5 | Implementation

Before starting, it is important to note that the implementation chapter was performed on a Linux Mint 22 Cinnamon operating system. Nevertheless, they will work on Windows 10/11, macOS and any other major distribution of Linux, with minor setup differences.

5.1.1 Node.js and npm Setup

In order to work with ApostropheCMS (Apos) or any Node.js application, it is required to install the latest version of Node.js and NPM.

This can be done by directly downloading it from the official Node.js website (Node.js Contributors, 2024), or, the recommended way, using a version manager such as Node Version Manager (NVM) (NVM Maintainers, 2024). That way, different versions of Node.js can be installed and it is possible to switch around those version without issues or reinstallation. That comes in handy when working with multiple projects that require different versions of Node.js, which would also be the case for WP, where an update of PHP would break the website if the PHP version is not compatible with the WP version. It is also comparable to SDKMAN, a popular version manager used for Java and other JVM based languages (like groovy) (SDKMAN Maintainers, 2024). Should issues occur during the installation of Node.js, developers can refer to community documentation and troubleshooting guides such as the DigitalOcean tutorial for Ubuntu systems (DigitalOcean Community, 2024) or the Radixweb guide for Windows and macOS (Radixweb Contributors, 2024).

After installation, verify if it was done corrctly with `npm --version` and `node --version` in the commandline or terminal. It is important to note that the versions can be different, which is fine.

```
$ npm --version
10.9.2

$ node --version
v22.14.0
```

5.1.2 MongoDB Setup

MongoDB is a modern NoSQL (NoSQL) database system designed to store and manage data in a document format, using JSON structures, rather than the traditional table based layout of relational databases. This makes it highly flexible and scalable for use in web applications, especially those that require changing or hierarchical data models, for example CMS systems.

Unlike SQL databases, that need an already defined schema, MongoDB allows each record (or document) to have a unique structure. This reduces setup overhead and makes it easier to store complex or nested data, such as user profiles, product metadata or page content.

It is not inferior or useless in comparison relation databases, but rather a quite good alternative, as described by Rathore and Bagui: "MongoDB stands as a pioneering NoSQL database management system, offering a modern alternative to traditional relational databases…." (Rathore and Bagui, 2024)

In the context of this thesis, MongoDB plays a central role in Apos, as described in Apos Section 4.2.2, which uses it as its primary database. Understanding how its differences from relational systems helps to make clear, why certain CMS platforms are more suited for flexible, modern web architectures. For readers seeking a more detailed understanding of the pro's and con's of MongoDB compared to relational databases like MySQL, the work of Deari et al. (Deari, Zenuni, Ajdari, *et al.*, 2018) provides a comprehensive comparison of document-based and relational database systems, including CRUD performance, data modeling and storage principles.

MongoDB needs to be set up in the local development environment. Installation is straightforward because there is extensive documentation, tutorials, guides and numerous other sources available for most platforms (MongoDB Documentation Team, 2024a). Similar to relational databases such as MySQL, MariaDB, or PostgreSQL, MongoDB can also be installed using Docker (MongoDB Documentation Team, 2024b). However, for non technical users, that would go beyond the scope of this what is aimed, therefore, it will be done with the regular installation.

In case of a linux and macOS machine, one can verify the installation by running `status mongod`. If the mongod service is not in status **running**, just make sure to start it with `start mongod` and check the status again.

If any issues occur during the MongoDB installation process, there are many helpful tutorials and community written guides available online. Resources such as Prisma's MongoDB setup guide (Prisma Data Guide, 2024) and beginner friendly articles like the one by Nikhil Sidana (Sidana, 2020) offer clear step by step instructions for troubleshooting and setup.

5.1.3 Integrated Development Environment (IDE) Setup

The choice of an Integrated Development Environment (IDE) is not critical for this project, or not even mandatory. However, for readers that want to know what was used, Visual Studio Code (VSCode) was selected due to its ease of setup and wide support within the JavaScript and frontend development community. Installation is straightforward and there is extensive official documentation available to guide the process (Microsoft Docs, 2024).

To work effectively with the frontend stack, the Astro (Astro) extension should be installed in VSCode (Astro Maintainers, 2024). This plugin ensures proper syntax highlighting, error detection and support for `.astro` files, which are part of the frontend framework used in this project.

5.1.4 Git Setup

Before continuing with the setup of the Apos Command Line Interface (CLI), Git should be installed on the local machine (Git Contributors, 2024). Git is a widely adopted version control system that enables

developers to track changes, manage code collaboratively and support deployment workflows. Git is essential for Apos because the Apollo template is hosted on GitHub. The Apos CLI also relies on Git for version control.

Once installed, it is recommended to configure the global username and email. These settings are attached to all commits made on the system and help ensure correct attribution within version history:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

5.1.5 ApostropheCMS Command Line Interface Setup

The creation and management of Apos projects, the Apos CLI should be installed on the system (ApostropheCMS Team, 2024). This tool simplifies various development tasks, including:

- **Project Creation**: Quickly start new projects using starter kits, such as the Marketing Starter Kit, to speed up development (ApostropheCMS Community, 2024).
- **Module Generation**: Automate the creation of custom modules, including pieces and widgets, reducing manual setup (ApostropheCMS Team, 2024).
- **Configuration Management**: Simplify project configuration and setup processes, which boosts consistency across development environments.

Using the Apos CLI, developers can significantly reduce setup time and adhere to the project structure and best practices.

5.2 Setup of ApostropheCMS and first startup

To simplify the initial development process, Apos provides a range of starter kits that include a preconfigured set of OOTB widgets, layout templates and styling options (ApostropheCMS Contributors, 2024k). These kits are designed to help developers quickly start projects for various use cases such as marketing, portfolios or landing pages, as shown in Figure 4.

While some starter kits are behind a paywall, many are open source and freely available through community contributions. However, it is important to note that most of these starter kits are configured for traditional (non headless) setups. An exception is the official Astro Integration Starter Kit, which is designed for use with a decoupled frontend architecture (ApostropheCMS Contributors, 2024l).

However, these starter kits will not be used in this instance, as the goal is to do the least amount of work to setup the project.

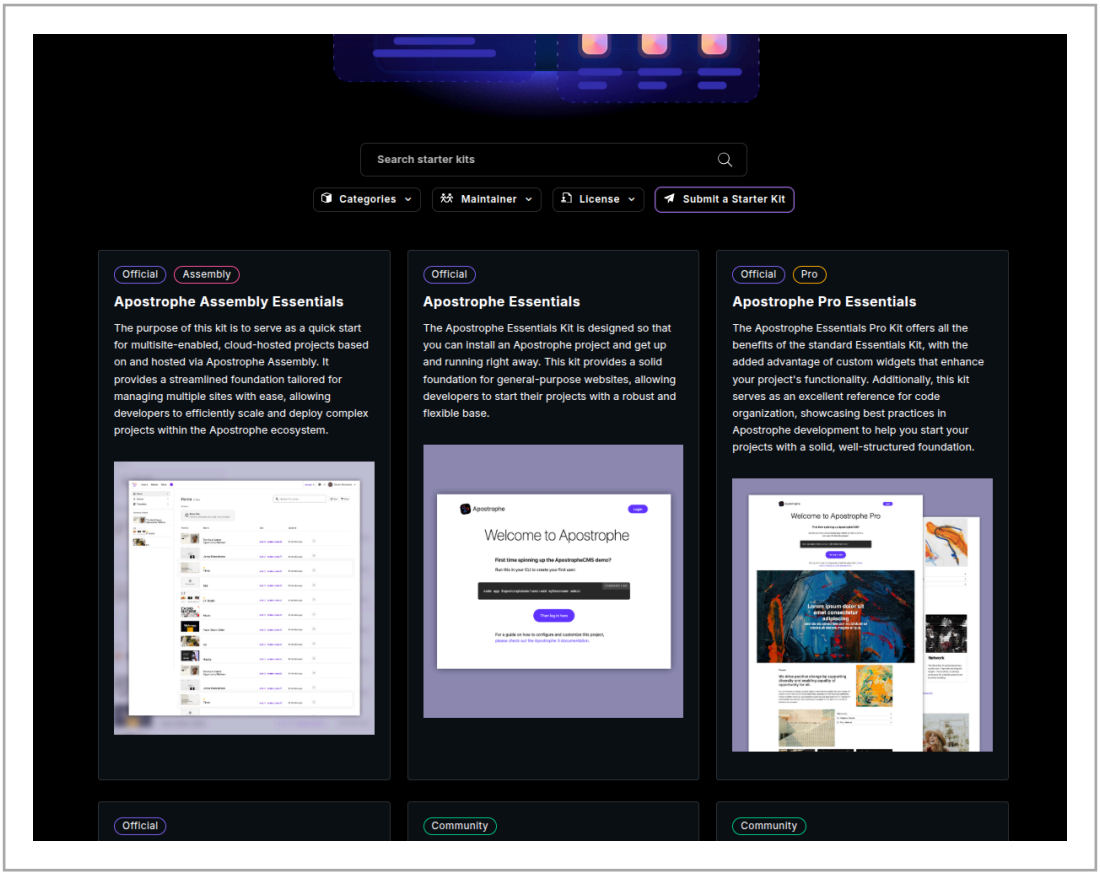

Figure 4: ApostropheCMS Starter Kits

5.2.1 Apollo Template Starter Kit

While the Astro Starter Kit is a good starting point for headless projects, it only includes the frontend portion and still needs additional setup to connect it properly to an Apos backend, which also has to be checked out separately. To reduce setup complexity, Apos provides the **Apollo template** (ApostropheCMS Team, 2024). Apollo includes both the frontend and backend in a single, unified repository, making sure they are configured to work together without compatibility issues.

This template comes with a variety of OOTB features, including widgets, layouts and pages. It also includes an initializer script, as an NPM task, that generates sample data, such as images and content pages, so developers can begin working immediately without building everything from scratch.

Complete setup instructions are available in the official documentation (ApostropheCMS Documentation Team, 2024).

To begin the local setup of the Apollo template, create a new folder for the project, for example `bikesite`, preferably within your user's home directory (e.g., `/Documents`) (ApostropheCMS Documentation Team, 2024). Inside this folder, open a terminal window and run the following command to clone the official Apollo repository:

```
git clone https://github.com/apostrophecms/apollo
```

Next, navigate into the cloned project folder:

```
cd apollo
```

Then install all required dependencies for both the frontend and backend using:

```
npm install
```

This step installs the NPM packages and configures both project layers automatically.

Once installation is complete, initialize the project with sample data using the following command:

```
npm run load-starter-content
```

This will populate the backend with example pages, widgets and images. During this process, the CLI will prompt the user to set a password for the initial admin account.

A shared environment variable must now be configured to establish secure communication between the Apos backend and the Astro frontend. While still inside the backend folder, run the following command:

```
export APOS_EXTERNAL_FRONT_KEY=your-secret-key-here
```

Repeat the same export in a separate terminal window from within the frontend folder.

With configuration complete, start both environments in development mode:

```
# In one terminal
npm run dev  # backend

# In a second terminal
npm run dev  # frontend
```

The application should now be running locally and accessible at the following URLs:
- Backend: http://localhost:3000
- Frontend: http://localhost:4321

Unlike some setups that require manual MongoDB configuration, the Apollo template uses a default MongoDB connection string compatible with local installations. If MongoDB is installed using default settings, no further configuration is needed to connect it to the Apos backend.

Opening the frontend URL should display the default Apollo landing page, as shown in Figure Figure 5.

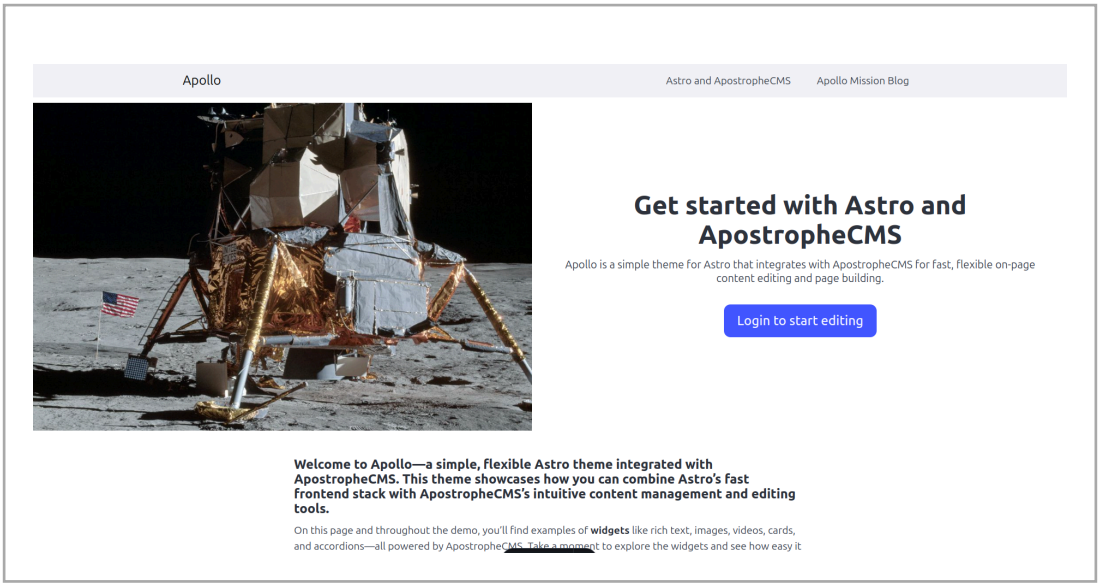

Figure 5: ApostropheCMS Apollo Template Homepage

To modify content or add new pages, log into the Apos backoffice using the frontend login route:
- Login URL: http://localhost:4321/login
- Credentials: Use the admin account configured during initialization

The backoffice interface is shown in Figure Figure 6.

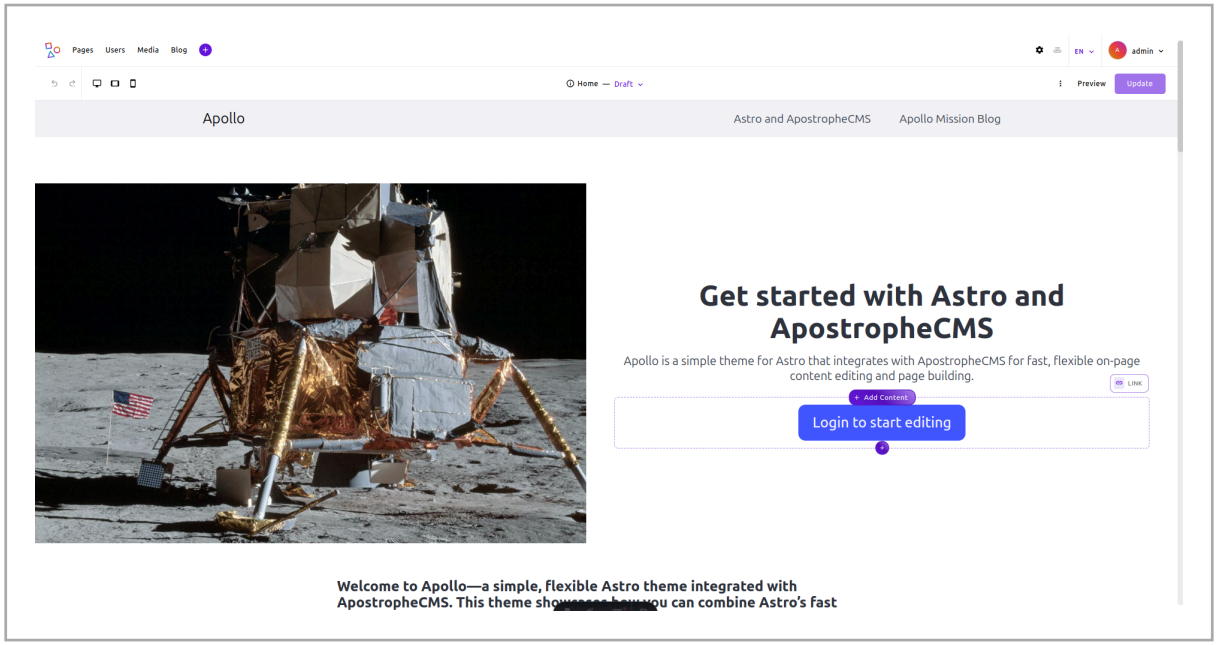

Figure 6: ApostropheCMS Apollo Template Backoffice

For the next steps, the backend project will be referred to as **backend** or Apos and the frontend project as **frontend** or Astro.

5.3 Set main language (i18n)

Apos includes OOTB support for I18N, enabling multilingual content management. In the case of the Apollo template, the default configuration includes four already configured locales: English, French, Spanish and German, with English set as the primary language. This default setup is illustrated in Figure Figure 7.

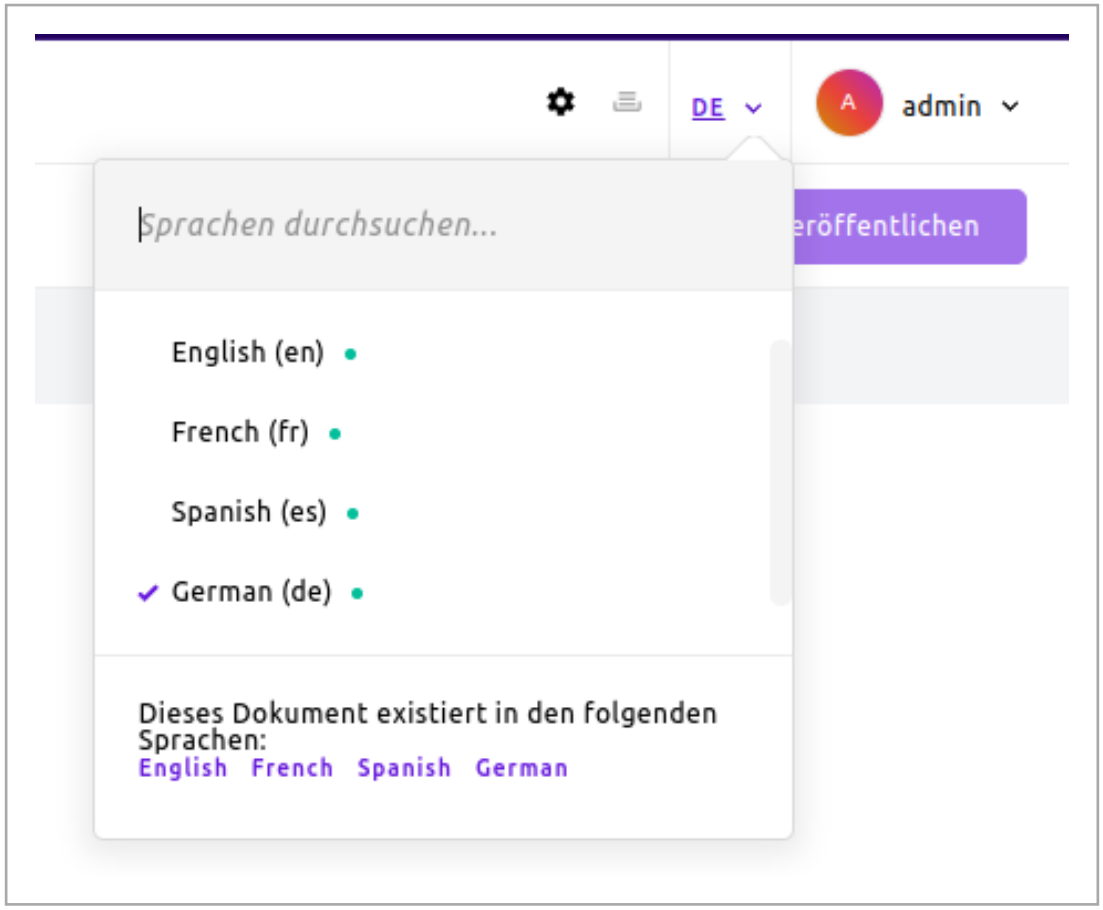

Figure 7: Apollo Template Language Selector (Default)

As outlined in the project requirements, the target audience for this website is primarily German speaking SME customers. To reflect this, it is necessary to set German as the default language and remove the other predefined locales, as they are not maintained, so no need to have them.

Unfortunately, this change is not configurable through the admin UI and must be made directly in the Apos code. Specifically, the file located at `/modules/apostrophecms/i18n/index.js` must be adapted. The original configuration looks as in Listing 3.

```
export default {
    options: {
      locales: {
        en: { label: 'English' },
        fr: { label: 'French', prefix: '/fr' },
        es: { label: 'Spanish', prefix: '/es' },
        de: { label: 'German', prefix: '/de' }
      },
      adminLocales: [
        { label: 'English', value: 'en' },
        { label: 'Spanish', value: 'es' },
        { label: 'French', value: 'fr' },
        { label: 'German', value: 'de' }
      ]
    }
};
```

Listing 3: I18N Module configuration before

This should be replaced with the following simplified version to enable only German as the default and only locale as shown in Listing 4.

```
export default {
    options: {
      locales: {
        de: { label: 'German' }
      },
      adminLocales: [
        { label: 'German', value: 'de' }
      ]
    }
};
```

Listing 4: I18N Module configuration after

After restart, the language selector is completely gone in the backoffice and only german translation remains as shown in Figure 8.

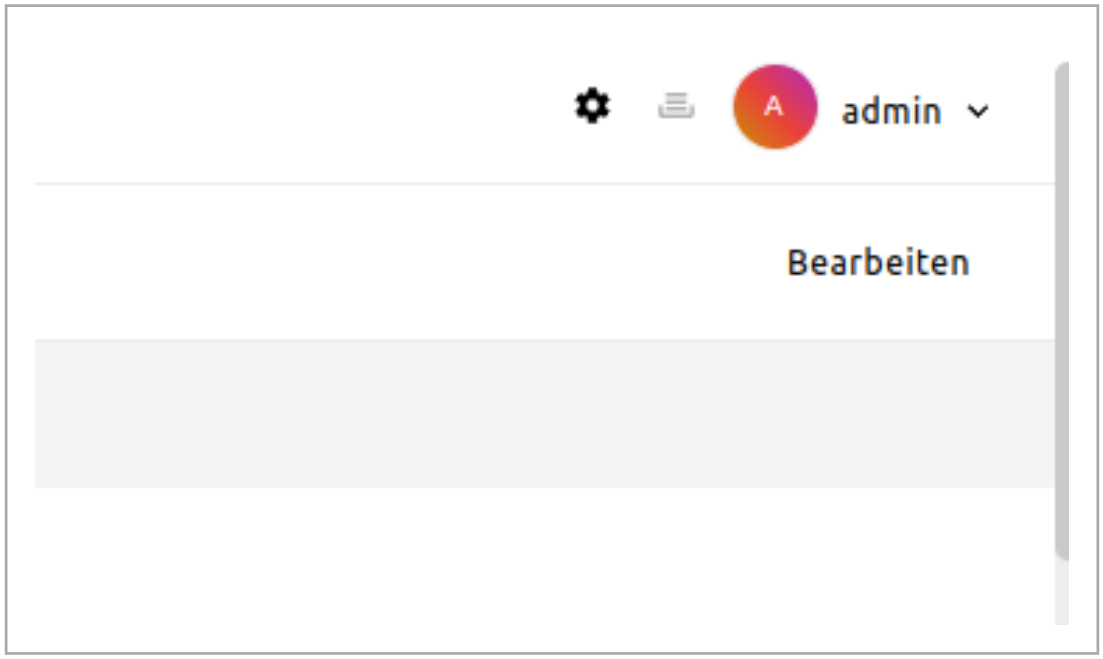

Figure 8: Language Selector Removed

5.4 Creation of pages

Page creation in Apos is visual and managed directly through the platform's backoffice. Editors can create new pages and structure them hierarchically using a drag and drop mechanism. This visual tree structure allows nesting of pages within each other, enabling the creation of site architectures that can reflect menu hierarchies or content groupings (sub menu) (ApostropheCMS Team, 2024).

It is also possible to decouple page titles from their URLs. Each page can be assigned a unique slug (URL path), independent of the visible title shown in navigation or content. For instance, a page titled "Our Services" could have a URL like `/offerings`, which allows for improved readability or SEO optimization (ApostropheCMS Team, 2024).

Pages are created using page types, which can be configured in the backend to determine the structure and available widgets on each page. For in depth guidance, the official Apos documentation provides multiple resources on how to set up and manage pages, including how to configure new page types, adjust slugs, manage visibility and use the page tree interface efficiently (ApostropheCMS Team, 2024).

5.5 Adapt Styling of Template

Unlike WP, Apos does not provide a centralized marketplace for installing themes. However, frontend styling can still be customized by using modern CSS frameworks such as **Bulma**, **Bootstrap**, or **TailwindCSS**.

The Apollo template is already shipped with **Bulma**, tt is used for layout structuring, spacing and responsive design. This makes it possible to adapt key elements like the header, navigation bar and footer while maintaining a consistent design system across all pages. Furthermore, by using established CSS frameworks, developers can take advantage of existing templates and components. This significantly reduces the time and effort required during the design phase (Jeremy Thomas and Contributors, 2024). While these templates may not offer the same level as a full WP themes, they still provide a good starting point, eliminating the need to build every layout from scratch.

5.5.1 Adapt Header Structure and Styling

Apos allows shared components such as headers or footers to be managed through its `global` module. This approach keeps the structure and styling of such components consistent across all pages, while still allowing the content (e.g., text or images) to be updated dynamically via the admin interface.

The new header design consists of a single row with two columns:

- The first column spans **80%** of the width and is positioned on the **left side** (for the text-based subline).
- The second column occupies the remaining **20%**, used to display a logo or badge image aligned to the right.

As shown in Figure Figure 9, the header fields are defined in the backend under `/modules/apostrophecms/global/index.js`. To avoid breaking the existing configuration, two new fields are added within the `headerGroup` configuration, as shown in Listing 5.

```
1
2 headerSubline: {
3     type: 'area',
4     label: 'Header Subline',
5     options: {
6       widgets: {
7         '@apostrophecms/html': {}
8       }
9     }
10  },
11  headerBadge: {
12    type: 'area',
13    label: 'Header Badge',
14    options: {
15      widgets: {
16        '@apostrophecms/image': {}
17      }
18    }
19  }
```

Listing 5: Add header fields to backoffice

The headerSubline field allows editors to insert rich-text HTML, while headerBadge enables image uploads such as logos.

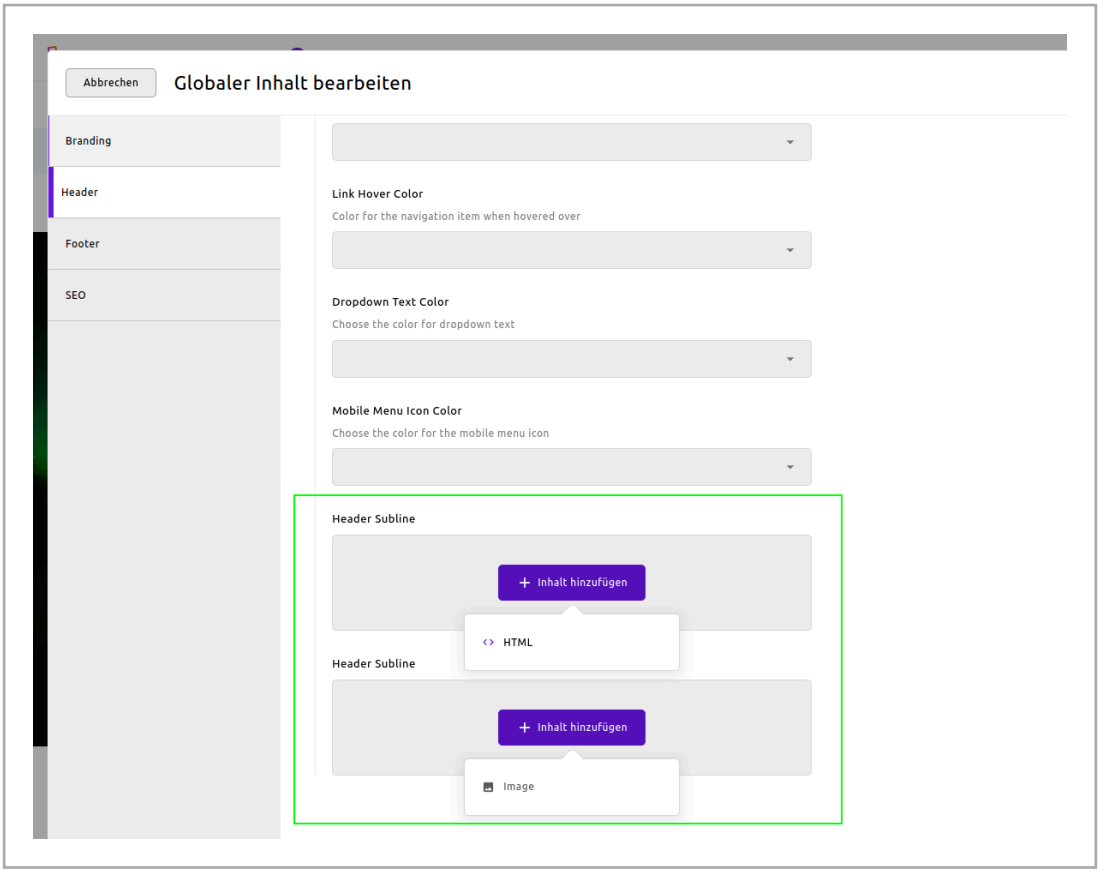

Figure 9: New Header Fields in ApostropheCMS

To display these dynamic fields in the frontend, modifications are made to the Astro component located at `/src/components/Header.astro`.

- Remove the existing header code (lines 46–112) to clear space for the custom layout.
- Add the Listing 6 code after line 4 to access the global content:

```
1 const headerGroup = aposData.global?.headerGroup;
2 const sublineWidget = headerGroup?.headerSubline?.items?.[0];
3 const badgeWidget = headerGroup?.headerBadge?.items?.[0];
```

Listing 6: Header.astro global properties

- Add this JSX code from Listing 7 below the three dashes `---`, which is the part where the JavaScript stops and JSX code starts in an .astro file, to render the new header section on every page:

```
{(sublineWidget || badgeWidget) && (
  <div class="header-bar px-4 py-2">
    <div class="columns is-multiline is-vcentered is-variable is-1">
      <div class="column is-12-mobile is-11-tablet">
        {sublineWidget && (
          <div class="html-widget" set:html={sublineWidget.code} />
        )}
      </div>

      <div class="column is-12-mobile is-1-tablet has-text-centered">
        {badgeWidget && (
          <img src={`${baseUrl}${badgeWidget._image[0].attachment._urls.original}`} alt="Header
badge" />
        )}
      </div>
    </div>
  </div>
)}
```

Listing 7: Header.astro content code

This logic makes sure that the header is only displayed if one or both fields are populated. The layout uses Bulma's grid system to guarantee responsiveness and alignment.

Replace the contents of the tag with the styling from Listing 8 to apply a visually clear style:

```
.header-bar {
  background-color: white;
  border-bottom: 1px solid #ddd;
  padding: 1rem 1.5rem;

  .site-branding {
    h1, h2, h3, p {
      margin: 0;
      font-weight: bold;
      font-size: 1.25rem;
    }
  }

  img {
    display: inline-block;
    max-height: 60px;
  }

  @media (max-width: 768px) {
    .column.is-2,
    .column.is-1 {
      text-align: center;
      margin-top: 0.5rem;
    }
  }
}
```

Listing 8: Header.astro styling

Once the new header structure is implemented and the styles are in place, the dynamic content can be maintained directly within the Apos backoffice. As shown in Figure Figure 10, the editor can populate the **Header Subline** with rich HTML content (e.g., a promotional message or welcome text) and upload a **Header Badge** image (e.g., a logo or certification mark). These fields are accessible under the **Global Settings** section and support live content editing without requiring code changes.

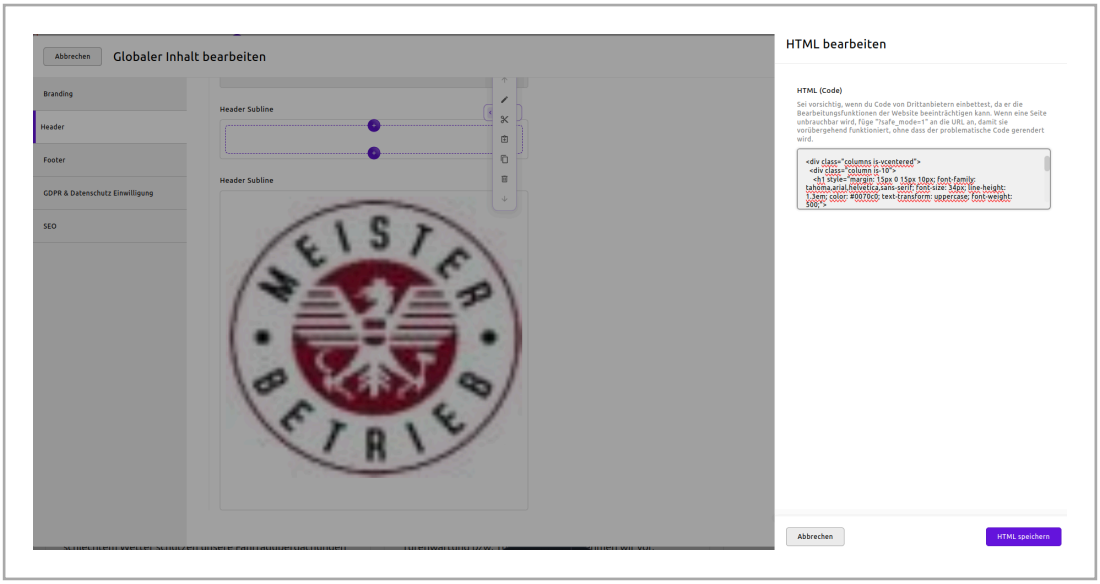

Figure 10: Maintain Header fields in Backoffice

After saving the changes in the admin panel, refreshing the frontend site reflects the updated content on all pages. As shown in Figure Figure 11, the header now displays the dynamically maintained subline on the left and the badge image on the right, displayed exactly to the layout and styling configured in the Astro component.

Figure 11: Customized Header in Frontend

5.5.2 Adapt navigation

The Apollo template already includes a functional navigation menu OOTB. However, to improve structure, maintainability and styling consistency, the navigation logic can be moved into its own reusable Astro component and styled using custom SCSS. This approach also makes the menu more adaptable to future changes in content or layout.

To modularize the navigation, create a new file named `Navigation.astro` inside the `/src/components/` directory. Paste the following .astro content of Listing 9 into the file.

```
---
const { navItems, homeTitle, homeUrl, currentUrl } = Astro.props;
import "../styles/navigation.scss";
---
<nav class="navbar has-shadow is-white" role="navigation" aria-label="main navigation">
  <div class="container">
    <div class="navbar-brand">
      <a class={`navbar-item has-text-weight-bold ${currentUrl === "/" ? "is-active" : ""}`}
href={homeUrl}>
        {homeTitle}
      </a>
      <a role="button" class="navbar-burger" aria-label="menu" aria-expanded="false" data-
target="mainNavbar">
        <span aria-hidden="true"></span>
        <span aria-hidden="true"></span>
        <span aria-hidden="true"></span>
        <span aria-hidden="true"></span>
      </a>
    </div>
    <div id="mainNavbar" class="navbar-menu">
      <div class="navbar-start">
        {
          navItems.map((item) => {
            const isCurrentOrChildActive = item.active || item.children.some((child) =>
child.active);
            return (
              <div class={["navbar-item", item.children.length > 0 && "has-dropdown is-hoverable",
isCurrentOrChildActive && "is-active"]
                .filter(Boolean)
                .join(" ")}>
                <a href={item.url} class="navbar-link">
                  {item.title}
                </a>
                {item.children.length > 0 && (
                  <div class="navbar-dropdown">
                    {item.children.map((child) => (
                      <a href={child.url} class={`navbar-item ${child.active ? "is-active" : ""}`}
>
                        {child.title}
                      </a>
                    ))}
                  </div>
                )}
              </div>
            );
          })
        }
      </div>
    </div>
  </div>
</nav>
<script is:inline>
  document.addEventListener("DOMContentLoaded", () => {
    const burger = document.querySelector(".navbar-burger");
    const menu = document.getElementById(burger.dataset.target);
    burger.addEventListener("click", () => {
      burger.classList.toggle("is-active");
      menu.classList.toggle("is-active");
    });
  });
</script>
```

Listing 9: Navigation.astro code

This component dynamically renders navigation items based on `navItems` list, highlights the current page and supports dropdown menus for child pages. It also includes an inline script to activate the burger menu functionality on smaller screens.

To apply a good styling to the navigation, create a new SCSS file named `navigation.scss` in the `/src/styles/` directory. Copy over the styling code of Listing 10 and Listing 11 into the file.

```scss
$blue-main: #2c59d4;
$blue-hover: #5280e4;
$blue-active: #003c8b;
$text-light: whitesmoke;
$text-dark: #1f2937;
$text-darker: #111827;
$dropdown-hover-bg: #f5f5f5;

.navbar {
  position: sticky;
  top: 0;
  z-index: 999;
  padding: 0.75rem 0;
  background-color: white;
  box-shadow: 5px 0px 15px rgba(0, 0, 0, 0.25) !important;
  transition: top 0.3s ease-in-out;

  &-burger {
    background-color: $blue-main !important;

    span {
      background-color: $text-light !important;
    }
  }

  &-menu.is-active {
    background-color: $blue-main !important;
    margin-top: 5px;

    .navbar-dropdown {
      border-top-left-radius: 6px;
      border-top-right-radius: 6px;
      background-color: $blue-main !important;
    }
  }
}

.navbar-item {
  background-color: $blue-main !important;
  color: $text-light !important;
  padding: 0.5rem 1rem !important;
  margin: 0 0.25rem;
  border-radius: 6px;
  transition: background-color 0.2s ease;
  display: flex;
  align-items: center;

  &:hover {
    background-color: $blue-hover !important;
    color: white !important;
  }

  &.is-active {
    font-weight: bold;
    background-color: $blue-active !important;
    color: white !important;
  }

  &.has-dropdown .navbar-dropdown {
    display: none;
    opacity: 0;
    transition: opacity 0.2s ease;
  }
}
```

Listing 10: navigation.scss code part1

```
.navbar-item {
    &.has-dropdown:hover .navbar-dropdown {
      display: block;
      opacity: 1;
    }

    &.has-dropdown:hover {
      border-bottom-left-radius: 0 !important;
      border-bottom-right-radius: 0 !important;
    }
}

.navbar-link {
    background-color: transparent;
    color: inherit !important;
    padding: 0 !important;
    font-weight: bold;
    border-radius: 4px;

    &::after {
      display: none !important;
      content: none !important;
    }
  }

  .navbar-dropdown {
    background-color: $blue-main !important;
    padding: 0.5rem 0;
    box-shadow: 0 4px 12px rgba(0, 0, 0, 0.15);
    border-radius: 8px;
    border-top-left-radius: 0;
    border-top-right-radius: 0;
    min-width: 180px;
    z-index: 1000;
    width: inherit;
    border-top: 0 !important;

    .navbar-item {
      padding: 0.5rem 1rem;
      margin: 0;
      background: transparent;
      color: $text-dark;
      font-weight: 500;
      border-radius: 0;
      white-space: nowrap;

      &:hover {
        background-color: $dropdown-hover-bg;
        color: $text-darker;
        font-weight: 600;
      }

      &.is-active {
        background-color: $blue-active;
        color: white;
      }
    }
  }
```

Listing 11: navigation.scss code part2

These styles build upon Bulma's layout system and apply a consistent color palette, hover effects and spacing for desktop and mobile views.

To enable the new navigation structure, update the `Header.astro` file as follows:

Import the component at the top.

```
import Navigation from "./Navigation.astro";
```

Add the Navigation component code as shown in Listing 12 after the header bar but before any tag.

```
<Navigation
  navItems={navItems}
  homeTitle={homeTitle}
  homeUrl={homeUrl}
  currentUrl={currentUrl}
/>
```

Listing 12: Header Navigation widget usage

Add or update the following helper variables as shown in Listing 13.

```
const homeTitle = aposData.home?.title || "Home";
const homeUrl = aposData.home?._url || "/";
const currentUrl = aposData.page?._url || "/";
```

Listing 13: Updated Navigation helper variables

Update the formatNavItem function to properly handle child navigation as shown in Listing 14.

```
function formatNavItem(page) {
  if (!page) return null;

  const children = page._children?.map(formatNavItem).filter(Boolean) || [];
  const isCurrent = page._url === aposData.page?._url;
  const isChildActive = children.some((child) => child.active);

  return {
    title: page.title || "",
    url: page.url || "#",
    active: isCurrent || isChildActive,
    children,
  };
}
```

Listing 14: Updated formatNavItem function

Once implemented, the navigation menu dynamically displays the site's page structure with proper styling and responsive behavior. Figure Figure 12 shows the result on desktop, while Figure Figure 13 illustrates the mobile burger menu in action.

Figure 12: Custom Navigation Menu – Desktop View

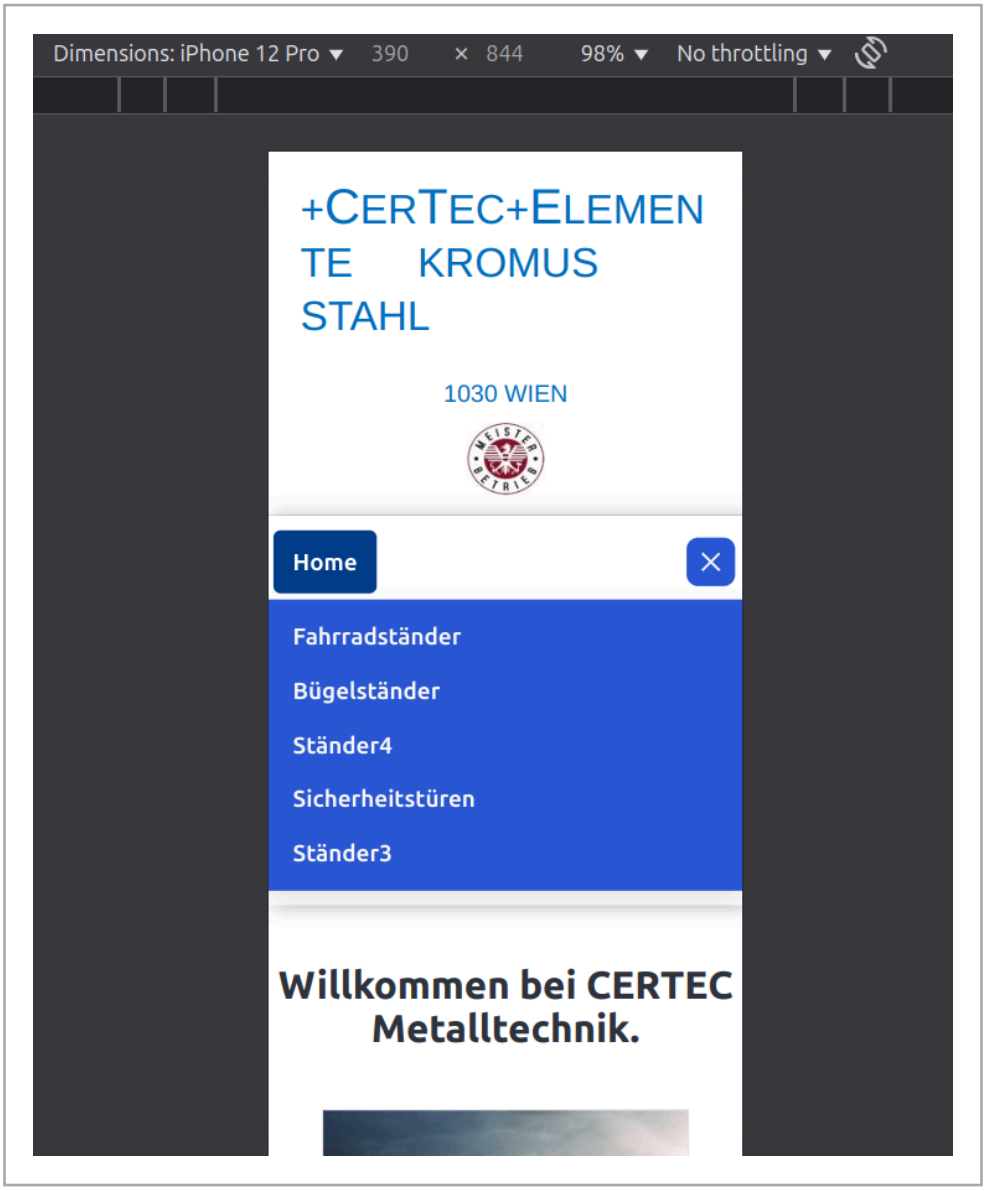

Figure 13: Custom Navigation Menu – Mobile View

5.5.3 Adapt Footer Structure and Styling

The final structural and styling adaptation required, at least for the parts that all pages share, is the footer of the website. Similar to the header, specific footer elements should be dynamically manageable through the Apos backoffice. To achieve this, a set of fields, as shown in figure Listing 15, will be added under the already present object `footerGroup` section of the global configuration module, located in Apos file `/modules/apostrophecms/global/index.js`.

```
    ........
    * All 5 fields are added here *
    ........
        footerBadge: {
          type: 'area',
          label: 'Fußzeile Bild',
          options: {
            widgets: {
              '@apostrophecms/image': {}
            }
          }
        },
        footerEmail: {
          type: 'area',
          label: 'Fußzeile Email',
          options: {
            widgets: {
              '@apostrophecms/image': {}
            }
          }
        },
        footerCompanyInfo: {
          type: 'area',
          label: 'Fußzeile Firmentext',
          options: {
            widgets: {
              '@apostrophecms/html': {}
            }
          }
        },
        footerPhoneInfo: {
          type: 'area',
          label: 'Fußzeile Telefontext',
          options: {
            widgets: {
              '@apostrophecms/html': {}
            }
          }
        },
        footerLinks: {
          type: 'area',
          label: 'Fußzeile Menü',
          options: {
            widgets: {
              '@apostrophecms/html': {}
            }
          }
        }
      }
    }
  }
```

Listing 15: Footer content configuration

The **footerGroup** configuration in the global settings includes five distinct fields, each delivering a specific content within the footer. These fields are all of type `area`, which means they support editing of widgets, allowing editors to insert different content types such as images or formatted HTML.

- **footerBadge** is used to maintain a small branding or certification image, such as a logo, badge, or partner emblem. It uses the `@apostrophecms/image` widget to allow uploading and displaying image content.
- **footerEmail** field accepts an image widget and is planned to display a graphical representation of the business's email address. This avoids exposing plain text email addresses directly in the HTML, which helps mitigate spam scraping a bit.
- **footerCompanyInfo** field uses the `@apostrophecms/html` widget and is meant to hold structured business information, such as company name, address, registration number, or a short company description. It supports rich text formatting to adapt to varying content needs.
- **footerPhoneInfo** field uses the `@apostrophecms/html` widget and is used to display contact phone numbers or support hotline details. Like the company info field, it supports basic formatting and layout control.

Finally, the **footerLinks** field provides content for rendering internal navigation or legal links, such as imprint, privacy policy, or terms of service. It uses the HTML widget as well, enabling content editors to define lists or inline links as needed.

This setup allows for flexibility in footer content management while ensuring that all key business and legal information can be maintained directly from the backoffice without any code changes.

Once these fields are saved, they will appear in the global content editor inside the Apos backoffice. Editors can populate them directly without developer input.

Unfortunately, the OOTB frontend for this modules is on backend only. To make it work on the apollo frontend project, the file `Footer.astro` is modified. At the top of the file, the footer fields are loaded into constants as shown in Listing 16.

```
const footerGroup = aposData.global?.footerGroup || {};
const footerBadge = footerGroup?.footerBadge?.items?.[0]?._image[0]?.attachment._urls.original ||
{};
const footerEmail = footerGroup?.footerEmail?.items?.[0]?._image[0]?.attachment._urls.original ||
{};
const baseUrl = import.meta.env.PUBLIC_SITE_URL || "http://localhost:3000";
const companyInfo = footerGroup?.footerCompanyInfo?.items?.[0];
const phoneInfo = footerGroup?.footerPhoneInfo?.items?.[0];
const footerLinks = footerGroup?.footerLinks?.items?.[0];
```

Listing 16: Footer get variables

Then, replace the existing HTML block in the file with the layout from Listing 17.

```
1  <footer class="footer certec-footer has-background-light" role="contentinfo">
2    <div class="container">
3      <div class="footer-content">
4        <div class="columns is-variable is-6 is-vcentered">
5          <!-- Left: Logo or Branding -->
6          <div class="column is-one-third has-text-centered-mobile">
7            {
8              companyInfo && (
9                <div class="html-widget" set:html={companyInfo.code} />
10             )
11           }
12           {
13             footerEmail &&
14             <img
15             src={`${baseUrl}${footerEmail}`}
16             alt="Footer Email"
17           />
18           }
19
20         </div>
21
22         <div class="column is-one-third has-text-centered">
23           {phoneInfo && <div class="html-widget" set:html={phoneInfo.code} />}
24         </div>
25
26         <div class="column is-one-third has-text-centered">
27           { footerBadge &&
28           <img
29             src={`${baseUrl}${footerBadge}`}
30             alt="Footer badge"
31           />
32           }
33         </div>
34       </div>
35       <div class="columns is-variable is-6 is-vcentered">
36         <div class="column has-text-centered py-0">
37           {
38             footerLinks && (
39               <div class="html-widget" set:html={footerLinks.code} />
40             )
41           }
42         </div>
43       </div>
44     </div>
45   </div>
46 </footer>
```

Listing 17: Footer content

The layout uses Bulma CSS classes for responsiveness and spacing. The content is displayed conditionally like on line 8 or 13, only if the corresponding field is not `undefined`. To apply a custom style to the footer, a minimal CSS block is added at the bottom of the same `Footer.astro` file, as shown in Listing 18.

```
<style>
  .certec-footer {
    padding: 2rem 1.5rem;
    background-color: #0070c0 !important;

    .footer-title {
      font-family: tahoma, arial, helvetica, sans-serif;
      color: white;
      text-transform: uppercase;
      margin-bottom: 0;
    }

    .footer-info {
      font-size: 1rem;
      line-height: 1.4;
      color: #333;
    }
  }

  .footer-content {
    max-width: 950px;
    margin: 0 auto;
    padding-left: 2rem;
    padding-right: 2rem;
  }
</style>
```

Listing 18: Footer styling

This makes sure consistent padding, background color and typographic styling fit with the rest of the design. After saving the changes and populating the fields in the Apos backoffice, the updated footer becomes visible for all pages. Figure 14 shows the editable fields in the backend, while Figure 15 displays the rendered frontend version of the footer.

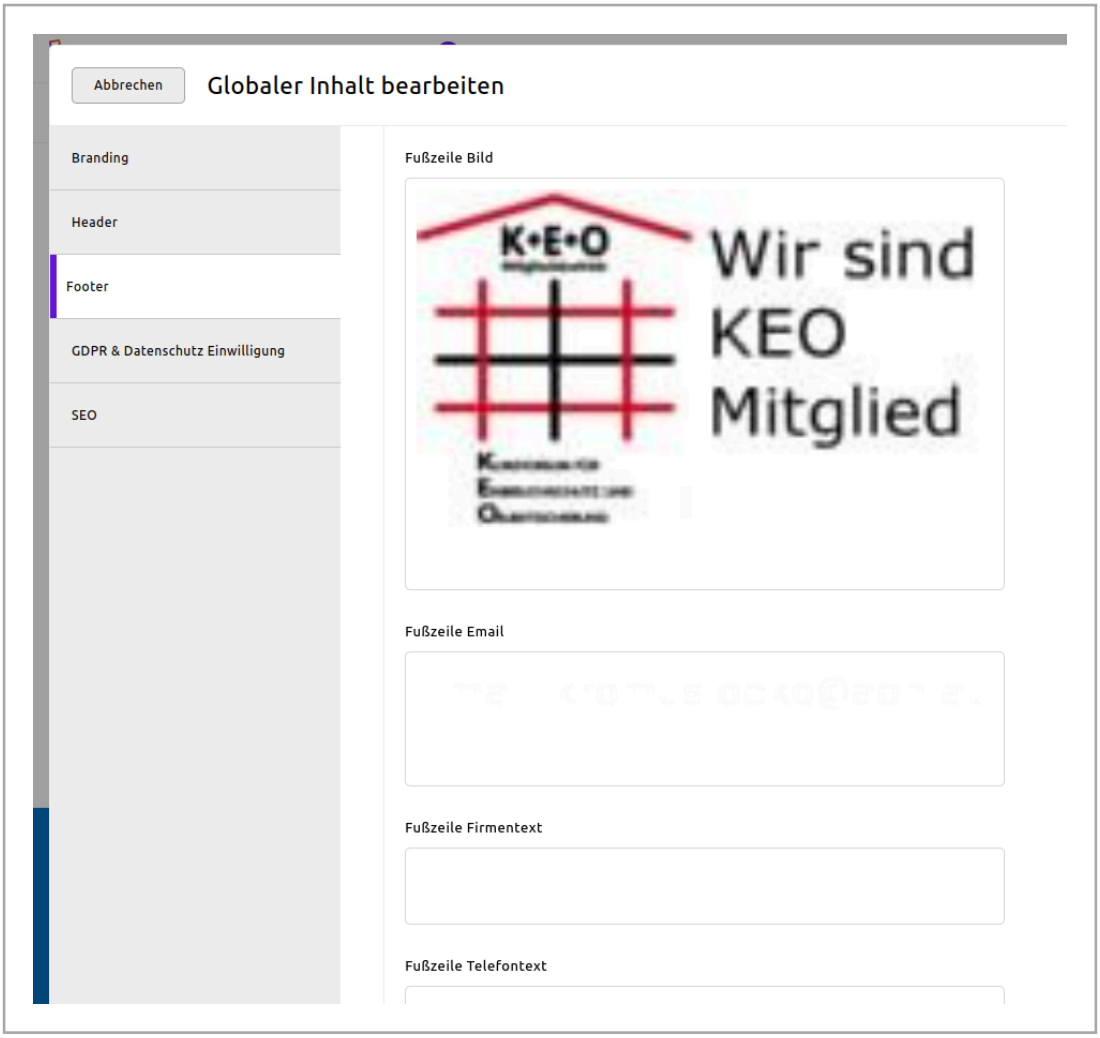

Figure 14: Maintain Footer Fields in Backoffice

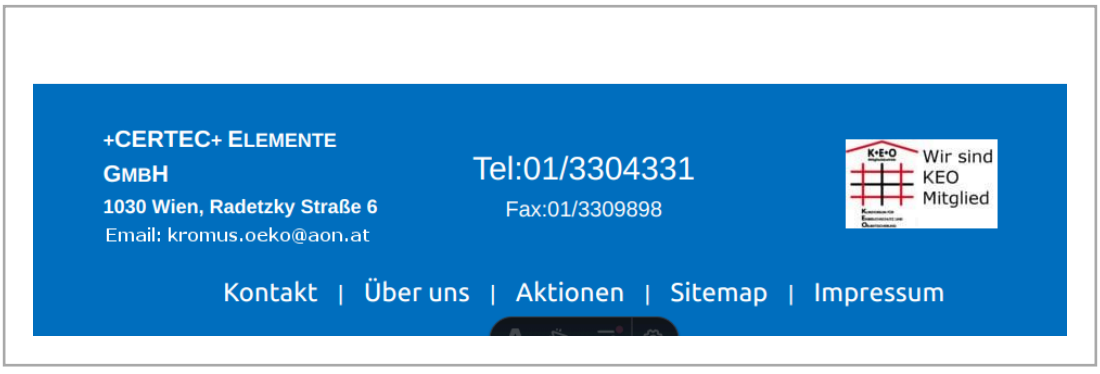

Figure 15: Updated Footer on Frontend

5.6 Cookie Consent Banner

To comply with GDPR and other privacy regulations, it is essential that the CMS includes a functionality to present a cookie consent banner to website visitors. In WP, this functionality is added through plugins, such as Cookie Notice (Hu-manity.co, 2024) or Complianz GDPR/CCPA (Complianz.io, 2024). However, Apos does not provide a OOTB cookie consent system. Not only that but there is not finished solution available in the Apos marketplace either.

To fulfill this essential requirement in Apos, a custom integration is necessary. For this implementation, the open source Osano CookieConsent JavaScript library is used (Osano, Inc., 2024). Alternative solutions like CookieConsent.com (CookieConsent.com, 2024) and others also exist, so it does not have to be this one but it is a good example.

Since cookie preferences are a global setting, a new group of fields `consentGroup` is added to the `global` module under `/modules/apostrophecms/global/index.js` as shown in Listing 19, next the to the `headerGroup`and `footerGroup` objects in Apos. This allows site administrators to manage the content of the consent banner dynamically from the backoffice.

```
consentGroup: {
    type: 'object',
    label: 'Einwilligungseinstellungen',
    fields: {
      add: {
        enableConsentBanner: {
          type: 'boolean',
          label: 'Cookie-Hinweis aktivieren',
          def: true
        },
        consentMessage: {
          type: 'string',
          label: 'Einwilligungstext',
          textarea: true,
          def: 'Diese Website verwendet Cookies, um sicherzustellen, dass Sie die bestmögliche Erfahrung machen.'
        },
        acceptLabel: {
          type: 'string',
          label: 'Button-Text: Akzeptieren',
          help: 'Dies ist der Text für den Button „Akzeptieren“.',
          def: 'Akzeptieren'
        },
        learnMoreLabel: {
          type: 'string',
          label: 'Button-Text: Mehr erfahren',
          help: 'Dies ist der Text für den Link zu weiteren Informationen.',
          def: 'Mehr erfahren'
        },
        learnMoreUrl: {
          type: 'url',
          label: 'URL für weitere Informationen',
          help: 'Die URL zu Ihrer Datenschutzrichtlinie oder weiteren Informationen.',
          def: '/datenschutz'
        }
      }
    }
  }
```

Listing 19: Cookie consent configuration for backoffice

This group of settings becomes a separate tab in the global settings editor **GDPR & Datenschutz Einwilligung**. As shown in Figure Figure 16, users can enable or disable the banner, define its message, labels and target URL without modifying any code.

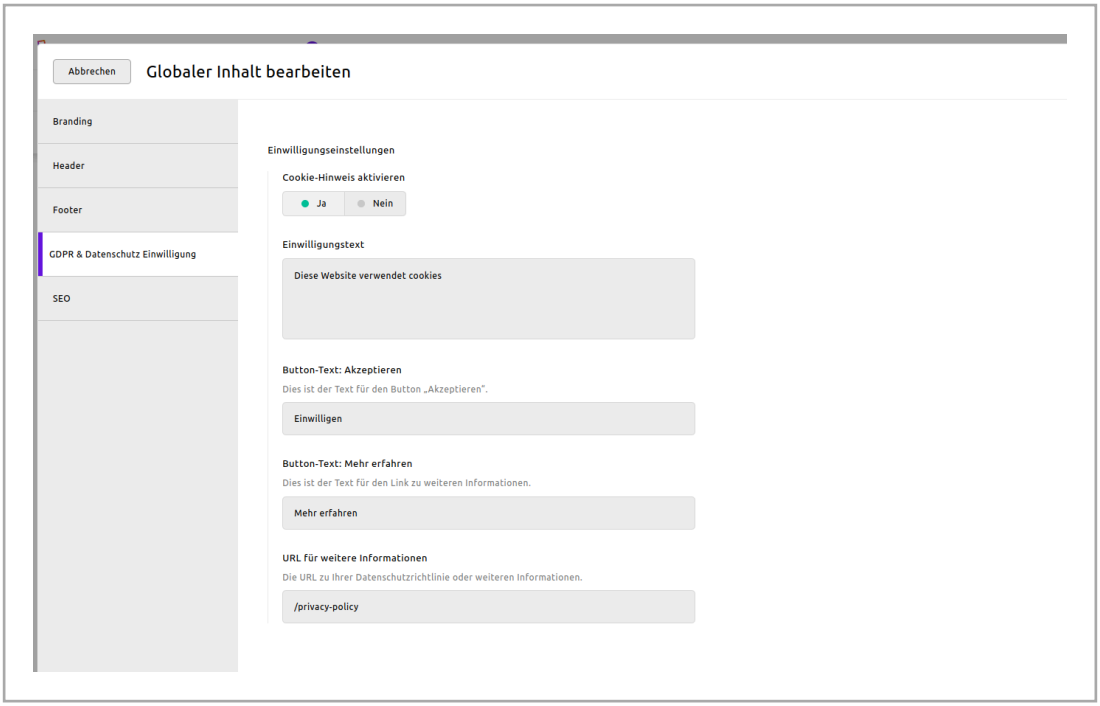

Figure 16: Maintain Cookie Consent Fields in Backoffice

In order to load and initialize the banner on the frontend, the consent values must be retrieved in the `Footer.astro` component in Astro. After fetching the `consentGroup` fields from the global settings, the Osano JavaScript library is loaded from a CDN and configured dynamically, with the final code looking as shown in Listing 20 and Listing 21.

```
---
// Load global settings from ApostropheCMS
const { aposData } = Astro.props;
const consent = aposData.global?.consentGroup || {};
const footerGroup = aposData.global?.footerGroup || {};
const footerBadge = footerGroup?.footerBadge?.items?.[0]?._image[0]?.attachment._urls.original ||
{};
const footerEmail = footerGroup?.footerEmail?.items?.[0]?._image[0]?.attachment._urls.original ||
{};
const baseUrl = import.meta.env.PUBLIC_SITE_URL || "http://localhost:3000"; // fallback for local
dev
const companyInfo = footerGroup?.footerCompanyInfo?.items?.[0];
const phoneInfo = footerGroup?.footerPhoneInfo?.items?.[0];
const footerLinks = footerGroup?.footerLinks?.items?.[0];

// Destructure cookie consent settings
const {
  enableConsentBanner,
  consentMessage,
  acceptLabel,
  learnMoreLabel,
  learnMoreUrl,
} = consent;

---
<footer class="footer certec-footer has-background-light" role="contentinfo">
    <!-- Load external CookieConsent library (only needed if enabled) -->
  <script
    defer
    src="https://cdn.jsdelivr.net/npm/cookieconsent@3/build/cookieconsent.min.js"
  ></script>
  <link
    rel="stylesheet"
    href="https://cdn.jsdelivr.net/npm/cookieconsent@3/build/cookieconsent.min.css"
  />

  <div class="container">
    <div class="footer-content">
      <div class="columns is-variable is-6 is-vcentered">
        <div class="column is-one-third has-text-centered-mobile">
          {
            companyInfo && (
              <div class="html-widget" set:html={companyInfo.code} />
            )
          }
          {
            footerEmail &&
            <img
            src={`${baseUrl}${footerEmail}`}
            alt="Footer Email"
          />
          }

        </div>

        <div class="column is-one-third has-text-centered">
          {phoneInfo && <div class="html-widget" set:html={phoneInfo.code} />}
        </div>

        <div class="column is-one-third has-text-centered">
          { footerBadge &&
          <img
            src={`${baseUrl}${footerBadge}`}
            alt="Footer badge"
          />
          }
        </div>
      </div>
```

Listing 20: Footer with Cookie Consent part 1

```
      <div class="columns is-variable is-6 is-vcentered">
        <div class="column has-text-centered py-0">
          {
            footerLinks && (
              <div class="html-widget" set:html={footerLinks.code} />
            )
          }
        </div>
      </div>
    </div>
  </div>

  <!-- Initialize cookie consent if enabled in settings -->
  {
    enableConsentBanner && (
      <div
        set:html={`<script>
      window.addEventListener("load", function () {
        window.cookieconsent.initialise({
          palette: {
            popup: { background: "#000" },
            button: { background: "#f1d600", text: "#000" }
          },
          theme: "classic",
          content: {
            message: "${consentMessage}",
            dismiss: "${acceptLabel}",
            link: "${learnMoreLabel}",
            href: "${learnMoreUrl}"
          }
        });
      });
    </script>`}
        />
    )
  }
</footer>

<style>
  .certec-footer {
    padding: 2rem 1.5rem;
    background-color: #0070c0 !important;

    .footer-title {
      font-family: tahoma, arial, helvetica, sans-serif;
      color: white;
      text-transform: uppercase;
      margin-bottom: 0;
    }

    .footer-info {
      font-size: 1rem;
      line-height: 1.4;
      color: #333;
    }
  }

  .footer-content {
    max-width: 950px;
    margin: 0 auto;
    padding-left: 2rem;
    padding-right: 2rem;
  }
</style>
```

Listing 21: Footer with Cookie Consent part 2

The main code changes in this version of the `Footer.astro` component is the integration of a cookie consent banner, which makes sure that the website meets data protection regulations such as the GDPR. The resulting banner as it appears in the frontend is shown in Figure Figure 17.

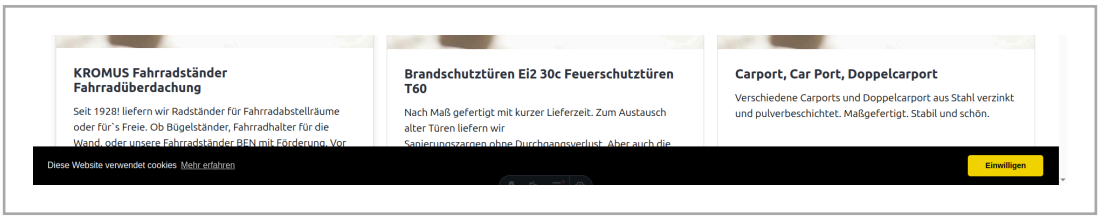

Figure 17: Cookie Consent Banner Displayed on Frontend

5.7 Contact Form Integration

Another essential functionality is the contact form. In WP, this feature is typically enabled via plugins such as Contact Form 7 (WordPress Contributors, n.d.) and its extension Honeypot for Contact Form 7 (Nocean, 2024), both of which are widely used to collect inquiries from customers while protecting against spam.

For Apos, a similar feature is implemented using the officially maintained `@apostrophecms/form` (A-postropheCMS Team, 2024) module. This module provides a flexible, widget based contact form system and integrates well into the backend project. Unfortunately, for the frontend, a custom widget needs to be created to render the form and handle submissions.

To install the module, the following command is run in the backend directory:

```
npm install @apostrophecms/form
```

Once installed, the module and all necessary widgets need to be registered in the root `app.js` as shown in Listing 22.

```
'@apostrophecms/form': {},
// The form widget module, allowing editors to add forms to content areas
'@apostrophecms/form-widget': {},
// Form field widgets, used by the main form module to build forms.
'@apostrophecms/form-text-field-widget': {},
'@apostrophecms/form-textarea-field-widget': {},
'@apostrophecms/form-select-field-widget': {},
'@apostrophecms/form-radio-field-widget': {},
'@apostrophecms/form-file-field-widget': {},
'@apostrophecms/form-checkboxes-field-widget': {},
'@apostrophecms/form-boolean-field-widget': {},
'@apostrophecms/form-conditional-widget': {},
'@apostrophecms/form-divider-widget': {},
'@apostrophecms/form-group-widget': {}
```

Listing 22: Modules added to app.js for form functionality

Next, the form widget must be enabled for the page types where it should be used. In the Apollo template, widget configuration for all page types is outsourced to `/lib/helpers/area-widgets.js`, to avoid duplicated code. The form widget is added to the `widgets` object like so:

```
'@apostrophecms/form': {},
```

With these settings applied, editors will see a new **Forms** menu item appear in the Apos backoffice navigation. This menu becomes available once the form module and its widgets have been properly

registered in the backend configuration. As shown in Figure Figure 18, the new menu item is displayed alongside existing content and configuration tools.

Clicking the **Forms** link opens an overview menu, where all existing forms are listed, including their names, visibility settings and options to edit or delete them. This interface allows editors to manage form configurations. Figure Figure 19 illustrates the form management screen, displaying all created forms in a structured and sortable list.

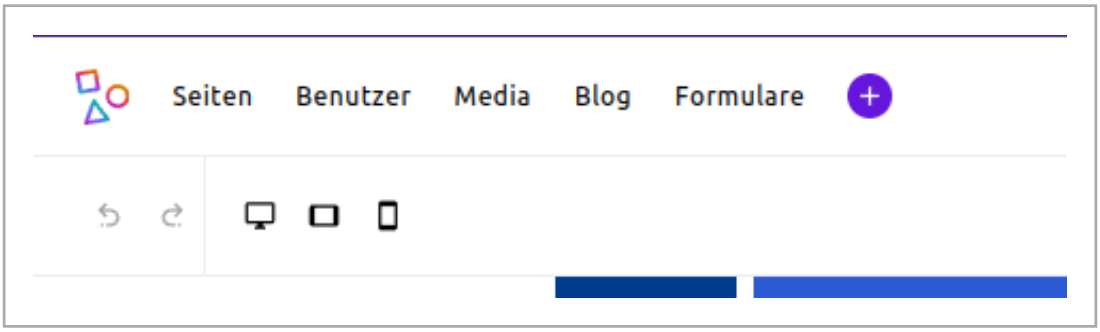

Figure 18: Forms “Formulare” menu Link in ApostropheCMS Backoffice

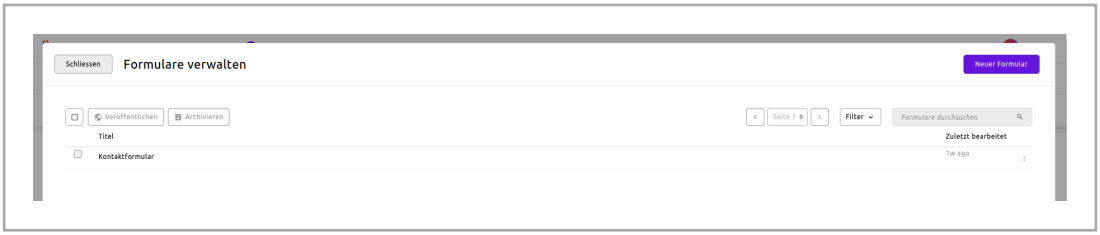

Figure 19: Forms Overview Page in ApostropheCMS

Next step is to create a form. This is done by clicking the **Add Form** (Neues Formular) button in the top right corner of the forms overview page. This opens a new form configuration screen, where editors can define the form’s name, visibility and fields. The form can be configured to be either public or private, in our case to be public.

The form configuration consists of three required fields:

- A mandatory email input field named “From”
- A mandatory text input field labeled “Subject”
- A mandatory text area for the user’s message

In the **After Submit** tab, editors can configure a confirmation heading and message that is shown to users after form submission. They can also enable confirmation emails and specify which field (in this case, “From”) should receive them. The recipient email for the form itself is entered as the final field in the configuration.

To render the form in Astro, a new widget file is created at `/src/widgets/FormWidget.astro` with the content of Listing 23, Listing 24 and Listing 25. This widget reads the form structure passed from Apos and dynamically renders the appropriate fields.

```
---
import RichTextWidget from './RichTextWidget.astro';
const { widget } = Astro.props;
const form = widget._form?.[0];
if (!form) return null;
---

<div id="formWrapper" class="form-widget box">
  <form
    id="contactForm"
    method="POST"
    action="/api/submit"
    data-form-id={form._id}
  >
    <input type="hidden" name="_id" value={form._id} />

    {form.contents?.items?.map((field) => {
      if (field.type === "@apostrophecms/form-text-field") {
        return (
          <div class="field">
            <label class="label" for={field.fieldName}>{field.fieldLabel}</label>
            <div class="control">
              <input
                class="input"
                type={field.inputType || "text"}
                id={field.fieldName}
                name={field.fieldName}
                required={field.required}
                placeholder={field.placeholder}
              />
            </div>
          </div>
        );
      }

      if (field.type === "@apostrophecms/form-textarea-field") {
        return (
          <div class="field">
            <label class="label" for={field.fieldName}>{field.fieldLabel}</label>
            <div class="control">
              <textarea
                class="textarea"
                id={field.fieldName}
                name={field.fieldName}
                required={field.required}
                placeholder={field.placeholder}
              ></textarea>
            </div>
          </div>
        );
      }

      if (field.type === "@apostrophecms/rich-text") {
        return <RichTextWidget widget={field} />;
      }

      return null;
    })}

    <div class="field is-grouped mt-4">
      <div class="control">
        <button type="submit" class="button is-link">{form.submitLabel || "Submit"}</button>
      </div>
    </div>
  </form>
```

Listing 23: FormWidget.astro code part 1

```
<div id="thankYouSection" style="display: none;" class="mt-5">
    <h2 class="title is-4">{form.thankYouHeading}</h2>
    {form.thankYouBody?.items?.map((field) => {
      if (field.type === "@apostrophecms/rich-text") {
        return <RichTextWidget widget={field} />;
      }
      return null;
    })}
  </div>
</div>

<script is:inline type="module">
  const formEl = document.getElementById("contactForm");
  const thankYou = document.getElementById("thankYouSection");

  formEl?.addEventListener("submit", async (e) => {
    e.preventDefault();

    const formDataRaw = new FormData(formEl);
    const formFields = Object.fromEntries(formDataRaw.entries());

    const finalFormData = new FormData();
    finalFormData.append("data", JSON.stringify(formFields));

    try {
      const res = await fetch("/api/submit", {
        method: "POST",
        body: finalFormData
      });

      if (res.ok) {
        formEl.style.display = "none";
        thankYou.style.display = "block";
      } else {
        console.error("Form submit failed:", await res.text());
      }
    } catch (err) {
      console.error("Form error:", err);
    }
  });
</script>
```

Listing 24: FormWidget.astro code part 2

```
<style lang="scss">
.form-widget {
  max-width: 600px;
  margin: 2rem auto;
  padding: 2rem;

  .label {
    font-weight: 600;
  }

  .input,
  .textarea {
    border-radius: 4px;
    border: 1px solid #ccc;
  }

  .button {
    font-weight: bold;
  }

  #thankYouSection {
    animation: fadeIn 0.4s ease-in-out;
  }

  @keyframes fadeIn {
    from { opacity: 0; transform: translateY(10px); }
    to { opacity: 1; transform: translateY(0); }
  }
}
</style>
```

Listing 25: FormWidget.astro code part 3

Each form field, such as text inputs, text areas and rich text blocks, are conditionally rendered depending on its type. The layout is styled using Bulma CSS classes to make sure responsive alignment and consistent design. Additionally, a small inline script handles the form submission. It takes the user input, sends the data to Astro middleware endpoint via POST request and displays a thank you message upon success, which is also dynamically fetched from the form widget. Finally, the component includes css styling to have a good look and feel. The form is centered on the page and the thank you message is animated to fade in after submission.

The final big Astro change for the form is adding a middleware endpoint in the frontend project to securely forward form submissions to Apos. This is necessary to make sure that all required headers, cookies and authentication context are included in the request. Without this middleware, the form data would be sent directly from the client's browser to the backend, bypassing the frontend server, which would violate the architecture of this setup (and also be a big security risk).

To implement it, an `api` folder is created inside the `pages` directory and a new file named `submit.ts` is added with the code shown in Listing 26.

```ts
import type { APIRoute } from "astro";

export const POST: APIRoute = async ({ request }) => {
  const data = await request.formData();
  const formFields = Object.fromEntries(data.entries());
  const formData = new FormData();
  formData.append("data", JSON.stringify(formFields));

  const backendUrl = import.meta.env.PUBLIC_BACKEND_URL || "http://localhost:3000";

  const res = await fetch(`${backendUrl}/api/v1/@apostrophecms/form/submit`, {
    method: "POST",
    body: data,
    headers: {
      "cookie": "hybridsite.csrf=csrf;"
    }
  });

  const text = await res.text();

  return new Response(text, {
    status: res.status,
    headers: { "Content-Type": "text/plain" }
  });
};
```

Listing 26: Form submit middleware code

Figure Figure 20 displays the final version of the form in the frontend, styled and functional, with dynamic content and confirmation support.

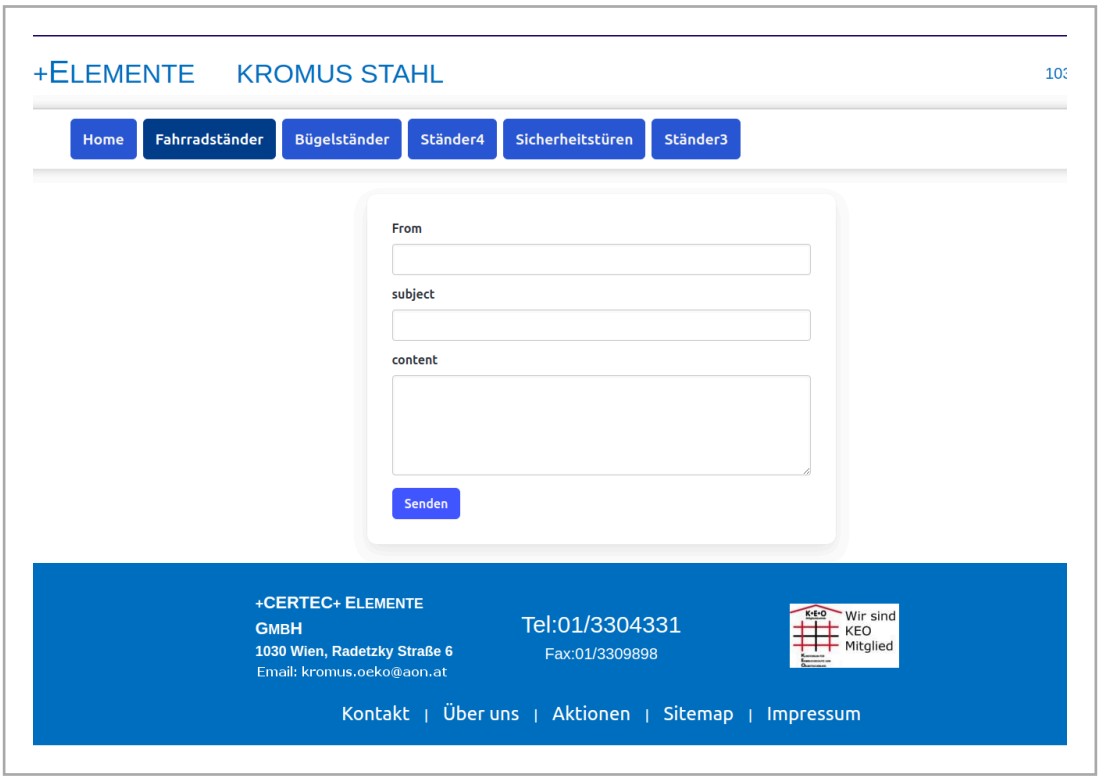

Figure 20: Contact Form Widget in Frontend

To verify that submitted forms are correctly received and confirmation emails are sent out, a local Simple Mail Transfer Protocol (SMTP) server is necessary during development. This allows developers to intercept and review outgoing email messages without relying on a live email provider.

For this purpose, the open source tool MailHog (MailHog Contributors, 2024) is used. There are several tools available for testing email functionality in a local development environment, including options like Papercut (Changemaker Studios, 2024), Mailtrap (Railsware Products, 2024) or MailDev (MailDev Contributors, 2024). For this project, the open source tool MailHog (MailHog Contributors, 2024) was chosen primarily due to its ease of use and straightforward setup. It runs a lightweight SMTP server locally and offers a clean web interface for inspecting outgoing emails, making it great for validating form delivery and confirmation messages during development.

Once MailHog is set up and running, see its GitHub repository for detailed installation instructions, it binds to ports 1025 (SMTP) and 8025 (HTTP), as shown in the output below:

```
2025/04/09 22:27:20 Using in-memory storage
2025/04/09 22:27:20 [SMTP] Binding to address: 0.0.0.0:1025
[HTTP] Binding to address: 0.0.0.0:8025
2025/04/09 22:27:20 Serving under http://0.0.0.0:8025/
Creating API v1 with WebPath:
Creating API v2 with WebPath:
```

To route outgoing emails from Apos to MailHog, the following configuration must be added to the `@apostrophecms/email` module in the Astro's `app.js` as shown in Listing 27.

```
'@apostrophecms/email': {
    options: {
        nodemailer: {
          host: 'localhost',
          port: 1025,
          secure: false,
          auth: null
        },
        from: 'Your Name <noreply@example.com>'
    }
},
```

Listing 27: Email configuration for MailHog

After restarting the Apos server, any submitted forms will send an email to the configured recipient and a confirmation email to the user. These messages will appear in MailHog's inbox interface, accessible at `http://localhost:8025`, as shown in Figure Figure 21.

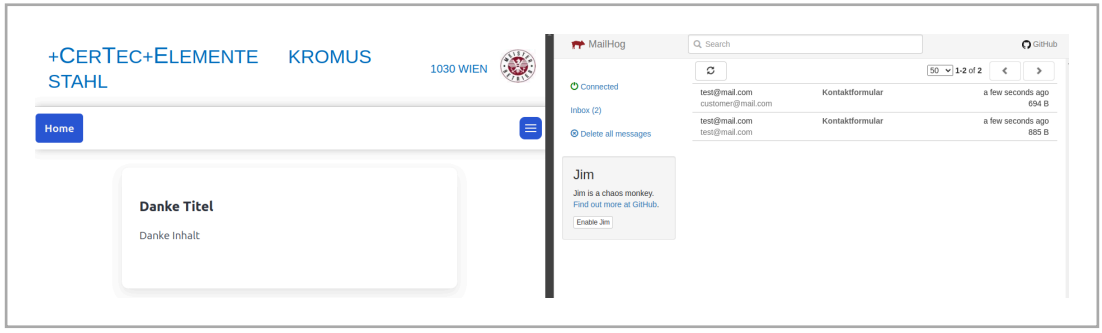

Figure 21: MailHog Inbox for Form Email Testing

5.7.1 Contact Form CAPTCHA Configuration

To protect the contact form from spam and automated bots and other abuse, reCAPTCHA by Google is add to the form workflow, as suggested by the offical Apollo github documentation. This offers a more realiable defense than traditional honeypot fields. As recommended by the Apos form module documentation (ApostropheCMS Team, 2024), reCAPTCHA is the preferred option for spam protection. Although Google's reCAPTCHA is quite popular, there are several alternative CAPTCHA solu-

tions that also offer privacy focused or open source implementations. For example there are hCaptcha, FriendlyCaptcha and others that are listed and compared on community maintained platforms (European Alternatives, 2024).

The free tier of Google reCAPTCHA (Google Cloud, 2024) allows up to 1,000 validated submissions per day, which is sufficient for SME application. After setting up an account and obtaining the credentials, the following options are added to the form module configuration in the Apos `app.js` as shown in Listing 28.

```
'@apostrophecms/form': {
    options: {
      recaptchaSecret: 'secret',
      recaptchaSite: 'sitesecret'
    }
  }
```

Listing 28: reCAPTCHA configuration

The corresponding `FormWidget.astro` file is then updated with the final code from Listing 29, Listing 30 and Listing 31 in the Astro project to include the necessary site key and dynamically load the reCAPTCHA script if it is enabled for the form.

```
---
import RichTextWidget from './RichTextWidget.astro';
const { widget } = Astro.props;
const form = widget._form?.[0];
if (!form) return null;
const recaptchaSiteKey = form.enableRecaptcha ? import.meta.env.RECAPTCHA_SITE_KEY : null;
---

<div id="formWrapper" class="form-widget box">
  <form
    id="contactForm"
    method="POST"
    action="/api/submit"
    data-form-id={form._id}
    data-recaptcha-sitekey={recaptchaSiteKey}
  >
    <input type="hidden" name="_id" value={form._id} />

    {form.contents?.items?.map((field) => {
      if (field.type === "@apostrophecms/form-text-field") {
        return (
          <div class="field">
            <label class="label" for={field.fieldName}>{field.fieldLabel}</label>
            <div class="control">
              <input
                class="input"
                type={field.inputType || "text"}
                id={field.fieldName}
                name={field.fieldName}
                required={field.required}
                placeholder={field.placeholder}
              />
            </div>
          </div>
        );
      }

      if (field.type === "@apostrophecms/form-textarea-field") {
        return (
          <div class="field">
            <label class="label" for={field.fieldName}>{field.fieldLabel}</label>
            <div class="control">
              <textarea
                class="textarea"
                id={field.fieldName}
                name={field.fieldName}
                required={field.required}
                placeholder={field.placeholder}
              ></textarea>
            </div>
          </div>
        );
      }

      if (field.type === "@apostrophecms/rich-text") {
        return <RichTextWidget widget={field} />;
      }

      return null;
    })}

    <div class="field is-grouped mt-4">
      <div class="control">
        <button type="submit" class="button is-link">{form.submitLabel || "Submit"}</button>
      </div>
    </div>
  </form>
```

Listing 29: FormWidget with recaptcha code part 1

```
  <div id="thankYouSection" style="display: none;" class="mt-5">
    <h2 class="title is-4">{form.thankYouHeading}</h2>
    {form.thankYouBody?.items?.map((field) => {
      if (field.type === "@apostrophecms/rich-text") {
        return <RichTextWidget widget={field} />;
      }
      return null;
    })}
  </div>
</div>

<script is:inline type="module">
  const formEl = document.getElementById("contactForm");
  const thankYou = document.getElementById("thankYouSection");
  const siteKey = formEl?.dataset.recaptchaSitekey;

  if (siteKey) {
    const script = document.createElement('script');
    script.src = `https://www.google.com/recaptcha/api.js?render=${siteKey}`;
    script.async = true;
    script.defer = true;
    document.head.appendChild(script);
  }

  formEl?.addEventListener("submit", async (e) => {
    e.preventDefault();

    const formDataRaw = new FormData(formEl);
    const formFields = Object.fromEntries(formDataRaw.entries());

    // If reCAPTCHA is enabled, fetch token before submit
    if (siteKey && window.grecaptcha) {
      try {
        const token = await grecaptcha.execute(siteKey, { action: 'submit' });
        formFields.recaptcha = token;
      } catch (err) {
        console.error("reCAPTCHA failed to load:", err);
        return;
      }
    }
```

Listing 30: FormWidget with recaptcha code part 2

```
    const finalFormData = new FormData();
    finalFormData.append("data", JSON.stringify(formFields));

    try {
      const res = await fetch("/api/submit", {
        method: "POST",
        body: finalFormData
      });

      if (res.ok) {
        formEl.style.display = "none";
        thankYou.style.display = "block";
      } else {
        console.error("Form submit failed:", await res.text());
      }
    } catch (err) {
      console.error("Form error:", err);
    }
  });
</script>

<style lang="scss">
.form-widget {
  max-width: 600px;
  margin: 2rem auto;
  padding: 2rem;

  .label {
    font-weight: 600;
  }

  .input,
  .textarea {
    border-radius: 4px;
    border: 1px solid #ccc;
  }

  .button {
    font-weight: bold;
  }

  #thankYouSection {
    animation: fadeIn 0.4s ease-in-out;
  }

  @keyframes fadeIn {
    from { opacity: 0; transform: translateY(10px); }
    to { opacity: 1; transform: translateY(0); }
  }
}
</style>
```

Listing 31: FormWidget with recaptcha code part 3

This implementation makes sure that the form submission is only processed if reCAPTCHA successfully verifies the user. The check is performed on the client side using Google's JavaScript API.

To finalize the setup, the reCAPTCHA site key is added to the frontend's `.env` file:

```
RECAPTCHA_SITE_KEY=<recaptchaSite key that was set in the backend>
```

When the page containing the form, `/kontakt` in this case, is loaded, the reCAPTCHA symbol becomes visible in the bottom right corner of the screen. This visual indicator confirms that the validation service has been successfully setup, loaded and is protecting the form. The functionality and placement of the reCAPTCHA integration are shown in figure Figure 22.

Figure 22: Visible reCAPTCHA on Form Page

5.8 SEO Configuration

Search Engine Optimization (SEO) is essential for the website and is partially supported by default in Apos. The platform provides a OOTB backoffice tab for managing page level SEO settings, including meta titles, descriptions and visibility options for search engines. This settings are visible in figure Figure 23 and also described in the official documentation (ApostropheCMS Team, 2024).

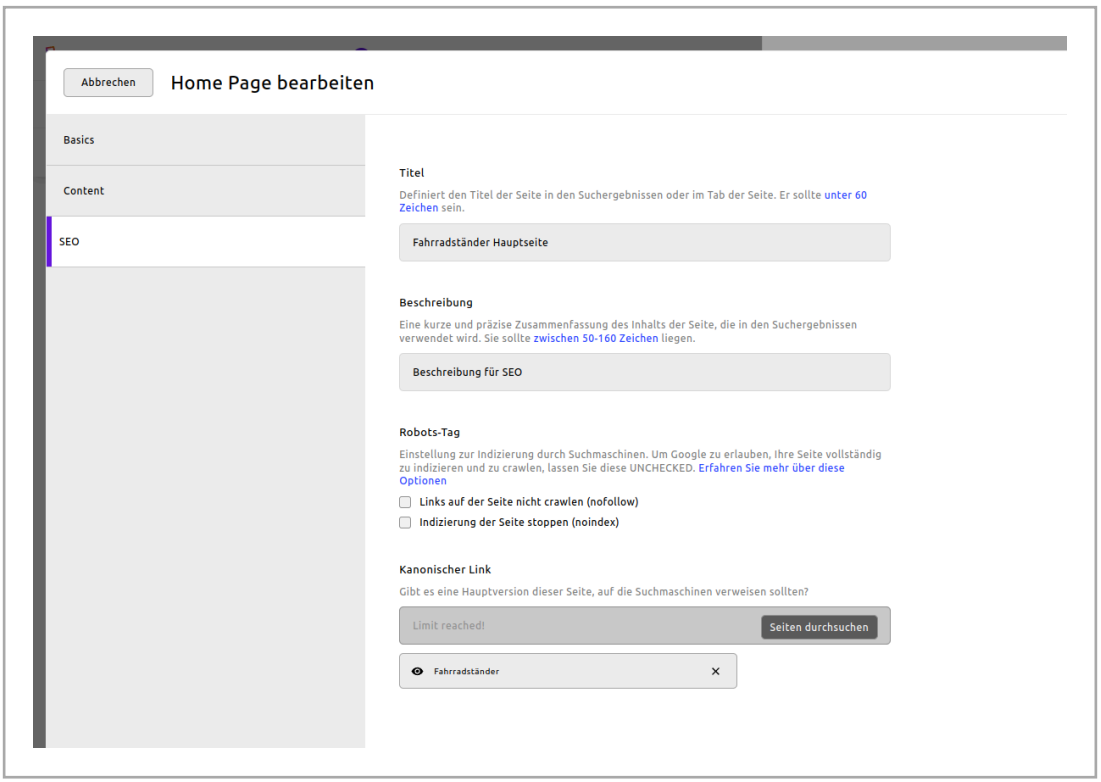

Figure 23: SEO Configuration in ApostropheCMS

In addition to the default tools, Apos also offers an advanced SEO Assistant (ApostropheCMS Team, 2024) plugin. While the SEO Assistant provides optimization suggestions and AI autofill functionality, just to name a few, it is not used in this project.

To properly integrate the meta title into the Astro HTML output, a manual update is required in the Apollo template. Specifically, in the `frontend/src/pages/[...slug].astro` file, the following line should be added after the `<head>` block (e.g., after line 35):

```
<meta name='title' content={aposData.page?.seoTitle} />
```

5.9 Media Management

Apos provides support for managing media assets, particularly images, through its integrated media library and widget system. Several widgets, such as the image, slideshow and rich text area widgets, come with OOTB functionality to upload, select and configure images directly from the backoffice. This allows editors to easily add media into content areas without needing to touch code.

The image selection is built into the editor UI. Users can upload new media or browse uploaded assets via a searchable media library. Uploaded files are stored on the server.

Each image widget supports additional configuration options, such as alternative text for accessibility, cropping and aspect ratio controls and metadata management. Editors can also reuse media assets across different pages, content blocks and widgets.

While Apos does not offer OOTB video hosting, external media, such as YouTube embeds or remote image URLs, can be supported using the rich text and HTML widgets, or through custom widget development.

More information on media usage and customization is available in the official documentation (ApostropheCMS Team, 2024). Nevertheless, for this point, there is no need to implement any custom code in the Astro project, as the Apollo template already provides a fully functional media library and image widget.

5.10 Maintain page content

The Apollo template provides a wide variety of OOTB widgets and layout components that enable non technical users to build and structure web pages directly from the Apos backoffice.

Two of the most important structural tools are the `GridLayoutWidget` and the `RowsLayoutWidget`, which serve as container components for placing other widgets. These layouts define how content is arranged on the page, including the number of columns, spacing and responsive behavior.

A variety of display widgets that come OOTB are:
- Image Widget
- Video Widget
- Form Widget
- Rich Text Widget
- Slideshow Widget
- Hero Section Widget
- Accordion Widget
- Card and Link Widgets

These widgets can either be placed within a page's content fields or wrapped inside layout widgets for control over structure and appearance.

Figure Figure 24 provides an overview of the available layouts and widgets in the Apollo template.

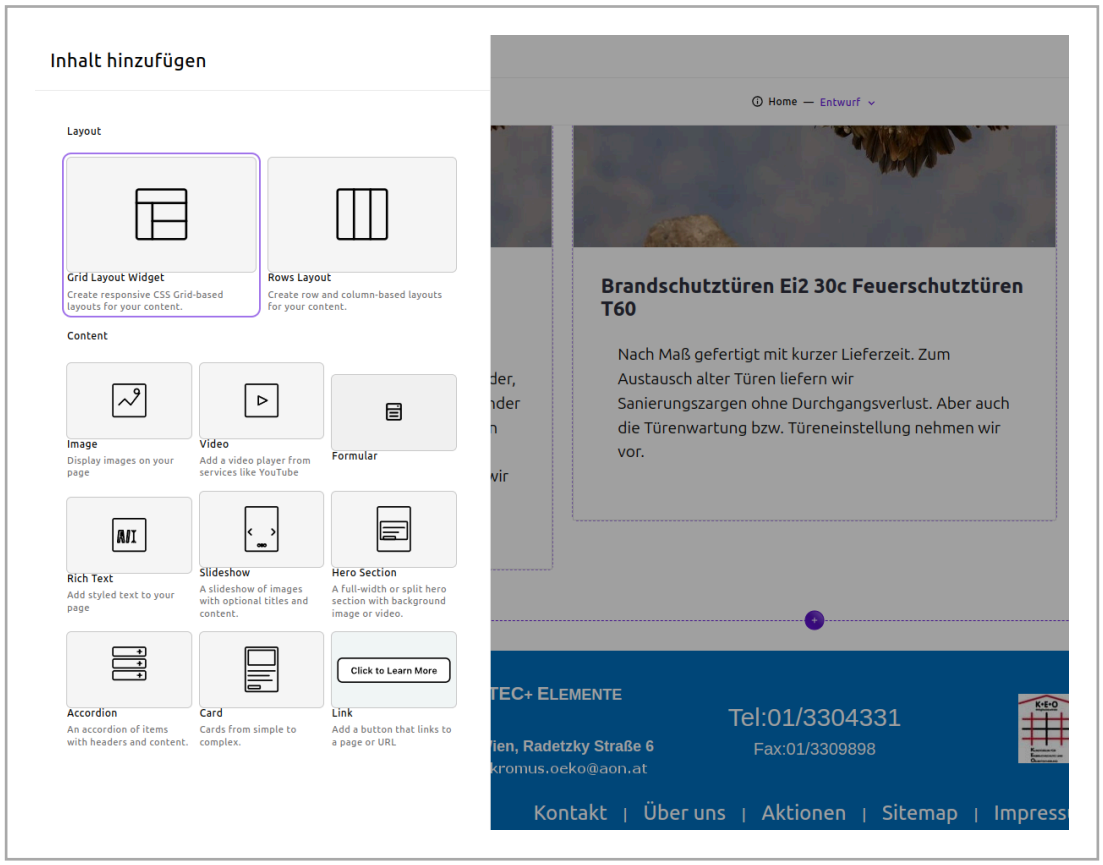

Figure 24: Apollo Widgets and Layouts Overview

The `RowsLayoutWidget` enables content editors to define multiple sections with fixed column widths. As shown in Figure Figure 25, a row can be split into three columns of equal width (33%), each housing a Card Widget that contains an image, title and supporting text.

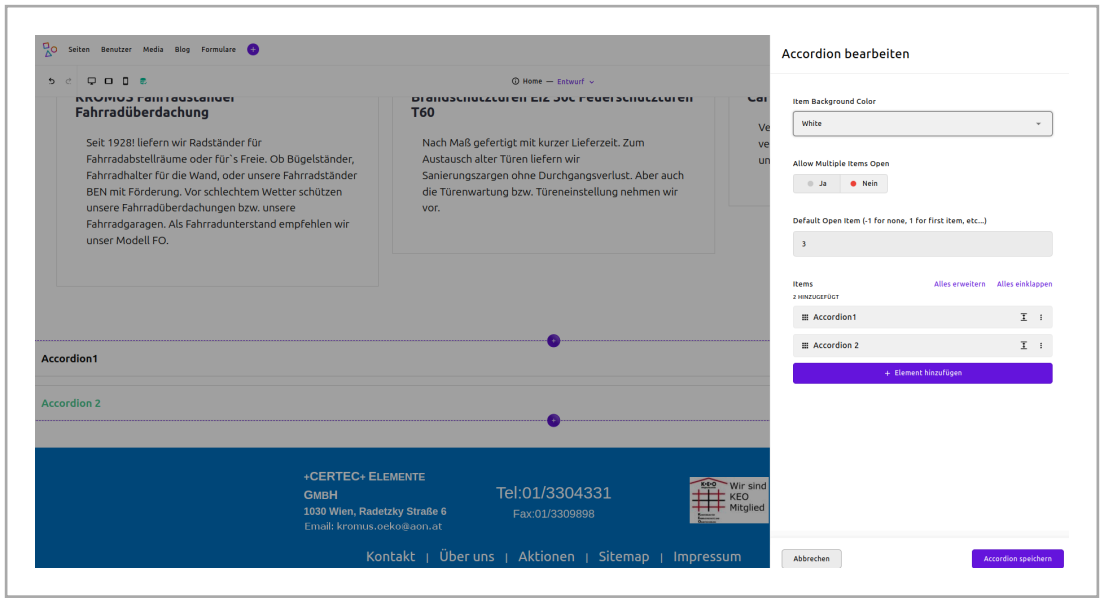

Figure 25: Rows Layout with Cards

The `AccordionWidget` offers collapsible content areas that can hold any number of widgets inside each section. Users can add as many items as needed, each with content such as text, images, or even forms. Figure Figure 26 shows this widget with multiple populated accordion entries.

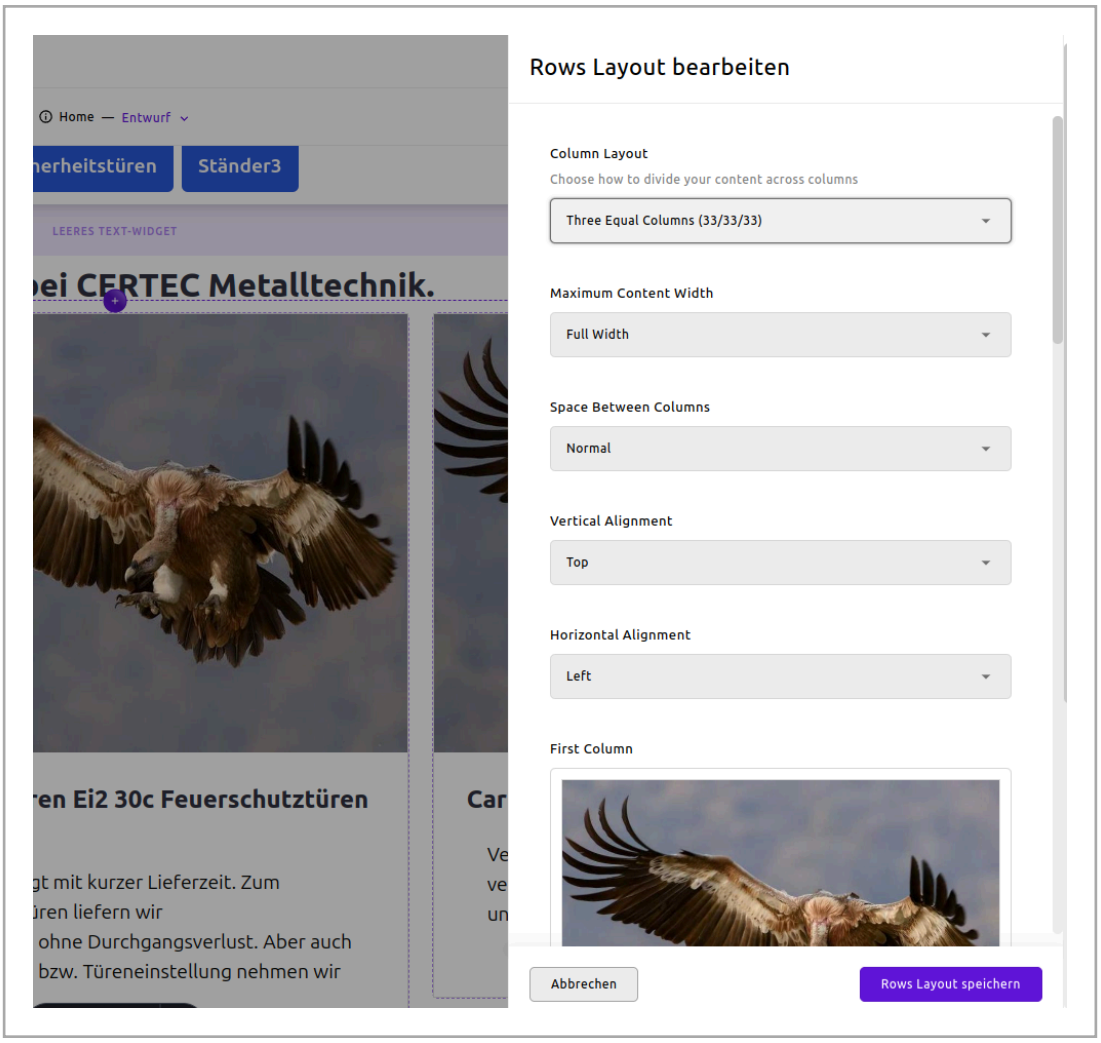

Figure 26: Accordion Widget Example

If additional widgets are required beyond those included in Apollo, developers can easily create their own. Apos provides a process for building custom widgets, especially for developers already familiar with implementing interactive components like the Form Widget.

Some widgets simply display content (e.g., text or images from the backend), while others can handle interactions or perform actions, similar to the form submission logic. The official documentation includes a guide for creating new widgets (ApostropheCMS Documentation Team, 2024).

All code changes and customization that were done in this chapter can be found at the following GitHub repository: https://github.com/TheFiler/apos_thesis_project.

5.11 Hosting Platform Selection

The next step is to choose a hosting provider for headless CMS setup built with Node.js and MongoDB. The selection criteria are based on Section 1.3.4. In addition to those criteria, due to the selection of Apos, the hosting provider must support Node.js and MongoDB as a database solution.

After evaluating a broad spectrum of platforms, from developer focused services to enterprise level cloud solutions, a shortlist of suitable providers was gathered. For the purpose of demonstrating the deployment process in this thesis, Scalingo was selected due to its simplicity and support for both Node.js and MongoDB. The actual hosting provider used by the SME remains confidential and will not be disclosed in this document. The summarized evaluation of platforms that support Node.js and MongoDB, including both European and global providers, can be checked in Table 2.

Provider	Node.js Support	MongoDB Hosting	Data Center Regions	Starting Price per month
NodeChef (NodeChef, 2024)	Yes	Yes	US, EU (Paris), Asia-Pacific	$9
DigitalOcean (CyberNews Editors, 2024)	Yes	No (external setup)	Global	$5
Heroku (AppSignal Team, 2022)	Yes	Via Add-on	US, Europe	Free tier
OVHcloud (OVHcloud, 2024)	Yes	No	Europe, North America, APAC	€2.99
Hostinger (CyberNews Editors, 2024)	Yes	No	Europe, US, Asia	$4.64
Scalingo (Scalingo, 2024)	Yes	Yes	Europe (France)	€7–15
Amazon Web Services (AWS) (Amazon Web Services, 2024)	Yes (Beanstalk/EC2)	Yes (Atlas/self-managed)	Global	$10–15+
Google Cloud Platform (GCP) (Google Cloud, 2024)	Yes (Cloud Run/App Engine)	Yes (Atlas/self-managed)	Global	$10–15+
Microsoft Azure (Microsoft Azure, 2024)	Yes (App Services)	Yes (Cosmos DB/Atlas)	Global	$13+

Table 2: Hosting Platform Comparison for Node.js + MongoDB Support

These platforms all provide a viable foundation for deploying Node.js CMS systems. Some (like NodeChef or Scalingo) offer OOTB MongoDB support, while others require external integration, such as with MongoDB Atlas. The balance between configuration flexibility and ease of use varies depending on the provider.

A comparative evaluation of these platforms against traditional PHP based WordPress hosting environments will be discussed in the evaluation chapter Section 6.

5.12 Production Deployment

To host the final Apos website, deployment is done on the Scalingo platform. It is a European Platform as a Service (PaaS) provider that supports Node.js applications and provides an integrated MongoDB addon, making it a suitable hosting solution for this project.

Before deployment, the backend Apos and frontend Astro projects are each committed to separate Git repositories. In this case, GitHub is used, as it offers free private repositories. However, other services such as GitLab, Bitbucket and others could be used as alternatives.

5.12.1 Backend Deployment

After registration on Scalingo, a new webapp resource is created (Scalingo Documentation Team, 2024) for the backend. Access to the GitHub apollo repositories is granted through the Scalingo dashboard. Next, a MongoDB addon is attached (Scalingo Documentation Team, 2024) to the webapp as shown in Figure Figure 27.

Figure 27: MongoDB Addon in Scalingo Webapp

Once the database is linked, the following environment variables must be configured:

```
APOS_EXTERNAL_FRONT_KEY=<frontend_key>
APOS_MONGODB_URI=$SCALINGO_MONGO_URL
APOS_RELEASE_ID=1.0.0
NODE_ENV=production
```

- `APOS_EXTERNAL_FRONT_KEY`: Shared key used by the frontend and backend to authenticate requests.
- `APOS_MONGODB_URI`: MongoDB connection string (provided automatically by the MongoDB addon).
- `APOS_RELEASE_ID`: Custom identifier for the version of the backend deployment.
- `NODE_ENV`: Should be set to `production` to optimize performance.

These variables are shown configured in the webapp's environment settings (Figure Figure 28).

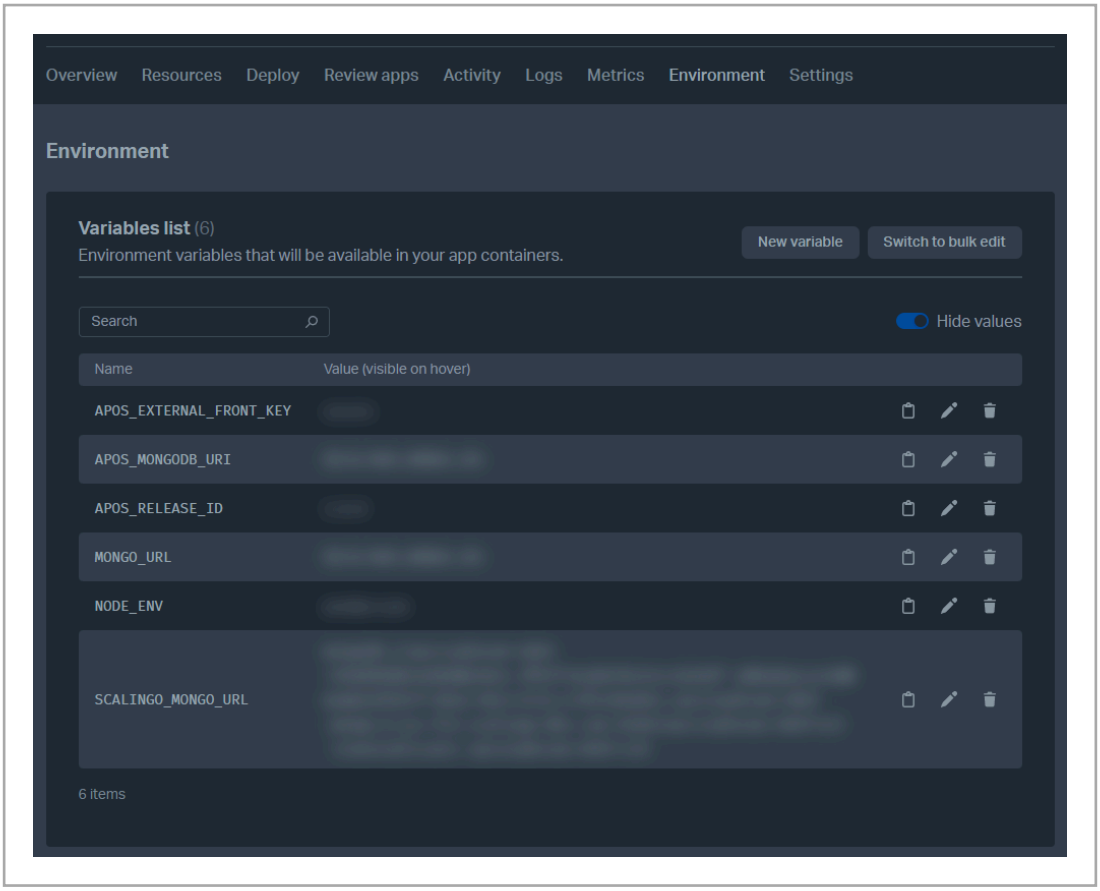

Figure 28: Scalingo Environment Variables

To complete the backend deployment, navigate to the Deploy tab and trigger a manual deployment (Figure Figure 29).

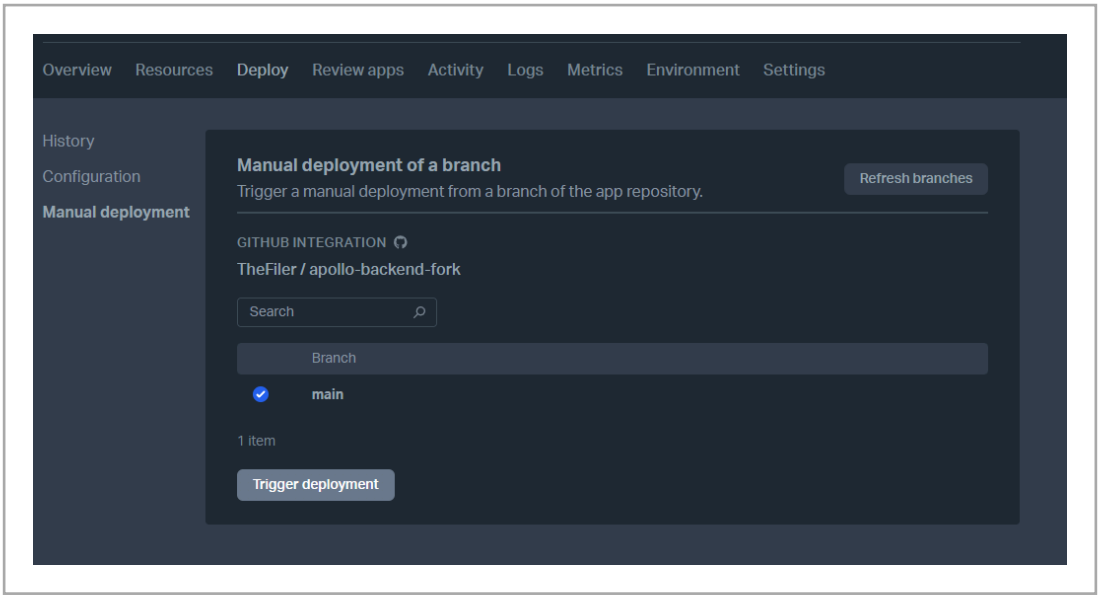

Figure 29: Trigger Manual Deployment on Scalingo

Once deployment succeeds, the backend can be verified by clicking the Open Application button. This opens the backend's status page, shown in Figure Figure 30.

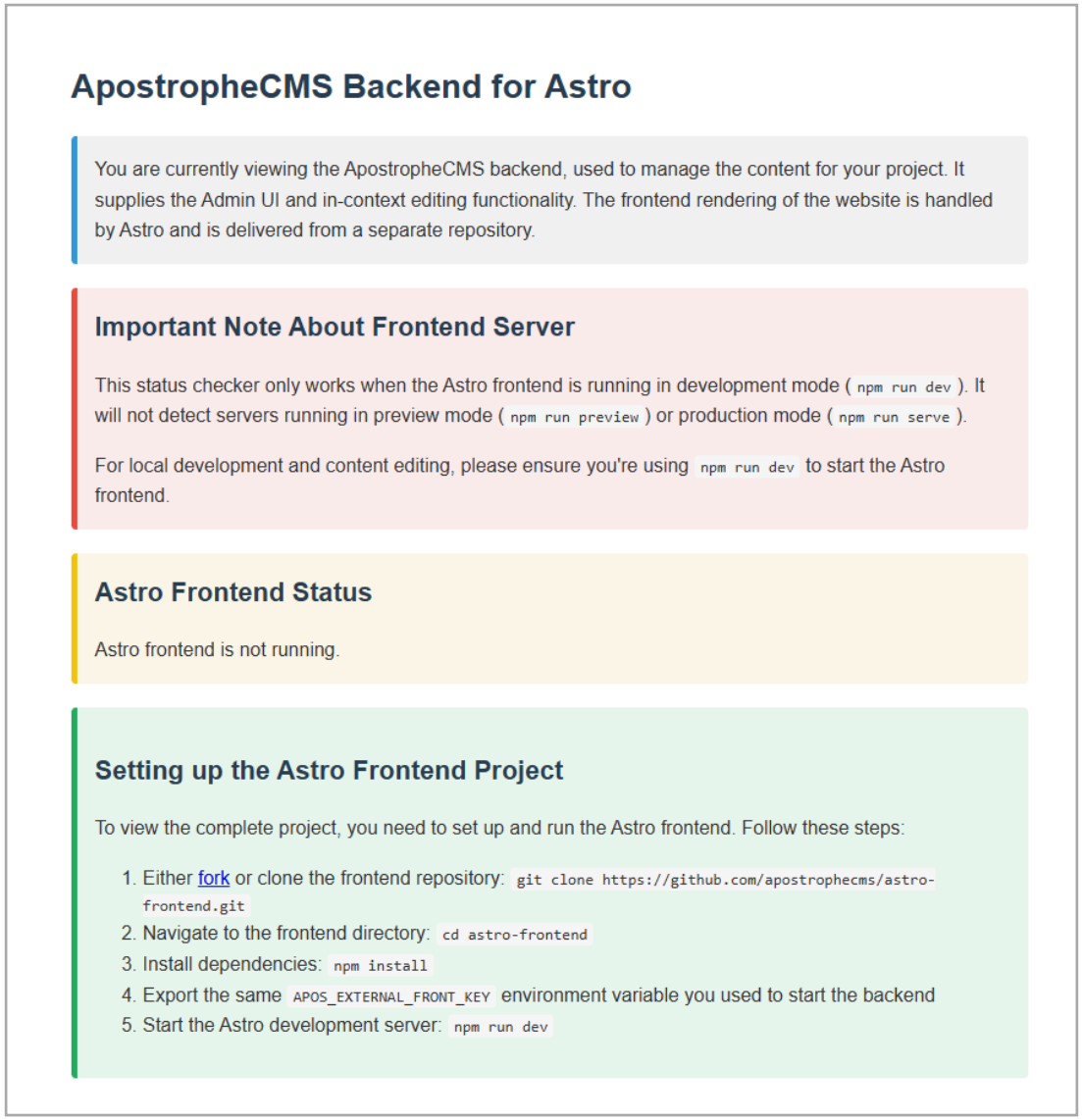

Figure 30: ApostropheCMS Backend Deployed Successfully

5.12.2 Frontend Deployment

Deployment of the frontend follows a similar process. First, the frontend source code must be moved into its own GitHub repository and linked to a new Scalingo webapp.

Before deploying, open the `astro.config.mjs` file in the frontend project and apply the following changes:

- Uncomment line 12 and line 30, as noted in the Apollo template repository (ApostropheCMS Contributors, 2024h).
- These changes ensure proper communication with the backend server over the network.

Then, configure the following environment variables in the Scalingo frontend webapp:

```
APOS_EXTERNAL_FRONT_KEY=<frontend_key>
NODE_ENV=production
APOS_HOST=<backend_scalingo_url>
```

- `APOS_EXTERNAL_FRONT_KEY`: Must match the key used by the backend.
- `NODE_ENV`: Set to `production` for optimized builds.

- `APOS_HOST`: The URL of the deployed backend webapp (can be found in the backend's Scalingo dashboard).

Finally, navigate to the Deploy tab and trigger the frontend deployment. Once that is deployed, the frontend can be verified by clicking the Open Application button. This opens the frontend of the website, where users can even login to the backoffice using /login path.

5.12.3 Hosting and Database Initialization

Although this deployment uses Scalingo, any hosting provider that supports Node.js and MongoDB can be used. The core deployment workflow remains the same, but environment variable management and deployment pipelines may differ slightly depending on the hosting service.

Depending on how the project was built, content setup may follow one of two approaches:

- If development took place locally, the MongoDB database can be exported and imported into the production instance.
- If development is done directly on the production instance, content can be initialized from scratch.

To initialize the site with sample content and configure an admin account, run the following command on the production server:

```
npm run load-starter-content
```

This script populates the site with sample pages, images and sets the admin password—just like in the local development environment. If it becomes necessary to export an existing MongoDB database or collection to a Scalingo hosted instance, this process can be performed using the official Scalingo export guide (Scalingo Documentation Team, 2024). As an alternative, MongoDB Compass offers a graphical interface for importing and exporting data (MongoDB Inc., 2024). For cases, such as server to server transfers or large dataset migrations, a guide by Vipin Nation provides additional information and examples (Nation, 2022).

Generally, regarding the whole Scalingo details on deployment, configuration and platform specific features, the official Scalingo documentation provides comprehensive guidance covering all aspects discussed in this section (Scalingo Documentation Team, 2024).