

EPTF Web GUI, Function Description

Tamás Levente Kiss

Version 1551-CNL 113 864, Rev. PA2, 2018-02-14

Table of Contents

System Requirements	1
3 rd party libraries	1
Functionality	1
DsRestAPI	2
WebGUI Framework	2
General Functionality	2
Web Application Interface	2
The Setup Descriptor	3
Models	4
Common Viewmodels	5
Common Views	5
Utilities	7
CustomizableApp	8
MVVM Pattern Synopsys	8
General functionality	9
View Interface	9
Viewmodel Interface	10
Available Viewmodels	10
GuiEditor	11
Architecture	11
Classes and Their Roles	13
Usage	18
Overview	19
Description of Files in the Feature	19
Installation	19
Configuration	19
Using the Framework	20
Options	20
Creating a New Application	20
Using the CustomizableApp	20
Basic Overview	21
Options	21
Using the DsRestAPI Console	22
Using the GuiEditor	23
Editors	23
Setup Editor	23
Viewmodel Editor	31
View Editor	31
UIConfig Editor	32

Online Help	32
Structure of the source code	32
Abbreviations	33
References	34

System Requirements

3rd party libraries

EPTF Web GUI uses the following 3rd party libraries:

Product Number	Product Name	Product Description
1/CAX 105 8330	CodeMirror 5.6.0	CodeMirror is used for editing JavaScript, CSS and HTML code in an online manner.
7/CAX 105 4268	DataTables 1.10.3	Product is used to show and to interact with data displayed in table views.
33/CAX 105 3414	jQuery 2.2.3	It provides a rich function set including, but not limited to HTML document traversal and manipulation, event handling, animation, and AJAX calls. Other 3rd parties depend on this library as well.
1/CAX 105 8528	jQuery Splitter 0.14.0	jQuery Splitter is a plugin that splits the web content with movable splitter between them.
16/CAX 105 4236	jQuery UI 1.11.4	jQuery UI is used to create user interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript Library.
1/CAX 105 8531	JSON Editor 0.7.23	JSON Editor takes a JSON Schema and uses it to generate an HTML form.
1/CAX 105 8835	Ajv JSON Schema Validator 3.8.0	Product is used for validating JSON descriptors based on a JSON schema.
4/CAX 105 6647	jsTree 3.0.9	jsTree draws interactive trees displaying tree-like data structures
2/CAX 105 6945	Flot 0.8.3	Flot is used by TitanSim BrowserGUI and WebGU to display charts (e.g.: CPS, RPS charts). It replaces a similar 3rd party component that was removed from TitanSim due to licensing problems.
1/CAX 105 8146	Html2Canvas 0.4.1	Html2Canvas is used to convert any html element to canvas, which then can be saved as an image file. Used by the UIHandler (BrowserGUI) and WebGUI features.

Functionality

The EPTF Web GUI feature provides the following functionality:

- Provides the framework, which is responsible for the loading and unloading of web applications
- Web Applications: [CustomizableApp](#), [RequestTester](#), [GuiEditor](#)

- `CustomizableApp` can show highly customizable GUIs.
- Custom requests can be issued directly to the server with the `DsRestAPI` Console web application.
- The setups that customize the `CustomizableApp` can be edited in `GuiEditor`.

DsRestAPI

The javascript API of `DsRestAPI` is used to communicate with the server. It is taken from the CLL [3]. For more information on the API functions please see [4].

The API can be easily replaced with another one, so connecting to other services is also possible.

WebGUI Framework

General Functionality

When started, the framework will create the application buttons and load the default application if specified. The list of available applications and the default application's name is contained in `MainConfig.json`.

The framework also handles the loading and unloading the applications. The applications are given the `WebAppModel` at start that can be used to interact with the main configuration and the configuration of the application. It also contains the `SetupModel` which can create, load, and save setups.

Web Application Interface

The applications must provide the following interface so that they can be integrated into the framework:

- `info()`
This must return an object with an icon and name property. These will be displayed on the buttons.
- `load(p_webAppModel, p_params, p_framework)`
This method is called when the application should start. The `webAppModel` is a model which can be used to access the configuration, the API and other useful functions. The parameters contain the application parameters specified in the main configuration or set by other applications. The framework can be used to switch to other applications and to change the parameters of other applications.
- `unload(callback)`
This method is called when the application should be unloaded (when switching applications). The callback must be called with a boolean `true` value for the unload to be completed or `false` to cancel the unload.

The Setup Descriptor

The `CustomizableApp` is used to display setups. Setups describe the user interface that is displayed in the application.

A setup consists of the following files:

- *Request.json* - contains the request [4], [5].
- *ViewModelInstances.json* - contains the list of viewmodel descriptors.
- *ViewInstances.json* - contains the list of view descriptors.
- *Imports.json* - contains the list of import descriptors.
- *Setup.css* - contains the css of the setup. A normal css file.
- *Setup.html* - contains the html of the setup. A html snippet which will be inserted into the main html when loading the setup.
- *Desktop.json* - contains data of the editors in `GuiEditor` (e.g. position, visibility)

The descriptors are json objects with the following members:

- Viewmodel descriptor:
 - `class`:
"the viewmodel class name"
 - `dataPathList`:
[[0,0], [0,1]]: The list of data connections. The data connections point to a request in the request tree.
 - `dataPathStrList`:
the list of data paths represented by strings (e.g. "EntityGroups.Scenarios.ScStart"), used to make it more human readable
 - `selectionToControlPathList`:
same as `dataPathList` but for selection connections
 - `selectionToControlPathStrList`:
same as `dataPathStrList` but for selection connections
 - `customData`:
{}: an object that can be used to customize the viewmodel instance
- View descriptor:
 - `class`:
"the view class name"
 - `viewModelIndexes`:
[13, 28]: the index of the connected viewmodels
 - `parentID`:
"the parent id of the view"
 - `idsCreating`:
["id13", "id28"]: the list of ids that are the parentIds of other views

- `customData`:
{}: an object that can be used to customize the view instance
- Import descriptor:
 - `setupName`:
"the setup name we want to import"
 - `parentID`:
"the parent id where the setup view will be inserted"
 - `setupParams`:
[]: a list of setup parameters
 - `requestsPath`:
[0,1]: the path of a request whose child the setup's request will become; when undefined, the imported setup's request will be added to the root request list

Models

The framework contains models which can be used to interact with the server.

WebAppModel

The `webAppModel` is created at start and is given to the applications in their load functions.

It contains functions to interact with the configuration files.

SetupModel

The setup model handles every setup operation from listing setups to creating new ones and saving them.

It uses the `ImportResolver` when loading a setup that contain imported setups. When importing a setup:

- The setup parameters have to be resolved.
- The request path has to be shifted by the number of new requests.
- The viewmodel indexes have to be shifted by the number of new viewmodel instances.
- The view ids have to be made unique.
- The imported setup html must be inserted as a view.
- The css files have to be merged.

WebAppBase

A class that can be used as a base class of applications.

Its load function can be called with a list of javascript files, a start function and the API. It will import the javascript files and call the start function with a callback that must be called when it is finished.

Common Viewmodels

FileSelector

The `FileSelector` viewmodel can be used to create a file browser.

Common Views

Framework provides common view elements that all applications can use.

Aligners

- Base Aligner is responsible for orienting its child views horizontally or vertically, based on the custom data, the child views are positioned proportionately according to the custom data or equally if it is not given. It also provides jQuery UI resizable functionality.
- ElementAligner is derived from the Base Aligner, implementing the same functionality, difference is the subviews given in custom data are the ones aligned, not the child views.

AutoGUI

`AutoGUI` is a minimalist display generator for responses. Tree-based display which uses indentation and different colors to separate different levels of the tree, children are nested.

Nodes can be collapsed. Initially the root's first child subtree is fully expanded.

When a node (or a separator line) which is not a leaf node is clicked the following happens:

- If all of the children are expanded, then they will be collapsed
- If all children collapsed, or only one is expanded then all will be expanded, but only the immediate children, their children may be expanded or not based on if it was previously expanded or not.

When a node which is a leaf node on the lowest level is clicked, a prompt pops up with an input box to change values.

Basic Button

Standard HTML Button element with or without image and/or text.

Checkbox or Switch

Standard HTML Checkbox element with or without an image of Switch and/or text.

ComboBox

Regular combobox, which can have a predefined list and a default value.

Condition

A view that only shows one of its connected child views based on the state of the connected viewmodel.

Div

Regular container element for other visual elements.

Labels

Label: Standard HTML Input element displaying value of a `DataSource` element. Based on custom data it may or may not be changed.

Scroll

A scroll bar view that can be used to scroll data that is not currently present in the html.

Status LED

Displays an image assigned for certain members of enumerated Status LED `DataSource` types with or without a Label.

Tables

- Vertical Table: A table visual element, where rows may be subviews. Data sorting and filtering can be set up.
- Element Table: Based on the Vertical Table. The columns may be subviews.
- Element Table for Large Data: Based on Element Table and Scroll class. Implements streaming data from a larger collection.

Tabs

- Base Tabs: A jQuery UI Tabs view.
- Tabs with Data: Either a horizontal or vertical tabs view that display data received from a connected viewmodel.

Json editor

A Json editor view that can be used to edit json objects with a json schema.

Code editor

A text editor with syntax highlighting, validation and formatting.

Context menu

A context menu view.

FileSelector

A file browser view that can be used to browse the http server.

Utilities

The framework contains common functions, classes and resources that can be used anywhere.

Utilities

Contains useful functions, like object copying, string operations, etc.

TaskUtils

Contains several classes and functions that make it easier to manage asynchronous operations

FileLoader

A small class that makes it easier to load and save a text file.

JsonLoader

A small class that makes it easier to load and save json files.

JsTreeUtils

Contains useful functions for dealing with jsTrees.

LineDrawer

A utility class that draws svg arrows. Needs two endpoints to draw the arrow. The endpoints are javascript objects that can have the following members:

- **getOffset**- the only required function, which returns the endpoint's offset
- **multiple**- whether multiple offsets are given; the nearest two will be connected by the arrow
- **getOffsets**- if multiple offsets are given, this function must return them
- **getZIndex**- the z-index of the endpoint: the arrow will choose the biggest from its two endpoints + 1
- **style**: either horizontal or vertical
- **isEnabled**- whether the endpoint is visible: if one of them is not visible, the arrow is not drawn

ViewUtils

Contains utilities that deal with views. Also contains the dialog classes.

Common functions usable by views:

- **checkVisibility(conditionViewmodel, id)**:
Hides or shows the element depending on the condition viewmodel's state.

- `addLabel(id, text, class):`
Adds a label at the top of the element.
- `getViewmodelsFromExpectedInterface(viewmodelList, classname):`
Returns the ordered list of viewmodels for the given view class. The view class must implement the static `expectsInterface` function.
- `processCss(customData, parentId):`
It will insert css rules that only apply below the element with `parentId`.
- `applyCss(customData, id):`
Adds the `customData.css` to the element style attribute.
- `jumpToEditor(id):`
Scrolls the viewport so the element with the id becomes visible.

DataSourceUtils

Contains functions that convert the help and request to a `jsTree` specific data structure.

Also contains functions that check response structure equality and whether a given response corresponds to a given request.

RequestBuilderFromHelp_full

Creates a large request from the `DataSource` help.

RequestBuilderFromHelp_manual

This is a utility class that is used to edit the whole request and the filters as well. It uses the help to validate requests and to automatically guess parameters when their typedescriptor is given in the help.

HelpTreeBuilder

Creates a tree from the flat help.

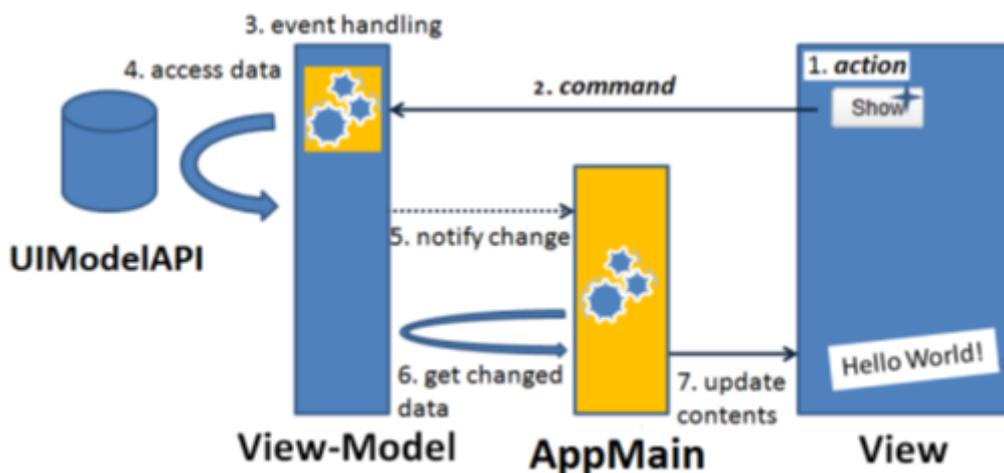
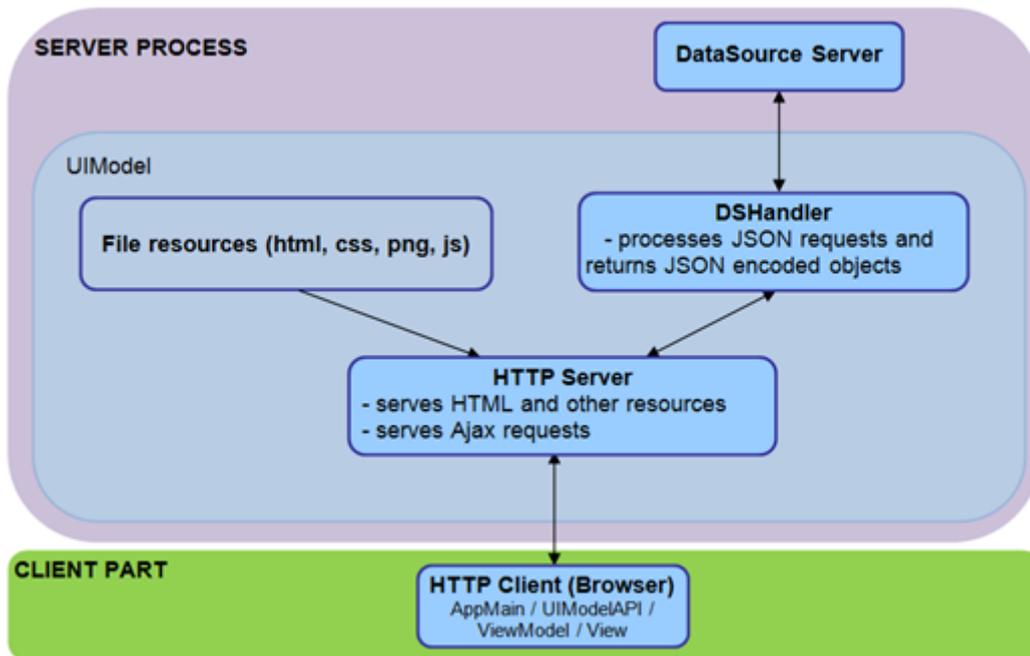
CustomizableApp

`CustomizableApp` can display a setup.

MVVM Pattern Synopsys

MVVM stands for Model-View-View Model. This pattern reverses the coupling direction, which allows multiple View Models to use a Model, multiple Views use a View Model. This approach also enables swapping components dynamically, and introducing isolated unit testing.

The `CustomizableApp` uses a component called Binder whose main task is to connect the views and viewmodels. The Binder also feeds the Models with the data from the server and notifies the Views to refresh themselves; hence it accompanies the application in the `Main.js` file.



The file and folder structure is straight forward, in the root of `CustomizableApp` the main, model and base view and viewmodel files are placed, and all the other views and viewmodels are located in a subfolder named accordingly.

General functionality

The main viewmodel and view is first initialized: this will create the view and viewmodel instances based on the setup.

After the initialization, the `applicationCreated` function of all viewinstances is called. Finally, the main loop starts in the binder which periodically calls the refresh method of the view instances.

WARNING

If there are asynchronous calls in `applicationCreated`, it will not necessarily be completed before the first refresh.

View Interface

A view is a JavaScript class that has the following public methods:

1. A Constructor accepting the following arguments:
 - a. List of connected viewmodels
 - b. Node ID to incorporate into the emitted HTML
 - c. Node ID of the container element
 - d. Custom Data
2. `applicationCreated`
3. `refresh(fullRefresh)`

The method `applicationCreated` is called when the View can start emitting actual HTML into its container element.

The method `refresh` is called when new data is available in the Model, and the View can fetch it from a viewmodel to refresh the display. The parameter `fullRefresh` will indicate if the structure of the data changed, so the visual element can be rebuilt accordingly.

If the view implements certain static functions, then better integration is provided into `GuiEditor`. See 2.4.2.2.8 for more information.

Viewmodel Interface

A viewmodel is a JavaScript class that has the following mandatory public methods:

1. A Constructor accepting the following arguments:
2. Base viewmodel
3. Options
4. Functions that get called on creation:
5. `setSelectionToControl(object)`: sets the selection connections object (the request pointed to by the selection path) one at a time
6. `setReponseDataPath(index, path)`: sets the data connections one at a time
7. `setBinder(binder)`: sets the binder whose `notifyChange` method can be called
8. An interface for the Views implementing functions such as: `select(index)`, `getTable()`, `getList()`

If the viewmodel implements certain static functions, then better integration is provided into `GuiEditor`. See [Sanity Checker](#) for more information.

Available Viewmodels

AutoGUI

Simply offers the raw response to be drawn by the view. Its primary aim is a predictable display of data. Not meant for usage other than development. It feeds the `AutoGUI` view with data containing every element below its data connection point, recursively.

Condition

The viewmodel serving the same named view. If the response was filtered its state will be false. If it was a Boolean value, it will use it as its state.

DynamicTable

`DynamicTable` can change its headers in runtime. This is necessary, as the headers are set (when the views are initialized) before the first response arrives.

It also has options for data manipulation and separator insertion.

Element Relay

This viewmodel expands fields from an element. In fact, it turns an element enumeration into a list, which can be fed into views that are `getList` capable.

Filter and Sort

Altering data for view with filtering and/or sorting.

Flex Aligner

Providing percentages to position the connected views.

Scroll for RangeFilter

Extra scrollbar handling for windowing tables from the response.

Table for Large Data

This viewmodel is streaming rows from the response to the view on the fly, speeding up display on view side.

GuiEditor

The `GuiEditor` application can be used to create and edit the setups which are displayed on the `CustomizableApp` GUI (for more information, see [Web Application Interface](#)).

WARNING

Editing these files manually will cause undetermined behavior both in `GuiEditor` and in the `CustomizableApp` GUI.

`GuiEditor` also contains small tools to create or edit available views and viewmodels, and to edit the configuration file of the GUI.

Architecture

`GuiEditor` uses an MVVM-like architecture similarly to `CustomizableApp`.

Classes and Their Roles

The Main module creates the main components and initializes them when the load function gets called by the framework.

Models

Models contain the parsed setup data and functions that manipulate them directly.

Every single view, viewmodel and import descriptor has its own model. Every model has a corresponding editor. The model also contains the editor's desktop data read from the *Desktop.json* file.

They handle changes which apply only to their part of the setup but happen elsewhere.

For example, when a request gets deleted, the data path will no longer be correct in the viewmodel descriptors, so we have to update them.

The main model also handles resources and loading / saving / deleting / creating setups.

Model (M)

Config, setup, view and viewmodel files handling. It creates the editor models at startup and when adding new editors.

Viewmodel Editor (M_VME)

Handles request data and selection connections and the custom data for the viewmodel descriptor.

Connections are references to the nodes of the request tree, for example, [1,2,3] means the 4th child of the 3rd child of the 2nd request.

Logic:

- adding, deleting, reordering connections
- `deleteConnectionsWithPrefix`:
When deleting a request, we have to remove all connections that point to it or one of its descendants.
- `updateConnections`:
When a request is deleted or added, we have to update the references to the tree.e.g: we delete [1,2,3], then [1,2,7,8] becomes [1,2,6,8]e.g: we add [1,2,3], then [1,2,7,8] becomes [1,2,8,8]
- moving requests:
consists of three parts:
 - we change the `fromPrefix` to `toPrefix`
 - or update the prefix: a node was deleted and a new node was added, we already know how to update it using the `updateConnections`

View Editor (M_VE)

Handling viewmodel connections, subviews and the custom data for the view descriptor.

Viewmodel connections are represented as an index, the view has a `parentID` and an `idsCreating` list.

Logic:

- adding, deleting, reordering viewmodel connections
- `viewModelDeleted`:
when deleting a viewmodel, the index of the preceding viewmodels has to be decreased in the descriptor
- the logic for handling subviews is in the corresponding viewmodel

Import Editor (M_IE)

Import descriptor handling. Similarly to viewmodels, the request path needs to be kept up-to-date.

Viewmodels

Viewmodels create an interface between the models and the views. They contain the logic for handling changes in the setup.

They handle the events that occur in the views (for example, creating a request after a drag and drop).

They also convert the descriptors to a format that the views can visualize.

Main viewmodel (VM)

Handling the setups, mainly a proxy to the model, refreshes the views through the binder on setup switching and creating a new setup.

Request Editor (VM_RE)

Mainly a proxy for utility classes that handle editing the request and filters (see [RequestBuilderFromHelp_manual](#)) and creating the help tree (see [HelpTreeBuilder](#)).

It converts the request, filters and help trees into a format that can be used with `jsTree`.

It can be used to find requests that have selection or filter (so we can highlight them in the view).

It also contains convenient functions that convert a request to a `sizeOf` or a `dataElementPresent` request.

Editor Container (VM_EC)

Handles creating, editing and deleting the view, viewmodel, import and html editor viewmodels (both in the beginning and one at a time).

It is also a proxy to the `SanityChecker`.

It handles listing available view and viewmodel classes, and the available setups.

Handles deleting a view-view connection (before connections are replaced with a new one)

Handles renaming view-view connections.

Logic:

- When a view or import gets deleted, we have to delete the following connections:
 - The connections to the subviews (only in case we deleted a view)
 - The connection to the parent view

Import Editor (VM_IE)

Only a proxy to its model.

Viewmodel Editor (VM_VME)

Mainly a proxy to its model. Also contains functions that the view can use to show a tooltip, create the jsTree, and to check if it is valid based on its custom data and connections.

View Editor (VM_VE)

In addition to being a proxy to its model and providing similar functions as the viewmodel editor, it also handles the logic behind view-view connections:

- Child id generation:
The child ids will always have the form: `parentId_classname_connectionIndex`.
- Renaming child ids:
This is used to keep the ids valid. When an id changes, it means we have to change the ids of the connected subviews.
- Child view order changed (from, to):
This happens when inserting, removing or reordering the child views. We rename the connections between the two indexes. We actually simulate the renaming, since only the indexes of the child connections change by either `+1` or `-1` depending on the relation of the indexes.
- Cycle detection:
When connecting this view to another, the `parentId` will change. If one of the original child ids is a prefix of the new `parentId`, then we have created a cycle.

Html Editor (VM_HE)

Handles accessing the html and css of the setup and the view connections.

When the html changes, the ids are collected. These can be used for view connections. In order to keep existing connections, we create a mapping between the old ids and the new ones.

Sanity Checker (VM_SC)

Handles the validation of views, viewmodels, their connections and custom data based on descriptors that are obtained by static functions of the classes.

These descriptors are the following:

- `class.getHelp`:
return the help info as a string
- `class.getCustomDataSchema`:
return the json schema that describes the custom data of the view or viewmodel
- `viewModelClass.providesInterface`:
return a list of function names that can be called by the connected views
- `viewModelClass.expectsConnection`:
return a description of connections, see `CViewModel_TableForLargeData.expectsConnection` for a complex example
- `viewClass.expectsInterface`:
return a list of expected interfaces, see `CView_BasicButton.expectsInterface` for an example

In the beginning, it imports all available javascript files of the viewmodels and views.

It can also be used to import files one at a time. This is used when saving javascript files in content editor, which validates the syntax and updates the validation functions.

Content Editor (VM_CE)

Handles the creation, loading and saving of the files for views and viewmodels.

We store the open files in a hashmap. Each entry contains the file loader, the file name, whether the file was edited, and whether it is already saved. When an action happens (a file gets saved for example), we simply update the appropriate parts (the file becomes saved in this example).

Most actions, like content changes, saving or closing an editor use the hashmap ids. However, the id is not always known (for example, when creating a new file), so we also store the name-id pairs. This is useful when we try to open or delete an already opened file.

When saving a file, we use the sanity checker to try to import the file. This shows if the file has syntax errors.

UIConfig Editor (VM_U)

Handles the loading and saving of the `UIConfig.json`. The json schema descriptor of the `UIConfig` can also be found here.

Views

The views display the setup.

The request appears as a `jsTree`.

The view, viewmodel, import and html editors are draggable boxes which can be collapsed and also contain jsTrees that represent the corresponding part of the setup.

The settings of the editor boxes are stored in the *Desktop.json* file of the setup.

Common view functions include the searching, handling the z-index changes and pressing the delete key.

Main view (V)

The main view of `GuiEditor`. It is responsible for switching between `GuiEditor` applications and handling setups (loading, creating, saving, etc).

Also stores the currently focused editor which can handle delete key press and z-index changes.

Request Editor (V_RE)

The view that is used to edit the request. The help and request jsTrees and their event handling functions can be found here.

There are also functions for handling changes that happens to the request outside this view.

Element Editor (V_EE)

A JSON editor for requests that also implements the common editor functions.

Trick: since the editor can be closed both from outside and from the editor itself, we bind a handler to the "remove" event of the editor.

Filter Editor (V_FE)

The filter editor for requests. It contains the jsTree which represents the filter and the functions that handle the events that occur on the tree.

Trick: copying a node in the `jsTree` will call a callback. If we recreate (delete and create again) the tree in this callback we get errors, since `jsTree` will try to call other methods on the now non-existing tree. So we use a zero timer to recreate the tree as the event queue will only be processed when all function calls complete.

Filter Element Editor (V_FEE)

Similarly to `V_EE` it is a JSON editor that edits a single part of a filter.

Editor Container (V_EC)

This is the central view that handles communication between the different editors. It is also a proxy to the connections view. The context menu, and custom data editor views and their options are also located here.

Base Editor

The base view for the small editors: view, viewmodel, import, html.

It contains as much common functionality as possible: the common view and editor functions, the context menu and custom data editing functions.

View (V_VE), Viewmodel (V_VME), Import (V_IE) and Html (V_HE) Editors

They are the editor boxes that visualize the corresponding parts of the setup.

Connections View (V_C)

Handles the connections between objects. A connection consists of its two endpoints and a `LineDrawer` instance (see [LineDrawer](#)). The connections are stored in a list.

When adding a connection, we only know one of its endpoints directly and have some information about the other end. So the endpoint we add will contain the connection type and an identifier that Editor Container will use to find the other endpoint. The identifier can be a completely different data type across connection types. For example in View-Viewmodel connections, the identifier is simply the viewmodel index that the view is connected to. But the identifier of a `Reuquest-Viewmodel` connection is actually a function that returns a path to the request which will be the other endpoint of the connection.

Endpoints also store the object from which they originate, so after deleting (or moving) an object (for example a view editor), we simply delete (or refresh) those connections whose either endpoint's object is the deleted (or moved) object.

A method where we hide all objects and show only those that can be reached from the given object or from which the given object is reachable is also implemented here.

Base Content Editor

The base class for view and viewmodel content editors. It handles the tab and panel manipulations and saving the edited content.

Trick: the tab ids are the same as the keys of the hashmap of the Content Editor viewmodel.

View and viewmodel content editors (V_VCE, V_VMCE)

These views handle the visualization of the view and viewmodel content editors that edit the javascript, html and css files.

Json Content Editor (V_JCE)

A JSON configuration editor view that is used to edit the application configuration files.

Usage

Overview

The EPTF Web GUI is developed as a framework for web applications. Currently available web applications include the new DataSource [6], [7] GUI client, the `DsRestAPI` Console and `GuiEditor`.

The framework uses the pull model based `DsRestAPI` javascript API [4], [5]. However, applications can provide their own implementation so it is possible to create user interfaces for other services as well in the framework.

Description of Files in the Feature

The feature includes the following structure:

```
└─ CustomizableContent
└─ Libs
└─ Utils
└─ WebApplicationFramework
└─ WebApplications
└─ favicon.ico [1150B] @2016-03-30T11:29:14.000Z
└─ Main.html [3748B] @2016-04-08T08:23:41.000Z
```

- **CustomizableContent:**
Contains custom content for applications. This directory should be writable by the user for some applications to work correctly.
- **Libs:**
Contains the 3rd party solutions listed in [3rd Party Libraries](#)
- **Utils:**
Contains the `DsRestAPI` (`Utils/DsRestAPI/DsRestAPI.js`) and other useful implementations intended for common usage.
- **WebApplicationFramework:**
Provides the base models for the `WebApplications` and the framework functionality.
- **WebApplications:** Contains the applications.

Installation

Installation is not needed. Instead, an http server that can serve the resources and handle the API requests is required.

The CLL `DsRestAPI` feature provides such http server, see [5] on how to use it.

Configuration

The main config of the framework, `MainConfig.json`, is located in the `CustomizableContent` directory.

Application specific options are contained in their respective customization directories as

AppConfig.json.

Specific options will be explained in the descriptions of the components that use them.

The config files can be edited manually or using [GuiEditor](#) (see section [UIConfigEditor](#)).

Using the Framework

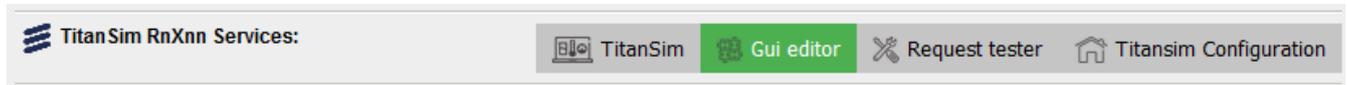


Figure: Available applications

The framework provides functionality to load the available web applications as well as common and useful functions and classes.

The framework will create the buttons that can be used to switch applications. The currently loaded web application is displayed with a green background.

Options

The available applications are contained in the *MainConfig.json* file in the `availableApps` array. Each application descriptor must contain a "directory" member, which is the directory where the application *Main.js* can be found.

If present the `defaultApp` option, which should be the name of an application, will be used to load the specified application at start.

Creating a New Application

A web application must have a *Main.js* file in a directory that can be served by the http server. The folder must be present in the `availableApps` array. This way, the *Main.js* file will be imported at start.

In the *Main.js* file, do the following:

1. Create the `WebApplications` array if it does not exist yet:

```
var WebApplications = WebApplications || [];
```
2. Add a new instance of your application to this array:

```
WebApplications.push(new Your_Application());
```
3. The instance must implement the web application interface functions described in section [Functionality](#).

Using the `CustomizableApp`

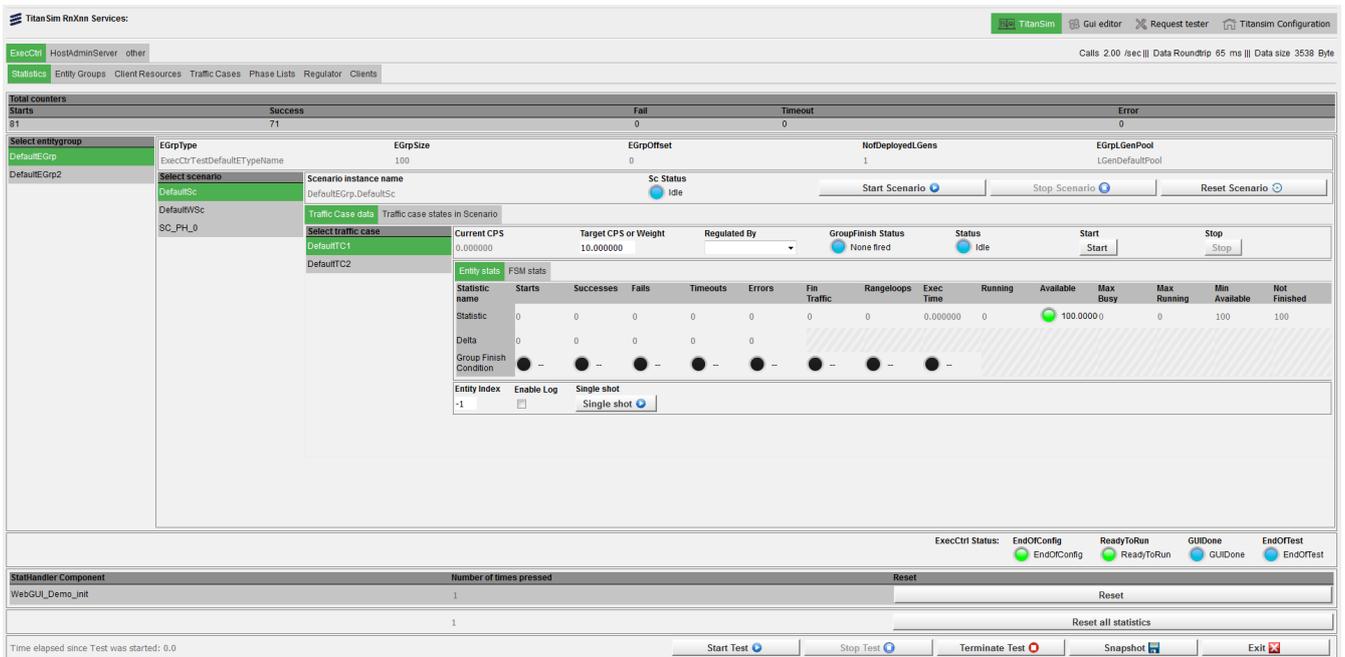


Figure: The `CustomizableApp`

The `CustomizableApp` is used to display setups. Setups describe the user interface that is displayed in the application. The structure of the setups is described in section [Functionality](#). A basic overview can be found below.

The aim of this application is to provide very high customizability. Setups can be edited to change the layout or use other view elements. It is even possible to create new views and viewmodels. All this can be done with the `GuiEditor` application (see section [Using the GuiEditor](#)).

Basic Overview

The setup contains a **request** [4], [5] which will periodically be sent to the server.

The data will be displayed by different **views**, like buttons and checkboxes. Views, like tables and aligners, are also responsible for the layout.

Viewmodels convert the response received to a data structure that the views can display. They also handle events which interact with the request, like selection changes or `setData` commands.

Options

The following `CustomizableApp`-specific options are available:

1. `setup`:
The default loaded setup
2. `overlayEnabledOnSelect`:
Whether an overlay with a loading sign is shown when changing the selection on a `request`. The overlay is displayed until a valid response arrives.
3. `overlayEnabledOnSetData`
4. `overlayOpacity`:

The opacity of the overlay, it must be between 0 and 1

5. **refreshInterval**:

The frequency in milliseconds with which requests are sent to the server, or -1 to disable periodic update (requests are still sent after selection change or setData requests)

6. **filesToInclude**:

The list of additional javascript files that will be loaded. The **DsRestAPI** files (or something implementing the **DsRestAPI** interface) must be included here.

Using the **DsRestAPI** Console

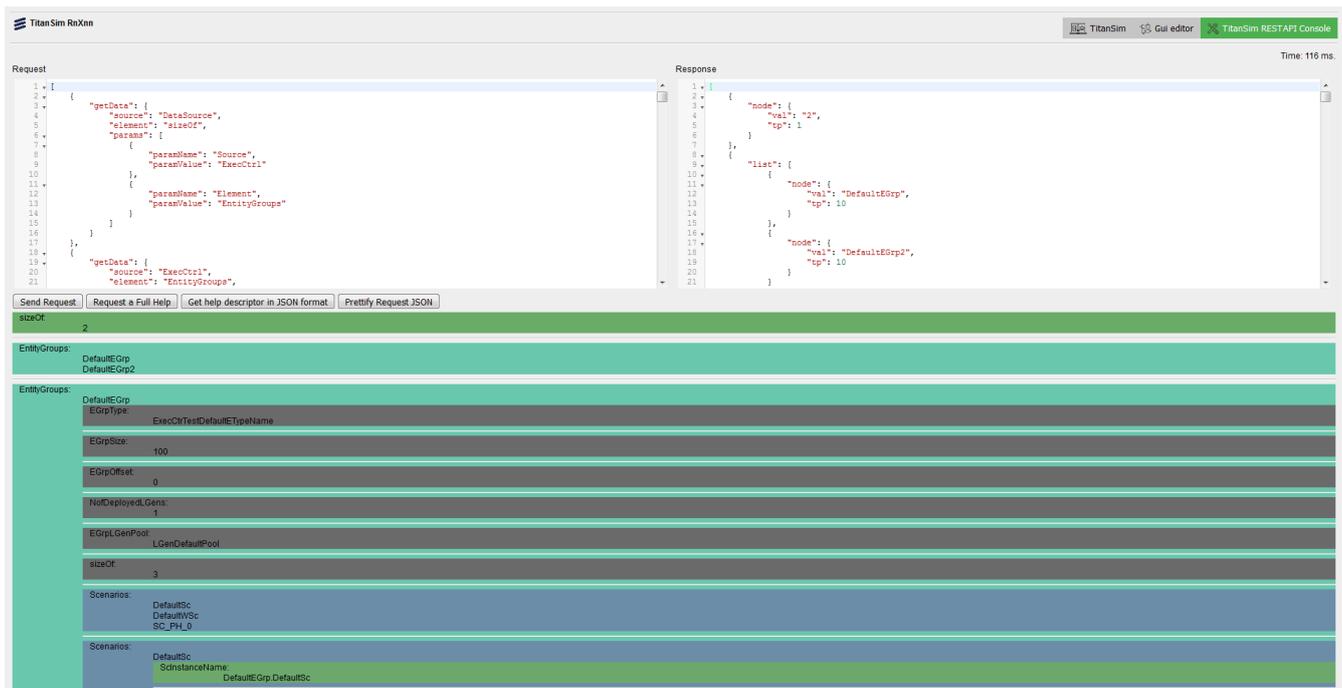


Figure: The DsRestAPI Console application

The **DsRestAPI** Console Application consists of two editor boxes, one for creating requests, and one to display the response. Below these for help there is a tree to show the relations of the elements of the **DataSource**. Above that is a text box to search in the tree.

The Search textbox will filter and highlight all the nodes which match the given text. Searching is not case sensitive, and partially matched nodes are listed as well. The result tree will show not just the matching nodes but all the nodes included in the path to get to that node.

After a request is constructed, by pressing the **Send Request** button a response can be received if it has a correct JSON syntax. If the request is syntactically incorrect an alert will inform the user. When the request is semantically incorrect, the response will be a JSON object containing an error message.

When the **Full Help to Request** button is clicked the Request editor box is filled with a request to get data from the full help tree.

With the **Get help in JSON format** button in the Response editor box the help tree will appear as a JSON object.

When a correct request is given indentation can be added with the **Prettify request JSON** button.

If the response is not an error, a visual element **AutoGUI** will show the result.

Using the GuiEditor

The **GuiEditor** application can be used to create and edit the setups, viewmodels and view elements which are displayed in the **CustomizableApp**.

Editors

The tabs at the top can be used to switch between different editors in the application.

The available editors are:

- **Setup Editor:**
it can be used to create and edit setups.
- **View Setup as Text:**
shows the edited setup as a JSON string, which can't be directly edited.
- **Viewmodel Editor:**
it can be used to create and edit viewmodel classes which can then be used in the Setup Editor.
- **View Editor:**
it can be used to create and edit views including their html, css and javascript files.
- **UIConfig Editor:**
some GUI settings can be edited here.

Setup Editor

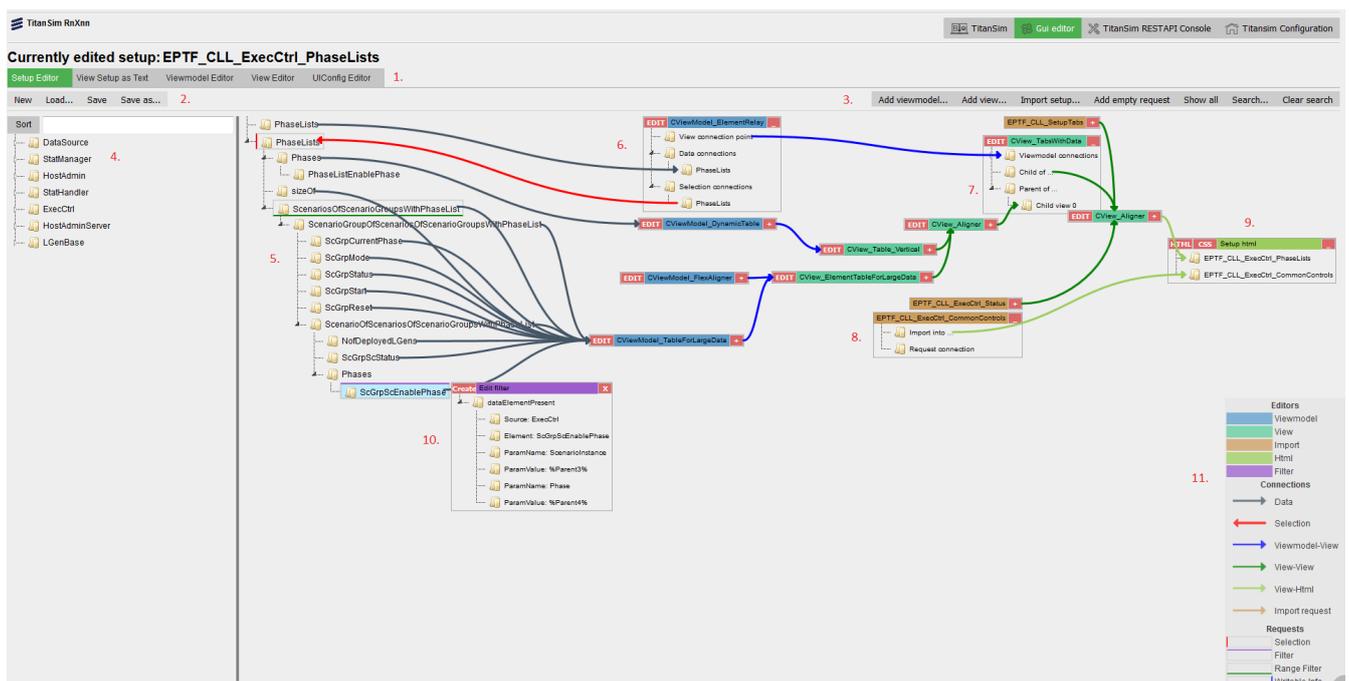


Figure: Setup Editor

The Setup Editor is where the GUI descriptors (the setups) can be edited [The Setup Descriptor](#).

The editor contains the following elements:

1. The editor switching tab.
2. Buttons to create, load and save setups.
3. Buttons that interact with the currently edited setup.
4. The **DataSource** help tree, which lists all known **dataElements** that can be queried in a **getData** request [6], [7].
5. The request represented as a tree [4], [5].
6. The viewmodels of the setup.
7. The views of the setup.
8. The imported setups.
9. The setup html and css editor.
10. The filter editor.
11. A legend is available by clicking in the bottom right corner.

General GUI Elements and Operations

Drag and Drop

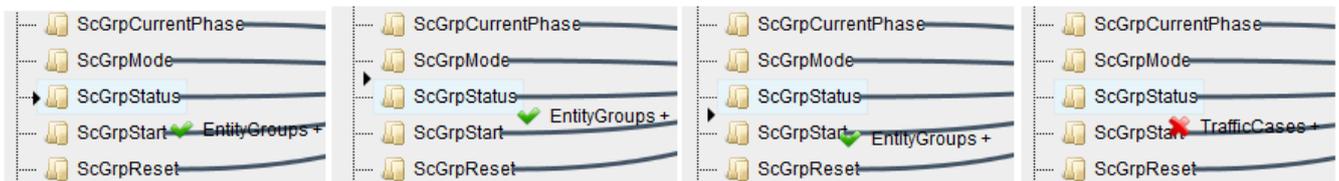


Figure Drag and drop: insert as child, sibling or not allowed

Drag and drop is allowed between certain trees. A checkmark will show whether the operation is allowed.

A small triangular marker will indicate the position to which the node will be dropped.

If it points directly at the target node, it will be inserted as the child of the target.

If it points above or below the target node, then it will be inserted as a sibling of the target.

Context Menu

Right-clicking some elements will bring up a context menu that contains useful options for editing the setup.

JSON Editor

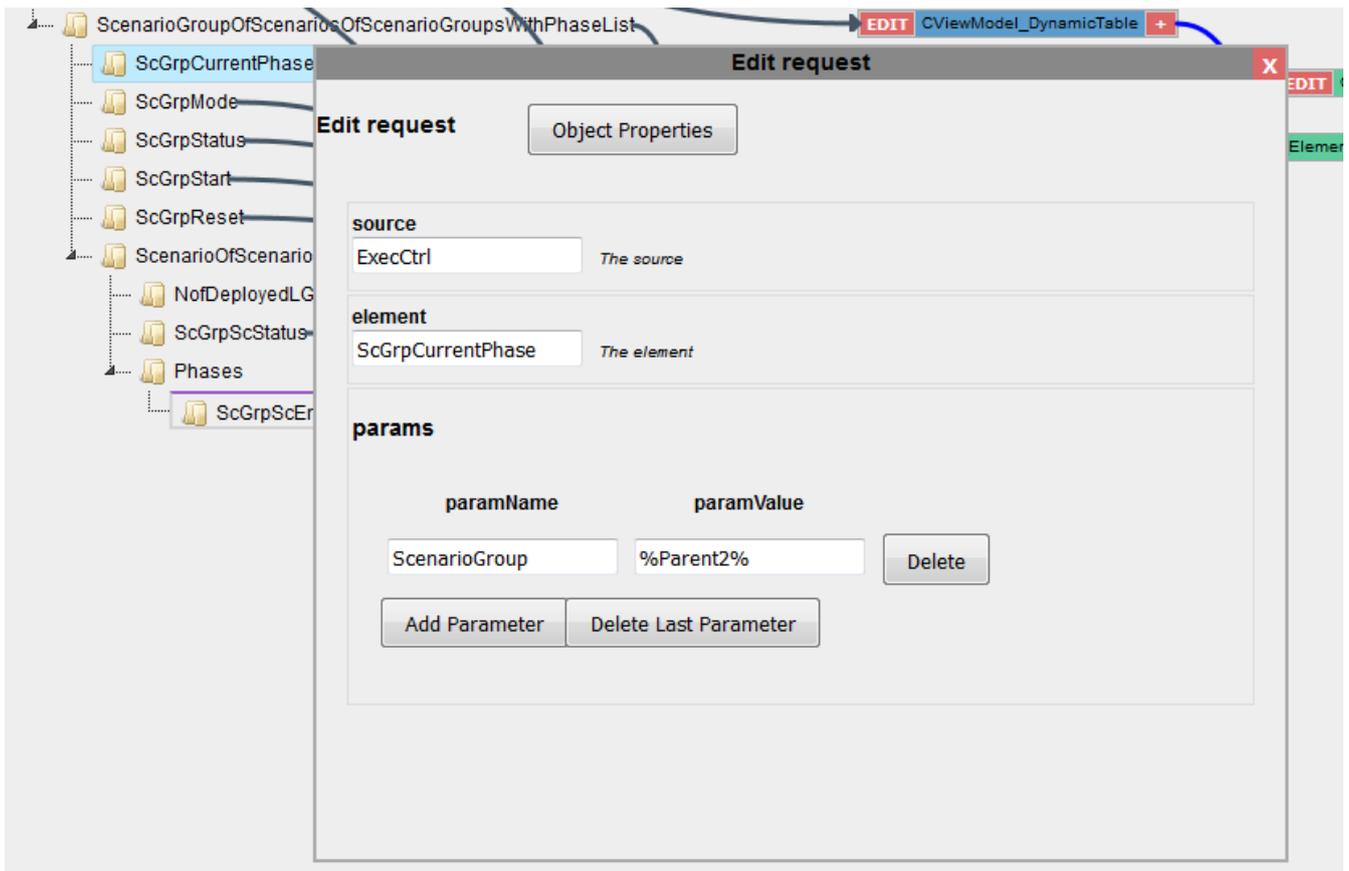


Figure 1. The JSON editor

Json editor can edit json data based on a json schema.

To add or remove members of an object, the **Object Properties** button can be used to toggle a list of available members or to create new ones when the schema allows it.

Code Editor

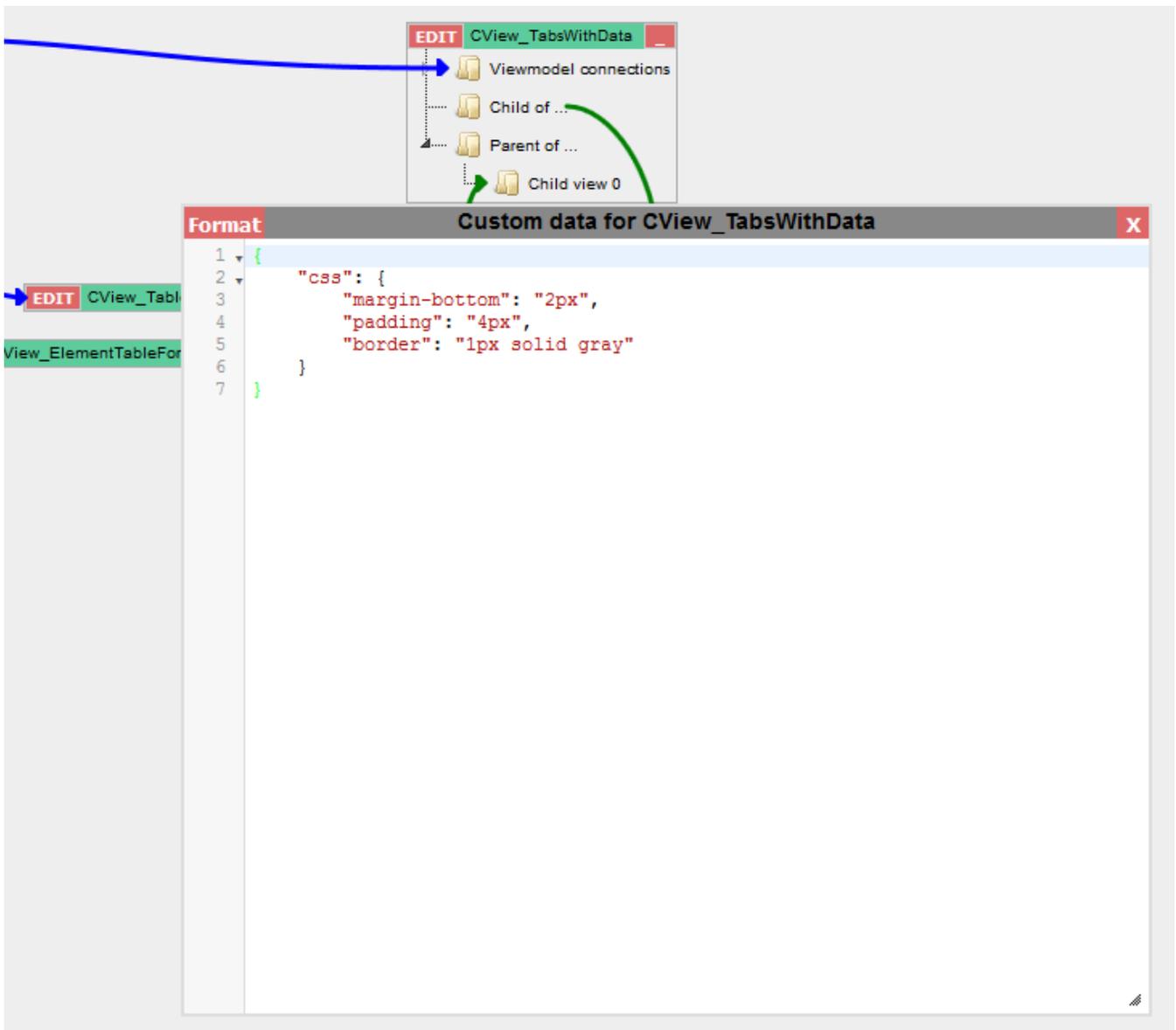


Figure 2. The code editor

It can be used to edit textual data with syntax highlighting where available.

Editing the Request

The request will be used to query the server for data, which can be shown on the GUI. To get more familiar with `DsRestAPI` requests, please see [4] and [5].

Requests that have selection have a red left border. Request with filters have a violet top border. Request with `rangeFilters` have a green bottom border. Requests with writable info have blue right border.

When hovering over a request tree node, the request json will be displayed in a small popup as text.

Methods of Editing the Request

1. Drag and drop:

The fastest way to edit the request is to use drag and drop from the help tree to the request tree. A darker background will show the area of the request tree. `GuiEditor` will try to guess the values of the parameters if the reference type name is given in the help descriptor of the

parameter. It will not allow the request to be inserted if a parameter with a reference type is missing. To force insert a request, hold down the ctrl key while dropping the node.

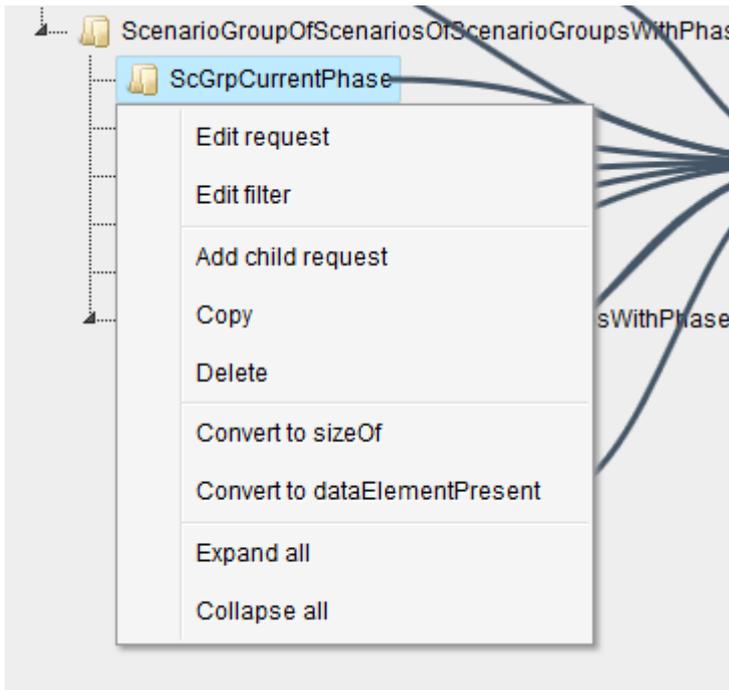
WARNING

The parameter values will be left undefined, they have to be filled in manually. To delete a request, drag and drop it to the help tree.

2. Add empty request button and Add child request menu element:

If the help is not available or not complete, these buttons can be used to insert empty requests which can then be edited with a json editor.

Menu Options

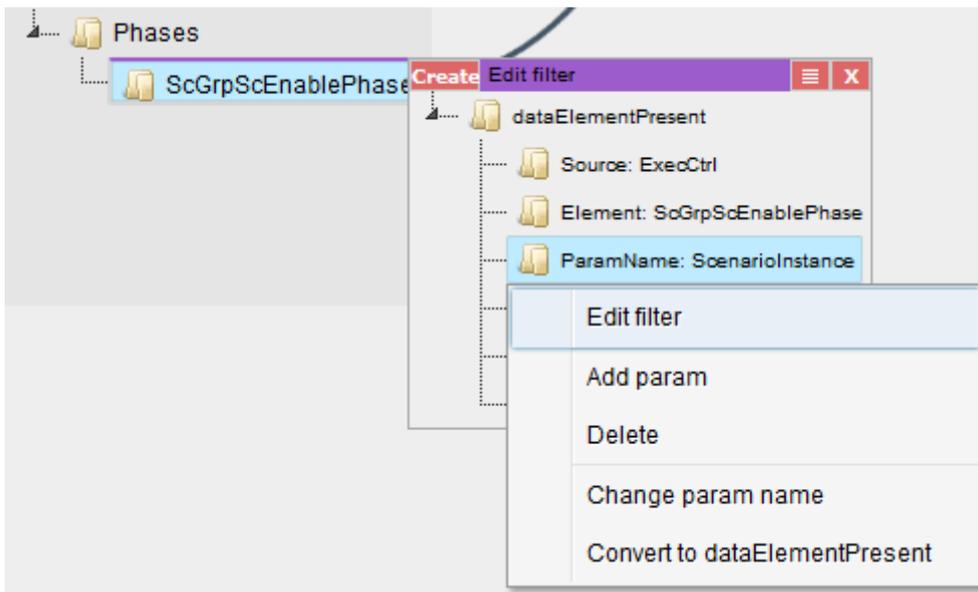


When right-clicking on a single request, the following options are available:

- **Edit request:**
Open a json editor that can be used to edit the request.
- **Edit filter:**
Open the filter editor to edit the filter of the request.
- **Add child request:**
Adds an empty child request.
- **Copy:**
Inserts the same request again. This is useful when only a single parameter must be changed but everything else is the same.
- **Delete:**
Removes the request.
- **Convert to sizeOf / dataElementPresent:**
The `sizeOf` and `dataElementPresent` requests are hard to create using the drag and drop or manual methods. These options make it easy.
- **Expand / Collapse all:**

Opens or closes all nodes of the tree.

Editing the Filters



Filters are used to disable some parts of the request. To get more familiar with **DsRestAPI** filters, please see [\[4\]](#).

On the filter editor, the **Create** button will create the filter whose **dataValue** is **true** by default.

Help tree elements can be dragged to the nodes of the filters. This will convert a **dataValue** to a request.

Request tree elements can also be dragged to the filter to create a reference to the parent requests.

Similarly to editing a request, the filter parts can also be edited with a json editor using the **"Edit filter"** option of the context menu.

Manual editing of the filter structure is possible with the other options in the context menu.

The button next to the close button will open the context menu of the root of the filter.

Editing the Viewmodel Instances

Viewmodels are used to convert the data returned for the request to something that views can understand. For example, a table viewmodel will convert the response, which is a tree, to a two-dimensional array.

Data connections are used to tell a viewmodel where to get its data. Selection connections are used to control the selection of the given request.

Viewmodels can be customized using their so called **custom data**. The custom data is a json object that can be used by the viewmodel to define or alter its behavior. For example, a table viewmodel might be customized to transpose the table or make the first row of the table the header. Most viewmodels provide a json schema which describe the available custom data options.

Viewmodels can be added with the **Add viewmodel...** button. A dialog will appear where the viewmodel class must be chosen. Some classes have help information which can also be viewed here.

To create a viewmodel connection, simply drag a request below the **Data** or **Selection connections** node.

WARNING | The order of the connections matter.

Data and selection connections will be represented with black and red arrows respectively.

The custom data of the viewmodel can be edited using the code editor with the **Edit** button or using the json editor with the appropriate context menu option.

Right clicking on the viewmodel editor title will show a context menu. The following options are available:

- **Edit with schema:**
Edit the custom data with the json editor.
- **Copy:**
Create a new viewmodel with the same class and custom data.
- **Change class:**
Change the class of the viewmodel.
- **Show only this:**
Hides all editors that are not connected to this one.
- **Show all:**
Show all hidden editors.
- **Delete:**
Remove the viewmodel.

Editing the View Instances

Views are the GUI elements that will appear on the screen. Views can be nested into each other. For example, connecting a button to an aligner will insert that button into the aligner on the GUI.

The custom data of a view can be used to alter how the view behaves or looks. For example, the custom data of all views can contain a **css** field that can be used to alter the look of that single view instance.

Views can be added with the **Add view...** button similarly to adding a viewmodel.

To connect a viewmodel to a view, simply drag the viewmodel's **View connection point** below the **Viewmodel connections** of the view.

To connect another view as a subview, drag the other view's **Child of...** node to the view's **Parent of...** node.

The connections between views are shown as green arrows.

WARNING

The order of both the viewmodel and view connections matter.

The custom data can be edited the same way as for the viewmodels. The context menu also contains the same options.

Editing the Setup html and css

The setup html and css can be used to define the base of the GUI. Views can only appear inside the setup html, so at least one div with an id is required.

The html editor can be used to edit the html and css of the setup using the code editor.

The ids of the html will be shown in the tree. Views can be connected to these ids by dragging their **Child of...** node and dropping it on a node in the tree. These connections are represented as light green arrows.

Importing Setups

Setups can be imported using the **Import setup...** button.

The imported setup must be connected to a view or the setup html. This can be done by dragging the **Import into...** node to the **Parent of...** node of a view or a node of the html editor.

A request can also be connected to the import by dragging a request to the **Request connection** node. This way, the request of the imported setup will be inserted below the connected request, so it is possible to write generic setups, which do not work by themselves, but can be imported.

The context menu contains options to delete the import or change the imported setup.

Searching

The **Search...** button can be used to search for any kind of string in the setup. When the string is found somewhere, that part of the GUI will be highlighted.

For example, searching for the string `button` will find all requests that query a button but also all views that are buttons.

The **Clear search** button will clear the search results.

Validating

Some views and viewmodels implement validating functions which are checked when their custom data or connections change. Description of these functions can be found in section [Functionality](#).

Invalid views and viewmodels will have a red border. When hovering over their title, the cause of the invalid state will be shown.

Viewmodel Editor

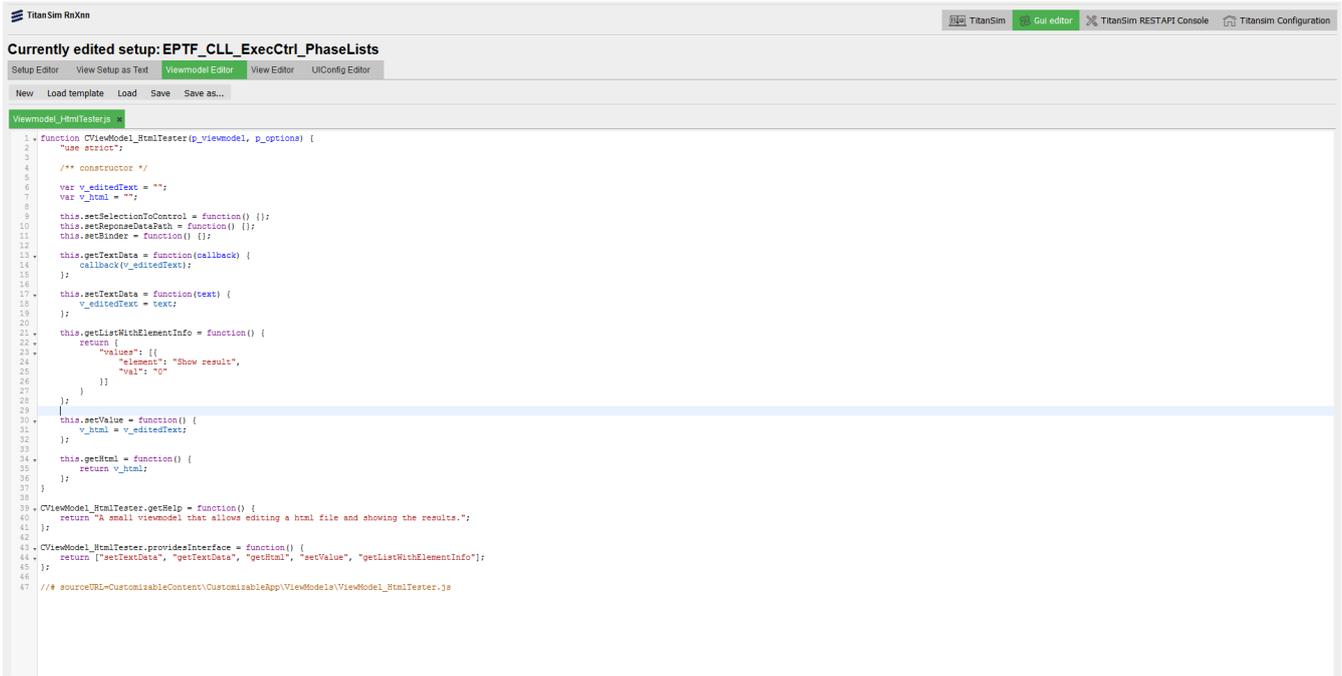


Figure: The Viewmodel Editor

The Viewmodel Editor makes it possible to create and edit custom viewmodel classes. The interface that viewmodels must implement are described in [Viewmodel Interface](#).

The following buttons can be found in the editor:

- **New / Load / Save / Save as:**

They all function as expected. Unsaved files will be shown with red letters.

- **Load template:**

With the load template button, a built-in viewmodel can be loaded and saved as a custom viewmodel.

WARNING

The viewmodel class name must be changed, otherwise either the original or the new one will be imported but not both.

For the custom viewmodel to appear in the Setup Editor the following conditions must be met:

- The edited file must be syntactically correct. This will be checked after saving it.
- The first line must contain the class definition:

```
function ClassName(p_viewmodel, p_customData) {
```

View Editor

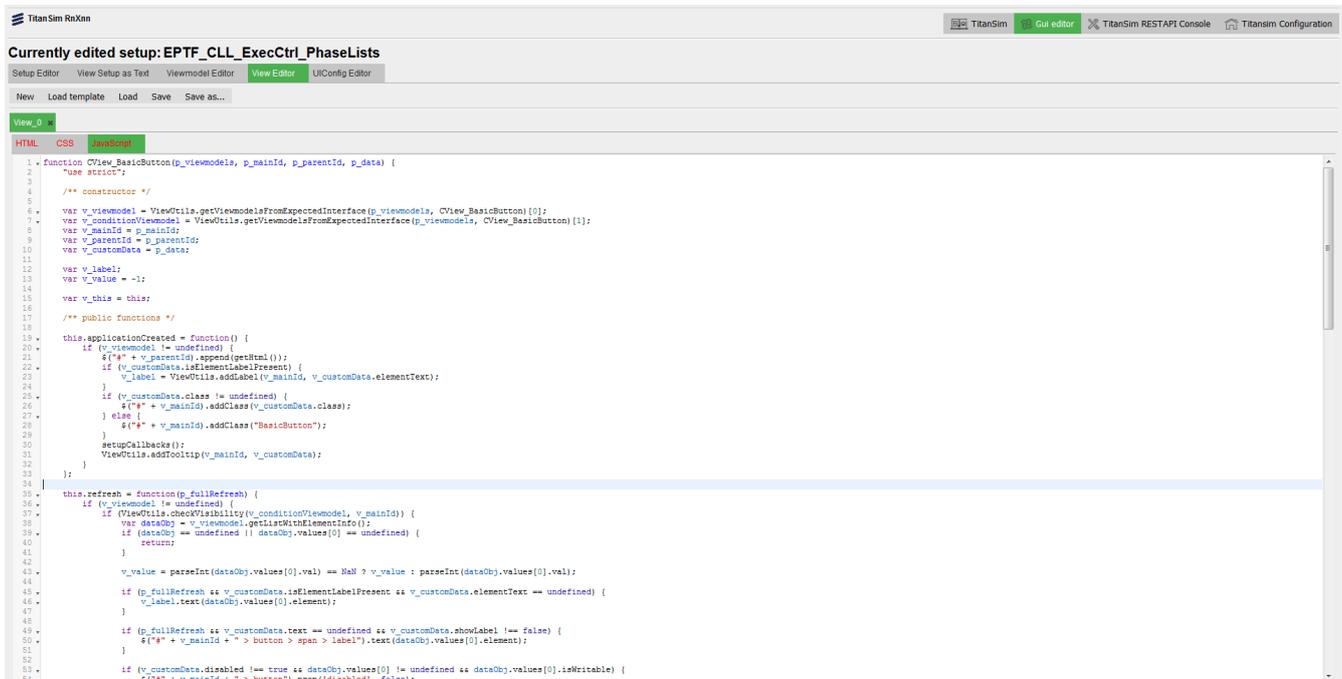


Figure 3. The View Editor

The Viewmodel Editor makes it possible to create and edit custom views including their html, css and javascript files which appear on another tab as three different editors. The interface that views must implement are described in [View Interface](#).

Everything else works the same way as in the Viewmodel Editor.

The first line of the javascript file of the view must be something like:

```
function ClassName(p_viewmodels, p_viewId, p_parentId, p_customData) {
```

UIConfig Editor

This editor can be used to edit the config files of the applications and the *MainConfig.json*. For details about the config file, please see section [Configuration](#).

Online Help

The Online help regarding the configuration and usage of *DsRestAPI* can be found [here](#).

Structure of the source code

The source code is organized into the following structure:

- *htdocs*

Contains the files of the EPTF Web GUI. See 3.2 for more details.

- *project*

A placeholder.

- *doc*

A placeholder.

License files:

Name	Purpose
<i>license.txt</i>	File containing the license information _
epl-v10.html_	The Eclipse Public License

Abbreviations

API

Application Programming Interface

CLL

Core Library

EPTF

Ericsson Performance Test Framework

GUI

Graphical User Interface

JSON

JavaScript Object Notation

REST

Representational State Transfer

SUT

System Under Test

TCC

Test Competence Center

TitanSim

New synonym for the EPTF Framework

TTCN-3

Testing and Test Control Notation version 3

UI

User Interface

XML

XSD

XML Schema Definition

References

[1] [RFC 6733](#)

Diameter Base Protocol, October 2012

[2] [RFC 3261](#)

SIP: Session Initiation Protocol, June 2002

[3] EPTF Core Library for TTCN-3 toolset with TITAN, Function Specification

[4] EPTF CLL DsRestAPI, Function Description

[5] EPTF Core Library DsRestAPI, User Guide

[6] EPTF CLL DataSource, Function Description

[7] EPTF CLL DataSource, User Guide