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

Project Thesis

Nikkel Lennart Heesen

Braunschweig, 18.01.2026

**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 Introduction

## 1.1 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. Shorter emergency response times are significantly associated with reduced mortality in urgent medical cases, and real-time data acquisition and rapid alerting systems can improve response capabilities and potentially save lives by enabling faster, more informed decision-making by first responders (1). This emphasize that effective emergency response depends not only on operational capabilities but also stressing the importance of how information is provided to human operators and to not overload them with informations (2).

At the same time, substantial progress has been made in the digitalization of emergency response infrastructures and the medical sector in general. 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 (3). 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.

The user interface represents the critical link between complex backend systems and human decision-makers. Research regarding human-computer interaction (HCI) shows tha tuser interface design has measurable effects on user performance, error rates, and cognitive workload. Variations in interface design have been demonstrated to influence task completion speed and accuracy; simpler and better-organized interfaces using recognizable patterns tend to reduce cognitive load and enhance performance compared to more complex and cluttered designs (4).

In emergency contexts, responders operate under stress and time constraints that might limit working memory and increase the risk of errors. Poorly designed user interfaces can amplify these limitations by presenting information in an unstructured or unintuitive manner, leading to delayed responses or even misinterpretation of critical data.

Emergency responders are often required to consult multiple systems to obtain a complete picture of an incident, increasing cognitive effort and response time (2). 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 accessibility requirements, the system aims to support faster, intuitive and

more informed decision-making. Ultimately, this work seeks to develop a thoughtful interface design that can contribute to more effective emergency response, while laying a foundation for easily adding critical information without the need to re-iterate accessibility patterns and their implementation from a systematic point of view.

## 1.2 Related work

### 1.2.1 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 (5). 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 (6). 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 (7). 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 Germany, 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 (8). EN 301 549 incorporates the Web Content Accessibility Guidelines (WCAG), particularly in Chapter 9, which addresses web-based user interfaces (9).

WCAG defines success criteria across different conformance levels (A, AA, and AAA), with EN 301 549 encouraging AAA compliance where feasible, while also stating in chapter 9.5 that it is not recommended to be required as a general policy. 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 impair-

ments, W3C guidelines also list situational impairments as a barrier (10) and HCI research emphasizes reduced analytical thinking that are common in emergency operations (11).

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 or mention it only as a side note. 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, implementing and adapting them within the context of an emergency-focused web application interface.

### 1.2.2 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 (12). 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.

## 1.3 Background

### 1.3.1 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 (13). 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 (3).

*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 |
| 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 |

### 1.3.2 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 (14). 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. For that it is essentially comprised of three parts: the template, script and style. Where the template defines the document structure, the script any logic and state and style can be any type of css or derivative of css. By decomposing the interface into reusable components, developers can manage complexity and enforce consistent interaction patterns across the application. This approach aligns with principles such as consistency, visibility, and predictability, which are essential for reducing cognitive load and supporting rapid information processing (15). 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.

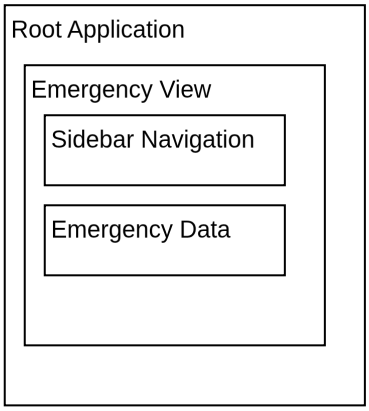


*Figure 1: Conceptual hierarchy of a component-based user interface with an example in the broader context of this thesis.*

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. 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.

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.

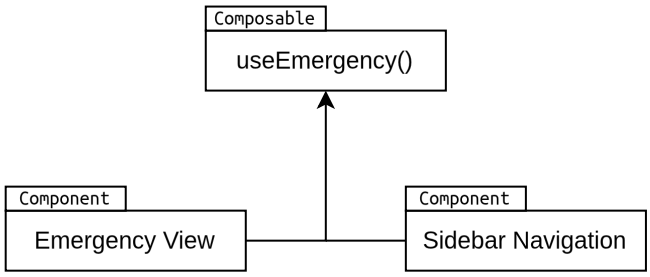


*Figure 2: Reuse of stateful logic through composables across multiple SFCs.*

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.

# 2 Method

## 2.1 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 (16): “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” (17). 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 |

### 2.1.1 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.*

## 2.2 Design

### 2.2.1 Top Level Design

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

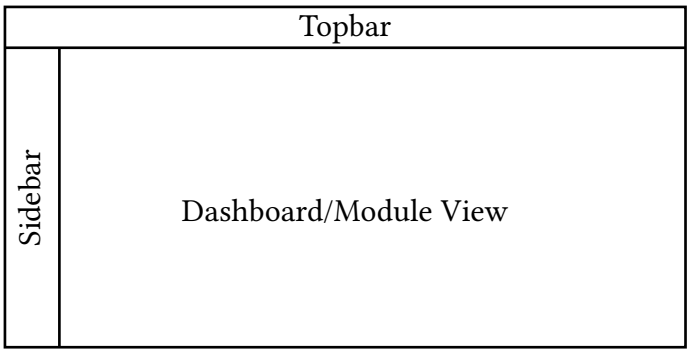


*Figure 3: Top Level Design*

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 important emergency information.

### 2.2.2 Color contrast design

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 4) defines all the used colors connecting them with the purpose-named variables.

**@css/colors.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;
$acc-blue:     #0d2edd;

: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};
  --acc-alt:                 #{$acc-blue};
}
```

*Code 4: 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. WCAG 2.1 defines contrast ratio as follows:

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

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 $\bar{c}$ and applying the following computation to them:

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

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 |

| | $acc-blue |
|---|---|
| --text-color: | 8.56 |

*Figure 4: 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.*

According to these contrasts, some color choices have been avoided. The `--acc-foreground` has been for example added to avoid the weak contrast of the normal text color on any accentuated background.

## 2.3 Implementation

### 2.3.1 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 5 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

### 2.3.2 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.

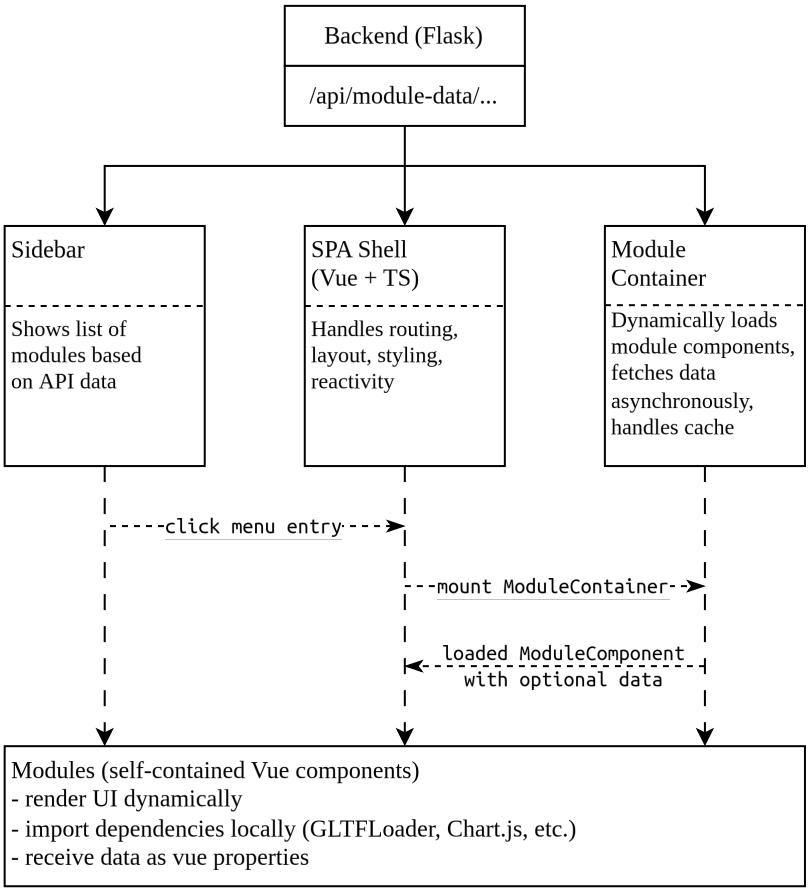


*Figure 5: Graph showing the workflow inside the SPA to load the modules.*

*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 5) and `py_data.ts` (see Code 6) 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 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.

**src/py_data.py**

```python
class JsonData:
    def json(self):
        return orjson.dumps(self).decode('utf-8')

@dataclass
class ModuleType(JsonData):
    module_type: str
    vue_component_file: str

@dataclass
class ModuleInstance(JsonData):
    name: str
    module_type: str
    fetch_init_data: bool = False
    refresh_interval_ms: int = -1
    icon_path: str = ""

@dataclass
class EmergencySchema:
    name: str
    modules: list[ModuleInstance]

@dataclass
class PyData(JsonData):
    emergencies: list[EmergencySchema]
    module_types: dict[str, ModuleType]
```

*Code 5: 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.*

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 6 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 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-

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 6: 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`.*

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.

### 2.3.3 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 redundancy to mitigate in this project is the one of implementing complex processes managing UI/UX specific tasks. As a core property of rather complex tasks that are re-used, comes the necessity of some of their behavior being configurable. Thus it is necessary to differentiate

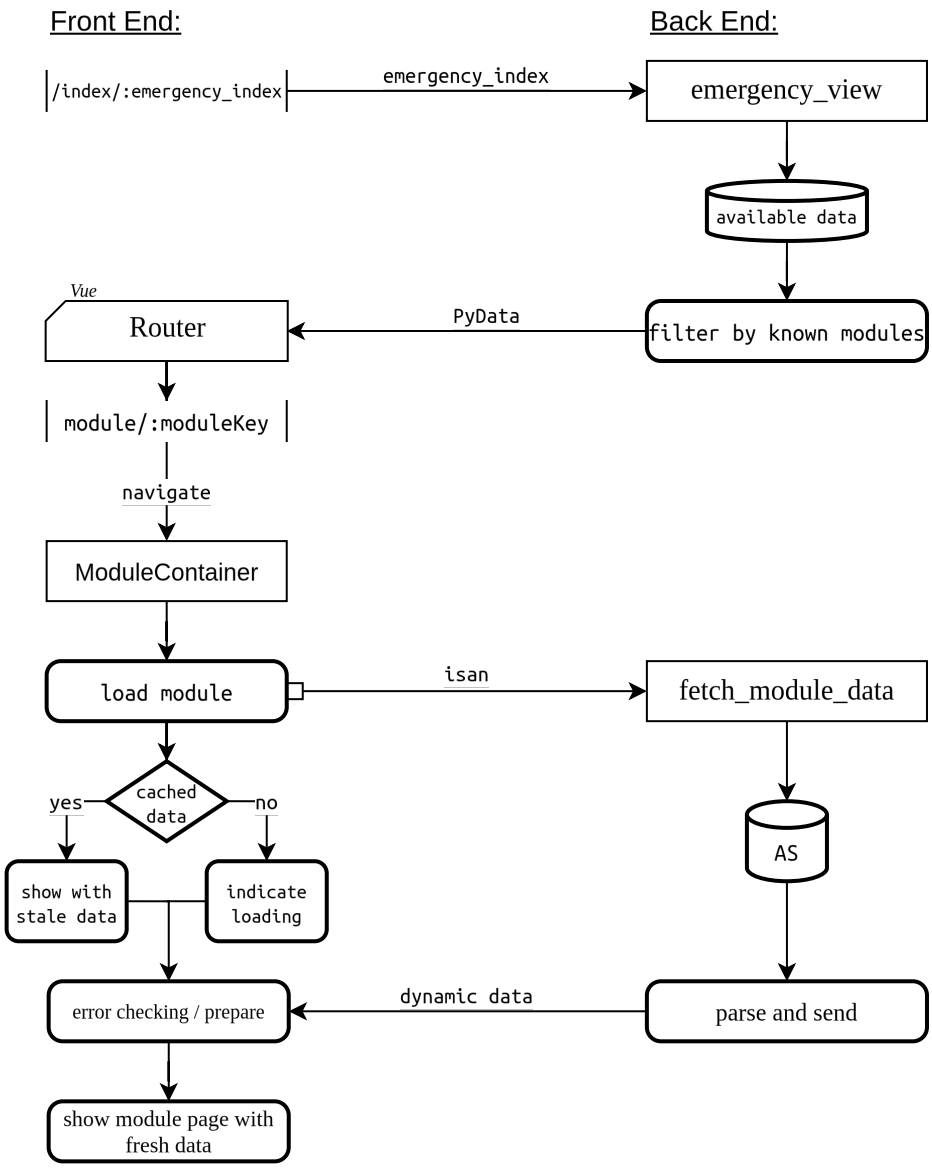


*Figure 6:*

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_path | Server root path to the .svg file containing the desired icon to represent that module. | fa-diamond |
| fetch_init_data | If set to `true`, data will be requested from the back end as explained earlier. | false |

| Property name | Description | Default |
|---|---|---|
| refresh_interval_ms | Only read if `fetch_init_data` is set to true. The interval in miliseconds in which the data will be refreshed from the back end. | −1 |

*1. Module container*

The whole implementation of the `ModuleContainer.vue` is too big to include it here. The heart of it is the `module_loader.ts` which implements the asynchronous loading process and spans over 150 lines. Instead Code 7 just shows the `template` implementation of the `ModuleContainer.vue`, which demonstrates what is shown at what state of loading.

**app/components/ModuleContainer.vue**

```
1:
2:  <template>
3:      <div v-if="handle?.data_state === DataLoadState.Stale">
4:          Stale Data
5:      </div>
6:      <div class="d-flex flex-column">
7:          <header><h1>{{ module_name }}</h1></header>
8:          <component
9:                v-if="handle &&
10:                     handle.component_state === ComponentLoadState.Loaded &&
11:                     showable_data_state"
12:               :is="handle.component"
13:               v-bind="handle.data ? { data: handle.data } : {}"
14:         />
15:     </div>
16:     <div v-if="handle?.component_state === ComponentLoadState.Loading">
17:         Loading module..
18:     </div>
19:     <div v-if="handle?.data_state === DataLoadState.Loading">
20:         Loading data...
21:     </div>
22:     <div v-if="
23:         handle &&
24:         (handle.component_state === ComponentLoadState.Error ||
25:         handle.data_state === DataLoadState.Error)"
26:     >
27:         <strong>Error:</strong> {{ handle.error_message }}
28:     </div>
29: </template>
```

*Code 7: The template part of the module container SFC.*

Most important here is Vue's `component` element (see Code 7, line 8), which will be whatever component is set to the `:is` property. So once the SFC representing the desired module is loaded, `handle.component` will be set and the `component` element can be shown as indicated by the condition in `v-if`.

Most of the loading logic is put away in the `module_loader.ts` composable. The component itself only manages the rendering according to what `DataLoadState` and `ComponentLoadState`

has been set. The `module_loader.ts` implements a `load_module` method for importing/loading the Vue component file and for fetching the according module data separately if desired. 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 can get outdated and if defined by the previously discussed property `refresh_interval_ms` will be refreshed.

To determine what module is currently desired the `ModuleContainer.vue` will determine the module name and type from the current route parameter `moduleKey`. The container will then try to import the correct module, which is after successful loading set to the `component` object. If data has been requested and received it gets automatically passed into the component via a conditional `v-bind`.

*2. Defining a module*

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 type. Hence it is necessary to point the front end to that file for a specific module type. For that the back end has a global variable `registered_modules`, which can be expanded by a `ModuleType` as follows:

```
registered_modules = {
    "interactive_map": ModuleType("interactive_map", "modules/interactive_map/Map.vue"),
    ...
}
```

This will correlate the implemented SFC `Map.vue` with the module type “interactive_map”

Additionally the discussed properties can be configured in the file `module_defaults.yaml`. YAML is a more readable version of json. The top keys of that file are `module_instance_defaults` and `module_type_defaults`. When defining a default for a whole module type, it can be done by setting the desired property for the module type as a key under `module_type_defaults`. For example to set an icon for all modules that are an instance of the module type `text`:

```yaml
module_type_defaults:
  text:
    icon_path: "/static/icons/text.svg"
```

Setting a property for a specific module instance is similar only that the instance name needs to be specified before the property name. The following example will set a different icon for the module instance `doorcode` of type text:

```yaml
module_instance_defaults:
  text:
    Doorcode:
      icon_path: "/static/icons/doorcode.svg"
```

With these examples configuration, any `text` module that has no further specified icon, will always use the `text.svg`. The exception being the `doorcode` module here, which is of type `text`, but because of the instance default being set to a different value, the type default will be overwritten.

### 2.3.4 Dashboard

The heart of this project is the dashboard, which is why most of the code lies here ( 1000 lines). The goal is to give the user the possibility to arrange all available information in a way he or she wants to. To fulfill that purpose each module can be added as a tile to the dashboard. This tile can then either get attached to a `dropzone` or just float anywhere where the user desires it to be. If tiles overlap, clicking a tile will always bring it to the top. Those tiles will give its module less space for the same data. Therefore modules should implement different visualizations, filtering their visualized data by importance in regards to how much space they have. This can be done via media queries for example.

The dashboard provides at most 4 `dropzones` to which only one tile each can be attached to. However the initial `dropzones` only have the capability to attach to one half of the dashboard area (horizontally or vertically). This is related to the size of `dropzones` on a smaller touch pad and trying to keep splitting simple. There would be simply no need to artificially reduce the screen space a tile takes before there is the need to attach another tile for viewing its information. Once a tile is attached to one half, another tile can be dragged onto:

- its center to replace that tile
- the left or right third of a tile to split vertically on a horizontal tile
- the top or bottom third of a tile to split it horizontally on a vertical tile

The implementation utilizes `pointerdown`, `pointermove` and `pointerup` events. These events combine mouse button events with touch pad events (18). To differentiate whether a tile is currently being dragged or has only been clicked to bring it to the foreground, for every `pointermove` event it is checked whether the pointer movement exceeds a certain threshold:

```
const dx = evt.clientX - d.start_x;
const dy = evt.clientY - d.start_y;
if (!d.moved && Math.hypot(dx, dy) > DRAG_THRESHOLD) {
```

```
    d.moved = true;
}
```

This is stored inside the `dragging` variable, which also stores the following dragging state information:

| Variable | Purpose |
|---|---|
| `moved` | Initially false, but set to true once the `DRAG_THRESHOLD` is exceeded. Determines what happens once `pointerup` is triggered. |
| `attaching_disabled` | Initially false, but once the user presses `Escape` the currently dragged tile will not attach to any `dropzone` and can be moved anywhere. |
| `index` | The index of the tile being dragged inside the main array of all tiles that currently need to be rendered by Vue. |

Since arranging these tiles to the users preference takes some time, the layout is cached and automatically reloaded. The `LayoutState` is comprised of two variables:

```
interface LayoutState {
    tiles: Tile[];
    quadrant_occupation: QuadrantOccupation;
};
```

So it stores the tiles, which hold their exact position if floating, and the current quadrant occupation to be able to render the attached tiles too. The `QuadrantOccupation` is a simple type that stores the tile index for each quadrant:

```
export interface QuadrantOccupation {
    left_top: number | null;
    left_bottom: number | null;
    right_top: number | null;
    right_bottom: number | null;
}
```

Each `dropzone` correlates to a certain quadrant or half. Dropping a tile on a `dropzone` correlating to a half results in two quadrants being set to that tile's index as opposed to one if the `dropzone` only identifies one quarter of the dashboard. To differentiate to what area a `dropzone` belongs to, the `DZTargetDirection` is added as a property:

```
export type DZTargetDirection = "left" | "right" | "top" | "bottom" | "whole" | "none";
```

The `dropzone` has its own type:

```
export interface DropZone {
    zone: AABB;
    target: AABB;
    direction: DZTargetDirection[];
    preview: boolean;
}
```

The `zone` defines coordinates for the area that needs to be hovered to activate the `dropzone` as a candidate once the tile is dropped onto it. If a tile is attached to a `dropzone` the according

instance is set as a property on that tile. In that case the coordinates of the `target` property are used to render that tile. The type is treated as a single source of truth, which is why the `preview` field is set to true once the `dropzone` is hovered. The template is implemented so that a preview for that `dropzone` will be rendered if `preview` is set to true. Coming back to the target direction, the field `direction` is desired to not guess the quadrant based on actual coordinates, but on actual semantics and definitions. The property is an array to represent multiple things:

- array containing one element, represents a half split at the according position
- array containing two elements, represents a quadrant at the according position

Once a tile is attached to a `dropzone` the according quadrant or two quadrants need to be set to that tile's index. Afterwards the available `dropzone` instances need to be updated for the remaining available quadrants. The following are some of `dropzone` permutations with attached tiles that are possible:

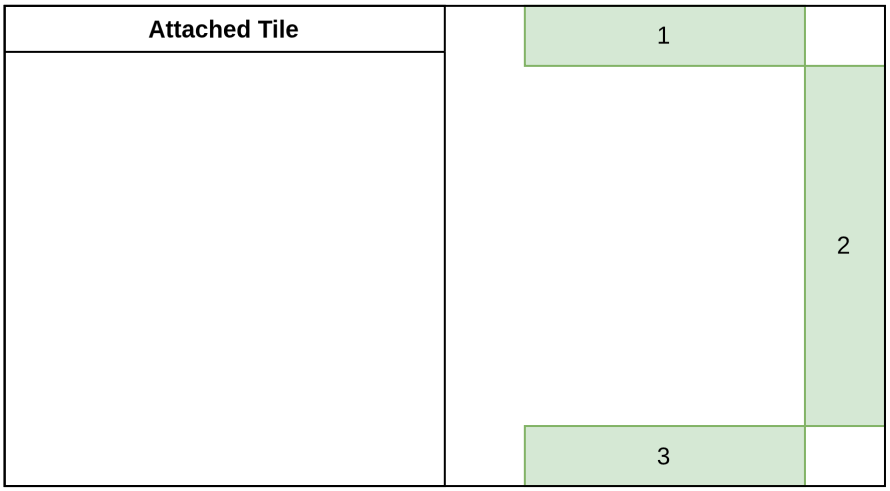


*Figure 7: A tile has been attached to the left dropzone, making it span the left half. The recomputed dropzone 2 represents the right half, while 1 and 3 represent the top and bottom right quadrant respectively.*

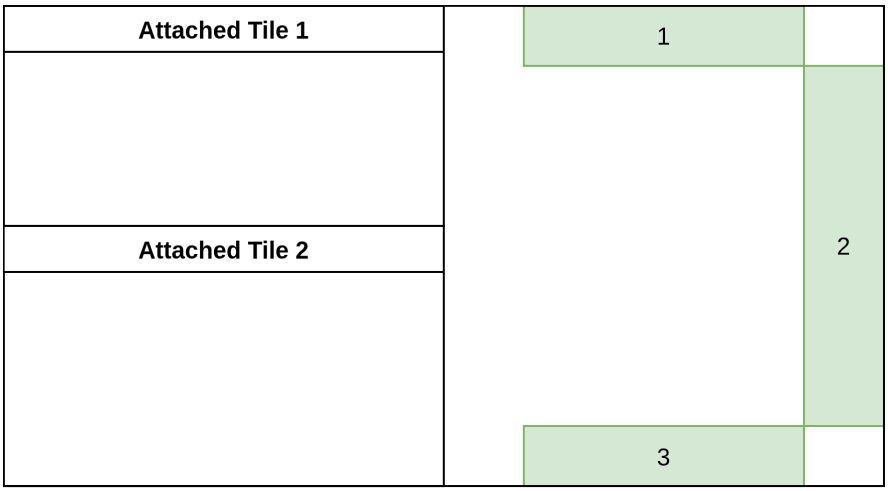


*Figure 8: This state could be achieved if a new tile has been dropped onto the bottom third of the attached tile in Figure 7. No recomputation of dropzones needed.*

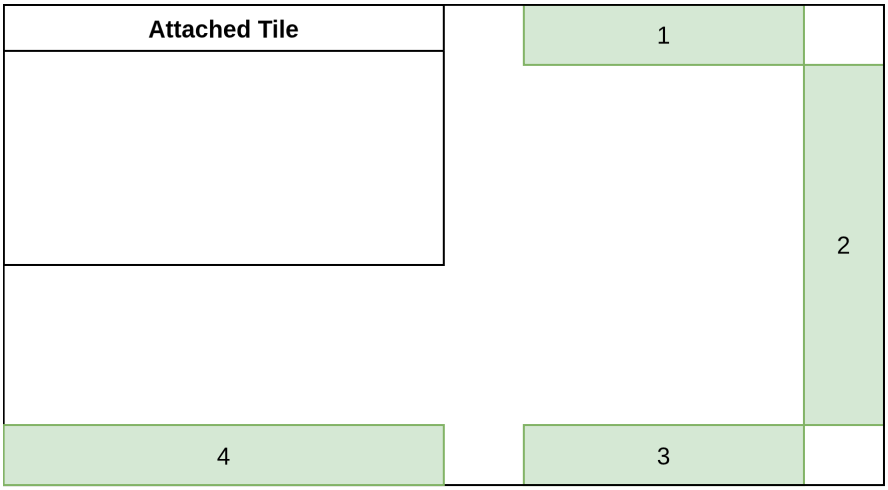


*Figure 9: If the bottom attached tile in Figure 8 is removed a dropzone needs to added specifically for that quadrant.*

# 3 Results

## 3.1 Emergency select page

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


*Image 10: Screenshot showing the page where all available emergencies will be listed and can be selected.*

## 3.2 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 side bar provides navigation within information 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 next to it) 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.

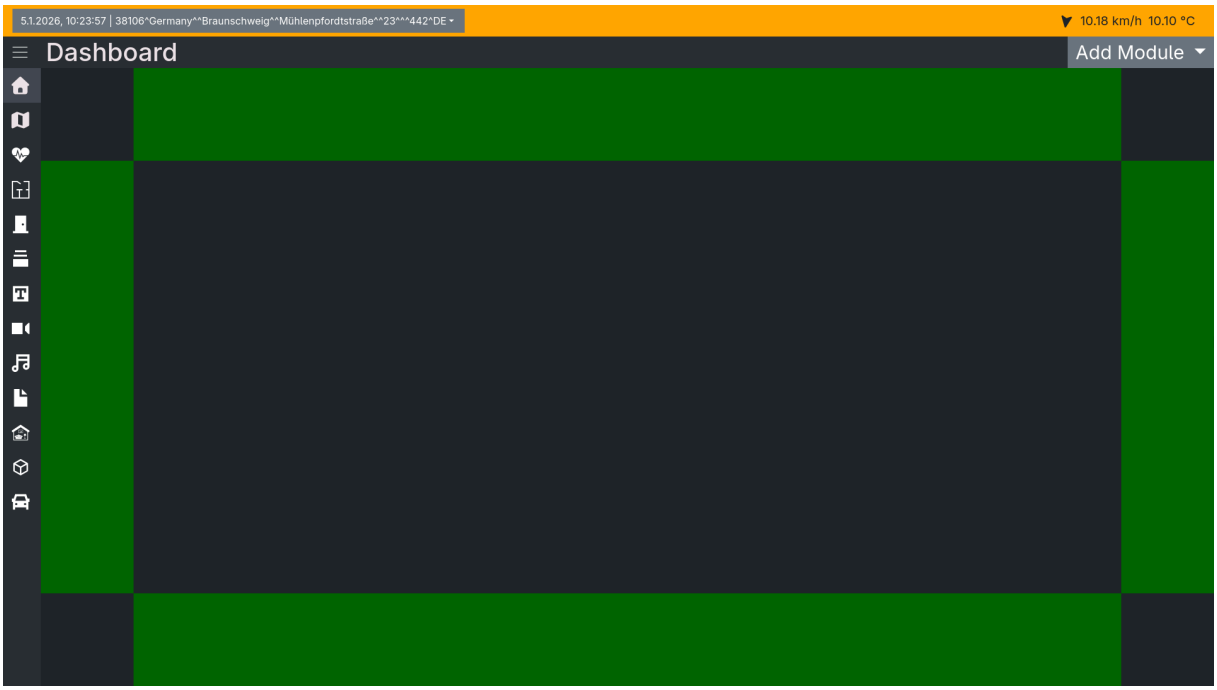


*Image 11: Screenshot of the core page of this project, where all the data for the selected emergency can be viewed.*

# 4 Discussion

The implementation and design process already happened planning to meet certain requirements. This chapter recalls some of these procedures for contrast ratio of texts and discusses their results and investigates further guidelines of the WCAG 2.1 for target area sizes. Additionally small navigation details that have been thought of are mentioned and brought into context.

## 4.1 Target area sizes

As a reference for screen size to evaluate the user interface, the NIDApad will be used (19). It is specifically developed for emergency situation by emergency responders. The screen supports a resolution of 1920x1080 and has a size of 11.6 inch resulting in a screen size rectangle of 25.68cm x 14.45cm.

In chapter 2.5.5 of the WCAG 2.1 it states the target size for pointer inputs for conformance level AAA to be at least 44x44 CSS pixels (20). It continues to specify exceptions, the two most important ones for this project being:

1. an **equivalent** satisfying the specified target size is present on the same page
2. the target is is **inlined** in a sentence or block of text

Since the screen size DPI can vary this does not represent on use-cases in the real world. The actual size of a target with a fixed size, like a button, on a 10 inch FullHD screen is smaller than on a 20 inch screen with the same resolution. Therefore I compare the actual button sizes in centimeter on the NIDApad to the mean of the thumb width and index width, being 1.6cm and 1.2cm respetively (21). Taking these average finger sizes into account, the following target size range can be computed as pixel per centimeter (ppc):

$$\begin{aligned} \mathrm{ppc}_x &= 1920\,\mathrm{px}/25.68\,\mathrm{cm} \approx 74.77 \\ \mathrm{ppc}_y &= 1080\,\mathrm{px}/14.45\,\mathrm{cm} \approx 74.74 \end{aligned}$$

Since for a 16:9 ratio the DPI is about the same we stick with $\mathrm{ppc} = \mathrm{ppc}_x := 74.77\frac{\mathrm{px}}{\mathrm{cm}}$ for further computations. The equivalent in pixels for the according finger sizes thus is:

| Thumb | Index finger |
|---|---|
| $1.6\mathrm{cm} \cdot \mathrm{ppc} = 119.632\mathrm{px} \approx 120\mathrm{px}$ | $1.2\mathrm{cm} \cdot \mathrm{ppc} \approx 89.724\mathrm{px} \approx 90\mathrm{px}$ |

The thumb pixel size will serve as a maximum and the index finger size as a minimum. These target sizes are almost three times as high as the highest conformance level of the WCAG 2.1. Instead of serving as a recommendation for a minimal square area it defines only a recommendation for one dimension, the width of the target area. This is because the width of our fingers aligns with the width of the screen rather than the height.

| Button | Area (px) | AAA | Finger width |
|---|---|---|---|
| Emergency selection on emergency select screen | 800(min) x 65 | yes | yes |
| Emergency selection on emergency view screen | 700(min) x 37.33 | partial | yes |
| Sidebar module button | 60 x 55 | yes | no |
| Sidebar module button (expanded) | 230 x 55 | yes | yes |
| Dashboard add module dropdown | 231 x 50 | yes | yes |
| Module selection in dropdown | 240(min) x 45 | yes | yes |
| Module tile close and maximize button | 40 x 40 | no | no |
| Module tile remove zone | 400 x 100 | yes | yes |
| Module tile remove confirm zone | 80 x 80 | yes | no |

*Table 3: This table lists all buttons and target areas with their fixed or minimal size in pixel and compares them to the discussed WCAG 2.1 (AAA) target area size from chapter 2.5.5 and to the computed finger width in pixels. For the WCAG comparison meeting the requirement partially means that only one dimension fulfills the minimum size. The finger width is only compared to the width, where partial means that it matches the minimum width of an index finger but not that of a thumb.*

Table 3 shows that most target areas fulfill both metrics. The selection of another emergency while managing the information of one, is deemed as a rare use-case, since usually the focus will be on the currently selected emergency. Furthermore the module tile close and maximum button have redundant target areas present. To maximize a module the according button on the sidebar can be used and to remove a tile from the dashboard it can be dragged onto a zone at the bottom that fulfills both metrics. The confirmation zone to remove a tile does not meet the requirement for the finger width since it is smaller, which is on purpose because removing a tile accidentally should be a hard to reach use-case. Additionally the user is dragging and thus gets continuous feedback whether the confirmation zone is currently hovered. At last the sidebar module button does only meet the width of the index finger or the thumb once the sidebar is expanded.

## 4.2 Text contrast

The color choices have already been shown in Section 2.2.2 and the contrast ratios between text colors overlapping has been computed (see Figure 4).

| Context | Contrast ratio | AAA |
|---|---|---|
| Standard text on standard background | 11.94 | yes |
| Module button text/icons on sidebar background | 10.31 | yes |
| Module button text/icons on hovered background | 8.91 | yes |
| Module button text/icons when active | 6.94 | no |
| Emergency selection | 7.81 | yes |
| Emergency selection when hovered | 4.55 | no |

WCAG 2.1 in chapter 1.4.6 requires a contrast ratio of at least 7:1 for text and images of text (22). Figure 4 shows low contrasts for the text color on the accentuated color (orange), this has been avoided because of this and the accentuated foreground color is used instead, which achieves the required contrast ratio. This is the case for the emergency selection button, where when hovered the background color transitions to a slightly darker orange, which does not reach the intended contrast ratio. Hovering is not a typical use-case on touchscreens and only an edge case, which is why this is regarded as not that important. Similar to the module button text and icon having slightly less of a contrast ratio once that module's page is currently visible. The information on that button is redundant information to the module's page displaying its module name as a big headline and the contrast ratio only misses the requirement by less than 0.1, which is why this also should not worsen the user experience.

## 4.3 General design choices

In general the application has been designed to make it easy for users to move between emergencies and other interface elements like different module pages. On the *emergency selection page*, users are forwarded directly if only one emergency is available. When multiple emergencies exist, each one is shown as a button containing the raw ISAN to not artificially hide any information. Another approach could be to extract that information and to put it into a more human-readable form.

In the *emergency view*, the weather is always visible in the top bar. This allows for quick checking of local conditions and has been added especially for weather warnings. Although no API has been implemented that delivers weather warnings, since those are mostly paid services, the layout is prepared for easy future extension; thanks to the component system only the according `Weather.vue` component needs to be changed for that. Clicking the current emergency in the top bar opens a dropdown menu showing raw ISAN as opposed to the reduced one that is shown as the current one. To not cause confusion the selected emergency's ISAN is highlighted with a blue color.

The sidebar uses only large icons at first to preserve as much screen space as possible for the main content. Especially for new users who may not yet know what module each icon represents, the sidebar can be extended to show text labels with the module name. This expanded view floats over the interface without resizing the main area. This prevents the user's dashboard layout from shifting or being compressed. Bot the top bar and the sidebar remain fixed in position so users always have a consistent reference point, regardless of which module they open.

On the *dashboard*, the dropzones are highlighted in green to clearly show where tiles can be placed. If a user drags a tile across a dropzone but does not want to attach it, pressing Escape cancels attaching entirely for the current drag action. This is not available on touch devices and might need further iteration in the future. The initial dropzones only allow for half-splits because adding options to place quads would double the amount of dropzones to eight and make interacting with them actually more difficult.

Designing the tile headers required special consideration. The headers show the module name and contain buttons for opening the according module page and removing the tile from the dashboard. Especially with tiles only using a quarter of the screen, the header occupies a lot of space that could be used for module information instead. The existence of the header is very important mostly because of the need to show what module that tile represents. Once the header is added to show the module name, the buttons themselves do not occupy more space from the module tile. Thus they have been kept for now with additionally adding a dropzone a tile can be dragged onto to remove it from the dashboard.

# 5 Conclusion

This thesis presented the design and implementation of a modular web-based curing system interface for the International Standard Accident Number (ISAN) system. The focus lay on improving information accessibility in emergency situations by integrating data from different systems into a single, coherent application allowing for easy extensibility in the future. The project followed a component-based architecture built with Vue and TypeScript on the front end and a Python/Flask back end, applying established software engineering principles. Central contributions of this work is the development of an asynchronous module loading mechanism and a freely configurable dashboard to show information from multiple modules on the same page in a layout the user desires.

The module loading mechanism loads modules dynamically, and their components are cached to reduce unnecessary workload and improve responsiveness. If configured to do so, modules can also retrieve data from the loader, loaded asynchronously and refreshed at specified intervals without the need to implement anything of that functionality. This data is cached across different views, ensuring that users can switch between the dashboard and full module pages without causing redundant requests. To simplify configuration, a YAML-based file was introduced, allowing module types and instances to be defined with minimal effort. This structure also provides a clear separation between module behavior and module implementation, suporting future extensibility.

The dashboard represents the core of the interface designed to allow flexible arrangement of module tiles. Users can position tiles freely or attach them to predefined dropzones, enabling them to create a layout that fits their preferences during an ongoing emergency. While the initial dropzones only allow half-screen splits, further splits into quads can be created by dropping tiles onto already attached ones. This approach avoids visual clutter and keeps the interaction predictable for the user. Module tiles can present reduced information, since they are viewed through smaller sizes, which encourages future module implementations to support adaptive visualization.

To evaluate accessibility and interaction quality, target area sizes and text contrast were examined. Most interface elements meet or exceed the WCAG 2.1 AAA recommendations or the calculated finger-width thresholds for touchscreen use. Where certain areas do not fully meet these metrics, like the tile header buttons, redundant interaction paths or deliberate design choices have been argued to justify these exceptions. Text contrast was also analyzed, showing that all essential text meets the required ratios, while minor deviations occur only in optional hover states.

The navigation and general layout were designed to remain stable regardless of the module or dashboard state. The topbar and sidebar stay fixed, providing consistent reference points during use. The sidebar can be collapsed to icons or expanded with labels, allowing users to choose between compactness and clarity. Although only basic weather data is currently included, the design anticipates future integration of warning systems with no structural changes needed.

Several areas for future work remain. Integrating a weather API that includes official warning data would make the topbar more informative. The emergency selection page could be improved by parsing and presenting ISANs in a more human-readable way without removing access to the raw identifier. On touch devices, adding a mode to drag tiles without triggering attachment would improve dashboard flexibility. Finally, future module developers can extend the system by providing additional compact visualizations tailored to the dashboard's reduced space.

In conclusion, the curing system interface of this thesis was developed with a modular UI design, combined with thoughtful concepts and adherence to accessibility principles, to support clearer, faster, and more reliable information access in emergency context. By providing a flexible architecture and a consistent visual framework, the system offers a foundation for further enhancements and integration into larger ISAN-based rescue chain workflows.

# Bibliography


1. Wilde ET. Do emergency medical system response times matter for health outcomes?. Health economics. 2013 July ;22(7):790–806.
2. Born C, Schwarz R, Böttcher T, Hein A, Krcmar H. The role of information systems in emergency department decision-making-a literature review. Journal of the American Medical Informatics Association : JAMIA. 2024 May ;31.
3. 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.
4. Gopal AD. How web-interface design impacts task performance, cognitive load, and user experience (UX): An exploration with a cohort of South African university students. The African Journal of Information and Communication (AJIC) [Internet]. 2025 Dec ;(36):1–22. Available from: https://ajic.wits.ac.za/article/view/23095
5. 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/
6. 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/
7. 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
8. Die neue BITV 2.0 [Internet]. Bundesfachstelle Barrierefreiheit; [cited 2026 Jan 14]. Available from: https://www.bundesfachstelle-barrierefreiheit.de/DE/Fachwissen/Informationstechnik/EU-Webseitenrichtlinie/BGG-und-BITV-2-0/Die-neue-BITV-2-0/die-neue-bitv-2-0_node
9. Harmonisierte Europäische Norm (EN) 301 549 [Internet]. Der Beauftragte der Bundesregierung für Informationstechnik; [cited 2026 Jan 14]. Available from: https://www.barrierefreiheit-dienstekonsolidierung.bund.de/Webs/PB/DE/gesetze-und-richtlinien/en301549/en301549-node.html
10. Diverse Abilities and Barriers [Internet]. Web Accesibility Initiative (WAI); 2024 [cited 2026 Jan 14]. Available from: https://www.w3.org/WAI/people-use-web/abilities-barriers/
11. Misra S, Roberts P, Rhodes M. Information overload, stress, and emergency managerial thinking. International Journal of Disaster Risk Reduction [Internet]. 2020 Dec ;51:101762. Available from: https://www.sciencedirect.com/science/article/pii/S2212420920312644
12. Uncomplicated Crisis & Emergency Management Software | D4H [Internet]. [cited 2026 Jan 13]. Available from: https://www.d4h.com/

13. 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.
14. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley; 1995.
15. Nielsen J. Usability Engineering. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1994.
16. 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.
17. Mohammed El Aouri. Duck Typing in JavaScript [Internet]. 2025 [cited 2026 Jan 12]. Available from: https://medium.com/@mohammedelaouri/duck-typing-in-javascript-3122786ddcf7
18. PointerEvent [Internet]. Mozilla; [cited 2025 Dec 6]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/PointerEvent
19. NIDApad [Internet]. medDV GmbH; [cited 2025 Dec 14]. Available from: https://www.meddv.de/nidapad/
20. Web Content Accessibility Guidelines (WCAG) 2.1 [Internet]. World Wide Web Consortium (W3C); 2025 [cited 2026 Jan 14]. Available from: https://www.w3.org/TR/WCAG21/
21. Boike S, Baker MJ, Ray L, Odenthal A, Bohn D, Van Heest A. Investigating Normative Measurements of the Thumb and Index Finger to Aid in Reconstructive Surgery. Journal of Hand Surgery Global Online. 2025 July 23;7(5):100792.
22. 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, 18.01.2026

---

(signature with first- and lastname)