Fachhochschule Nordwestschweiz
Hochschule für Technik

Component-Architecture for Loosely-Coupled Responsive Webapplications

Architecture documentation

within the scope of the master thesis

Author:
Stefan Florian Röthlisberger

Advisor:
Prof. Martin Kropp

Contents

Glossary

Bower is a package manager for the web. The dependencies are defined in bower.json and can be installed with the commandline tool bower (http://bower.io).

DOM Document Object Model: A way to refer to HTML elements as objects.

Web Components Web components is an umbrella term that contains the following features: HTML Templating, HTML Imports, Shadow DOM and Custom Elements (http://webcomponents.org).

1 Introduction

This document describes the architecture of the developed application. It is not the final documentation and presents the architecture in the as-is state in the first week of February 2015. This document has been written solely for the attestation of the first semester of the master thesis.

2 Overview

2.1 Libraries

The following third-party libraries haven been used in the developed application:

- RequireJS: For Asynchronous Module Definition (AMD) Javascript modules. Provides a pattern to define modules that provide lazy loading of modules, dependency injection of other modules and encapsulation.
- Polymer: For defining custom HTML elements (Web Components)
- Polymer/core-*: Several existing custom elements defined by the polymer project. They all use the core-* prefix. Not to be confused with the AMD modules of the framework that start with the same prefix.
- interact.js: Drag&Drop
- javascript-route-matcher: Matcher for URL patterns (e.g. /project/:id where id is an extractable variable)
- eventEmitter: To emit events and register event-listeners (addOnceListener, emitEvent)
- heir: Small library with helper-functions for inheritance

2.2 Application structure

The structure of the application does not follow a traditional MVC approach. Instead it is composed of nested HTML custom elements that encapsulate the functionality they have. For example a visual custom element for a form knows how to handle the information entered itself and does not need a controller class in some unapparent directory somewhere in the application. The visual interface (HTML) and the code handling it are in the same place. This also means that the code responsible for the element's behavior is encapsulated and does not affect anything else.

The visual presentation of the application uses the concept of workspaces. A workspace denotes the combination of a layout, one or more widgets that may use

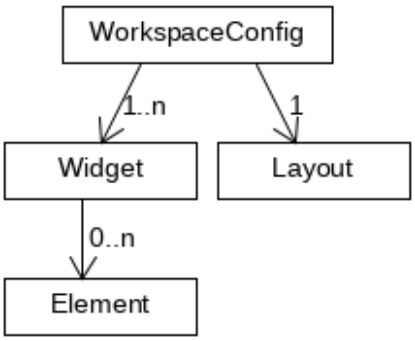

Figure 1: Building blocks of the application structure

elements and a URL-path. Figure 1 shows the building blocks of a workspace. The root node of the tree is the workspace-configuration that defines the workspace. The widgets are placed in the layout and use elements. The elements are reusable custom HTML elements that provide services like access to a REST API or visual elements like a form.

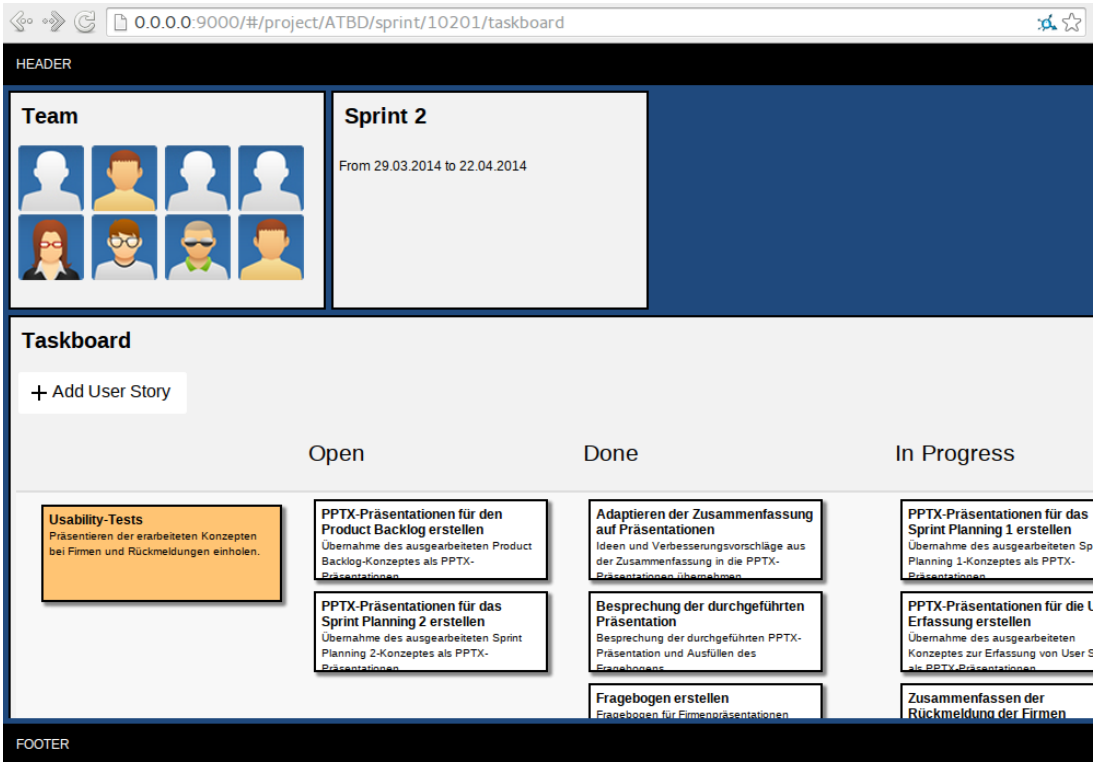

Figure 2: Workspace using a layout with multiple widgets

As an example of the structure, Figure 2 shows a workspace using a layout that has multiple widgets. In this case, three widget are available, with the Taskboard-widget as the most prominent one. As an example of elements, all avatars in the Teamview-widget as well as all user-story- and task-cards in the Taskboard-widget are custom elements.

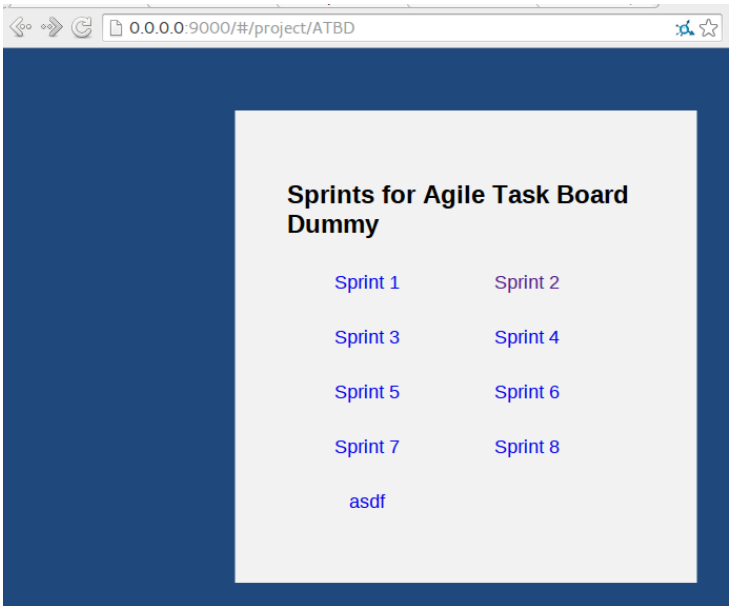

Figure 3: Workspace using a layout with a single widget

2.3 File-structure

The file-structure of the application has a *bower_components* directory, that contains all the third-party libraries managed by Bower. The *index.html* file is the default HTML-file served by webservers if no specific file has been provided in the URL. Thus, it serves as the entry point for the application. It loads all global includes like the global stylesheet definitions in *main.css*, the Javascript-files *app.js* and *core.js* as well as third-party libraries and custom elements.

```
/app/
- bower_components/
- elements/
- layouts/
- widgets/
- workspace-configs/
- app.js
- core.js
- index.html
- main.css
```

Listing 2.1: The applications file structure

The remaining four directories that represent the application structure are described in detail in the following sections.

2.3.1 Workspace-Config

The workspace-configuration (see Listing 2.2) is the configuration-file that is loaded first and defines the workspace's dependencies like the layout and the widgets as well as its path. The path is an URL-pattern that a particular workspace has. It is the mandatory direct link to a workspace and is actually the fragment identifier of the URL. That is the last part an URL can have and is the string after the '#'-symbol. The parts in the path prefixed with a colon are variables. For example the path in the Listing 2.2 has two variables: *projectId* and *sprintId*. This workspaces direct URL would be *http://example.com/path/to/app/dir/#/project/5/sprint/7/sp2* The path must be unique and should be chosen to logically represent the applications structure. It also must contain the variables needed by the widgets in the workspace.

```
{
  "name": "Sprint Planning 2 Workspace",
  "path": "/project/:projectId/sprint/:sprintId/sp2",
  "layout": "touchwall",
  "mainWorkspace": "sprintplanning2",
  "widgets": {
    "teamview": "widget1",
    "sprintinfo": "widget2"
  }
}
```

Listing 2.2: Workspace-configuration

The *mainWorkspace* property is mandatory and defines the most prominent widget in the respective layout. The additional widgets under *widgets* are optional and are defined with the name of the widget as key and ID of the div-element the widget is to be inserted in as value. So the *touchwall*-layout in Listing 2.2 has a div-element with the following definition: * *.

The workspace-configs are JSON files located in the *workspace-configs* directory. For each workspace-config to be available to the application, the name of the config-file must be added to the *workspace* property in the *config* module in *app.js*.

2.3.2 Layouts

Layouts are located under the *layout* directory and are polymer-elements extending the frameworks *app-layout* element. The layout names must adhere to *workspace-layout-X*, where X is the name of the layout and the name of the HTML-file it is defined in (e.g. touchwall.html -> *workspace-layout-touchwall*).

2.3.3 Widgets

Each widget has its own directory under *widgets*. For example in Listing 2.3 the *sprint-planning2* widget. Each widget has a *widget.json*, that defines the widget and an *index.html*, that contains the actual code.

```
widgets/
-taskboard/
– index.html
– widget.json
-sprintplanning2/
– index.html
– widget.json
```

Listing 2.3: File structure of widgets

Each widget has a *name* and needs to define the used dependencies under *elements* [1]. Those dependencies are the names of the HTML-files found under *elements* (e.g. ads-rest.html -> ads-rest).

```
{
  "name": "Sprint Planning 2 widget",
  "description": "",
  "elements": [
    "ads-rest",
    "ads-taskcard",
    "ads-userstorycard",
    "ads-form"
  ]
}
```

Listing 2.4: Widget configuration

In the Listing 2.4, the widget uses elements out of the four HTML-files defined under *elements*. For example the *ads-rest* import defines a dozen elements to access the Agile Data Service REST API like *ads-rest-task*.

The content of a widget must adhere to the format in Listing 2.5 to be included by the framework. The framework selects the element *template* with the class *content* to be included in the defined spot in the layout, as configured in the workspace-config. The HTML file may contain custom element definitions like polymer elements. In the

[1] It must be noted that those element-dependencies are not checked because that would require the loader to know all custom elements in all the HTML-files in the directory elements. It would also require that the *index.html* of a widget is parsed and scanned for tags that are used but not in any of the specified elements in *widget.json*.

example the content of the widget is a custom element with the name *ads-teamview*.

```
<template class="content">
  <ads-teamview></ads-teamview>
</template>
```

Listing 2.5: Declaration of the widget content

2.3.4 Elements

Elements are reusable custom HTML elements and are found in the directory *elements*. In the developed application, polymer has been used to create custom elements. They are referenced by widgets and are defined in HTML-files. Currently there are two prefixes used to define the elements:

- app-*: Elements of the framework.
- ads-*: Elements specific to the actual application

The name of a custom element must contain a minus '-' character [2] so that the custom elements do not clash with current and future native HTML elements.

2.4 Web Components

The application makes heavy use of new HTML5 features, especially the technologies included in the umbrella term Web Components:

- HTML Templating: Declaring inert DOM subtrees in HTML and munipulating them to instantiate document fragments with identical contents.
- Shadow DOM: Establishing and maintaining functional boundaries between DOM trees. Provides HTML and CSS encapsulation.
- Custom Elements: Define and use new types of DOM elements.
- HTML Imports: Include and reuse HTML documents in other HTML documents.

The first three in the listing above are used implicitly by polymer when creating a polymer-element. HTML imports are used in the applications main *index.html* to include the frameworks app-* elements as well as third-party polymer-elements (prefixed with core-*).

[2] https://w3c.github.io/webcomponents/spec/custom/#concepts

3 Framework

The framework is the core of the application and handles the bootstrapping. The application is started in *app.js*. It is composed of three main modules (as shown in Figure 4) defined in *core.js*:

- core-loader: Loads the workspace-config and all dependent parts like the layout, the widgets and the elements.
- core-router: Just a wrapper for the modules, delegates to the app-router element.
- core-dnd: Abstracts the interact.js library and provides drag&drop support.

It also uses and provides several custom elements:

- app-globals: Provides access to global variables like the route-parameters (the variables in the workspace-config's path property)
- app-layout: Base element for all layouts.
- app-router: Handles URL fragment identifier changes and the mapping between the fragment identifier and the workspace.

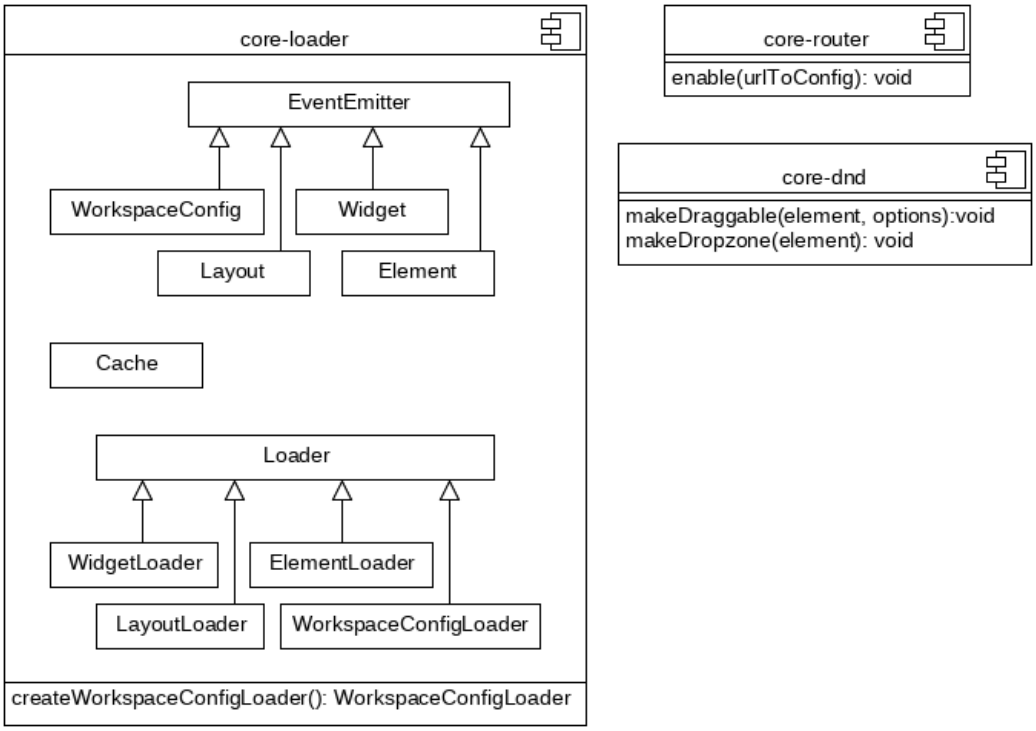

Figure 4: Framework overview

3.1 core-loader

The *core-loader* module (see Figure 4) consists of classes representing the four different building blocks a workspace is made of and loader-classes for each of them. The four representing classes inherit from *EventEmitter*, the class of the third-party library *eventEmitter*. The two methods *addOnceListener* and *emitEvent* are used during the bootstrapping of the specific instance and will be explained later. The loader-classes inherit from the class *Loader*, which provides common functions used by all loaders. The module exposes just one method that returns a new *WorkspaceConfigLoader* instance.

For each of the four workspace building blocks, there is a loader-class that loads the corresponding class (see Figure 5). All loader-classes inherit from *Loader* that has two class-variables *cache* and *loading*. Both are *Cache* objects that store the workspace building blocks that have been loaded or are being loaded respectively. The class also provides functions to load JSON and HTML files. Both functions take an URL and the caller supplies a callback function that is called once the JSON or HTML file has been loaded. This is due to the asynchronous nature of Javascript and allows the calling code to continue and handle the resource once the time-consuming network operation has been completed.

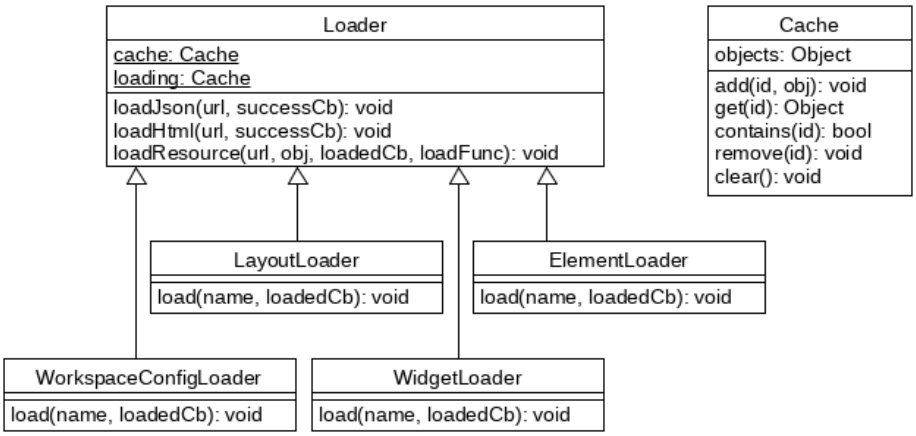

Figure 5: Loaders

The *loadResource*-function is called by all *load*-functions of the subclasses to load the workspace building block. The *url*-parameter specifies the URL of the resource, the *obj*-parameter is a new instance of a workspace building block (e.g. *Widget*), *loadedCb* is the function that is called if the workspace building block has been loaded. The *loadFunc* is the function supplied by the subclass that actually loads the workspace building block. Because all function-calls involving network operations are asynchronous, when a workspace is loaded, the building blocks can be in three states. They either have

been loaded, are being loaded or have not been loaded yet. Loaded building blocks are stored in the Cache *cache*, the loading building blocks are in the Cache *loading*. Only if a building block is in neither Cache, the supplied *loadFunc*-function is called. The aforementioned functions *addOnceListener* and *emitEvent* are called in the *loadResource*-function. If a workspace building block already has been loaded and is in the *cache* Cache, the supplied *loadedCb*-function is called directly. If the building block is currently being loaded and is in the *loading* Cache, a callback is registered with *addOnceListener*, that calls the *loadedCb*-function once the event '*loaded*' has been emitted. If the building block needs to be loaded, it is, with the supplied *loadFunc*-function. This function also takes a function as argument that is called after the subclass has loaded all its dependencies. In this function the *loadResource*-function handles the caches and calls the *loadedCb*-function. For all the others which caught the building block in the loading state, get a callback via *emitEvent* with the event '*loaded*' to their callback registered with *addOnceListener*.

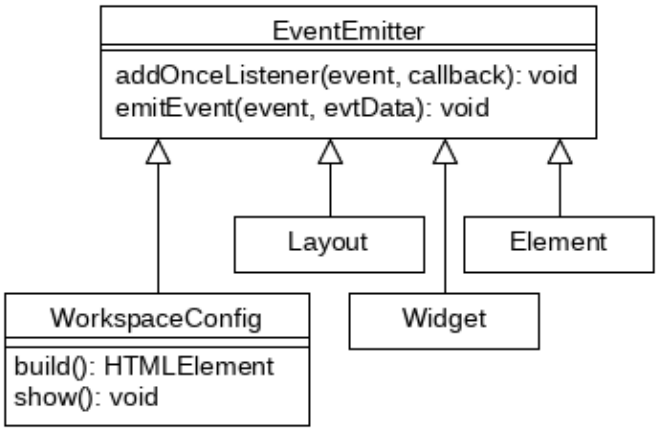

Figure 6: Workspace building blocks

The WorkspaceConfig-class has two functions *build* and *show*. The first adds all widgets to he layout and the second replaces the currently shown configuration with itself. The *show*-function calls the *build*-function if necessary. The *show*-function is called by the *app-router* element when the workspace-configuration has been selected through the fragment identifier in the URL.

Figure 7 shows how the *WorkspaceConfigLoader* loads the *WorkspaceConfig* and all its dependencies.

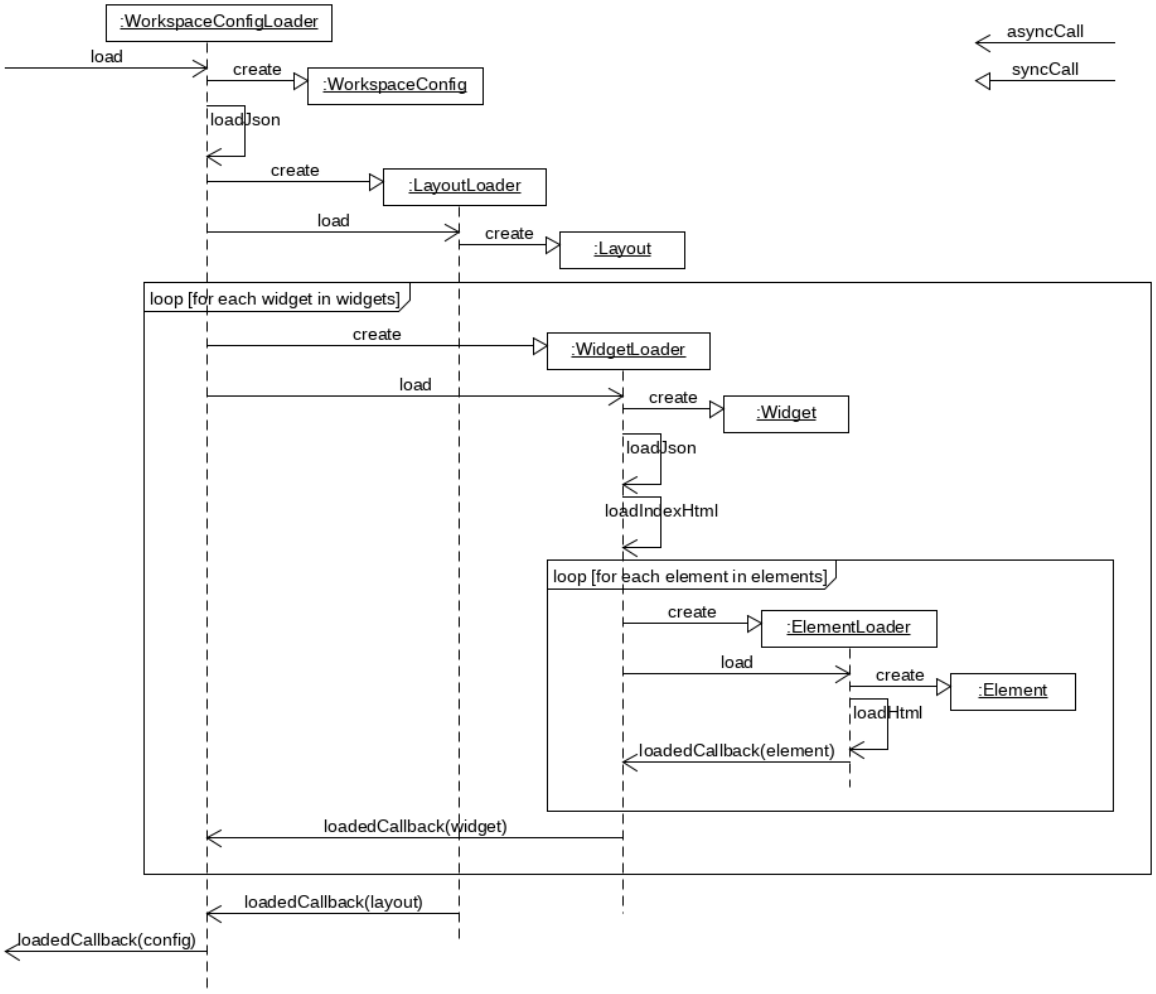

Figure 7: How the workspace is loaded

3.2 core-router

The module *core-router* manages the URL-paths of the workspaces and handles URL-changes that apply to the part of the URL that comes after the hash character '#', the so called fragment identifier. The core of the functionality is implemented in the element *app-router*. The module *core-router* is basically just a wrapper for the framework modules to easily access the functionality of the router. The only exposed method is *enable* that takes a map of URL-paths to *WorkspaceConfig*s, which is called in *app.js* after all workspace-configs have been loaded.

The element *app-router* uses the javascript-route-matcher third-party library to parse and match the fragment identifier (*path*-property in workspace-config) of the URL. It registers a callback function with the Javascript hashchange [3] event that is fired when

[3] https://developer.mozilla.org/en-US/docs/Web/Events/hashchange

the fragment identifier is changed. In that function, the fragment identifier is matched against the registered workspace paths. If one matches, the function *show* (see Figure 6) of that workspace is called.

3.3 core-dnd

The module *core-dnd* abstracts the interact.js third-party library and offers two easy methods to make element either a draggable or a dropzone.