Design and Implementation of a Dashboard for SilageControl

Lennart Rik Hemmerling

Bachelor's Thesis
September 29, 2022

Software Engineering Group
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Wilhelm Hasselbring
Malte Hansen, M.Sc.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

_____

Abstract

A dashboard can provide great opportunities for the visualization of data. With the help of widgets, it is possible to display a variety of information in a clear layout within one overview. Dashboards can be used to visualize various aspects of industrial production, but they can also be used to display aspects of learning platforms.

We created an application with the purpose of visualizing data regarding the production of silage. Therefore, we implemented a dashboard, which will help to depict data trends for optimizing the fermentation process, where plant material is compressed to prevent the formation of mold. It is possible to alter existing widgets as well as create new ones by writing configurations with *Typescript*. A configuration can be stored as a module, which allows a developer to use it in further applications as well. Within the setup of our project, we created tools to build new charts and modules. These can be implemented in projects which make use of *React*.

For validating our product, we initiated two evaluations, where the first evaluation focused on the usability of our application. Probands were asked to fulfill tasks with our dashboard. In the second evaluation, we tested our project environment, where probands developed new features for a widget. As a result, the evaluations show that the application, while suited for visualization, needs improvements in navigation. The project environment, on the other hand, is suitable for the development of new widgets and offers components that can be used for that purpose.

Contents

Chapter 1

Introduction

Creating expandable web projects is the key to modern applications. To reach this goal, one must decide on several technology alternatives. These technologies include frameworks, modules, languages, and runtime environments.

Part of this thesis is the development of a coding environment. The project can be found at *https://github.com/silolytics/platform-frontend*.

1.1 Motivation

The *NodeJs* runtime allows creating expandable web projects using a modular architecture. Any Node.js project contains modules, which are versioned and can be downloaded with a package manager. These modules can be used to add functionality to the product, while many of these modules can be swapped out at any time during development. The module *Lerna* can be used to organize several related NodeJs projects into one *GitHub* repository. Lerna is able to link local projects with each other. These are called *packages*. These packages can be used like Node.js modules, but during development, changes in code apply to all local instances.

To produce *silage*, plant material needs to be compressed. Therefore, a rolled tractor is used to drive over a silo. But due to the formation of mold, the production of silage needs to be observed. The *SilageControl* project[1] [2] aims to provide monitoring software for the production of silage. These processes require different user groups. Some workers may need to consult data while on a silo. The data which is monitored includes 3D capturing of the silo with the help of sensors which are installed on the rolled tractor. Additionally captured are weather data through online services, temperature data, and more.

Our goal is the creation of an expandable project environment with the intention to hold several similar web apps in the future. A package for reusable web components, including charts for data visualization, as well as a web application for a specific user group, is part of the implementation.

[1] https://www.silolytics.de/silage-control

[2] The project is supported (was supported) by funds of the Federal Ministry of Food and Agriculture (BMEL) based on a decision of the Parliament of the Federal Republic of Germany via the Federal Office for Agriculture and Food (BLE) under the innovation support programme.

1.2 Goals

In this paper we cover analysis of the maintainability and usability of a newly created web project environment for the SilageControl system.

1.2.1 G1 - Design of a Project Environment

Part of this work is to establish ways to automatically test modules and their components' functionality, as well as find technology that fits this requirement. The research focuses on usability studies, modules, testing, the used technologies, and the visualization of data to accomplish the goal of an expandable and maintainable project base.

1.2.2 G2 - Implementation

The designed structure is used to implement a website for a certain user group.

G2-1 - Design of an Expandable React Project Environment

A Lerna project featuring more than one Node.js package is our base. There are modules containing functional code and components, while other modules carry composed websites and other network projects and often refer to the first. The project makes use of *linting* and is set up for cooperative use.

G2-2 - Implementation of UI Components

A task is to create several UI components containing page layouts, buttons, and charts.

G2-3 - Data Visualization with Charts

The components feature several configurable charts. These can include bar charts, line charts, and pie charts. Data will be captured in real use cases, or it will be mocked for testing purposes.

G2-4 - Implementation of a Website using the Components

The components are used to compose a website for the user group *contractor*. It has a responsive design on several screen sizes, and it is possible to download the website as a *progressive web app*. The application features a dashboard, a detailed view for different dashboard widgets, and a list of silage heaps, which is responsive to screen size and input type. The dashboard should give an overview of the selected silage heap.

1.2.3 G3 - Evaluation

Furthermore, the implementation is tested for usability.

G3-1 - Use Cases

Probands are asked whether aspects of the application were easy to understand and whether the functionality could be used without much effort.

G3-2 - Development

The project is evaluated for its usability by developers. Therefore, probands are asked to implement a list of requested features. They then evaluate the process.

1.3 Document Structure

In Chapter 2 foundations and technologies for the following concept and implementation of the web application are presented. The project structure and work processes are described in Chapter 3.

Chapter 2

Foundations and Technologies

2.1 SilageControl

SilageControl is a project by the *Silolytics GmbH*. The Silolytics GmbH is a startup founded at Kiel University that aims to build monitoring software for the production of silage. This is accomplished by collecting and visualizing information about silos. With the help of sensors installed on rolled tractors, a 3D image is created to analyze the volume of silos. In the future, data about the compression of silos, the humidity of the silage, weather, and potentially more will be collected. Data trends will help to understand the processes regarding a silo as well as provide a basis for decision making.

2.2 Modules

A modular software architecture has advantages in re-usability and composition in comparison to a hierarchical approach. As pointed out by Chinthanet et al. [2021], a good module can be defined by the state of the documentation, the repository, the software, and the GitHub activity in the form of interaction with developers. By applying these requirements to the search for technology, one can find modules that are easy to install and understand, testable, and can easily be used in different projects. Moreover, an interest of this bachelor's thesis is to create such a module environment, aiming for re-usability for future development.

2.3 User experience design

User experience design (UXD) is mandatory for most web applications as it describes the usability someone has with the product. Therefore, it is needed to integrate this requirement into the development process, which can be challenging when working agile [Alhammad and Moreno 2022]. On top of that, *lean UX* describes the demand for established integration of UXD in agile iterations, because it can be time-consuming to reevaluate and recreate parts of the website structure.

2.4 Testing

An extensible application skeleton requires exhaustive test coverage. To accomplish that, a test environment is built of unit tests for functional code, as well as tests for UI components and their reactivity to changes in state. However, testing complex UI often includes the problem of *flaky tests* [Romano et al. 2021], which are non-deterministic because of, f. e., the order of execution (not resetting or setting up test states decently). As pointed out by Lam et al. [2019], the number of flaky tests in a project does not correlate to the number of failures in the production build, which results in the responsiveness of carefully setting up and maintaining a test environment, because flaky tests might not be detected easily. But in addition to that, an indication is that flaky tests tend to manifest in different runtime behavior for the allegedly same input state, so randomizing execution orders might give a hint on this behavior [Lam et al. 2019].

2.5 Charts and Data

Presenting data trends has become a major aspiration in web and mobile development [Brehmer et al. 2019]. Wu et al. [2020] shows that mobile applications and mobile formats of websites often take non-optimal approaches to displaying charts on smartphone screens. Important information gets eclipsed or is not shown at all, while the layout of a chart component does not change responsively with screen size. Brehmer [2019] gives examples of how charts can be implemented for several screen sizes.

2.6 Node.js

Node.js[1] is a *JavaScript runtime* that is used for creating network applications. Its modular architecture allows developers to quickly build JavaScript projects. Many web frameworks work with Node.js. For managing and downloading modules (packages), one can use *npm* (Node Package Manager) or *yarn*. Both tools rely on a file in the project root, the package.json. It contains information about installed modules and their versions, as well as additional information and settings regarding the project.

2.7 JSX

JSX is a syntax extension to JavaScript. It profits from direct manipulation by the JavaScript logic, which makes it a powerful UI language in comparison to *HTML* templates, for example. JSX looks like HTML, but it can be stored as variables and it features state-dependent code with the help of curly brackets (Listing 2.1).

[1] https://nodejs.org/en/

Listing 2.1. JSX in JavaScript code

```
1 const [variable, setVariable] = useState()
2 const component = <p>This text is {variable} and will rerender when changed.</p>
```

2.8 React

React[2] is an open source framework. It uses JSX to create web views, which allows high control over the UI state. By creating state-driven components, React can be used to compose complex but permanent websites.

2.9 Recharts

Recharts[3] is a Node.js module for React that aims to provide a variety of graph related components for use with a React implementation. Charts take a list of data points to display.

2.10 Typescript

Typescript[4] is a programming language that provides additional syntax to JavaScript. It allows the developer to define data structures in such a way that many errors that might occur in production will instead be thrown in development. By using structures like interfaces, it is possible to define precisely what a function is expecting as input, and the *Typescript compiler* (tsc) will print errors if the actual input does not match the specification. Typescript is interpreted as plain JavaScript code and uses JavaScript's primitives as well as objects, arrays, tuples, and custom types and interfaces, which can all be composed as alternatives to each other, giving a powerful tool to encourage ways to use the created code. As Bogner and Merkel [2022] mentioned, there is not much empirical data on the usability of programming languages. It does not seem that, in aspects like bug proneness and code length, statically typed languages have great advantages over dynamically typed languages, as shown with JavaScript and Typescript. Furthermore, while having empirical evidence that Typescript is less prone to code smells, Bogner and Merkel [2022] claim that Typescript applications frequently have a higher rate of committed bug fixes.

[2] https://reactjs.org/
[3] https://recharts.org/
[4] https://www.typescriptlang.org/

2.11 ESLint

Linting describes an automated process by which coding conventions can be realized by scanning project files and fixing lines of code to satisfy those. A Typescript version of *ESLint*[5] is used to automatically satisfy coding conventions. It should not be necessary to follow through with any conventions by hand, and the code should be checked when trying to push changes to the remote.

2.12 Jest

Jest[6] is a Node.js module for testing the UI/UX of a JavaScript-related project that includes a configuration for React with Typescript. It will check for a component's behavior regarding user input, state changes, and mocked data. Further on, we use Jest to test systems built for this project that are distributed over the applications and modules.

2.13 Lerna

Lerna[7] is an open source Node.js module. It is used to combine multiple Node.js projects into one GitHub repository. It can collect redundant modules of its sub-projects to avoid duplicates, which reduces the time to install dependencies while consuming less storage. On top of that, Lerna can link local projects as if they were officially distributed and installed modules, resulting in automated updates while in development. The *package.json* file at top level can be used to define scripts which handle multiple projects at once, for example, starting or testing a website with all its local dependencies.

2.14 Visual Studio Code

Visual Studio Code[8] is an *IDE* (integrated development environment) that can be used to develop Typescript projects. It analyses code and marks typing errors. It features several web project-related plugins. Visual Studio Code is used for the development process.

[5] https://eslint.org/
[6] https://jestjs.io/
[7] https://lerna.js.org/
[8] https://code.visualstudio.com/

Chapter 3

Envisioned Approach

3.1 Concept

The foundation of this project is an interconnected module environment. Therefore, we create reusable modules, which are maintainable and testable, and compare different approaches to find the ideal concept for the described implementation. A module dependency graph is presented in Figure 3.1.

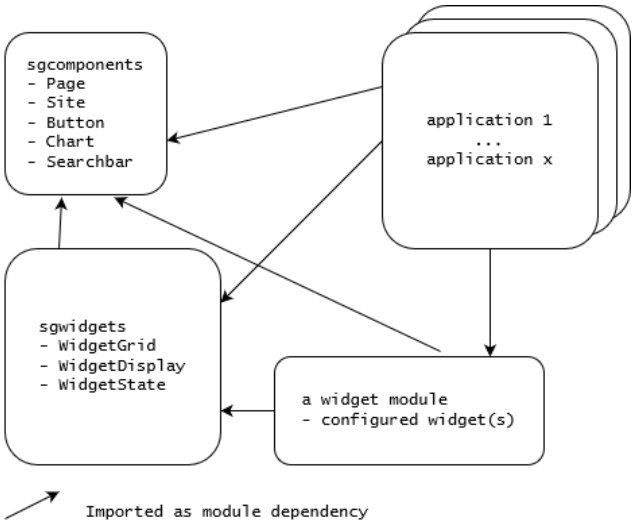

Figure 3.1. Visualization of module dependencies.

Keeping in mind that we are laying the groundwork for future projects, we write documentation. Further on, Lerna is used to keep modules organized. As described above, Lerna can be used to combine multiple Node.js projects into a single GitHub repository.

An exemplary concept of an application can be seen in Figure 3.2, Figure 3.3 and Figure 3.4. Contractors will use the application while at a silage heap to obtain information. This will happen on desktop computers as well as on tablets. Therefore, it has to be responsive across screen sizes and input types. Another use case is the comparison of silage

heaps, to receive information about improvements in the silage creation process. Therefore, there is a need to select silage heaps from a list for comparison (Figure 3.2), while each silage heap can be displayed as a dashboard (Figure 3.3) containing widgets. These widgets can be expanded in a detailed view (Figure 3.4).

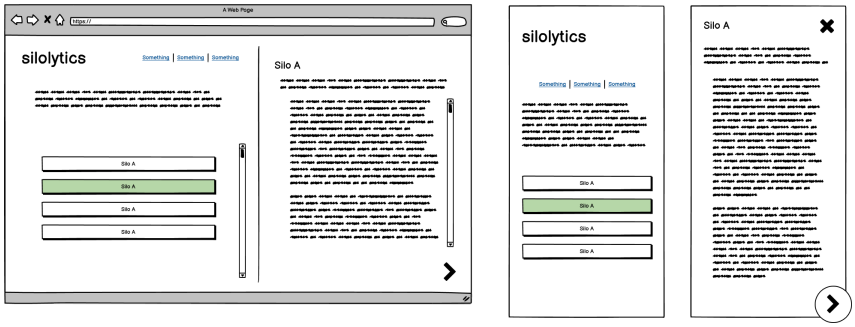

Figure 3.2. Conceptional visualization of a silo selection page, showing a desktop view on the left and a smartphone view on the right side.

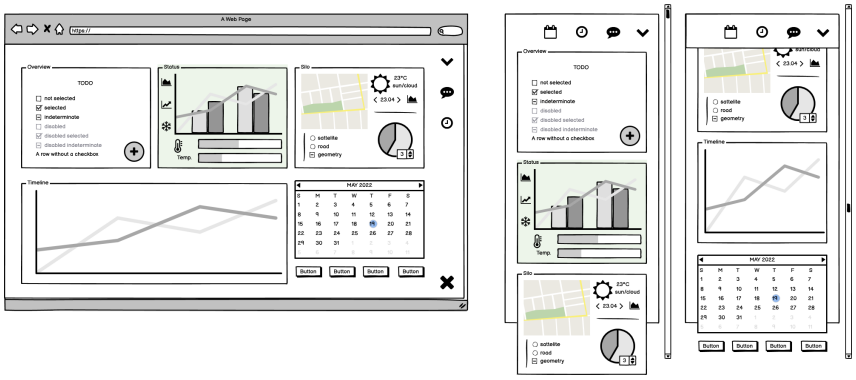

Figure 3.3. Conceptional visualization of a silo dashboard, showing a desktop view on the left and a smartphone view on the right side.

The purpose of the created modules is to ease the development of applications within the project. Therefore, they feature different UI components and configurations from which an application can be composed.

The application is evaluated for usability and its development environment.

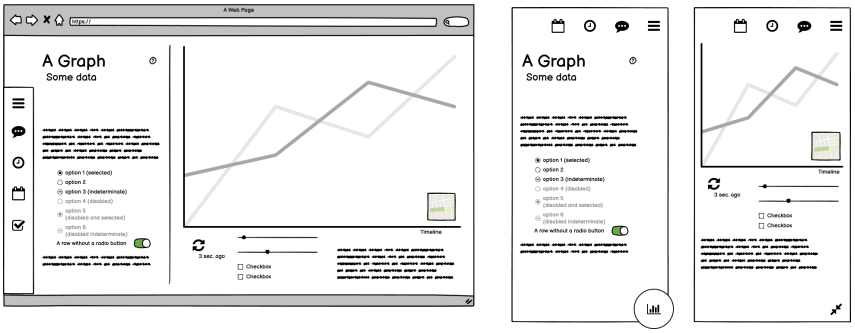

Figure 3.4. Conceptional visualization of a detailed view of a dashboard widget, showing a desktop view on the left and a smartphone view on the right side.

3.2 Implementation

At the top level, the project contains a Lerna setup, a package.json, and a file for Lerna specific settings (Figure 3.5). A folder contains any subordinated Node.js projects, which are managed by Lerna. When installing remote packages, a Lerna script runs to minimize storage by gathering redundant modules and distributing them to lower level projects. On top of that, Lerna makes it possible to import these projects from one another as if they were published modules, without the need to manually update changes.

Git[1] is used to version the project environment. We create new features on separate branches, and pipelines ensure that the code is tested and cleaned before being pushed.

Part of our work is a module for reusable React components. It features a file as an entry point, which makes use of JavaScript's named exports (Listing 3.1). This way, import statements of the module are a subset of all created components, allowing code that is easy to read and maintain.

Listing 3.1. Importing components in an application

```
3  import {
4     MyPageLayout,
5     MyButton,
6     MyGraphs
7  } from 'silolytics-components'
8
9  const component = <MyPageLayout> <MyButton /> <MyGraphs.Bar /> </MyPageLayout>
```

The component module contains layouts to allow the composition of applications with fewer lines of code. The layouts handle screen sizes and ensure that the content is well organized and visible. Therefore, we use Typescript's typed objects as component properties,

[1] https://git-scm.com/

which help to implement the created components during development. Typescript features interfaces to accomplish that, which are stored in a file accessible to the components and websites using the component module. The types are imported along with the components. Visual Studio Code will automatically generate suggestions based on the corresponding types and highlight errors in code.

The website imports and configures components. When refactoring components, they are updated around the projects while the component interfaces ensure that properties stay consistent. For data visualization, the website uses components which are powered by the Recharts module, by giving data from a backend on to the charts. The component module features several charts, to cover the Recharts module and at the same time reduce the amount of configuration needed on the websites.

It is possible to import *CSS* files into the websites, which contain rules to change a component's appearance by class name. We document these classes to encourage reuse and to ease the styling of an application. Further on, class names can be overwritten by other modules. This can help to implement changes in style at several points in an application without altering the imported module itself.

3.3 Evaluation

The general UX design has to be evaluated in user scenarios. For that purpose, we ask students and employees of Kiel University as well as practitioners for testing. Reported problems with the UX design and change requests are analyzed accordingly. Furthermore, we evaluate the project environment. Any feedback gathered during the evaluation is documented and analyzed.

3.3.1 Usability

Questionnaires are created to collect data from test scenarios where certain functionalities are fulfilled by the testers. They rate the usability of these functionalities on a scale.

3.3.2 Requirements

The functionalities provided by the web application as well as the project environment are evaluated by employees of Silolytics GmbH, by students and employees of Kiel University, and by practitioners in the field of software engineering, to check whether the requirements of an expandable coding base are met.

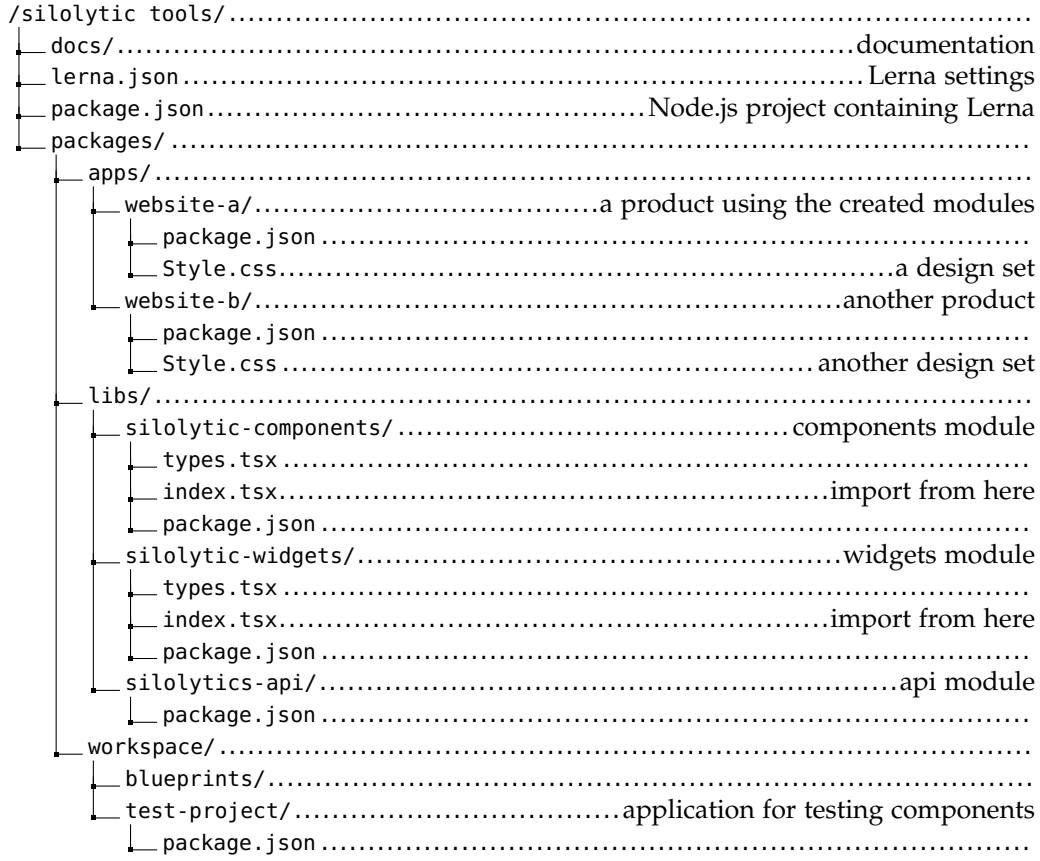

Figure 3.5. Shown above is the file hierarchy of the repository of our Lerna project. Folders are marked with a slash. The files ending with "json" are read as JavaScript Object Notation and are often used to store settings. Files ending with "tsx" are Typescript files, which, among other things, are used to describe React components. The "css" ending marks files that are used for styling. The folder used for documentation mostly contains Markdown files, which are supported by GitHub.

Chapter 4

Implementation

The structure of this project is inspired by the modular design of React and React-related packages. Therefore, multiple web applications can use the same component modules.

This is accomplished by using the node module Lerna, which handles multiple node projects within one git repository.

4.1 Lerna

With the Lerna setup, it is possible to install Node.js modules, which are used in more than one nested project, on the top level automatically by specifying the *hoist* option for the Lerna bootstrap command. The Node.js modules are then referenced in the sub-projects. However, it is necessary to be clear about versions. For example, the module ESLint demands to have the same version over the complete project, while it is possible to specify that it installs minor version updates automatically. If a developer is running into such problems, the resolution often is to delete all Node.js modules and reinstall them using Lerna. Therefore, we created a npm script (Listing 4.1) which deletes all nested, as well as the top level *node_modules* folders, using a combination of the lerna clean script and a remove command, while there is a post-install hook which runs the Lerna setup on installation.

Listing 4.1. Cleaning the repository from all node_modules folders.

```
10  // package.json on top level
11  ...
12     "clean": "lerna clean --yes && rm -r ./node_modules",
13  ...
```

After running the npm install command while using automatic minor versions, it might happen that a newer version of a dependency is installed without a note, which can result in unexpected errors, as mentioned above, because in reality, not every time an update is released it is versioned correctly. In the case of Recharts, we ran into a problem, which caused jest tests to fail, because a new minor version of Recharts was not compatible with our jest configuration anymore. To overcome this, one can remove the circumflex prefix of versions in the package.json file to disable the automatic installation of newer versions of modules, which are marked as stable remotely. The circumflex is part of the semantic

versioning and marks a module dependency to be updated to a newer minor version if there is a newer and stable version online.

By setting the Lerna project to use independent versions, versioning of own modules can be ignored, which is handy when working with a small team and setting up the project. Nested projects are organized in the packages folder by default. This can be changed by specifying the packages property of the Lerna configuration file (Listing 4.2). In our case we choose to create three sub-directories within the packages folder: *packages/libs/*, *packages/apps/*, and *packages/workspace/*. *packages/libs/* contains libraries and component modules, *packages/apps/* contains web applications and potentially other products that use our libraries, and *packages/workspace/* contains blueprints as well as projects for testing modules.

Listing 4.2. Lerna settings

```
14  {
15     "version": "independent",
16     "npmClient": "npm",
17     ...
18     "packages": ["packages/libs/*", "packages/workspace/*", "packages/apps/*"]
19  }
```

4.2 React

React features reactive UI elements that are written in JSX. With this, it is possible to change the UI by using e.g. React hooks, which store data and, when changed, will trigger an update of the components. A React component can be a class or function, where React class components can implement functions, which are called on specific life cycle events, while React function components use hooks to accomplish that, as can be seen in Listing 4.3.

Listing 4.3. Example of a React component using a state.

```
21  const ButtonExample: React.FC = () => {
22     const [clicked, setClicked] = useState(false)
23
24     return (
25        <input
26           className={clicked ? "exampleButtonClicked" : "exampleButton"}
27           type="button"
28           onClick={() => setClicked(true)}
29        ></input>
30     )
31  }
```

```
32
33
34    export ButtonExample
```

These React components can then be imported and are handled like other JSX. In combination with Typescript, React offers types for functions and classes, as well as types that can be used to allow certain primitives to be used side-by-side with React components. A property for a text component could either be a React component or a string and, for example, function callbacks can be defined in a specific way (Listing 4.4).

Listing 4.4. Example of a React component

```
35 ...
36 import { StyleProps } from "../types"
37
38 interface SearchbarProps extends StyleProps {
39    items: Array<any>
40    callback: (string) => boolean
41    component: React.ReactNode
42    ...
43 }
44
45 const Searchbar: React.FC<SearchbarProps> = ({style, className, items, ...}) => {
46    return (
47       <div>
48          ...
49       </div>
50    )
51 }
52
53
54 export default Searchbar
```

4.3 React as Peer Dependency

A *peer dependency* is a kind of module dependency that allows the developer to mark a module to be dependent on a specific version of another module. In opposite to a default dependency, such modules do not install when used as a plugin for another module, which uses a different version. Instead, the package manager, in our case, npm, throws an error. It is necessary to install React as a peer dependency in projects which will be used as modules for our applications, so that only one version of React is installed in an application. When using independent React modules, there will be problems with React hooks because

they are managed by React internally and interfere when React is used as a dependency in the imported module.

4.4 Components

The first module to implement is a React component library. It features several UI elements that can be used to build an application. These components include a web page wrapper as well as layout components and input elements such as buttons and search fields. The page components can be used to implement responsive sites without having to write style definitions with CSS. But if needed, styles can be changed by overriding classes, which are documented along with the module. In a React application, the developer needs to create a CSS file, define the class which shall be overridden, and import it into the new site (Listing 4.5).

Listing 4.5. Overriding styles within an application.

```
// Style.css
.Page {
   background-color: blue;
}

...

// CustomPage.tsx
import 'Style.css'
...
   return <Page></Page>

...
```

4.4.1 Components of the Component Module

In the following, we present some of the relevant components, which we create within the components module.

ApplicationWrapper

The *ApplicationWrapper* is used as a parent component for an application. It is styled in a way that child components are centered on the screen.

Page

The *Page* component stretches out over the screen and is handy for defining the maximum width a website should have. It also features a strategy to handle content for broad displays as well as smartphone displays. It should be wrapped in an ApplicationWrapper, which centers the Page component.

Site

The *Site* component is handy for wrapping multiple UI elements. It is used as a child of the Page component and features properties which help sort elements in a responsive way. For example, the align property decides whether UI elements are centered or left-aligned. In combination with a Page component as parent, one or two Site components can be displayed responsively, while on small displays the second Site component is displayed as an overlay to shorten the length of web pages for smartphone users, which results in less scrolling, especially if combined with charts and information in the form of text, where a user might want to jump from one to the other. An implementation starts with an import statement (Listing 4.6-l.:68) of the needed components. Then multiple JSX components can be defined with Pages wrapping one (Listing 4.6-l.:74) or two (Listing 4.6-l.:83) Site components.

Listing 4.6. Using pages and sites.

```
68  import {
69     Page,
70     Site, SiteAlign,
71     Button
72  } from 'sgcomponents'
73
74  const SinglePage = (
75     <Page>
76        <Site align={SiteAlign.LEFT}>
77           <p> Some Text </p>
78           <Button />
79        </Site>
80     </Page>
81  )
82
83  const DoublePage = (
84     <Page>
85        <Site align={SiteAlign.LEFT}>
86           ...
87        </Site>
```

```
88
89          <Site align={SiteAlign.LEFT}>
90             ...
91          </Site>
92       </Page>
93    )
```

Charts

Furthermore, different chart types are implemented. With the help of Typescript, several chart-related data structures are defined. These data structures help create complex applications, especially when using an IDE like Visual Studio Code, which marks possible solutions and highlights type errors during development. A *Chart* component takes one or more typed objects as configuration, which can be seen in Listing 4.7, to provide an implementation for composed charts that are displayed on top of each other (Figure 4.1). Chart configurations can be stored in variables (Listing 4.7-l.:97) and are given to a chart component as a property (Listing 4.7-l.:114). When using Typescript, it is often necessary to install additional type packages because many Node.js modules do not use Typescript and therefore are not typed. However, the Typescript community releases these packages for many modules as a workaround, but not every package is flawless. In the case of Recharts, there were several inconsistencies.

Another problem with the Recharts module was that composed charts can only be rendered correctly when every point on the x axis is shared among different curves. This can be fixed by implementing the data points as a property of child components directly instead of combining them into one set on the parent level. Bar charts, on the other hand, feature this property as a type, but are not using it. This results in bar charts having to rely on the parent property. We created a workaround, where a list of chart configurations is split into bar chart configurations and other chart configurations. The bar charts are inserted into the parent component, while other charts are given to the child components. Pie charts are handled separately, but use the same configuration options.

Listing 4.7. Using chart configurations with multiple data sets.

```
94 ...
95
96 const component = () => {
97    const dataSet1: ChartData = {
98       type: ChartDataType.AREA,
99       style: {
100         ...
101      },
102      points: [ [1, 1], [2, 2], [3, 4] ],
```

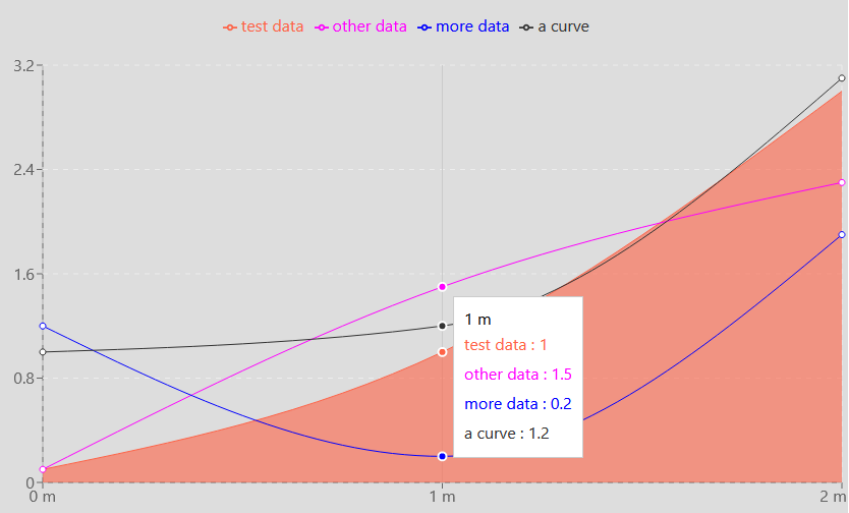

Figure 4.1. A chart containing multiple data sets

```
103       ...
104    }
105
106    const dataSet2: ChartData = {
107       ...
108    }
109
110    return (
111       ...
112
113       <Chart
114          data={[
115             dataSet1, dataSet2
116          ]}
117       />
118
119       ...
120    )}
```

Searchbar

A *Searchbar* component is featured by the component module, to be able to quickly implement a large list of items, which can be filtered by custom aspects. Therefore, the

Searchbar component has a property called *find*, which is a field for specifying a callback function, which is called every time an item in the list is looked at (Listing 4.8-l.:125). In the function head, an item, as well as the current search text, is given, while a boolean is returned. The boolean indicates whether the item will still be shown, depending on the current search text. The items (Listing 4.8-l.:124) are of the *any* type, which allows us to insert any kind of structure or variable, to create flexibility for the implementation of the component. A label (Listing 4.8-l.:123) is necessary for accessibility, which is a string describing the use of the Searchbar.

Listing 4.8. An example of a searchbar implementation. "items" is the list of items to filter. "setSubSet" is used to create a list of items which were not filtered out by the find function.

```
121   ...
122   <Searchbar
123      label="Silo-search"
124      items={items}
125      find={(item, val) =>
126         val === ""
127         || ((item.name) as string).toLowerCase().includes(val.toLowerCase())}
128      onChange={setSubSet}
129   />
130   ...
```

Button

We implement the *Button* component, which is a styled version of the React button component and can be used in a similar way. It also features CSS classes, which can be overwritten in an application.

4.5 Widgets

The widgets module is used to implement a widget system in different applications. It features a grid layout component, which originates in the components module but is wrapped and named *WidgetGrid*. This component can display different widgets by specifying a list of widget configurations, which again are typed using Typescript. This configuration is of type *WidgetState*. There are different sorts of widgets that display text and data with the help of the chart component of the components module, but a developer can implement custom widgets by using the appropriate type and configuration fields.

4.5.1 Implementing the Widget System within an Application

Most widgets can be displayed in a more detailed view, which is implemented with a component of the widget module, the *WidgetDisplay*, and uses the Page and Site components as well as other UI elements from the component module. The WidgetGrid implements a function, which is used as a callback and is triggered when a user clicks a button to show the widget in detail. This can be seen in Figure 4.2. Furthermore, the WidgetDisplay implements a property for the *WidgetStateWrapper*, as well as a function to update the wrapper. When implementing both components in an application, the developer can connect them within a wrapper component in a reactive way by using the React state or store to synchronize both component states. This results in updates being propagated through all components, where everything that is dependent on the changes gets re-rendered.

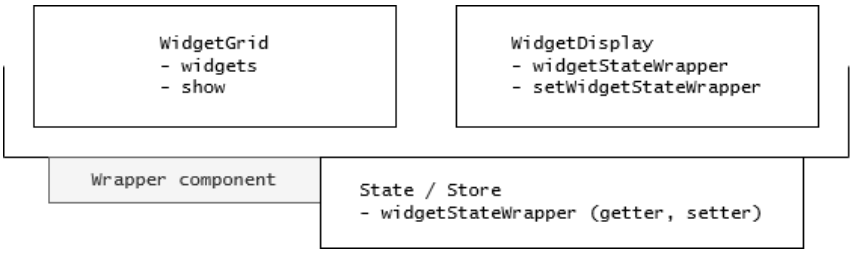

Figure 4.2. Visualization of an implementation of the WidgetGrid with the WidgetDisplay.

4.5.2 Life Cycle of Widgets

Acquiring data is managed by the widget system internally. The developer can specify functions that are used to fetch data asynchronously and are called during the life cycle of widget components to optimize network usage. These functions are called when the widget loads and when the user clicks a refresh button. The fetched data is saved in a wrapper object, which also holds the widget configuration. This is presented in Figure 4.3. When implementing custom components for widgets, the wrapper and a refresh function are given in the function head. Calling the refresh function will update the data according to the specified method, which will trigger a re-render event on custom components as well as the widget and the display component (Listing 4.9, Listing 4.10). One can, for example, implement the refresh function as an event on a button component.

4.5.3 Widget Configuration

This section gives a brief overview of the possibilities for creating new widgets.

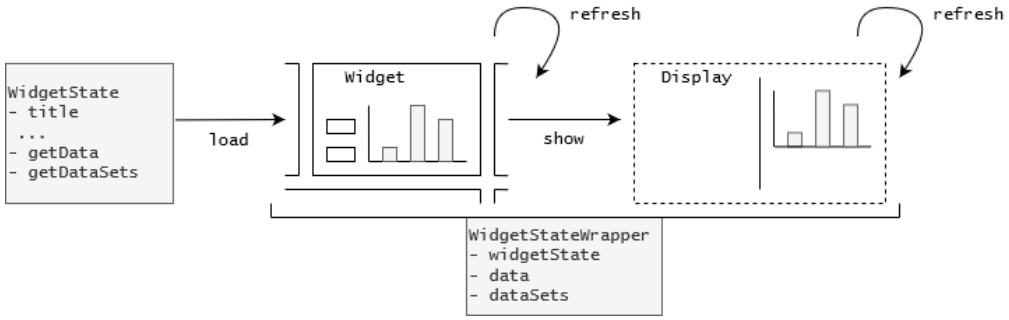

Figure 4.3. Visualization of the widget life cycle.

WidgetDisplayType

The *WidgetDisplayType* decides what functions a widget should fulfill. At this state of our work, a widget can implement three different types. The first is the *simple* type, which is a minimal widget with text components. The second type is the *data* type, which displays a chart component along with the text. The third type is the *custom* type. With a custom widget, a developer can implement a new component by using JSX within the corresponding fields of a WidgetState. On top of that, the type which is used by the WidgetDisplay can differ from the type a widget uses within the WidgetGrid. A developer could, for example, implement a custom display component but keep the data type of the widget component because it fits the requirements of the new widget configuration. That would result in a data widget that uses a newly created custom display component.

Other Properties and Custom Components

By implementing fields such as title, subtitle, and text, a developer can specify the information that is carried by simple or data widgets and displays. However, even a custom component can implement such fields. When a custom component (Listing 4.10-l.:164) is rendered, the WidgetStateWrapper (Listing 4.9-l.:151) of the current widget configuration is passed in the function head. This way, the component can implement the fields that were specified within the WidgetStateWrappers property *widgetState*, which is the widget configuration. The fields that are used by the simple and data widgets (Listing 4.9-l.:140) can be composed in the custom component (Listing 4.10-l.:167). An exemplary configuration of a WidgetState can be found in Listing 4.10, where the widget type is defined in Listing 4.10-l.:157 and the type of the WidgetDisplay is defined in Listing 4.10-l.:158. They share the same options because they are directly dependent on each other through the widget configuration with fields like title (Listing 4.9-l.:140) and others.

Listing 4.9. The widget configuration type along with its wrapper.

```
export enum WidgetDisplayType {
   CUSTOM="custom",
   SIMPLE="simple",
   DATA="data"
}

export interface WidgetState {
   widgetType?: WidgetDisplayType
   displayType: WidgetDisplayType
   title?: React.ReactNode
   text?: React.ReactNode
   ...
   getData?: (options?: any) => Promise<any | undefined>
   getDataSets?: (options?: any) => Promise<Array<ChartData>>
   ...
   widgetComponent?: WidgetStateComponent
   displayComponent?: WidgetDisplayStateComponent
   ...
}

export interface WidgetStateWrapper {
   widgetState: WidgetState
   data?: any
   dataSets: Array<ChartData>
}
```

Listing 4.10. Using widget configurations.

```
156   const widgetConfiguration: WidgetState = {
157      widgetType: WidgetDisplayType.DATA,
158      displayType: WidgetDisplayType.CUSTOM,
159      title: 'Example Widget (This is text or component)',
160      getDataSets: async(options: any) => {
161         // await fetching data from backend
162         // return data as ChartData
163      },
164      displayComponent: ({widgetState, dataSets}, refresh) => {
165         return (
166            <Site>
167               {widgetState.title}
168            </Site>
169         )
170      }
171   }
172
173   export widgetConfiguration
```

Localization

Localization of widgets can be accomplished by using the Node.js module *react-i18next*, a module for implementing more than one translation within the application. It features, among other concepts, React hooks and components, which re-render when the language is changed. One way to accomplish that is to use a react-i18next component instead of text for fields like *title* or *text*, as shown in Listing 4.11-l.:180. The translations are stored as objects in JSON-files and are accessed by using their path as an argument in a translation function (Listing 4.11-l.:181).

Listing 4.11. Using widget configurations and localization

```
174   import { Translation } from 'react-i18next'
175   ...
176
177   const widgetConfiguration: WidgetState = {
178      ...
179      title: (
180         <Translation>
181            { (t) => <h1> { t('path.to.translation') } </h1> }
182         </Translation>
183      ),
```

```
184       ...
185    }
```

4.6 Applications

In the following, we present an approach to creating an application.

4.6.1 Routing

Routing of web applications is implemented with the help of react-router. It is a module for React, where sites are given as properties to a routing component. The top-level component then renders a site depending on the address. An implementation of a routing system is shown in Listing 4.12.

Listing 4.12. Using react-router as routing system

```
186 import { BrowserRouter, Routes, Route } from "react-router-dom"
187
188 import { ErrorPage } from "sgcomponents"
189 import Login from "./pages/Login"
190 import Silos from "./pages/Silos"
191 import Data from "./pages/Silo"
192
193 export default function Router() {
194    return (
195       <BrowserRouter>
196          <Routes>
197             <Route path="/">
198                <Route index element={<Login />} />
199                <Route path="silos" element={<Silos />} />
200                <Route path="silo/:id" element={<Silo />} />
201                <Route path="*" element={<ErrorPage />} />
202             </Route>
203          </Routes>
204       </BrowserRouter>
205    )
206 }
```

The index property (Listing 4.12-l.:198) marks a route to be the landing page. The path property (Listing 4.12-l.:199) specifies an address ending with which the route will be accessible, while pages that are accessible through any address ending are marked with a star (Listing 4.12-l.:201). The address resolution is fulfilled top-down, which allows catching

bad requests. A component given to a route as a property can be a Page component, therefore the router is nested in the ApplicationWrapper.

4.6.2 Implementation of the WidgetGrid and WidgetDisplay

We implemented the widget system by adding a drawer to the application's detailed view. When a widget is selected, the drawer minimizes and an underlying WidgetDisplay is set to display the selected WidgetStateWrapper. A button in the drawer navigation bar indicates whether the drawer can be maximized again (Figure 4.6). As long as a widget has been selected (Figure 4.5), the drawer can be minimized to show the widget display. The ability of the widgets to minimize the drawer is given by the implementation of the show-property of the WidgetGrid. The specified function not only sets the current WidgetStateWrapper but also sets the drawer state to be retracted.

4.6.3 Comparing Silage Heaps

A feature of the created application is the possibility to compare silage heaps. Therefore, we implemented a system where multiple silage heap ids are propagated to the detailed view of our application. These are shown as a list at the top of the WidgetGrid (Figure 4.6). A user can switch between the selected ids by clicking the corresponding button in the list. The list of ids is given to the detailed view by setting a GET appendix in the address. This is marked in the path of the router component carrying the Silo page (Listing 4.12-l.:200). The appendix is built from one or more ids, separated with a plus sign. The Silo page dissolves the appendix by recreating a list of ids. It is possible to save a certain list of silage heaps as a link by copying the address with its appendix.

4.6.4 Localization

Different languages are handled by implementing the module react-i18next.

4.6.5 Realized Sites and Widgets

The created application is a basis for future improvement. It is made of three routes, while several aspects and functions are mocked. The first route is a login, featuring a button to produce an address-related error, as a proof of concept, as well as a button to switch the language. This can be seen in Figure 4.4.

A second route is the silage heap selection page. It features a list of silage heaps, from which multiple heaps can be selected (Figure 4.5) with the help of checkboxes.

The third route is the dashboard, including several widgets and their detailed views.

We implemented three widgets using test data. Figure 4.6 shows five widgets, while the lower two do not use test data. The first widget shows general information about the current silage heap, including meta data like the name of the heap, an id, and the date on

Figure 4.4. The login page.

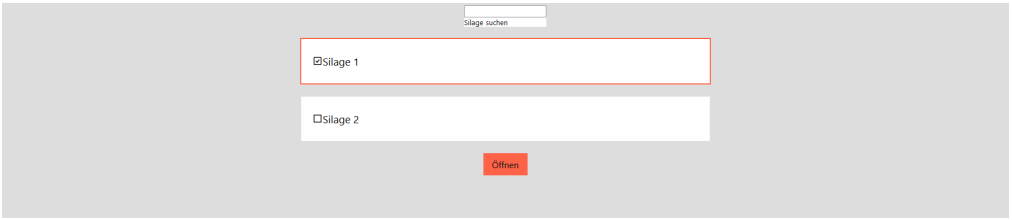

Figure 4.5. The silage heap selection.

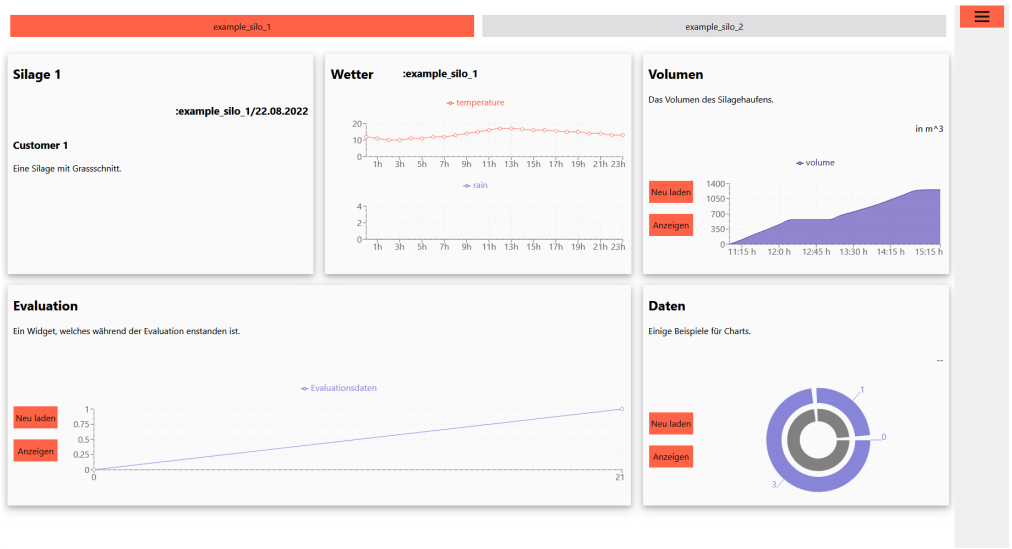

Figure 4.6. The dashboard contains five widgets. The first widget is a meta widget, which contains data about the silage heap. The second is a weather data widget. The third widget is used to display data about the volume of a silage heap. The last two widgets are not part of the requirements, while the latter shows possibilities to implement charts and the other was created during the evaluation.

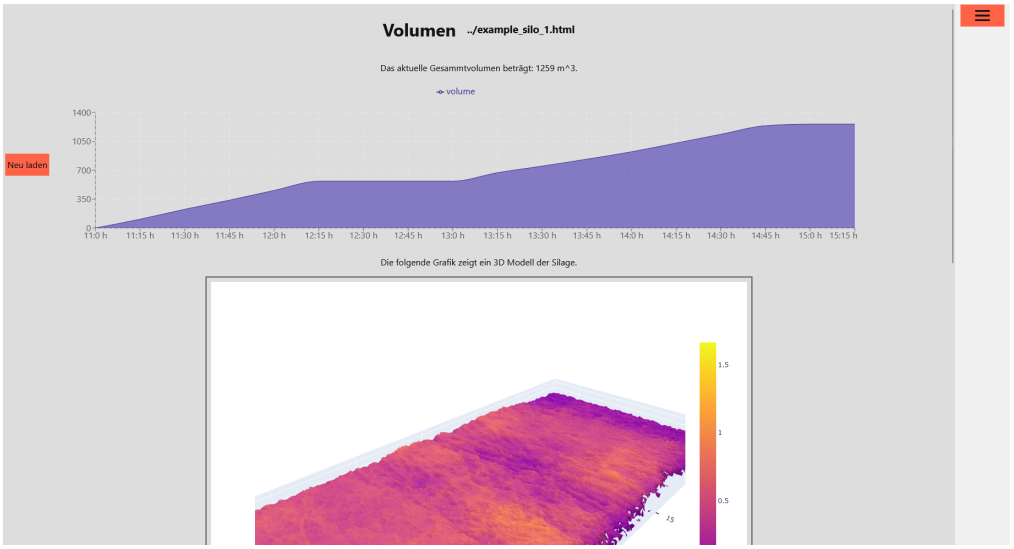

Figure 4.7. The detailed view of the volume widget.

which the heap was created, as well as the owner of the silage heap. The second widget shows data about current weather conditions, including temperature and precipitation. The third widget displays data about the volume of the silage heap. When opening the volume widget, a detailed view is shown, which can be seen in Figure 4.7, where the user can see a 3D model, as well as a picture of the manual measurement, which is done with the help of GPS. The fourth widget was created for the purpose of the development evaluation, while the fifth implements different charts, which were not covered by test data.

The 3D model is realized with the help of an *iframe*, which is a JSX tag to embed another website within an application. Due to the 3D model being a python plot in the form of an HTML site, one can render the model with the help of an iframe, as can be seen in Listing 4.13. Therefore, the address under which the 3D model can be found is set as the *src* property (Listing 4.13-l.:215).

Listing 4.13. Using an iframe to display another HTML site.

```
207    ...
208
209    const Plot3DModel: React.FC<Plot3DModelProps> = ({src, width, height, title,
           debounceMs}) => {
210       const [startLoading, setLoading] = useState(false)
211       if(!startLoading) setTimeout(() => setLoading(true), debounceMs)
212
213       return ({
214          startLoading ?
215          <iframe style={{border: 'solid 2px #555', padding: 0, margin: 0, maxWidth
                : '98vw'}} title={title} src={src} width={width} height={height} />
216          : "Loading..."
217       })
218    }
```

It is possible to download the application as *PWA* (progressive web application) by using the corresponding function of the browser. This is accomplished by a *service worker,* which cashes the application so a user can save the last downloaded state of our application locally.

Chapter 5

Evaluation

In this chapter, we describe the realization of our evaluations. The evaluation is split into two parts. The first part is a usability study of the created application. The second evaluation focuses on a development task, where probands were asked to implement certain feature requests into a module, which we created for the evaluation purpose. In this chapter, we explain the setup of the evaluation as well as present our results. Furthermore, we discuss the outcome.

For test purposes, we mocked the data which is shown in the application.

5.1 Use Case Evaluation

In this evaluation, the probands were asked to fulfill certain tasks with the application. We created a questionnaire covering an introduction to the silage process, fields for personal data, and some exercises which the probands were asked to solve. Afterwards, the probands were asked to fill out an evaluation form to give an estimate of how intuitive the application already is.

5.1.1 Goals

With the help of our questionnaire we want to evaluate which features are intuitive already and what can be improved in the future. Therefore, the probands were not introduced to the application at all.

5.1.2 Questionnaire

To get feedback from our probands we created a questionnaire. The questionnaire can be found in Appendix A. This form contains an introduction, fields to fill in personal data, as well as tasks to fulfill during the evaluation and questions to fill out after the evaluation.

Rating

After completing the exercises, the probands were asked to evaluate the application. Therefore, the questionnaire contains several questions with a numerical grade, as well as

questions for text-based feedback. The grades are set to be between one and five, where one means that there is a strong disagreement and five means that the proband fully agrees.

Personal Information

The questionnaire contains a section requesting personal data from our probands. This data is used to ascertain the quality of our application based on the probands' previous experience and to be able to point out deficits in how intuitive our design implementation is. The information collected in this survey can be seen in Table 5.1.

Table 5.1. Personal data received by the the probands.

ID	Profession
UA1	Student
UA2	Researcher
UA3	Practitioner
UA4	Other

ID	Prerequisites
UA5	Agriculture
UA6	Programming
UA7	UI/UX

Statements

The statements focus on the usability of features of the application. The following table lists statements that were answered by the probands (Table 5.2).

Table 5.2. Statements regarding the use case evaluation.

ID	Statement	Grade/Text
UB1-1	Is it intuitive to navigate through silage heaps?	Grade
UB1-2	What were the problems?	Text
UB2	Is it intuitive to find data for specific timestamps?	Grade
UB3	How intuitive was comparing different silage heaps?	Grade
UB4-1	If you had problems with navigating, did they occur more than once?	Text
UB4-2	What were they?	Text
UB5-1	Was it possible to use the application on the given screen?	Grade
UB5-2	What went wrong?	Text
UB6	Additional remarks.	Text

Experimental Setup

The application is started in a browser (Edge / Firefox). After reading the questionnaire, the probands start with the given tasks. They are asked to find multiple values for the given

silage heaps without any prior knowledge of where to find them. These values include the temperature and volume at a given time, the maximum height of the silage heap, the name of the customer, and the difference between the volume measured by sensors and the volume measured by GPS. Then they evaluate the application.

5.1.3 Results

The results of the use case evaluation are presented here. The statements each proband has given can be found in Appendix A.

Personal Information

In this evaluation, 13 people participated. Seven of the probands stated that they were students, six stated that they were researchers, while two of the probands stated they were practitioners. The data regarding the prerequisites is listed below in Table 5.3.

Table 5.3. Statements regarding prerequisites.

ID	Mean
UA5	2.0
UA6	3.7692
UA7	2.9231

Evaluation

The following shows some of the data that we gathered through the evaluation (Table 5.4).

Table 5.4. Statements regarding usability.

ID	Mean
UB1-1	3.7692
UB2	4.5
UB3	3.5385
UB5-1	4.7692

Only one proband gave no negative feedback on the applications' navigation. All other probands had problems with finding back buttons and with the way these buttons behaved. Most probands did not use the show-button of widgets at first; they tried to open a widget by clicking on empty space within it. Many probands tried to open a drawer with the drawer-button while the dashboard was already extracted.

5.1.4 Discussion

In the following section, we will discuss the results. Therefore, we use the presented results (Table 5.3, Table 5.4), as well as the statements the probands have given and our own observations.

Personal Information

Most participants did not have any prior knowledge regarding agriculture. However, many of them stated that they have programming skills, and some stated that they have experience with UI/UX.

Evaluation

Most probands had problems with navigating through the application. In particular, switching from one silage heap to the other seemed to not be intuitive enough. Many probands stated that they were not finding a back button and that the back button of the browser did not work as expected. That has to do with the drawer navigation, where the dashboard and detailed views are wrapped under a single address, which results in the back button navigating to the list of silage heaps in spite of the drawer being collapsed. A more intuitive design of the browser's back button would result in the drawer being opened if the button is pressed in the detailed view. Another improvement is to implement a back button within the drawer navigation, as some probands state. Furthermore, the drawer navigation should be placed on the left side or on top instead of the right, because the buttons would be located under the navigation of the browser, which would make them easier to notice, as one participant suggested.

While most probands did not have problems with finding data for specific timestamps, it was difficult for many probands to compare silage heaps with the application. This manifested in many probands not noticing that the list of silage heaps features a silage selection, so it was necessary for them to navigate between the silage heaps, by using the back-navigation instead of the switch buttons on top of the dashboard. In future versions, it should be better indicated that it is possible to select more than one silage heap before continuing to the dashboard. Another suggestion was to change the design of the button with which one can navigate to the dashboard so that it is clear that it is deactivated while no silage heap is selected. The button could also indicate the number of selected silage heaps for further clearance.

The widgets mostly gained positive feedback, but only one participant indicated that the shown data could be displayed on one site. However, the charts were missing proper unit descriptions, as probands stated.

While we asked to find values like height and dimensions of the silage heap to encourage using the 3D model, many probands stated that these values should be accessible as written numbers because using the 3D model for that purpose was not very precise.

Another common feedback was that the button, which toggles the drawer, has an unintuitive default state. As long as no widget is selected for the detailed view, it shows an open-drawer icon, which does not fit the fact that it has no function at this moment.

The information about the owner's name seemed to be hard to find. While the name of our test data did not represent an actual name, fields with such information should be tagged.

5.2 Development Evaluation

With the development evaluation, we estimate whether the chosen setup and the implementation of our data types are intuitive with regard to future development of the created system. Therefore, the probands are asked to implement changes to a module which we created for the purpose of this evaluation.

5.2.1 Goals

With the help of a questionnaire, we want to validate the created development environment, to see whether it is possible to create new widgets easily and how well the configuration of widgets were implemented.

5.2.2 Questionnaire

To get feedback from our probands, we again created a questionnaire. The questionnaire can be found in Appendix B. It features an introduction, a form for personal data, tasks describing the features to implement, and questions to evaluate the data structures and how intuitive the implementation was.

Rating

We use the same rating system as with the use case evaluation, where grades are rated from one to five, with one being a strong disagreement and five meaning that the proband fully agrees. There are fields for text-based feedback, too.

Personal Information

The questionnaire contains a section requesting personal data from our probands. The data is used to correlate skills regarding software development with the results of the evaluation. The information collected in this survey can be seen in Table 5.5.

Table 5.5. Personal data received by the the probands.

ID	Profession
DA1	Student
DA2	Researcher
DA3	Practitioner
DA4	Other

ID	Prerequisites
DA5	Programming
DA6	UI/UX
DA7	JavaScript
DA8	Typescript
DA9	HTML/CSS
DA10	Visual Studio Code

Statements

The statements focus on the design of features and components of the created setup. The following table lists statements that were answered by the probands (Table 5.6).

Table 5.6. Statements regarding the development evaluation.

ID	Statement	Grade/Text
DB1	How intuitive was the implementation of the new features?	
DB1-1	Title	Grade
DB1-2	Size	Grade
DB1-3	Custom widget	Grade
DB1-4	Chart	Grade
DB1-5	Color	Grade
DB2-1	How easy was implementing the chart?	Grade
DB2-2	Were there any problems?	Text
DB3	How useful is the possibility of using different data types within the properties in your opinion?	Grade
DB4	Did the Typescript typing (auto-complete, type checking) make the given task easier to solve?	Grade
DB5	How would you rate the WidgetState system with its life cycle to create different widgets?	Grade
DB5-1	Custom components	Grade
DB5-2	Configuration interface	Grade
DB6	Did any problems occur while developing?	Text
DB7	Additional remarks.	Text

Experimental Setup

Due to the fact that the evaluation is executed in a small amount of time, probands do not have to use any documentation to fulfill the requested tasks. Furthermore, any problems

that occur during the evaluation, which could be resolved by reading documentation about our project, third party modules or the used languages, are given at any point. The probands should be able to complete every task.

The project is opened in a Visual Studio Code instance. After reading the questionnaire, the probands are briefly informed about the given setup and an example widget configuration is shown to them. While implementing the new features in a widget, the probands get all the information they need. Then they evaluate the process.

The goal was to implement a new widget, which is then displayed in the application. The probands implement fields of the widget configuration, components of the component module, and make changes to the styling of widgets within the application. It is needed to alter the title of the widget as well as add another field for the size of the widget. They modify the widget to use a custom widget component and use JSX code to display a new widget with the title, a button, and a chart. Finally, the probands change the background color for all widgets, which are displayed in the application by using a CSS file.

5.2.3 Results

The following shows the results of the development evaluation. All the statements the probands gave can be found in Appendix B.

Personal Information

In this evaluation, 10 people participated. Six of the probands stated that they were students, four stated that they were researchers, while one of the probands stated that they were practitioners. The data regarding the prerequisites is listed below in Table 5.7.

Table 5.7. Statements regarding prerequisites.

ID	Mean
UA5	4.1
UA6	3.1
UA7	2.6
UA8	2.5
UA9	2.8
UA10	3.0

Evaluation

The following shows some of the data that we gathered through the evaluation (Table 5.8).

Overall, the probands agreed with the given statements. Only one proband had problems with implementing the fields of a WidgetState. This participant stated to not have any

Table 5.8. Statements regarding the development of features within our project.

ID	Mean
DB1-1	4.5
DB1-2	3.7
DB1-3	3.7
DB1-4	3.7778
DB1-5	4.7
DB2-1	4.375
DB3	4.75
DB4	4.9
DB5	4.3333
DB5-1	4.5556
DB5-2	4.2222

experience with the languages. While many probands said that the property named *big* is not intuitive enough, the implementation of the properties was possible.

5.2.4 Discussion

The discussion of the development evaluation can be found in the following. Therefore, we use the presented results (Table 5.7, Table 5.8), as well as the statements the probands have given and our own observations.

Personal Information

One proband stated to be a practitioner. This proband also gave high values for the fields of programming skills, UI/UX, JavaScript, HTML/CSS, and Visual Studio Code skills. While most probands indicated that programming is one of their prerequisites, only four of them stated that they have high JavaScript skills and three of them stated that they have high Typescript skills.

Evaluation

Most of the exercises were intuitive to accomplish for many probands.

Regarding the size of widgets, many suggested that a property like *size* would be more fitting, which would take a number to indicate the span of a widget instead of a boolean, which indicates whether the widget has twice the size. Another idea a proband had was to enable widgets to span over half the length, so that two widgets of the same size would fit a row instead of three.

All probands agreed that Typescript helped with implementing the new features. The suggestions through Visual Studio Code, as well as the error messages, were helpful in

most cases.

Naturally, the implementation of the custom widget component was the most difficult exercise. Many probands set the wrong type at first, resulting in the WidgetDisplay having the *custom* type. Then it was necessary to combine JSX with Typescript, which for many probands was confusing at first. Implementing the show-button within the custom component was difficult because the show-function had to be wrapped in an anonymous function to work with the Typescript-types of the button component. The show-function also features an argument, a WidgetStateWrapper, which was confusing, too. The argument should be at least optional, because in most cases, a developer would want to use the corresponding WidgetStateWrapper of the WidgetState they are currently developing.

Another common note was that the chart component would go beyond the borders of the widget in the vertical direction by default, when only defining the *data* property of a chart. While this is unintuitive at first, it can be solved with optional properties. One could set the *maxHeight* property, for example, because widgets have a constant height over all screen sizes. However, it would be better to make the chart fit the rest of the space available in a widget by default, which might be difficult due to the number of elements a widget has to display.

5.3 Conclusion

As pointed out, the application needs to be improved in terms of usability, where the navigation within the application was the main problem. The project environment can be further improved by extending the possibilities of the configurations and changing types for a more intuitive use while developing.

Chapter 6

Related Work

In this chapter, similar approaches to our problem are presented. Therefore, we look at the technology called *Grafana*[1], which is a toolbox for building chart-driven dashboards. Furthermore, we also examine the *Titan Control Center*[2], which is a technology for the visualization of sensor data in the field of industrial big data analytics. Finally, we present a dashboard implementation for *ExplorViz*[3], a project for visualizing software projects.

Grafana comes with a free version, which is based on an open source project, and a professional version. The application features several tools to connect data from different origins by using an editor. It is not necessary to have programming skills to create new chart visualizations.

Grafana works with several technologies, like different types of databases. Due to the data which we display potentially being of any origin, we can not guarantee that Grafana would be suited for future requirements.

The Titan Control Center [Henning and Hasselbring 2021] is an open source technology that allows to create dashboards (Figure 6.1) for large scale industry by being able to connect a large number of *IIOT* (Industrial Internet of Things) sensors into data almost in real time. Among other things, it is possible to implement anomaly detection, forecasting, and aggregation by using *microservices*, which are added to the application by subscribing to an event stream.

While the Titan Control Center is suited for large-scale sensor data processing, the requirements of the SilageControl project demand a flexible web application. The Titan Control Center therefore does not meet this requirement, as it is not possible to create arbitrary web applications with it to fit requirements for which a dashboard can not be the solution. Furthermore, the data which is displayed in our application is preprocessed in contrast to the data which is aggregated in a Titan Control Center application.

[1] https://grafana.com/
[2] https://www.softwareimpacts.com/article/S2665-9638(20)30041-5/fulltext
[3] https://www.explorviz.net/

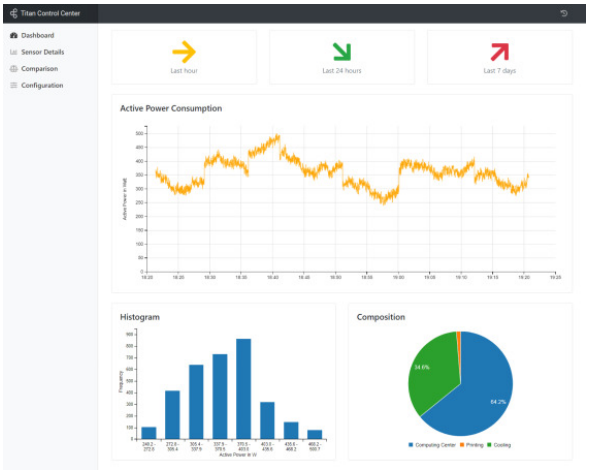

Figure 6.1. A visualization with the Titan Control Center [Henning and Hasselbring 2021].

Krippner and Hasselbring [2019] shows an implementation of a dashboard for ExplorViz. The dashboard is built with *Ember.js*[4], a framework for creating website projects. The charts are realized with *Chart.js*[5]. A user can configure a dashboard with the help of the UI. It provides different widgets for analytic purposes in relation to the ExplorViz visualization of software projects. These contain information about, for example, the occurrence of programming languages.

[4]https://emberjs.com/
[5]https://www.chartjs.org/

Chapter 7

Conclusion and Future Work

In this chapter, we want to summarize the results of the evaluations and our own experience while developing the project. We also want to summarize the potential improvements to our project, as well as the application we created.

7.1 Conclusion

As part of this thesis, we created a Node.js project with Typescript support as a basis to build multiple applications in the context of the production of silage. While this requires the analysis of data trends, it was necessary to implement UI components with the aim of displaying several data sets and information that will be gathered in the future. Our approach was to implement a widget system, containing a dashboard as well as detailed views, which can be implemented in different applications due to the modular design. To display data, we implemented several types of charts and developed a life cycle for widgets so they can fetch data remotely.

With the created components, we built an application that can be downloaded as a PWA and is responsive over several screen sizes. We also embedded a plotted 3D model within the application.

For validation of the project and the application, we decided to make two evaluations. One evaluation focused on the usability of the application, where probands were asked to fulfill different tasks with the application, to find problems with navigating and displaying data. The other evaluation focused on the usability of the created modules, where probands developed certain features within the application. While the evaluation of the application itself indicates that there are some problems with the UI/UX design, the development environment in which the application was created got mostly positive feedback.

Concluding, the project and its modular design are a promising basis for future implementations, while the application we created needs some improvements to be of good usability.

7.2 Future Work

As more requirements are formulated, there will be a need for more components. With real world data, it will be necessary to further improve the application. It is also necessary

to conduct additional evaluations, especially regarding the usability of the application. It would also be of use to further advance the types used by the WidgetState. For example, the basic fields like *title* and *big* have some potential regarding their possibilities to develop new widgets, where fields like *title* could additionally offer types similar to the custom components of a widget, which are part of the widget's life cycle, while the size of widgets could be represented by a number, to provide more possibilities regarding the span of widgets over the dashboard.

As of now, there is no login system. It may be of use to implement the login process in a new module, making it reusable for multiple applications, while at the same time guaranteeing that it is protected against attacks in all implementations.

Further improvements to the localization are necessary to create an accessible application. Therefore, the current localization should be refactored and could be extended to provide additional languages.

With a backend, it will be possible to define Typescript-types, which would help process remotely fetched data within the application.

Though it is possible to implement composed charts, it is currently not used in the application. However, this could be implemented for the purpose of comparing charts with each other, while this would need further evaluation to verify whether there is any benefit.

Appendix A

Use Case Evaluation

Evaluation – Use Cases

Given is an application, featuring several widgets, which represent aspects of different silage heaps.

General Personal Data

Profession: Student | Researcher | Practitioner | Other

Prerequisites:

agriculture (1 – 5)

programming (1 – 5)

UX / UI (1 - 5)

Introduction

What is silage, why would you want to monitor?

A silo is created to ferment, among other things, grass and grains. This is important because the grass would mold otherwise. The fermented grass is later used as fodder for animals. Silage is condensed with the help of tractors, which is often done by wage workers. To optimize the fermentation, it is needed to monitor the process and to be able to estimate trends over time as well as comparing silage heaps with each other.

What is monitored?

The volume of a silage is monitored with the help of 3D capturing by sensors, which are attached on a rolled tractor. Weather data is fetched from online services.

The data used for the evaluation is mocked for testing.

Tasks

1. Silage heap 1:

Find out what temperature was recorded at 11h.

Did the volume change in the next hour?

What is the maximum height of the silage heap?

2. Silage heap 2:

Who is the owner of this silage (customer)?

What is the difference between the volume of Silage 1 and Silage 2 at 12h?

What are the dimensions (x + y) of the silage heap?

Figure A.1. Use Case Evaluation - questionnaire.

Table A.1. Use Case Evaluation - personal information.

ID	Profession	Agriculture	Programming	UI/UX
U1	student, researcher	1	4	3
U2	researcher	2	5	5
U3	student	1	3	2
U4	student	1	4	3
U5	researcher	1	4	2
U6	student	1	1	1
U7	student	1	4	4
U8	student	2	4	3
U9	practitioner	1	5	4
U10	researcher	2	4	3
U11	practitioner	5	2	1
U12	student, researcher	4	5	4
U13	researcher	4	4	3

Table A.2. Use Case Evaluation - statements part 1.

ID	UB1-1	UB1-2	UB2	UB3	UB4-1	UB4-2	UB5-1	UB5-2	UB6
U1	4	Öffnen des zweiten Silagehaufens	5	3	y	Öffnen von anderen Datensätzen; Manövrieren im 3D Modell	5	Einheiten von Skalen fehlen teilweise; Ablesen von Dimensionen des 3D Modells	-
U2	5	No problem!	5	4	n	I did not have a problem	5	-	-
U3	4	Warum nicht alles auf eine Seite?	4	4	y	Homepage und/oder back button	5	-	-
U4	2	Button zum Auswählen war nicht statisch, Textboxen waren nicht intuitiv zum Auswählen	2	3	y	Es war nicht von Vornherein klar, wo man navigieren konnte und wo evtl. Informationen versteckt waren	4	Man konnte die Anwendung verwenden und alle Diagramme lesen	Sporadisches Design, welches für nicht-affine Personen schwer sein kann zu verstehen; Informationen müssen offensichtlicher sein und bei wagen Informationsinhalten vielleicht wenigstens Verschachtelung
U5	3	Es fehlen Back-Button in der Anwendung; Die einzelnen Elemente waren nicht einheitlich	4	3	n	Zurück-Button würden helfen	5	-	Unterschiedlich große Buttons und Überschriften in der Volumen-Ansicht; Stammdaten fehlen (...) für die dargestellten Attribute
U6	4	Zurückknopf	5	4	y	2x fehlte Zurückknopf	5	-	-
U7	3	Beim Zurückgehen kommt man direkt wieder in die Silo-Liste, obwohl man es in der 3D-Silage-Ansicht nicht erwarten würde	5	3	n	-	5	-	-
U8	4	geringfügig Kontext unklar	5	4	n	Graphiken z.T. etwas schwer zu interpretieren, kurze Erklärung vllt hilfreich	5	-	-
U9	4	Wechseln zwischen Silagen / zurück zum Start nicht über Buttons möglich / Option nicht auffindbar	5	3	y	Siehe 1.1	5	Alles ging :)	Drawer Button hat inkonsistentes Verhalten
U10	4	besser nicht alles auf einer Seite, besser Reiter o.ä.	5	2	nein	-	4	Nachmessungs-Volumen	Maßeinheiten!; Platzierung; fehlende

Table A.3. Use Case Evaluation - statements part 2.

ID	UB1-1	UB1-2	UB2	UB3	UB4-1	UB4-2	UB5-1	UB5-2	UB6
U11	4	aaf. die Fragen noch ausführlicher stellen und genauer erklären worum es geht	4	5	nein	-	5	Keine Probleme	Die Übersichtsseite ist sehr übersichtlich und gut angeordnet; Die Detailseite gefällt mir auf Grund der Größe der einzelnen Abbildungen und der Anordnung untereinander sehr gut
U12	4	s.u. Menu Button, 1 White Screen, Platzierung der Button	5	4	y	Zurück-Button fehlte; Menu Button	5	-	Menu Button eher oben links; Symbol für Menu Button teils verwirrend; Hatte 1 Whitescreen; ein Zurück Button wäre schön; "Show" eher unter dem Diagramm, "Reload" eher oben rechts; Einheiten an den Diagrammen; Vor "Customer 2" noch "Besitzer*in" schreiben; Bei der oberen Leiste sieht es so aus, als könnte man sie anklicken; Titelleiste bei 1 Silo nicht rendern; Anzeige von mehreren Silos nicht intuitiv; In der Liste der ... erst etwas mysteriös
U13	4	Die Auswahl der Silagehaufen ist intuitiv, bei längeren Listen oder unklaren Namen könnte ich mir jedoch vorstellen, dass dies komplizierter wird	-	4	y	Von der Detail Silo Ansicht zurück war der Button etwas schwer erkennbar	4	-	Die Aufteilung der Informationen in die Widgets fand ich sehr intuitiv & gelungen. Auch die Möglichkeit sich Detaildaten anzeigen zu lassen finde ich super

Appendix B

Development Evaluation

Evaluation – Development

Given is a widget module featuring a widget configuration, which implements a data widget. The overall task is to get familiar with the WidgetState system and implement minor changes to the given widget, as well as the evaluation of this process. The evaluation will be focused on the created systems usability, how difficult it was to find solutions for the tasks and whether the coding environment is intuitive. The overall goal of this evaluation is to find out, whether the systems architecture allows to create widgets easily and is intuitive to use, help with problems regarding the used languages, as well as obscurities, which are documented, will be given at any point.

General Personal Data

Profession: Student | Researcher | Practitioner | Other

Prerequisites:

programming (1 – 5)

UX / UI (1 – 5)

JavaScript (1 – 5)

Typescript (1 – 5)

HTML/CSS (1 – 5)

VS-Code (1 - 5)

Tasks

1. The command ‘ npm run lohna:dev ‘ starts the application, while ‘ npm run evaluation:dev ‘ starts a Typescript compiler, which watches for changes.

2. Add the widget to the application. The widgets can be found in the file packages/apps/lohna/src/widgets/index.tsx.

The widget needs to be added to the list of widgets, which is a property of the WidgetGrid.

Check whether the widget is displayed correctly.

3. Change the widgets title in <h2>Example widget</h2> and make it use two fields (big).

4. Change the widget to render a custom widget component. The custom component is composed of the title and a show button, as well as a chart.

5. Change all widgets background color (‘#ff0000’) for the application. The styling can be found in the file packages/apps/lohna/src/Style.css.

Figure B.1. Development Evaluation - questionnaire.

Table B.1. Development Evaluation - personal information.

ID	Profession	Programming	UI/UX	JavaScript	Typescript	HTML/CSS	VS Code
D1	student	4	3	4	4	5	3
D2	student	3	2	1	1	1	2
D3	student, researcher	4	3	1	2	2	3
D4	researcher	4	2	1	1	3	1
D5	student	4	4	3	3	3	3
D6	student	4	3	4	4	3	4
D7	practitioner	5	4	5	3	4	4
D8	researcher	4	3	1	1	1	2
D9	student, researcher	5	4	4	4	4	4
D10	-	4	3	2	2	2	4

Table B.2. Development Evaluation - statements.

ID	DB1-1	DB1-2	DB1-3	DB1-4	DB1-5	DB2-1	DB2-2	DB3	DB4	DB5	DB5-1	DB5-2	DB6	DB7
D1	5	3	4	3	5	-	Nicht von der Chart-Component so wirklich gewusst	4	5	4	4	3	Nicht genau gewusst, welche Komponenten tatsächlich existieren und welche props sie haben	-
D2	2	2	2	2	2	5	fehlende Erfahrung	-	5	5	5	5	fehlende Erfahrung	-
D3	5	3	4	4	5	4	Wurden durch Fehlermeldungen erklärt	5	4	4	5	4	Programmiersprache war nicht bekannt	-
D4	5	4	4	4	5	4	Nö	5	5	3	4	4	Nö	-
D5	5	5	5	5	5	5	nein	5	5	5	5	5	nein	-
D6	5	4	4	4	5	4	Chart sprengt Rand des Widgets	5	5	4	4	4	Wissenslücken in HTML, aber Dokumentation sehr hilfreich dabei	-
D7	5	5	3	4	5	4	Durch fehlendes jsx/React Wissen beim Umgang, sonst easy	5	5	5	5	3	Das show ein Argument benötigt und deshalb in eine Funtion gewrapped werden muss	Typescript rules!
D8	5	4	5	-	5	-	-	-	5	-	-	-	-	-
D9	4	3	3	4	5	5	-	5	5	4	5	5	Chart läuft über -> CSS	Widgets scheinen sehr flexibel konfigurierbar zu sein
D10	4	4	3	4	5	4	Sobald man den Tag hatte, war die Implementierung sehr intuitiv	4	5	5	4	5	Nachdem man einen Überblick hat, ist es sehr intuitiv ein neues Widget anzulegen & die vorgefertigten Module helfen sehr	Ein sehr gutes Konzept, um die Daten anzuzeigen. Die Modularität der Anwendung ist sehr gut auf die Daten abgestimmt

Bibliography

[Alhammad and Moreno 2022] M. M. Alhammad and A. M. Moreno. Integrating User Experience into Agile. In: ACM, May 2022. (Cited on page 5)

[Bogner and Merkel 2022] J. Bogner and M. Merkel. To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub. In: ACM, Mar. 2022, page 12. (Cited on page 7)

[Brehmer 2019] M. Brehmer. Techniques for Data Visualization on both Mobile & Desktop. In: Apr. 2019. URL: https://www.visualcinnamon.com/2019/04/mobile-vs-desktop-dataviz/. (Cited on page 6)

[Brehmer et al. 2019] M. Brehmer, B. Lee, P. Isenberg, and E. K. Choe. A Comparative Evaluation of Animation and Small Multiples for Trend Visualization on Mobile Phones. In: Oct. 2019. (Cited on page 6)

[Chinthanet et al. 2021] B. Chinthanet, B. Reid, C. Treude, M. Wagner, R. G. Kula, T. Ishio, and K. Matsumoto. What makes a good Node.js package? Investigating Users, Contributors, and Runnability. In: ACM, June 2021, page 20. (Cited on page 5)

[Henning and Hasselbring 2021] S. Henning and W. Hasselbring. The Titan Control Center for Industrial DevOps analytics research. In: Elsevier B.V., Feb. 2021. (Cited on pages 43, 44)

[Krippner and Hasselbring 2019] F. Krippner and W. Hasselbring. Design und Implementierung eines Dashboards für ExplorViz. In: Sept. 2019. (Cited on page 44)

[Lam et al. 2019] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In: July 2019, page 11. (Cited on page 6)

[Romano et al. 2021] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang. An Empirical Analysis of UI-based Flaky Tests. In: IEEE, Mar. 2021, page 13. (Cited on page 6)

[Wu et al. 2020] A. Wu, W. Tong, T. Dwyer, B. Lee, P. Isenberg, and H. Qu. MobileVisFixer: Tailoring Web Visualizations for Mobile Phones Leveraging an Explainable Reinforcement Learning Framework. In: Aug. 2020, page 11. (Cited on page 6)