readme.md

How to write a widget

Cloudify UI provides an easy way to add widgets to the system.

Widget file structure

A widget consists of several files. 2 are mandatory.

- widget.js - Holds the widget's definition - **mandatory**.
- widget.png - A preview image of the widget - **mandatory**.
- widget.html - A widget template file. This is used only if you want to write a widget without using react (using vanilla js with html template). The template file is optional.
- widget.css - A css file that the widget uses. This file is optional.

A widget should be placed in the widgets library, in a library carrying the id of the widget.

For example, if we are creating a blueprint widget (id=blueprint) then the blueprint library should be:

```
/widgets
   /blueprint
      widget.js
      widget.html
      widget.png
      widget.css
```

We have 2 ways of writing widgets. The first one is using vanilla JS. In this approach we allow attaching an html template file and we support some callbacks that we will described later. The second and recommended way is to use React. Since react requires build, you can either build the widget.js file yourself, or use our build system for this. To use our build system you will have to put your widget.js file into an 'src' library along with any other files that you may require. In the widget.js file you can use 'import' for the other files and split your widget to several file. You can also use any ES6 features.

In this approach the filesystem will look like so:

```
/widgets
   /blueprint
      /src
         widget.js
      widget.js - will be generated by the build system
      widget.html
      widget.png
      widget.css
```

Widget definition

Each widget.js file should have only one call to a global function called *Stage.defineWidget*. This function takes one argument - an object which contains structured widget definition.

Example

The following code demonstrates how easy it is to create a simple widget. You can copy this code and put it in a *widget.js* file to produce a fully working widget (refer to the previous paragraph for file structure guidelines).

```
Stage.defineWidget({
    id: 'sampleWidget',
    name: 'A basic example',
    description: 'This widget displays "Hello World!" text',
```

```
    initialWidth: 4,
    initialHeight: 2,
    showHeader: false,
    showBorder: false,
    initialConfiguration: [],
    isReact: true,

    render: function(widget,data,error,toolbox) {
        return (
            <span>Hello World!</span>
        );
    }
});
```

Available widget configuration options:

As seen in the example above, there is a number of configuration fields that you can provide when designing a widget. Two of them are mandatory: *name* and the *render()* function. Omitting the *name* field will result in a runtime error in browser console. Whilst omitting the *render()* function will not produce an error but will prevent the widget from rendering in the UI.

The following table presents all available configuration options:

option	type	default	description
id	String	-	The id of the widget definition. Should match the directory name that we placed this widget in
name	String	-	**Required** The display name of this widget. This name will show in the 'add Widget' dialog, and will also be the default widget name once added to the page.
description	String	-	An optional description of the widget. It will be shown in the 'add Widget' dialog below the widget name.
initialWidth	Integer	-	The default (initial) width of the widget when added to a page
initialHeight	Integer	-	The default (initial) height of the widget when added to a page
color	String	red	one out of : red , orange , yellow, olive, green, teal,blue, violet,purple,pink,brown,grey,black
showHeader	Boolean	true	If we should show a widget header or not. If an header is not shown the user cannot change the widget name (which is only visible in the header)
isReact	Boolean	false	You should set this to true to write a React widget
fetchUrl	String or Object	-	If fetchUrl exists, the data from this URL will be fetched by the application and passed to the render and postRender methods. If you want to fetch multiple URLs, you need to pass an object where the key is a name you pick for this data, and value is the URL. *Important* the render will be called once before the data is fetched (to allow showing loading or partial data) and once the data is fetched.
initialConfiguration	Array	-	A list of widget configuration options. These options will appear when clicking 'configure' icon on the widget in edit mode. It can also be accessed in the widget to determine the current selected configuration.
pageSize	Integer	-	The initial page size for widgets that supports pagination

Note: initialConfiguration supports 4 generic pre-made config fields (see *fetchUrl* for example):

- Stage.GenericConfig.POLLING_TIME_CONFIG(int) - How often to refresh the data (in seconds)
- Stage.GenericConfig.PAGE_SIZE_CONFIG() - Takes no arguments and set's the page size to default 5
- Stage.GenericConfig.SORT_COLUMN_CONFIG(string) - Column name to sort by
- Stage.GenericConfig.SORT_ASCENDING_CONFIG(boolean) - Change sorting order (true=ascending)

fetchUrl

There are two primary ways of pulling data from remote sources: *fetchUrl* and *fetchData()*.

fetchUrl is an object member and may be defined either as a string or an object with multiple string properties (*property:URL*) where each property represents a separate URL to query. A single URL's results will be available directly in the *data* object. In case *fetchUrl* is defined with multiple URLs, the results will be accessible by property name of this URL (i.e. *data.nodes*).

Single URL example

```
fetchUrl:  'localhost:50123/public/nodes'
// ...
render: function(widget,data,error,toolbox) {
    let your_data = data;
    //...
}
```

Mulitple URL example

```
fetchUrl: {
    nodes: '[manager]/nodes?_include=id,deployment_id,blueprint_id,type,type_hierarchy,number_of_instances,host_id,re
    nodeInstances: '[manager]/node-instances?_include=id,node_id,deployment_id,state,relationships,runtime_properties
    deployments: '[manager]/deployments?_include=id,groups[params:blueprint_id,id]'
}
// ...
render: function(widget,data,error,toolbox) {
    let nodes = data.nodes.items;
    let deployments = data.nodeInstances.items;
    //...
}
```

As seen in the example above, URLs provided in *fetchUrl* can be parametrized with several special tokens:

```
fetchUrl: '[manager]/executions?is_system_workflow=false[params]'
```

- The `[manager]` token will be replaced with the current Cloudify manager's IP address (or proxy, if applicable).
- The `[params]` token, on the other hand, is quite special. This placeholder can be expanded into a number of things depending on usage:
  - `[params]` alone anywhere in the URL will be expanded to default pagination parameters (_size, *offset*, _sort) if available (see *initialConfiguration*). This mode is **inclusive** - all params availavble in the widget will be appended to URL.
  - `[params:param_name1(,param_name2)]` will be replaced with "&param_name1:param_value1" in the URL. Please note that this can be used both to selectively pick pagination parameter as well as custom parameters (see *fetchParams()*). This mode is **exclusive** - parameters not specified explicitly will be skipped. When using selective param picking (`[params:param_name]`) you can use a pre-defined `gridParams` tag to include all pagination parameters (_size, *offset*, _sort) instead of specifying explicitly each of the three.

fetchUrl - Inclusive params

The following example illustrates *fetchUrl* with both tokens along with resulting URL:

```
initialConfiguration: [
    Stage.GenericConfig.POLLING_TIME_CONFIG(60),
    Stage.GenericConfig.PAGE_SIZE_CONFIG(),
    Stage.GenericConfig.SORT_COLUMN_CONFIG('column_name'),
    Stage.GenericConfig.SORT_ASCENDING_CONFIG(false)
],
fetchUrl: {
    nodes: '[manager]/nodes[params]'
},
fetchParams: function(widget, toolbox) {
    return {
        sampleFuncParam: 'dummy'
    }
}
```

Result URL: http://localhost:3000/sp/?su=/api/v3.1/nodes?&_sort=-column_name&_size=5&_offset=0&sampleFuncParam=dummy

This url can be divided into 3 separate parts:

Field	Example	Description
manager address	http://localhost:3000/sp/?su=/api/v3.1/	The internal value of Cloudify manager [manager]
endpoint name	nodes?	Remaining part of the REST endpoint address
generic params	&_sort=-column_name&_size=5&_offset=0	Parameters that were implicitly added to request. These parameters are inferred from the GenericConfig objects in initialConfiguration and are responsible for pagination of the results. It is possible to omit them by explicitly specifying param names to be used like so [params:my-param]. Alternatively, gridParams (sort, size, offset) can be simply removed from *initialConfiguration*.
custom params	&sampleFuncParam=dummy	Custom parameters can be defined in *fetchParams()* function. Each custom parameter must be returned as a property of an Object returned by *fetchParams()* function.

fetchUrl - Exclusive params

The same URL, this time with explicit param names (and the `gridParams` tag):

```
initialConfiguration: [
    Stage.GenericConfig.POLLING_TIME_CONFIG(60),
    Stage.GenericConfig.PAGE_SIZE_CONFIG(),
    Stage.GenericConfig.SORT_COLUMN_CONFIG('column_name'),
    Stage.GenericConfig.SORT_ASCENDING_CONFIG(false)
],
fetchUrl: {
    nodes: '[manager]/nodes[params:sampleFuncParam,gridParams]'
//  which is essentially the same as
//  nodes: '[manager]/nodes[params:sampleFuncParam,_size,_offset_,_sort]'
},
fetchParams: function(widget, toolbox) {
    return {
        sampleFuncParam: 'dummy'
    }
}
```

Result URL: http://localhost:3000/sp/?su=/api/v3.1/nodes?&sampleFuncParam=dummy&_sort=-column_name&_size=5&_offset=0

Available widget functions:

Apart from configuration options presented in the above table, there is a number of function hooks available which can be used to manipulate the widget. Some for widget appearance manipulation and some to manage the data available to the widget.

init()

`Init()` is called when the widget definition is loaded, which happens once when the system is loaded. This can be used to define some global stuff, such as classes and objects we are going to use in our widget definition.

render(widget, data, toolbox)

`Render()` is called each time the widget needs to draw itself. It can be when the page is loaded, when widget data was changed, when context data was changed, when widget data was fetched, and etc.

render parameters are:

- The widget object itself (see description in 'Widget object' section below)

- The fetched data (either using fetchUrl or fetchData method). The data will be null if fetchData or fetchUrl was not defined, and also until the data is fetched it will pass null to the render method (if you expect data you can render 'loading' indication in such a case)
- The toolbox object (see description in the 'Toolbox object' section)

`Render()` is focal to the appearance of the widget as the return value of this function will be rendered to UI by React engine. As such it is important to understand how to build widgets. The following example illustrates the simplest usage:

```
render: function(widget,data,error,toolbox) {
    return (
        <span>Hello World!</span>
    );
}
```

Please note that `render()` may only return a single DOM node (refer to JSX spec for more detail). In order to render more than one HTML element they must be wrapped in a parent element (a `div` is usually a good choice):

```
render: function(widget,data,error,toolbox) {
    return (
        <div>
            <span>Hello World!</span>
            <p>Writing Cloudify UI widgets is <strong>super</strong> easy</p>
        </div>
    );
}
```

Using ready components in render()

Although using plain HTML tags gives you extreme flexibility, usually it is much quicker to design your widget with the use of Cloudify UI ready-made components. These components were designed with UI uniformity and ease-of-use in mind, and as are very easy to learn and use. The following example illustrates how to use a *keyIndicator* component:

```
render: function(widget,data,error,toolbox) {
let {KeyIndicator} = Stage.Basic;
    return (
        <div>
            <KeyIndicator title='User Stars' icon='star' number={3} />
        </div>
    );
}
```

The result of above code:

Take a note of how the *KeyIndicator* component is imported into the widget. From within the render method it is defined as `let {KeyIndicator} = Stage.Basic;` Similarly, you can import multiple components in the same line, ie: `let {KeyIndicator, Checkmark} = Stage.Basic;`

There is a number of components ready for use in the Stage.Basic library. A comprehensive documentation is due.

Accessing data in render()

There can be several independent data sources for your widget. Two most commonly used are the `configuration` and `data` objects. The following example illustrates how to access both of them:

```
Stage.defineWidget({
    id: 'sampleWidget',
    name: 'A basic example',
    description: 'This widget polls data from two different sources',
    initialWidth: 2,
    initialHeight: 2,
    showHeader: false,
    showBorder: false,
    isReact: true,
```

```
    initialConfiguration: [
        {id: 'confText', name: 'Conf Item',  placeHolder: 'Configuration text item', default: 'Conf text', type: Stag
    ],

    fetchData(widget, toolbox, fetchParams){
        return Promise.resolve({fetchedText: 'Fetched text'});
    },

    render: function(widget,data,error,toolbox) {
        let {Loading} = Stage.Basic

        if (_.isEmpty(data)) // Make sure the data is already fetched, if not show a loading spinner
            return (<Loading message='Loading data...'></Loading>)
        else
            return (
                <div>
                    <p>confItem value: {widget.configuration.confText}</p>
                    <p>fetchedText value: {data.fetchedText}</p>
                </div>
            );
    }
});
```

The above widget prints will display two lines containing the strings defined in the data sources: "Conf text" and "Fetched Text". Please note how the widget makes sure data has been loaded has completed before rendering it. Skipping this check would result in an error in browser console.

initialConfiguration, as the name suggests is only used if there are no user defined values for these properties. A user can change them by entering the 'Edit Mode' where he can modify widget's configuration. From that point, the current widget will use the value provided by the user. To reset it to it's default value, the widget must be removed and re-added to the workbench.

Moreover, please remember to remove and re-add the widget to the dashboard if changing the `initialConfiguration` field. It is only loaded for newly 'mounted' widgets.

postRender(el, widget, data, toolbox)

Non-React widgets only. PostRender is called immediately after the widget has been made visible in the UI. This function has access to the same objects as the *render* function with one addition - the `el` object containing a reference to the widget's container (parent) object.

fetchData(widget, toolbox, fetchParams)

An alternative to using *fetchUrl* is the *fetchData()* function. It provides greater flexibility when you need to pre-process your results or chain them into nested Promises (ie. Pull a list of URLs and resolve each of those URLs). The return value for fetchData() is expected to be a promise. As such if you would like to return a primitive value you would need to wrap it in a promise:

```
fetchData(widget, toolbox, fetchParams){
    return Promise.resolve({key:value});
}
```

Please note that should the result be a single primitive value you still need to return it as a property of an Object, since referencing the Object directly is illegal in React. With this in mind, the following example would not work:

```
// THIS WILL NOT WORK
fetchData(widget, toolbox, fetchParams){ return 10; }
render(widget,data,toolbox){
    return (
        <div>
            {data}  // This will produce a runtime error
        </div>
    )
}
```

Instead, you can return the `int` value as a property of the object like so:

```
fetchData(widget, toolbox, fetchParams){ return {myInt: 10}; }
render(widget,data,toolbox) {
    return (
        <div>
            {data.myInt}  // OK
        </div>
    )
}
```

fetchData() receives the same familiar arguments:

- current widget reference,
- toolbox reference,
- fetchParams (equal to `[params]` in *fetchUrl*)

Note: *fetchUrl* and *fetchData()* are mutually exclusive, that is if you define fetchUrl in your widget, then *fetchData()* definition will be ignored.

Widget object

Widget object has the following attributes

attribute	description
id	The id of the widget (uuid)
name	The display name of the widget (The widget definition name is the default name for the widget, but the user can change it)
height	The actual height of the widget on the page
width	The actual width of the widget on the page
x	The actual x location of the widget on the page
y	The actual y location of the widget on the page
definition	The widget definition object as it was passed to defineWidget method. The only additional field there that the widget can access is the 'template'. The template is fetched from the html and added on the widget definition.

Toolbox object

The toolbox object gives the widget tools to communicate with the application and with other widgets. It also gives some generic tools that the widget might require.

The toolbox gives access to the following tools:

getEventBus()

getEventBus() is used to listen (register) to events and trigger events, that is to broadcast an event (usually a change it made that will affect others). For example, if a blueprints widget creates a new deployment it needs to let all the other widgets know that the deployment list was changed. The listening widgets will then call 'refresh' Event buss supports the following methods:

- on (event, callback, context)
- trigger (event
- off(event,offCallback)

for example:

```
componentDidMount() {
    this.props.toolbox.getEventBus().on('deployments:refresh',this._refreshData,this);
}

componentWillUnmount() {
    this.props.toolbox.getEventBus().off('deployments:refresh',this._refreshData);
}
```

```
_deleteDeployment() {
    ...
    actions.doDelete(deploymentToDelete).then(()=>{
        ...
        this.props.toolbox.getEventBus().trigger('deployments:refresh');
    }).catch((err)=>{
        ...
    });
}
```

getManager()

Returns manager object. Used to read current manager's properties. Available calls:

- getIp()
- getCurrentUsername()
- getManagerUrl(url, data)
- getApiVersion()
- getSelectedTenant()
- doGetFull(url, params, parseResponse, fullData, size)

getExternal()

Provides access to connected manager's rest api. The URL is the service URL without the /api/vX.X

```
doGet(URL,params)

doPost(URL,params,data)

doDelete(URL,params,data)

doPut(URL,params,data)

doUpload(URL,params,file,method)
```

It also exposes a method to only construct the URL. It should be used carefully since some request headers needs to be passed to the manager.

```
getManagerUrl(URL,data)
```

for example:

```
return this.toolbox.getManager().doDelete(`/deployments/${blueprint.id}`);


doUpload(blueprintName,blueprintFileName,file) {
    return this.toolbox.getManager().doUpload(`/blueprints/${blueprintName}`,_.isEmpty(blueprintFileName) ? null
        application_file_name: blueprintFileName+'.yaml'
    },file);
}
```

Please note that it is recommended to use *fetchData()* instead of *doGet(URL, params)* since *fetchData()* also utilizes doGet() but also gives easy access to helper params.

getInternal()

Same as *getExternal*() but on a secured connection. All headers are appended with an 'Authentication-Token'.

getNewManager(ip)

Returns a manager object connected on the specified IP. May be needed in order to join a different manager (eg. for cluster joining).

getContext()

A widget context gives access to the application context. Using the context we can pass arguments between widgets, for example when a blueprint is selected, set the context to the selected blueprint, and all the widgets that can filter by blueprint can read this value and filter accordingly. The context supports these methods:

- setValue(key,value)
- getValue(key) - returns value

getConfig()

Returns global widget configuration as defined in conf/widgets.json.

refresh()

If we did some actions in the widget that will require fetching the data again (for example we added a record) we can ask the app to refresh only this widget by calling refresh().

loading(boolean)

Will show/hide a loading spinner in widget header. **Not allowed in render() and postRender()** methods as it changes store's state leading to render() and postRender() re-run.

drillDown(widget,defaultTemplate,drilldownContext)

Drilling down to a page requires passing the drilldown page template name. Templates will be described in the next section. When a widget is on a page, and drilldown action done (through link click event to a button for example), if it's the first time we access this drilldown page, the app will create a new page based on the passed template. Once this page is created the user can edit it like any other page. All next accesses to this page will use this page. Also you can pass a 'drilldownContext' to the drilldown page. This context will be saved on the URL and will be available through the app context. This value will be saved upon refresh, so if a user drilldown to a page, and then refreshes the page, the context will be saved (for example - selected deployment in drilldown deployment page)

for example: When selecting a deployment we drill down to a deployment page. It looks like this:

```
    _selectDeployment(item) {
        this.props.toolbox.drillDown(this.props.widget,'deployment',{deploymentId: item.id});
    }
```

The 'deployment' template looks like this:

```
{
  "name": "Deployment",
  "widgets": [
    {
      "name": "topology",
      "widget": "topology",
      "width": 12,
      "height": 5,
      "x": 0,
      "y": 0
    },
    {
      "name": "CPU Utilization - System",
      "width": 6,
      "height": 4,
      "widget": "cpuUtilizationSystem",
      "x": 0,
      "y": 5
    },
    {
      "name": "CPU Utilization - User",
      "width": 6,
      "height": 4,
      "widget": "cpuUtilizationUser",
      "x": 6,
      "y": 5
    },
    {
      "name": "Deployment Inputs",
      "width": 5,
      "height": 3,
      "widget": "inputs",
```

```
      "x": 0,
      "y": 9
    },
    {
      "name": "Deployment Events",
      "width": 7,
      "height": 3,
      "widget": "events",
      "x": 5,
      "y": 9
    }
  ]
}
```

Drilldown page templates

Drill down page templates are defined in the '/templates' library.

The library looks like this:

```
/templates
   template1.json
   template2.json
   ...
   templates.json
```

The templates.json contains a list of the available templates (temporary until we'll have a server that will handle this). Each template file contains one page template configuration.

template configuration has a name which is the default page name, and list of widgets. Each widget will have the following fields

field	description
name	Widget default name
widget	The id of the widget to use
width	The initial width of the widget on the page
height	The initial height of the widget on the page
x	The initial x location of the widget on the page
y	The initial y location of the widget on the page

If x and/or y are not defined the page will be auto arranged (not recommended)

For example:

```
{
  "name": "template-name",
  "widgets": [
    {
      "name": "topology",
      "widget": "topology",
      "width": 12,
      "height": 5,
      "x": 0,
      "y": 0
    },
    ....
  ]
}
```

Additional libraries that are available to a widget

moment a date/time parsing utility. Moment documentation

for example:

```
        var formattedData = Object.assign({},data,{
            items: _.map (data.items,(item)=>{
                return Object.assign({},item,{
                    created_at: moment(item.created_at,'YYYY-MM-DD HH:mm:ss.SSSSS').format('DD-MM-YYYY HH:mm'),
                    updated_at: moment(item.updated_at,'YYYY-MM-DD HH:mm:ss.SSSSS').format('DD-MM-YYYY HH:mm'),
                })
            })
        });
```

jQuery

for example:

```
    postRender: function(el,widget,data,toolbox) {
        $(el).find('.ui.dropdown').dropdown({
            onChange: (value, text, $choice) => {
                context.setValue('selectedValue',value);
            }
        });
    })
```

Lodash

for example:

```
    _.each(items, (item)=>{
        ...
    });
```

Widget template

The widget template is an html file written with lodash template engine.

Widget template if fetched when the widget definition is loaded, and its passed to the render function. To access it use widget.definition.template. To render the template using the built in lodash templates engine use `_.template(widget.definition.template)(data);` , where 'data' is any context you want to pass on to the template. For example, a simple render function will look like this:

```
    render: function(widget,data,toolbox) {
        if (!widget.definition.template) {
            return 'missing template';
        }
        return _.template(widget.definition.template)();
    }
```