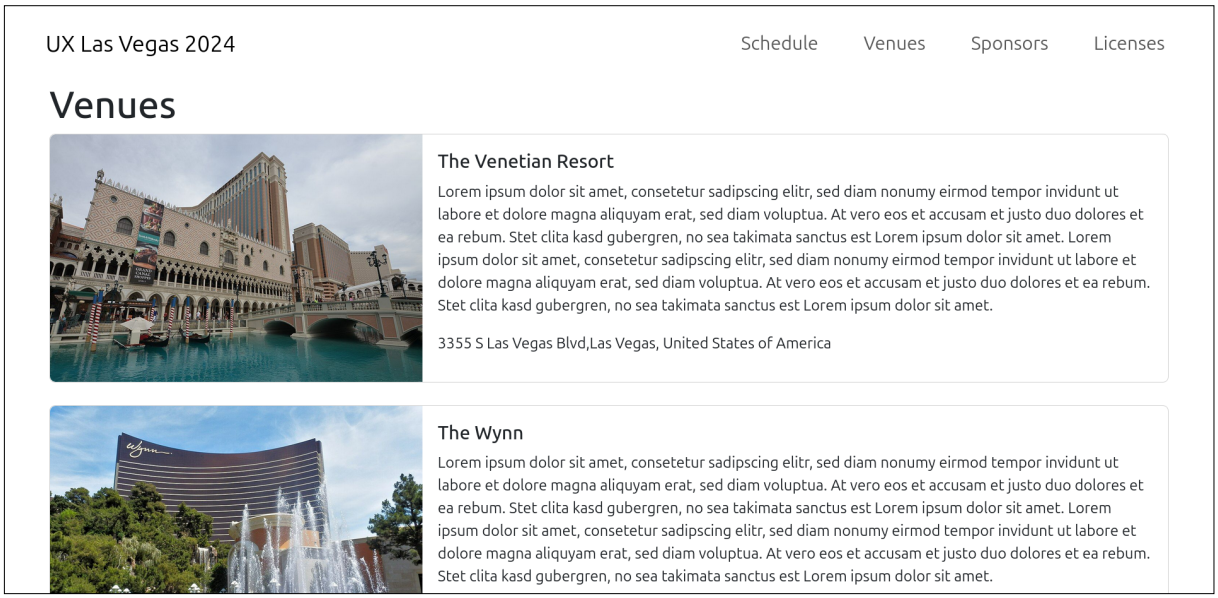

Figure 5.1: Web page displaying the venues of the conference, with dynamic routes in the background.

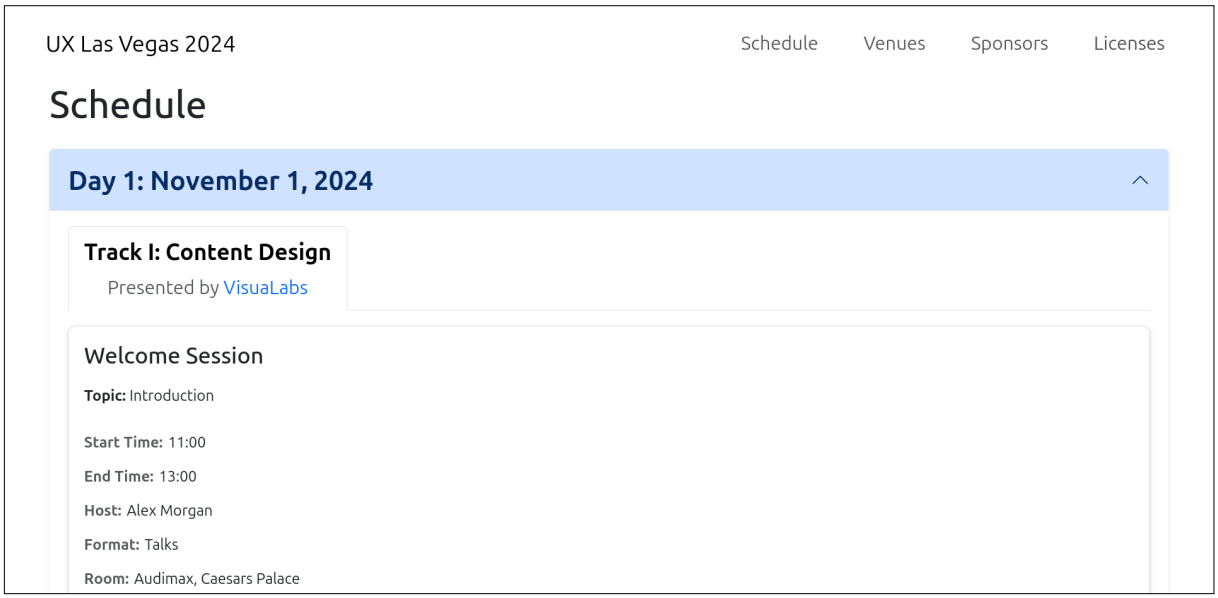

Figure 5.2: Conference schedule page with embedded Island containing the actual schedule

5.4 Conclusion

In conclusion, for the use case of creating a conference web site, we would choose Strapi as the CMS, REST for the API, and Astro as the Static Site Generator. Strapi impressed us with its usability and the automatic generation of API endpoints. While GraphQL minimizes overfetching and underfetching, we would choose REST for its simplicity, which aligns well with the scale and requirements of the project. Astro stood out for its usability and performance, leveraging its island architecture and support for server-side rendering to deliver an optimized user experience. This stack ensures a practical, efficient, and high-performing solution for the conference web site.

Bibliography

Astro [2024a]. *Astro Islands Architecture*. 21 Nov 2024. https://docs.astro.build/en/concepts/islands/ (cited on page 14).

Astro [2024b]. *Astro Routing*. 22 Nov 2024. https://docs.astro.build/en/guides/routing/ (cited on page 14).

Astro [2024c]. *Astro: The Web Framework for Content-Driven Websites*. 21 Nov 2024. https://astro.build/ (cited on pages 13–14).

Fielding, Roy T and Richard N Taylor [2002]. *Principled Design of the Modern Web Architecture*. ACM Transactions on Internet Technology (TOIT) 2.2 (May 2002), pages 115–150. doi:10.1145/514183.514185. https://ics.uci.edu/~taylor/documents/2002-REST-TOIT.pdf (cited on page 9).

Google [2024]. *Angular Routing*. 22 Nov 2024. https://angular.dev/guide/routing (cited on page 14).

GraphQL [2024]. *GraphQL*. GraphQL Foundation, 28 Nov 2024. https://graphql.org/ (cited on page 9).

Hawksworth, Phil [2020]. *What is a Static Site Generator?* 13 Apr 2020. https://netlify.com/blog/2020/04/14/what-is-a-static-site-generator-and-3-ways-to-find-the-best-one/ (cited on page 13).

Netlify [2023]. *The State of Web Development*. 2023. https://netlify.com/resources/ebooks/the-state-of-web-development-2023/ (cited on pages 13–14).

Netlify [2024]. *Jamstack*. 21 Nov 2024. https://jamstack.org/ (cited on page 13).

Petersen, Hillar [2016]. *From Static and Dynamic Websites to Static Site Generators*. Bachelor's Thesis. University of Tartu, Estonia, 12 Aug 2016. 32 pages. https://core.ac.uk/download/83597655.pdf (cited on page 14).

Strapi [2024]. *Strapi*. 21 Nov 2024. https://strapi.io/ (cited on page 4).

Wordpress [2024a]. *Advanced Custom Fields*. 01 Dec 2024. https://wordpress.org/plugins/advanced-custom-fields/ (cited on page 4).

Wordpress [2024b]. *Wordpress*. 01 Dec 2024. https://wordpress.org/ (cited on page 3).