ulm university universität ulm

Development of a web-based application for the degradation configuration of IoT components

Steffen Knoblauch
1045346

Project report
VS-2021-20P

Examined by
Prof. Dr.-Ing. Franz Hauck

Supervised by
M.Sc. Alexander Hess

Institute of Distributed Systems
Faculty of Engineering, Computer Science and Psychology
Ulm University

July 14, 2021

Issued: July 14, 2021

I hereby declare that this thesis titled:

Development of a web-based application for the degradation configuration of IoT components

is the product of my own independent work and that I have used no sources or materials other than those specified. The passages taken from other works, either verbatim or paraphrased in the spirit of the original quote, are identified in each individual case by indicating the source.
I further declare that all my academic work was written in line with the principles of proper academic research according to the official "Satzung der Universität Ulm zur Sicherung guter wissenschaftlicher Praxis" (University Statute for the Safeguarding of Proper Academic Practice).

Ulm, July 14, 2021

Steffen Knoblauch, student number 1045346

CONTENTS

1 INTRODUCTION

More and more IoT (Internet of Things) devices are finding their way into our daily lives, making it possible to combine a wide variety of end devices and sensors. This makes it possible to design new composite systems that combine the functionalities of the subcomponents. Such systems can encounter internal problems, for example, when a component no longer operates properly. The system has to react to this and, for example, cease functionality or operate in another way.

This is where the SORRIR project comes in, in order to develop a self-organising, resilient execution platform for IoT services. To make such a system resilient, i.e. resistant to errors of the subcomponents of the system, different levels were introduced in which such a system can operate.

A system that is composed of several subcomponents may be forced to change the level if, for example, one of these subcomponents is no longer functional. In the worst case, up to the complete deactivation of the system. In order to prevent a system from going directly into the deactivated state whenever an error occurs, various intermediate levels, the so-called degradation levels, can be defined. For example, a degradation level can continue to provide some of the original functionality, but with some limitations depending on the not-working subcomponent.

An example for such a system is a smart barrier in a parking garage, which in the ideal case recognizes a car by means of a camera and is opened automatically. The ideal case means that all subcomponents are working correctly. If, for some reason, the camera failed, the system would no longer be able to function. However, with the introduction of degradation levels, a part of the functionality could be guaranteed. A fallback for the system could be that the barrier can be manually opened with an ID card. To configure such degradation levels a JSON file is required which defines the different degradation levels and the transitions between them. For example, what should happen when a certain subcomponent is no longer functional.

This is the starting point of this project, in which the goal is to create a graphical interface in form of a web application. This web application will be used to configure a system with the different degradation levels and finally export it as JSON file. In order to change or create a configuration, several components for the web application are required, like the definition of subcomponents or the import of an existing configuration.

This report starts with a short description of the technology stack, in section 2, that was used to implement the web application. It is followed by the architecture, in section3, that includes a description of the general components, their usage, and some additional information about the implementation. The last section is the conclusion that contains a short summary as well as potential further enhancements.

2 TECHNOLOGY STACK

The goal of this project is the development of a web application and due to the large number of different frameworks and libraries, the selection was relevant from the beginning. To develop a web application it is possible to develop without a framework or library, but nowadays there are also various frameworks available to facilitate the development, such as React, Vue or Angular.

The choice for this application fell on React, because React is not only very popular [3], but also allowed me to learn the required fundamentals and use it for the first time. React is a JavaScript library to create single-page applications, but also provides typings for TypeScript [2]. In order to facilitate future development, the choice fell on TypeScript. TypeScript provides a more structured approach for the development of web applications, for example by defining interfaces as model classes that can be validated at compile time.

With these technologies as a starting point, the next step is the decision of what build system should be used. It would be possible to use a custom build system based on webpack or similar bundlers, but for React there is an officially supported way: create-react-app. This is a npx package that will set up a single-page application with all the required configuration, like the bundler and the typescript compiler. It is possible to later exchange and adapt parts of the configuration or the build system, but for this project it was not required to further adapt the default setup.

In addition to react some additional libraries were used:

- **material-ui**: a library that offers standard material design components
- **react-syntax-highlighter**: a syntax highlighter that is used to display the configuration as JSON in the import and export
- **ajv**: a validator that is used in the import to validate that the JSON fits the configuration schema
- **react-dnd**: a drag and drop library for react

3 ARCHITECTURE

This section will give an overview of the general concepts and will show with examples how to use the application.

3.1 GENERAL CONCEPTS

The complete code was structured with too general ideas in mind. On the one hand it should be modular in the sense that the application consists of components that can be reused on multiple occasions. For example a dialog for the deletion can be always the same and only the handling and the description text have to be adapted based on the usage. The modularity will be introduced implicitly by React, because the general idea is the creation of components that will be updated internally by react.

Another relevant aspect was the future extension of the application, in order to be able to integrate more steps for the creation of the configuration. For example adding additional properties for the subcomponent configuration.

3.2 APP-COMPONENT

The App is the central component that will contain all the relevant components for the application. The App keeps track of the current configuration and provides the access for the components via a context provider. This allows the access and update of the configuration from all the components, which will also trigger the React lifecycle to re-render a component.

3.3 WELCOME PAGE

The welcome page is the first component that will be displayed when you start the application. In figure 3.1 you can see that there are two buttons that allow to start the configuration. There are two choices on how to start the configuration, the start with an import and without an import, based on the choice a user makes the wizard (compare section 3.4) will be started with different steps.

3.4 WIZARD

The creation of the configuration can be divided into several steps. To facilitate the configuration, a wizard has been developed that allows the step-by-step configuration. In addition to that it is also possible to navigate arbitrary back and forth between the different steps of the configuration.

The wizard component allows to dynamically configure what steps will be shown and in what order. The wizard consist in general of a menu-bar, a stepper to show navigate between the steps and the content it will show which is internally called a view.

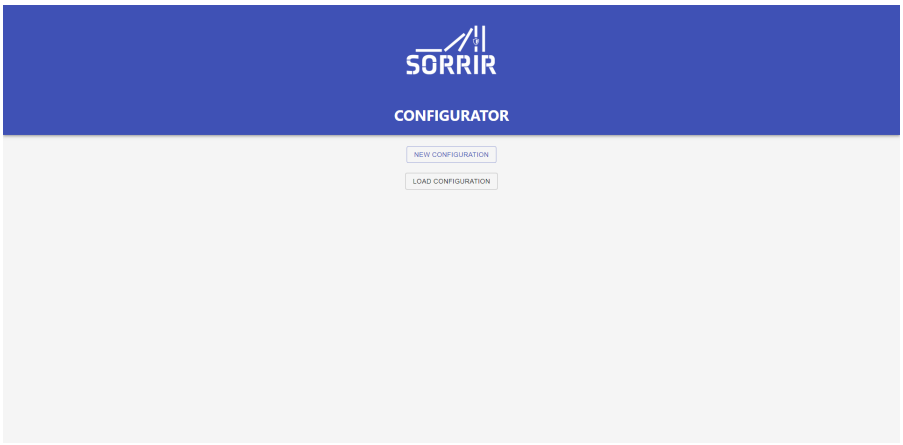

Figure 3.1: Welcome page

Figure 3.2 shows the menu bar and the stepper for the configuration with an import. You can navigate through the different steps by clicking on the steps in the stepper, whereby the active step is highlighted. The menu bar will update its label with the label that was specified for the current step, for example in the figure the label was set to "Import".

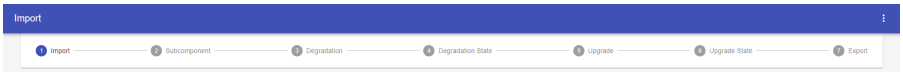

Figure 3.2: Wizard

In addition to the navigation between the different steps there is a menu on the top right. At the moment this only has one menu item that allows to restart the configuration, but it is possible to further add more functionality. The restart will show a dialog to confirm the reset of the current configuration and will, after the confirmation, send the user back to the welcome page.

In order to extend the wizard with additional steps following actions have to take place:

- Extension of the view enum (a enum that contains all available views in the application) ("src\util\Views.ts")
- Setting a label for the new view ("src\util\ViewLabelResolver.ts")
- Adding the view that will be displayed for the step in the wizard. This has to be a react component ("src\components\wizard\ViewSelector\ViewSelector.tsx")

After these steps the wizard can be configured to display the new view as an additional step.

3.5 IMPORT

The import view is only inserted as a step when the wizard is started with the "LOAD CONFIGURATION" button on the welcome page. The import page

allows to import an existing configuration file and will update the current configuration in the app. The import will also override all previous changes in the configuration. The configuration will be displayed with syntax highlighter that will be updated with the new configuration after a successful import.

There are several cases when the import has to fail to enforce a correct state of the configuration and therefore it will be validated that file contains a valid configuration. If this is not the case the import will display all errors that occurred during the import validation. The first part of the validation is to check if the file contains a string that can be parsed to a JSON object. if this is not the case an error message similar to the message in figure 3.3 will be displayed.

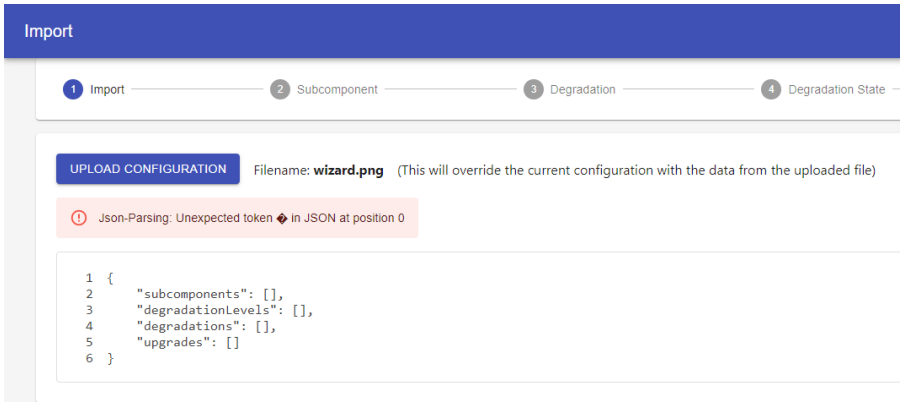

Figure 3.3: Import invalid file

The second part of the validation is the validation of the JSON data itself. The JSON data requires several properties to be a valid configuration. Details on this part of the validation can be found in section 3.6. The found problems of the second part of the validation will also be displayed. An example can be seen in figure 3.4 where there are several problems with the imported object. If a problem occurs for an element of an array, for example in the subcomponents, the index of the element will also be added to the error message. This will help a user to identify the problem with the imported file and update it accordingly.

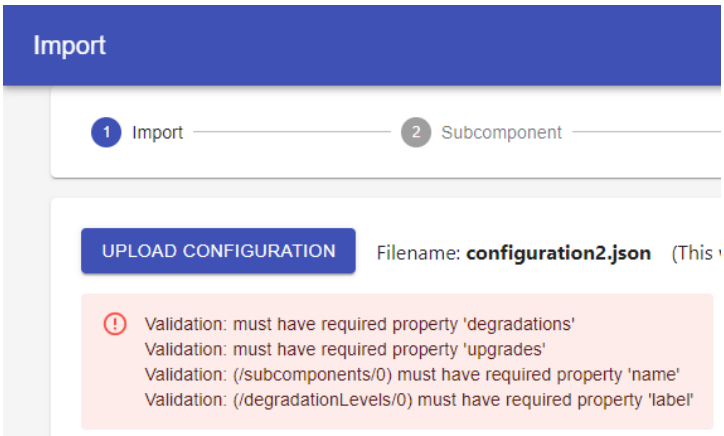

Figure 3.4: Import invalid JSON

3.6 VALIDATION

The validation of the configuration at the import is a central part of the application, because it enforces a valid configuration. Another validation is not necessary because all current configuration steps will always update the configuration correct and end up in a valid state.

For the validation the *ajv JSON schema validator*[1] was used, because it has several advantages over a custom implementation. A custom implementation requires not only to write the complete parsing logic, but also has to be adapted if changes to the configuration schema are required in the future. The configuration schema is a description of all the properties a JSON data has to have to be a valid configuration.

There are two different schema languages that allow the definition of JSON data. On the one hand there is the *JSON schema*[2] and on the other hand there is the *JSON Type Definition*[3]. Both schema description languages are supported by ajv, but the decision was made in favor of the JSON schema. Although there is an RFC for the JSON Type definition and only a draft for JSON Schema, industry support is not yet entirely in place [1]. Since both are supported by ajv, the decision was made to use JSON Schema because of the better support, but this can easily be changed to the JSON Type Definition.

The change from the JSON Schema to the JSON Type Definition is for this application quiet simple, because you just have to replace the schema file and adapt the configuration of ajv, which can be found in src\util\

3.6.1 JSON SCHEMA

This section provides a short overview of the configuration schema, but the complete definition can be found in the source code.

The first part of the configuration are the subcomponents. A system that has to be configured consists of several subcomponents, these are stored in an array in the configuration. The subcomponents are highlighted in green in figure 3.5 a together with the shadowmodes. Each subcomponent has a set of shadowmodes that are all the operational states a subcomponent can have. For example the typical states are an *on*-state or an *off*-state.

The next part of the configuration are the degradation levels which represent the levels in which the system can be. In figure 3.5 the blue highlighted parts are all relevant parts that are related to the degradation levels. The degradation level dependencies are a set of all the states the subcomponents have to be in for a degradation level or in other words the dependencies of a degradation level. Only the subcomponents and the states that are relevant for a level are stored in this array and the *don't care* cases will be ignored to save storage space. The degradation level states are the internal states of each degradation level. They describe the states the state machine of each degradation level can be in.

The central part of the configuration are the upgrade and degradation of the system, in figure 3.5 highlighted in red. Both the upgrade and the downgrade

1 https://github.com/ajv-validator/ajv
2 https://json-schema.org/
3 https://datatracker.ietf.org/doc/html/rfc8927

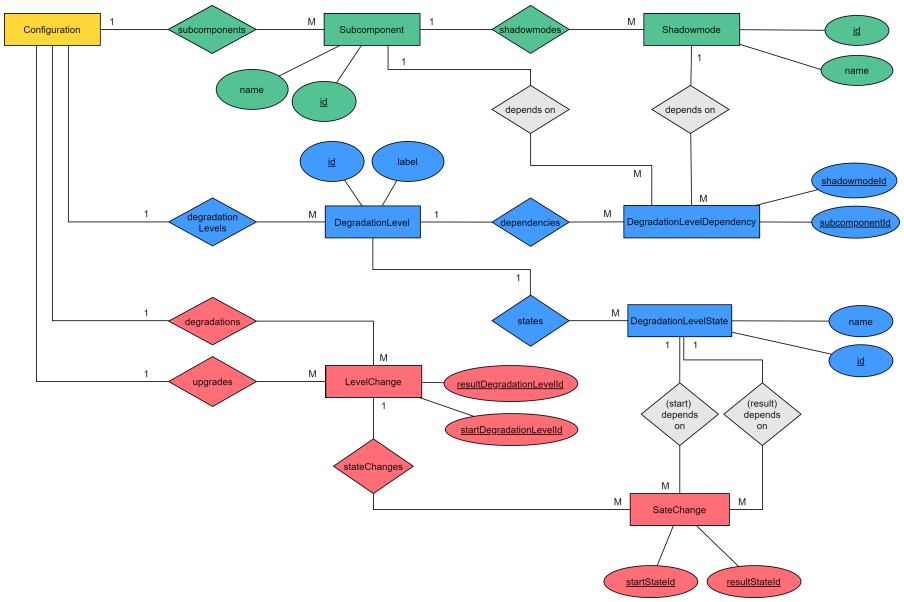

Figure 3.5: JSON configuration schema mapped on an er-diagram

from on degradation level to another degradation level can be represented as level change. A level change object consists of the degradation level where the change starts and where it results in. A simple example could be a system that starts at the off-state and after an upgrade results in the on-state. In addition to the level change, it is also required to define in what internal state the start and the result degradation level has to be. In order to map the states each level change contains a set of mapping of the states of the start and result degradation level.

3.7 SUBCOMPONENT CONFIGURATION

The subcomponent configuration is the first step when you start the wizard without importing an existing configuration. Otherwise it is the second step that allows the adjustment of the imported subcomponents. This component offers a table that displays all defined subcomponents and allows to sort and navigate them and in addition to this the general creation, update or deletion of them. In figure 3.6 an example is shown with several defined subcomponents.

The table allows to select subcomponents and depending on the amount of selected subcomponents different actions are possible. If only one subcomponent is selected it is possible to either delete, edit the selected one or create a new subcomponent (compare figure 3.7 on the top right corner of the table).

If a user selects multiple subcomponents it is only possible to delete all of them or to create a new one. (compare figure 3.8).

For the deletion, the creation and the update of a subcomponent different dialogs are displayed. The deletion dialog is shown in figure 3.9 and shows the amount of deleted subcomponents. The deletion of one or more subcomponent will also delete the references that exist in other components. For example a degradation level depends on a subcomponent and therefore has a degradation level dependency object that will be deleted too.

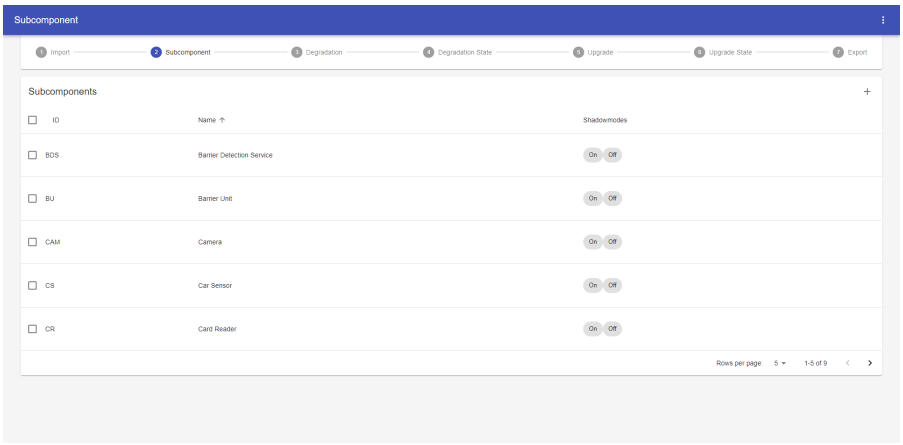

Figure 3.6: Subcomponent view

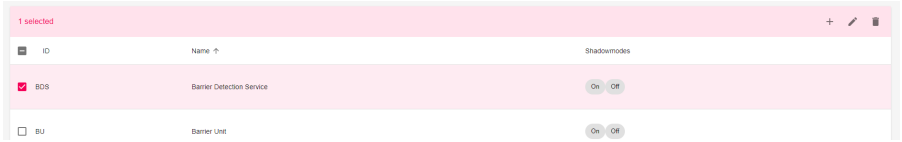

Figure 3.7: Subcomponent single selection

Figure 3.8: Subcomponent multi selection

The create and edit dialog will allow to create or edit a subcomponent. If no name is set for a subcomponent the dialog will autofill the id with the value of the name. In addition to this the dialog will validate if there are empty fields and if another subcomponent already exists with the id set in the dialog. This can be the case when either a new component is created with an existing id or the id of an existing one is changed. Changing the id of an existing subcomponent will also update the dependencies of the degradation levels that depend on this subcomponent. An example of the create dialog is shown in figure 3.10. For each subcomponent it is required to define one or more shadowmode and therefore a custom input was designed. This input allows to enter a value in the text field and after the "enter" button is pressed the value will be added as chip on the right side. For example the *On* or *Off* shadowmodes in figure 3.10. Deleting a shadowmode will also update the degradation level dependencies that rely on this shadowmode.

3.8 DEGRADATION LEVEL MANAGEMENT

The creation, update and deletion of degradation levels is combined with the functionality to define upgrades and degradations, as described in section 3.9.

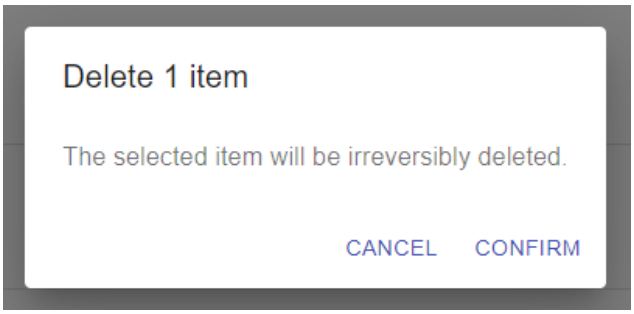

Figure 3.9: Subcomponent delete dialog

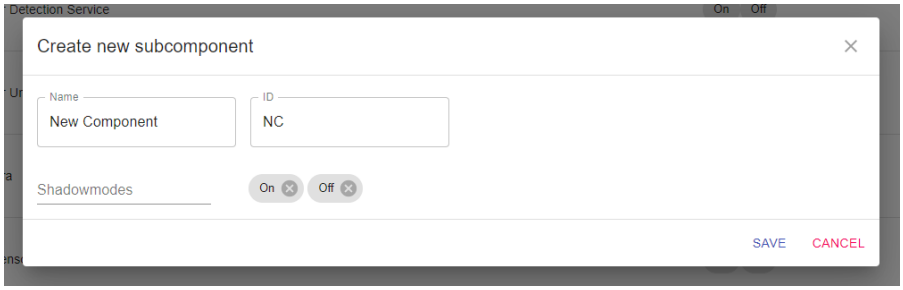

Figure 3.10: Subcomponent create dialog

The degradtion level tree allows to select single and multiple degradation levels and depending on that there are different actions possible. If only one is selected there are three possible actions: creating a new degradation level, edit the selected one or delete the selected one (compare figure 3.11). The selection of multiple degradation levels (compare figure 3.12) allows only the deletion of the selected levels or the creation of a new level. The deletion for both cases will display a dialog similar to the one shown in figure 3.9.

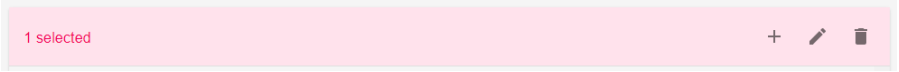

Figure 3.11: Degradation level single selection

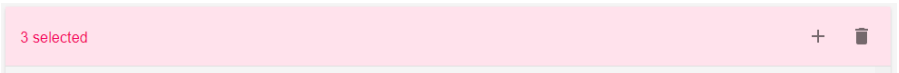

Figure 3.12: Degradation level multi selection

The creation and edit dialog (shown in figure 3.13) allows to either create or edit a degradation level. The dialog will validate the id to enforce unique ids for each degradation level in the configuration. Depending on the subcomponents that are already defined, the dropdown selection inputs will be generated dynamically. These dropdown selection inputs allow to define the dependencies of the degradation level by selecting a shadowmode for each subcomponent. If the shadowmode is set to *DC* (don't care) then no value will be created in the configuration (or an existing value will be deleted in the edit case). The internal states are a custom input that allows again to add a state for degradation level

by entering a string and pressing enter. After the enter button was pressed the state will be added as chip on the right and can be deleted with the X-button. The deletion of the states of a degradation level will also trigger an update of the configuration and will remove all references that require the state (the state changes).

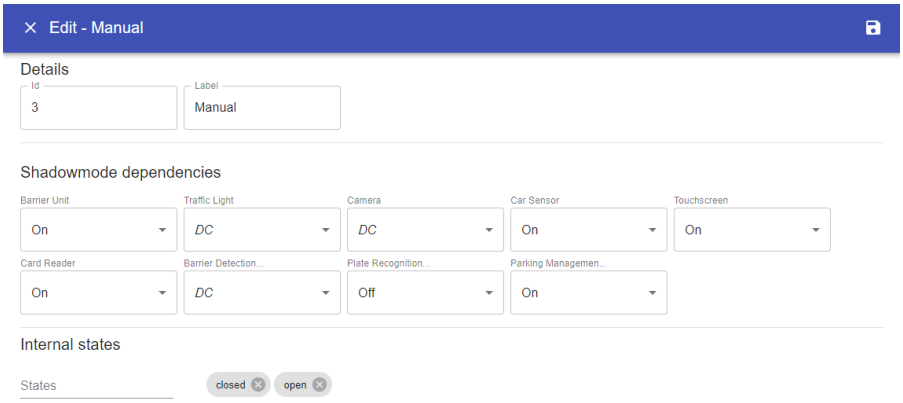

Figure 3.13: Degradation level dialog

3.9 DEGRADATION AND UPGRADE CONFIGURATION

The second part of the degradation level configuration is the specification of the degradations and upgrades. In order to be able to specify the hierarchy of the degradation levels a custom component was created that allowed to drag and drop the degradation levels into a tree structure.

An example for a possible degradation is shown in figure 3.14 that also shows several features. All unsorted degradation levels will be displayed in a sidebar and can be tracked to any position in the tree. If there is no unsorted degradation level left, the sidebar will not be displayed. The selection can vary from one to many selected degradation levels whereby the degradation levels can be in the tree and the sidebar at the same time. An example for the selection is also shown in figure 3.14 where the selection is highlighted in red. In order to remove a degradation level from the tree it is possible to drag the node onto the delete zone at the top. This will remove the degradation or upgrade from the configuration. The degradation level will be added as unsorted degradation level again. The OFF degradation level will always be the top node in the tree and can't be removed or changed. It will have as id 'o' and won't explicitly be saved in the configuration.

The tree allows to insert (drag and drop) degradation levels above or below a sorted level. In figure 3.15 an example for the drop above and below a degradation level in the tree is shown. Hovering over the drop zone (above or below of a node in the tree) will highlight the zone in blue and allows to add the dragged degradation level at this position.

Changing the position of a degradation level will reset the configuration or more precise the level change itself. The level changes stores the start and result state of the degradation levels that have to be configured again in the next view. Inserting the node above another one will therefore update the level changes

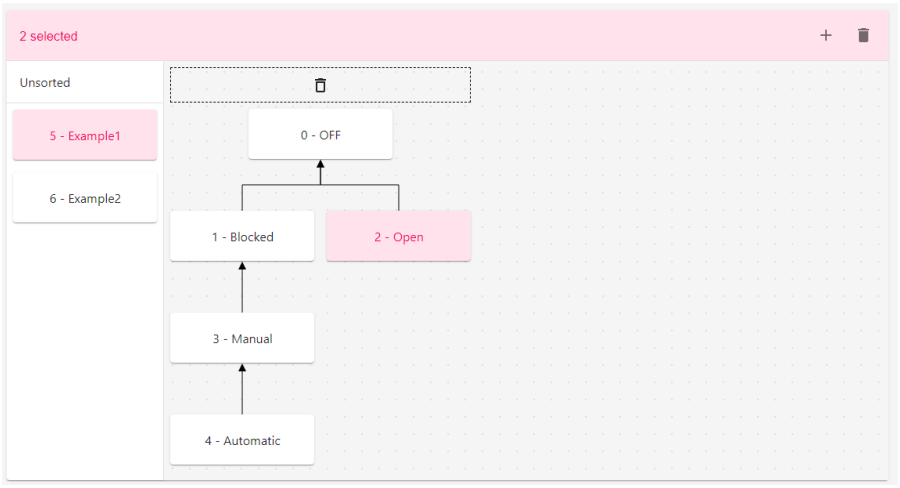

Figure 3.14: Degradation level tree

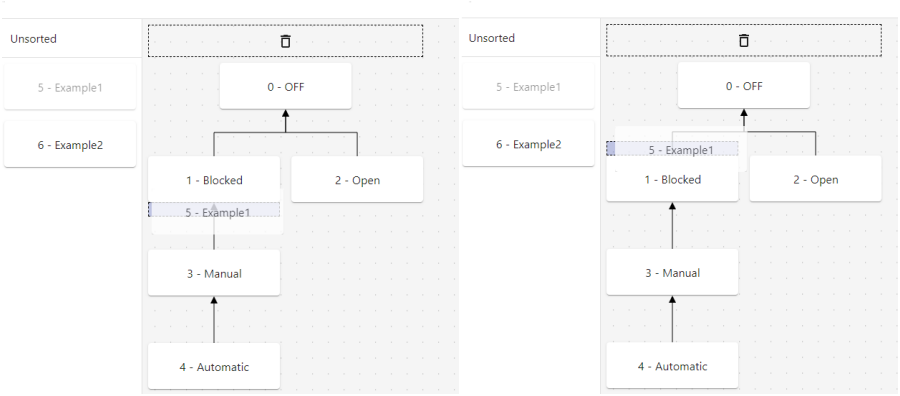

Figure 3.15: Degradation level tree drop behavior (left: below, right: above)

(degradation or upgrade) for the already connected degradation levels, but reset the state changes.

The tree show up to this point is used to define the degradations, but allows not to specify the upgrades. For the upgrades the tree will show an inverse version, which means that the arrows will not start from a degradation level and point in the direction of the OFF degradation level. Instead the arrows will point starting from the OFF degradation level to the child nodes. The specification of the upgrades will be therefore done in a separate step in the wizard that allows a completely independent configuration from the degradations. In order to simplify the process, there is an additional button that allows the automatic creation of the upgrade tree based on the already specified degradations tree. The button, in figure 3.16 in the sidebar on the top right, will open a confirmation dialog that will ask a user to confirm the changes. After the confirmation the previous upgrade tree will be deleted and the inverse of the degradation tree will be calculated and saved in the configuration as new upgrade tree. An important note is here, that both of the trees can be changed independently (before and afterwards the auto creation) and it is therefore possible to create two completely different trees. This explains also why the configuration contains two separate arrays for the tree configuration: one for the upgrades and one for the degradations.

In contrast to this, the deletion/update or creation of a degradation level will influence both trees. For example the change of the id of a degradation level will update the id in both the upgrade and the degradation tree.

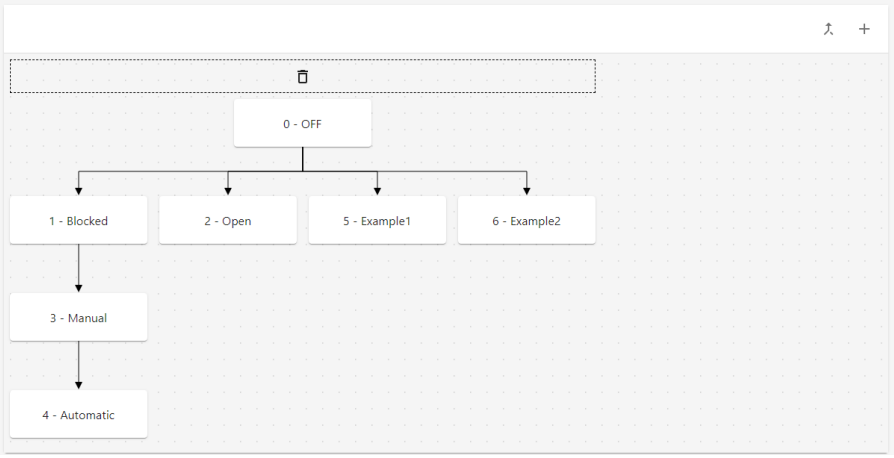

Figure 3.16: Upgrade degradation level tree

3.10 LEVELCHANGE CONFIGURATION

Another important part for the configuration is the specification of the states the level changes start and result in. To realize this, every level change contains a set of state changes which will contain the start and result state of the involved degradation levels. In order to configure these states another component was implemented that allows to define the result state for each start state of a level change. Since there are level changes for the upgrades and the downgrade, the configuration for these states have to be done for both cases and therefore these are two independent steps in the wizard. Example for the degradation state changes can be found in figure 3.17 and examples for the upgrade state changes in figure 3.18.

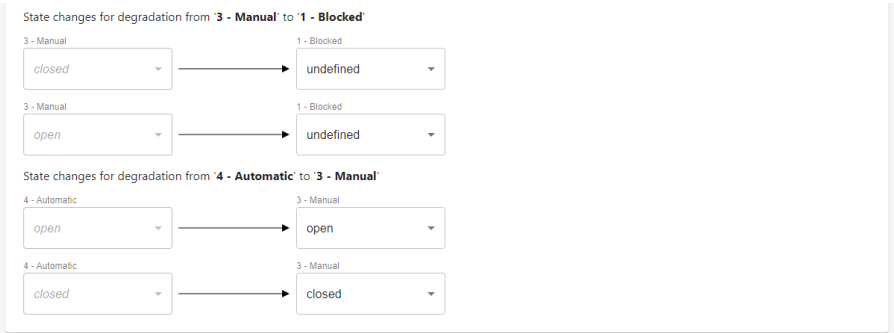

Figure 3.17: Degradation state change configuration

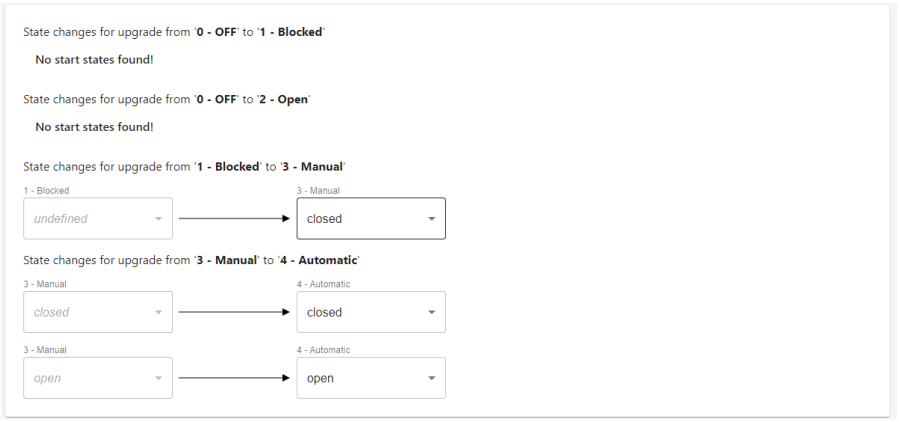

Figure 3.18: Upgrade state change configuration

3.11 FILE STRUCTURE

This section will give a short overview of the file structure of the project to facilitate further enhancements:

```
root
├── docs
├── examples
├── public
└── src
    ├── components
    ├── context
    ├── models
    └── util
```

The docs directory contains the documentation for this project, like this report. The examples folder contains example files for the configuration. In the public folder the public resources for the web application are stored. For example the favicon or the index.html.

The src folder contains the application that consists of different parts. In components folder are all components of the application which means their typescript files as well as their css files. Each component has its own folder that contains these files as well as other sub components. For example the importView contains a component for the file import. The context folder contains the definition of the configuration context that is used to store the current configuration on application level. The models folder contains a set of interfaces that describe the configuration and its components to allow validating the typescript code. There is a set of different functionalities in the util folder that help to solve some general problems, for example the validator for the schema with the corresponding schema description file.

4 CONCLUSION

In summary, the result of this project provides various functions. Using the wizard, you can navigate back and forth in the configuration and make adjustments accordingly. The configuration as such consists of several individual steps, such as the import of an existing configuration or the configuration of subcomponents. The import of an existing configuration is validated with a schema file to enforce a valid configuration in the application. Based on the subcomponents, degradation levels can be defined and inserted into a tree for the degradations or upgrades using drag and drop. For each defined degradation or upgrade, one can specify the states of the involved degradation levels in an additional step.

The code allows easy customization with new additional steps in the wizard, which also facilitates integration of React components. For the future, further adjustments are possible, for example, an extension for the import validation with additional properties, for example, to prevent cycles in the degradation level tree.

BIBLIOGRAPHY

[1] ajv contributors. *Choosing schema language*. (last visited: 2021-07-09). 2021. URL: https://ajv.js.org/guide/schema-language.html#comparison.

[2] Facebook. *Adding TypeScript*. (last visited: 2021-07-09). 2021. URL: https://create-react-app.dev/docs/adding-typescript/.

[3] Stack Overflow. *Stack Overflow Developer Survey 2020*. (last visited: 2021-07-09). 2021. URL: https://insights.stackoverflow.com/survey/2020.