Web UI Development

Release 15.3.0.6

CONTACT Software

Nov 01, 2018

Contents

CHAPTER 1

Web UI technical background

The technical foundation for the frontend (browser) of the CONTACT Elements Web UI are the JavaScript frameworks React and Redux. React is used to implement the visual parts of the UI, while Redux provides a framework to store the frontend application state, and to orchestrate the flow of data and events through the application.

The backend (server) logic is implemented in Python using the Morepath framework.

1.1 Rendering an application

When a CONTACT Elements URL is accessed through a web browser, a Morepath application in the backend is triggered. This Morepath application is responsible for the generation of the HTML page that is returned to the browser.

The HTML page in turn contains `<script>` tags for the JavaScript libraries to be included in the page. For the description below, it is assumed that the library `cs-web-components-base` is included.

1.2 Disable cache in development scenarios

Due to performance goals every js and css file gets cached by the browser or the web server's cache module. To avoid caching problems when delivering new software releases, each js and css file contains a checksum as part of its filename. These hash-extended filenames are only determined on server startup. However, in development scenarios the hashed filenames are changing every time the code changes. To avoid the need of restarting the server processes on every code change you need to set the variable `CADDOK_ELINK_DEBUG` located in `site.conf` to `TRUE`. This forces the web server to look up the hashed filename on every request. Setting this variable to TRUE is required in development scenarios, but should never be activated in production mode, as it will lead to decreased performance!

1.3 Frontend Registry

Due to the highly configurable and extendable nature of CONTACT Elements, the Web UI has no complete information about the available components, and the data elements these components may need. Therefore, the Web UI provides a registry, where the libraries loaded into the browser register their content. A frontend application

is configured through the backend, and uses the registry to look up names from the configuration at runtime. See *Naming rules* (page 5) for guidelines for the names.

In each application, a Singleton registry object is created. This registry basically is a simple key / value store, divided into two areas:

1.3.1 Component registry

The component registry allows libraries to assign a name to React components. Using these names, the backend can describe a frontend application as a tree of components, and properties for these components.

Components are registered as follows, where the value of `componentNameSpace` is inserted by the webpack configuration:

```
import { Registry } from 'cs-web-components-base';
import MyComponent from './MyComponent.jsx';
Registry.registerComponent(`${componentNameSpace}-MyComponent`, MyComponent);
```

1.3.2 Reducer registry

Redux holds the complete application state in a single `store` instance. This store is subdivided into a set of named parts. For each part, there exists a so-called "reducer function" (or simply "reducer"), that takes the current state and an action object that describes a state change, and returns a new state.

The Web UI provides several reducers, e.g. to manage the objects loaded through the REST API. In addition, each library may have its own reducers for specific components (e.g. a tree component needs to store the expanded state of its nodes). Either a single reducer can be registered:

```
import { Registry } from 'cs-web-components-base';
import { myReducer } from './reducers/reducers.js';
Registry.registerReducer(`${componentNameSpace}-myReducer`, myReducer);
```

or an object with several reducers can be registered in one call:

```
import { Registry } from 'cs-web-components-base';
import { myReducer1, myReducer2 } from './reducers/reducers.js';
Registry.registerReducers({myReducer1, myReducer2});
```

At application startup, all registered reducers are collected from the registry, and are combined to form the application store. The names under which the reducers are registered form the keys of the Redux store.

1.4 Immutable objects

All objects in the store are Immutable objects. This allows the efficient identification of changes, only shallow compares for object identity are needed. The components work with the Immutable structures, a conversion (via .toJS) is only done where absolutely necessary. Besides preventing easy change detection, such a conversion leads to the construction of many JavaScript objects, which can degrade performance.

1.5 Styling

In order to define consistent styles and make the writing of css rules easier, the Sass language can be used. It allows the developers for example to use globally defined variables and mixins to build a style sheet, or to write nested style rules avoiding annoying repeating of prefixes or namespaces.

Each web application has two entry points:

- `src/styles.scss` for style definitions and
- `src/variables.scss` for variables that will be used globally in all .scss files of the application

When styles are modified, the command `webmake styles` must be used to compile the styles defined by all web applications into a global css file, which will be created in your instance folder. This .css file will be shipped by all web applications that extend BaseApp.

The compiler will include a special scss variable `componentNameSpace` in your application, which should be used to namespace all styles defined for your application. This variable is defined in the `namespace.json` entry for your web application (c.f. *The apps.json File* (page 42)). It has the same value as the variable `componentNameSpace`, defined in the `webpack.config.js` file of your web application. For applications created using `webmake create` (c.f. *Creating an application template* (page 39)), this is equal to the python module name of your application with dots replaced by dashes, i.e. if your application is defined in `my.application`, the corresponding componentNameSpace is `my-application`.

Usually each component will define its own styles in an own file. To do that, firstly create an .scss file to write the style rules, for example:

```
/* my-component.scss */
/* nested under a root class for a component */
.#{$componentNameSpace}-my-component {
    .my-container {
        margin: 5px;
    }

    .my-label {
        padding-left: 10px;
    }
}
```

Next your applications scss entry point `styles.scss` needs to include the newly created file. If `src/styles.scss` does not exist, simply create the file. Append the following line to it:

```
@import "./my-component";
```

Now you need to recompile the global style file, by running `webmake styles`. The `global-styles.css` file in your instance folder should now contain the following lines of code:

```
.my-application-my-component .my-container {
        margin: 5px;
}

.my-application-my-component .my-label {
        padding-left: 10px;
}
```

1.6 3rd party JavaScript libraries

- Bootstrap
- Immutable
- React
- React-Bootstrap
- React Router
- Redux
- React Redux
- Redux Thunk

- react-router-redux

1.7 Development toolchain

To build a JavaScript library from source, we use a set of tools that are based on Node and npm:

- npm together with `package.json` files for the management of dependencies to 3rd party libraries.
- Babel to translate ES 2015 and *JSX* to JavaScript that is understood by the browser
- webpack uses configuration files (named `webpack.config.js` by default) to describe the content of a library, and the steps necessary to build it. The `cs.web` application package provides a `webpack.common.config.js` in its root directory, which is extended by the web applications to implement their own `webpack.config.js`.

CHAPTER 2

Guidelines for developing Web UI applications and components

2.1 Babel Presets

- We use all plugins from Env preset and React preset to compile our javascript files.
- We also use the transform-object-rest-spread plugin to allow rest properties for object destructing or to spread properties for object literals. This proposal is currently in stage 4 of the standardization prozess, and will be contained in one of the next standard versions. Example usage:

```
const { settings, label, ...others } = this.props;
...
// settings should be excluded from props
<SomeComponent label={label} ...others />
```

2.2 JavaScript style guide

The build process of the libraries delivered by CONTACT Software includes a call to the ESLint style checker. The concrete styling rules are contained in the `.eslintrc.json` file in the package `cs.web` package.

2.3 React style guide

- We use ES2015 classes to implement React components, not the `React.createClass` variant.
- If possible, we use plain JavaScript functions, see Stateless Functions
- Every React component has a `propTypes` definition. This serves as documentation to the user, and enables runtime checks by the React library.

2.4 Naming rules

Naming rules have two main purposes: avoid name clashes without the need for coordination, and making the source of some programming artifact obvious. The component architecture already provide a namespace system,

in the form of module names. As all relevant programming artifacts are located in those modules, we use the module names as basis for our naming rules (see *Glossary* (page 221) for definitions of the following terms).

2.4.1 Libraries

Library names are used in the backend, to specify the libraries that are needed by an application. In addition, the library name becomes part of the URL of its contained resources. A library has the same name as a Python module that is located in the same directory, but with dashes (-) instead of dots. Most open source JavaScript libraries use dashes (e.g. `react-bootstrap`), and we adhere to this convention.

Example: `cs-web-components-base-ui-lib`

2.4.2 Resources

Resource URLs are already unique, because they contain the library name. However, the JavaScript debuggers in web browsers often list only the last components of the URL. Therefore, for JavaScript resources, we include the library name in the file name.

2.4.3 CSS classes

For CSS class names, the main consideration is that unrelated parts of an application don't influence each other. This can happen if two components define the same CSS class, but with different semantics.

- CSS classes that are defined by a component always include the component name. Only if a CSS class is shared by several components, the library name can be used instead.

```
// componentNameSpace == 'cs-web-components-base'
.#{$componentNameSpace}-ui-lib-frame { ... }
```

- We follow the Block Element Modifier methodology to define CSS class names for elements contained in components, and for flag classes that denote a different appearance / behaviour of components.
- Redefinition of global styles is forbidden. If a global style can't be used as is, the redefinition must be restricted to descendants of a component specific class.

```
// componentNameSpace == 'cs-web-components-base'
.#{$componentNameSpace}-ui-lib-frame {
      .textfield { ... }
}
```

- The same rule holds, if styles of a component must be changed in some context.

```
// componentNameSpace == 'cs-documents'
.#{$componentNameSpace}-ui-app {
    .cs-web-components-base-ui-lib-frame { ... }
}
```

- Each component must have at least one CSS class set on the root element, in order to allow customization.

Attention: Due to the way Internet Explorer interprets flex-base it is sometimes necessary to specify a flex value such as 0px to avoid components taking their maximum possible size inside a flex layout.

Since, in production mode, the SCSS compiler optimizes sizes with value 0 by removing the unit, this may lead to problems in IE. This can be avoided by instead specifying a flex-base value of 1px.

2.4.4 Redux reducers

The names of the reducer functions are used as keys into the global Redux store (the application state). Reducer names start with the name of the library they are a part of, with an additional describing suffix; the separator character is a -.

For reducers provided by the Web UI, a simple name (without dashes) can be used. Because every other reducer name must have at least one -, there is no danger of collision. On the other hand, it is expected that these "standard" reducers are used by most applications, and shorter names are more convenient.

2.4.5 Redux actions

The Redux action objects that are used to send data to the store, are modeled after the Flux Standard Actions conventions.

The `type` members of the action objects must be unique over the actions from all libraries that make up an application. Therefore, they also start with the library name.

2.5 Promise

The fetch APIs and the async action creators(thunks) return javascript promise objects. The caller should take care of error handling. Recommended is to add a catch() call at the end of chaining, even with an empty function if the error can be ignored.

2.6 Error Boundaries

React 16 introduced Error Boundaries, that can be used to contain errors in the render phase, so that exceptions don't leave the whole page corrupted.

These Error Boundaries should be placed around uses of external code, where the author of a component has no control over the inner workings of child components.

CHAPTER 3

Branding and Styling

All styles are compiled into one global CSS stylesheet, by default, using SCSS style definitions. Most style definitions in this sheet can be overridden by application packages, which are listed in the package's `apps.json` file.

3.1 Overriding styles

To override styles, copy the file `variables.scss` from `cs.web\templates\styles` into your web application's root directory, `src`. Uncomment any variables you wish to override and set the values as desired.

To apply the customization, run `webmake -I instance styles`. In a buildout environment, run `instance\bin\webmake styles`.

3.2 Customizing Fonts

To inline custom fonts or vector icons in your web application you can specify them as usual in a `@font-face` attribute in one of your `.scss` files.

```
@font-face {
  font-family: 'cs.fontexample';
  src: url('fonts/fontexample.eot');
  src: url('fonts/fontexample.eot?#iefix') format('embedded-opentype'),
  url('fonts/fontexample.woff') format('woff'),
  url('fonts/fontexample.ttf') format('truetype'),
  url('fonts/fontexample.svg') format('svg');
}
```

Make sure you import the `.scss` file into your web application entry e.g. `/src/index.js`.

Note: Your webpack.config has to define `url-loader` for all of the formats you want to inline.

Webpack will generate a `<componentNameSpace>.css` and `<componentNameSpace>.css.map` file. You need to hook these files into the cs.web library registry. Afterwards, in your

APPLICATIONS_LOADED_HOOK signal handler, add the new library, where you add the generated files mentioned above.

```
w3_lib = static.Library("cs-web-fonts-example", "0.0.1", build_path)
w3_lib.add_file("<componentNameSpace>.css")
w3_lib.add_file("<componentNameSpace>.css.map")
static.Registry().add(w3_lib)
```

For your configurable UI you can include the new library into your `.json` files and use your new fonts. Alternatively load them globally by adding the library into your connect of the GLOBAL_CUSTOMIZATION_HOOK.

```
@sig.connect(GLOBAL_CUSTOMIZATION_HOOK)
def _get_page(request):
    request.app.include("cs-web-fonts-example", "0.0.1")
```

3.3 Image URLs

If you want to include image files in your components and reference them using URLs, then you cannot use URLs relative to your SCSS files. These URLs must be resolvable using paths supplied by the server. The common way is to declare an image file as an icon in the server configuration. Afterwards, you can use the URL `/resources/icons/byname/IDENTIFIER`, with `IDENTIFIER` being the icon identifier you just configured in the icon definition.

CHAPTER 4

Configuring Web Applications

Different applications using the Web UI will use many of the same components. Therefore, configuration in this context refers to a means of describing the components and their relationship, that make up an application. Developers should be able to accomplish that without writing additional JavaScript or *JSX* code.

Additionally, when no suitable component for a use case exists, it must be possible to implement that component, and make it available to the configuration mechanism. There should be a way to replace parts of a configuration with custom code, and the other way around, without the need to completely rewrite the application.

4.1 Basic configuration framework

The Morepath application implemented in *cs.web.components.base.main* (page 54) can be used to create web applications that are implemented completely in JavaScript / *JSX*. To that end, the name of the top level component is given to the frontend initialization code, and that component is rendered. A consequence of this approach is that each change to the application must be done in code.

For applications where the need to customize the application's structure can be anticipated, *cs.web.components.configurable_ui* (page 10) implements an alternative solution. The module *cs.web.components.configurable_ui* (page 10) provides the basic functionality to describe the structure of a frontend application through configuration. Part of this configuration is transmitted to the frontend, and is interpreted there to derive the actual component structure.

class cs.web.components.configurable_ui.**ConfigurableUIModel**

This abstract morepath model class implements methods to read an application configuration from JSON files, and provide morepath applications access to the resulting configuration data.

The configuration is exported as a dictionary, whose content will be provided to the frontend JavaScript code. The dictionary will contain at least the follwing keys:

- `pageRenderer`: The name of the top level React component that will be rendered by the frontend. Subclasses must overwrite `page_renderer` to contain a React component name. This component will be used in the frontend as the top level React element on the page.
- `frameComponent`: The configuration for the common frame, that will be the same for all or nearly all pages. The actual content will be displayed inside the frame. The value for `frameComponent` is determined by the method *page_frame* (page 11).

app_conf

Returns the application configuration, that is to be used by the frontend to create the UI.

libraries
Returns a list of JavaScript libraries that are required by the configured application. The libraries are included as <script> tags in the generated HTML.

setup_functions
Returns a list of functions to be executed during the setup phase of the application. Typically, these functions augment the appSetup object that is passed to the frontend.

plugin_contexts
Returns a list with plugin context names that are used on the page. The plugin context names are used to determine the needed plugin configuration, that must be passed to the frontend.

set_page_frame(*frame_id*)
Sets the ID of the page frame. This ID is returned if calling *page_frame* (page 11). Note that you have to call this function before *load_application_configuration* (page 11) is called to have any effect for the application.

page_frame()
ID of the page frame to use for the application. May be overwritten by subclasses to display a different frame (eg. without a sidebar). You can set the ID by calling `set_page_name`.

add_library(*libname*, *libversion*)
Adds the JavaScript library `libname` with version `libversion` to the list of libraries that are needed by the frontend.

classmethod load_config(*config_clname*)
Returns a list of dictionaries. Each dictionary contains a configuration of the type `config_clname` that has to be derived from `csweb_config_page`. The configuration is cached so you might not get the actual values.

add_configuration_to_context(*cfg*)
Store required libraries, setup functions and plugin contexts that are encountered when reading a configuration file. All of these are collected, and processed when the page is rendered.

insert_component_configuration(*attribute*, *config*)
If the dict `config` contains an entry `configuration`, the corresponding value is used as a file name, and its content (interpreted as JSON) is set as `attribute` in the application configuration.

Otherwise, `config` must have a key `component` that has the name of a React component as its value. This name is set unchanged as `attribute` in the application configuration.

The frontend in turn interprets strings as component names, and everything else as a component configuration object.

load_application_configuration()
Load and insert any application specific configuration into this instance. To be overwritten by subclasses, if there is anything they need to load. What exactly needs to be loaded depends on the page renderer component that the subclass uses.

Note: Don't forget to call super!

This method is guaranteed to be called only once, so subclasses don't need to implement any checks for duplicate calls.

class cs.web.components.configurable_ui.**ConfigurableUIApp**
Morepath application class for configurable UIs. This class only exists as a point to attach a morepath @view directive, and as a baseclass for applications.

Subclasses must be created to create specific mount points.

4.1.1 Providing setup data from the backend

For configured applications, there exist two ways to insert additional data into the application setup data object, which is described in *Providing setup data from the backend* (page 47):

1. By connecting to a `cdb.sig` signal, this method is suitable if th data is needed for each instance of the configured application.

```
@sig.connect(<model class>, ConfigurableUIModel, "application_setup")
def update_app_setup(model, request, app_setup):
    # ...
```

   where *model class* is a subclass of *cs.web.components.configurable_ui.ConfigurableUIModel* (page 10). The parameters of the connected function are:

   - `model`: The morepath model that represents the UI application in the backend, this will be an instance of *model class*.
   - `request`: The morepath request that triggered the application.
   - `app_setup`: A dictionary, whose content will be used for the static JSON object mentioned above. The dictionary must be modified in-place, the return value of the function will be discarded. All setup functions are passed the same dictionary instance, so that the called functions may merge their individual setup deep into the possibly nested structure.

2. By adding setup functions to component configuration files (see *Configuration files for components* (page 15)). The setup functions have the same parameters and semantics as described above. Using this method allows to specify additional data only for certain configurations.

Note: Object pages and class pages provide additional signals, see below.

In addition to adding information to the `app_setup` dictionary, the setup functions can specify additional JavaScript libraries to be loaded, by calling *cs.web.components.configurable_ui.ConfigurableUIModel.add_library* (page 11).

4.2 Application Page

The most simple configured page is a so-called application page. There is no backend code required to render the page, only for specifying the URL and the configuration to use. See *Application page configuration* (page 14) for a description of the configuration.

class `cs.web.components.configurable_ui.`**SinglePageModel**
: Model for a page whose content is defined through a page configuration. Subclasses only need to overwrite `page_name`. The model uses the page name to select an entry with matching `name` from the *Standalone Page Configuration*. That entry in turn describes the UI to render.

4.3 Generic Page

The module `cs.web.components.generic_ui` provides generic "object detail" and "class" applications for all REST visible classes. The actual HTML representations are not hardcoded, but are read from configuration files.

class `cs.web.components.generic_ui.`**GenericUIApp**
: Morepath app to render a generic (configured) UI for an "object detail" or "class" page. The application is mounted under `/info`, and provides paths of the form `/info/{rest_name}[/{object_key}]`.

`cs.web.components.generic_ui.`**select_view**(*configurations*, *classdef*, *viewname=None*)
: Determine the view configuration to use from the list supplied in `configurations`.

If `viewname` is not None, use a configuration with that name if one exists and is visible to one of the user's roles. The class hierarchy is searched starting from the given `classdef` up to the root class, if no configuration was found '*' is tried as fallback.

If `viewname` is None, the algorithm works as follows:

- Determine, from the complete class hierarchy starting at `classdef`, all viewnames that are visible to one of the user's roles.
- If more than one view is found, select the view(s) with the highest `priority` value.
- If more than one view is found, select one that is defined nearest to the given `classdef` in the class hierarchy (with fallback as above).
- Finally, select one at random from the remaining candidates

4.4 Object Page

class cs.web.components.generic_ui.detail_view.**DetailViewModel**(*rest_name*, *keys*, *viewname=None*)

Morepath model class for an "object detail" page, ie. a page that shows a single object. The page content is read from the configuration, based on the objects class name, with fallback to the REST visible name. The REST visible name is always used for the URL.

classmethod provides_detail_view(*classdef*)

Returns `True` if there is a detail page for classdef. At this time this means that the class is accessible with the REST API and that there has to be a detail page configuration. If there is no page explicitely configured for the class but a default detail view the operation `CDB_ShowObject` has to be active to make this function return `True`.

In addition to the methods described in *Providing setup data from the backend* (page 12), object pages provide a `cdb.sig` hook to add setup data on a class basis. This is useful for cases where some data is only needed for some of the classes that use a common configuration. Usage is as follows:

```
from cs.web.components.generic_ui.detail_view import DETAIL_VIEW_SETUP
@sig.connect(<cdb.object.Object subclass>, DETAIL_VIEW_SETUP)
def _app_setup(obj, request, app_setup):
    # ...
```

where `obj` is an instance of the class used in the connect call. Otherwise, the behaviour corresponds to the one outlined in *Providing setup data from the backend* (page 12).

4.5 Class Page

class cs.web.components.generic_ui.class_view.**ClassViewModel**(*rest_or_ui_name*, *viewname*)

Morepath model class for a "class" page. A class page provides an entry point for searching and creating instances of the class. As a default, only the root of a class hierarchy is directly accessible through a separate URL, but it is possible to define UI names for subclasses, and configure pages for them.

Like object pages, class pages can add setup data with an additional hook. The difference here is that a class name is supplied instead of an object.

```
from cs.web.components.generic_ui.class_view import CLASS_VIEW_SETUP
@sig.connect(<cdb.object.Object subclass>, CLASS_VIEW_SETUP)
def _app_setup(clsname, request, app_setup):
    # ...
```

4.6 Configuration

The configuration area can be accessed using the navigation tree by selecting *Administration/Configuration → Configuration → User interface → Web UI*

Component specifications

This section uses the term "component specification" in several places. A component specification must have one of the following forms:

- An entry *File*, that specifies the name of a component configuration file (see *Configuration files for components* (page 15)). You can concatenate variable parts and constant parts using +. Constant parts have to be quoted. An entry usually looks like this:

```
$CADDOK_MODULE + "/configurations/default-detail.json"
```

  `$CADDOK_MODULE` is replaced with the installation path of the module the configuration entry belongs to. You can use `$CADDOK_MODULE:<module_id>` to address a module explicitly. The calculated path is shown in the field *File (calculated)* after you have left any field the path depends on.

- An entry *React Component*, that directly specifies the name of a React component that can be found in the *Frontend Registry* (page 1).

4.6.1 Page frame configuration

The page frame renders navigation bars, side bars and so on, i.e. components that are not part of the actual page content. Normally, all pages will use the same page frame, but there may be special cases where, e.g. no sidebar should be shown.

The frame configuration can be accessed using the navigation tree by selecting *Administration/Configuration → Configuration → User interface → Web UI → Advanced → Page Frames*.

A frame configuration has these attributes:

- *ID*: Identifier for the page frame, referenced from the page configurations.
- A component specification

4.6.2 Application page configuration

The configuration for application pages can be accessed using the navigation tree by selecting *Administration/Configuration → Configuration → User interface → Web UI → Pages → Special Pages*.

The configuration has these attributes:

- *ID*: The name given by the *Application Page* (page 12) to identify the configuration.
- *Page Frame*: A reference to a *Page frame configuration* (page 14)
- A component specification

4.6.3 Object page configuration

The configuration for an *Object Page* (page 13) can be accessed using the navigation tree by selecting *Administration/Configuration → Configuration → User interface → Web UI → Pages → Detail Pages*. The algorithm that is used to select the relevant entry to use for a concrete object is defined in the function *cs.web.components.generic_ui.select_view* (page 12).

The configuration has these attributes:

- *ID*: The id of this page
- *View Name*: A view name, that can be used to explicitly address a specific entry by name
- *Classname*: The CONTACT Elements class that the entry applies to. If you use * the view is available for all classes
- *Page Frame*: A reference to a *Page frame configuration* (page 14)
- *Priority*: A numeric value, that is used as a tie-breaker if more than one entry is applicable. The entry with the highest priority value is selected.
- *Label ID* / *Label*: The human readable name of the view.
- A component specification

The dialog register *Role assignement* is used to specify which role should use a specific view. This feature can be used to display different views for members of different roles.

4.6.4 Class page configuration

The configuration for a *Class Page* (page 13) can be accessed using the navigation tree by selecting *Administration/Configuration → Configuration → User interface → Web UI → Pages → Overview Pages*. The entry format is the same as described above in section *Object page configuration* (page 14) except for the fact that you specify a *UI Name* (page 15) instead of a *Classname*.

4.6.5 GUI Names

The GUI Name configuration allows you to assign GUI names to classes that are not a root class (and therefore are not addressable with a REST name). The GUI names are then used to look up the correct class page configuration. The configuration can be accessed using the navigation tree by selecting *Administration/Configuration → Configuration → User interface → Web UI → Advanced → GUI Names*.

4.7 Configuration files for components

Component configuration files are located in the `configurations` subdirectory. The content of a component configuration file is a JSON object, with the following keys:

- `configuration`: *Component configuration* (page 15) for the root component of the application (Required).
- `libraries`: Array of *Library configuration* (page 16) entries (Optional).
- `setupFunctions`: Array of fully qualified Python names for setup functions (optional).
- `pluginContexts`: Array of plugin IDs (see *Frontend Plugins* (page 32)) that are used on the page (optional).

4.7.1 Component configuration

- `name`: Name for the component, the implementation of the component is looked up in the *Frontend Registry* (page 1).
- `properties`: React properties for the configured component.
- `outlet`: The name of a configured outlet, see *Outlets* (page 16) for details. Currently, this is only valid in *Object Page* (page 13) configurations.
- `children`: List of configurations for child components, rendered as normal children.

- `components`: Map with named configurations, the resulting React elements become props of the configured component.
- `componentClasses`: Like components, but React classes instead of elements are passed as props.

The entries under the `children`, `components` and `componentClasses` keys are recursively treated as component configurations.

4.7.2 Library configuration

Each library configuration is a JSON array with two elements:

- `library name`: the name under which the library has been registered
- `library version`: the version of the library to use

The effect of listing libraries in the configuration file is to add these libraries as <script> tags to the generated HTML. Entries for libraries must be given, if some components used in the component configuration are not included by default.

4.8 Outlets

An outlet defines a place in the component tree, where one or more child components may be inserted according to the *Outlet configuration* (page 16). When CONTACT Elements parses a configuration file, all occurences of `outlet` are replaced with the result of interpreting the configuration. This works recursively: it is possible to have an outlet in a configuration that was inserted by an outlet in the first place.

Note: Currently, outlets can only be used in *Object Page* (page 13) configurations.

4.8.1 Outlet configuration

Outlet description

An outlet description specifies a name that can be used as an outlet.

- *Name*: The name of the outlet, this is referenced from the configuration files.
- *Description*: A description text detailing the intended usage of this outlet.

Outlet definition

An outlet definition serves as an entry point to evaluate an outlet in the context of a specific class. When an outlet definition is searched, the class hierarchy is respected.

- *Name*: The name of the outlet.
- *Class Name*: Applicable class name, $\star$ can be used as fallback.

Outlet position

An outlet definition includes one or more outlet positions. Each outlet position defines one child component. The position numbers specify the ordering of the children. An outlet position is associated with role ids, and is taken into account only if the current user has one of these roles. It is possible to define more than one outlet position with the same position number. In this case, the one with the highest priority is used.

- *Name*: The name of the outlet.

- *Class Name*: The class name.
- *Position*: Position number for ordering.
- *Priority*: Relative priority between entries with the same position.
- *Label ID*: A label for the position.
- *Icon ID*: An icon for the position.
- *Properties (JSON)*: A string in JSON format, that provides properties to the React component that results from this position.
- *Child Name*: Reference to an *Outlet child* (page 17).
- *Setup Class*: The fully qualified Python name of a class that inherits from `OutletPositionCallbackBase`. This can be used to create or adapt the configuration dynamically. See also *Outlet generators* (page 17).

Label ID and *Icon ID*, if set, overwrite the settings from the child. *Properties (JSON)* is merged with the properties defined for the child, with the values provided here taking precedence.

class `cs.web.components.outlet_config.`**OutletPositionCallbackBase**
A class that gives you the opportunity to customize the configuration of outlets. You should derive from this class to implement your own behaviour if you have configured a fqpyname in the OutletPosition.

classmethod adapt_initial_config(*pos_config*, *cldef*)
This callback allows you to manipulate the configuration of the position. You may change `pos_config` or return a list of dictionaries that should be used instead of this configuration. `cldef` is the class definition of the object that contains the data displayed by the outlet.

classmethod adapt_final_config(*component_config*, *cldef*)
This callback allows you to manipulate the configuration of the position after the configuration had been transferred to the form that will be transferred to the frontend. You have to change component_config in place. `cldef` is the class definition of the object that contains the data displayed by the outlet.

Outlet child

An outlet child provides the configuration for a React component that should be rendered as an outlet position.

- *Name*: The name of the child, referenced from outlet positions.
- *Label ID*: A label for the child.
- *Icon ID*: An icon for the child.
- *Properties (JSON)*: A string in JSON format, that provides properties to the React component that results from this child's configuration.
- *File / React Component*: A component specification.

4.8.2 Outlet generators

There are some standard generator functions that can be used with the outlet configuration.

class `cs.web.components.outlet_generators.`**OutletPositionRelationshipsCallback**
A callback that generates outlet positions for all relationships that are prepared to be used with the Web UI. The generator adapts the standard configuration so be sure to set the correct child name in your configuration.

You can exclude relationships if you add their role name to a list named `excludeRelships` that you can configure in the JSON properties. A configuration might look like this:

```
{"excludeRelships": ["Cdb_organization_to_cdb_person"]}
```

You can also explicitely set the relationships you want to offer by providing a list named `includeRelships`.

4.9 Display Contexts

To render an object using a specific dialog configured in CONTACT Elements, a display context can be used. The configuration for a display context associates a display context name and a classname with a dialog. To utilize a display context, you use a *DisplayContextLoader* (page 162) as parent to one or more React components.

Typically, the children of a `DisplayContextLoader` will display the data not in the same way as described in *Web UI Forms* (page 25), but in some form suitable to the context.

Display contexts do not rely on operations, but instead map the given object's attributes to the dialog configuration. Therefore, no user exits are executed. However, it is possible to attach a *Dialog Hook* (page 21) to the dialog configuration, and provide computed values in this way.

CHAPTER 5

Kernel operations and the Web UI

There is an API that allows an application developer to run operations in the way the Windows client does. This means you get a form generated from the *Mask configuration* where you can enter your data and run the operation by pressing the button. At this time there are many *restrictions* (page 24) using this approach compared to the behaviour of the Windows client. Because of these restrictions there is a separate *configuration* (page 19) if an operation can be used this way with the Web UI.

5.1 Web UI Operation Config

Some components, e.g. *FormWithOperations* (page 188), use an API that allows them to run kernel operations in a way similar to the Windows client. Because of some *restrictions* (page 24) you have to activate the functions for the use with the Web UI. This configuration only applies to this kind of operations - it does not restrict the usage of other (REST)-APIs that also use the kernel operations to implement their behaviour. The configuration resides in the register *Web UI* of the standard operation configuration.

The register has these members:

- *Available with Web UI*: If you want to use the operation you have to set this checkbox. Otherwise the behaviour is the same as if the configuration does not exist at all. Uncheck the box if you want to deactivate the operation temporarily. If you do not plan to reactivate the operation you should delete the configuration object instead.
- *Essential/Frequently used*: A hint for the implementation of the UI. Operations that have been marked this way might be displayed more prominently, e.g. with an extra button in a title bar.
- *Prefer classic UI with hybrid client*: A hint that the Windows client should handle the execution of the operation if possible.
- *Presentation ID*: The ID of a component that is able to render the form of the form based operation. See section *Presentation IDs* (page 20).
- *Mask name*: Allows you to set a dialog for the operation. If the field is empty the dialog configured in the basic operation configuration is used. This allows you to provide optimized forms for displaying in a web context.

5.1.1 Presentation IDs

This field defines a default way of how the operation should present configured forms to the user. For legacy reasons, it also controls how results of the operation should be handled. Though for most use cases the value `modal` should be appropriate, you may want to learn about the possible options. Sensible default values for this field are:

- `modal`: if the operation displays a configured form to the user, it will be displayed in a modal dialog, disabling interaction with the rest of the user interface.
- `navigate-new-tab`: the operation's result will be displayed in a new tab.
- `no-form`: the display of a form will be supressed, even if one is configured for the operation. The operation will behave as though the user has pushed the form's submit button. Note that this option may interfere with certain features like execution of wizard logic.
- `delete`: Special case for `CDB_Delete`: the display of a form will be supressed, instead the user will be queried if he wants to delete the objects in question. If the user submits the dialog, the operation will be executed.

Note that usually the `no-form` option will usually not be necessary, since the operation execution will skip the `Show Dialog` stage, if the operations form contains no registers.

You may enter different values into the field, if your application provides special operation handlers (see *Executing operations* (page 26)), which you want your operations to use. Note, that if a presentation id is not set in the current scope, it will behave as if the `modal` option has been set.

5.2 Dialog Hooks

If you use the forms defined as masks in CONTACT Elements you can adapt the form behaviour by adding own code that is called whenever the user changes a specific value of the form. There are two kinds of dialog hooks - hooks that are executed in the backend and hooks that are executed in the frontend.

5.2.1 Configuration

Dialog Hook Function

A dialog hook function object defines which piece of code has to be executed and defines if this code is part of the frontend or the backend.

The dialog has these members:

- *Label*: The name of the hook. The name is used to reference the function.
- *Function*: The name of the function that implements the hook. This is the full qualified python name if the hook is implemented in the backend or the name of a JavaScript function for frontend hooks.
- *Backend*: The checkbox has to be set if the hook is implemented in the backend.
- *Description*: A text that describes the function of the hook.
- *Module*: The software module this configuration belongs to.

Predefined hook function

- `EmulateLegacyDialogItemChange`: This backend hook function emulates the behaviour of the Windows client. An operation is constructed and usually an user exit of kind `dialogitem_change` is called. The action is quite expensive - usually you should prefer to migrate the code.

- `EmulateLegacyDialogButton`: This backend hook function emulates the behaviour of the Windows client. An operation is constructed and usually an user exit of kind `button_pressed` is called. The action is quite expensive - usually you should prefer to migrate the code.

Dialog Hook

If you want a dialog hook to be executed you have to connect a dialog hook function with a dialog configuration. You will usually add a hook using the register *Dialog Hooks* of the mask configuration. The configuration contains this fields:

- *Mask Name*: The name of the dialog that is adapted by the hook.
- *Attribute*: The attribute that triggers the hook. If the value of the given attribute changes the hook is called. There are some special values that can be used as attribute names:
  - `*`: This value activates a hook for all changes of the dialog. This is ok for frontend hooks but might lead to performance problems if you implement the hook in the backend.
  - `::PRE_SUBMIT::`: This value can be used if you want to check the dialog values before the dialog is closed as the result of clicking on a submit button.
  - `::PRE_DISPLAY::`: This value can be used to configure backend hooks that calculates additional values before the dialog ist shown. The hook parameter of the callback function is derived from *cs.web.components.ui_support.dialog_hooks.DialogHookPreDisplay* (page 23). At this time frontend hooks are not called for `::PRE_DISPLAY::`.
- *Label*: The name of the *hook function* (page 20) to be called.
- *Function*, `Backend`: Some information retrieved from the *hook function* (page 20).
- *Position*: If there are several hooks defined for an attribute the position defines the calling order. Hooks with a low position are called first. Note that frontend hooks are always called before backend hooks.
- *Active*: The checkbox has to be set if the hook should be executed.
- *Module*: The software module this configuration belongs to.

5.2.2 Implementation

In general you should prefer to write frontend hooks because they do not need a roundtrip to the server.

Frontend Hooks

If you implement a frontend hook you have to ensure that the library that contains the JavaScript function is part of the page where the mask is shown. The call of frontend hooks is synchron.

Backend Hooks

To implement a backend hook you have to implement a python function with one parameter. This parameter can be used to retrieve information about the actual mask and to manipulate the form. It is of type *cs.web.components.ui_support.dialog_hooks.DialogHook* (page 22). An implementation might look as follows:

```
def handle_categ_change(hook):
    """
    (De)-activates the dialog field for the subcategory depending
    on the actual category
    """
    categ = hook.get_new_values["categ"]
    if has_subcategs(categ):
        hook.set_writeable("categ2")
```

```
    else:
        hook.set_readonly("categ2")
        hook.set("categ2", "")
```

At this time backend dialog hooks are called asynchron which means that the user might continue his work before the hook result is applied to the form.

cs.web.components.ui_support.dialog_hooks

This module implements the infrastructure for calling dialog hooks implemented in the backend.

class cs.web.components.ui_support.dialog_hooks.**DialogHook**
Class that provides the context to implement dialog hooks that are part of the backend.

Construction of the hook. This is usually done by the system.

set_request(*request*)
Sets the request that is e.g. used to generate links.

get_hooks()
Returns a list of type `WebUIDialogHookFunction` that contains the hooks to be called with the actual request.

get_operation()
Retrieve the `cdbwrapc.Operation` object this hook works with. The call is quite expensive so you should try to avoid to call this function if possible.

get_operation_state_info()
Returns the `cdb.platform.mom.operations.OperationStateInfo` object for the operation the hook works with. You can use this object to get further information about the operation context.

get_operation_name()
Returns the operation name for the operation the hook works with.

get_changed_fields()
Returns a map of fields that have changed with the identifier as key and the new value as value.

get_new_values()
Returns a dictionary that contains the actual values of the dialog.

get_new_value(*name*)
Returns the typed value of the field `name` which should be the attribute identifier. Using the attribute name is also supported but less performant. Raises a `KeyError` if name is not accessible

get_new_object_value(*name*)
Like *get_new_value* (page 22) but if `name` is not part of the new values and the operation works on an object the value is retrieved from that object.

set(*name*, *value*)
Set the value of the field identified by `name` to the JSON representation of `value`.

set_readonly(*fieldname*)
Prevents the field specified in `fieldname` from being edited on the display mask. `fieldname` can be a string or a list of strings.

set_writeable(*fieldname*)
Allows the field specified in `fieldname` to be edited on the display mask. `fieldname` can be a string or a list of strings.

set_mandatory(*fieldname*)
Turns the field specified in `fieldname` into a mandatory field on the display mask. `fieldname` can be a string or a list of strings.

set_optional(*fieldname*)
Turns the field specified in `fieldname` into an optional field on the display mask. `fieldname` can be a string or a list of strings.

get_fieldtype(*fieldname*)
Returns the type of the attribute `fieldname` that is used in the UI. Usually this is the type defined in the data dictionary. The most common types are `sqlapi.SQL_CHAR`, `sqlapi.SQL_INTEGER`, `sqlapi.SQL_FLOAT` and `sqlapi.SQL_DATE`. If the type could not be determined the function returns `-1`.

apply_dlg_changes(*changes*)
Set the changes you got by calling `Operation.get_dialog_changes`.

set_error(*title*, *message*)
Adds an error message to be displayed in the frontend

set_dialog(*dialog*)
Sets a dialog to be shown in the frontend

set_next_dialog(*dialog_name*, *classname=None*)
Sets a dialog to replace the current dialog in the frontend.

If `classname` is not `None`, the dialog is replaced by a dialog from the CONTACT Elements dialog configuration in the context of the given class.

If `classname` is `None`, the operation state of the hook is used to get the specific class. If the class can not be determined, a dialog without the context of a specific class is used.

class cs.web.components.ui_support.dialog_hooks.**DialogHookPreDisplay**(*hook*, *values*, *wizard_progress*, *op-state*)
A dialog hook that is called to manipulate values before they are sent to the frontend. At this time the hook only supports setting values and wizard progess data.

Get the context object for `hook` which is a `WebUIDialogHook` object or just a string that contains the hooks name. `values` is a dictionary that contains the actual values. `type_dict` is dictionary that contains the types of the actual form.

get_new_values()
Returns a dictionary that contains the actual values of the dialog.

perform()
Executes the configured callable and returns the values set during the execution in the JSON Format of the values.

5.2.3 Examples

The following code-snippet gives an example how a backend-based dialog hook may display a simple choice to the user:

```
def example_hook(hook):
    from cs.web.components.ui_support.frontend_dialog import FrontendDialog
    fe = FrontendDialog('This is a FrontendDialog',
                        'Please select a choice to continue')
    fe.add_button('Call Server', 0, FrontendDialog.ActionCallServer, is_
↪default=True)
    fe.add_button('To Dialog', 0, FrontendDialog.ActionBackToDialog, is_
↪cancel=True)
    hook.set_dialog(fe)
```

5.3 Restrictions running operations with WEB UI

There are some differences compared to the behaviour of the Windows client if you are running an operation using the Web UI.

5.3.1 Operations running in a relationship context

Some components, e.g. the `RelationshipTable`, allow you to run an operation within a relationship context. If you create a new object in a relationship context with a separate link class the creation dialog for the link class will currently not be displayed during the action. If it is not possible to create a link object without such a dialog - e.g. if there are mandatory fields - the creation of the link will fail.

5.3.2 User exit call times

- User exits with the mode `post_mask` are not executed.
- There are no user exit calls if a dialog item changes its value (mode `dialogitem_change`). These calls have been replaced by *Dialog Hooks* (page 20).

At this time there is no plan to reactivate these User Exit modes.

5.3.3 User Exits results with UI impact

At this time context adaptor functions that should lead to additional user interaction have not the expected effect if used with the Web UI. Some examples are

- `ctx.show_message()` that should display a message box
- `ctx.next_mask()` that sets a follow up dialog
- `ctx.start_selection()` that displays a selection catalog

5.3.4 User Exits that trigger file actions

Functions that trigger an up- or download will not have the expected result. Examples are

- `redirect_file()`
- `set_export_files()`
- `set_external_filename()`
- `set_mail_body()`
- `set_mail_recipients()`
- `set_mail_subject()`

5.3.5 Formular manipulation by User Exits

Some function have no effect at this time, e.g.

- `set_focus()`
- `enable_ok()`
- `disable_cancel()`
- `set_active_register()`
- `set_button_label()`

5.3.6 Follow up operations

Operations started by calling `set_followUpOperation()` will not be executed.

5.3.7 Web UI Forms

Not all configuration details of the legacy mask configuration are applied to the Web UI Forms.

Catalogs

If the field the catalog belongs to is not readonly the catalog provides a simple type ahead mechanism. This mechanism uses the input to search for possible values that starts with the actual user input. Entries that match the initial search conditions of the catalog are listed first. If the maximum number of type ahead proposals is not reached with this search conditions all conditions that are not marked as mandatory for the query dialog will be removed to find more proposals.

At this time structure catalogs are not available. The creation or modification of objects within the catalog is not possible. Catalogs that check their values are not supported. Catalog proposals are not implemented.

Itemtypes

Fields of type `eLinkControl` are ignored. You have to place the content of the eLink control somewhere else on the detail page.

Fields of type `Button (client)`, `Generator` and `Spacer` are also ignored.

Items that deal with files (`FileOpenBrowser`, `FileSaveBrowser` and `DirectoryBrowser`) are ignored.

Field content

The configuration of wether a field can only contain upper or lower characters is ignored. The checkbox *Use monospaced font* is also ignored. The personal settings that define the alignment of the field content are ignored.

Layout

The field label is always displayed above the field. The configured color of a field is ignored. Each form column has the same size - the *Resize mode* of a field is ignored.

All registers are displayed one below the other.

5.4 Integrating Operation Handling into User Interfaces

In order to integrate operation handling into custom user interfaces, two aspects need to be adressed:

- Firstly, the available operations for an object or a class need to be retrieved from the server.
- Secondly, these operations need to be executed, possibly involving user interaction, collecting data, as well as updating the pages to display results of operations.

These different aspects of operation execution may be handled separately from each other, so that implementors may use the provided APIs in a flexible way.

5.4.1 Retrieving operation information from the server

The server provides a REST API to retrieve operation information objects, which can be used as an input to the operation handling API. Operation information may be adressed in three different ways:

- By name, e.g. `CDB_Create`, `CDB_Modify`,...
- By operation contexts, i.e. a set of operations that may defined in the backend
- All operations given a class

The lowest level of retrieving operations are the Redux Actions that query operations and operation contexts from the server. Though it is possible to fetch operations by invoking the REST API directly, using the provided Redux Actions minimizes the number of requests sent, if operations are fetched multiple times, due to them being required in multiple places in the interface.

The two methods provided by the interface are:

- *fetchOperationContext* (page 67)
- *fetchOperationInfo* (page 67)

These will insert the requested operations into the reducer `operationsById` and the requested operation context objects into `operationContextById`. Assuming the Redux State is bound to variable `state`, you may retrieve the operation info objects by accessing the Redux State:

```
function mapStateToProps(state, ownProps) {
  const myOperation = state.operationsById.getIn([
    ownProps.myClass,
    'operations',
    ownProps.myOpName,
  ]);

  return {myOperation};
}
```

As an alternative to these interfaces, Web UI also provides a higher order component – *WithOperations* (page 74) – that may be used to supply the wrapped component with a set of operations in its props. This set is always restricted to an object class, either determined by naming the type, or calculated from the set of objects provided to the HOC. For most use cases, using `WithOperations` should be sufficient.

5.4.2 Executing operations

In order to execute operations from the server, there are multiple ways to execute an operation.

FormWithOperations: The simplest way is by embedding the *FormWithOperations* (page 188), component, which does not require manually fetching the operations. This component may simply be embedded into a page, where it may be used to execute operations using the classic CONTACT Elements masks.

Custom Handling: If a developer wants more control over the way the user may start, submit and cancel operations or customize the way in which data required for the operation is collected, OperationScope-based operation execution may be used. This mechanism basically consists of three components:

- *OperationTrigger* (page 89)
- *OperationHandler* (page 86)
- *OperationScope* (page 89)

An *OperationTrigger* (page 89) initiates an operation by dispatching a *runOperation* (page 115). E.g. the application bars in the Web UI implement an `OperationTrigger` that displays the available operations for the pages entities and starts the execution of the selected operation.

An *OperationHandler* (page 86) is responsible for executing the operation: It may collect data, either by displaying a configured mask or rendering a frontend-based form, execute the operation, and finally update the page accordingly (if custom update handling is required).

In order for `OperationTrigger` and `OperationHandler` to communicate with each other, an `OperationScope` is required. This component is above both trigger and handler in the component hierarchy. A handler then registers itself in the scope, and the trigger uses the scope to determine the correct handler to be invoked for the scope.

Handlers and Operation Information objects both define a *Presentation ID*, that is used to identify the correct handler for an operation. E.g., the id `modal` is used to execute an operation using a modal dialog.

`OperationScopes` may be nested, in that case, the trigger will search upwards in the hierarchy until a matching handler is found.

Each web application is automatically rendered with two operation handlers. Firstly, one that handles those operations that are based on E-Link applications, e.g. the `share`-operation, by redirecting to the appropriate E-Link page. Secondly, there is a default operation handler that handles all form-based operations by rendering the form in a modal dialog. If you do not define an operation scope with custom operation handlers in your page, these two handlers will handle the execution of your operations nonetheless.

5.4.3 Starting operation execution

To start the execution of an operation a component needs to dispatch a `runOperation` action. It is not recommended to dispatch this action directly but either use one of the provided components such as `TableOperationToolbar` and `ContentOperationToolbar` or to implement a custom OperationTrigger.

5.4.4 Handling Operation Results

Operation Results may be handled in different ways. If the OperationHandlers `submitOperation` method is executed, the server will respond with a response object that may either contain a `ui_link` field, or an `object` field.

The `ui_link` field usually means, that the result of the operation can be displayed by navigating the browser to the URL value of the field, while the `object` field may, e.g., contain a modified or created object.

By default `OperationHandler` will handle these results as follows:

- `ui_link` will result in the provided URL to be navigated in the browser tab in which the operation was submitted
- an `object` will result in an update of the store, which in turn will update connected components in the page.

Custom behaviour may be implemented by `OperationTrigger` and `OperationHandler` components by calling `setSubmitHandler` and providing a callback that should be executed.

5.4.5 Handling of `CDB_Delete`

Due to technical reasons, the operation `CDB_Delete` should usually be invoked using `navigation-id='delete'`. For one reason the delete operation has no associated form, while default operation handlers are based on a configured form from the backend. The other reason is the handling of deleting the context object of detail pages. Since detail pages usually rely on the context object, the detail page is not valid after deletion, and the default `delete` Handler handles this case.

5.4.6 Navigation-based Operations

Operations like `CDB_ShareObject`, which should navigate directly to a new tab but don't expect any input parameters, should be configured with `navigation-id='navigate-new-tab'`. Note, that in order to avoid popup-blockers, the OperationTrigger for these Operations should only be invoked from the browser's event handling thread (e.g., by pressing a button).

5.4.7 Operations without user interaction

If an operation should always be run without any user interation, you should configure the operation with `navigation-id='no-form'`.

5.5 Examples

Since your web application contains an operation scope and two operation handlers by default, the simplest case to execute an operation in your page is to simply render an operation trigger, e.g. a button that will call startOperation in its `onClick`-handler.

5.5.1 Simply Triggering an Operation

```
import ImmutablePropTypes from 'cs-web-components-externals';
import React from 'react';
import {Alert} from 'react-bootstrap';
import {WithOperationTrigger} from 'cs-web-components-base';

const OperationButton = WithOperationTrigger(
   () => {
       const operation = props.operations.get(0);
       if (operation === undefined) {
          return (<Alert>Operation undefined.</Alert>);
       }
       return (
           <Button onClick={() => props.startOperation(
                                  operation,
                                  {object: props.contextObject})}>
               {"Edit object"}
           </Button>
       );
   }
);

OperationButton.propTypes = {
   contextObject: React.PropTypes.object,
   operations: ImmutablePropTypes.map
}
```

The button-component defined above may be instantiated with an operation, e.g., provided by *WithOperations* (page 74) and a `contextObject`, on which the operation is executed.

```
class MyComponent extends React.Component {
    constructor(props) {
       super(props);
       this.OperationButton = WithOperations(
          OperationButton,
          {operationNames: ['CDB_Modify']}
       );
    }

    render() {
        return (
            <this.OperationButton
                contextObject={this.props.contextObject} />
        );
    }
}
```

CHAPTER 6

Embedded Web Applications

Web applications may be used in the context of the desktop client. This section describes, how the interaction between the desktop client and embedded web applications works out of the box, and how it can be improved by application developers.

6.1 Handling Modifications in Embedded Web Applications

Embedding web applications as ELinkControls in dialogs allows the user to modify data. If, however, the dialog is closed or reloaded, all changes that are not persisted get lost.

The behaviour of the desktop client is to notify the user of such ongoing changes, and prompting if the dialog should really be closed. For operations run in the web application this behaviour usually works out of the box. If, on the other hand, the web application employs custom components or low-level REST calls to modify data, the custom code needs to signal ongoing modifications.

This is achieved by dispatching into the Redux store `modifications`. Let's look at a simple example of a component that changes its modification state:

```
import React from 'react';
import {Checkbox} from 'react-bootstrap';

const ModificationDemo = props => (
   <Checkbox onChange={event => props.setComponentModified(
                                  this, event.target.checked)}
             value={props.modifications.get(this)}>
      Modified
   </Checkbox>
);

ModificationDemo.propTypes = {
  setComponentModified: React.PropTypes.func,
  modifications: React.PropTypes.bool
};

function mapStateToProps(state) {
  return {
      modifications: state.modifications
  };
```

```
}

export default connect(
  mapStateToProps, {setComponentModified}
)(ModificationDemo);
```

The component `ModificationDemo` provides a simple checkbox which updates its modification state by dispatching setComponentModified (Line 5). In the store, modifications are stored per component, so the component passes itself as a key, as well as its modification state as a boolean value. Components may also read their modification state by connecting to the store.

Note that a component must actively reset its modification state if changes are discarded or persisted.

CHAPTER 7

Recently Used Objects

The system provides a REST call to store objects that have been recently used by the user. This call is, e.g. used by the standard detail view of an object. Administrators can have a look at the entries actually stored by selecting *Administration/Configuration → Configuration → User interface → Web UI → User Settings/Data → Recently Used* in the navigation menu.

There are 2 *Default Dettings* that change the behaviour of *Recently Used*. The setting `csweb.max_history_objs` defines the maximum number of *Recently Used* entries for each user. The setting `csweb.history_objs_per_class` defines the number of *Recently Used* entries per class.

The standard settings allow the user to overwrite these entries using the *Personal Settings* operation.

CHAPTER 8

Frontend Plugins

Sometimes there is the need to extend the frontend behaviour with plugins that can be added dynamically. An example might be a component that shows several objects of different types, where each type in turn should be rendered using a special React component. Frontend plugins provide a mechanism for developers to add new types, and components to display them, by implementing and configuring new plugins. See *Plugins and Plugin Container Components* (page 58) for concrete use cases.

There are two kinds of components involved: plugin components, which render specific objects, and plugin containers, that provide the environment for the plugin components. Plugin containers have the following responsibilities:

- They specify the contract that plugin components have to implement. This contract consists of the React properties that the container provides to the plugin components.
- They implement the selection strategy to find the correct plugin component for the data to be rendered, using the configuration for a specific plugin ID. The helper function *getPlugin* (page 99) is provided as a general mechanism to do this.
- At render time, they render the concrete plugin component instances according to the data and the configuration, and provide them with their data, as defined by the contract.

Plugin components are normal react components, where the only requirement is that they fulfill the contract of the plugin container.

8.1 Plugin configuration

The configuration of plugins can be accessed is located in the subsection *Plugins* of the Web UI configuration.

8.1.1 Plugin

A plugin aggregates *plugin configurations* (page 33) that fulfill the same purpose. The configuration has this attributes:

- *ID*: The plugin id. This ID is e.g. referenced by the function `cs.web.components.plugin_config.Csweb_plugin.get_plugin_config` (page 33).

- *Fully qual. Python name*: The name of a class derived from *cs.web.components.plugin_config.WebUIPluginCallbackBase* (page 33). Have a look at the documentation of this class to see, how the configuration depends on this class.
- *Description*: Informal text to describe the plugin
- *Module*: The module this plugin configuration belongs to.

8.1.2 Plugin configuration

A plugin configuration contains the data of a specific plugin. The configuration has these attributes:

- *Plugin-ID*: References the *Plugin* (page 32) the configuration belongs to.
- *Discriminator*: A text the frontend uses to decide if this plugin is suitable for the concrete situation. The format and interpretation depends on the plugin.
- *React-Component*: The component that is used to render the plugin
- *Setup*: Python Callable that has to be called to ensure that the plugin will work.
- *Priority*: If more than one plugin configuration can be used to fulfill the purpose the plugin with the highest priority is chosen.
- *Module*: The module this plugin configuration belongs to.

If the JavaScript code that loads the plugin depends on further libraries you have to add these libraries in the register *Libraries*.

8.2 cs.web.components.plugin_config

Code dealing with frontend plugin configuration can be found in the module *cs.web.components.plugin_config* (page 33). The module contains functionality that deals with the Web UI plugin configuration.

class cs.web.components.plugin_config.**Csweb_plugin**(***values*)

classmethod clear_cache()
Clear the cache used by *get_plugin_config* (page 33).

get_callable()
Returns the object specified in `self.fqpyname` or the standard callable *WebUIPluginCallbackBase* (page 33) if no fqpyname is specified. Raises an `ue.Exception` if the configuration for the callable is wrong.

get_config()
Retrieve the configuration for the plugin. The function will return a list of dictionaries ordered by the priority of the plugin configurations. Eeach dictionary contains the attributes `discriminator`, `component` and `setup` of type string and the attribute `libraries` that contains a list of tuples of a library name and its version. The values depends on the WEB UI plugin configuration.

classmethod get_plugin_config(*plugin_id*)
Retrieve the configuration for the plugin with the given `plugin_id` using the *get_config* (page 33) method of the plugin. The content of the result is described there. This method caches the result to optimize the performance. The function returns `None` if there is no Web UI configuration for the given `plugin_id`.

class cs.web.components.plugin_config.**WebUIPluginCallbackBase**
A class that gives you the opportunity to customize the configuration of plugins. You should derive from this class to implement your own plugin behaviour.

classmethod adapt_config(*plugin_configs*)
This callback allows you to manipulate the list `plugin_configs` after the configuration has been read from the database. The function returns the adapted list - it is allowed to change plugin_config in place and return this list.

classmethod check_values(*ctx*)
This function is called as a part of the `post_mask` userexit of a plugin configuration. Overwrite this function if you want to do additional checks for the values the user has entered. If the checks are not ok, raise an exception.

classmethod generate_config(*plugin_config*)
This callback allows you to overwrite the generation of a single configuration entry. `plugin_config_entry` is an object of type `csweb_plugin_config`. The default implementation returns a dictionary that contains at least the attribute `discriminator`. The attributes `component`, `libraries` and `setup` are set if the configuration contains a value.

CHAPTER 9

The Dashboard

The CONTACT Elements Web UI provides a dashboard application, that allows users to build their own customized starting page for CONTACT Elements. The dashboard application itself implements the framework for displaying and customizing dashboards. The widgets shown on the dashboard are for the most part distributed with specific application packages. The dashboard application provides the mechanisms needed to inject these widgets into the generic framework.

Dashboard: A specific configuration of dashboard layout and dashboard elements.

Widget: A blueprint for dashboard elements, implements the logic for displaying some content.

Dashboard Element: A widget instance that is rendered on a dashboard.

9.1 Design

The dashboard is a starting point for the CONTACT Elements products. It can be used to get an overview, execute minor tasks, and launch into workflows. Widgets should serve one or more of these three use cases, and do little else beyond that.

Overview: The user wants to quickly get relevant information from diverse contexts

Launch: The user wants to quickly jump to specific views, applications or objects

Execute: The user wants to carry out simple actions

A few design rules should be observed for widgets in order to maintain a homogeneous user experience and application character.

- Focus on a limited set of user stories and solve for them (see above)
- Provide concise information
- No horizontal scrolling
- Avoid menus, modals, toolbars

9.2 Configuration

The configuration for dashboard widgets consists of the following attributes:

- *ID*: The unique ID for the widget.
- *Name*: A name that is shown to the user of the dashboard.
- *Description*: A short description of the widget.
- *React Component*: The React component implementating the component. This attribute is looked up in the *Frontend Registry* (page 1).
- *Library*, *Version*: An optional JavaScript library that contains the widget implementation.

9.3 Implementation

Widget implementations are React components. They receive the data to be rendered as React properties, and use the API provided by the `cs-web-dashboard` library. For details, see the API reference and the widget template given below.

9.3.1 API reference

- *DashboardItem* (page 219)
- *actions* (page 219)

9.3.2 A template for widget implementations

```
import React from 'react';
import {connect, ImmutablePropTypes} from 'cs-web-components-externals';
import Dashboard from 'cs-web-dashboard';

/*
 * A template for a dashboard widget. Use this as a blueprint for implementing
 * new dashboard widgets.
 */
class TemplateWidget extends React.Component {
    constructor(props) {
        super(props);
        this.onConfigure = this.onConfigure.bind(this);
    }

    onConfigure() {
        // Show configuration UI, must call `setSettings` to store changed
        // configuration data.
    }

    render() {
        return (
            <Dashboard.DashboardItem
                    item={this.props.item}
                    title='Insert title here'
                    configCallback={this.onConfigure}>
                <YourContentHere />
            </Dashboard.DashboardItem>
        );
    }
}
TemplateWidget.propTypes = {
    item: ImmutablePropTypes.map.isRequired,
    columnWidth: React.PropTypes.oneOf(['small', 'medium']).isRequired,
    setSettings: React.PropTypes.func.isRequired
};
```

```
const actions = {
    setSettings: Dashboard.setSettings
};
export default connect(undefined, actions)(TemplateWidget);
```

CHAPTER 10

Building And Deployment

CONTACT Elements provides a script/module – `webmake` – to support the different recurring tasks during development and deployment of web applications. The tool provides, on the one hand, integration of the different utilities involved in the process of web application development, e.g., yarn and webpack. On the other hand it is integrated into the Python setuptools used by the CONTACT Elements component architecture for handling applications.

Using `webmake` as a link, web applications are completely integrated into the standard development and deployment procedures used to develop applications based on CONTACT Elements.

`webmake` can be used either as a utility that is called from the command line, or as a library of Python functions. How these functions are integrated into the setuptools extensions of the component architecture is described in depth in section *Integration with setuptools and snapp* (page 42). Developers usually use the routines of `webmake` by using its command line interface.

The different tasks that are supported by the script are given as a command, which is preceded by a set of general options, and followed by command-specific arguments.

General options are:

- `--verbose`: The script generates verbose output. Useful for debugging purposes.
- `--dry-run`: The script will not execute any actions that modify the filesystem or database. Note that combined with `--verbose` this will still produce output. Combined, the two options can be used to explore if an invocation of the script will yield the expected results, without actually modifying the configuration of the instance/packages used.

General options are followed by a command, which may be one of the following:

- `create`: Create a web application package based on a template. See *Creating an application template* (page 39)
- `jsdoc`, `jsdoc-clean`: Create and clean up API documentation for web components. See *Generating documentation* (page 39).
- `devupdate`: Prepare the dependency location node_modules in the instance directory
- `build`, `yarn`, `webpack`, `clean`, `cache-clean`: Prepare, build and clean up web applications.
- `licensereport` generates a license report of dependencies.
- `styles`: compile overridable styles into the global stylesheet

Note: Command examples follow POSIX utility conventions, see http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html#tag_12_01

10.1 Creating an application template

Creating an initial template can be achieved with:

```
webmake [--verbose] [--dry-run] \
        create <package>
        [--templates TEMPLATE [TEMPLATE ...]]
```

Parameters necessary for the *create* commands are:

- `<package>`: The script requires a name, with which the app will be registered. The package needs to be a valid python package identifier. It should specify an (existing or non-existing) python package that could be located inside a cdb module, but must be located inside a cdb package.

Optional parameters are:

- `--templates`: a list of additional templates to install. See also *Specifying Templates* (page 39).

At first the path of the web application will be created, based on the package identifier provided and already installed CONTACT Elements application packages.

E.g. if the package provided is `cs.web.components.foo`, the location of the package `cs.web` will be determined. Inside a python package `cs.web.components.foo` will be created, containing a directory structure as detailed in *Application Layout* (page 44).

In order to deploy this as a webapp, the backend module `<package>/main.py` needs to be added to an applications `default.conf`.

10.1.1 Specifying Templates

The application configuration that is created depends upon the templates that are specified by the call to `webmake`.

The default template to be installed is `base_impl`. Different templates can be used during creation by providing the parameter `--templates`, which accepts a list of templates to be installed. Application templates are used to create the basic application data, feature templates extend this by different features. Available templates are:

Application Templates:

- `base_impl`: Creates a skeleton for writing a custom application. See *Frontend-specific tasks* (page 48) for more information on writing an application based on custom components.
- `base_conf`: Creates a skeleton for configuring an application. See *Configuring Web Applications* (page 10) for more information on configuring applications.

Feature Templates:

- `selenium`: Extends the skeleton created by `base_impl` for writing automated UI tests.

10.2 Generating documentation

API Documentation is written by attaching JSDoc doclets to the Javascript entities to be documented. This documentation is transformed to reStructured Text – the format used to write documentation in CONTACT Elements – and can then be integrated into the docset.

10.2.1 Using webmake to generate `.rst`

In order to include your Javascript API documentation in a CONTACT Elements doclet, you need to transform doclets into reStructured Text files. `webmake` provides the command `jsdoc` to invoke `jsdoc` with the appropriate parameters.

This command will search for all components in the provided bundle sources, and – for each one having a jsdoc string attached – generate a file that may be included in the provided docset.

```
webmake jsdoc cdbpackage docset [packages [packages ...]]
```

Parameters required for the jsdoc command are:

- `cdbpackage`: Name of cdbpackage in which to look for webapps. Ignored if packages are specified
- `docset`: A relative path to a docset in which to generate documentation, e.g. `doc/mydocset/en`.

Optional parameters are:

- `packages`: A list of packages for which to create documentation. If non are provided, all packages listed in `apps.json` are used.

For example:

```
webmake jsdoc cs.web doc/web_ui_dev/en cs/web/components/base/js
```

generates the documentation for bundle *cs-web-components-base* (page 57) in `doc/web_ui_dev_en/_jsapi`. This way, e.g., documentation for ApplicationFrame can be included in docset `web_ui_dev` by including `_jsapi/ApplicationFrame`.

Cleaning up: **In order to clean up generated reStructured Text files** the command:

```
webmake jsdoc-clean <pkg-name> <docset>
```

may be used.

10.2.2 Mapping JS entities to `.rst` files

The template to generate reStructuredText from jsdoc Doclets supports the following Javascript entities:

- *Package*: For each package, a folder is generated. Optional documentation is taken from the packages `package.json`.
- *Modules*: Javascript modules correspond to .js/.jsx files. By default no file is generated. If you want to generate documentation for a module, you may add a jsdoc comment, and put an `@module` declaration inside the comment. Given a module `Module` the generated reference will be `<bundle>-<path_to_file>.__module__`
- *Default exports*: The reference for the default export of a module is `<bundle>-<path_to_file>.Module.__default__.rst`
- *Classes*: Documentation for classes is generated in their respective files. The filename for a class `Class`, defined in a module `Module` will be `<bundle>-<path_to_file>.Class`, if the class is a named export of the module. If a module documentation is generated, the class will be listed there.
- *Functions*: Functions are not defined as top-level doclets, so no file will be generated for these. Such a doclet will be generated in the surrounding top-level doclets documentation file

`<bundle>` is the name of the javascript bundle, usually with - as separators, while `<path_to_file>` is the path to the file, with dots as separators. The documentation will be generated in `<docset_path>/_js/<bundle_name>`. `<docset_path>/_js/toctree.rst` contains the index of generated files, and should be included in your documentation at an appropriate location. Besides this automatically generated index, documentation should only be referenced by reference.

Another generated file - <docset_path>/_js/<bundle_name>/mapping.json - contains a mapping from top-level file references to generated rst-files.

For details on documenting Javascript APIs, see *Documenting Components* (page 49).

10.2.3 Integration with snapp doc

Analogous to the setuptools integration of webpack (c.f. *Integration with setuptools and snapp* (page 42)) the jsdoc process is integrated into snapp doc. This command is extended to check for the presence of a jsdoc parameter to the CONTACT Elements application's setup.py. As an example for a setup-call extended for jsdoc we present the setup.py file for cs.web:

```
pkgtools.setup(
   name="cs.web",
   version="15.2.0",
   install_requires=['cs.platform',
                     'cs.base'],
   docsets=[
       "doc/web_user/de (html,chm,pdf)",
       "doc/web_ui_dev/en (html)",
   ],
   ...
   jsdoc=[
       "doc/web_ui_dev/en"
   ],
   ...
)
```

snapp doc checks, for each docset it builds, if this docset is specified in the list provided as argument to the jsdoc parameter of the setup call. If it is in the list, a _jsapi folder will be generated, containing the doclets as reStructured Text, when calling snapp doc.

10.3 Building web applications

As explained in section *Web UI technical background* (page 1), application bundles and their dependencies are transformed into bundles using webpack, while dependency management is done using yarn, the npm replacement.

How these utilities are integrated with webmake is explained in section *The build process* (page 41). Web applications are usually defined inside Python packages in a cdb application package (c.f. *Application Layout* (page 44)). These can be either specified manually, or read from apps.json (c.f. *The apps.json File* (page 42)).

10.3.1 The build process

While web applications are implemented in ES6, these sources must be bundled and transpiled to be served to the browser. This is achieved by the utilities yarn (for dependency management) and webpack (for transpilation and bundling). Additionally, a global stylesheet is generated from all SCSS style definitions.

These commands are integrated into webmake as follows:

- the command webmake devupdate copies the template file package-base.json from cs.web as package.json into the instance directory and runs yarn install inside it.
- the command webmake build <pkg-name> builds package <pkg-name>, e.g., cs.web, i.e., it runs yarn install to install missing dependencies, and webpack to create an application bundle.
- the command webmake yarn <pkg-name> <command> runs an arbitrary yarn command.
- the command webmake webpack <pkg-name> runs webpack on the web applications in package <pkg-name>. Usually, during development you simply invoke webpack, and only call build if dependencies have changed.

- the command `webmake styles` compiles the global stylesheet `global-styles.css` into the instance directory.

Additional commands exist to clean files generated during the build process:

- the command `webmake cache-clean` cleans up the cache of `only-if-changed` (see below).
- the command `webmake clean` cleans up the cache of `only-if-changed`, the `node_modules` folders created by `yarn`, as well as the `bundle` files created by `webpack`.

The `webpack` command allows the path of the app as an optional parameter. If provided, the command will be carried out for this app in its respective directory. If it is omitted, the command will be run for all web application bundles specified in `apps.json` (c.f. *The apps.json File* (page 42)).

Standard web application bundles shipped by CONTACT Software GmbH use the npm `only-if-changed` package, which suppresses `webpack`, if the timestamp of all source files is not newer than a timestamp cached by the package during the last call of `webpack`. Note that this does only consider changes to the file's timestamp. If, e.g., the build configuration is changed, the build cache needs to be invalidated manually by invoking `webmake cache-clean`.

10.3.2 The `apps.json` File

The commands explained above usually either work on a single web application bundle or on all bundles defined inside an application. In order to find these bundles a file `apps.json` is created on the top level of the application package. This file contains a list of paths to the javascript bundle roots, relative to application package root.

E.g., suppose you have an application in the folder `my.application` and the Javascript roots (the folder containing your `webpack.config.js` and `package.json`) of your web application bundles in the folders `my.application/my/application/user_app/js` and `my.application/my/application/news_app/js`, then you should have an `my.application/apps.json` file, containing

```
[
    "my/application/user_app/js",
    "my/application/news_app/js"
]
```

Then you may build the web bundles in your application by issuing

```
$> webmake build my.application
...
$> webmake styles
```

To build only one specific bundle, e.g. `user_app` issue

```
$> webmake webpack my.application my/application/user_app/js
```

Note that this file is necessary for integration with setuptools and snapp.

10.3.3 Integration with setuptools and snapp

setuptools: The `webmake` commands `build` and `clean` are integrated into setuptools. The CONTACT Elements setuptools extension in `cdb.comparch.pkgtools.setup` searches for an `apps.json` in the current working directory (which should be the package directory during buildout process). When an `apps.json` file exists, setuptools will assume that web-specific tasks need to be carried out during development and deployment. It will

- extend the setuptools commands `bdist_egg` and `develop` to invoke the `build` command of `webmake` (c.f. `cdb.comparch.pkgtools.apply_npm_wrappers`).
- include web-specific files which would not be included in the default data_files (c.f. `cdb.comparch.pkgtools.add_web_files`).

`bdist_egg` will result in a production build (using `webpack -p`), while `develop` will result in a development build. Since this change in configuration is not detected by `only-if-changed`, the cache needs to be cleared for this to work. While `bdist_egg` cleans the cache before invoking `webpack`, this needs to be done manually by invoking `webmake cache-clean <pkg-name>`.

CHAPTER 11

Implementing Web Applications

This chapter explains the individual steps to develop an application. This process is usually started by creating a skeleton application using the application generation option of `webmake`. Use of the script and its different options are described in *Building And Deployment* (page 38).

Section *Application Layout* (page 44) explains the default layout used for CONTACT Elements web applications, which is created in the respective application folder after issuing `webmake create`.

The process of application development itself can be divided into creation of the backend (see *Backend-specific tasks* (page 45)) and frontend (see *Frontend-specific tasks* (page 48)). The backend provides configuration and data for the frontend, while the frontend determines how an application is rendered in the browser.

Internationalization and localization of resources is explained in *Internationalization* (page 50).

The last section – *Customizing BaseApp* (page 51) – explains how the application properties provided by `BaseApp` can be customized by applications, to extend or modify the application frame.

11.1 Application Layout

The typical layout of a CONTACT Elements web application, as is also created by the `cs.web.make` script is as follows:

```
<package>
|
+-- js                          Frontend files
|   |
|   +-- build                   Deployable files generated by webpack
|   |   |
|   |   +-- <app_name>.js       Bundled code of the application
|   |   |
|   |   +-- <app_name>.js.map   Codemap to unbundled code
|   |
|   +-- src                     Javascript/JSX source files
|   |   |
|   |   +-- actions             .js files for Redux actions go into this folder
|   |   |   |
|   |   |   +-- ...
|   |   |
|   |   +-- components          Folder for components defined in this application
```

```
|   |   |   |
|   |   |   +-- ...
|   |   |
|   |   +-- containers
|   |   |   |
|   |   |   +-- ...
|   |   |
|   |   +-- reducers            .js files for Redux reducers go into this folder
|   |   |   |
|   |   |   +-- ...
|   |   |
|   |   +-- helpers.js          File with helper functions
|   |   |
|   |   +-- index.js            .jsx file containing the main component of the app
|   |   |
|   |   +-- i18n.js             Helper functions for i18n
|   |
|   +-- [package.json]          Optional packages, required by the application
|   |
|   +-- webpack.config.js       Webpack configuration template
|
+-- tests                       Python based tests (e.g., nose/selenium)
|   |                           created via selenium template
|   +-- test_*.py
|
+-- __init__.py
|
+-- main.py                     Contains a Morepath app and views for the app
```

The application package must reside in a subfolder of an CONTACT Elements application module.

The basis for the application is a python package, which must, at least, contain setup code for the application, and optionally setup code for, e.g., morepath and i18n.

Javascript files go into subfolder `js`, which contains the `src` folder as well as configuration files and internationalization data. The utility `webpack` is used to bundle these resources into one or more `.js`-files. These files are then registered as a library, allowing the components defined in the files to be referenced by other applications (see *Registering Libraries* (page 45)).

The file `webpack.config.js` is used to configure `webpack`. It extends the configuration in `webpack.common.config.js` from `cs.web`. When a web application is created with `webmake` the utility takes care of setting up the webpack configuration file.

The testing folder is created by template `selenium`.

11.2 Backend-specific tasks

The basic tasks for implementing the backend of an CONTACT Elements web application are the registration of static components (JavaScript libraries, etc. – see *Registering Libraries* (page 45)) and, if the application should provide a navigatable frontend, the implementation of the application backend code (see *Mounting a Morepath-Application* (page 46)).

Another task that involves backend code is internationalization, but, since this task requires also frontend action, internationalization is explained in its own section, *Internationalization* (page 50).

11.2.1 Registering Libraries

A set of JavaScript bundles that consists of common functionality, e.g. a web application, is registered in the system as a library.

This setup code is usually created by `webmake create` (see *Building And Deployment* (page 38)), but may be modified in advanced scenarios.

```
from cdb import rte
from cdb import sig
from cs.platform.web import static

@sig.connect(rte.APPLICATIONS_LOADED_HOOK)
def _register_libraries():
    lib = static.Library("my-app",
                         "0.0.1",
                         os.path.join(os.path.dirname(__file__), "js", "build"))
    lib.add_file('my-app.js')
    lib.add_file('my-app.js.map')
    static.Registry().add(lib)
```

A library is created with the application name as library identifier, the current version of the library and the base path to the bundles of the library. Bundle files are then added by specifying the path relative to the base directory, specified during library creation. To make the library available, it needs to be added to the library registry.

11.2.2 Mounting a *Morepath*-Application

While a library definition as defined in *Registering Libraries* (page 45) is sufficient for an application to provide components for other applications to use, an application may also provide a frontend, which can be reached by a given path. This requires a frontend path to be registered in CONTACT Elements.

The backend implementation of CONTACT Elements frontend applications is usually based on the *Morepath* Web Framework (see https://morepath.readthedocs.io/en/latest/), and a basic understanding of how *Morepath* applications work is required to develop the backend of an CONTACT Elements web application.

```
from cs.web.components.base.main import BaseApp, BaseModel


class App(BaseApp):
  pass


@Root.mount(app=App, path='/foo')
def _mount_app():
  return App()
```

CONTACT Elements frontend applications should be derived from the `BaseApp` class, using the `BaseModel` class, resulting in an application definition as above.

At a minimum, the following views need to be implemented:

- `base_path`: The base path of the app. This is usually equal to the path on which the *Morepath* application has been mounted.
- `app_component`: Returns the name of the base component that is displayed when the application is loaded.

So – in addition to the above – a basic frontend application also has the following definitions:

```
App.view(model=BaseModel, name="app_component", internal=True)
def _setup(self, request):
  request.app.include("my-app", "0.0.1")
  return "my-app-MainComponent"


App.view(model=BaseModel, name="base_path", internal=True)
def get_base_path(self, request):
  return request.path
```

Note that the call to `request.app.include` is necessary to include the library registered in *Registering Libraries* (page 45). In this way JavaScript libraries provided and required by this application are served to the user.

The `app_component` view specifies the main component of the application. For further details see *Reusable Components* (page 48):

11.2.3 Providing setup data from the backend

In most cases, the frontend application needs configuration or setup data from the backend, e.g. values extracted from the CONTACT Elements data dictionary. This data will be inserted into the HTML page generated through the backend in the form of a static JSON encoded object. *cs.web.components.base.main.BaseApp* (page 54) fills the keys `appSettings`, `formats` and `links` in this object, and *cs.web.components.configurable_ui.ConfigurableUIApp* (page 11) adds the key `applicationConfiguration` and `pluginConfiguration`, these keys should not be replaced by other applications.

An application can extend the setup data by providing specific data under its own namespace. The setup data is a python dict object, with an additional method `merge_in`:

```
from cs.web.components.base.main import BaseApp


class MyApp(BaseApp):
    def update_app_setup(self, app_setup, model, request):
        super(App, self).update_app_setup(app_setup, model, request)
        app_setup.merge_in(["links", "my-app"], {
            "someLink": "/some/link"
        })
```

Note: Don't forget the super() call. Otherwise the standard setup data will be missing.

An application can also provide setup data globally, so that all front end applications can access them:

```
from cs.web.components.base.main import BaseApp, GLOBAL_APPSETUP_HOOK

@sig.connect(GLOBAL_APPSETUP_HOOK)
def update_app_setup_globally(app_setup, request):
    app_setup.merge_in(["links", "always-available"], {
        "someLink": "/some/link"
    })
```

11.2.4 mpq to query Morepath configuration

To quickly find out where a piece of configuration is configured, you can use the `mpq` script. You can use it like this:

```
$ bin/mpq view name=edit
```

or:

```
$ bin/mpq path model=cs.example.model.MyModel
```

See also the usage section in the Morepath documentation.

11.3 Frontend-specific tasks

11.3.1 Reusable Components

In order for an application bundle to provide React components, these need to be:

- registered in the Registry singleton object defined in `cs-web-components-base`
- marked for export by the application bundle

This is usually done in the `index.js` of the application. The template files generated by `webmake` provide an example `HelloWorld` component.

For the directory structure, refer to *Application Layout* (page 44). The file `./components/HelloWorld.jsx` implements the component `HelloWorld` while `./main.jsx` registers the component using the applications component namespace.

```
1 import { Registry } from 'cs-web-components-base';
2 import { prefixNS } from './helpers';
3 import HelloWorld from './components/HelloWorld'
4
5 Registry.registerComponent(prefixNS('MainComponent'), HelloWorld);
6
7 module.exports = [
8   HelloWorld
9 ]
```

In line 5 the component is registered, using the helper function prefixNS. This is necessary to use the component in a configured application.

The utility `webpack` which is used to create an application bundle from sources, only includes components that are used inside or exported from the bundle. Since reusable components may not necessarily be used in the application bundle, they should always be listed in the module.exports. This is achieved in lines 7-9.

This will also result in the component being importable as:

```
1 import { HelloWorld } from 'my-bundle-name';
```

Components provided by CONTACT Elements are also exported by the Registry in *cs-web-components-base* (page 57). E.g., in order to use *Favorites* (page 165) in your application, a simple import statement is all that is needed:

```
1 import { FavouriteView } from 'cs-web-components-base';
2
3 function Link(props) {
4     return (<FavouriteView />);
5 }
```

11.3.2 Application State with Redux

Managing Application State with Redux requires two parts:

- Actions that can be triggered to initiate the modification of application state.
- A reducer function, which is responsible for dispatching these and actions, and modifying the application state accordingly.

In order for a reducer to be invoked, it needs to be registered using the instances Registry.

```
import { Registry } from 'cs-web-components-base';
import { prefixNS } from './helpers';
import reducer from './reducers/reducers';
```

```
Registry.registerReducer(prefixNS('reducer'), reducer);
```

For details on how to implement reducers and action creators see redux.js.org . The default template installed by `webmake create` also installs an example in `js/src/actions/actions.js` and `js/src/reducers/reducers.js`

11.3.3 Accessing setup data

To access the setup data described in *Providing setup data from the backend* (page 47), use the `getAppSetup` function provided by `cs-web-components-base`. It returns the setup data as an Immutable object.

```
import { getAppSetup } from 'cs-web-components-base';

const basePath = getAppSetup().getIn(['appSettings', 'basePath']);
```

11.3.4 Documenting Components

Documentation for CONTACT Elements Web UI components is done using JSDoc and a specialized template that creates Restructured Text from JSDoc comments.

A basic example for a component with documentation in the style of Web UI's component documentation is:

```
import React, { Component, PropTypes } from 'react';
import { FavouriteView } from 'cs-web-components-base';

/**
 * A component that greets someone.
 *
 * React Properties
 * ----------------
 *
 * +-------------------+--------------+--------------+-----------------------------
 ↪--------------+
 * | Property          | Type         | Default      | Use                        ␣
 ↪               |
 *␣
 ↪+===================+==============+==============+============================================+
 * | title             | string       | \-           | Name of the person that␣
 ↪should be greeted |
 * +-------------------+--------------+--------------+-----------------------------
 ↪--------------+
 *
 * Automation Properties
 * ---------------------
 *
 * The component can be retrieved by a generated ``data-ce-id`` of
 * the form ``"hello-" + greetee``.
 *
 */
export default class HelloComponent extends Component {
   render() {
     return (<div data-ce-id={"hello-" + greetee}>
                Hello {greetee}
              </div>);
   }
}

HelloComponent.propTypes = {
```

```
32     greetee: PropTypes.string
33 };
```

Suppose this is the content of `js/src/components/HelloComponent.jsx` in module `xy.web.components`: In order to generate documentation for this component in docset `doc/web_ui/en` issue the command:

```
snapp -d <path_to_application> doc doc/web_ui/en
```

This will create a file in `doc/web_ui/en/src/_js/`, which is automatically included in the javascript API index. Note that jsdoc is also invoked by `snapp doc`.

To reference the component in your documentation use the generated reference:

```
:ref:`xy-web-components-components.HelloComponent.__default__`
```

11.4 Internationalization

Internationalization is based on labels configured in CONTACT Elements. The labels that can be used by the Web UI must have labels ids that consist of three parts, separated by `.`. The parts are:

- The constant `web`, as a marker that the label should be made available to the frontend.
- A domain id, that serves as a namespace to separate labels for different applications.
- The actual label id used by the frontend.

11.4.1 Using localized strings

i18n helpers

The following code can be used to create i18n helpers, that contain the domain for the current JavaScript library as default. The method described in *Creating an application template* (page 39) generates this code into a file named `i18n.js`.

```
import {i18ndb} from 'cs-web-components-base';

export const Format = i18ndb.makeFormat('my-domain');
export const formatStr = i18ndb.makeFormatStr('my-domain');
```

Using the i18n helpers

The following example renders the same label twice, once by getting the string to render, and once through a React component. The React component has an optional property `domainId`, that can be set to access labels from other domains.

```
import {Format, formatStr} from './i18n.js';

function MyComponent(props) {
    return (
        <div>
            <span>{formatStr('hello_world')}<span>
            <Format messageId='hello_world' />
        </div>);
}
```

Localized strings without helpers

To access localized string from other domains, the function `formatString` allows to supply the domain as a parameter:

```
import {i18ndb} from 'cs-web-components-base';

function MyComponent(props) {
    return <div>{i18ndb.formatString('label-id', 'other-domain')}</div>);
}
```

11.5 Customizing BaseApp

11.5.1 How to define navbar_items

The views *cs.web.components.base.main.get_navbar_items()* (page 56) and *cs.web.components.base.main.get_additional_navbar_items()* (page 56) each return a list of *navbar_items* represented in JSON. Each entry is a dictionary containing the following fields:

link: The hyper reference for the item **imageSrc:** The icon to display for the item **imageAlt:** Alt text for the item

Example: Loading an external website

```
@TestApp.json(model=BaseModel,
              name="additional_navbar_items",
              internal=True)
def get_additional_navbar_items(self, request):
    return [{"link": "http://www.google.de",
             "imageSrc": "http://www.google.de/images/branding/product/ico/googleg_
↪lodp.ico")),
             "imageAlt": "Google"}]
```

Example: Embedding the navbar icon

```
from cs.web.components.base.main import generate_embedded_image

@TestApp.json(model=BaseModel,
              name="additional_navbar_items",
              internal=True)
def get_additional_navbar_items(self, request):
    return [{"link": "http://www.google.de",
             "imageSrc": generate_embedded_image("/path/to/image"),
             "imageAlt": "Google"}]
```

CHAPTER 12

Testing cs.web Web Applications

12.1 Testing with Selenium

Frontend Tests can be implemented using the Selenium Framework for Browser Automation (c.f. http://www.seleniumhq.org/). `cs.web` provides several utilities to ease this task.

If testing of an application is already planned upon creation of the application, `webmake create` can be invoked with the parameter `--templates tests`, which will create a template in `tests/test_main.py`

12.1.1 Identifying `cs.web` components using Selenium

In order to run tests based on Browser-Automation with the Selenium framework, it is necessary to identify components in the rendered DOM and to identify the state they are in (e.g., a collapsible can either be collapsed or expanded) in order to interact with them.

The UI components provided by `cs.web` provide the attributes `data-ce-id` and `data-ce-state`, which can be used to query the DOM.

When subclassing `cs.web.automation.WebTest` it is easy to interact with elements using these attributes.

```
class WebTest(...):
    def get_ce_element(self, elements_id, parent=None):
        return CeElement(...)

class CeElement(...):
    def get_ce_state(self):
        return ...
```

The following code opens the sidebar and waits for the animation to finish:

```
sidebar_element = web_test.get_ce_element('ApplicationSidebar')
if sidebar_element.get_ce_state() == 'collapsed':
    burger_element = web_test.get_ce_element('Burger')
    burger_element.click()
```

CHAPTER 13

App Development References

This chapter provides references on the different components used to realize web applications with CONTACT Elements, both for backend and frontend tasks.

The basis for implementing applications based on JSX is the BaseApp application. Section *The BaseApp Morepath application* (page 54) provides a reference on how to use this class and the views associated with it.

For configuring or implementing the frontend of web applications, CONTACT Elements provides different libraries of standard components:

- `cs-web-components-externals`: the external libs for building application using *redux*
- *cs-web-components-base* (page 57): basic components and setups for building application using *redux*, depends on `cs-web-components-externals`

Section *cs-web-components-base* (page 57) describes the reusable components provided by CONTACT Elements. For info on using these components in implemented applications, see section *Reusable Components* (page 48) in chapter *Implementing Web Applications* (page 44). For info on how to use these components in configured applications, see *Configuring Web Applications* (page 10).

Local Navigation

13.1 Python APIs

13.1.1 The BaseApp Morepath application

Classes

class cs.web.components.base.main.**BaseApp**

Every Web UI application that wants to use the common framework should be a subclass of *BaseApp* (page 54). This application class is abstract, and is not mounted.

include(*libname*, *libver*)

Adds a dependency to this app, so that its embeddable resources will be embedded alongside this app.

render_includes()

Generates HTML <script>tags for embedding all resources this app depends on.

update_app_setup(*app_setup*, *model*, *request*)

Provide app specific setups for frontend:

```
class MyApp(BaseApp):
    def update_app_setup(self, app_setup, model, request):
        super(MyApp, self).update_app_setup(app_setup, model, request)
        app_setup.merge_in(["links", "my-namespace"], {
            "someLink": "/some/link"
        })
```

class cs.web.components.base.main.**BaseModel**

The Morepath model class all descendents of *BaseApp* (page 54) must either use directly, or through a derived class.

fillApplicationsList(*toolbar*)

Helper function to determine the list of applications assigned to `toolbar`

get_account_menu()

Determine the list of applications that should be available directly from the users's account menu. The default implementation adds all operations that are defined in the Toolbar named `webui_account`.

get_applications()

Determine the list of applications that should be available directly from the application menu. The default implementation adds all operations that are defined in the Toolbar named `webui_navigation`.

Morepath Views

cs.web.components.base.main.**get_page**(*model*, *request*)

This is the main function used to generate a HTML page. It includes all of the 3rd party JavaScript libraries, and the base components from the CONTACT Elements Web UI as <script> tags.

get_page (page 54) calls several internal Morepath views (see below), that can be overriden by applications to customize the generated HTML.

Attributes:

View Name	None
Return Type	html

cs.web.components.base.main.**default_document_title**(*model*, *request*)
Return the document title (the value of the title tag in html). Should be overriden by applications.

Attributes:

View Name	document_title
Return Type	string

cs.web.components.base.main.**setup**(*model*, *request*)
Static data, that can be computed in the backend, is included as a JSON encoded object in the <head> HTML element. As in *get_page* (page 54), this function calls (possibly) overriden internal views to do its work.

Attributes:

View Name	setup
Return Type	dictionary

cs.web.components.base.main.**favicon**(*model*, *request*)
Define the icon to be schown for the application by the browser. Must return HTML code to be embedded in the <head> tag.

Attributes:

View Name	additional_head
Return Type	html

cs.web.components.base.main.**default_additional_head**(*model*, *request*)
Return application specific HTML code to be embedded in the <head> tag.

Attributes:

View Name	additional_head
Return Type	html

cs.web.components.base.main.**default_enable_notify_changes**(*model*, *request*)

cs.web.components.base.main.**get_app_component**(*model*, *request*)
Return the registered name of the React component that represents the root element for the application. Must be implemented for all applications.

Attributes:

View Name	app_component
Return Type	string

cs.web.components.base.main.**get_base_path**(*model*, *request*)
Return the part of the URL path (without scheme / host / port) that is handled by the backend. The frontend uses this information to set up client side routing. Must be implemented for all applications.

Attributes:

View Name	base_path
Return Type	URL path

cs.web.components.base.main.**get_application_title**(*model*, *request*)
Could be overriden to define a custom application title.

Attributes:

View Name	application_title
Return Type	string

cs.web.components.base.main.**get_navbar_items**(*model*, *request*)
This view generates the default navbar_items of the application. Could be overriden to avoid rendering default nav bar items.

See also *cs.web.components.base.main.get_additional_navbar_items()* (page 56)

Attributes:

View Name	navbar_items
Return Type	json

cs.web.components.base.main.**get_additional_navbar_items**(*model*, *request*)
This view is an extension point for navbar_items. Must be overriden if additional nav bar items should be provided by derived application.

See also: *How to define navbar_items* (page 51)

Attributes:

View Name	additional_navbar_items
Return Type	json

cs.web.components.base.main.**get_app_items**(*model*, *request*)
Could be overriden to avoid rendering default application menu items.

Attributes:

View Name	application_menu_items
Return Type	json

cs.web.components.base.main.**app_help_id**(*model*, *request*)
A string representing the help id to the application specific help text. This will be used to generate the link to open the help text in documentation.

Must be overriden if help page should be linked for derived application.

Attributes:

View Name	application_help_id
Return Type	string

cs.web.components.base.main.**app_help_link**(*model*, *request*)
Link to application specific help.

If the link to application specific help should not be generated in the usual way, as using help ID that gets returned from *cs.web.components.base.main.app_help_id()* (page 56) view, this `application_help_link` view should be implemented the to return the expected link.

Attributes:

View Name	application_help_link
Return Type	link

13.2 Javascript-APIs

13.2.1 cs-web-components-base

Utilities

- *key-handlers* (page 101)
- *fetch* (page 83)
- *helpers* (page 98)

Redux Actions

- *notification* (page 62)
- *object-actions* (page 63)
- *actions/operations* (page 66)

Redux Reducers

- *reducers/object-store* (page 126)

Modules

- *Dialog* (page 81)
- *registry* (page 128)
- *Table* (page 130)
- *Tree* (page 141)
- *Messages* (page 108)

Basic Components

Components with basic functionality, the smallest reusable building blocks.

- *BannerContainer* (page 202)
- *Button* (page 67)
- *CreateButton* (page 188)
- *FileDropzone* (page 162)
- *FormControl* (page 94)
- *HelpReference* (page 163)
- cs-web-components-base-components.Icon.__default__
- *IconAndLabel* (page 68)
- *components/navigation/Link* (page 70)
- *ObjectFavoriteButton* (page 188)
- *Overlays* (page 124)
- *components/objectwidgets/ObjectLabel* (page 70)
- *components/SVGIcon* (page 72)

- *SearchField* (page 177)
- *CatalogField* (page 191)
- *Tag* (page 178)
- *Throbber* (page 179)
- *Thumbnail* (page 73)
- *TriStateCheckBox* (page 199)
- *formcontrols/WithTypeAhead* (page 96)
- *formcontrols/WithHyperlink* (page 95)
- *Zoomer* (page 179)

Main Components

Main set of components providing specific features and functionality.

- *ButtonGroup* (page 157)
- *ButtonToolbar* (page 157)
- *Collapsible* (page 158)
- *CompactHeader* (page 167)
- *ConnectedRelationshipTable* (page 167)
- *ConnectedRelationshipTreeTable* (page 168)
- *ContentBlock* (page 159)
- *ContentOperationToolbar* (page 161)
- *FileList* (page 169)
- *components/objectwidgets/ObjectSearch* (page 71)
- *Organizer* (page 173)
- *PersistentCollapsible* (page 175)
- *RelationshipGroup* (page 170)
- *StructureTree* (page 216)
- *CatalogField* (page 191)
- *Tab* (page 105)
- *Tags* (page 178)
- *DisplayForm* (page 85)

Plugins and Plugin Container Components

The functionality of some components can be configured through a plugin system.

- *ObjectContent* (page 176)
- *tiles/PersonTile* (page 140)
- *RelatedObjects* (page 176)
- *TileHelpers* (page 140)
- *PDFViewer* (page 217)
- *ImageViewer* (page 158)

Infrastructure Components

Components providing data infrastructure and operations functionality.

- *ContextObjectSetter* (page 180)
- *ContextObjectWrapper* (page 180)
- *FileSelectionProvider* (page 169)
- *Listener* (page 152)
- *OperationModal* (page 172)
- *OperationTrigger* (page 89)
- *Provider* (page 152)
- *WithOperations* (page 74)

Layout Components

Components taking care of layouting.

- *SplitterLayout* (page 200)
- *VerticalBlockLayout* (page 202)

Operations

Modules for executing operations.

- *OperationTrigger* (page 89)
- *OperationHandler* (page 86)
- *OperationScope* (page 89)
- *operations/actions/operations* (page 112)

13.3 Known Problems

13.3.1 Text Input in Internet Explorer 11

Due to a bug in the key handling for text fields in Internet Explorer 11, the input of text fields may become corrupted. The issue is explained in `Issue 7027` of the React project.

If the components state is update in parallel to a change to a text field, a race condition between Reacts state handling and the key event handling of IE11 may cause the input of text fields to be overwritten with old data.

As a workaround it is necessary to synchronize state changes that are triggered by an onChange event in the TextInput component, to be executed after the changes have been persisted. A simple example may look as follows:

```
onValueChange(newValue) {
   setState(
       {value: newValue},
       () => doAsynchronousStateChange()
   );
}

render() {
   return (
```

```
      <TextInput onValueChange={this.onValueChange}
                 value={this.state.value}
                 ... />
   );
}
```

13.3.2 Conflicting Properties in configured Components

Note that when developing applications using the json-based configuration for pages, the components that are directly configured into the site are automatically provided with several properties, e.g. from `react-router` or the `contextObject` property. These properties may clash with locally defined properties that are not specified for the components. If properties are passed down via the Spread-Operator, even components further down in the component hierarchy may be affected. If you experience unusual behaviour due to properties having unexpected value, you should have a look at the properties passed to your components from the configuration framework.

13.4 Concept for selected objects

As a convention, all components that derive their content from an object retrieved via the REST API, receive that object in a property named `contextObject`. If the selection can contain multiple objects (eg. in a table with multi-select enabled), these objects are provides as `contextObjects`. The components `ContextObjectSetter` and `ContextObjectWrapper` can be used as wrappers around other components, that define a selection in one component as contextObject in another component.

The selections are held in the Redux store with names assigned through the wrapper classes' properties. These names form a tree-like structure, where the child nodes denote selections that depend on their parent selection. The parts of a selection name form a path separated by `/`.

Example: Using the context wrapper components

This example shows how to connect a tree component, showing an organization structure, to a table view of the persons that belong to the organization currently selected in the tree.

In one part of the page, the tree is rendered, and the current selection is made available under the path `orgTree`:

```
<ContextObjectSetter setPath="orgTree">
    <RestTree .../>
</ContextObjectSetter>
```

Elsewhere, the table is rendered, taking its `contextObject` from the selection under `orgTree`, and providing its selection in turn as `orgTree/employee`:

```
<ContextObjectWrapper readPath="orgTree" setTag="employee">
    <RelationshipTable .../>
</ContextObjectWrapper>
```

13.4.1 Index

cs-web-components-base

context-objects

Description

Contents

Functions

setContextObjectList

```
setContextObjectList(path: string, values: Immutable.List)
```

Takes:

- *path*: Path to store the selection for, segments separated by '/'
- ***values*: A list of (mixed) objects or strings, a string** is assumed to be the @id of an object

Store a list of selected objects for a path. The path is used to retrieve the objects from the store. We always store the @id's of the objects, not the objects themselves, to avoid stale object states.

Objects not already present in the store will be fetched, so that not every using component has to do this.

setContextObjectsBatchLoad

```
setContextObjectsBatchLoad(path: string, values: Immutable.List)
```

Takes:

- *path*: Path to store the selection for, segments separated by '/'
- *values*: A list of object identifiers [@id, restname, keys}]

Store a list of selected objects for a path. The path is used to retrieve the objects from the store. We always store the @id's of the objects, not the objects themselves, to avoid stale object states.

Objects not already present in the store will be fetched, so that not every using component has to do this.

The objects will be fetched from the server by a single call.

actions/fetching

Description

Contents

Functions

singleFetch

```
singleFetch(url: string, callOnce: func)
```

Takes:

- *url*: URL to fetch from
- *callOnce*: Callback that is invoked on successful fetching

Wrapper around getJSON that manages multiple simultaneous GET requests for a URL. A request is uniquely identified by its URL and may be in three states:

- Unknown: There is currently no pending request for this state.
- Pending: A request is pending, subsequent calls to singleFetch yield the Promise associated with this URL
- Error: A request was done for the URL and failed.

Since each call to the returned promise's then will be executed (multiple times if the user code is called multiple times) parameter callOnce allows to attach a callback that is invoked only once this Promise is fulfilled.

The signature of callOnce is as follows:

```
callOnce(data) => result|Promise
```

notification

Local Navigation

Description

Contents

Functions

autoNotifyPromiseRejection

Similar to Higher Order Component: "Higher Order" dispatcher to add error notification for other thunks that return promise.

object-actions

Local Navigation

Description

This module provides actions to manipulate the global object store. Use this module's action creators to access objects via the REST API from the frontend.

The reducers that store these objects are defined in *reducers/object-store* (page 126). See this modules description on how the object-actions API handles errors and pending requests.

All actions in the store will return thenable objects that will resolve if the operation is completed, or be rejected, if the operation fails.

Contents

Functions

createObject

```
createObject(collectionUrl: string, values: object, refetchRelships: array):␣
↪ReduxAction
```

Sends a PUT request to create the provided object from the server.

deleteObjectFromStore

```
deleteObjectFromStore(url: string): ReduxAction
```

This action creator sends a DELETE request for the object identified by `url` to the server, and – if successfull – removes the deleted object from the store.

editObject

```
editObject(obj: object, newValues: object): ReduxAction
```

Takes:

- *obj*: a REST API object to be modified
- ***newValues*: an object specifying the** new values for this object

Sends a POST request to modify the given object to the server.

ensureFileTargetsAvailable

```
ensureFileTargetsAvailable(objectId: string): ReduxAction
```

/** Search for the object in the global store. If that object is not part of the global store or the object does not contain the key `targets` in the `relship:files` section `fetchObject` is called.

See also

- *objectsById* (page 127)

fetchClassRelships

```
fetchClassRelships(classname: String, force: Boolean): ReduxAction
```

Takes:

- *classname*: The name of the class for which to get the data
- *force*: If true, reload if already stored

Fetch from the backend the relationship metadata for a given class

fetchCollection

```
fetchCollection(url: string, params: object): ReduxAction
```

Takes:

- *url*: url of the collection to fetch
- *params*: an object specifying additional query parameters

Fetch a collection of objects and store them in *objectsById* (page 127).

Parameter `getter` should only be used for special cases where the collection data are accessible via another key as `objects`. E.g. in activitystream the result should be accessed by providing `data => data.postings`.

See also

- *objectsById* (page 127)

fetchObject

```
fetchObject(url: string, viewname: string, withFiles: string): ReduxAction
```

Retrieve an object from the server and store it in the global store. If fetching fails an error will be stored in *errorsById* (page 126) for the url used to fetch the object.

If an object is already in the store, a new fetchObject call will reload the object from the server. The @type link for the object will be fetched when the object is returned from the server.

See also

- *objectsById* (page 127)

fetchObjectWithRelships

```
fetchObjectWithRelships(url: string, viewname: string, relships: array):␣
↪ReduxAction
```

This action creator fetches the object identified by the provided URL, and – if successfull – the relations provided in array `relships`.

See also

- *fetchRelship* (page 65)
- *fetchObject* (page 64)

fetchRelship

```
fetchRelship(url: string, viewname: string, params: object)
```

Takes:

- *url*: url of the relship to fetch
- *viewname*: the backend view to load

Fetch a relship defined on an object. The url parameter should be retrieved from an object retrieved via REST API.

```
import { fetchRelship, connect } from 'cs-web-components-base';

const ExampleComponent = props => {
  const object = props.objectsById.get(props.objectId);
  const objectUrl = object.getIn(['system:relships', 'relships', props.relshipId]);
  this.props.fetchRelship().then(() => {

  });
};

const mapStateToProps = state => {
  return {getObjectById: objectId => state.objectsById.get(objectId)};
};

connect(mapStateStateToProps, {fetchRelship})(ExampleComponent);
```

fetchRelships

```
fetchRelships(relships: array): ReduxAction
```

Fetch an array of relships. Each relship should be an object specifying the parameters for *fetchRelship* (page 65)

See also

- *fetchRelship* (page 65)

fetchType

```
fetchType(url: string, force: boolean): ReduxAction
```

Takes:

- *url*: The url of the type to be ensured.
- *force*: reload an already loaded type if set

This action creator loads the type identified by `url`. By default the type will not be reloaded (as we assume this data to be static), if the action is executed multiple times. To reload an already loaded type, set `force` to `true`.

Type URLs are usually obtained from an object's type field:

```
const object = state.objectById(some_url);
fetchType(object.get('@type')).then(some_func);
```

See also

- *typesById* (page 127)

fetchTypes

```
fetchTypes(urls: array, force: boolean): ReduxAction
```

Takes:

- *urls*: an array of urls
- *force*: set true to reload

Fetch a list of types.

See also

- *fetchType* (page 66)

actions/operations

Local Navigation

Description

This module provides actions to retrieve operation information objects from the server. This forms the basis for retrieving information about available operations for a class or object in general or in a specified operation context, and is the first step to executing operations in Web UI.

Contents

Functions

fetchOperationContext

```
fetchOperationContext(objectType: string, contextName: string)
```

Takes:

- *objectType*: a CE classname
- *contextName*: the name of an operation context as defined in Elements

Retrieve an operation context with operation info objects already resolved.

fetchOperationInfo

```
fetchOperationInfo(objectType: string)
```

Takes:

- *objectType*: a CE classname

Retrieve the operation infos for the object type provided.

fetchRelshipOperationInfo

```
fetchRelshipOperationInfo(url: string)
```

Takes:

- *url*: the URL to retireve the operations for the relationship context

Retrieve the operation infos a relationship context

Button

Local Navigation

Description

The Button module provides different types of button components. All components in this module receive the following properties:

Name	Type	Default	Description
buttonStyle	string	-	Determines the button style
className	string	-	A custom css className to attach to the component
title	string	-	A tooltip for the component

In addition to these properties, each component expects a custom set of properties, corresponding to its rendered content.

Contents

Classes

- *DropdownIconButton* (page 154)
- *DropdownIconTextButton* (page 155)
- *DropdownTextButton* (page 155)
- *IconButton* (page 156)
- *IconTextButton* (page 156)
- *TextButton* (page 156)

Functions

LinkAsIconTextButton

Renders a Link disguised as an IconTextButton. In addition to the properties in *Button* (page 67) and *components/navigation/Link* (page 70) it receives the following properties:

Name	Type	Default	Description
label	string	-	The label that is displayed in the button
iconName	string	-	Load an icon defined in the backend by its resource id
iconSrc	string	-	Load an icon by the provided URL.

LinkAsTextButton

Renders a Link disguised as a TextButton. In addition to the properties in *Button* (page 67) and *components/navigation/Link* (page 70) it receives the following properties:

Name	Type	Default	Description
label	string	-	The label that is displayed in the button

IconAndLabel

Local Navigation

Description

The IconAndLabel module provides a basic layout components for rendering an icon and a text label. In addition convenience components are provided for SVGIcon and Link rendering.

React Properties

Property	Type	Default	Use
className	string	undefined	Additional CSS class
label	node	-	Label to be displayed
allowEmptyIcon	bool	-	If true and icon cannot be rendered, renders a white space instead. Else only label is rendered.

Contents

Functions

Constants

IconAndLabel: Basic component that expects an icon and label node.

React Properties

Property	Type	Default	Use
icon	node	undefined	Icon to be displayed

SVGIconAndLabel: Convenience component for rendering a SVG icon label.

React Properties

Property	Type	Default	Use
src	string	-	See SVGIcon property “src”.
name	string	-	See SVGIcon property “name”.
size	string	“sm”	See SVGIcon property “size”.

SVGIconAndLink: Convenience component for rendering a SVG icon with a link.

React Properties

Property	Type	Default	Use
src	string	-	See SVGIcon property "src".
name	string	-	See SVGIcon property "name".
size	string	"sm"	See SVGIcon property "size".
to	string	-	See Link property "to".

components/navigation/Link

Local Navigation

Description

Link wrapper component, that generates a react-router Link if the "to" prop points to the current page (client side routing). Otherwise, a standard <a> tag is returned.

components/objectwidgets/ObjectLabel

Local Navigation

Description

A label that is used to display an object.

Note: ObjectLabel is deprecated. Please use the IconAndLabel module.

React Properties

Property	Type	Default	Use
text	node	-	Label content to be displayed
link	string	undefined	Link to follow when clicking this label
icon	string	undefined	URL to icon for this object
iconSize	string	undefined	Size of icon for this object
statusColor	string	undefined	Color of this object's status
statusLabel	string	undefined	Status description text
className	string	undefined	Additional CSS class
onTextClick	string	undefined	Function to call when clicking this label
onIconClick	string	undefined	Function to call when clicking this object's icon

components/objectwidgets/ObjectSearch

Local Navigation

Description

A component that allows an interactive search in the EnterpriseSearch, history and favorites.

React Properties

Property	Type	Default	Use
value	string	''	search value
searchPlaceholder	string	undefined	Placeholder text if the search value is empty
searchClasses	stringarray	undefined	Predefined search classes
excludeClasses	stringarray	undefined	Classes that should be excluded from search
hideDefaultTags	boolean	false	Hide predefined search classes tags in search field
onSelectItem	func	undefined	A callback fired when an object from the drop down is selected. An object with the rest url, title, icon url and search type is passed in

components/StatusIcon

Local Navigation

Description

The component displays a status "icon" in predefined sizes.

React Properties

Property	Type	Default	Use
label	string	•	Status description
color	string	•	Status color (css-compatible)

components/SVGIcon

Local Navigation

Description

The component display SVG icon in predefined sizes.

React Properties

Property	Type	Default	Use
name	string	•	Name of configured icon
src	string	•	URL to load the icon
size	string	sm	Size of that image. Can be: sm, md, lg
fallbackSVG	string	•	URL to load the fallBackSVG

Note: If `src` is given, it will be used to load the icon. Otherwise the component tries to generate the url for configured icon according to the `name`. If `src` is not an SVG, the component renders `fallbackSVG`. If `fallbackSVG` is not given, the component renders `src` as an image.

components/TableWrapper

Local Navigation

Description

A Control to show a table defined by its column definition and row data.

React Properties

Property	Type	Default	Use
tableDef	Immutable.Map	undefined	The table definition
rowValues	Immutable.List	undefined	A list of rows. Each row is an array of cell values corresponding to each column.
onSelectionChanged	func	-	Function to be called when item is selected. The selected indices will be passed to this function.
table	func	**Predefined.** DefaultTable	The table
initFilter	string	empty	Initial filter string
initSelected	**Immutable.** OrderedSet	empty	Initial selections

Thumbnail

Local Navigation

Description

The component displays images and fallbacks in predefined sizes and a consistent format.

Property	Type	Default	Use
imgSrc	string	•	URL to load the image
fallbackSrc	string	•	URL to load the fallback
size	string	md	Size of that image. Can be: sm, md, lg

Note: If `imgSrc` is given, it will be used to load the image. Otherwise `fallbackSrc` will be used to load the fallbackImg. If `imgSrc` and `fallbackSrc` are not given, the component renders a placeholder image.

WithOperations

Local Navigation

Description

This module provides the `WithOperations` Higher Order Component. This HOC provides the wrapped component with an operations property that contains a set of operations, selected by WithOperations based on its parameters.

Note: Operations must be Web-UI enabled, or they will be filtered out of the list, before passing it to the wrapped component. Operations which are not Web-UI enabled can be recognized by inspecting the payload of the fetch call. The field `submitURL` will be undefined for this operation.

Operations can be restricted by either giving an operation context name, or a list of operation names (or both). The signature of WithOperations is as follows:

```
WithOperations(
  Component: Component,
  {
    contextType: String,
    operationContextName: String,
    operationNames: [String]
  }
):Component
```

The structure of the args-parameter is as follows:

- contextType: type of the object, if not specified the components contextObjects will be used to determine the object
- operationContextName: string
- operationNames: array

The parameters `operationNames` and `operationContextName` specify the method of how to determine the set of operations provided to the wrapped component and should be considered mutually exclusive.

Operations always belong to an object class. This class is either given directly as the parameter `contextType` to `WithOperations`, or – if no `contextType` is specified – the component will use the base class of the provided `contextObjects` as the class for which to request operations.

Note: The context type of the objects or classes for which operations should be fetched must be available for this component to work. If you use objects without *fetchObject* (page 64), you should use *fetchType* (page 66) to fetch relevant type information.

The set of operations provided to the wrapped component is determined either by providing argument `operationContextName` or `operationNames`. If an operation context is specified, all operations listed in the context will be passed to the wrapped component, if a list of operations is specified, these operations will be passed.

React Properties passed to the wrapped component

Property	Type	Use
operations	Immutable.List(opInfo)	The operations requested from the backend

React Properties of the wrapping component

Property	Type	Default	Use
contextObject	Immutable.Map	-	For a single select operation on the provided object.
contextObjects	**Immutable.List(** contextObject)	-	For a multi select operation on the selected objects.

Note that contextObject and contextObjects are mutually exclusive, and none should be provided if a `contextType` is specified.

Example

```
const MyOperationList = WithOperations(
  props => {
    return (
      <ul>
        props.operations.map(opInfo => (<li>opInfo.get('label')</li>))
      </ul>
    );
  },
  ['CDB_ShowObject', 'CDB_Modify', 'CDB_Workflow']
);
```

Contents

Functions

WithOperations

Implementation of the `WithOperations` HOC.

containers/detail-wrapper

Local Navigation

Description

Wraps a React component, so that it can be used as the detailComponent in a generic frame.

Contents

Classes

Functions

Buttons

Description

This module provides generic and concrete default buttons to be used in Dialog Footers. Though it is possible to use the more generic *TextButton* (page 156) component, it is encouraged to use these Buttons to enforce a unified User Interface across applications.

Generic Buttons

These Button components override the `buttonStyle` of *TextButton* (page 156) with a style specific to the semantics of the button. Other properties are according to *TextButton* (page 156).

Use these buttons if you would like to provide a label for a button that is not found among default buttons.

Default Buttons

These Buttons provide a default label to symbolize semantic actions common to all web applications, like `Cancel` or `Save`.

The `buttonStyle` of these components may be overridden, though it is generally not advised to do so, since the default matches the semantics of the action symbolized by the label used.

Even though these components provide a generic tooltip, users are advised to provide a custom tooltip, that matches the application by passing either an i18n label id (see *Internationalization* (page 50)) as property titleId or a string title which directly sets the tooltip.

React Properties

All default buttons expect the following properties:

Name	Type	Default	Description
buttonStyle	string	default	Determines the color of the button
title	string	-	A tooltip for the component
titleId	string	yes	A tooltip provided as label id

Contents

Functions

Add

```
Add(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Add` button. See *Default Buttons* (page 76)

Apply

```
Apply(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Apply` button. See *Default Buttons* (page 76)

Cancel

```
Cancel(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Cancel` button. See *Default Buttons* (page 76)

Change

```
Change(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Change` button. See *Default Buttons* (page 76)

Close

```
Close(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Close` button. See *Default Buttons* (page 76)

Confirm

```
Confirm(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

A generic Confirm Button. See *Generic Buttons* (page 76)

Copy

```
Copy(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Copy` button. See *Default Buttons* (page 76)

Create

```
Create(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Create` button. See *Default Buttons* (page 76)

Delegate

```
Delegate(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Delegate` button. See *Default Buttons* (page 76)

Delete

```
Delete(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Delete` button. See *Default Buttons* (page 76)

Export

```
Export(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Export` button. See *Default Buttons* (page 76)

Import

```
Import(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Import` button. See *Default Buttons* (page 76)

No

```
No(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `No` button. See *Default Buttons* (page 76)

Ok

```
Ok(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Ok` button. See *Default Buttons* (page 76)

Rename

```
Rename(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Rename` button. See *Default Buttons* (page 76)

Reset

```
Reset(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Reset` button. See *Default Buttons* (page 76)

Save

```
Save(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Save` button. See *Default Buttons* (page 76)

Select

```
Select(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Select` button. See *Default Buttons* (page 76)

Share

```
Share(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Share` button. See *Default Buttons* (page 76)

Submit

```
Submit(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Submit` button. See *Default Buttons* (page 76)

Yes

```
Yes(props: object): element
```

Takes:

- *props*: React Properties

Returns: a rendered element

Default `Yes` button. See *Default Buttons* (page 76)

Dialog

Local Navigation

Description

This module provides sets of components to render different types of dialogs, as well as typical components which are rendered inside a dialog.

Dialog (page 183) and derived components are containers that display their children in a modal dialog simulated with CSS and Javascript.

Displaying Buttons in Dialogs

The module *Buttons* (page 76) provides a set of stock buttons to use in Dialogs. If your dialog will not use any buttons, you may omit the footer of the dialog by passing in the property `hideFooter`. The components *SingleSelection* (page 187) and *SingleListSelection* (page 187) can be used to display dialogs without a footer.

Resizing

Dialogs can either be configured to use a fixed width or be resizable. The properties `size` and `sizes` can be supplied a value or a list of values from the constants `SIZE_*`.

Options

Options are special boxes for notifying the user of events, or letting the user make simple choices, such as ok/cancel or yes/no. Option Components provided are:

- *Alert* (page 185)
- *Message* (page 185)
- *Notice* (page 185)
- *YesNo* (page 186)
- *YesNoCancel* (page 186)

React Properties for Options

Option Dialogs require a special subset of Dialog Properties listed below.

Name	Type	Default	Description
title	string	-	The title displayed in the dialogs header
onHide	func	-	Function that is invoked when the dialog wants to hide itself
size	string	SIZE_SMALL	The size of the dialog
show	bool	true	True if the dialog should be shown

Contents

Modules

- *Buttons* (page 76)

Classes

Functions

Constants

SIZE_LARGE: Dialog has a width of 90vw **SIZE_MEDIUM**: Dialog has a width of 70vw **SIZE_SMALL**: Dialog has a width of 50vw

fetch

Local Navigation

- *Description* (page 83)
- *Contents* (page 83)
  - *Functions* (page 83)

Description

Provide a set of wrapper functions to fetch data from the backend, or send data to the backend. All functions return a Promise object, so callers have a consistent way to attach follow-up actions to them.

Contents

Functions

deleteObject

```
deleteObject(url: string): thenable
```

Takes:

- *url*: the URL to call

Issue a DELETE request for a resource identified by the URL.

fetchAndCheck

```
fetchAndCheck(url: string, init: object): thenable
```

Wrapper around the standard fetch API, that throws errors for all returned status codes not in the 2xx range. See https://github.com/github/fetch#handling-http-error-statuses This is a low level function that has the same arguments as the standard (to be ...) HTML5 fetch function.

getJSON

```
getJSON(url: string, params: Object): thenable
```

Takes:

- *url*: the URL to call. If needed, can already contain query parameters
- *params*: optional query parameters to append to the URL

Issue a GET request, and interpret the result as JSON.

postForm

```
postForm(url: string, formData: FormData): thenable
```

Takes:

- *url*: the URL to call
- *formData*: the request payload

Issue a POST request with the payload formatted as form data, and interpret the result as JSON.

postJSON

```
postJSON(url: string, data): thenable
```

Takes:

- *url*: the URL to call
- *data*: the request payload, will be formatted as JSON

Issue a POST request with JSON formatted request body, and interpret the result as JSON, if a response body was received.

putJSON

```
putJSON(url: string, data: object): thenable
```

Takes:

- *url*: the URL to call
- *data*: the request payload, will be formatted as JSON

Issue a PUT request with JSON formatted request body, and interpret the result as JSON. For responses with HTTP 204, 205 the result is undefined as these response codes must not have a payload.

form-constants

Local Navigation

Description

This module provides constants used by the form module.

Contents

Functions

Constants

ButtonActions: The possible actions that a pre-submit dialog can associate with buttons. The values must match the corresponding constants in the backend (see class cs.web.components.ui_support.frontend_dialog.FrontendDialog), to avoid another translation step. **OperationMode**: Internal modes for a component that shows a UI to perform a backend operation. The UI will typically reflect the current mode in the way it renders, and which actions are available to the user. *OperationHandler* (page 86) Components may access the current mode of the operation via `props.operationState.get('mode')`.

Possible values are:

- INITIAL: Default mode, no user interaction yet
- STARTING: Set when runOperation is called, but state has not been initialized
- STARTED: The user started an operation, interactions with the operation's UI are happening here
- SUCCEEDED: The operation has been successfully completed.
- FAILED: The execution of the operation has failed.
- CANCELLED: The operation has been cancelled by the user.

The values `SUCCEEDED`, `FAILED` and `CANCELLED` will be reset directly after it has been propagated to the operation handler via `componentWillReceiveProps`. **ResultType**: These constants correspond to CDB::IOperation::ResultType, defined in src/cdbidl/CDBIOperation.idl.

DisplayForm

Local Navigation

Description

This component renders form configurations as provided by the FormInfoBase backend component. Pass the JSON payload received as property `formData`, which should be an `Immutable.Map`.

OperationHandler

Local Navigation

Description

Operation Handlers are container components who can be used to allow the user to control the execution of an operation. Usually they will present a configured form to the user, and – on user interaction – send actions to the saga running the operation.

In order to do that, the HOC *WithOperationHandler* (page 88) provides several functions and fields to the wrapped component to interact with the saga.

Handler Names

The operation handler is identified by its handler name. This can either be set for a whole class of operation handlers when creating the operation handler component with *WithOperationHandler* (page 88) or specifically for one instance of a component, by passing the property handlerName to the component. Note that this property must not change during the handlers React component lifecycle, as the handler will be registered in componentWillMount and unregistered again on componentDidUnmount.

Operation State

The current state of the operation will be passed in as property `operationState`. This `Immutable.Map` contains the following fields describing the current state of the operation:

- **mode**: The mode the operation's execution is currently in. See *form-constants* (page 85).
- **info**: The operation info object, describing the operation that is executed.
- **params**: Parameters with which the operation was initialized.
- **values**: The current values of the operation state.
- **registers**: The registers/fields of the operation.
- **unchecked**: A list of unchecked fields.
- **query**: Presubmit Dialog Hooks allow a dialog to be displayed to the user. If a dialog hook calls the set_dialog function, the dialog configuration will be passed to the operation handler in this field.

- **queryCancel**: Boolean that is set, if the user wants to cancel the operation, but values have changed preventing cancellation.
- **displayForm**: Boolean that is set if the form should be displayed.
- **submitDisabled**: Boolean that is set if the OperationHandler should disable submitting the operation.
- **pendingChecks**: an `Immutable.Map` containing the fields for which asynchonuous value checks are currently pending.

The following fields are provided to determine the current mode of the operation handler:

- **isInstanceActive**: An operation has been started, though the operation state may not yet be initialized.
- **isInstanceRunning**: An operation has been started, and its operation state has been initialized.

Function Properties passed to the wrapped component

Basic interaction is achieved via the following functions:

runOperation: start the execution of an operation. For parametrization, see *Operation Parameters* (page 115).

```
runOperation(
  operation: Immutable.map,
  operationParameters: object,
  operationHandler = undefined: string,
)
```

onChangeValues: If the user changes one or more values, call this function to update the operation state. `operationState.get('values')` will yield the updated values. See also: *changeOperationValues* (page 113)

```
onChangeValues(
  values: Immutable.map,
  unchecked: boolean,
)
```

onSubmit: This will enter presubmit state. Validity Checks and Dialog Hooks will be run, and the operation will be submitted, or the operation state will be updated accordingly.

```
onSubmit()
```

For ways to cancel a running operation, the following functions are provided:

onCancel: Cancel the operation. If values have been changed, this will result in `operationState.get('queryCancel')` to be set. The operation handler may now either continue the operation or cancel it, by invoking `queryCancelBack` or `queryCancelForce`. To bypass checking for changed values, set parameter `force` to `false` See also *cancelOperation* (page 113)

```
onCancel(force = false)
```

queryCancelForce: If `queryCancel` is set, this will cancel the operation.

```
queryCancelForce()
```

queryCancelBack: If `queryCancel` is set, this will continue the operation.

```
queryCancelBack()
```

forceCancel: Force cancellation of the operation, regardless of changed values. This is a conveniance function for calling cancelOperation with parameter `force` set.

```
forceCancel()
```

If a presubmit hook displays a dialog to the user, the following functions should be mapped to the buttons of the dialog if available:

preSubmitDialogBackToDialog: Update the operation's values according to `attribute`, and return to the configured form.

```
preSubmitDialogBackToDialog(
  attribute: {
    name: string,
    value
  }
)
```

preSubmitDialogCallServer: Update the operation's values according to `attribute`, and invoke the dialog hooks again.

```
preSubmitDialogCallServer(
  attribute: {
    name: string,
    value
  }
)
```

preSubmitDialogSubmit: Update the operation's values according to `attribute`, and submit the operation. Note that you should not call `onSubmit` in the context of a dialog hook.

```
preSubmitDialogSubmit(
  attribute: {
    name: string,
    value
  }
)
```

preSubmitDialogCancel: Cancel the operation. This will query the user for loss of changed data. There is no way to override this. Note that calling `onCancel` or `forceCancel` will not work in the context of a dialog.

```
preSubmitDialogCancel()
```

The `attribute` value that three of these functions may provide is used to update the values in the operation state. These are usually provided in the dialog configuration and should be retrieved from the operation state by the component managing the user interation.

Contents

Functions

WithOperationHandler

HOC for creating *OperationHandler* (page 86) components. Wrapping a component with this HOC has two effects:

- The component is connected to the form_with_operations actions and reducers allowing the component to execute operations, display the operation state and interact with the saga executing the operation.
- it is registered as a provider using the provider-name provided as a property. OperationTrigger components may trigger an operation to be executed using this component.

Parameters:

- Component: The component to be wrapped
- defaultHandlerName: The default handler name, with which the component will be registered. Component Instances may override this, by passing in the property `handlerName`.

OperationScope

Local Navigation

- *Description* (page 89)

Description

The OperationScope provides a context for the execution of kernel operations across a hierarchy of components.

Using the React Context, OperationScope provides a function for components to register themselves as OperationHandlers. OperationTrigger components use the surrounding OperationScope to trigger operation execution for operation handlers, registered in the scope.

OperationTrigger

Local Navigation

- *Description* (page 89)
  - *Handler Names* (page 89)
  - *Running Operations* (page 90)
- *Contents* (page 90)
  - *Functions* (page 90)

Description

This module provides the HOC *WithOperationTrigger* (page 90) to implement operation Wrapping a component with an OperationTrigger provides a `runOperation` callback.

Handler Names

While operations may specify a default handler they should be invoked with by setting an appropriate value for field `presentation_id` of the operation configuration (see `operations_presentation_ids`) specialized applications want to customize the way in which operations are handled. When creating an operation trigger component with the `WithOperationTrigger` HOC, a handler name may be passed as the second parameter to have instances of this component use the operation handler identified by this name.

Note that this parameter may in turn be overriden by passing a handlerName property to a specific instance of the operation trigger component.

So the way a handler name is determined is as follows:

1. `<MyOperationTrigger handlerName={'foo1'} />`
2. `const MyOperationTrigger = WithOperationTrigger(MyComponent, 'foo2')`

3. `operationInfo.get('presentation_id')`

Running Operations

The wrapped component may invoke an operation fetched with WithOperations by calling runOperation, passed as property.

```
runOperation(operation, operationParameters)
```

This callback will trigger an operation in the surrounding operation scope. `runOperation` receives an operation information object as provided to frontend components by the *WithOperations* (page 74) HOC, as well as an object that specifies the parameters to the operation.

Given an operation `opInfo`, the operation may be called like

```
class MyComponent extends React.Component {
   doRunOp() {
       const {runOperation, opInfo, contextObject} = this.props
       runOperation(opInfo, {objects: Immutable.List([contextObject, ...])});
   }

   render() {
       // Render the component
   }
}

export default WithOperationTrigger(MyComponent);
```

For a comprehensive list of possible parameters, see *Operation Parameters* (page 115).

Properties passed to the wrapping Component (this component) are:

Name	Type	Default	Description
handlerName	string	-	Override the handlerName provided by the operation. Optional.
defaultOperationParameters	object	-	defaultOperationParameters are merged into the parameters provided to startOperation. Optional.

Properties passed to the wrapped component are:

Name	Type	Description
runOperation	func	Start the provided operation in the surrounding scope.

Contents

Functions

WithOperationTrigger

The Higher-Order Component WithOperationTrigger should be used to connect your component to the frontend-based operation state, which stores data relevant to the operation execution and determines the way user interaction (retrieving data required for the operation) is run.

The WithOperationTrigger HOC wraps a component and passes a wrapped `runOperation` function to it, which has the following interface:

```
runOperation(
    operation: Immutable.Map,
    {
        contextObjects: Immutable.List,
        parameters: Immutable.Map,
        resultInNewTab: Boolean,
        successActions: [ReduxActions...],
        failedActions: [ReduxActions...],
        cancelActions: [ReduxActions...],
        generalActions: [ReduxActions...],
        suppressHandler: Boolean,
        forceHandler: Boolean,
        nonBlocking: Boolean,
        dialog: String
    }
)
```

For a detailed description of all parameters see the action runOperation

form/dialog_hooks

Local Navigation

Description

Implementation of the dialog-hooks API

Contents

Classes

Functions

callDialogHooks

```
  callDialogHooks(
 dialogHooks: Immutable.List,
 oldValues: Immutable.Map,
 changes: Immutable.Map,
 operationState: Immutable.Map,
 newValues: Immutable.Map,
 registers: Immutable.List,
 preventSubmitReasons: Immutable.Set,
 formStateSetter: function,
```

```
  displayErrors: function
): undefined
```

Takes:

- ***dialogHooks*: List of dialog hooks to call, in the order** as given in the list.
- *oldValues*: Mask values before applying changes
- *changes*: New values to set
- *operationState*: Current operation state from backend
- *newValues*: Mask values after applying changes
- ***registers*: Mask settings: list of registers, each** containing a list of fields
- ***preventSubmitReasons*: Set of strings: if at least one** entry is in the set, the submit button of the form will be disabled. The hooks that set values, are also responsible for removing them, once the reason no longer applies.
- ***formStateSetter*: Callback function that can be used by the** async backend hooks to inject changes into the dialog.
- ***displayErrors*: Callback function that can be used to display errors** received from async backend hooks.

Returns: object Potentially changed values and mask settings as an object with the keys "values", "registers" and "preventSubmitReasons".

ATTENTION: this is a preliminary API, and subject to change in future versions of the product!

Call the configured dialog hooks for a configured dialog.

Besides dialogHooks and formStateSetter, the parameters of this function are used to construct a DialogHooksContext object. That object provides the API for the hook implementations, and manages the changed values and settings.

callPreSubmitHooks

The user has pressed "Submit", now run any configured hooks prior to actually submitting the operation. Hooks can prevent the submit to proceed, or ask the user a question about whether they actually want to submit, and/or present some options for the user to choose from. The hooks can be defined either in the frontend or backend.

BitmapIcon

Local Navigation

Description

This functional component shows a picture or icon. It's main purpose is for rendering images inside configured forms. If you want to display images in another context, consider using *components/SVGIcon* (page 72) or cs-web-components-base-components.Icon.__module__ instead.

React Properties

Property	Type	Default	Use
src	string	•	URL of the picture or icon to be shown

CheckBox

Local Navigation

- *Description* (page 93)

Description

A `CheckBox` can be set in 2 states: `checked` or `unchecked`. It has one more property in addition to those defined in form_control_common_props:

Property	Type	Default	Use
checked	bool	•	Display the checkbox as checked

formcontrols/CheckBoxMenuItem

Local Navigation

- *Description* (page 93)

Description

A `CheckBoxMenuItem` can be set in 2 states: `checked` or `unchecked`. It has one more property in addition to those defined in form_control_common_props:

Property	Type	Default	Use
checked	bool	•	Display the checkbox as checked

The CheckBox's `name` property is also used as the MenuItem's `eventKey`.

Interactivity is handled somewhat different from regular checkboxes because of the MenuItem wrapper:

- The `checked` state is controlled by properties only, there is no internal state
- Thus, you have to provide the required `onChange` property, which is used as the MenuItem's

Email

Local Navigation

- *Description* (page 94)

Description

This component can be used to show an email address. It provides also the action for sending email to that address via `mailto:` protocol. It takes such properties just as the <*TextInput* (page 199)>.

formcontrols/FormActions

Local Navigation

- *Description* (page 94)

Description

This component shows a list of buttons in proper layout suits a form:

Property	Type	Default	Use
buttons	Immutable.List	•	List of definitions for buttons, contains properties for each button: at least `label` for the button label and `onClick` event handler

FormControl

Local Navigation

- *Description* (page 94)

Description

This module provides several components that are based on the different `<input/>` components that HTML5 provides:

- *Button* (page 190)
- *BitmapIcon* (page 92)
- *CheckBox* (page 93)
- *ComboBox* (page 192)
- *ComboBoxCatalog* (page 193)

- *Calendar* (page 190)
- *Email* (page 94)
- *NumericEdit* (page 195)
- *Password* (page 95)
- *Radio* (page 195)
- *TextArea* (page 198)
- *TextInput* (page 199)

Password

Local Navigation

- *Description* (page 95)

Description

This component can be used to enter password. The user inputs would be masked. It takes such properties just as the <*TextInput* (page 199)>.

formcontrols/WithHyperlink

Local Navigation

- *Description* (page 95)

Description

This component enhances another <*TextInput* (page 199)>-based form control, so that it will be possible to open a hyperlink from there. The extended component has following additional properties:

Property	Type	Default	Use
targetLink	string	•	The link of the target, usually an URL
targetWindow	string	_blank	Where to open the `targetLink`, compare the `target` attribute on `<a>` tag
onNavigateLink	func	•	If given, this function will be called with `targetLink` as argument to navigate, instead of opening the link automatically. `targetWindow` is ignored.

Example:

```
import {TextInput, WithHyperlink} from 'cs-web-components-base';
// the new extended component
const LinkedTextInput = WithHyperlink(TextInput);
...
// in render method of some component
render() {
    return (
        <LinkedTextInput label="Some Field"
                         value={this.state.someText}
                         targetLink={this.state.someURL} />);
}
```

formcontrols/WithTypeAhead

Local Navigation

- *Description* (page 96)

Description

This component enhances another <*TextInput* (page 199)>-based form control with `type ahead` feature. It will open a drop down list to show possible options during typing. The extended component has following additional properties:

Property	Type	Default	Use
onSelectAt	func	•	A callback fired when an option from the drop down is selected, the index of the option is passed in
onTAStepAt	func	•	A callback fired when an option from the drop down is navigated by pressing arrow keys, the index of the option is passed in
onTACancel	func	•	A callback fired when the user close the drop down by pressing "Escape" key
value	any	•	Data to be displayed
matched	Immutable.List	•	A list of options, which match the typed data
hasMoreMatches	bool	•	Indicates if list of matches was truncated
TypeAheadItemRenderer	component	•	Renderer for the options in the drop down list
forceOpenMatchesOnFocus	bool	•	Dropdown opens when input receives focus.

Example:

```
import React from 'react';
import {FormControl} from 'cs-web-components-base';
import Immutable from 'immutable';

const testData = Immutable.List([
    'Bayern',
    'Berlin',
    'Brandenburg',
    'Bremen',
    'Sachsen',
    'Sachsen-Anhalt'
]);

function filterTestData(value) {
    const lowered = value.toLowerCase();
    return testData.filter(item => lowered !== '' && item.toLowerCase().
↪indexOf(lowered) == 0);
}

const TypeAheadText = FormControl.WithTypeAhead(FormControl.TextInput);

// can be used in render() of some other component as: <StaticTypeAhead />
class StaticTypeAhead extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            text: '',
            matches: Immutable.List()
        };
        this.onTextChange = this.onTextChange.bind(this);
        this.onSelectAt = this.onSelectAt.bind(this);
    }

    onTextChange(event) {
        const value = event.target.value;
        this.setState({
            text: value,
            matches: filterTestData(value)
        });
    }

    onSelectAt(idx) {
        const hit = this.state.matches.get(idx);
        this.setState({
            text: hit,
            matches: filterTestData(hit)
        });
    }

    render() {
        return (
            <div>
                <label className="text-muted">type 'b' or 's'</label>
                <TypeAheadText
                    value={this.state.text}
                    matched={this.state.matches}
                    onChange={this.onTextChange}
                    onSelectAt={this.onSelectAt}/>
            </div>
        );
    }
```

```
};
```

helpers

Local Navigation

Description

Various helper functions.

Contents

Functions

compose

```
compose(base: ReactComponent, hocs: array): ReactComponent
```

Takes:

- *base*: A component to which the HOCs are applied
- ***hocs*: A list of HOCs which are applied to the** component

Create a component by applying a list of HOCs to a provided base component.

Signature:

```
compose(
  base: ReactComponent,
  ...hocs: (ReactComponent => ReactComponent)
): ReactComponent
```

Usage Example:

```
const AsHeading =
  Component => props => <h1><Component ...props /></h1>;
const MyComponent =
  props => props.title;
const MyHeadingComponent = compose(
  MyComponent,
  AsHeading
);
```

contains

```
contains(container: Node, component: Node)
```

Takes:

- *container*: The node which may be containing `component`.
- *component*: The node which may be contained by `container`.

If container may be document, use this instead of `Node.contains` to ensure compatibility with Internet Explorer.

See also: https://developer.mozilla.org/en-US/docs/Web/API/Document#Internet_Explorer_notes

ensureLinkProtocol

For links that are stored in an attribute somewhere, relative paths don't make much sense; these cases mean most of the time that the "http" part was forgotten. This function checks the input parameter to see if it is either an absolute path, or contains a protocol part. If not, "http" is prepended.

getAppSetup

Return application setup values(window.appSetup) as an immutable object.

getIcon

```
getIcon(iconID: string, iconParams: object)
```

Takes:

- *iconID*: The icon ID from the backend
- *iconParams*: Optional variables used by the icon configuration

Return the URL to retrieve a configured icon by its ID from the backend. The icon configuration may contain variables, in this case the suitable values must be given also.

getLocalPathForURL

Helper function to calculate the route path for given URL. If the possible path is found, it can be use for e.g. in-page navigation(client side routing). Otherwise `undefined` will be return.

getObjectImage

Returns image file associated with object.

getPlugin

```
getPlugin(context: string, predicate: callable, fallback: ReactComponent):␣
↪ReactComponent
```

Takes:

- *context*: The plugin ID that is to be searched

- *predicate*: A callable to filter the entries
- *fallback*: Component to return if no match is found

Returns: the plugin component to use, or undefined

Return the first plugin component in the pluginConfiguration for `context` that is matched by predicate. Returns `fallback` if no match is found. predicate must be a function: predicate(value, index, iterable) -> bool

Usage example:

```
getPlugin('content-view', re => contentType.search(new RegExp(re, 'i')) !== -1);
```

isCEDesktop

Return whether the current application is running in Contact Element Desktop.

isSupportedImageType

Return whether the image content type is currently supported.

isSupportedVideoType

Return whether the media content type is currently supported.

makeCancelable

```
makeCancelable(promise: Promise): object
```

Helper function to make a promise cancelable. For reasoning and usage see https://github.com/facebook/react/issues/5465#issuecomment-157888325

NOTE: In a promise chain, make the LAST promise cancelable! Otherwise a promise may resolve even when already cancelled (see E045116).

makeSequence

```
makeSequence(callables: array): thenable
```

Use promise chaining to issue the upload calls sequentially, see http://www.html5rocks.com/en/tutorials/es6/promises/#toc-creating-sequences

Usage is like this:

```
const callables =
  someArray.map(item => () => doSomething(item, [other args]));
const promise =
  makeSequence(callables).then(...);
```

parseQuery

Parse query parameters from search string of a location object

prefixNS

```
prefixNS(name: string): string
```

`componentNameSpace` should be a constant defined in your `webpack.config.js`. This function creates a namespaced name based on this constant. You should use this for all components you register in the *registry* (page 128), as well as your CSS classes.

toDisplayFormat

Helper function to personalize a value.

updateQuery

update query parameters on a location object

Constants

Console: Wrapper around environments console object

key-handlers

Local Navigation

Description

This module allows shortcuts to be registered on a page. These shortcuts may either be local to an element or global, i.e. they can be triggered from the whole page (except for blacklisted elements).

Keyboard shortcuts are made up of a sequence of a sequence of 1 to 5 keychords, where each keychord is a combination of several modifiers (supported modifiers are `Control`, `Alt` and `Shift`) and a keyname, that is either a printable character, or one of the supported special keys. They may be described using descriptive strings like `"Alt+escape k"`.

Shortcut Syntax

The syntax for defining shortcuts is:

```
shortcut = (<keychord> ' ')* <keychord>
keychord = (<modifier> '+')* <keyname>
modifier = ('Control' | 'Alt' | 'Shift')
```

Handlers

Shortcut Handler functions may accept one parameter, which is the DOM Event that triggered the shortcut.

Example

The following Example shows a component that handles a list of shortcuts:

```
import React from 'react';
import ReactDOM from 'react-dom';
import {
  addGlobalShortcut,
  addLocalShortcuts,
  removeAllLocalShortcuts,
} from 'cs-web-components-base';

class ShortcutExample extends React.Component {
  constructor(props) {
    super(props);
    this.focus = this.focus.bind(this);
  }

  componentDidMount() {
    this._element = ReactDOM.findDOMNode(this);
    addGlobalShortcut('Ctrl+x Ctrl+e', this.focus);
    addLocalShortcuts(
      this._element,
      [
        ['Escape', this.cancel],
        ['Enter', this.submit],
      ]
    );
  }

  componentWillUnmount() {
    removeAllLocalShortcuts(this._element);
    removeGlobalShortcut('Ctrl+x Ctrl+e');
    this._element = null;
  }

  focus() { this._element.focus(); }
  submit() { ... }
  cancel() { ... }
}
```

Contents

Functions

addGlobalShortcut

```
addGlobalShortcut(shortcut, handler)
```

Takes:

- *shortcut*: The serialized form of the shortcut.
- *handler*: The handler function to be triggered.

Add a global shortcut to the page's keymap.

addGlobalShortcuts

```
addGlobalShortcuts(shortcutList)
```

Takes:

- *shortcutList*: A List of pairs `[shortcut, handler]` to be added.

Add a list of global shortcuts to the page's keymap.

addLocalShortcut

```
addLocalShortcut(component, shortcut, handler)
```

Takes:

- *component*: The HTMLElement on which the shortcut should be defined.
- *shortcut*: A serialized representation of the shortcut.
- *handler*: A function to be invoked, when the shortcut is triggered.

Add a local shortcut to HTMLElement component.

addLocalShortcuts

```
addLocalShortcuts(component, shortcutList)
```

Takes:

- *component*: The HTMLElement on which the shortcuts should be defined.
- *shortcutList*: An array consisting of pairs `[shortcut, handler]`.

Add a list of local shortcuts to HTMLElement component.

removeAllLocalShortcuts

```
removeAllLocalShortcuts(component, shortcuts)
```

Takes:

- *component*: The HTMLElement from which to remove the shortcut.
- *shortcuts*: An array of shortcuts serialized as strings.

Remove all shortcuts defined locally for HTMLElement component.

removeGlobalShortcut

```
removeGlobalShortcut(shortcut)
```

Takes:

- *shortcut*: The serialized form of the shortcut.

Remove a globally defined shortcut by its serialized form from the page's keymap.

removeGlobalShortcuts

```
removeGlobalShortcuts(shortcutList)
```

Takes:

- *shortcutList*: An array consisting of pairs `[shortcut, handler]`.

Remove a list of global shortcuts to the page's keymap.

removeLocalShortcut

```
removeLocalShortcut(component, shortcut)
```

Takes:

- *component*: The HTMLElement from which to remove the shortcut.
- *shortcut*: The serialized shortcut representation

Remove a shortcut defined locally on an HTMLElement.

removeLocalShortcuts

```
removeLocalShortcuts(component, shortcuts)
```

Takes:

- *component*: The HTMLElement from which to remove the shortcut.
- *shortcuts*: An array of shortcuts serialized as strings.

Remove an array of shortcuts defined locally on an HTMLElement component.

layouts/SimpleLayout

Local Navigation

- *Description* (page 104)

Description

Trivial layout, that simply renders all children inside a div with an optional `className` prop.

layouts/SplitLayout

Local Navigation

- *Description* (page 105)

Description

Layout that renders two children side by side vertically or horizontally. It bases on React Split Pane, for details please visit https://github.com/tomkp/react-split-pane

Tab

Local Navigation

Description

Components for creating a tab layout. A tab's titles are displayed in Titles within the Bar. A tab's content is rendered within Panes within the Content container. Panes and corresponding Titles are linked via the `eventKey` parameter. Bar and Content are again wrapped in the Container component.

Example of use:

```
<Tab.Container>
    <Tab.Bar
        onSelectFromMenu={myOnSelect}>
        <Tab.Title title="First Tab" eventKey="A"/>
        <Tab.Title title="Second Tab" eventKey="B"/>
        ...
    </Tab.Bar>
    <Tab.Content>
        <Tab.Pane eventKey="A">
            // render tab content here
        </Tab.Pane>
        <Tab.Pane eventKey="B">
            // render tab content here
        </Tan.Pane>
        ...
    </Tab.Content>
</Tab.Container>
```

Contents

Classes

Functions

Container

Wraps the Bar and Content components

Name	Type	Default	Description
activeKey	string/number	-	Event key of the currently active tab.
className	string	-	Optional style class for custom layouting.

Content

Wraps the Pane components.

Pane

Renders the content of one tab page.

Name	Type	Default	Description
eventKey	string	-	Unique key to associate pane with the bar entry.

Constants

EVENT_KEY_ADD: Constant for the add new tab control. Can be used used in the `onSelect` callBacks passed to Container and Bar. See also the `addControlLabel` property of Bar.

Example of use:

```
function myOnSelect(key) {
    if (key === Tab.EVENT_KEY_ADD) {
        // create a new tab pane and title
    }
    ...
}
```

TabSource: Renders a tab in the Bar with the label passed in `title`. The icon is optional. Additonal controls such as a tab toolbar can be passed as children.

Name	Type	Default	Description
title	string	-	Label for the tab.
eventKey	string/number	-	Unique identifier for associating with correspondiog Pane.
iconName	string	-	Identifier of the icon.
iconSrc	string	-	Url of the icon.
onChange	func	-	If passed, causes label to render as input. On changing form callback is called.
disabled	bool	-	Disables tab.

Actions

Local Navigation

Description

This module provides action creators to post messages. All post functions defined in this module receive a javascript object as argument, specifying the general attributes of the message, as well as an unspecified number of displayInfos, that determine how the message should be displayed to the user.

The general properties object may contain the following fields:

- `level`: One of the Level constants defined in *Messages* (page 108)
- `title`: A string that will be displayed as the title of the message.
- `owner`: The component that posted this message.
- `message`: A string containing the message that will be displayed to the user.

Display Infos are generated using special factory functions, such as *createBanner* (page 110) or *createNotification* (page 110)

On dispatching, these action creators return the unique id assigned to the message, which can be used to further interact with the message. The possible interactions depend on the display infos used.

For an example on how to use these action creators, see *Messages* (page 108)

Contents

Functions

postError

A wrapper around *postMessage* (page 107) that sets `Level.ERROR` as the default level.

postInfo

A wrapper around *postMessage* (page 107) that sets `Level.INFO` as the default level.

postMessage

```
postMessage(attributes: object, displayInfos: displayInfos): string
```

Takes:

- *attributes*: The general attributes of this message
- *displayInfos*: spread parameter that specifies how this message is displayed

Returns: the id of the message

Post a message to the user. `displayInfos` is a

postWarning

A wrapper around *postMessage* (page 107) that sets `Level.WARNING` as the default level.

removeMessageFromDisplay

```
removeMessageFromDisplay(messageId: string, displayId: string)
```

Takes:

- *messageId*: The message to be removed.
- *displayId*: The display adapter from which to remove the message.

Remove the message identified by `messageId` from the display adapter identified by `displayId`.

updateDisplayInfo

```
updateDisplayInfo(messageId: string, displayId: string, displayInfo: object)
```

Takes:

- *messageId*: The message to be updated.
- *displayId*: The display adapter to be updated.
- *displayInfo*: The new attribute values.

Update the display info of the message identified by `messageId` for the display adapter identified by `displayId` with the attributes in `displayInfo`.

Messages

Local Navigation

Description

This module provides the basic API to display messages to the user. The module provides actions and components to display messages as notifications or banners, and can be easily extended to provide custom methods of display.

In order to post a message to the user, use the `Actions.post*` action creators described in *Actions* (page 106).

Example: Displaying a message

```
import {Messages, Button} from 'cs-web-base-components';
import {connect} from 'cs-web-components-externals';
import React from 'react';

class Example extends React.Component {
  constructor(props) {
    super(props);
    this.displayBanner = this.displayBanner.bind(this);
  }

  displayBanner() {
    this.props.postMessage(
      {
        level: Messages.Level.INFO,
        title: "banner-message",
        message: "I'm a banner-message!",
        owner: this,
        global: true
      },
      Messages.createBanner({bannerContainerId: 'my-banner-container'}),
      Messages.createNotification({timeout: 5000})
    );
  }

  render() {
    return (
      <div>
        <Messages.BannerContainer id="my-banner-container" />
        <Button.TextButton label="Display Banner" onClick={this.displayBanner}>
      </div>
    );
  }
}

const ConnectedExample = connect(
  undefined,
  {postMessage: Messages.Actions.postMessage}
)(Example);
```

Contents

Modules

Classes

Functions

createBanner

```
createBanner(arguments: object): object
```

Takes:

- *arguments*: Spread Parameter: `{bannerContainerId}`

Returns: a displayInfo object

Display message inside the banner container specified by `bannerContainerId`

createNotification

Display a message as a notification in the top right corner of the viewport. The first parameter is a spread argument, that may receive the `timeout` of the notification.

removeNotification

Remove the notification for the message identified by `messageId`.

Constants

Level: The level of the message. Possible values are:

- `EMPTY`
- `PROGRESS`
- `INFO`
- `SUCCESS`
- `WARNING`
- `ERROR`

Handlers

Local Navigation

Description

This module provides default submit handler actions. For using these handlers, see also: *Handling results* (page 116). For an example on implementing a custom handler and invoking default handlers from it, see *operations/sagas/handlers* (page 118).

Contents

Functions

addObjectToHistory

A Handler for successful operation execution.

If an object is included in the submit-response, its detail page address will be added to the system's history.

executeFailureHandlerFns

Handle legacy failure handler functions.

This will emulate the behaviour of failure handler functions as used in cs.web 15.2.0 and cs.web 15.2.1. Please note that full compatibility can not be ensured, and should be tested when porting to cs.web 15.3.0.

executeSuccessHandlerFns

Handle legacy success handler functions.

This will emulate the behaviour of submit handler functions as used in cs.web 15.2.0 and cs.web 15.2.1. Please note that full compatibility can not be ensured, and should be tested when porting to cs.web 15.3.0.

handleOperationSuccess

Default handler for successful operation execution.

Inspects the `result_type` field of the operation result and dispatches an action accordingly:

- `NO_TYPE`: No action is taken.
- `OBJECT`: The received object is dispatched to the Redux Store.
- `HTML_VIEW`: The browser navigates to the url provided in the operation result by setting `window.location`.

Other available result types are currently not supported and will result in a warning being issued.

navigate

```
navigate(path, options)
```

Takes:

- *path*: The location to navigate to
- *options*: Configuration object

A Handler for successful operation execution.

Navigate to the location specified by path. If path is undefined, try to use the field web_ui_link of the result.

If option pushHistory is set, this handler will use `browserHistory.push` instead of setting `window.location`.

postError

A Handler for failed operation execution.

Displays an error message if the operation fails.

updateObjectStore

A Handler for successful operation execution.

If an object is included in the submit-response, it will be updated in/added to the object store.

operations/actions/operations

Local Navigation

Description

This module provides the Redux action creators that are the interface for executing operations, and sending changes to the form state. Usually you will dispatch the *runOperation* (page 115) action by invoking the corresponding method from either your *OperationTrigger* (page 89) component or your *OperationHandler* (page 86) component.

The other actions in this module control the communication between the saga running the operation and the form that the user interacts with, as well as dialog hooks. You only need to use those, if you are implementing your own *OperationHandler* (page 86) and do not rely on the ConfiguredForm component.

These actions can be divided by the direction of the communication:

- Actions sent from the form to the saga, usually in response to user input.
- Actions sent from the saga to modify the form state, which usually results in updates to the displayed form.

Warning: Besides *runOperation* (page 115) the actions provided by this module are considered internal, and may be subject to change without prior note.

Contents

Functions

addFiles

```
addFiles(instanceName: string, files: array)
```

Takes:

- *instanceName*: Handler for user interaction

- *files*: a list of file objects to be added to the current state.

Adds a list of files to the operation state. Dispatched by form.

cancelOperation

```
cancelOperation(instanceName: string, force: bool)
```

Takes:

- *instanceName*: Handler for user interaction
- *force*: if true, the user won't be queried.

User requested to cancel the operation during form execution. Dispatched by form, handled by saga.

By default this queries the user, if he wants to dispose of changes he has made to the form. Use parameter `force` to suppress this behaviour.

changeOperationValues

```
changeOperationValues(instanceName, values, unchecked)
```

Takes:

- *instanceName*: Handler for user interaction
- ***values*: An Immutable.Map containing the attribute-value pairs for** the form
- ***unchecked*: This flag is set by the form if the values have been** validated, for example when they are set by a catalogue.

Invoked by form when the user alters the value of one or more fields. This will invoke dialog hooks configured for the changed fields. Invoked by the form/operation-handler and handled by `changeFormSaga`.

clearErrors

```
clearErrors(instanceName: string)
```

Takes:

- *instanceName*: Handler for user interaction

Operation State contains no errors. This is sent by operations saga, and should remove outdated messages.

queryCancel

```
queryCancel(instanceName)
```

Takes:

- *instanceName*: Handler for user interaction

Ask the user if he wants to cancel, as this will cause data to be lost. This action is issued by the saga and handled by the store. The Form Component should respond with either `queryCancelForce` or `queryCancelBack`.

queryCancelBack

```
queryCancelBack(instanceName: string)
```

Takes:

- *instanceName*: Handler for user interaction

User notifies operation cancellation to be cancelled, i.e. the operation is continued. This action is sent by the form/operation-handler and handled by the saga in response to `queryCancel` being set.

queryCancelForce

```
queryCancelForce(instanceName)
```

Takes:

- *instanceName*: Handler for user interaction

User notifies operation to be cancelled even though values have been changed. This action is sent by the form/operation-handler and handled by the saga in response to `queryCancel` being set.

queryDismiss

```
queryDismiss(instanceName: string)
```

Takes:

- *instanceName*: Handler for user interaction

Dismiss a dialog from dialog_hook after a result has been dispatched. This action is sent by `queryUserSaga` to form.

queryResultBack

Return back to form from dialog_hook dialog. This is sent by the form and handlded by `queryUserSaga`.

queryResultCall

Call back to server from dialog_hook dialog. This is sent by the form and handlded by `queryUserSaga`.

queryResultCancel

```
queryResultCancel(instanceName: string)
```

Takes:

- *instanceName*: Handler for user interaction

Cancel operation-handler from dialog_hook dialog. This is sent by the form and handled by `queryUserSaga`.

queryResultSubmit

```
queryResultSubmit(instanceName: string, attribute: object)
```

Takes:

- *instanceName*: Handler for user interaction
- *attribute*: entries `name` and `value` specify an attribute to be updated by the dialog hook.

Submit operation from dialog_hook dialog. This is sent by the form and handlded by `queryUserSaga`.

queryUser

```
queryUser(instanceName: string, dialog: FrontendDialog)
```

Takes:

- *instanceName*: Handler for user interaction
- *dialog*: The Dialog definition

Display a dialog from dialog_hook to the user. This action is handled by the form. See also `queryUserSaga`.

removeFiles

```
removeFiles(instanceName: string, files: array)
```

Takes:

- *instanceName*: Handler for user interaction
- *files*: a list of file objects to be removed from the current state.

Remove a list of files to the operation state. Dispatched by form.

runOperation

Execute the operation identified by operationInfo. This function takes two positional paramaters:

- instanceName, which is usually filled in by an OperationTrigger component.
- parameters, an object detailing which operation to execute and how to execute it.

Operation Parameters

The `parameters` object itself, may contain the following fields if `runOperation` is called from OperationTrigger or OperationHandler:

Name	Type	Description
contextObjects	Immutable.List	A List of objects to operate on. Required for operations that act on objects.
parameters	Immutable.Map	These parameters are passed as query parameters to the submit operation call. Wether the operation requires any of these depends on the operation implementation.
resultInNewTab	boolean	If set, the result will be opened in a new tab. Should only be set, if the result is a URL.
successActions	array	A list of actions to be dispatched if operation is successfully executed.
failedActions	array	A list of actions to be dispatched if operation execution fails.
generalActions	array	A list of actions to be dispatched regardless of wether the operation fails or completes successfully.
dialog	string	Specify an alternate dialog to the one configured in the operation's operation configuration.
suppressHandler	boolean	Suppress OperationHandler execution, even though a dialog is configured
forceHandler	boolean	Force OperationHandler execution, even though no dialog is configured.

If you're dispatching the operation directly, the following parameter needs also be provided:

Name	Type	Description
operationInfo	Immutable.Map	An object describing the operation.

Operation Instances

instanceName is used to identify this operation execution in various contexts, such as form display and window handling (for tab navigation). This parameter is usually set when runOperation is invoked by *OperationTrigger* (page 89).

If you are planning on using this action without relying on the *OperationHandler* (page 86). component, you should ensure that instanceName gets a unique value. Furthermore you need to take care of (de-)initializing a state for this instance, by calling `initFormInstance` and `clearFormInstance`.

Operations on objects and types

If the operation is run on a set of objects (which is determined by the field `activation_mode` in `operationInfo`), you should set the parameter contextObjects to an Immutable.List containing the objects, the operation acts on. Operations that operate on Classes, such as CDB_Search or CDB_Create use the type that is part of the operationInfo object.

Additional Parameters

Some operations require additional arguments in addition to the objects they operate on, e.g., CDB_Create may be triggered with the UUID of a template. These additional arguments should be put into field parameters.

Handling results

The results of running an operation can be handled by providing a list of actions for each result type. These must be plain Redux Actions, as the operation saga modifies the payload to include the operation result in the field

action.payload.operation, which will be identified as the `operation field` in the rest of the section.

In all three cases, the operation field contains the following values:

- **info**: identifies the operation that was executed

The following parameters allow to handle the three available result types:

- **successActions**: A list of actions that will be dispatched on successfull completion of an operation. The operation field will contain the field result, which contains the result of the operation, sent from the server
- **failedActions**: A list of actions that will be dispatched when the operation fails. The operation field error will contain the error that was returned by the server.
- **cancelActions**: A list of actions that will be dispatched when the operation is cancelled. No further information will be added to the operation field.
- **generalActions**: These actions will be run, in any case (wether the operation was submitted, failed, or cancelled).

Default Handlers are provided by module *Handlers* (page 110). For an example on how to implement your own custom handler, see *dispatchResultHandler* (page 118).

Opening tabs

If the operation should display its result in a new tab, you need to set the field `resultInNewTab`. This field may take on two possible values, depending on wether the operation displays a form or not:

- `OPEN_ON_SUBMIT`: If the call to submit will be the last user interaction.
- `OPEN_ON_RUN`: If no form is displayed.

Display configured forms

Displaying a form and interacting with the user is done by an OperationHandler. These handlers are usually only invoked if a configured form is available, as indicated by the operationInfo object. The operation will then be run with the default attribute values defined in the operationInfo object.

If you want to for some reason override this default behaviour, the field `suppressHandler` – if set – will suppress the handler invocation, even though a form is configured, while the field forceHandler will invoke a handler, even though no form is configured. The latter allows the API user to invoke their own front-end based form.

Overriding the operations default form

Usually, the operation saga will display the form configured for the operation to the user. If an alternative configured form should be displayed to the user, the form name may be passed to runOperation in field dialog.

Non-Blocking Paramater

If set, the running operation instance will not block other OperationTriggers. May be used for operations which do not modify data, such as CDB_ShowObject.

A typical example is the form component.

submitGeneralError

```
submitGeneralError(instanceName: string, errors)
```

Takes:

- *instanceName*: Handler for user interaction
- *errors*: An Immutable.List containing error messages

A general error occured during operation execution. This is sent by operations saga, and should be displayed to the user by form.

submitOperation

```
submitOperation(instanceName: string)
```

Takes:

- *instanceName*: Handler for user interaction

User requested to submit the operation during form execution. Dispatched by form, handled by saga.

submitValueError

```
submitValueError(instanceName: string, fields: Immutable.List.<Immutable.Map>)
```

Takes:

- *instanceName*: Handler for user interaction
- *fields*: an Immutable.List containing Immutable.Map objects. Each map has a `field` and a `type` field, specifying the type of error that occured.

Operation State contains invalid values. This is sent by operations saga, and should be displayed to the user by form.

operations/sagas/handlers

Local Navigation

Description

Contents

Functions

dispatchResultHandler

```
dispatchResultHandler(handler, action)
```

Takes:

- *handler*: The action creator to be dispatched.

- *action*: The action with which the handler saga was invoked.

Invoke a handler action from a handler saga.

This is useful, if you, e.g., want to invoke another handler from your own custom handler saga.

Example:

```
import {ReduxSaga} from 'cs-web-components-externals';
import Operations from '../operations';
import {Registry} from '../registry.js';

const {takeEvery} = ReduxSaga.effects;

const CUSTOM_HANDLER_ACTION = "CUSTOM_HANDLER_ACTION";

function *myCustomHandlerSaga(action) {
   const {dispatchResultHandler, Handlers} = Operations;
   if (resultNeedsSpecialHandling(action)) {
      // Your handler code here ...
   } else {
      // Invoke default handler
      yield call(
        dispatchResultHandler,
        Handlers.updateObjectStore,
        action
      );
   }
}

function *myCustomHandlerWatcher() {
   yield takeEvery(CUSTOM_HANDLER_ACTION, myCustomHandlerSaga);
}

Registry.registerSaga(myCustomHandlerWatcher);
```

The custom handler in the example above to handle a result in a custom way, if `resultNeedsSpecialHandling` evaluates to true, else it updates the object in the store.

operations/sagas/operations

Local Navigation

Description

This module implements the operation execution logic in the front end.

It's main entry point is runOperationSaga. The main steps are implemented as sagas themselves:

1. `startOperation`: Get Operation State and Form data from backend and initialize frontend state
2. `execOperation`: Handle form interaction, exec presubmit and submit stage
3. Handle result data by dispatching actions for success, cancellation, and error.

Contents

Functions

execFormSaga

Display a form and wait for either submit or cancel. CHANGE_OPERATION_VALUES events are handled in `form.js:changeFormSaga`.

execOperationSaga

Runs in a loop executing the frontend logic for the operation. Each iteration, the saga

1. waits till the user submits or cancels the form. This may either cancel or continue the operation or reinitiate form interation (See `getOperationValues` saga).
2. executes the presubmit stage (containing form validation, dialog hooks and wizard logic) This may either cancel or continue the operation or reinitiate form interation (See `execPresubmit` saga).
3. finally submit the operation state to the backend for operation execution. This may either return to the form (when asynchronuous form checks are still pending), signal an error or return a result. (See `execSubmit` saga).

execPresubmitSaga

Executes Presubmit Stage. Returns an object that is analyzed for control flow. This may either return to form execution stage, yield an error, cancel the operation, or cause execOperation to continue with submit stage.

1. Check form for missing values for mandatory fields. If this fails the user is notified, and form execution continues.
2. Call presubmit hooks and evaluate results:
   - Dialog Hooks yield errors
   - Dialog Hooks query the user
   - Wizard Logic: Replace form data in state with return values from dialog hook.
3. Wizard Logic: Retrieve a new form from backend and replace form data in state.

queryUserSaga

Display a FrontendDialog to the user, and wait for one of the possible result.

When running submitOperation, the Operations presubmit DialogHooks will be run. If a dialog is configured, it will be displayed to the user by this saga by invoking the queryUser action.

The dialog may update an arbitrary attribute value in the operation state, and continue the operation.

The dialog component must dispatch one of the following actions to continue operation execution:

- queryResultBack: Update attribute, go back to form
- queryResultCall: Update attribute, invoke dialog hooks again
- queryResultCancel: Cancel operation
- queryResultSubmit: Update attribute, submit operation

runHybridOperation

If targetUrl is set the operation will be run in legacy mode, by navigating to the provided CDB-URL.

runOperationSaga

Top-level entry point for running operations.

This initializes and executes an operation, and dispatches the provided result handlers.

See operations/actions/operations.js for a description of the structure of payload.

selectOperationAction

Higher Order Selector for Actions on Operation Instances.

This Higher Order Function takes an instanceName and creates a selector that will select an action based on the provided instanceName and actionType.

startOperationSaga

```
startOperationSaga(runOperation)
```

Takes:

- *runOperation*: payload.

Retrieve the initial operation_state and mask configuration from backend, if available, and initializes the Redux operation-state.

submitOperationSaga

```
submitOperationSaga(payload, newState)
```

Takes:

- *payload*: the payload provided to runOperation.
- *newState*: the current operation state.

Submit a running operation.

Overlay-Constants

Local Navigation

Description

Constants for use with overlays.

These are exported as fields of *Overlays* (page 124). Use them like:

```
import {Overlays} from 'cs-web-components-base';

const {HorizontalAlignment} = Overlays;
```

Contents

Functions

Constants

HorizontalAlignment: An enumeration for specifying horizontal alignment for aligned/layouted overlays. Values are

- `HorizontalAlignment.LEFT`
- `HorizontalAlignment.RIGHT`

HorizontalAlignmentType: PropType for HorizontalAlignment property. **VerticalAlignment**: An enumeration for specifying vertical alignment for aligned/layouted overlays. Values are

- `VerticalAlignment.BOTTOM`
- `VerticalAlignment.TOP`

VerticalAlignmentType: PropType for VerticalAlignment property.

overlays/Controlled

Local Navigation

Description

This module provides a HOC that provides functionality to display and hide overlays as a result of user interaction.

The main point of this HOC is the `onHide` callback. This will allow you to close the overlay according to user's expectations:

- Either by pressing the `Escape` key, when the overlay is focused
- or by focusing another element.

Note that the actual visibility state has to be stored by the user, so you are able to handle the `onHide` callback to your liking.

Removed React Properties

- onBlur: Used for `onHide`.

Added React Properties

Name	Type	Description
grabFocus	bool, default `false`	Should focus be set to the overlay on show
onHide	function	Called when the overlay should be hidden
hideOnScroll	boolean	If set, `onHide` will be invoked when scrolling happens in a container.

Warning: This component is not considered part of the supported cs.web API. This documentation is solely provided for informational purposes and the documented interfaces may be subject to change without prior notification.

overlays/DropdownButton

Local Navigation

Description

This component renders a button that displays a *Dropdown* (page 206) containing a menu if clicked.

Provided components are:

- DropdownIconButton
- DropdownIconTextButton
- DropdownTextButton

Use the following menu child components provided by this module to display menu content:

- *MenuItem* (page 209)
- *MenuHeader* (page 209)
- *MenuDivider* (page 209)

React Properties

Name		Description
buttonStyle	string	Semantic Style of the Button Component.
verticalAlign	VerticalAlignment-Type	How the Menu should be aligned in relation to the button.
horizontalAlign	HorizontalAlignment-Type	How the Menu should be aligned horizontally.
containerRef	object	Reference to a containing component that may cover the anchor component.
size	string	Size of the string to be displayed.

Example

```
import {Overlays} from 'cs-web-components-base';

const {
  HorizontalAlignment,
  VerticalAlignment,
  DropdownIconButton,
  MenuItem,
  MenuDivider,
} = Overlays;

function MyMenu(props) {
  return (
    <DropdownIconButton
         buttonStyle="info"
         iconName={"csweb_option_horizontal"}
         horizontalAlign={HorizontalAlignment.LEFT}
         verticalAlign={VerticalAlignment.TOP}
         title={"Hello World"}
         size={'sm'}
         {...props}>

      <MenuItem onSelect={() => {...}}>Hello World</MenuItem>
      <MenuItem>Hello World</MenuItem>
      <MenuDivider />
      <MenuItem>Hello World</MenuItem>
    </DropdownIconButton>
  );
}
```

Overlays

Local Navigation

Description

The `Overlays` module provides implementations of typical overlay-based components, base components to implement own overlay components, as well as Helper components that realize features that are usually associated with overlays, such as Menus.

These components implement overlay functionality using React Portals. The resulting components render their content in an overlay div that is absolutely positioned under the documents body tag.

Ready to use

Components that can be used out of the box are:

- *Bubble* (page 205): A Speech Bubble to display information associated with an element on the site
- *Dropdown* (page 206): A general component for rendering dropdowns.
- cs-web-components-base-overlays.DropdownButton.__default__: Components for displaying a button that displays a dropdown menu if clicked.
- *TypeAhead* (page 211): Dropdown that has been modified to be used as a Dropdown for type-ahead functionality.
- *ContextMenu* (page 206): An overlay component to display context menus.

Roll your own

The module also provides base components to implement your own overlay based components. Note that to keep a consistent style you should rather try to implement your ideas, using the components described above.

Base components provided by the `Overlays` module are the following:

- *Overlay* (page 210): This provides a basic overlay implementation, without any layouting functionality.
- *AlignedOverlay* (page 203): Extended Overlay that handles layouting.

There also exist extended version of these components that extend the base components for managing user-initiated close events:

- Overlay. *Controlled* (page 211): Overlay, extended for closing the overlay by user interaction.
- AlignedOverlay. *Controlled* (page 204): AlignedOverlay, extended for closing the overlay by user interaction.

These components set the focus on the overlay when it is opened and register a handler on its `blur` event, as well as for the keyboard shortcut `Escape`. The user of these components still needs to manage the visibility state of the component, but can modify it by providing an `onHide` callback.

Menus

The `Overlays` module provides a few components to help you implement dropdown menus:

- *Menu* (page 208): A container for MenuItems.
- *MenuItem* (page 209): Represents an entry in a menu.
- *MenuHeader* (page 209): A headline for use in menus.
- *MenuDivider* (page 209): A divider for use in Menus.

Constants for implementing own overlays and parametrizing existing ones are defined in *Overlay-Constants* (page 121).

Note: The components provided in this work in `cs.web`-based applications without major problems. However, the default overflow behaviour of the `document.body` of the application may cause minor issues when Dropdown-based components go into scrolling mode. This can be fixed by setting `app_setup[appSettings][renderFixedBody] = True` in your backend application. Please ensure that your application layout and logic are not affected by this.

reducers/fetching

Local Navigation

Description

This module provides meta entities, such as *errorsById* (page 126), that stores failed requests and *fetchingById* (page 126), which stores pending requests.

Contents

Functions

errorsById

Stores errors that occured during communication with the REST API by the REST URL used.

fetchingById

Stores the Promise associated with a pending requests by the request URL. Completed or failed requests will be removed from this store.

reducers/object-store

Local Navigation

Description

This module provides access to the global object store. The actions used to modify this store are defined in *object-actions* (page 63).

Firstly, this module contains reducers that store certain entities of the REST API, such as objects, types or relation.

Interaction of Object Stores And Meta Stores

Failure and pending operations on the object store are also handled using Redux. E.g., if an object is retrieved via *fetchObject* (page 64) given an url `url`, the Promise returned by the action will be stored in *fetchingById* (page 126). The presence of this entry represents a pending request for the given `url`. When the object retrieval either fails or completes successfully, the Promise is removed from *fetchingById* (page 126), and depending on the result of the operation an error message is put into *errorsById* (page 126), or the received object is put into *objectsById* (page 127). Finally the Promise associated with the request is resolved.

An example on how to connect to a store is given in *objectsById* (page 127)

Contents

Functions

objectsById

This store is an Immutable map that stores all objects fetched via REST API by their id.

Example

```
function ExampleComponent(props) {
  return <div>{props.myObject ? props.myObject.get('titel') : 'undefined'}</div>;
}

function mapStateToProps(state, ownProps) => {
  return {myObject: state.objectsById.get(ownProps.myObjectId)};
};

connect(mapStateStateToProps)(ExampleComponent);
```

relshipsByClass

Relationship meta data by classname

relshipsById

Stores objects related to an entity by a relation specified by an url.

typesById

This reducer stores type information retrieved for the object-type that can be retrieved under the url `object.get('@type')`.

registry

Local Navigation

- *Description* (page 128)

Description

This module provides the a global Registry object, that has functions to register and retrieve React components, redux reducers and frontend form exits.

table/column_aggregator/index

Local Navigation

- *Description* (page 128)

Description

table/column_dragger/index

Local Navigation

- *Description* (page 128)

Description

Allow drag the column header.

table/column_orderer/index

Local Navigation

- *Description* (page 128)

Description

table/column_resizer/index

Local Navigation

- *Description* (page 129)

Description

Allow to change column width.

table/column_search/index

Local Navigation

Description

Provides columns search functionality in a table.

Components

- Provider: Is required for column search
- ColumnActions: Adds a search input control (FormControl), if configured for the column, to the column header
- **ToolbarButton: Is required for column search.** Adds a button to show/hide column header search input controls Adds a button to excecute search if search input controls are visible

Properties

Property	Type	Default	Use
initURL	string	•	TODO: equal to initURL from ConfiguredForm
initValues	Immutable.Map	•	Values to initial fill search input controls
onSearchHeaderSubmit	func	•	Callback to start search operation

Remarks

TODO:

table/filterable/index

Local Navigation

Description

table/groupable/index

Local Navigation

Description

Table

Local Navigation

Description

The *Table* (page 130) module provides several table components with different feature sets as well as an API for defining custom table components.

This document explains the general interface to render tables, by providing column definition and row data. For details on how to configure custom table components see *table/Manager* (page 136).

Providing Column Information

Column Definition: The data layout is provided as an immutable list of objects, each of which represents a column in the table. Possible fields of a column object are (Required fields given italic):

- label: The label displayed in the table header for the column

- tooltip: The tooltip displayed in the table header for the column
- *id*: should be unique key used to identify the column in the DOM, and by various table features
- width: Sets a fixed width for this table column. Omit this field to have this column take the available space. Should be omitted for at least one column two allow the table to take all avaible space.
- contentRenderer: Allows to define a custom renderer component for this column.
- getFormattedValue: A function that gets the value and should return the user representation of the value.
- sortFunction: A function that compares to values of the column. The function will be called in the way `cmp(v1, v2)` where v1 and v2 are lists of two elements. The first element is the value to be compared. The second contains all entries of the row where the value resides.
- getLink: A function that will be called with the value, row and column as parameter and should return the url of the link.
- action: A function that will be called with the value, row and column of the focused cell. This function will be invoked, when the user presses `Enter`.

An example:

```
const columns = Immutable.fromJS(
  [
    {
      id: 'active',
      label: 'Active',
      width: 80
    },
    {
      id: 'name',
      label: 'Name'
    }
  ]
);
```

Column Ordering: The feature *table/column_orderer/index* (page 128) enables a table to be reordered. To give an initial column ordering, pass an immutable ordered set specifying column id as property `orderedColumns` to the table.

Cell Rendering: By default the data value that is provided to each cell (c.f. *Row Data* (page 132)) is directly rendered into the DOM.

If this behaviour is undesired, it is possible to specify a custom cell renderer for each column definition, by specifying a React Component on the field `contentRenderer`. As an example consider that a given column will contain a boolean value:

```
const columns = Immutable.fromJS(
  [
    {
      id: 'is_visible',
      label: 'Visible',
      contentRenderer: props =>
        <span style={{opacity: props.value ? 1 : 0}}>
          <Glyphicon  glyph="flag" />
        </span>
    },
    ...
  ]
);
```

The property object of the renderer contains the following fields:

- row: The row definition of the row currently rendered.
- column: The column definition of the column currently rendered.

- value: The entry in the row definition that is to be rendered in the current cell.

Note that this only applies to the default row rendering implementation and may vary if custom row renderers are defined (c.f. *Row Data* (page 132)).

Column Actions: A column definition may specify a special action by specifying a function for the field `action`. This function will be called with the currently selected `value`, `row` and `column` when the user presses `Enter` and a cell is selected.

An example:

```
function columnAction(value, row, column) {
    alert(`Selected value: ${value}`);
}

const columns = Immutable.fromJS(
  [
    // ...
    {
      id: 'name',
      label: 'Name',
      action: columnAction
    }
    // ...
  ]
);
```

If no column action is defined and the `getLink` property is set for the focused column, the default behaviour is to navigate to the link returned by `getLink`.

Row Data

A row definition is an immutable object that specifies an unique‘‘id‘‘ string, as well as a list of `columns`. Each column is an object, which will be passed to the cell renderer specified for the column. A simple example to display a list of javascript objects:

```
const raw_rows = [
  {
     is_visible: true,
     name: "name1"
  },
  {
     is_visible: false,
     name: "name2"
  },
  {
     is_visible: true,
     name: "name3"
  },
]

const rows = Immutable.fromJS(raw_rows.map(value, index) =>
  {
    id: `${index}`,
    columns: [
      value.get('is_visible'),
      value.get('name')
    ]
  }
);
```

If `rowAction` is passed to the table, this function will be invoked, when an action is triggered and no column is selected.

Custom Row Renderers: Custom Row renderers may be specified for each row individually to override the default row rendering behaviour. To specify a custom row renderer, specify the React component which should render the row in field Renderer. The renderer is passed the following props:

- row: The row data

Configurable Table Features

While the features described above are available to all types of tables, additional features may be configured for a table component, which may extend the properties required for the table, as well the column definitions and row data.

For info on which features are available in which preconfigured table component, see *Preconfigured Tables* (page 136). For a comprehensive list of available features, see *Features* (page 136).

Examples

```
import React from 'react';
import Immutable from 'immutable';
import { Table } from 'cs-web-components-base';

const columns = Immutable.fromJS([
            {id: 'attr1', label: 'First', width: 100},
            {id: 'attr2', label: 'Second', width: 100},
            {id: 'attr3', label: 'Third'},
            {id: 'attr4', label: 'Fourth'}
        ]);

const orderedColumns = Immutable.OrderedSet(['attr1', 'attr3', 'attr4', 'attr2']);

const rows = Immutable.List().withMutations(list => {
    const n = 100;
    for (let i = 1; i <= n; i++) {
        list.push(Immutable.Map({
            id: `row${i}`,
            columns: Immutable.List([
                i,
                Math.floor((Math.random() * n) + 1),
                Math.floor((Math.random() * n) + 1),
                Math.floor((Math.random() * n) + 1)
            ])
        }))
    }
});

class MenuExample extends React.Component {
    render() {
        return (
            <div onClick={()=>console.log('clicking')}>test menu</div>
        );
    }
};

const myMenu = {
    toolbarMenu: [MenuExample],
    settingPanels: [MenuExample]
};
```

```
const SimpleTable = Table.SimpleTable;

const ManagedTable = Table.Manager();

const SortedTable = Table.Manager({
    providers: [Table.sortable],
    columnActions: [Table.sortable]
});

const FilteredTable = Table.Manager({
    Table: Table.Fixed(Table.Table),
    providers: [Table.sortable, Table.filterable],
    columnActions: [Table.sortable],
    toolbarButtons: [Table.filterable, Table.SettingPanels],
    toolbarMenu: [myMenu],
    settingPanels: [myMenu, Table.columnHider]
});

const SelectTable = Table.Manager({
    providers: [Table.sortable, Table.filterable, Table.selectable],
    columnActions: [Table.sortable, Table.selectable],
    toolbarButtons: [Table.filterable]
});
const PagedTable = Table.Manager({
    providers: [Table.sortable, Table.filterable, Table.selectable, Table.
↪pagination],
    columnActions: [Table.sortable, Table.filterable, Table.selectable, Table.
↪pagination],
    toolbarButtons: [Table.sortable, Table.filterable, Table.selectable, Table.
↪pagination],
    footerPanels: [Table.sortable, Table.filterable, Table.selectable, Table.
↪pagination]
});

const GroupedTable = Table.Manager({
    providers: [Table.sortable, Table.filterable, Table.selectable, Table.
↪groupable],
    columnActions: [Table.sortable, Table.filterable, Table.selectable,
                    Table.columnDragger],
    toolbarButtons: [Table.sortable, Table.filterable, Table.selectable],
    headerPanels: [Table.groupable],
    footerPanels: [Table.sortable, Table.filterable, Table.selectable]
});

const ResizableTable = Table.Manager({
     columnActions: [Table.columnResizer, Table.columnDragger, Table.columnOrderer]
});

export default class TableTest extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            columns: columns
        };
        this.
    }

    render() {
        const stl = {height: '200px', marginBottom: '20px'};
        const stl2 = {height: '300px', marginBottom: '20px'};
        const columns = this.state.columns;
        const initSortColumns = Immutable.fromJS([
            {id: 'attr2', desc:true}
```

```
        ]).toSet();
        return (
            <div>
                <div style={stl}>
                    <SimpleTable columns={columns} orderedColumns={orderedColumns}␣
↪rows={rows}></SimpleTable>
                </div>
                <p>Manager</p>
                <div style={stl}>
                    <ManagedTable columns={columns} orderedColumns={orderedColumns}
↪ rows={rows}></ManagedTable>
                </div>
                <p>Sort</p>
                <div style={stl}>
                    <SortedTable columns={columns}
                                 orderedColumns={orderedColumns}
                                 initSortColumns={initSortColumns}
                                 rows={rows}></SortedTable>
                </div>
                <p>Filter</p>
                <div style={stl}>
                    <FilteredTable columns={columns}
                                 orderedColumns={orderedColumns}
                                 initSortColumns={initSortColumns}
                                 initFilter="20"
                                 rows={rows}></FilteredTable>
                </div>
                <p>select</p>
                <div style={stl}>
                    <SelectTable columns={columns}
                                 orderedColumns={orderedColumns}
                                 singleSelection={false}
                                 onSelectRows={(selected) => console.log(selected.
↪toJS())}
                                 rows={rows}></SelectTable>
                </div>
                <p>pagination</p>
                <div style={stl}>
                    <PagedTable columns={columns}
                                orderedColumns={orderedColumns}
                                rows={rows}
                                initPageSizeOptions={Immutable.Set([10, 20])}
                    ></PagedTable>
                </div>
                <p>resizable</p>
                <div style={stl}>
                    <ResizableTable columns={columns}
                                orderedColumns={orderedColumns}
                                rows={rows}
                    ></ResizableTable>
                </div>
                <p>groupable</p>
                <div style={stl}>
                    <GroupedTable columns={columns}
                                orderedColumns={orderedColumns}
                                rows={rows}
                                initGroupColumns={Immutable.OrderedSet(['attr2',
↪'attr3'])}
                    ></GroupedTable>
                </div>
            </div>
        );
    }
```

```
};
```

Contents

Preconfigured Tables

Components

- *table/Manager* (page 136)
- *table/Table* (page 139)
- *ToolbarMenu* (page 213)

Features

- *table/sortable/index* (page 138)
- *table/column_orderer/index* (page 128)
- *table/filterable/index* (page 129)
- *table/groupable/index* (page 130)
- *table/pagination/index* (page 137)
- *table/selectable/index* (page 137)
- *table/column_search/index* (page 129)

Contents

Modules

- *table/column_aggregator/index* (page 128)
- *table/column_dragger/index* (page 128)
- *table/column_orderer/index* (page 128)
- *Scrolled* (page 137)
- *table/selectable/index* (page 137)
- *table/sortable/index* (page 138)
- *table/Table* (page 139)

Functions

table/Manager

Local Navigation

- *Description* (page 137)
  - *Writing custom extensions* (page 137)

Description

The Manager function is responsible for creating a table component based on a set of options, that control the different features and controls of this component.

The components used in specifying options are usually feature modules, that pack all options relevant to this feature into one module, so that it can be easily used in the definition of a table component. For details on how these feature components are implemented, see section *Writing custom extensions* (page 137).

options are provided as an Object that contains an entry for each type of feature that may be activated. The different types for which feature components may be specified are:

- **Table:** The **Table** option allows to specify a React component that overrides the default table renderer component. The default component used here is *table/Table* (page 139). Usually you do not want to provide your own component here.
- **providers:** A list of **Providers**. These omponents alter the way in which table data is rendered. Providers are HOCs which are instantiated in reverse order in which they occur in the provided list, with **Table** as the base component. As a result of this,. properties flow from the first listed provider to the last, which finally passes it to the **Table** component.
- **columnActions:**
- **toolbarButtons:**

Writing custom extensions

A table feature such as *table/sortable/index* (page 138) is usually provided as a module which exports the different options provided to realize the feature, such as column actions or a provider.

table/pagination/index

Local Navigation

- *Description* (page 137)

Description

Scrolled

Local Navigation

- *Description* (page 137)

Description

Usage:

table/selectable/index

Local Navigation

Description

Provides selecting functionality in a table.

React Properties

This feature defines some more props for the table component:

Property	Type	Default	Use
singleSelection	bool	true	Single selection mode
withSelector	bool	false	Whether to display an additional column with checkbox for selection
onSelectRows	func	-	Function to be called when item is selected. The IDs of selected rows will be passed to this function.
initSelected	func	-	Initial selections to pass to the table

table/sortable/index

Local Navigation

Description

Provides sorting functionality in a table, by default by stringifying cell values.

Options

- Provider: implements sorting of rows
- columnActions: adds a button to each column header for sorting

Redux State

TODO document Redux components

Properties

`Table.sortable.Provider` by default stringifies cell values to use as a key for sorting. To change this behaviour, contentRenderers may define an additional field `sortFunction` :

```
MyContentRenderer.sortFunction = (a, b) => myExampleCompare(a, b);
```

This function should takes two parameters to be compared and returns -1 if the first is smaller than the second, 1 if the second is smaller than the first and 0 if both are equal.

A column definition may also define `sortFunction`, which will then override the contentRenderer's or default functions.

table/Table

Local Navigation

Description

Default table component implementation.

table/table_to_tree/index

Local Navigation

Description

Transform a normal table to a tree table via extracting hierarchical information from the rows.

React Properties

This feature defines some more props for the table component:

Property	Type	Default	Use
treeIDColumnID	string	-	Which column contains the id of the current node in the tree
treeParentColumnID	string	-	Which column contains the id of the parent node in the tree

table/treeview/index

Local Navigation

Description

Display the table rows hierarchically. It turns a specific column into tree view. It can be toggled between tree table and a normal table.

React Properties

This feature defines some more props for the table component:

Property	Type	Default	Use
treeColumnID	string	-	Which column should be used to display as tree view
treeNodes	list	false	Top level tree nodes with nested information of children nodes
onToggleTreeView	func	-	Function to be called when the tree view is toggled on/off.

tiles/FallbackTile

Local Navigation

Description

tiles/PersonTile

Local Navigation

Description

TileHelpers

Local Navigation

Description

Contents

Functions

Tile

Tile

TileHeader

TileHeader

Tree

Local Navigation

Description

This module provides diverse components to generically render tree widgets. The module provides ready-to-use components to display structure and object data provided by the CONTACT Elements server (see *Ready-To-Use Components* (page 141)), as well as an API to create custom tree components (see *Creating custom tree components* (page 141)).

Ready-To-Use Components

- *StructureTree* (page 216)
- *RestTree* (page 213)

Creating custom tree components

Custom trees are created based on the *tree/Tree* (page 146) component. Details for configuring this component can be taken from the components documentation.

- *tree/Tree* (page 146)
- *tree/TreeFilter* (page 148)
- *tree/TreeContainer* (page 148)
- *tree/TreeRenderer* (page 150)
- *tree/NodeRenderer* (page 142)
- *tree/SearchAdapter* (page 145)

tree/NodeRenderer

Local Navigation

Description

Tree Renderer API. This allows to create custom tree renderers. The high level functions provided in this module can be used to generate custom tree renderers.

The concepts used here are Renderers, Enhancements and Extractors.

Renderers

Renderers are either base renderers or enhancements. Base Renderers are renderers, while enhancements are HOCs that return a renderer if provided a renderer as parameter. Available components are:

Base Renderers:

- *NodeRenderer* (page 144): Renders a text string. Search results are highlighted.

Enhancements:

- *WithHoverLink* (page 144): Render a link to the right of the content, when mouse is hovered over over content.
- *WithLink* (page 145): Render a link to the right of the content.
- *WithIcons* (page 145): Render a set of icons to the left of content.

Renderers and enhancements are usually provided as HOCs that need to be parametrized to obtain a renderer or enhancements.

Extractors

An extractor is usually provided as argument to a generator. It extracts the relevant information for rendering from a node.

An extractor may either be provided as an array of strings - then it will be used as argument to the getIn function of the node to extract the information required by the generator - or as a function. This function will be applied to each node to extract the information for the generator.

As an array an extractor looks like this:

```
const e = ['content', 'icon'];
```

The equivalent function is:

```
const e = node => node.getIn(['content', 'icon']);
```

Composing Renderers and Enhancements

Node renderers may be created using the create function. This function takes as argument a renderer and a list of enhancement, and returns the renderer, by applying the provided enhancements to the renderer.

Suppose we want to render a node which has the following content field:

```
{
    label: <string>,
    icon_url: <string>,
    link: <string>
}
```

In order to render a node for this structure with link and icon, the following renderer component may be used:

```
const RestNodeRenderer =
    NodeRenderer.create(
        NodeRenderer.NodeRenderer(['content', 'label']),
        NodeRenderer.WithIcons(node =>
            [
                {
                    url: node.getIn(['content', 'icon_url']),
                    title: node.getIn(['content', 'label'])
                }
            ]
        ),
        NodeRenderer.WithHoverLink(['content', 'link'])
);
```

Implementing custom renderers

It is also possible to implement custom renderers. An elaborate example can be found in cs.sharing: RecipientListTree provides a delete button for each node, which issues a REST call to delete this node.

A custom renderer enhancement is a function that takes a NodeRenderer as a parameter and returns a component that renders the provided NodeRenderer, usually along additional components:

```
const MyNodeRendererEnhancement(props) => {
  return props => <NodeRenderer {...props} />
}
```

This example is a no-op: It simply renders the provided node. Simple enhancements can be realized easily. Suppose you want to render a folder or file-icon, depending on wether your node has children:

```
const WithFsIcon(props) => {
  return props => (
    <span>
      {
        <Icon src={props.get('has_children') ?
                   "path/to/folder.svg" :
                   "path/to/file.png"}/>
      }
      <NodeRenderer {...props} />
    </span>
  );
}
```

Note that this particular example may be easily achieved using the WithIcons enhancement.

Contents

Functions

create

```
create(renderer: ReactComponent, enhancements: array): ReactComponent
```

Takes:

- ***renderer*: a base renderer, e.g.,** *DefaultRenderer* (page 144)
- *enhancements*: an array of HOCs

Create an enhanced renderer from a base renderer and a list of enhancements.

DefaultRenderer

```
DefaultRenderer(props: object)
```

Simple default renderer that uses the nodes `content` field as label.

generateExtractor

```
generateExtractor(fnOrArray): function
```

Input is a function or array - an extractor. If it is an array, it returns a function that retrieves the value at that path from a provided node. If it is a function the function itself is returned. Used so either Paths for Immutable.Map.getIn or functions may be passed to Renderer-Generators.

Use this to create your own API compliant Renderer-Generators.

NodeRenderer

```
NodeRenderer(nodeToLabel): ReactComponent
```

Takes:

- ***nodeToLabel*: extractor returning a string or component** used as label by the renderer.

HOC that creates a base renderer component. This simply displays the extracted label.

WithHoverLink

```
WithHoverLink(nodeToLink): HOC
```

Takes:

- ***nodeToLink*: should extract a string representing a** URL from the node.

This Generator enhances a given NodeRenderer with a link symbol rendered to the right of the component. The link is only visible when the mouse is over the component.

WithIcons

```
WithIcons(nodeToIcons): HOC
```

Takes:

- *nodeToIcons*: extractor that generates a list like the above

This generator enhances a given NodeRenderer with a list of icons that will be rendered left to the component

The list generated by the provided exctractor should have the form:

WithLink

```
WithLink(nodeToLink): HOC
```

Takes:

- ***nodeToLink*: should extract a string representing a** URL from the node.

This Generator enhances a given NodeRenderer with a link symbol rendered to the right of the component.

tree/SearchAdapter

Local Navigation

Description

The SearchAdapter interface is used as an Interface between the ToolbarSearch widget and arbitrary components that want to support searching. The interface may be used to encapsulate arbitrary search algorithms.

The actions provided by the widget are the following (function names in SearchAdapter):

- onSendQuery: invoked when a query should be sent. returns a promise on success
- nextResult: invoked when next result should be displayed.
- previousResult: invoked when previous result should be displayed.
- cancelSearch: clear results.

The toolbar updates the state of the navigationstate according to the return value of the function, which will be invoked when the promises returned by the interface are resolved:

- getNavigationState: returns a JSON object containing boolean values for:
  - previousResult
  - nextResult

TODO use immutable.map for search results

Contents

Classes

Functions

tree/Tree

Local Navigation

Description

The Tree class is a highly customizable component to display tree structures. It provides an interface to search for nodes in the tree, and to navigate through the search result. Furthermore, keyboard navigation is provided to navigate through the trees nodes.

Specifying the tree's structure

Subclasses of Tree must at least set the `rootNode` attribute in the component's state. The root node must be an `Immutable.Map` instance, with the following entries:

- id: a unique ID for the node
- content: the actual content belonging to the node, the node renderer component must be able to render this
- expanded: bool, describes the current state of the node
- children: an `Immutable.List` instance with child nodes
- has_children: bool, describes whether the node has children. If this is set to `true`, but `children` is empty, `fetchNodeChildren` is called to retrieve the child nodes.

Deferred loading of children

If your tree implementation can be expanded dynamically, your subclass must also implement the method `fetchNodeChildren`. This method will be invoked, if a node indicates it has children that are not yet fetched (the nodes has_children property is set to true, but the property children is an empty list) (from the server). fetchNodeChildren must be a thenable that:

- fetches the children by application-specific means
- updates the tree node whose children have been fetched, by invoking `super.updateTreeNode(...)` which is a thenable that resolves after the tree has been updated.
- resolves after the tree has been updated.

```
attachChildren(node, children) {
  return node.set('children', Immutable.List(children));
}

fetchNodeChildren(node) {
  return runRestGet(node.get('child_url')).then(children => {
     const nodeWithChildren = this.attachChildren(node, children);
     return super.updateTreeNode(nodeWithChildren);
  });
}
```

If your tree implementation does not know in advance, wether there are child nodes to be fetched, you may initialize all nodes with has_children set to true, and when you discover that no child nodes are available for a given node fetchNodeChildren may set the has_children property to false, instead of updating children.

Searching for nodes

Two properties control, how the tree may be searched:

- **SearchAdapter** is a property that allows to enhance the mechanism by which the tree's structure is navigated during search.
- **matcher** controls how the search algorithm determines a match.

A matcher must be a function that contains the following signature:

```
(node, search) => Boolean
```

`node` is the node for whom a match should be determined, while `search` is the string that is input into the searchwidget.

SearchAdapters are provided by module tree.SearchAdapter. Available are SimpleSearchAdapter, which implements a basic top-down depth-first search on all already loaded nodes and BackendSearchAdapter, which fetches children using fetchNodeChildren.

Rendering the title

The title is rendered as follows: If a TitleRenderer React.Component is specified, this component will be rendered. It will be provided the tree's props and state as props. If no TitleRenderer is specified the prop title will be used as the title. The prop title can be overwritten by specifying a state-variable title when subclassing the tree.

Customizing tree rendering

The property `TreeRenderer` allows to customize the way the tree component is rendered. Rendering occurs through a series of HOCs terminated by a tree renderer component. The renderer itself is responsible to render the tree structure provided as `rootNode` property, while the nested HOCs may modify the tree's properties, and enhance the tree's UI representation.

The default configuration is as follows:

```
TreeContainer.DefaultRenderer = compose(
  Tree.TreeRenderer,
  Tree.TreeFilter.ShowSearchResults,
```

```
  Tree.TreeContainer,
);
```

In addition, the component ModalTreeContainer provides a simplified default renderer for use in modal dialogs, which avoids the use of modal components.

Customizing node rendering

The tree component provides a module NodeRenderer, which provides a set of HOCs to customize the trees ability to render nodes. Examples of using this API may be found in StructureTree and RestTree, which use configurable default components provided by NodeRenderer to extract labels and icons from the tree-nodes provided by a domain-specific API to render their nodes.

React Properties

Property	Type	Default	Use
className	string	-	Attach a custom classname to the tree component.
NodeRenderer	string	-	React component that will be used to render the node content
hasFocus	bool	true	If true, the tree should handle keybpard events for navigation
TitleRenderer	component	-	
TreeRenderer	component	-	Custom tree rendering
onSelect	func	-	Callback on node selection
title	string	-	
hideRoot	bool	false	If true, root node is not rendered

tree/TreeContainer

Local Navigation

- *Description* (page 148)

Description

This module provides a HOC that renders a *ContentBlock* (page 159) displaying widgets for the tree's operations in the blocks toolbar area and the TitleRenderer in the title area component.

tree/TreeFilter

Local Navigation

- *Description* (page 149)
- *Contents* (page 149)
  - *Functions* (page 149)

Description

This module provides filter HOCs for use in tree components. A filter is a tree HOC of the form

```
TreeRenderer => TreeRenderer
```

which renders the provided TreeRenderer argument, but replaces the tree's rootNode with a modified version, that usually removes nodes from the tree's model.

Contents

Functions

ShowPaths

```
ShowPaths(getNodes): undefined
```

Takes:

- *getNodes*: a function `props => [treeNodes]`

Returns: a filter HOC

A function that creates a filter to display a subtree based on a set of nodes, that should be included in the subtree.

The parameter getNodes provides a function that must return an array of nodes. Input to the function is the trees props object. The filtered tree will only include the paths leading to the nodes in the array returned by getNodes.

ShowSearchResults

```
ShowSearchResults(TreeRenderer): undefined
```

Takes:

- *TreeRenderer*: a TreeRenderer component

Returns: a TreeRenderer component

A filter HOC that displays all search results in the active search if a search is active, or the whole tree structure if no search is active.

ShowSelectedSearchResult

```
ShowSelectedSearchResult(TreeRenderer): undefined
```

Takes:

- *TreeRenderer*: a TreeRenderer component

Returns: a TreeRenderer component

A filter HOC that displays the selected search result if a search result is selected, or the whole tree structure if no search is active.

tree/TreeRenderer

Local Navigation

- *Description* (page 150)

Description

CatalogPreview

Local Navigation

- *Description* (page 150)

Description

Catalog preview component

Property	Type	Default	Use
contextObject	Immutable.Map	•	The object to render

FlatCatalog

Local Navigation

- *Description* (page 150)

Description

This component renders a catalog. You might retrieve most of the properties using the backend function `FormInfoBase.get_catalog_config`.

Property	Type	Default	Required	Use
formData	Immutable.Map	•	•	Form data to be used to query catalog data entries
contextClass	string	•	•	Class of context objects for catalog data
activeLanguage	string	•	•	In which language should catalog data be loaded
selectURL	string	•	yes	URL to retrieve the result of the user selection
catalogTableURL	string	•	yes	URL to load table data and configuration for displaying the catalog entries
proposalCatalogURL	string	•	•	URL to load table data and configuration for displaying the proposal table entries
proposalLabel	string	•	•	Proposal table title
onCancel	function	•	•	Function that is called if the user cancels the selection. The default handler closes the tab
onSelect	function	•	yes	Called with the result of the selectURL as parameter
userSettings	object	•	•	Additional user settings

SplitLayoutWithPersistence

Local Navigation

- *Description* (page 151)

Description

Extended SplitLayout component which saves the last settings in the user settings or loads them from the user settings.

Property	Type	Default	Use
componentID	string	•	Settings are persisted in the user settings with [componentID, settingKey]
settingKey	string	•	""
size	string or number	132px	The size of the second pane if no other settings is specified
visible	bool	true	Is the second panel initially visible. Used if no other settings is specified
initialFetch	bool	•	initially fetch settings

Listener

Local Navigation

Description

The channel listener retrieves its values from the channel broadcasted by its provider. These values are then injected as properties into its only child. To provide the same properties to multiple components, wrap each of these components into one listener with the same channel and mapping.

React Properties

Property	Type	Default	Use
channelName	string	-	The channel identifier for this listener to subscribe to This property cannot be changed after initial setup!
channelMapping	object	-	A mapping that maps from property names to the keys in the channel. This is used to retrieve the actual values from the channel.
channels			A list of {channelName, channelMapping}

Provider

Local Navigation

Description

The channel provider which sends updated values using the new channel mechanism down to its listeners. Any values sent from parent providers are overwritten using local properties. This component must only have one child.

React Properties

Property	Type	Default	Use
channelName	string	-	The channel identifier for listeners to subscribe to. This property cannot be changed after initial setup!
channelValues	object	-	An object specifying the mapping of channel keys to actual values

ResultWithDetails

Local Navigation

Description

Renders a search result, and possibly a detail view if one is configured. This component also handles the propagation of the result table selection to the detail view.

Contents

Functions

onSelectionChanged

If a detail view is configured, propagate changes in the resault table selection. We use setContextObjects and ContextObjectWrapper, so that the REST objects for the selection will be loaded automatically.

selectedObjectIDs

Convert the selectionIds from the table to the object IDs for the actual REST objects; that is what setContextObjects uses.

SearchResult

Local Navigation

Description

Renders a table with a search result and possibly additional details, or shows an error, throbber etc. depending on the current state of the search.

SearchTabs

Local Navigation

Description

Shows a list of tabs, each with an individual search instance.

AmountBadge

Local Navigation

Description

Displays the amount of something.

React Properties

Property	Type	Default	Use
amount	number	-	Amount to be displayed.
tooltip	string	-	Localized tooltip to explain the displayed amount.

DropdownIconButton

Local Navigation

Description

Note: This component has been deprecated. Please use cs-web-components-base-overlays.DropdownIconButton.__default__ instead.

A dropdown button that renders an icon as content. In addition to the properties in *Button* (page 67), it receives the following properties:

Name	Type	Default	Description
buttonStyle	string	"primary"	Determines the button style
iconName	string	-	Load an icon defined in the backend by its resource id
iconSrc	string	-	Load an icon by the provided URL.
size	string	"sm"	The icon size to be displayed.
children	element	-	Child components are rendered in the buttons popup menu

DropdownIconTextButton

Local Navigation

- *Description* (page 155)

Description

A dropdown button that renders an icon and a text label. In addition to the properties in *Button* (page 67), it receives the following properties:

Name	Type	Default	Description
buttonStyle	string	"outline"	Determines the button style
label	string	-	The label that is displayed in the button
iconName	string	-	Load an icon defined in the backend by its resource id
iconSrc	string	-	Load an icon by the provided URL.
children	element	-	Child components are rendered in the buttons popup menu

DropdownTextButton

Local Navigation

- *Description* (page 155)

Description

A React component that renders a button with a dropdown menu which contains the provided children. In addition to the properties in *Button* (page 67), it receives the following properties:

Name	Type	Default	Description
buttonStyle	string	“outline”	Determines the button style
label	string	-	The label that is displayed in the button
children	element	-	Child components are rendered in the buttons popup menu

IconButton

Local Navigation

- *Description* (page 156)

Description

This button contains an icon, either fetched by name from the backend, or by a provided arbitrary source URL. Depending on the source you want to use, either property `iconName` or `iconSrc` should be provided. In addition to the properties in *Button* (page 67), it receives the following properties:

Name	Type	Default	Description
tabIndex	integer	-	Override the default tabIndex of this component
buttonStyle	string	“primary”	Determines the button style
iconName	string	-	Load an icon defined in the backend by its resource id
iconSrc	string	-	Load an icon by the provided URL.
size	string	“sm”	The icon size to be displayed.
onClick	func	-	A callback function, which is executed when the button is pushed.

IconTextButton

Local Navigation

- *Description* (page 156)

Description

This button renders both an icon and a text label. Depending on the source you want to use, either use the property `iconName` for fetching from the backend or `iconSrc` for an arbitrary source url. In addition to the properties in *Button* (page 67), it receives the following properties:

Name	Type	Default	Description
buttonStyle	string	“outline”	Determines the button style
label	string	-	The label that is displayed in the button
iconName	string	-	Load an icon defined in the backend by its resource id
iconSrc	string	-	Load an icon by the provided URL.
onClick	func	-	A callback function, which is executed when the button is pushed.

TextButton

Local Navigation

Description

TextButton is a simple button that displays a text label. In addition to the properties in *Button* (page 67), it receives the following properties:

Name	Type	Default	Description
buttonStyle	string	"outline"	Determines button style
label	string	-	The label that is displayed in the button
onClick	func	-	A callback function, which is executed when the button is pushed.

ButtonGroup

Local Navigation

Description

Container component for grouping Button components into a single horizontal bar with small delimiters. Items to group are passed as children.

React Properties

Name	Type	Default	Description
className	string	-	A custom css className to attach to the component
withRedundantDropdown	boolean	true	true if the last button is a redundant DropdownButton

ButtonToolbar

Local Navigation

Description

Container component for laying out several Button or ButtonGroups components with large delimiters into a single horizontal bar. The button items to be laid out are passed as children.

React Properties

Name	Type	Default	Description
className	string	-	A custom css className to attach to the component

Collapsible

Local Navigation

Description

A container that consists of a header and a body. Inside the body, child components are displayed. The header consists of a clickable expander control and a header text line. If the control is clicked, the body is collapsed/expanded.

PersistentCollapsible (page 175) is a variation of this component, that persists its state.

React Properties

Property	Type	Default	Use
header	string/node	-	Text to be displayed in header
initialCollapsed	bool	true	Initial state of the component
onCollapsed	func	undefined	Callback invoked when component state changes
children	nodes	-	Components to be displayed in body

Automation Properties

CollapsibleControl: The control displayed in the header of the collapsible. Click on it to toggle the collapsibles state. The component defines a data-ce-state that represents the state of the collapsible.

Value	Use
collapsed	Collapsible is collapsed
expanded	Collapsible is expanded

ImageViewer

Local Navigation

Description

A simple plugin component to display an image file.

React Properties

Property	Type	Default	Use
url	string	-	The URL of the image file to render
filename	string	-	The filename of the image file

ContentBlock

Local Navigation

Description

The ContentBlock component provides a unified way to enhance a component with a title and one or more toolbars, as well as the ability to collapse the component. The title is alway visible, no matter if the component is collapsed or not. The toolbar(s) on the other hand will only be visible as long as the component stays expanded. Furthermore the title can contain an icon, if specified.

Examples

```
<ContentBlock collapsible={true}
              expanded={true}
              title={<h3>Hallo Welt</h3>}>
   <ContentBlock.Header>
      <Button ... />
      <Button ... />
   </ContentBlock.Header>
   <ContentBlock.Body>
     <TreeRenderer {...this.props}/>
   </ContentBlock.Body>
</ContentBlock>
```

React Properties

Property	Type	Default	Use
title	string/component	""	A title string or component to be displayed on the upper left of the component
icon	component	null	A component to be displayed in the header
contentStyle	string	'default'	A semantic style, according to Bootstrap states. One of 'default', 'primary', 'warning', 'danger', 'info', 'success'.
collapsible	boolean	true	true if the component should be collapsible
expanded	boolean	true	true if the initial/current state should be expanded
onExpansionChanged	function	-	Invoked when components expansion state has changed. Receives new expansion state as argument.
headerDragger	function	Identity	A function that will be wrapped around the header to make it draggable. The function is typically a property injected by DragSource()
children	[components]	-	the child components in the ContentBlock

ContentBlockGroup

extends React.Component

Local Navigation

Description

ContentBlockGroup can be used to group several *ContentBlock* (page 159) instances. Grouping such elements will lead to an accordion like layout. All panels inside this layout group can be expanded independently. The default expansion state of each component is defined due to its property (see *ContentBlock* (page 159)).

Examples

The ContentBlockGroup can be used to group *ContentBlock* (page 159) or self defined components based on *ContentBlock* (page 159).

Using ContentBlock

```
<ContentBlockGroup>
   <ContentBlock ...>
      ...
   </ContentBlock>
   <ContentBlock ...>
      ...
```

```
   </ContentBlock
   <ContentBlock ...>
      ...
   </ContentBlock>
</ContentBlockGroup>
```

Using self defined components

If using self defined components, these ones must return an element of type *ContentBlock* (page 159). If so, they can be embedded in ContentBlockGroup as well.

```
<ContentBlockGroup>
   <MyComponentX ... />
   <MyComponentY ... />
   <MyComponentZ ... />
</ContentBlockGroup>
```

React Properties

Property	Type	Default	Use
children	[components]	-	the child components of the ContentBlockGroup (only *ContentBlock* (page 159) types are supported)

ContentOperationToolbar

extends React.Component

Local Navigation

- *Description* (page 161)

Description

This component displays the configured operations for the provided `contextObject` in a button toolbar. Optionally, an `operationContextName` may be provided, from which to display operations. If a button is pressed, the operation is triggered in the surrounding operation scope.

In order to disable the toolbar when an operation is running, the `isOperationRunning` property may be used.

Name	Type	Default	Description
contextObject	object	-	The object whose operation to display.
operationContextName	object	-	Optional: If provided, only operations from this context will be available.
iconSize	string	sm	Size of the toolbars buttons.
isOperationRunning	bool	false	Buttons will be disabled while operation is running if this is set to true.

DisplayContextLoader

Local Navigation

Description

DisplayContextLoader takes a contextObject and a displayContext name, and calls the backend to render the object using the configured form associated with the display context and the object's class. The result is passed as properties `fields` and `values` to the children.

React Properties

Property	Type	Default	Use
contextObject	Immutable.Map	-	The object to render using the display context
displayContext	string	-	Name of the display context to use

FileDropzone

Local Navigation

Description

This component enables dropping files that dragged from file system in it, or to select files via dialog.

Examples

```
class MyFiles extends React.Component {
    onDrop(files) {
        console.log('received', files);
        this.setState({
            files: files
        });
    }

    render() {
        <div>
            <FileDropzone onDrop={this.onDrop.bind(this)}>
```

```
                <p>Drag and drop files here</p>
            </FileDropzone>
            {this.state && this.state.files ?
                <FileDropzone.Preview files={this.state.files}/> : null
            }
        </div>
    }
}
```

React Properties

Property	Type	Default	Use
onDrop	func	•	A callback, fired when files get dropped here
autoHide	boolean	false	whether to hide content in this component until files are being dragged

Static Methods

getPreview(file): Get the generated URL to the specific file that is dropped or chosen. The result can be used e.g. as value to `src` attribute on `img` tag.

Instance Methods

choose(): Open the browser native file choosing dialog.

HelpReference

Local Navigation

Description

Renders a help button. Clicking the button will open the help page defined by the given helpId.

React Properties

Property	Type	Default	Use
helpId	string	-	The Help-ID of the help page
helpUrl	string	-	The URL of the help page if no id is provided

Icon

Local Navigation

Description

The component display image in predefined sizes.

React Properties

Property	Type	Default	Use
name	string	•	Name of configured icon
src	string	•	URL to load the icon
size	string	sm	Size of that image. Can be: sm, md, lg

Note: If `src` is given, it will be used to load the icon. Otherwise the component tries to generate the url for configured icon according to the `name`.

ImageFiles

Local Navigation

Description

Displays the images in a list of files. The files need to be provided as `cdb_file` objects.

React Properties

Property	Type	Default	Use
files	array	-	List of files to display

ObjectImagePreview

Local Navigation

Description

An adapter for *ImageFiles* (page 164), that extracts and displays the files listed under `relship:files` in an object.

React Properties

Property	Type	Default	Use
contextObject	JS object	-	The object whose relship:files should be displayed

ObjectApplicationBar

extends React.Component

Local Navigation

Description

TODO write documentation

Applications

Local Navigation

Description

Sidebar component that displays a list of installed applications.

Favorites

Local Navigation

Description

Displays the WebUI favorites of the current user. Note that these are different from the favorites created using the Windows Client.

Automation Properties

Favorites: The view itself can be retrieved using this key.

FavoritesEntry_n: Replace n with a number. The entries can be retrieved using these IDs. They are numbered starting from 0.

History

Local Navigation

Description

Displays the history items of the current user. Note that these are different from the history items created using the Windows Client.

AutomationProperties

History: The history view itself can be retrieved using this key.

HistoryItem_n: Replace n with a number. The entries can be retrieved using these IDs. They are numbered starting from 0.

ClassRelshipLoader

Local Navigation

Description

Wrapper component that loads relationship metadata for a class from the backend. The class can be given either directly via the `classname` property, or it can be extracted from a `contextObject` property. ClassRelshipLoader is a FACC that calls its child as a funtion, with the relship data as parameter of type Immutable.Map.

Property	Type	Use
classname	String	Name of the CE class to load data for. Takes precedence over contextObject.
contextObject	Immutable.Map	An object whose classname is used to load data for. Is ignored if classname is given.

CompactHeader

Local Navigation

Description

A component to show the thumbnail, type, description and operations of an object, which will fit into e.g. a detail area.

It can also be used as configured component. If the default result handling of the operations is not wanted, e.g. it should stay on current page instead of navigate to the object page, the callback functions should be specified in order to disable the default handling .

React Properties

Property	Type	Default	Use
contextObject	Immutable.Map	-	The object to be showed
onOperationSuccess	func	undefined	The callback function in case an operation ends successfully
onOperationFailure	func	undefined	The callback function in case an operation fails

ConnectedRelationshipTable

extends React.Component

Local Navigation

Description

Wraps a RelationshipTable with logic to retrieve relationship configuration data from the application setup, and handles expanded state. This component is intended for use from page configuration files.

ConnectedRelationshipTreeTable

extends React.Component

Local Navigation

Description

Wraps a RelationshipTable that renders a TreeTable with logic to retrieve relationship configuration data from the application setup, and handles expanded state. This component is intended for use from page configuration files.

EditableObjectLabel

Local Navigation

Description

A label that is used to display an object. If callbacks for editing or deleting the object are provided, controls for the respective actions are displayed if the user is hovering over the label. A click on the controls triggers the respective action.

React Properties

Property	Type	Default	Use
data-ce-id	string	EditableObjectLabel	Overwrites id to retrieve this dom element
text	string	-	Label text to be displayed
icon	string	undefined	URL to icon for this object
onEditClicked	function	undefined	Callback for edit action
onDeleteClicked	function	undefined	Callback for delete action

Automation Properties

EditableObjectLabel: The label itself can be retrieved using this id. By passing in `data-ce-id` as a React Property, this value can be overridden.

EditButton: The control that is used to edit the object represented by this label.

DeleteButton: The control that is used to delete the object represented by this label.

ExpandedStateHelper

Local Navigation

- *Description* (page 169)

Description

FACC helper component to manage the expanded state of the wrapped child

FileList

Local Navigation

- *Description* (page 169)
  - *React Properties* (page 169)

Description

Render a list of files attached to an object via `relship:files`. This control can also trigger operations for these files. The available operations are defined in the operation context `WebUIFileControl`.

React Properties

Property	Type	Default	Use
contextObject	JS object	-	The object whose relship:files should be displayed
quickAccess	JS Array	`["CDB_Import", "CDB_Export", "CDB_Delete"]`	Names of operations to be displayed as icons in the panel header of the filelist
uploadURL	JS string	-	URL to upload files to. If not provided, the URL of the context object will be used.

FileSelectionProvider

Local Navigation

- *Description* (page 170)
  - *React Properties* (page 170)

Description

A convenience component, that wraps a `Provider` useful for file selections around its children. This provider supplies an `onSelect` function and the current `selection` to any `Listener` present in the wrapped children. All props given to this component beside `className` and `channelName` will be propagated to the children.

React Properties

Property	Type	Default	Use
className	String	-	CSS class to use on the wrapper
channelName	String	fileSelection	The name of the channel to use

RelationshipFetcher

Local Navigation

Description

Component build after the FACC pattern that retrieves relationship data as a table, and passes it on to its function child.

RelationshipGroup

Local Navigation

Description

Render a control that contains relationships of an object. The relationships that are usable have to be provided in the app setup.

React Properties

Property	Type	Default	Use
include	array	•	A list of relationship REST names. These relationships will be displayed. If the array is undefined the relationships that are configured as dialog registers are used
exclude	array	•	A list of relationship REST names. These relationships will not be displayed.
tableHeight	string	`300px`	The height of each relationship table.
tableClassName	string	•	Style classes of each relationship table.
specOptions	object	•	A mapping of relationship specific options. Indexed by the relationship name. Each value is also a mapping of possible properties of `RelationshipTable` component.

RelationshipTable

Local Navigation

Description

A Control to show the result of a relationship navigation in a table.

React Properties

Property	Type	Default	Use
contextObject	Immutable.Map	-	The parent object of the relationship
relshipName	string	undefined	The *Semantical Name* of the relationship as configured in the system
tableName	string	undefined	The name of the table that is used to display the result. If the name is not defined the system will use the table that is configured for the given relationship
singleSelection	bool	false	Allow only single or multi selection
height	string	-	Height of the control. If set to 'auto', the table size will auto fit its rows.
hideTableHeaderPanels	bool	false	If true, the feature panels over the table would be turned off to save places.
tableClassName	string	-	Style classes of the table component.
columnAggregator	object	-	Settings for an aggregation row.

RelationshipTableConfig

Local Navigation

- *Description* (page 172)

Description

Component build after the FACC pattern that build a table configuration suitable as prop for TableWrapper

OperationDeleteDialog

extends React.Component

Local Navigation

- *Description* (page 172)

Description

A specialized OperationHandler that displays an Alert-dialog to query deletion of an object.

OperationModal

extends React.Component

Local Navigation

Description

An operation handler that executes the given operation by rendering the mask configured in CONTACT Elements in a modal dialog.

OperationToolbar

extends React.Component

Local Navigation

Description

Display the Toolbar on top of core operations based forms.

React Properties

Property	Type	Default	Use
selectCB	function	-	Called when Button is selected.

Organizer

Local Navigation

Description

A component to allow reorganize items between different lists via Drag & Drop.

React Properties

Property	Type	Default	Use
itemLists	Immutable.List	-	Lists of items to be displayed
placeholder	string	-	Placeholder text if a list is empty
onItemListsChange	func	-	Callback fires if the itemLists has been changed. The changed lists is passed in.

Each list in the `itemLists` should be an Immutable.Map and contain `label`, `items` and optionally, `id` and `unsortable`. If `unsortable` is set to true, the items can not be dragged to sort in its hosted list.

The `items` in each list should be an Immutable.List of item objects. Each such object should be an Immutable.Map and contain `id` and optionally, `label` and `renderer`. If no `renderer` exists, the item would be displayed using a default renderer. The default renderer will render `label` of that item, which can be e.g string or components. As fallback the `id` would be displayed.

ApplicationFrame

Local Navigation

Description

This component renders the *CDBTitleBarContent* (page 175) and the *NavigationPane* (page 174), along with the content of a web application.

React Properties

Property	Type	Default	Use
children	React Components	•	

NavigationPane

Local Navigation

Description

The component that is usually displayed on the left side of the CONTACT Elements Web UI. This component shouldn't be directly used, as it is rendered as part of *ApplicationFrame* (page 174).

React Properties

Property	Type	Default	Use
collapsed	bool	false	State of the sidebar

Automation Properties

ApplicationSidebar: This element represents the sidebar as a whole. The `data-ce-state` allows to determine if the sidebar is currently visible.

Value	Use
collapsed	Sidebar is collapsed
expanded	Sidebar is expanded

CDBTitleBarContent

Local Navigation

Description

The application titlebar that is usually displayed on the top of the CONTACT Elements Web UI. This component shouldn't be directly used, as it is rendered as part of *ApplicationFrame* (page 174).

Automation Properties

PersistentCollapsible

Local Navigation

Description

Extends *Collapsible* (page 158) to persist its state.

React Properties

Property	Type	Default	Use
header	string/node	-	Text to be displayed in header
onCollapsed	func	undefined	Callback invoked when component state changes
collapsedKey	nodes	-	A unique key to persist the state to
children	nodes	-	Components to be displayed in body

ObjectContent

Local Navigation

Description

A plugin container component that displays the "content" of an object in the form of a file. Which file to use is determined by the backend, through the URL of the context object + '/file'. The content_type of the file is used as discriminator to find a plugin component that is suitable for displaying the file format.

The plugin components receive the URL of the file they should render in the `url` property, and the name of the file as `filename`.

React Properties

Property	Type	Default	Use
contextObject	Immutable.Map	-	The object whose content is shown
pluginContextName	string	'content-view'	The plugin ID for the configuration
showPlaceholder	bool	-	Whether to show an empty view if no context object selected

RelatedObjects

Local Navigation

Description

A plugin container component to show objects related in some way to the contextObject. Which objects should be displayed is configured in the backend in the 'Assigned objects' configuration.

The objects are displayed as small 'tiles', where the tiles are dynamically determined React components, based on the class of the objects (ie. the `discriminator` in the plugin configuration must be a class name, see *Frontend Plugins* (page 32) for details). The tile components receive the object they must render in their `contextObject` property.

React Properties

Property	Type	Default	Use
contextObject	Immutable.Map	-	The object that is the origin of the relations shown
pluginContextName	string	'class-tile-small'	The plugin ID that configures the tiles

ProgressBar

Local Navigation

- *Description* (page 177)

Description

A `ProcessBar` shows progress in percent.

Property	Type	Default	Use
value	number	•	The progress value in percent
hideLabel	bool	•	Hide the label text

SearchField

extends React.Component

Local Navigation

- *Description* (page 177)

Description

Name	Type	Default	Description
productName	string	-	The brandable product name.
searchCallback	func	-	Callback invoked when enter is pushed.
searchTextChangedCallback	func	-	Callback invoked when content changes.
searchText	string	-	Initial content of the search field.
searchButtonSize	string	-	Size of the search button.

This component is an extended TextInput component with type ahead feature that accesses the Enterprise Search. It will open a drop down list to show possible options during typing. `productName` is inserted into the placeholder of the field.

Tag

Local Navigation

- *Description* (page 178)

Description

A simple label component that renders a value as a tag. The component provides callbacks for clicking the tag and the provided glyph. The glyph is usually used as a remove-button. In order to render these components, consider using *Tags* (page 178)

Name	Type	Default	Description
name	string	-	The value to be displayed.
id	string	-	Id of the tag, used in the callbacks.
onClickIcon	func	-	Callback that is executed, when the icon is clicked.
onClickTag	func	-	Callback that is executed, when the tag is clicked.

Tags

Local Navigation

- *Description* (page 178)

Description

Container for displaying *Tag* (page 178) elements. This component gets a tag definition in form of an array, and renders tag elements based on that configuration.

The structure of a tag configuration is:

```
{
  id,   // Identifier of the tag
  name, // Name to be displayed
}
```

Properties accepted by this component are:

Name	Type	Default	Description
hidden	bool	false	Tags will be rendered with display none
glyph	string	delete	Glyph displayed in the right corner of each tag.
onClickIcon	func	-	Callback that is executed, when a tag's glyph is clicked.
onClickTag	func	-	Callback that is executed, when a tag is clicked.

Throbber

Local Navigation

- *Description* (page 179)

Description

A control that shows a loading animation.

Zoomer

Local Navigation

- *Description* (page 179)

Description

This Component provides basic controls for zooming in and out of embedded media content, e.g. images. It displays the current zoom level as percentage, and provides buttons to zoom in and out, to reset, as well as to scale the content to best fit its container.

Note that this component does not handle zooming logic, manipulation of zoom level depends on the content that is modified, and should be implemented in the component that renders the content.

The following properties should be provided to Zoomer:

Name	Type	Default	Description
currentScale	number	-	The current scale of the associated content, this should be stored in the state of a parent component.
onZoomIn	func	-	Increase the zoom level, and zoom into the associated content.
onZoomOut	func	-	Decrease the zoom level, and zoom out of the associated content.
onReset	func	-	Reset the zoom level to a default value.
onZoomToFit	func	-	Set the zoom level to a value that best fits the layout in which the content is rendered.
onZoomToFitHorizontal	func	-	Set the zoom level to a value that best fits the width of the layout.
onZoomToFitVertical	func	-	Set the zoom level to a value that best fits the height of the layout.

ConfiguredComponent

Local Navigation

- *Description* (page 180)

Description

Takes an application configuration, and returns a React component that renders this configuration. Configuration members: * name: name for the root component, looked up in the registry * properties: properties for the root component * children: List of configurations for child components, rendered as normal children * components: Map with named configurations, resulting elements become props * componentClasses: like components, but classes are passed, not elements

ContextObjectSetter

Local Navigation

Description

`ContextObjectSetter` provides a callback function `onSelect` to its single wrapped component. This callback can be used to set the selected object(s) for a selection whose name is given in the `setPath` property.

React Properties

Property	Type	Default	Use
setPath	string	•	Path for local selection

ContextObjectWrapper

Local Navigation

Description

`ContextObjectWrapper` takes the object(s) stored under the selection named by the property `readPath`, and supplies them as `contextObjects` to its single wrapped child component. The last object in that list (or undefined if the list is empty) is supplied as `contextObject`. This tracks the behaviour of selection in tables: the last object in the list is the last that has been selected.

In addition, an `onSelect` callback is supplied, that serves to set a dependent selection. The name of that selection is formed by appending the value of the property `setTag` to `readPath`, separated by `/`.

React Properties

Property	Type	Default	Use
readPath	string	•	Path to read selection from
setTag	string	'default'	Name for local selection

DetailWrapper

Local Navigation

- *Description* (page 181)

Description

DetailWrapper determines the current objects URL from a base URL provided by the backend, and the routing params provided in the props. When mounted, or when the object URL changes, an action to fetch the current object from the REST API is dispatched.

The current object is provided as property `contextObject` to the single child.

This component is normally not used by itself, but is automatically employed by the frontend routing.

DataContainer

Local Navigation

- *Description* (page 181)

Description

One "data container" in an object header.

DataContainerField

Local Navigation

- *Description* (page 181)

Description

As single attribute in a data container. The field type determines how the value gets rendered.

DataContainerRow

Local Navigation

- *Description* (page 182)

Description

Component to render the data containers in the object header. This component assumes that it is rendered inside a `DisplayContextLoader`, that provides the `fields` and `values` properties.

DetailOutletRelationshipTable

extends React.Component

Local Navigation

- *Description* (page 182)

Description

A wrapper component for `RelationshipTable` intended for usage in a tab on a detail page.

DetailOutletRelationshipTreeTable

extends React.Component

Local Navigation

- *Description* (page 182)

Description

A wrapper component for `RelationshipTable` intended for usage in a tab on a detail page. Renders the table as a TreeTable.

DORelationshipTableWithContentView

Local Navigation

- *Description* (page 183)

Description

A wrapper component for `RelationshipTable` intended for usage in a tab on a detail page with content view on the side.

ObjectDetails

Local Navigation

Description

Renders the tabs on detail pages. Which tabs exist is defined as an outlet in the configuration.

Dialog

Local Navigation

Description

This module provides the base dialog component, on which both stock dialogs and custom dialogs are based.

Dialog Sizes

Dialogs may either be resizable or fixed size. The properties `sizes` and `size` (of which at most one should be specified), allow you to specify either an `Immutable.Map` with all sizes possible for this dialog or one size constant that represents the dialogs fixed size.

Available size values are:

- `Dialog.SIZE_SMALL`
- `Dialog.SIZE_MEDIUM`
- `Dialog.SIZE_LARGE`

If different sizes are provided, a button will be displayed in the dialog's titlebar, which allows to toggle the possible sizes of the dialog.

Keyboard Control

The Dialog Component defines two shortcuts it handles:

- When pressing Enter, the Function specified with the property `onAction` will be executed. This should correspond to the action that is executed when the main button action of the dialog will be executed.
- When pressing Escape the Function specified with the property `onHide` will be executed. This should correspond to the action when the user presses a Cancel or Close Button.

React Properties

Name	Type	Default	Description
buttons	node	-	The buttons displayed in the modals footer
dialogClass-Name	string	-	Custom CSS Class attached to the modal
hideFooter	bool	`false`	Footer will not be displayed. Use this for dialogs that will only display a cancel button.
fixedHeight	bool	`false`	If set, dialog will have a fixed height, relative to screen height
onAction	func	-	Executed when the user presses Enter.
onHide	func	-	Executed when the user presses the Close button or Escape key.
size	DialogProp-Types.size	-	If set the dialog will be displayed in this size and changing size is disabled.
sizes	Immutable.list	All three sizes	The available sizes for the dialog.
show	bool	`true`	If set, the dialog will be displayed.
showCancel	bool	`false`	If set, display a close button in the header.
title	string	-	Title of the dialog, displayed in the header

Examples

```
import {Dialog} from 'cs-web-components-base';

const SimpleDialog = props => (
    <Dialog.Dialog title="A Simple Dialog"
                   buttons={[<Dialog.Buttons.Ok
                               key="ok"
                               onClick={props.confirmAndHide} />,
                             <Dialog.Buttons.Cancel
                               key="cancel"
                               onClick={props.hide}/>]}
                   show={props.show}
                   size={Dialog.SIZE_MEDIUM}
                   showCancel onHide={props.hide}>
        {props.content}
    </Dialog.Dialog>
);
```

InputDialog

Local Navigation

- *Description* (page 185)

Description

ActionButton is a React Element, though it is not necessary to provide an onClick handler, as the local onClick-Handler will be provided by this component. But you may overwrite the Buttons title-property to set a custom tooltip text. It defaults to rendering an Ok-Button.

InputControl may be an arbitrary FormControl-element. It defaults to TextInput.

Alert

extends React.Component

Local Navigation

- *Description* (page 185)

Description

Error

extends React.Component

Local Navigation

- *Description* (page 185)

Description

Message

extends React.Component

Local Navigation

- *Description* (page 185)

Description

Notice

extends React.Component

Local Navigation

Description

YesNo

extends React.Component

Local Navigation

Description

This component displays a dialog providing a Yes/No choice. The additional property onConfirm should provide a callback that is – in addition to onHide – invoked, when the user pushes one of the two buttons. The function receives a boolean argument, which reflects the users choice.

Usage

```
import {Dialog, Console} from 'cs-web-components-base';

const SimpleYesNo = props => {
    const onConfirm = yes => {
        if (yes) {
            Console.log('Yes was pushed');
        } else {
            Console.log('No was pushed!');
        }
    };
    return (
        <Dialog.Dialog title="A Simple Choice"
                       show={props.show}
                       size={Dialog.SIZE_MEDIUM}
                       onConfirm={onConfirm}
                       onHide={props.hide}>
            {props.content}
        </Dialog.Dialog>
    );
};
```

YesNoCancel

extends React.Component

Local Navigation

Description

This component is similar to *YesNo* (page 186) but provides an additional cancel option, which bypasses the execution of the `onConfirm` callback.

SingleListSelection

extends React.Component

Local Navigation

Description

SingleSelection

extends React.Component

Local Navigation

Description

DynamicLibLoader

Local Navigation

Description

FACC component that realizes dynamic loading of JS files. The child function is called with a boolean argument, that is true iff the script is loaded. If more than one DynamicLibLoader instance is rendered with the same scriptUrl property, the library is loaded only once, and the setup funtion is also called only once.

React Properties

Property	Type	Default	Use
scriptUrl	string	-	The URL of the script to load
setup	function	-	An optional setup function. If set, it will be called exactly once after the script has finished loading.

ObjectFavoriteButton

Local Navigation

- *Description* (page 188)

Description

ObjectFavoriteButton renders an IconButton that signals whether the given contextObject is marked as a fovorite, and allows to toggle the favorite state by clicking on the icon.

CreateButton

Local Navigation

- *Description* (page 188)

Description

Button that invokes CDB_Create for the provided contextType when clicked.

Properties:

- contextType (required): the classname for the class to be created.
- onSuccess (optional): Callback to invoke when operation successfully completes. Overrides default behaviour.
- onFailure (optional): Callback to invoke when operation fails completes. Overrides default behaviour.

FormWithOperations

Local Navigation

- *Description* (page 188)

Description

This component displays the configured info mask for the provided context object. Furthermore operations that are available for the object will be displayed as buttons, above the form.

Required Properties:

- instanceName: A unique name identifying the component instance
- contextObject: The object to be displayed
- opContextName: The name of an operation contextname

Optional Properties:

- dialogNames: An object providing alternative dialogs for operations. This is optional.

DialogHooksContext

Local Navigation

Description

Context for dialog hooks: provides values and API to hook implementations.

Contents

Functions

applyBackendResult

Internal method: consume the backend's answer from calling hooks.

getBackendRequestData

Internal method: prepare data to POST to the backend when calling hooks.

isFieldChanged

Determine if the original change (disregarding changes from hooks that were already executed) included the given fieldname.

isHookApplicable

Check if a hook does apply to this change: the changed attribute name is configured, or the hook applies to all changes.

preventSubmit

Manage flags that may prevent the "Submit" action of the dialog.

FrontendDialog

Local Navigation

Description

Representation of a dialog to be shown from a hook function

PreSubmitDialog

extends React.Component

Local Navigation

Description

Shows a dialog as defined by a pre-submit hook.

Button

Local Navigation

Description

Form control button component

Property	Type	Default	Use
label	string	•	Button text
tooltip	string	•	Button tooltip
onClick	func	•	Callback for button click event

Calendar

Local Navigation

Description

This component can be used to enter a date. It also provides a date picker/calendar view for date selection. It has following properties in addition to properties of <*TextInput* (page 199)>:

React Properties

Property	Type	Default	Use
valueFormat	string	ISO 8601	Format of the value for data exchange
displayFormat	string	ISO 8601	Format to display the value
dateOnlyFormat	string	-	Format to display the value without time parts if the time parts are `00`
onDateChange	func	-	A callback fired when the date value gets changed. The value is formated to `valueFormat`.
allowExpression	boolean	-	Allow search expression like '>=01.08.2017 AND <=25.08.2017' See dateConverter
onInvalidInput	func	-	A callback fired when the input value changes but is invalid. The input value is passed to that function. In this case the `onDateChange` won't be called.
dateConverter	class	CalenderConverter	Expression convert and parse methods

Note: `onDateChange` will be called, if the date is changed via date picker, or when the input field loses focus after the value gets modified there.

CatalogField

Local Navigation

- *Description* (page 191)

Description

This component contains a text input field and a button to open a catalog, which let the user to select data from a table. The selection will be processed then on the server side and possible changes will be returned. The processing of the data is based on catalog configuration and its logic in CONTACT Elements. This component has following properties in addition to properties of <*TextInput* (page 199)>:

Property	Type	Default	Required	Use
textReadOnly	bool	•	•	Whether to set the input field as read-only
formData	Immutable.Map	•	•	Form data to be used to query catalog data entries, including the output fields the catalog should set
contextClass	string	•	•	Class of context objects for catalog data
activeLanguage	string	•	•	In which language should catalog data be loaded
selectURL	string	•	•	URL to submit the user selection, as result the changed fields and values would be returned as mapping
onCatalogChange	func	•	•	A callback fired when a catalog entry is selected
typeAheadURL	string	•	•	URL to load suggested entries supporting type ahead
catalogTableURL	string	•	yes	URL to load table data and configuration for displaying the catalog entries
proposalCatalogURL	string	•	•	URL to load table data and configuration for displaying the proposal table entries
proposalLabel	string	•	•	Proposal table title
userSettings	string	•	•	Additional user settings

ComboBox

Local Navigation

- *Description* (page 192)

Description

This component provides a drop-down list of data entries for selecting. It has following properties in addition to properties of <*TextInput* (page 199)>:

Property	Type	Default	Use
textReadOnly	bool	•	Whether to set the input field as readonly
items	Immutable.List	•	List of available entries
itemComponentClass	node	•	Optional component can be used to display an entry in the drop-down list
emptyItemClass	node	•	Optional compenent to handle rendering of empty values
onToggle	func	•	A callback fired when the drop-down list is toggled. `function(Boolean isOpen) {}`
onSelect	func	•	A callback fired when an entry is selected. `(eventKey: any, event: Object) => any`

ComboBoxCatalog

Local Navigation

- *Description* (page 193)

Description

This component enhances the <*ComboBox* (page 192)> to allow loading data entries from specified URL. The selection will be processed like the <*CatalogField* (page 191)>. It has following properties in addition to <*ComboBox* (page 192)>:

Property	Type	Default	Required	Use
itemsURL	string	•	•	URL to load entries for the drop-down list
formData	Immutable.Map	•	•	Form data to be used to query catalog data entries
contextClass	string	•	•	Class of context objects for catalog data
activeLanguage	string	•	•	In which language should catalog data be loaded
selectURL	string	•	•	URL to process the user selection, as result the changed fields and values would be returned as a mapping
onCatalogChange	func	•	•	A callback fired when a catalog entry is selected

GroupControl

Local Navigation

- *Description* (page 194)

Description

This component can be used to group related form controls. The primary component is supplemented by an additional expansion button, which allows the additional components to be displayed.

Property	Type	Default	Use
children	element	• •	Primary component and template for other components
others	array	•	List of property mappings for other components, every object must have a (unique) id
expanded	bool	false	Group initial expanded
iconSrc	string		URL to load the icon used for the expansion button
titleExpanded	string		Title for expanded component
titleCollapsed	string		Title for collapsed component

NumericEdit

Local Navigation

- *Description* (page 195)

Description

This component can be used to enter a numeric value. It can round the value up or down if specified. Following properties in addition to properties of <*TextInput* (page 199)> can be set:

Property	Type	Default	Use
decimalPlaces	number	•	Decimal places to display and round the value up or down
onValueChange	func	•	A callback fired when the numeric value gets changed.
minValue	number	•	Minimum allowed value
maxValue	number	•	Maximum allowed value
variableDecimal	bool	false	If true and decimalPlaces are undefined, the number of decimal places is calculated that the total value does not exceed 16 characters

Note: `onValueChange` will be called with the rounded value. `onChange` should only be used if the original user input is needed.

Due to the double-precision floating-point format, which javascript uses to represent a number, this component only accepts number inputs between -9007199254740991 and 9007199254740991. And that gives 15–17 significant decimal digits precision. It means if there are more than 15 digits, the precision can not be guaranteed.

Radio

Local Navigation

- *Description* (page 195)

Description

A `Radio` can be set in 2 states: `checked` or `unchecked`. It has one more property in addition to those defined in form_control_common_props:

Property	Type	Default	Use
checked	bool	•	Display the Radio as checked

ReactComponentControl

Local Navigation

- *Description* (page 196)

Description

The component `<ReactComponentControl>` allows custom controls. It has following properties:

Property	Type	Default	Use
componentName	string	•	The component that is used to render this control. The following properties are passed through: value, readOnly, label, tooltip, contentType, onValueChange
value	any	•	Data to be displayed
readOnly	bool	•	Control should be read only
label	string	•	Label of the control
tooltip	string	•	Tooltip of the control
contentType	string	•	Content type of the value
lines	number	•	Line number from mask configuration
onValueChange	func	•	Equivalent callback of `onChange`, called
contextObjects	Immutable.list	•	List of context objects
operationState	Immutable.map	•	Operation state

Slider

Local Navigation

- *Description* (page 197)

Description

The component `<Slider>` allows numeric inputs and has following properties in addition to those defined in form_control_common_props:

Property	Type	Default	Use
value	string	-	Current value for the slider
max	string	-	Biggest value for the slider
min	string	-	Smallest value for the slider
compact	bool	false	Display in compact mode without label and bottom margin
beforeControl	node	-	A component or a list of components to be placed before the input field
afterControl	node	-	A component or a list of components to be placed after the input field
onChange	func	-	Equivalent callback of `onInput`, called everytime the value changes
onValueChange	func	-	Called everytime this.state.value changes It's better to use this instead of onChange

Example:

```
import {FormControl} from 'cs-web-components-base';
...
// in render method of some component
render() {
    return (
        <FormControl.Slider
                onValueChange = { (value) => {
                    this.setState({
                        value: {value}
                    });
                }}
                min = "0"
                max = "10"
                value = "0"/>
}
```

Spinner

Local Navigation

Description

This component extends NumericEdit to show spinner buttons for increasing/decreasing values

Property	Type	Default	Use
step	number	1	The step in which to in-/decrease value when the respective button is pushed.
minValue	number	•	Minimum allowed value of Spinner
maxValue	number	•	Maximum allowed value of Spinner

TextArea

Local Navigation

- *Description* (page 198)

Description

The component `<TextArea>` allows textual inputs in multiple lines, works like a standard `<textarea>` HTML tag. It has following properties besides those defined in form_control_common_props:

Property	Type	Default	Use
value	any	•	Data to be displayed
compact	bool	false	Display in compact mode without label and bottom margin
rows	number	•	To extend the size of this component to bottom margin
beforeControl	node	•	A component or a list of components to be display specific lines of text, compare the `rows` attribute on `<textare>`
afterControl	node	•	A component or a list of components to be placed after the input field
onValueChange	func	•	Equivalent callback of `onChange`, called with the field value instead of the event

Example:

```
import {TextArea} from 'cs-web-components-base';
import {Button} from 'react-bootstrap';
...
// in render method of some component
render() {
    const myButton = <Button onClick={this.openSomeDialog}>Open Dialog</Button>;
    return (
```

```
        <TextArea label="Some Field"
                  value={this.state.someValue}
                  onChange={this.handleChangeSomeValue}
                  afterControl={myButton} />);
}
```

TextInput

Local Navigation

- *Description* (page 199)

Description

The component `<TextInput>` allows textual inputs and has following properties in addition to those defined in form_control_common_props:

Property	Type	Default	Use
value	any	-	Data to be displayed
compact	bool	false	Display in compact mode without label and bottom margin
beforeControl	node	-	A component or a list of components to be placed before the input field
afterControl	node	-	A component or a list of components to be placed after the input field
onValueChange	func	-	Equivalent callback of `onChange`, called with the field value instead of the event
onEnter	func	-	Callback that is invoked when the user pushes 'Enter'. Function will receive current value of the field as parameter.

Example:

```
import {TextInput} from 'cs-web-components-base';
import {Button} from 'react-bootstrap';
...
// in render method of some component
render() {
    const myButton = <Button onClick={this.openSomeDialog}>Open Dialog</Button>;
    return (
        <TextInput label="Some Field"
                   value={this.state.someValue}
                   onChange={this.handleChangeSomeValue}
                   afterControl={myButton} />);
}
```

TriStateCheckBox

Local Navigation

- *Description* (page 200)

Description

This component extends the <*CheckBox* (page 93)> with a third state: `indeterminate`. That means its state is neither `checked` nor `unchecked`. It has following properties in addition to those defined in form_control_common_props:

Property	Type	Default	Use
value	string, one of: • "0": unchecked • "1": checked • ""(empty): indeterminate	""	To display the check box in different states
onStateChange	func		A callback fired when state of check box is changed. `(nextState) => any`

Note: `onStateChange` will be called with the new state if user clicks on the check box. The states will be set in this order: `indetermined`, `unchecked`, `checked` and back to `indetermined`.

FixedSidebarWithEditor

Local Navigation

Description

This component is an operation providing extension of FixedSidebar.

SplitterLayout

Local Navigation

Description

This component serves as a container for horizontally or vertically split panes.

There are 2 types of panes: primary and secondary panes. Primary panes are automatically resized if the container size gets changed, while secondary panes keep their sizes. Secondary panes can have a splitter bar to allow resizing(by dragging) and collapsing(by clicking) manually.

Examples

```
<SplitterLayout>
    <SplitterLayout.SecondaryPane>
      <div>left pane, resizable, collapsible</div>
   </<SplitterLayout.SecondaryPane>>
   <SplitterLayout.PrimaryPane>
     <div>main area, resized automatically</div>
   </SplitterLayout.PrimaryPane>
    <SplitterLayout.SecondaryPane>
      <div>right pane, resizable, collapsible</div>
   </<SplitterLayout.SecondaryPane>>
</SplitterLayout>
```

React Properties

SplitterLayout

Name	Type	Default	Description
vertical	bool	false	Whether the panes are layed out vertically

SplitterLayout.PrimaryPane

Name	Type	Default	Description
fixSizing	bool	true	Whether to force the content taking exact size of the pane area (and being cut off by overflow)

SplitterLayout.SecondaryPane

Name	Type	Default	Description
fixSizing	bool	true	Whether to force the content taking exact size of the pane area (and being cut off by overflow)
resizable	bool	true	Whether to allow resizing the pane manually
collapsible	bool	true	Whether to allow collapsing the pane manually
defaultSize	string	10rem	Default size for the pane
hidingBreakPoint	number	-	Size in px, if specified, the pane will be collapsed automatically if its size is less than this value
initCollapsed	bool	false	Whether to collapse the pane initially
onChange	func	-	Callback function, which will be called after resizing or collapsing the pane, with an object {collapsed, size} as argument

Bar

Local Navigation

- *Description* (page 202)

Description

Renders the TabTitles and an overflow menu that lists every tab.

Name	Type	Default	Description
activeKey	string/number	-	Event key of the currently active tab.
onSelectFromMenu	func	-	callBack for when a tab is selected from the overflow menu.
addControlLabel	string	-	If passed, renders an additional TabTitle with `eventKey` === EVENT_KEY_ADD
diabled	bool	-	Deactivates add control and menu (not tabs, these must be deactivated manually).

VerticalBlockLayout

Local Navigation

- *Description* (page 202)

Description

Layout that renders 1, 2, or 3 vertical blocks and behaves as following:

- each of the children components will be rendered into a block
- if there are more than 3 children components passed in, only the first three ones get rendered
- in case the height of this layout component changes, it will change the height of the last block to fit in, and fix the heights of the other blocks

This component will hide the overflowed content and not show scrollbar. It is the responsibility of the child component to handle its size and overflow.

BannerContainer

extends React.Component

Local Navigation

- *Description* (page 203)

Description

This container enables components to display messages sent to the user via the messages framework as banners.

The component should be rendered into a container at the location at which the banners should be displayed, usually on top of the component that is subject of the message. As a website may contain multiple BannerContainer components an `id` should be provided to each instance. This `id` will be used in the createBanner message factory to specify which container should display the banner. If an id is used multiple times, messages will be displayed in all containers with that id.

Name	Type	Default	Description
id	string	-	The id of the bannercontainer.

AlignedOverlay

Local Navigation

Description

Extended Overlay that handles layouting. A layouting function needs to be provided to calculate the styling/positioning of the overlay. The overlay will then be positioned in relation to the property `parentRef`. Exact positioning is determined by `layoutFunc`.

Layout Function

Functions provided as a layouting function to this component must accept the following parameters:

- **layoutConfig**: Parameters that depend on the layout function used
- **horizontalAlign**: The horizontal alignment of the overlay
- **verticalAlign**: The vertical alignment of the overlay
- **parentRect**: An object with the absolute position and the dimensions of the parent component.
- **overlayRect**: An object with the dimensions of the content of the overlay.

The return value should be an object, containing at least properties `top` and `left`, which specify the absolute position of the overlay in `document.body`. Since the return value is passed as the element-style of the overlay diff, it is also possible to use CSS for advanced options, like enabling scrolling.

Children

As children a single element should be passed. This element will be rendered as the content of the overlay.

React Properties

Name	Type	Description
parentRef	element	The element to which the overlay will be aligned.
onBlur	function	Sets an `onBlur` handler on the overlay div This is useful if you want to close the div when it loses focus. Note that for this to work, you need to initially request the focus on the component.
setOverlayRef	function	Callback to receive a reference to the overlay DOM element, e.g., to request the focus, when it is rendered.
setContentRef	function	Callback to receive a reference to the content of the overlay.
visible	boolean	The overlay is only rendered if this flag is `true`
className	string	Append additional classes to the overlay's class attribute.
useTimer	boolean	If set, a timer will check for responsive changes and trigger relayouting if necessary
layoutFunc	function	A function that calculates the position/layout of the overlay, based on its content, the window dimensions and an optional container
horizontalAlign	HorizontalAlignmentType	The horizontal alignment of the component. Will be passed as a parameter to the layout function.
verticalAlign	VerticalAlignmentType	The vertical alignment of the component. Will be passed as a parameter to the layout function.

Controlled

Local Navigation

Description

AlignedOverlay, extended for closing the overlay by user interaction.

React Properties

In addition to those defined for *AlignedOverlay* (page 203) this component receives the following properties

Name	Type	Description
onHide	function	This handler is called when the user presses `Escape` on the overlay or it loses focus. Use to modify visibility state of component.

See *Roll your own* (page 125) for details.

Bubble

Local Navigation

Description

Provides an overlay that is layouted relative to an anchor component (the component, to which the bubble will be attached), and styled like a speech bubble.

When rendering, a Bubble should get two child components: The first will be used as the anchor component, while the second will be used as the content of the bubble.

When the Component is used in a context, where the anchor may not be in the visible viewport of its containers (e.g. it is rendered inside a container like a scrollpane where it ends up in the overflow), you may pass in a reference `containerRef`. When the anchor component is not in the visible area of the component, the overlay will be hidden.

Members

- cs-web-components-base-overlays.Bubble.Dropdown

React Properties

Name	Type	Description
bubbleStyle	string	Semantic Style of the Bubble Component
verticalAlign	VerticalAlignmentType	How the Bubble should be aligned in relation to its parent component.
containerRef	object	Reference to a containing component that may cover the anchor component.

Example

```
function(props) {
  return (
    <Bubble verticalAlign={VerticalAlignment.BOTTOM}
            bubbleStyle={'warning'}
            {...props}>
      <IconButton buttonStyle="warning"
                  style={{
                    marginTop: '50vh',
                    marginLeft: '50vw'
                  }}
                  iconName={"csweb_option_horizontal"}
                  title={"Hello World"}
                  size={'sm'} />
      <div style={{margin: '5px'}}>
```

```
        <h5>Hello World</h5>
          <ul>
            <li>This is an example</li>
            <li>for a bubble over a button</li>
          </ul>
      </div>
    </Bubble>
  );
}
```

ContextMenu

Local Navigation

Description

An overlay component to display context menus.

Example

```
import {Overlays} from 'cs-web-components-base';

const {ContextMenu} = Overlays;

class ContextMenuContainer extends React.Component {
  constructor(props) {
    super(props);
    this.onContextMenu = this.onContextMenu.bind(this);
  }

  onContextMenu(evt) {
    this._contextMenu.open(evt.pageX, evt.pageY);
    evt.preventDefault();
  }

  render() {
    return (
      <div onContextMenu={this.onContextMenu}>
        <ContextMenu ref={c => {this._contextMenu = c}}>
          {getMenuItems()}
        </ContextMenu>
        {"Hello Worlds"}
      </div>
    );
  }
}
```

Dropdown

Local Navigation

Description

A general component for rendering dropdowns.

Alignment on this component is as follows: Vertical Alignment dictates, wether the dropdown should be rendered on top or on the bottom side of the anchor component. Horizontal alignment dictates on which side the component should be aligned flush with the parent component. For example, when aligned left, both the overlay and the anchors absolute x position will be equal. If aligned right, both (element.absx + element.width) will be equal for anchor and overlay.

The layouting algorithm used for this type of overlay takes into account the space available in the browser's viewport: If the content of the dropdown does not fit the available space horizontally or vertically, it will try the following:

- If there is enough space when using a different alignment mode (e.g. horizontal bottom instead of horizontal top), it will used that.
- If it does not fit in the other mode, it will use a scrollbar along that axis of alignment and use the original alignment option.

Passing Children as FACC

To enable both the layouting component, as well as the component rendered inside the dropdown to control the state of the dropdown, these components are passed to Dropdown via `props.children` inside a function.

This function receives the following parameters:

```
{
  onShow: func,
  onHide: func,
  onToggle: func,
},
visible: boolean
```

When called, `onShow` opens the dropdown if it is closed, `onHide` closes the dropdown if it is open and `onToggle` toggles its visibility state.

The parameter `visible` reflects the current state of the dropdown. An example is found at the end of the section.

React Properties

Name	Type	Description
containerRef	object	Optional reference to scroll container
toggleRef	function	Called to set a reference to the toggle component.
horizontalAlign	HorizontalAlignmentType	Horizontal Alignment of Dropdown
verticalAlign	VerticalAlignmentType	Vertical Alignment of Dropdown

Example

```
function MyComponent(props){
  return (
    <Dropdown>
      {
        ({onToggle, onHide}, visible) => {
          // Button should look pressed, when dropdown is open.
          const clsNames = classNames(
            {
              [className]: Boolean(className),
              [prefixNS('open')]: visible,
            },
            prefixNS('dropdown-toggle')
          );
          return ([
            <IconButton key={'toggle'}
                        className={clsNames}
                        {...others}
                        onClick={onToggle}/>,
            <div>I am a dropdown</div>
          ]);
        }
      }
    </Dropdown>
  );
}
```

Static

Local Navigation

- *Description* (page 208)

Description

A component that replicates the Anchor/Overlay interface of Dropdown, but removes the visibility state management.

Menu

Local Navigation

- *Description* (page 208)
- *Properties* (page 209)

Description

A container for MenuItems.

This component controls the communication between its *MenuItem* (page 209) children, and the surrounding overlay.

Properties

Name	Type	Description
onHide	function	Handler for hiding menu when item is clicked.
children	element	Content of the menu.

MenuDivider

Local Navigation

- *Description* (page 209)

Description

A divider for use in Menus. Renders a thin line between the items before and after it.

MenuHeader

Local Navigation

- *Description* (page 209)

Description

A headline for use in Menus. Renders the string provided by property `label`.

MenuItem

Local Navigation

- *Description* (page 209)
- *Properties* (page 210)

Description

Represents an entry in a menu.

Properties

Name	Type	Description
onSelect	function	Invoked when item is selected
onClick	function	Alias for `onSelect`
children	element	Content of Item

Overlay

Local Navigation

- *Description* (page 210)
- *Children* (page 210)
- *React Properties* (page 210)

Description

This provides a basic overlay implementation, without any layouting functionality. It simply renders its child component in a div, positioned absolutely on the document by using `React.createPortal`.

Children

As children a single element should be passed. This element will be rendered as the content of the overlay.

React Properties

Name	Type	Description
onBlur	function	Sets an `onBlur` handler on the overlay div This is useful if you want to close the div when it loses focus. Note that for this to work, you need to initially request the focus on the component.
onFocusOut	function	Sets an onFocusOut handler on the overlay div. Note that this uses DOM Events instead of React Events (as of 16.3 React does not support focusin/focusout events).
setOverlayRef	function	Callback to receive a reference to the overlay DOM element, e.g., to request the focus, when it is rendered.
setContentRef	function	Callback to receive a reference to the content of the overlay.
visible	boolean	The overlay is only rendered if this flag is `true`
positionStyle	object	This allows to set the element style of the overlay div, use this to absolutely position the div on the page
className	string	Append additional classes to the overlay's class attribute.

Controlled

Local Navigation

- *Description* (page 211)
- *React Properties* (page 211)

Description

Overlay, extended for closing the overlay by user interaction.

React Properties

In addition to those defined for *Overlay* (page 210) this component receives the following properties:

Name	Type	Description
onHide	function	This handler is called when the user presses `Escape` on the overlay or it loses focus. Use to modify visibility state of component.

See *Roll your own* (page 125) for details.

TypeAhead

Local Navigation

- *Description* (page 211)

Description

This class provides a *Dropdown* (page 206)-based component to add type-ahead functionality to input fields.

Key differences to standard Dropdowns are:

- A different layout function. You can not specify a horizontal alignment, instead the components width will always be the same as the width of the toggle.
- The grabFocus property is set to true, as text input fields will usually want to keep the focus on the input field itself.

For properties and usage info, see *Dropdown* (page 206).

RichTextEditor

Local Navigation

- *Description* (page 212)

Description

RichTextEditor provides the possibility to enter enhanced text with properties of color, size, etc. It receives the following properties:

Name	Type	Default	Description
Content	string	-	The content as stringified json to be shown in the editor.
readOnly	bool	false	Specifies if the editor is shown in readonly mode
styleMap	map	{}	Additional styles like font color, sizes, etc.
additionalButtons	list	[]	Additional buttons for new entities, styles, etc.
additionalDecorators	list	[]	Additional decorators for additional entities
height	number	•	Specifies the height of the editor. Normally that is 100vh - 34 px. In case height is set to 0 the styling is removed.
containerRef	object	•	When using the RichTextEditor component inside a scrollable container provide the ref to this container here.

ColumnContent

Local Navigation

Description

Default content renderer component used by ./Table.jsx. +——————+——————+——————+——————————————— | Property | Type | Default | Use | +==================+==============+==============+===================== | value | any | - | Value of the content to be rendered. If | | | | | Value is a map that contains the key | | | | | `text` this text is taken as table | | | | | content. If value contains the key `link` | | | | | the value of `link` is a map that has to | | | | | contain the key `to` and can contain the | | | | | key `title`. The content will be rendered | | | | | as link for this case. | | | | | If value contains the key `icon` the value| | | | of `icon` is a map that conatins either | | | | | `src` with the icon url or `name`. The | | | | | icon will be rendered automatically. | +——————+——————+——————+———————————————+ | column | map | - | Information about the column. Special keys | +——————+——————+——————+———————————————+ | row | map | - | The complete row as returned by the REST-API| +——————+——————+——————+———————————————+

The `column` map might have this keys:

- `getFormattedValue`: A function that gets the value and should return the user representation of the value.

- `sortFunction`: A function that compares to values of the column. The function will be called in the way `cmp(v1, v2)` where v1 and v2 are lists of two elements. The first element is the value to be compared. The second contains all entries of the row where the value resides.
- `getLink`: A function that will be called with the value, row and column as parameter and should return the url of the link.

Fixed

extends React.Component

Local Navigation

- *Description* (page 213)

Description

Usage:

ToolbarMenu

Local Navigation

- *Description* (page 213)

Description

Usage:

```
class MenuExample extends React.Component {
    render() {
        return (
            <div onClick={()=>console.log('clicking')}>test menu</div>
        );
    }
};

const myMenu = {
    toolbarMenu: [MenuExample]
};

// Can be used as <SpecialTable />
const SpecialTable = Table.Manager({
    toolbarMenu: [myMenu]
});
```

RestTree

Local Navigation

Description

RestTree is a subclass of `Tree`, that implements a tree view based on the traversal of references using the generic REST API.

Resolving references

TreeDownReferences are specified as a JSON object containing, for each class that may appear in the tree (as it is the root node, or by following treedownreferences) the relationship to follow. An example of the syntax of treeDownReferences is given in *Example Usage* (page 214).

When a node in the tree should be expanded, treeDownReferences is searched for the class of the object and each of its base classes. Objects that are related via the relations found that way, will be displayed as child nodes.

`checkChildren` may be set to `true` to prefetch Children of not yet expanded nodes. This enables a tree to know if a tree node is a leaf before it is expanded. Note that this may impact performance for large structures.

Rendering the title

By default the objects system:description attribute is rendered as the trees title. If the property titleAttribute is specified, this attribute will be rendered as the title instead. If property title is specified this will be rendered instead of an attribute.

If a custom title renderer component is specified, `titleAttribute` and `title` properties will be ignored.

Example Usage

Javascript:

Configuration:

```
{
  "name": "cs-web-components-base-ContextObjectSetter",
  "properties": {"setPath": "orgTree"},
  "children": [
    {
      "name": "cs-web-components-base-RestTree",
      "properties": {
        "treeDownReferences": {
          "cdb_organization": [
            {
              "id": "Suborganizations",
              "viewName": "complete-target"
            },
            {
              "id": "Cdb_organization_to_cdb_person",
```

```
            "viewName": "complete-target"
          }
        ],
        "angestellter": [
          {
            "id": "someRelation",
            "viewName": "complete-target"
          }
        ]
      }
    }
  }
]
}
```

React Properties

Property	Type	Default	Use
contextObject	Object	•	The REST API object that is the root of the tree
titleAttribute	string	•	Specify an object attribute to be rendered as title.
treeDownReferences	Object	•	Specifies the references that should be followed
onSelect	function	•	Callback function that will be invoked when a tree node is selected
checkChildren	bool	•	If set to true prefetch children of children to decide if expander should be displayed

BackendSearchAdapter

Local Navigation

- *Description* (page 215)

Description

Implements a search adapter that uses a backend based search. On querying for the next result, the algorithm starts to load deferred child nodes top down depth first, until it encounters the next match.

SimpleSearchAdapter

Local Navigation

- *Description* (page 216)

Description

Implements a simple search on a Tree component without using deferred loading. Using this method, the search results are limited to the nodes that have already been loaded.

StructureTree

Local Navigation

Description

Tree class that loads and displays classic structures. It may be used inside a ContextObjectSetter to set the contextObject of other components according to its selection. Additionally, parameter `structureName` should be set to the name of the structure to be displayed.

Example Usage

The following configuration expects an `organization` object and displays the structure CDB_Organization, associated with that object.

```
{
   "name": "cs-web-components-base-ContextObjectSetter",
   "properties": {"setPath": "orgTree"},
   "children": [
         {
            "name": "cs-web-components-base-StructureTree",
            "properties": {"structureName": "CDB_Organization"}
         }
   ]
}
```

React Properties

Property	Type	Default	Use
contextObject	Object	•	The REST API object that is the root of the tree
onSelect	function	•	Callback function that will be invoked when a tree node is selected
structureName	string	•	The name of the structure to be displayed

Toolbar

Local Navigation

Description

Types

Search Widget:

```
{
  type: "search"
  adapter: SearchAdapter instance
  mode: SEARCH_MODE_TYPE_AHEAD || SEARCH_MODE_CONFIRM
}
```

Buttons:

```
{
  type: "button"
  icon: string
  func: function
}

{
  type: "button",
  icon: "euro",
  func: () => console.log("Hello World")
}
```

Menu:

```
{
  type: "menu"
  icon: string
  conf: [
     {
       type: "item"
       icon: string
       title: string
       func: function
     }
     ...
  ]
}
```

cs-web-components-externals

cs-web-components-pdf

PDFViewer

Local Navigation

Description

A plugin component to render a PDF file. The user can select which page to show, and can select a scale factor.

React Properties

Property	Type	Default	Use
url	string	-	The URL of the PDF file to render
filename	string	-	The filename of the image file
scaleToFit	bool	true	Fit scale to available width
minified	bool	false	Show fewer controls
toolbar	node	null	Additional toolbar to show

cs-web-components-storybook

detail-wrapper

Local Navigation

Description

Wraps a React component, so that it can be used as the detailComponent in a generic frame.

DetailWrapper

Local Navigation

Description

DetailWrapper determines the current objects URL from a base URL provided by the backend, and the routing params provided in the props. When mounted, or when the object URL changes, an action to fetch the current object from the REST API is dispatched.

The current object is provided as property `contextObject` to the single child.

This component is normally not used by itself, but is automatically employed by the frontend routing.

cs-web-components-theme

cs-web-dashboard

actions

Local Navigation

Description

Contents

Functions

setSettings

```
setSettings(item: Immutable.Map, settings: Immutable.Map)
```

Takes:

- *item*: The item represented by the dashboard element
- *settings*: The data for the element

Redux action method to store the settings for a dashboard element.

DashboardItem

Local Navigation

Description

A DashboardItem must be used by every widget implementation to render the outer frame of an element. The react children of the DashboardItem will be rendered as the element's body.

React Properties

Property	Type	Default	Use
item	Immutable.Map	•	Data to be displayed in the element
title	string	•	A string to be rendered as the element's title
configCallback	function	•	Configuration callback to store settings
appLink	string	•	A string containing an application url. If specified a button in the upright corner will be shown to this application.
appNewWindow	bool	false	Specifies wether or not to open the linked application in a new tab / window.

The `item` holds some system defined properties for use by the framework, and a key `settings` that is also an Immutable.Map, and contains the data for a specific element. This property is passed to the widget implementation by the framework, and must be forwarded as is to the DashboardItem.

If `configCallback` is present, the DashboardItem will render a configuration button in the element's header that will call this function. The function should display some UI to configure the element, and finally call the Redux action method `setSettings` to store the configuration.

ObjectListWidget

Local Navigation

- *Description* (page 220)

Description

Generic widget that displays a list of objects either as a simple bootstrap table, when placed in a small column, or as a configured table otherwise. This must be used inside a component that supplies the URL to query for data. NOTE: the collectionURL property must include the "_as_table" query parameter!

cs-web-search

CHAPTER 14

Glossary

Resource A single file that is accessible from the frontend through an HTTP request. Typically JavaScript, CSS, image files etc.

Library A set of resources that are used together. Libraries have a name and a version, and possibly dependencies on other libraries.

Component A named artifact in JavaScript code, mostly used for React components. Components are contained in libraries. They can use other components, and so imply a dependency between the corresponding libraries.

Application

Web Application From a users perspective, this is an HTML page visible through a web browser, that represents some useful functionality. From a technical perspective, an application is represented by an HTML document, and all the JavaScript libraries that are loaded by this document.

JSX A JavaScript extension that is used to implement React components.

HOC

Higher Order Component A Higher Order Component is a function that takes a React Component as parameter and returns a React Component. A generic signature is

```
hoc(component: ReactComponent): ReactComponent
```

List of Figures

List of Tables

Python Module Index

w

Index

H

I

J

L

O

P

R

S

U

W