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 19, 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 19, 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. Such combinations allow the design of new composite systems that combine the functionalities of the subcomponents to solve new challenges. A combined system can encounter internal problems, for example, when a subcomponent 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 for this project, which is about creating a graphical interface in the 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 a 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 that was used to implement the web application in chapter 2. It is followed by the architecture 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 was the development of a web application and due to the large number of different frameworks and libraries, a selection at the beginning of the project was required. The development of a web application can be done with and without a framework or library, but nowadays there are 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 use it for the first time and learn the required fundamentals. React is a JavaScript library to create single-page applications, but also provides typings for TypeScript [2]. 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. In order to facilitate future development, the choice fell on TypeScript.

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 are used:

- **material-ui**[1] (version 4.11.4): a library that offers standard material design components
- **react-syntax-highlighter**[2] (version 15.4.3): a syntax highlighter that is used to display the configuration as formatted JSON in the import and export
- **ajv**[3] (version 8.5.0): a validator that is used in the import to validate that the JSON matches the configuration schema
- **react-dnd**[4] (version 14.0.2): a drag and drop library for react

1 https://www.npmjs.com/package/@material-ui/core
2 https://www.npmjs.com/package/react-syntax-highlighter
3 https://www.npmjs.com/package/ajv
4 https://www.npmjs.com/package/react-dnd

3 ARCHITECTURE

This chapter will give an overview of the general concepts that were followed in the implementation. Supplementary to this, the core components of the application are introduced with some examples on how to use them.

3.1 GENERAL CONCEPTS

The complete code was structured with two general goals in mind. The first goal is that the application should be modular. This means it consists of components that can be reused on multiple occasions. For example a dialog for the deletion of an object can always have the same design and only the internal handling and the displayed text have to be adapted based on the use case. This 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. React facilitates this, because it is possible to create extensible components that can be adapted in the future.

3.2 APP-COMPONENT

The App is the central component that manages the application state and contains all the other components in the DOM after it is rendered. The App keeps track of the current configuration and provides access to it via a context provider[1]. This allows all components to access and update the configuration, which will also trigger the React lifecycle to re-render the current visible components.

3.3 WELCOME PAGE

The welcome page is the first component that is actually displayed when a user opens the application. Figure 3.1 shows the welcome page that consists in general of two buttons. The "NEW CONFIGURATION"-Button starts the wizard (compare section 3.4) to create a new configuration. The "LOAD CONFIGURATION"-Button starts the wizard, but will add an additional step that allows a user to import an existing configuration. This allows a user to update an existing configuration, for example with new subcomponents.

3.4 WIZARD

It is possible to divide the creation of a configuration into several steps. Displaying the steps separately to the user facilitates the configuration, because she is able

1 https://reactjs.org/docs/context.html

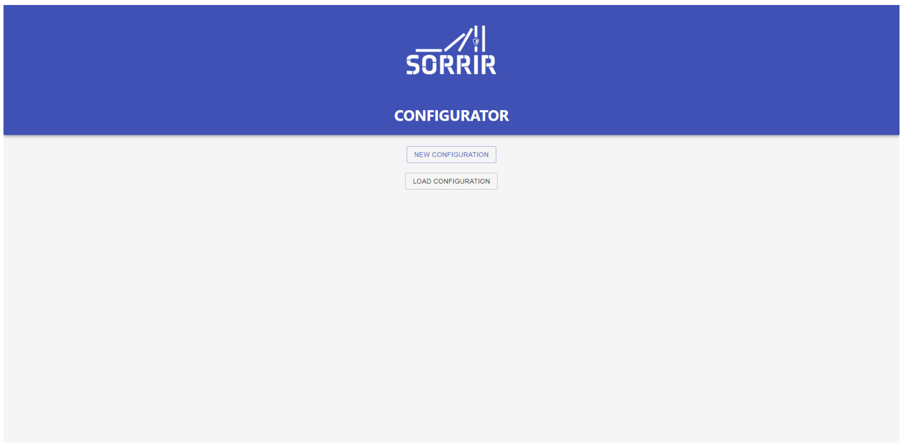


Figure 3.1: Welcome page

to focus on one aspect at a time. To provide this functionality, a wizard was implemented that allows the step-by-step configuration. The wizard supports not only a linear configuration, which means each step after another step, but also to navigate arbitrary back and forth between the different steps of the configuration.

The wizard component allows you to dynamically configure the steps and their order. The wizard consists of a menu bar, a stepper and the content. The content can be any React component or HTML Tag and is internally called a view.

Figure 3.2 shows the wizard for the configuration with an import. The menu bar is on the top of the website and updates its label with the label that was specified for the current step, for example in the figure the label was set to "Import". The menu bar contains a menu on the right that allows you to display menu items that execute an action. The menu items represent global actions that can be executed at any point in the configuration. At the moment the menu contains only 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.

The stepper is located underneath the menu bar and shows the individual steps of the configuration. The stepper highlights the current step and allows a user to navigate to any other step by clicking on it. The arbitrary navigation in the wizard also creates a constraint for the development of the different components that are displayed. It is necessary that each component that will update the global configuration has to leave the configuration in a valid state. For example a user creates a degradation level that depends on previous defined subcomponents. If she afterwards deletes one of these subcomponents the dependencies of the degradation level aren't valid anymore and therefore it is necessary to update the configuration or more precisely the degradation level.

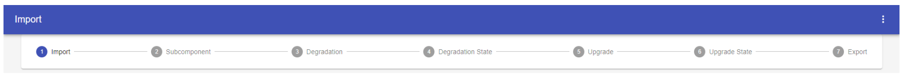


Figure 3.2: Wizard

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

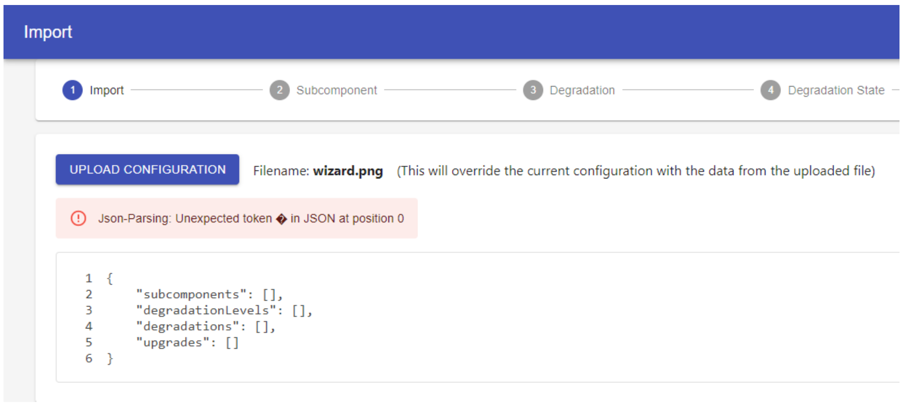
- Extension of the view enum (a enum that contains all available views in the application) ("src\util\Views.ts")
- Setting a label for the new view ("src\util\ViewLabelResolver.ts")
- Adding the view that will be displayed for the step in the wizard. This has to be a React component ("src\components\wizard\ViewSelector\ViewSelector.tsx")

After these steps, the wizard can be configured to display the new view as an additional step. The view can then display any React component or valid HTML and has access to the configuration via the context from React. The only constraint for a fully functional view is to guarantee that any change of the configuration will leave the global configuration in a valid state.

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 importing an existing configuration file and will replace the current configuration. The import will therefore override all previous changes in the configuration. The current configuration is displayed with a syntax highlighter that will be updated with the new configuration after a successful import. In the case of an unsuccessful import the configuration is not updated and a list of errors is shown.

There are several cases when the import has to fail to enforce a valid state of the configuration and therefore it will be checked if the 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.



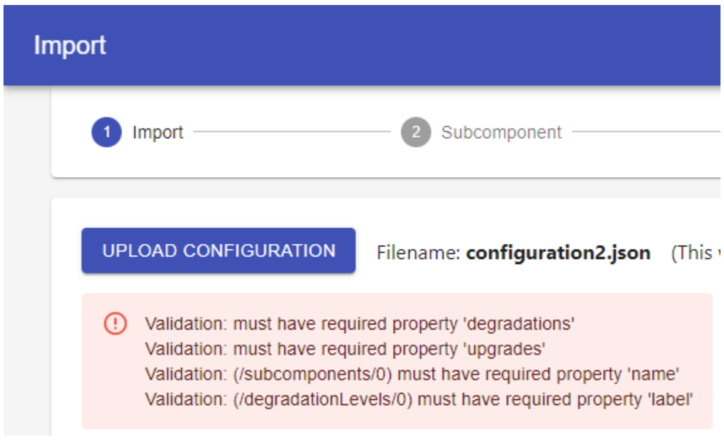
Filename: **wizard.png** (This will override the current configuration with the data from the uploaded file)

Json-Parsing: Unexpected token � in JSON at position 0

```
{
    "subcomponents": [],
    "degradationLevels": [],
    "degradations": [],
    "upgrades": []
}
```

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 and update it accordingly.



Validation: must have required property 'degradations'
Validation: must have required property 'upgrades'
Validation: (/subcomponents/0) must have required property 'name'
Validation: (/degradationLevels/0) must have required property 'label'

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. A validation at another point in the configuration is not necessary because all current wizard steps leave the global configuration in a valid state.

For the validation the *ajv JSON schema validator*[2] 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 object has to have to be a valid configuration.

There are two popular schema languages that allow the definition of JSON data. On the one hand there is the *JSON schema*[3] and on the other hand there is the *JSON Type Definition*[4]. 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, the JSON Schema is better supported in the industry [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 in the future to the JSON Type Definition.

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

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

3.6.1 JSON SCHEMA

This section provides a short overview of the configuration schema, the complete definition can be found in the provided source code. For a better understanding of the JSON configuration schema a mapping to an er-diagram with highlighted parts is shown in figure 3.5.

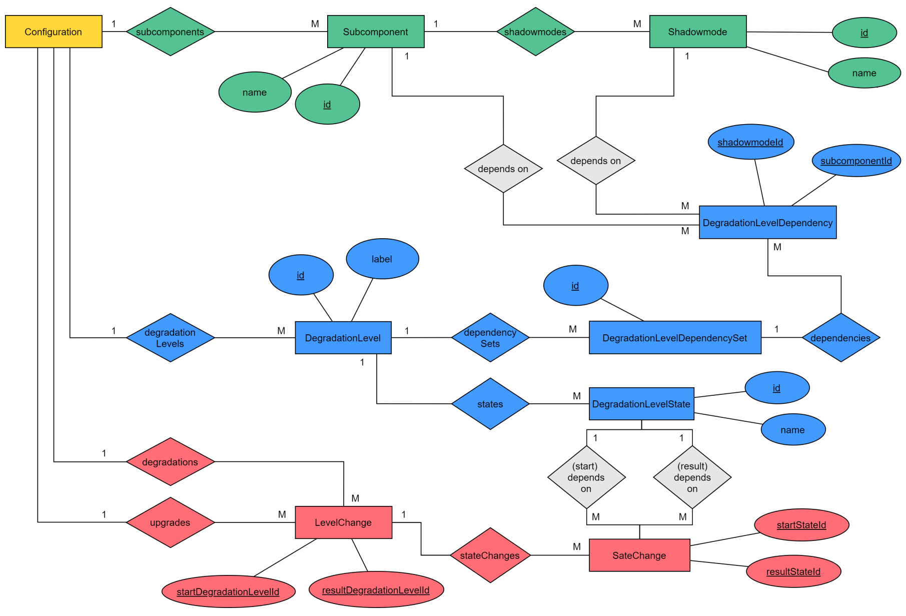


Figure 3.5: JSON configuration schema mapped on an er-diagram

The first important part of a configuration are the subcomponents. A system consists of several subcomponents that are relevant for the degradation/upgrade. The subcomponents are stored in an array in the configuration and highlighted in green in figure 3.5. 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.

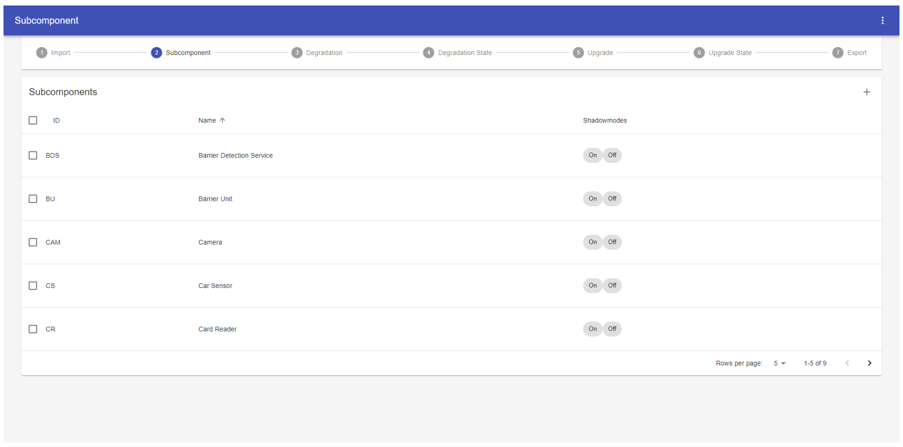
The next part of the configuration are the degradation levels which represent the levels in which the system can be. In figure 3.5 all relevant parts that are related to the degradation levels are highlighted with blue. A degradation level is available (and can be active) when one of the attached sets of dependencies is true. A dependency set is true when all the relevant subcomponents are in the specified shadowmodes or in other words when the subcomponents run in a defined state that matches the dependencies of the degradation level. For example a degradation level can depend on a crucial part of the system and that this part is in the On-state. If it is not in the On-state a degradation into another level has to be done. In order to be able to not only define a single set of dependencies, it is possible to add several sets of dependencies. Each set represents a state of the system, or more precisely the state of some subcomponents, in which they have to be in. If one of the sets of a degradation level fits the current system state, this degradation level is the active degradation level. Only the subcomponents and the states that are relevant for a degradation level are stored in the dependency array of a set. The *don't care*(DC) 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 final and central part of the configuration is the upgrade and degradation specification of the system, in figure 3.5 highlighted in red. Both the upgrade and the downgrade from one 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 (on system level, not on subcomponent level). 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. The export of the configuration will not contain any state changes that end in a DC state to save storage. The state changes that start in the default Off-state of a system are saved with an id of null as *startStateId*.

The grey highlighted parts in figure 3.5 represent relations between the different aspects of the configuration. They are implemented using the ids of the relevant objects and therefore create internal dependencies. This is important for the application, because due to these dependencies, it is required to update the global configuration when a related object is updated or deleted. For example if you delete a subcomponent or change it's id, the changes have to be propagated and for example the dependencies in the degradation levels have to be updated or deleted.

3.7 SUBCOMPONENT CONFIGURATION

The subcomponent configuration is the first step when a user starts the wizard without importing an existing configuration. Otherwise it is the second step that allows the adjustment of the imported subcomponents. A table that displays all defined subcomponents and allows to sort and navigate through them was developed for this purpose. In addition, the table allows the creation, update or deletion of them. In figure 3.6 an example is shown with several defined subcomponents.

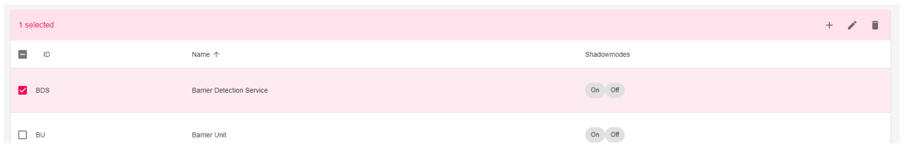


☐ ID	Name ↑	Shadowmodes
☐ BDS	Barrier Detection Service	On Off
☐ BU	Barrier Unit	On Off
☐ CAM	Camera	On Off
☐ CS	Car Sensor	On Off
☐ CR	Card Reader	On Off

Rows per page: 5 1-5 of 9

Figure 3.6: Subcomponent view

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 delete or edit the selected one or to create a new subcomponent (compare figure 3.7 on the top right corner of the table).

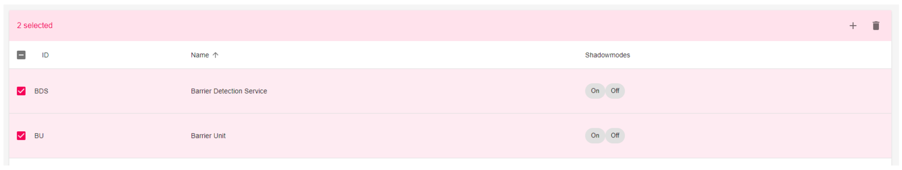


1 selected

	ID	Name ↑	Shadowmodes
☑	BDS	Barrier Detection Service	On Off
☐	BU	Barrier Unit	On Off

Figure 3.7: Subcomponent single selection

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).

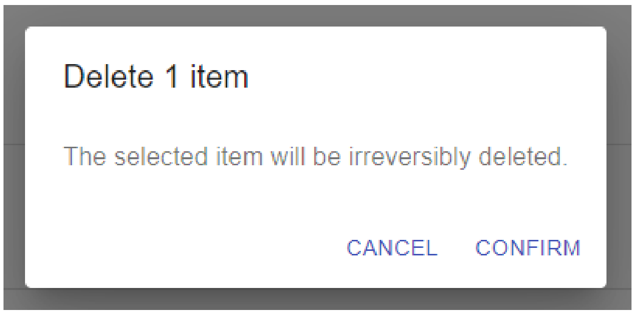


2 selected

	ID	Name ↑	Shadowmodes
☑	BDS	Barrier Detection Service	On Off
☑	BU	Barrier Unit	On Off

Figure 3.8: Subcomponent multi selection

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 subcomponents that will be deleted. 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 set that contains a degradation level dependency. This dependency will then be deleted too.

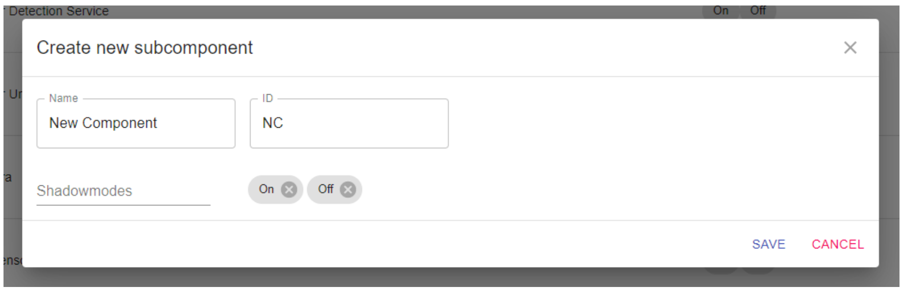


Delete 1 item

The selected item will be irreversibly deleted.

Figure 3.9: Subcomponent delete dialog

The create and edit dialog will allow a user to create or edit a subcomponent. If no name and id is set for a subcomponent the dialog will autofill the id with the value of the name (when typing it in). 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 shadowmodes and therefore a custom input was designed. This input allows a user to enter a value in the text field and after the "enter" button is pressed the value will be added as a chip on the right side. For example the *On* or *Off* shadowmodes in figure 3.10. For the creation of a new subcomponent a default shadwomode *Off* is set. Deleting a shadowmode will also update the degradation level dependencies that rely on this shadowmode.


Create new subcomponent

Name: New Component

ID: NC

Shadowmodes: On, Off

Figure 3.10: Subcomponent create dialog

3.8 DEGRADATION LEVEL MANAGEMENT

The creation, update and deletion of degradation levels is combined in one step with the functionality to define upgrades and degradations, as described in section 3.9. The degradation level DAG allows a user to select one or more 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.

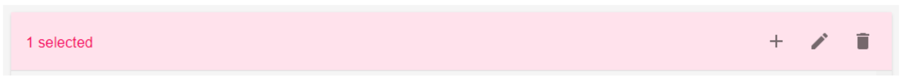


Figure 3.11: Degradation level single selection

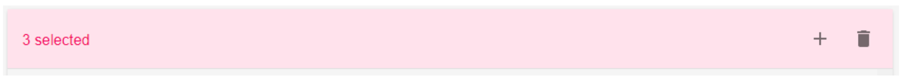
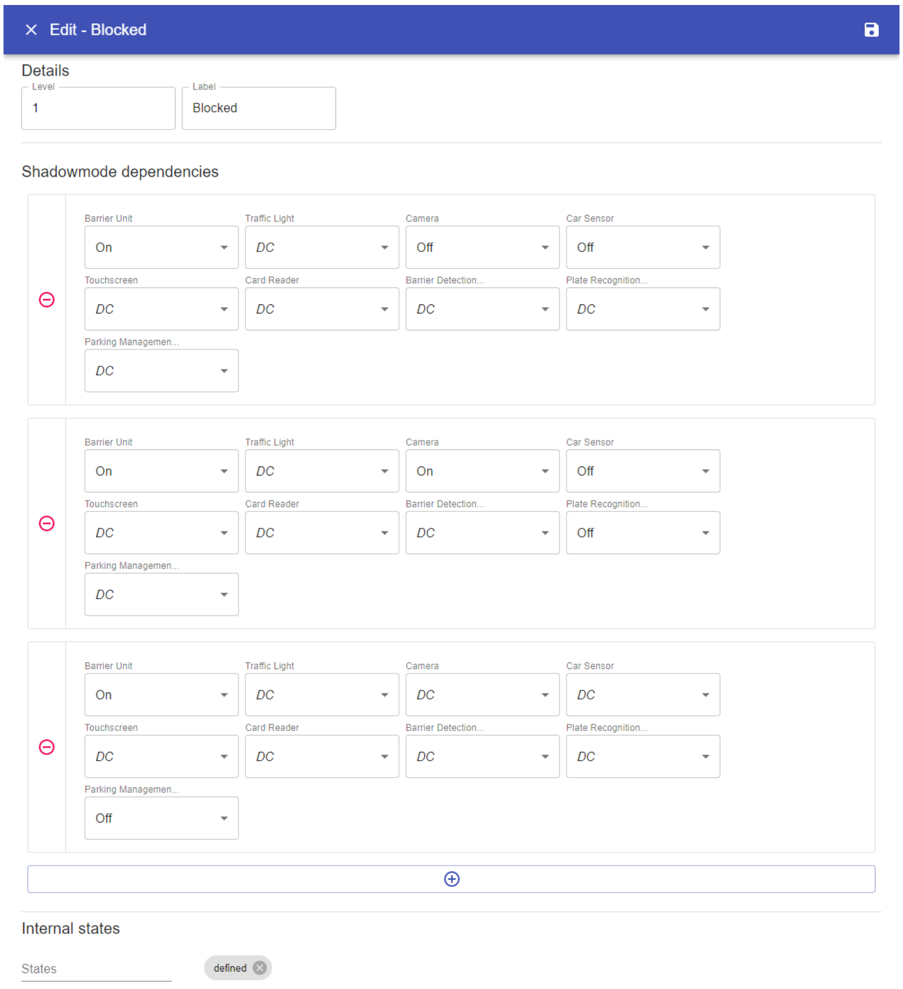


Figure 3.12: Degradation level multi selection

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 (labeled as level for the user) to enforce unique ids for each degradation level in the configuration. In addition to the validation of the id, the create dialog will set the smallest unused id possible as default value. A degradation can have multiple sets of dependencies that can be updated and deleted in the creation and edit dialog. Depending on the existing sets the sets will be generated dynamically. For each set there a box is rendered that contains a dropdown for each subcomponent. In the dropdown the user can select the desired shadowmode or DC if it is not relevant for this dependency set. 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). In order to be able to manage the dependency sets, they have an internal id that will also be in the JSON export. However, The user cannot interfere with the id directly. The internal states of a degradation level can be updated using a custom input that allows adding a state by entering a string and pressing enter (compare: custom input in the subcomponent dialog). After the enter button is pressed the state will be added as a chip on the right and can be deleted with the X-button. The deletion of states also triggers an update for the global configuration and will remove all references that require the state (at the moment: the state changes).



Edit - Blocked

Details

Level: 1
Label: Blocked

Shadowmode dependencies

Barrier Unit	Traffic Light	Camera	Car Sensor
On	*DC*	Off	Off
Touchscreen	**Card Reader**	**Barrier Detection...**	**Plate Recognition...**
DC	*DC*	*DC*	*DC*
Parking Managemen...			
DC			

Barrier Unit	Traffic Light	Camera	Car Sensor
On	*DC*	On	Off
Touchscreen	**Card Reader**	**Barrier Detection...**	**Plate Recognition...**
DC	*DC*	*DC*	Off
Parking Managemen...			
DC			

Barrier Unit	Traffic Light	Camera	Car Sensor
On	*DC*	*DC*	*DC*
Touchscreen	**Card Reader**	**Barrier Detection...**	**Plate Recognition...**
DC	*DC*	*DC*	*DC*
Parking Managemen...			
Off			

Internal states

States: defined

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 allows a user to drag and drop the degradation levels into a DAG.

An example for a possible degradation is shown in figure 3.14 that also shows several other features. All unsorted degradation levels will be displayed in a sidebar and can be tracked to any position in the DAG. 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 DAG 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 DAG 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 an unsorted degradation level again. The OFF degradation level will always be the top node in the DAG and can't be removed or changed. It will have the value '0' as id and won't explicitly be saved in the configuration.

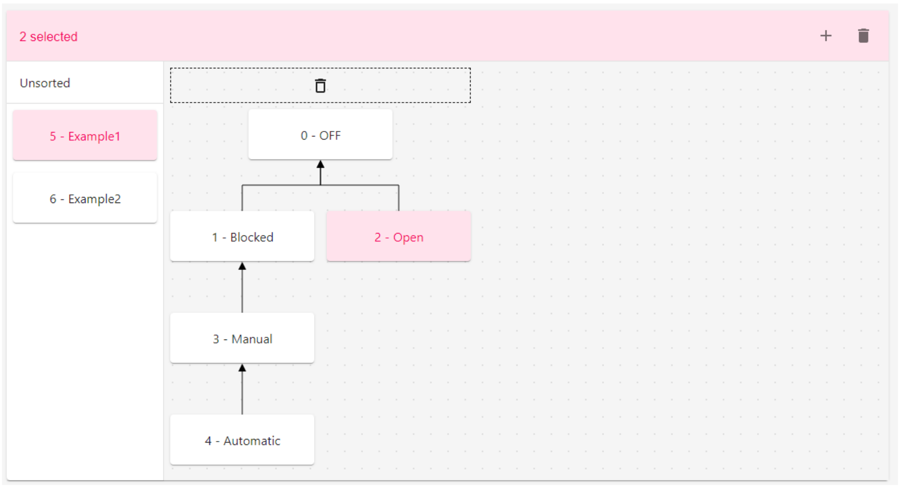


Figure 3.14: Degradation level DAG

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

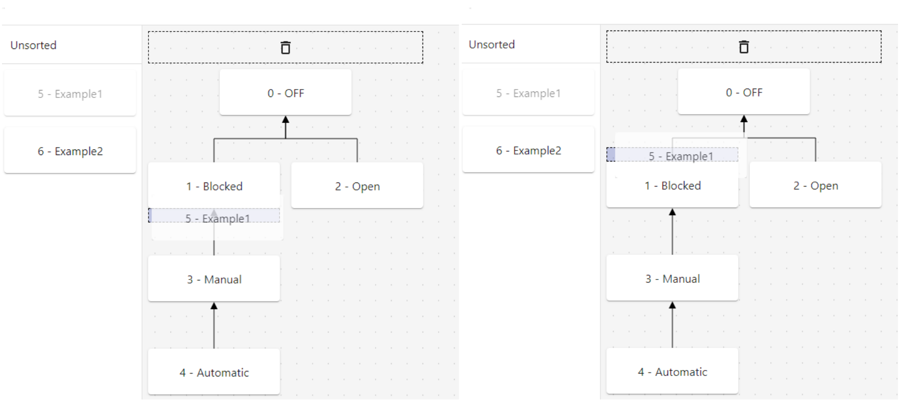


Figure 3.15: Degradation level DAG drop behavior (left: below, right: above)

Changing the position of a degradation level will reset the configuration or more precisely the level change itself. A level change can either be an upgrade or a degradation in the global configuration, with the start and result level inverted. The level changes store the internal start and result state of the degradation level. These states have therefore to be updated after a degradation level was moved. Inserting the node above another one will therefore update the level changes (degradation or upgrade) for the already connected degradation levels, but reset the state changes.

The examples for the DAG up to this point only show the usage for degradations. For the upgrades the DAG shows an inverse version, which means that the arrows start from a degradation level and point in the direction away from the OFF degradation level (to the child nodes). The specification of the upgrades is done in a separate step in the wizard which 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 DAG. The button, in figure 3.16 in the sidebar on right, opens a confirmation dialog that will ask a user to confirm the changes. After the confirmation the previous upgrade DAG will be deleted and the inverse of the degradation DAG will be calculated and saved in the configuration as the new upgrade DAG. An important note is here, that both of the DAGs can be changed independently (before and after the automatic creation) and it is therefore possible to create two completely different DAGs. This explains also why the wizard contains two separate steps and the configuration contains two separate arrays for the DAG 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 DAGs. For example the change of the id of a degradation level will update the id in both the upgrade and the degradation DAG.

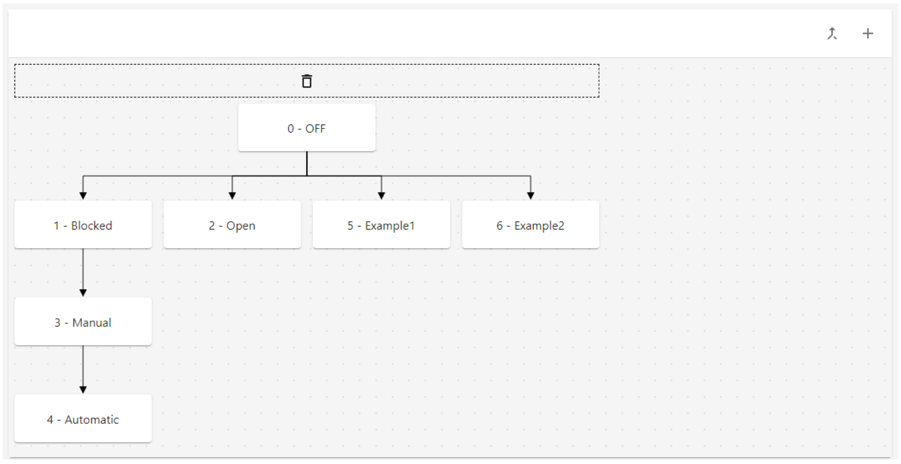
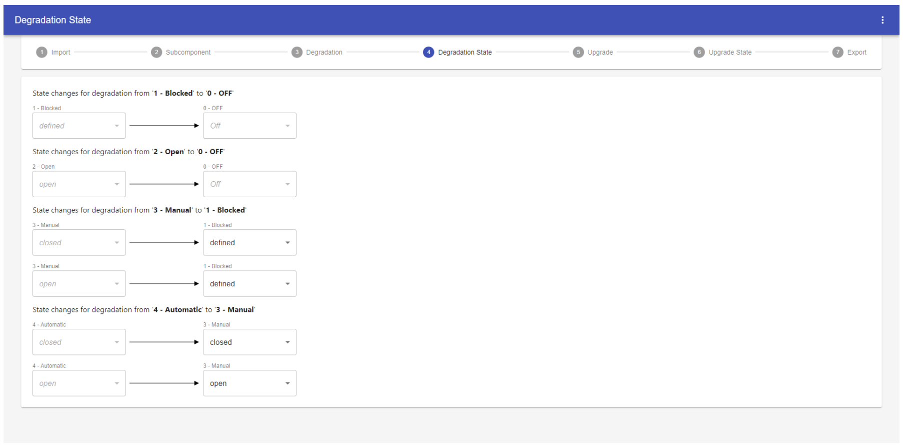


Figure 3.16: Upgrade degradation level DAG

3.10 LEVELCHANGE CONFIGURATION

The internal behavior of a degradation level is described using a state-machine with different states. In the creation/update of a degradation level it is possible to define the states of these internal state-machines. This brings us to the last relevant part for the configuration: the configuration of the state-machine states the level changes start and result in. There are two independent steps in the wizard to define the state changes for both the degradations and upgrades. For each of these level changes a set of state changes can be defined that will contain the state of the start level and the state in which the result level will result in. In order to configure these states an additional component was implemented that allows to define the result state for each start state of a level change. Examples for the degradation state changes can be found in figure 3.17 and examples for the upgrade state changes in figure 3.18.



State changes for degradation from '**1 - Blocked**' to '**0 - OFF**'

1 - Blocked: defined → 0 - OFF: Off

State changes for degradation from '**2 - Open**' to '**0 - OFF**'

2 - Open: open → 0 - OFF: Off

State changes for degradation from '**3 - Manual**' to '**1 - Blocked**'

3 - Manual: closed → 1 - Blocked: defined

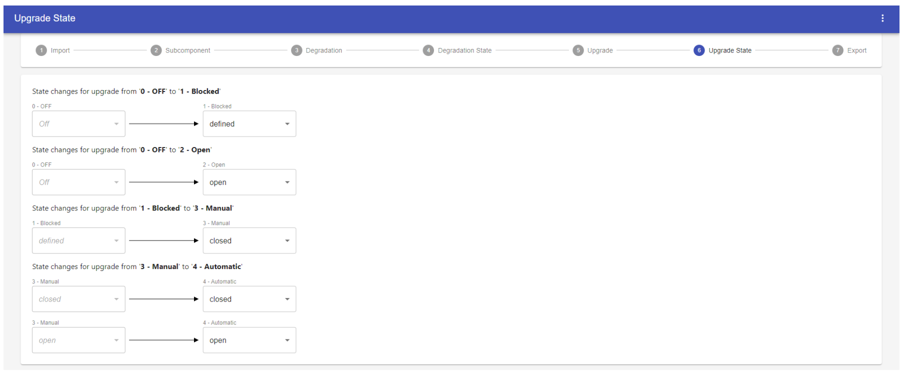
3 - Manual: open → 1 - Blocked: defined

State changes for degradation from '**4 - Automatic**' to '**3 - Manual**'

4 - Automatic: closed → 3 - Manual: closed

4 - Automatic: open → 3 - Manual: open

Figure 3.17: Degradation state change configuration



State changes for upgrade from '**0 - OFF**' to '**1 - Blocked**'

0 - OFF: Off → 1 - Blocked: defined

State changes for upgrade from '**0 - OFF**' to '**2 - Open**'

0 - OFF: Off → 2 - Open: open

State changes for upgrade from '**1 - Blocked**' to '**3 - Manual**'

1 - Blocked: defined → 3 - Manual: closed

State changes for upgrade from '**3 - Manual**' to '**4 - Automatic**'

3 - Manual: closed → 4 - Automatic: closed

3 - Manual: open → 4 - Automatic: open

Figure 3.18: Upgrade state change configuration

The state change component will display one row for each of the states (of the start degradation level). A row contains a disabled dropdown with the start state, an arrow to point in the direction of the level change and a second dropdown to select the state in which the result degradation level will be in.

In the case that the state changes are configured for degradations, it is possible that the result degradation level is the Off degradation level. In this case the result dropdown(compare figure 3.17) is disabled and the state change will not be saved. In the configuration of the state changes for upgrades it is possible that the start degradation level is the Off degradation level. If this is the case, the result state can be selected. The state of the Off level will then be stored in the exported configuration with the id *null*.

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 itself. In the 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 subcomponents. For example the importView contains a component for the file import. The context folder contains the React 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 at compile-time. 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.

3.12 BUILDING THE PROJECT

The project is a typical npm project and can be used accordingly:

1. Checkout the source code
2. Navigate to the root directory
3. Install all the dependencies: npm install (this is required when you want to start the application for the first time or when you changed the dependencies in the package.json file)
4. Run the application on your local machine: npm start

This will start the local development server which can be reached under: '*http://localhost:3000/*'

4 CONCLUSION

In summary, the result of this project is a configurator that 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 DAG 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.

The code allows an easy customization with new additional steps, which also facilitates integration of new 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 DAG. Another adjustment could be the exchange of the syntax highlighter, because it renders many HTML tags and it might slow down for large configuration files.

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.