Designing a Modular Web Interface for the ISAN Curing System: Enhancing Information Accessibility in Emergency Situations

Project Thesis

Nikkel Lennart Heesen

Braunschweig, 15.01.2026

Advisors:	Viktor Sobotta
Supervisor:	Prof. Dr. Thomas M. Deserno
Matriculation Number:	4810083

Technische Universität Braunschweig
Carl-Friedrich-Gauß-Fakultät
Peter L. Reichertz Institut

Hello I am an abstract

Contents

1 List of Notes

2 Introduction

2.1 Motivation

Emergency response scenarios are characterized by extreme time pressure, high cognitive load, and the need for rapid, well-informed decision-making. In such contexts, the timely availability of accurate information is critical. The World Health Organization (WHO) has reported that a significant number of fatalities in emergency and disaster situations could be prevented if first responders had faster access to relevant and reliable information (World Health Organization, 2019). These findings emphasize that effective emergency response depends not only on operational capabilities but also on how information is provided to human operators.

At the same time, substantial progress has been made in the digitalization of emergency response infrastructures. Modern fire departments and emergency services operate a wide range of specialized systems, including alerting, dispatch, and reporting platforms. A standard like the International Standard Accident Number (ISAN) allows incidents to be uniquely identified and to correlate data from different systems / data silos. From a technical standpoint, the infrastructure required to exchange life-saving data across heterogeneous systems is largely available. However, the existence of such systems alone does not ensure that information can be accessed and interpreted efficiently in real-world emergency situations.

From a human–computer interaction (HCI) perspective, the user interface represents the critical link between complex backend systems and human decision-makers. Research in HCI has demonstrated that interface design has objective effects on user performance, error rates, and cognitive workload (Norman, 2013; Nielsen, 1994). In emergency contexts, responders operate under stress and time constraints that limit working memory and increase the risk of errors (Endsley, 1995). Poorly designed user interfaces can amplify these limitations by presenting information in an unstructured, fragmented, or unintuitive manner, leading to delayed responses or misinterpretation of critical data.

Despite established usability principles and modern web technologies, many emergency management systems continue to suffer from fragmented workflows and insufficient integration of relevant data sources. Emergency responders are often required to consult multiple systems to obtain a complete picture of an incident, increasing cognitive effort and response time (Blandford et al., 2012). Such interaction patterns conflict with fundamental HCI principles, including the reduction of cognitive load and the prioritization of critical information.

The motivation for this thesis is to address these challenges by designing and implementing a user-centered curing system interface that consolidates incident-related data from multiple sources into a single web application. By leveraging standardized identifiers such as ISAN and applying established HCI principles, the system aims to support situational awareness and enable faster, more informed decision-making. Ultimately, this work seeks to demonstrate how thoughtful interface design can contribute to more effective emergency response and support the overarching goal of saving lives.

2.2 Related work

- research of design choices and standards in emergency software.
- analysis of other software projects similar to this

2.2.1 Design standards and accessibility in interactive software

The design of interactive systems, especially in safety-critical domains such as emergency response, is guided by international standards that aim to ensure usability, accessibility, and reliability in general. In the context of human–computer interaction, these standards provide a shared framework for evaluating and designing user interfaces that support effective human performance under diverse conditions.

One of the most prevalent standards in the EU regarding this area is the DIN EN ISO 9241 series, which defines ergonomic requirements for human–system interaction. It serves as an umbrella standard covering both hardware and software systems up to workplace ergonomics, with a strong emphasis on usability principles such as effectiveness, efficiency, user satisfaction (1). ISO 9241-11 employs different definitions and concepts regarding usability. Usually ISO 9241-110 is of particular relevance, which formulates core dialogue principles including suitability for the task, self-descriptiveness, conformity with user expectations, and error tolerance (2). However, ISO 9241-110:2020 explicitly states that its principles are not universally applicable to every usage context and must be interpreted in relation to the specific task domain and user group like in safety-critical domains (3). This limitation is especially relevant in the case of emergency response systems, where users operate under extreme time pressure, high stress, and potentially degraded perceptual conditions. As a result, while ISO 9241 provides valuable general guidance, it does not prescribe concrete design solutions tailored to emergency scenarios, leaving significant room for domain-specific interpretation and adaptation.

In addition to usability standards, **accessibility** regulations play a central role in contemporary user interface design. In the European context, the Barrierefreie-Informationstechnik-Verordnung (BITV 2.0) defines legal requirements for accessible digital systems used by public institutions. BITV 2.0 is closely aligned with EN 301 549, a European standard that specifies accessibility requirements for information and communication technology. EN 301 549 incorporates the Web Content Accessibility Guidelines (WCAG), particularly in Chapter 9, which addresses web-based user interfaces.

WCAG defines success criteria across different conformance levels (A, AA, and AAA), with EN 301 549 encouraging AAA compliance where feasible. Criteria such as enhanced contrast ratios, scalable text, and clear visual hierarchy are particularly relevant for emergency response interfaces, as they support readability in challenging environments and reduce visual strain. While accessibility standards are often associated with users with permanent impairments, HCI research emphasizes their broader relevance for situational impairments, such as reduced visibility, fatigue, or stress—conditions that are common in emergency operations.

Existing standards thus establish an important foundation for usability and accessibility, but they do not fully address the specific interaction challenges of emergency response software. The need to prioritize critical information, support rapid situation assessment, and minimize cognitive load requires design decisions that go beyond general-purpose standards. Consequently, this thesis builds upon established usability and accessibility guidelines while exploring their application and adaptation within the context of an emergency-focused curing system interface.

2.2.2 Existing emergency response software

A number of software systems have been developed to support emergency response operations by aggregating incident-related information and assisting coordination. Analyzing such systems provides insight into established design patterns, interaction paradigms, and functional priorities within the domain.

1. rescueTablet

rescueTABLET is a digital emergency management platform widely adopted by fire departments and emergency organizations in Germany, reportedly used by more than 750 organizations. The system is designed to support incident command by providing an overview of operational data, resource allocation, and situational information.

From an interaction design perspective, rescueTABLET follows a dashboard-oriented layout combined with a persistent sidebar navigation. Core information is presented using tiles and panels that group related data, allowing users to access different functional areas with minimal navigation depth. This design reflects a common pattern in emergency software, where information is spatially organized to support rapid scanning and recognition. The use of consistent visual structures across views supports user orientation and reduces the need for relearning during high-stress situations.

2. Tablet Command

Tablet Command is an emergency response platform primarily used by fire services in the United States and Canada. According to publicly available information, it is employed by more than 800 public safety agencies and has been used to manage over 170,000 incidents as of 2024. The system is explicitly designed for fire service operations, with a strong emphasis on unit and resource management.

The user interface of Tablet Command also adopts a dashboard-centric structure with a sidebar-based navigation model. The system places particular focus on tracking units, assignments, and incident progression, reflecting its operational priorities. A high degree of configurability allows agencies to adapt the interface to local procedures and preferences, which can support organizational fit but may also introduce variability in interaction patterns across deployments.

In addition to its client-facing interface, Tablet Command provides a web-based backend portal for configuration and management tasks. The release of a major version update in

2021 introduced a revised interface structure, further emphasizing modular dashboards and centralized access to incident data. From an HCI standpoint, this separation between operational and administrative interfaces reflects a role-based interaction model, which can reduce complexity for users in the field.

3. D4H

D4H is a comprehensive emergency management platform used globally by over 100,000 responders across hundreds of organizations in more than 37 countries and six continents (4). According to their website, it has been recognized as a market leader in winter 2025 and ranked the number one emergency management software on G2. These adoption metrics indicate that D4H serves a wide variety of operational contexts, from local fire departments to multinational emergency response organizations.

From a design perspective, D4H emphasizes a modular and role-based interface. Operational dashboards consolidate key incident information, resource allocation, and communication channels in a single view, while secondary modules allow task-specific interaction, such as personnel management, equipment tracking, or post-incident reporting. This separation of concerns reflects a deliberate attempt to reduce cognitive load for users who must make rapid, high-stakes decisions.

Navigation within D4H relies on persistent sidebar menus and context-sensitive panels, similar to other emergency platforms such as rescueTABLET and Tablet Command. Information is typically presented through tiles, tables, and summary panels, facilitating quick scanning and recognition of critical incident details. The configurable nature of the interface allows organizations to tailor the layout and functionality to their operational procedures, which supports adoption across diverse user groups but may require careful user training to maintain consistency.

The global reach of D4H also highlights the importance of flexible localization and adherence to accessibility and usability standards. In emergency contexts where users operate under stress, cognitive load, or adverse environmental conditions, interface clarity, consistency, and predictability are crucial. D4H's design exemplifies how large-scale emergency software can implement these principles at both operational and strategic levels.

In summary, D4H complements the other examined systems by demonstrating how modularity, role-based access, and configurability can be integrated into globally adopted emergency management software. Its design provides additional insights into balancing flexibility, usability, and cognitive efficiency in complex, real-time systems—principles that inform the design of the curing system interface developed in this thesis.

2.3 Background

2.3.1 The ISAN project

This thesis is build on the International Standard Accident Number (ISAN) project by the Peter L. Reichertz Institute (PLRI) of Braunschweig and Hannover.

The rescue chain is typically comprised of three different information and communication technology (ICT) systems: a curing system, responding system and an alerting system. With the rise of more and more smart devices the alerting system can span all kinds of technology from a smart watch or any kind of smart wearable to a smart car or complete smart home. The responding system to such an emergency is a composite of any kind of units like: firefighter, ambulances, air or sea rescue units. Medical institutions like the hospital, a specific emergency room or a reha clinic are part of the curing system. If provided even before subject's delivery with personal data, medications or vital signs, these institutions could provide better health care (5). Today, these stages are supported by largely isolated systems, resulting in manual data re-entry, delays and information loss.

The ISAN functions as a shared unique identifier linking distributed systems without requiring tight coupling or a centralized architecture. Instead of attempting to standardize all emergency data formats, ISAN provides a minimal common reference, similar in spirit to a primary key in distributed databases. The goal of the ISAN project has two core objectives:

1. establish a fully automatic emergency alert system, particularly from IoT devices such as smart cars or wearables.
2. ensure interoperability across isolated ICT systems operated by different organizations, vendors and even countries.

This proposed unique identifier aims to work with a multiplicity of different scenarios and hardware (6).

1. Structure of the ISAN

The ISAN embeds accident metadata and a unique identifier directly into a compact string following the fixed-order paradigm. The metadata consists of:

1. time + uncertainty
2. location + uncertainty
3. altitude + uncertainty (optional)

This results in a total of seven fields, each following the tag-value paradigm. Tag and values are separated by the "|" character. The tag is a single-character literal indicating the format with "0" always indicating the preferred choice. All kinds of different formats for time, location, altitude and unique identifiers are discussed, but the preferred once are:

Value field	**Supported format**
Time	**ISO 8601b (basic)**, ISO 8601e (extended), Unixtime, RFC 5322s, RFC 5322l, ISO 8601bu
Time uncertainty	**ISO 8601bu (basic)**, ISO 8601eu (short)
Location	**ISO 6709**, Open Location Code, ISO 19160 subset
Location uncertainty	**RFC 5870**
Altitude	**ISO 6709**, ISO 4157
Altitude uncertainty	**RFC 5870**, ISO 4157

Value field	Supported format
Unique identifier	**MAC address**, bluetooth device address, international mobile equipment identity (IMEI), international mobile station equipment identity (IMSI), manufacturer serial number (MSN), vehicle identification number (VIN), random number

2. Master case index

The emergency care chain is comprised of a great number of heterogeneous systems resulting in siloed data. Looking at combined data of a patient's accident and emergency case, including data of is or her treatment and care plus corresponding outcomes in a comprehensive way across all sectors is difficult. Emergency medical services (EMS) collect rescue data, initial findings and continuous health monitoring data during a patient's transport. A patients medical data is recorded in a hospital forming an electronic health record (EHR)

2.3.2 Vue: components and composables

Modern web-based user interfaces are more and more commonly developed using component-based frameworks, which support the construction of complex interfaces through the composition of smaller, self-contained units. Component-based UI development has been widely adopted due to its benefits for modularity, maintainability, and consistency (7). Vue is a JavaScript framework that follows this paradigm and is particularly suited for developing interactive, state-driven web applications with re-usability and scalability in mind. In this thesis, Vue provides the structural foundation for implementing a user interface that must present dynamic and time-critical information in a distinct, clear and reliable way.

Components lie at the core of Vue, more specifically single file components (SFC). A SFC encapsulates a specific part of the user interface, including its visual structure, interactive behavior, and local state. By decomposing the interface into reusable components, developers can manage complexity and enforce consistent interaction patterns across the application. From a human–computer interaction (HCI) perspective, this approach aligns with principles such as consistency, visibility, and predictability, which are essential for reducing cognitive load and supporting rapid information processing (8). In emergency response systems, where users operate under stress and time pressure, such consistency enables faster recognition of interface elements and more reliable interaction.

Conceptually, a component-based user interface can be understood as a hierarchical structure, where higher-level components compose and coordinate lower-level elements, as illustrated in Figure 1.

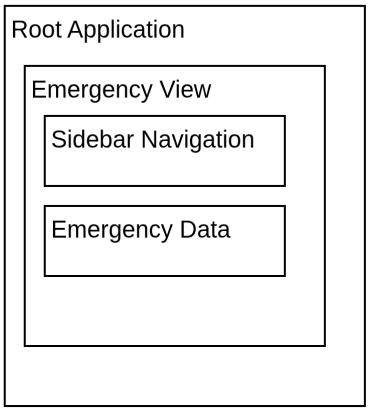

Figure 1: Conceptual hierarchy of a component-based user interface with an example in the broader context of this thesis.

While components structure the user interface, Vue further supports the separation of concerns through the use of composables. Composables are reusable functions that encapsulate stateful logic and can be shared across multiple components. Introduced as part of Vue's Composition API, this concept reflects broader software engineering principles such as separation of concerns and functional abstraction (Parnas, 1972). Composables allow application logic—such as data retrieval, state synchronization, or domain-specific behavior—to be implemented independently of the visual representation.

From an HCI-oriented software engineering perspective, composables contribute indirectly but significantly to usability. By externalizing complex logic from UI components, composables enable components to remain focused on presentation and interaction. This clearer separation supports a closer alignment between the system's internal structure and the user's mental model of the interface, an important factor for usability in safety-critical systems (Norman, 2013). Additionally, shared composables ensure consistent handling of cross-cutting concerns—such as incident context or standardized identifiers—across the interface, supporting coherent situational awareness.

The relationship between components and composables can be visualized as shown in Figure 2, where multiple components rely on shared composables for logic and state.

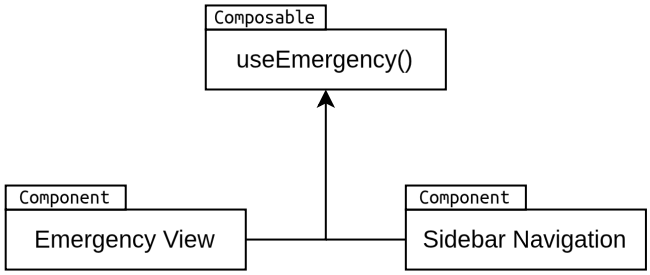

Figure 2: Reuse of stateful logic through composables across multiple SFCs.

Together, components and composables form an architectural pattern that supports modularity, scalability, and clarity in user interface development. In the context of this thesis, these concepts provide the technical and conceptual basis for designing a user interface that integrates information from multiple data sources while adhering to established HCI principles. By leveraging component-based UI design and reusable logic abstractions, the system aims to

reduce cognitive load, support rapid information access, and improve interaction efficiency in emergency response scenarios.

3 Method

3.1 Environment and technologies used

Typescript switches javascripts dynamic and weak typing with a gradual and strong typing system. Strong typing can be described like follows by Liskov and Zilles (9): “whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function.” A gradual typing system lies between a dynamic and static typing system. In the case of typescript it introduced type annotations:

```
let x: number;
```

, which result in static type checking. But it also allows the following examples:

```
let x: any;
let x: number | Car;
```

That special use-case of a variable being either a number or a `Car` object is hopefully never encountered, but the point is that in these two examples the type needs to be determined during runtime. For this typescript uses like javascript the infamous duck test: “If it walks like a duck and it quacks like a duck, then it must be a duck” (10) So even though in the end it is still in the hands of the developer to utilize the static typing properties of typescript, it also makes code more readable knowing what the type is; e.g. of an argument passed to a callback you need to implement. Additionally while CSS is not possible to workaround when designing web applications, variants like sass introduce syntactic sugar making styling easier and ensuring scalability through reusable design components. Vue offers great reactivity with a component-based architecture ensuring great modularity. As a build tool using Vite accelerates development by being a very fast build tool with optimized bundling in general while it also provides hot-module reloading which especially accelerates UI development. The templating engine of Flask is Jinja good for flexible integration between the python back-end and the local client app.

Back-end relies on Python which is widely adopted and has an extensive ecosystem. Flask offers a minimal yet flexible framework for rapid service development, while Connexion enforces OpenAPI-driven design, enabling automatic validation, documentation and maintainable APIs.

Together, this stack provides strong reactivity for fast interactive visualizations on the front end and robust workflows on the back end while enabling well-supported maintenance through comprehensive tooling.

Table 1: Technology stack

	Front-end	**Back-end**
Languages	HTML, SASS, TS, Jinja	Python
Frameworks	Vue.js (with Vite)	Flask/Connexion

3.1.1 Development setup

To streamline front-end development, hot reloading is an essential tool. Vite's development server can track changes and reloads the webpage automatically. Frameworks like Symfony have integrations and project skeletons for development and production, which configures everything. For Flask I found the project flask-vite. This project provides only beta versions yet and is maintained by one person, with only few contributions from other people. I decided to write my own helper functions for a better integration in the environment of the already existing ISAN environment and full transparency of what happens to make extending functionality easier.

Why not just start the vite development server and access it directly? The vite development server will serve the application to the client. Requests made from that app to the server are typically handled by the back end with the route resolving to the host that served the app. This could be solved by a proxy setting inside vite (see Code 1). That way requests matching a certain pattern will be sent to the back-end server or the defined server respectively. However this still doesn't account for template rendering happening in the back end before the app is served to the client. Creating DOM Nodes and filling them with content happens to some degree on the back end. The solution is to access the flask server with our back end directly instead of the vite server. Hot reloading of changes inside the back end is now natively handled by the flask server. In order for hot reloading on the front end to work we first detect whether the vite development server is running (see Code 2). If no vite development server is detected the statically built assets are included as they should for production too. Otherwise the assets will be requested from the vite development server. The function `script_entries()` (see Code 3) returns the script elements for each asset accordingly. Additionally a client for the vite development server needs to be imported. Since the same-origin policy forbids such requests, they need to be specifically allowed for the according assets via the `crossorigin` attribute.

Elaborate why it makes sense to be on the back end

vite.config.ts

```
1:  export default defineConfig({
2:    server: {
3:      proxy: {
4:        '/api': 'http://localhost:5000',
5:      },
6:    },
7:  })
```

app/vite_integration.py

```
1:  def is_vite_running() -> bool:
2:    try:
3:      requests.get(VITE_SERVER, timeout=0.2)
4:      return True
5:    except requests.RequestException:
6:      return False
```

Code 2: Dynamically checking wether the development server is responsive

app/vite_integration.py

```
def script_entries():
        vite_dev = is_vite_running()
        urls = get_script_entries(vite_dev)

        if vite_dev:
            urls.append(VITE_SERVER + '@vite/client')

        cors = "crossorigin" if vite_dev else ""

        return Markup("".join(
            f'<script type="module" src="{path}" {cors}></script>'
            for path in urls
        ))
```

Code 3: Returns the current known scripts either from the development server via crossorigin or with a direct path to the built assets.

3.2 Architecture

3.2.1 General architecture

In general there are two different ways to approach this: via back end or front end rendering. Typically rendering is better handled in the front end, our browsers are well-optimized for that today (WebGL, WebGPU). Moreover the reactivity is important on the front end anyways. So reactivity handled by back-end rendering means the back-end needs to send and re-render the whole page again and again. Frameworks like Vue are optimized updating only the important changed reactive parts of a website. Updating something small like a text would result in requesting the page again from the back end being re-rendered with new data. Instead it is more robust to let Vue handle rendering and updating the page with the single entry point being the `index.html` (single page application). Data requests and writes can be send to the back end asynchronously being potentially more complicated. But for the curing systems data writes are not a typical use-case as its main purpose is to **display** data collected from different data silos. Therefore a SPA (single page application) with front-end rendering is the architecture that was used to develop the curing system.

The general flow as illustrated in Figure 3 includes the following steps:

1. SPA shell requests module list after an emergency was selected and renders sidebar accordingly
2. user clicks module -> `ModuleContainer` dynamically imports Vue component and fetches data if configured to do so
3. `ModuleContainer` renders UI using the data, imports any required JS/TS libraries internally and let's the Vue module component render it's contents

Add route definitions

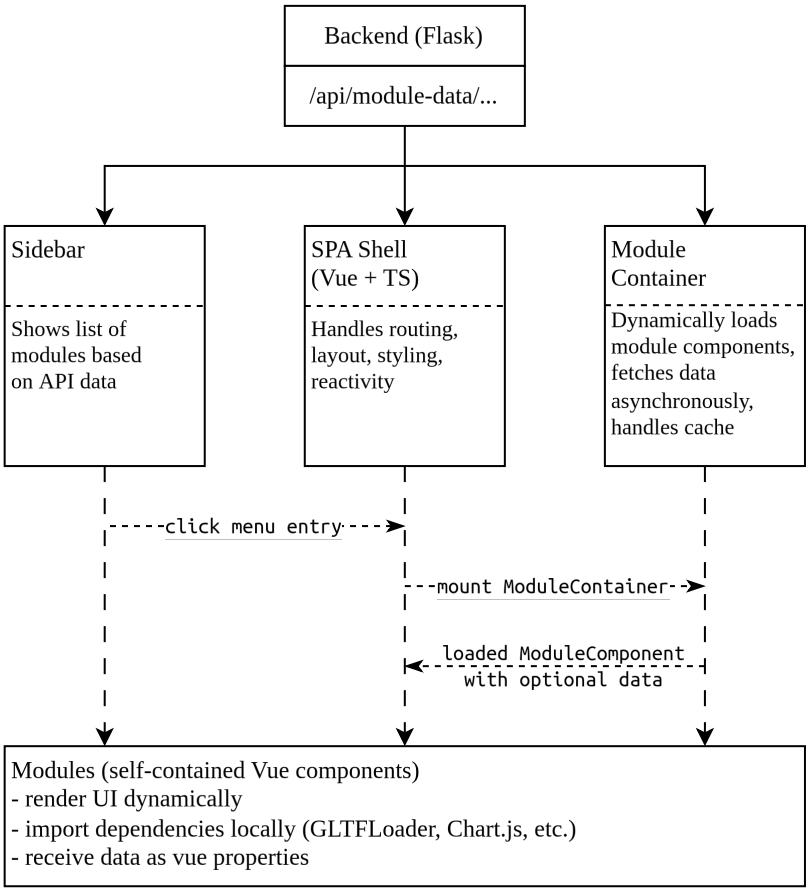

Figure 3: Graph showing the workflow inside the SPA to load the modules.

3.2.2 Front end - back end communication

For communication between the front end and back end data two different types are needed: static data and dynamic data. It is important to differentiate for each information what its scope is.

1. Static data

Static data represents information that is loaded once per initial request of the page. Without any user specific interaction the back end embeds data into the HTML via Jinja's rendering like follows:

```
<script id="py-data" type="application/json">{{ py_data|safe }}</script>
```

Usually Jinja escapes HTML characters for safety reasons. For that reason and only because we know what exactly we passed Jinja to fill in we specify it as safe for Jinja so that the json string remains healthy. To ease declaring the layout of this data, `dataclasses` are used in python and `zod` in typescript, where both declarations need to represent the same structure. They are located in files named `py_data.py` (see Code 4) and `py_data.ts` (see Code 5) respectively as what they really represent is data sent from the python back end, received by the front end.

While the back end code provides functionality to output the code into json via the class `JsonData` which every `dataclass` object inherits from, the front end code implements the

src/py_data.py

```
1:  class JsonData:
2:      def json(self):
3:          return orjson.dumps(self).decode('utf-8')
4:
5:  @dataclass
6:  class ModuleType(JsonData):
7:      module_type: str
8:      vue_component_file: str
9:
10: @dataclass
11: class ModuleInstance(JsonData):
12:     name: str
13:     module_type: str
14:     fetch_init_data: bool = False
15:     refresh_interval_ms: int = -1
16:     icon_path: str = ""
17:
18: @dataclass
19: class EmergencySchema:
20:     name: str
21:     modules: list[ModuleInstance]
22:
23: @dataclass
24: class PyData(JsonData):
25:     emergencies: list[EmergencySchema]
26:     module_types: dict[str, ModuleType]
```

Code 4: The python file (without imports) containing the declarations for data sent to the front end. The `py_data.ts` file needs to reflect this data layout.

function `getPyData()` in line 28 to 35. The previously mentioned Jinja rendered variable put into the DOM Element `py-data` will be queried here. Its contents parsed from json are then validated by `zod` according to the declared data layout. Furthermore the result is cached (see line 26), hence subsequent calls do not have to undergo parsing again.

Almost at every point on the front end this data can be expected to be available. Therefore it represents critical data that is needed for basic functionality. For this project this means the app needs to know what emergencies exist and what modules it can show for each one.

2. Dynamic data

In contrast dynamic data is only required after certain user interactions. In this case for example once a module is requested to be loaded by the user, the `ModuleContainer` will check whether the module type is defined to request dynamic data from the back end. Figure 4 visualizes what routes lead to what type of data being send to the front end. Once the route `module/:moduleKey` is appended to the index route the `ModuleContainer` will try to fetch data from the back end. If cached data is present it will show this data immediately until the asynchronous fetch comes back, otherwise it indicates that the module is loading. When a module is defined to request such data, we treat this data as critical to that module and therefore not mount any of its elements until that data is fetched successfully. It is important to understand

app/py_data.ts

```
export const ModuleTypeSchema = z.object({
    module_type: z.string(),
    vue_component_file: z.string(),
});

export const ModuleInstanceSchema = z.object({
    name: z.string(),
    module_type: z.string(),
    fetch_init_data: z.boolean().default(false),
    refresh_interval_ms: z.number().default(-1),
    icon_path: z.string(),
});

export const EmergencySchema = z.object({
    name: z.string(),
    modules: z.array(ModuleInstanceSchema),
});

const PyDataSchema = z.object({
    emergencies: z.array(EmergencySchema),
    module_types: z.record(z.string(), ModuleTypeSchema),
});

export type PyData = z.infer<typeof PyDataSchema>;

let cachedData: PyData | null = null;

export function getPyData(): PyData {
    if (cachedData == null) {
  cachedData = PyDataSchema.parse(
      JSON.parse(document.getElementById('py-data')?.innerText ?? '{}')
  );
    }
    return cachedData;
}
```

Code 5: The typescript file (without imports) containing declarations of data received from the back end. These declarations reflect the data layout defined by the back end in file `py_data.py`.

that the semantics of that fetched data cannot be validated by the `ModuleContainer`. Due to this it needs to be validated by the module itself. If successful it will replace the stale data with the received fresh data automatically. Noteworthy is that a module can be shown on its own page or as a tile on the dashboard. In both cases the same cache will be used. The use-case of adding a module tile to the dasboard and then wanting to fullscreen this module on its own page, therefore results in immediate presentation of that same data. Additionally the user can specify whether the data needs to be continously refreshed defined by a time interval i in miliseconds. The same procedure will be repeated every i miliseconds accordingly, while the module is mounted on the dashboard or on its own page.

3.2.3 Modular system

The main purpose of implementing a modular system is to remove redundancy. While code redundancy is hailed as one of the biggest evils in programming, the most important

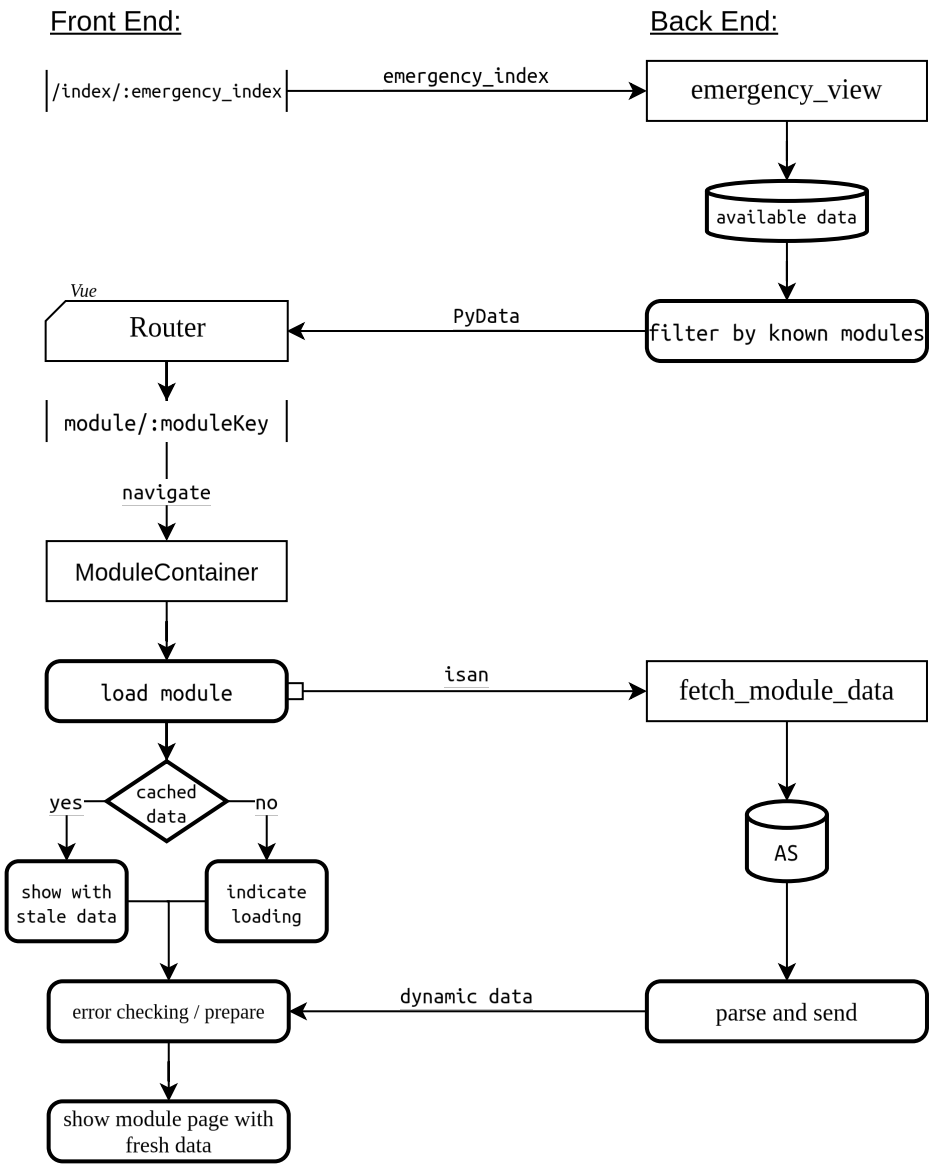

Figure 4:

redundancy to mitigate in this project is the redundancy of implementing complex processes managing UI/UX specific tasks. As a core property of rather complex tasks comes the necessity of configuring some of their behavior. Thus it is necessary to define configurable properties and make them easy to configure for any developer who wants to implement a module.

The available data received by the alerting system is categorized by a module type and a specific module name, for example specific module names (or instances as we are going to call it) could be “doorcode” or “DAR” (digital accident report), but both have the same module type “text”. The following properties can be defined on the scope of a module instance or the whole type.

Property name	**Description**	**Default**
icon	Server root path to the .svg file containing the desired icon to represent that module.	fa-diamond

require-ments and purpose

1. Module container

The following is how the ModuleContainer.vue looks like:

```
<template>
    <component
  v-if="state === LoadState.Ready || state === LoadState.StaleData"
  :is="module_component"
  :data="module_data"
    />
    <div v-else-if="state === LoadState.LoadingModule">
  Loading module..
    </div>
    <div v-else-if="state === LoadState.LoadingData">
  Loading data...
    </div>
    <div v-else-if="state === LoadState.Error">
  <strong>Error:</strong> {{ error_message }}
    </div>
</template>
<script lang="ts" setup>
import { useModuleLoader, LoadState } from '../module_loader';

const { module_component, module_data, state, error_message} = useModuleLoader();

</script>
```

Most of the loading logic is put away in the `ModuleLoader` composable. The component itself only manages the rendering according to what `LoadState` has been set. The `ModuleLoader` implements a prefetching method for the loading the vue component and for fetching the according module data separately. Moreover the vue component and module data are cached. While the vue component is a static asset and will not change during a client session, the module data will get outdated (`LoadState.StaleData`). Therefore fresh data will always be fetched, but if cached data is available the module will already be shown.

The ModuleContainer has a dynamic `component` object from vue. The container will read in the url parameter and will try to import the correct module, which will then be passed to the `component` object. Then data is requested from the backend for that module, which gets automatically passed into the component.

Adding a module might include the following files:

Filename	Description	Obligatory
`<module-name>.vue`	The core SFC of the module that implements the logic, design and interactivity to fulfill the purpose this module is supposed to serve.	Y
`<module-name>.ts`	Some logic regarding e.g. subtasks or re-usable logic through different components might be recommended to move to an external file, which can then be imported.	N
`<module-name>.py`	For adding back-end data responses to the SFC at the moment always requires at least expanding the `fetch_module_data` function inside the `app.py`. But it is recommended to outsource any deeper logic in a separate python file.	N

The vue file is needed, because the `ModuleContainer.vue` tries to load this file, since it is the main entry point to render a module.

2. Defining a module

3.3 Design

3.3.1 Top Level Design

- fairly simple
- sidebar: for modules (and dashboard?)
- topbar: emergency selection (and dashboard?)

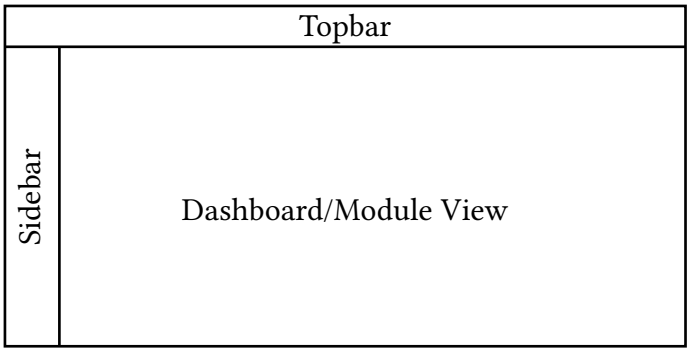

Figure 5: Top Level Design

Sidebar and topbar are represented by a Vue component respectively. They will and should always be visible, since they are essential to navigate between visualizations of desired information. The dashboard and module view will take most of the screen as they will show the emergency information. The module view is supposed to be as generic as possible to integrate different kind of projects. This way extending functionality at a later point should not result in rewriting the code base of this project.

3.3.2 Colors and contrast

Defining colors throughout even a medium-sized application can result in a mess of same purpose colors defined at multiple locations in code resulting difficult maintainability and even inconsistent colors. A central file defining the color scheme ensures maintainability and scalability. If desired a different theme could be added, through a media query for dark themes for example, just in this file replacing the values for the same variables; no further changes needed. In this thesis the file `colors.scss` (see Code 7) defines all the used colors connecting them with the purpose-named variables.

@css/colors.scss

```scss
$black-0: #212529;
$black-1: #2b3035;

$grey-0: #343a40;
$grey-1: #444a50;

$white-0: #e2e2e6;

$green: #080;
$green-darker: #060;
$red: #CC1213;

$tile-bg: #222;

$acc-0: #ffa500;
$acc-1: #cf7500;


:root {
  --body-bg: #{$black-0};

  --text-color: #{$white-0};

  --sidebar-bg: #{$black-1};
  --sidebar-color: #{$white-0};
  --sidebar-btn-hover: #{$grey-0};
  --sidebar-btn-active: #{$grey-1};

  --content-view-header-bg: #{$black-1};
  --content-view-bg: #{$black-0};
  --dropzone: #{$green-darker};

  --tile-border: #{$green-darker};
  --tile-btn-hover: #{$green-darker};
  --tile-bg: #{$tile-bg};

  --warn: #{$red};

  --acc: #{$acc-0};
  --acc-secondary: #{$acc-1};
  --acc-foreground: #{$black-0};
}
```

Code 7: Central scss file defining color variables used throughout the website.

What follows is a contrast analysis of those colors that overlap and contrast is needed to provide important information. The suggested contrast ratio by WCAG 2.1 Level AAA for text is 7:1 (11). Contrast ratio is defined as:

$$L_1 + 0.5 / L_2 + 0.5\,,\ L_1 > L_2$$

where L_1 and L_2 are the relative luminance. The relative luminance is computed as follows:

$$L = 0.2126 \cdot R + 0.7152 \cdot G + 0.0722 \cdot B$$

Each of the channels R, G, B in this formula are computed by taking the original normalized color channel $\overline{c}$ and applying the following computation to them:

$$C = \begin{cases} \overline{c}/12.902\,, & \overline{c} \leq 0.04045 \\ (\overline{c}+0.055/1.055)^{2.4}\,, & \text{otherwise} \end{cases}$$

The important color variables used for text are `--text-color` and `--acc-foreground`. Both of them need to be compared to the background colors they are displayed on. As the css-style variables are purpose-named and hence would result in redundant comparisons the contrast ratio is computed relative to the color values that are used for background purposes.

	$black-0	$black-1	$grey-0	$grey-1	$acc-0	$acc-1
--text-color:	11.94	10.31	8.91	6.94	1.53	2.62
--acc-foreground:	0	1.16	1.34	1.72	7.81	4.55

Figure 6: Table of color contrast ratios. The rows are color variables used for foreground / text, while the columns represent the color values used as a background throughout the website.

4 Results

4.1 Emergency select page

This page is only visible if there are more then one registered ISANs received from the Responding System. If there was only one emergency received it will automatically forward to the emergency view page. Otherwise it provides rather big buttons showing the raw ISAN.

4.2 Emergency view page

This page is the one that is designed as the main page where the user will be. On the top is a menu with an orange background, which on the left side shows a reduced version of the currently selected ISAN only showing date, time and location and on the right shows weather data for according emergency location. The current (reduced) ISAN can be clicked to show a dropdown with all known current emergencies.

While the top bar provides more meta information, the left bar provides navigation regarding the selected emergency. Therefore it shows all available modules, meaning modules for which data has been received and modules for received data which have a module front end definition. When entering the page, the list only shows big icons of each module, but it can be expanded to also show the module name. To open a module the according icon or module name can be clicked. Opening a module will show its representation in the main area at the center of the screen, but the top and left navigation bars will stay the same. Moreover the current opened module (or dashboard) will have a different background indicating the current location. No module will be opened automatically. Instead an empty dashboard (if the page is visited the first time) is shown.

The dashboard has its own top bar with the headline “Dashboard” and a big button on the right, which opens a dropdown to add or remove (indicated by the checkbox) a module to the dashboard as a tile. Furthermore the dashboard shows green zones. Those green zones are dropzones. While not automatically when added to the dashboard, a module tile will snap to the according zone when dropped onto. The initial dropzones only allow to split for half of the screen and not a quarter. For that a tile can be dropped onto another tile that is already pinned to one of the dropzones: if done in the middle of that tile it will be replaced, but if done left or right for vertical splitting or top or bottom for horizontal splitting it will split the previous tile in half and place the dragged one accordingly to the location it was dragged to.

Each module tile has two buttons: the maximize button on the top left and the close button on the top right. The close button removes that tile from the dashboard and the maximize button opens the according module page. If the page is closed the current tile arrangement will be cached so that once it is reopened the previous arrangement is loaded and displayed. In-between those buttons it shows the name of the module this tile belongs to.

5 Conclusion

Bibliography

1. ALEKVS. What Is ISO 9241? A Complete Guide to HCI and Usability Standards [Internet]. 2025 [cited 2026 Jan 13]. Available from: https://www.alekvs.com/what-is-iso-9241-a-complete-guide-to-hci-and-usability-standards/
2. Kring Friedhelm. Die Grundsätze der Dialoggestaltung nach ISO 9241-110 [Internet]. 2025 [cited 2026 Jan 13]. Available from: https://www.weka-manager-ce.de/betriebsanleitung/grundsaetze-dialoggestaltung-iso-9241-110/
3. ISO 9241-110:2020(en) Ergonomics of human-system interaction - Part 110: Interaction principles [Internet]. International Organization for Standardization (ISO); [cited 2026 Jan 13]. Available from: https://www.iso.org/obp/ui/#iso:std:iso:9241:-110:ed-2:v1:en
4. Uncomplicated Crisis & Emergency Management Software | D4H [Internet]. [cited 2026 Jan 13]. Available from: https://www.d4h.com/
5. Haghi M, Barakat R, Spicher N, Heinrich C, Jageniak J, Öktem GS, et al. Automatic information exchange in the early rescue chain using the International Standard Accident Number (ISAN). In: Healthcare. 2021. p. 996.
6. Spicher N, Barakat R, Wang J, Haghi M, Jagieniak J, Öktem GS, et al. Proposing an international standard accident number for interconnecting information and communication technology systems of the rescue chain. Methods of information in medicine. 2021 ;60(S1):e20–31.
7. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley; 1995.
8. Nielsen J. Usability Engineering. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1994.
9. Liskov B, Zilles S. Programming with abstract data types. In: Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages. Santa Monica, California, USA: Association for Computing Machinery; 1974. pp. 50–9.
10. Mohammed El Aouri. Duck Typing in JavaScript [Internet]. 2025 [cited 2026 Jan 12]. Available from: https://medium.com/@mohammedelaouri/duck-typing-in-javascript-3122786ddcf7
11. Contrast (Enhanced) (Level AAA) [Internet]. W3C; [cited 2026 Jan 13]. Available from: https://www.w3.org/WAI/WCAG21/Understanding/contrast-enhanced.html

Statutory Declaration

I hereby declare in lieu of an oath that I have written this Master's Thesis “Designing a Modular Web Interface for the ISAN Curing System: Enhancing Information Accessibility in Emergency Situations” independently and that I have cited all sources and aids used in full and that the thesis has not already been submitted as an examination paper.

Braunschweig, 15.01.2026

(signature with first- and lastname)