Customizing Properties View in IBM Case Manager v5.2.1

Authors: Kai Zhang, Brent Taylor
Special thanks to: Mingliang Guo, Yajie Yang

Introduction

IBM Case Manager (ICM) is an industry-leading case management product. It is widely used in business solutions for banking, insurance, government, and other industries. Properties view is a core widget of ICM client. It enables the user to view and edit the properties from cases, tasks, documents, and even from external data. The out-of-the-box capabilities of properties view enable the solution developer to design the layout of the properties view, to choose the editors for the properties, and to configure the property's attributes such as the title, the validation pattern, the width, etc.

While the out-of-the-box properties view designer provides rich configurations and covers most of the common seen business requirements, it is not rare that in the real world solution, you want some of the properties to be displayed and edited in special way, in order to satisfy the requirement for the specific business. Starting from IBM Case Manager 5.2.1, we provide properties view customization for these scenarios.

Previous solution

On IBM Case Manager prior to 5.2.1, there is limited public APIs and documentation for customizing the properties view widget. You have to write a new page widget for those properties to be handled specially. This requires extra skills and time in solution development, and is inflexible for the future requirement changes.

New solution in ICM 5.2.1

To enhance the customization capabilities, ICM 5.2.1 introduces new interfaces and public APIs that enable you to develop your customized editors for properties view. The customized editors are packaged as extensions package, which can be easily deployed to the system using IBM Case Manager Configuration tool . After being deployed, they can be used in the Case Builder and the Case Client in the same manner as the standard out-of-the-box properties editors, and you can leverage full capabilities of properties view designer to design your view with those customized editors. The new feature also provide advanced customization beyond property editors, such as customized data types and controllers for handling external data.

Overview of properties view customization

The properties view is built with a three-tier MVC structure, and customization points are provided on different tires. Let us start with the architect of the properties view.

- Editor - the view tier to present the property to the user and which the user can use to modify its

value.

- Controller - the controller tier connects the view tier and the model tier, providing the functionality of action coordination, input validation, change set management, etc. Each property in the model is bound to a property controller. A property collection controller acts as a container for all the property controllers associated with a model. The controllers are shared among multiple widgets.
- Model - the client-side representation of the persistent data. ICM provides a set of model API to manipulate the data in ICM server. It is possible for the user to customize a model of external data stored in 3rd party data sources.
- Registry configuration - the configuration for the view tier (the editors), determining the editors available in the properties view, and their mappings to appropriate data types. Registry configuration is managed by RegistryManager module.
- Integration configuration – the configuration for the controller tier, determining which type of controller to use with a particular data type, and the binding of the controller attributes to the data in the model tier. The integration configuration is managed by ControllerManager module.

Properties view provides mergeConfiguration method in both the RegistryManager module and the ControllerManager module, enabling injecting the custom configuration for the view tier and the controller tier. Using together with other APIs and facilities provided by ICM, you can extend properties view by changing the behavior of the built-in widgets, registering your custom editors, and integrating external properties with custom data types, etc. (The topic on external properties is covered in the article Using External Properties in IBM Case Manager v5.2.1)

Developing your custom assets

ICM 5.2.1 provides two basic approaches for you to inject your properties view custom code to the system

- Script Adapter widget – a handy tool to host the custom scripts on a page
- Extensions package – a more comprehensive infrastructure to deploy custom assets

The following table shows the comparison between the two approaches, and you may choose one that best suits your business needs.

	Script Adapter widget	**Extensions package**
Scope	Page, supports runtime customization only	Global, supports both runtime customization and design time configuration customization
Development tool	Case Builder	Standard Java and JavaScript IDE and build tools
Complexity	Single JavaScript module	Multiple JavaScript modules and Java code packaged in ICN plugin
Packaging	Inside the solution	Extensions package
Deployment	Deploys with the solution	Deploys separately in IBM Case Manager configuration tool
Suitable scenarios	Minor tweaks on the built in functionality during runtime	Complex customization, new custom editors and controllers, multiple modules, etc.

Customizing properties view with the Script Adapter

Script Adapter widget is a handy option for hosting the scripts that extend the properties view. You code in Script Adapter widget is part of the solution package, and gets deployed along with the solution without additional steps. It is suitable for minor tweaks on the built-in properties view functionality. A recommended means to do such tweaks is to manipulate the controller attributes using Script Adapter code. We provide an example below to demonstrate how to do customization using the controller attributes.

Use case study: customizing prompt and error messages

In this example we have a string property “Length of service” that must be in the format of YYMM format. For instance, if you have been serving for the company for 10 and a half years, the property should be like “1006”. There are additional requirements that you are clearly indicated the special format of the property, and if you input a wrong value, you get a clear message about the mistake. Now let's formalize the requirements and solutions below in the table.

Requirements for length of service property

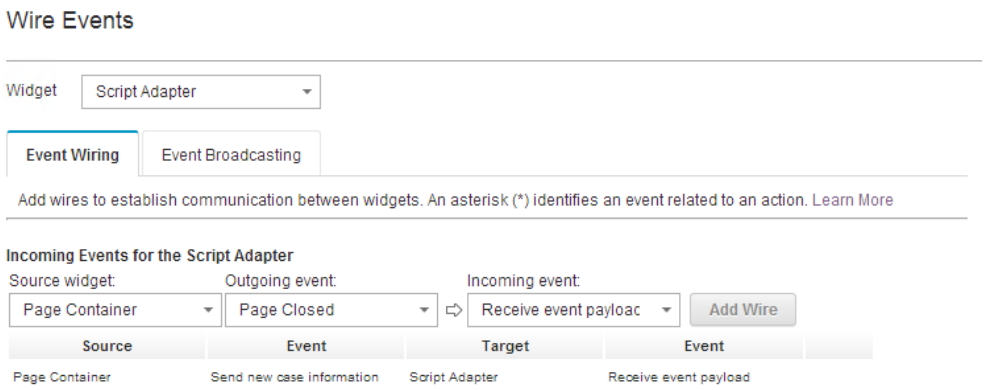
	Requirement	Solution
1	The input value must be in YYMM format	Add a pattern in the properties view design to validate the value
2	Provide a clear prompt message for the format	Customize the “promptMessage” attribute of the property's controller
3	Provide a clear error message for the format for invalid input	Customize the “invalidMessage” and “maxMessage” attribute of the property's controller

Implementing the requirements

1. Set the validation pattern in the properties view designer as `\d{2}(0\d|1[012])`

2. Add Script Adapter Widgets to the relevant pages, e.g., Add Case, Case Details, Work Details, etc. Wire the Script Adapter Widgets with the following events

- Send case information (for Case Details and Work Details page), or
- Send new case information (for Add Case page)



Put the following code in the script adapter

```
require([
    "icm/base/Constants",
    "icm/model/properties/controller/ControllerManager"
], function(Constants, ControllerManager) {
    // Retrieve the editable and the controller objects from the event payload.
    var editable = payload.caseEditable;
    var coordination = payload.coordination;
    var collectionController = ControllerManager.bind(editable);
    var controller = collectionController.getPropertyController("F_CaseFolder",
"CARD_LengthofService");

    // Use the BEFORELOADWIDGET coordination topic handler to obtain the controller binding
    // for the editable and to update the properties.
    coordination.participate(Constants.CoordTopic.BEFORELOADWIDGET, function(context, complete, abort)
{
        // Start a batch of changes.
        collectionController.beginChangeSet();

        // Set the controller attributes.
        controller.set("promptMessage", "Enter the length of service in YYMM format.");
        controller.set("invalidMessage", "The value of this property must be in YYMM format.");
        controller.set("maxMessage", "The value of this property must be in YYMM format.");

        // Complete a batch of changes. This tells the subscribed widget to refresh.
        collectionController.endChangeSet();

        // Call the coordination completion method
        complete();
    });
    // Use the AFTERLOADWIDGET coordination topic handler to release the controller
    // binding for the editable.
    coordination.participate(Constants.CoordTopic.AFTERLOADWIDGET, function(context, complete, abort) {
        //Release the controller binding for the editable.
        ControllerManager.unbind(editable);
        //Call the coordination completion method.
        complete();
    });
});
```

Deploy the solution, and you can test the message in case client as below

Customizing properties view with extensions package

For more complicated customization involving multiple code modules, extensions package is more suitable. Extensions package is a new infrastructure introduced in ICM 5.2.1. It provides a means to package multiple custom assets in a single standardized and deployable zip package. The package can be deployed easily using the IBM Case Manager Configuration tool to ICM develop environment and production environment. Your customized properties view assets must be packaged into a extensions package before you can deploy and use them in your ICM system. In the following sections, we will show you how to use extensions package to build a custom editor.

Use case study: custom Yes No Check Box editor

In this example, we have a boolean property to indicate whether the customer's signature is provided. The property has three states – true if the signature is provided, false if no signature is provided, and null if the signature is provided but is not clear or is in question, and needs following up with the customer. Besides that, more user-friendly options on the UI is preferable, such as Yes and No, rather than True and False. As none of the OOTB editors fits the requirement, we need to create a new editor and plug it into the properties view. Let's formalize the requirements as below in the table

	Requirement	**Solution**
1	A boolean property editor handles - true, false, and null	A custom editor as below Signature ☐Yes ☐No
2	The options on the UI should be Yes and No, and the end use can choose none or one of them	
3	The editor can be used in the same way as other OOTB editors, such as drag and drop it in the properties view, use it with case properties, task properties, and external properties.	

The following sections will demonstrate how to implement the custom editor step-by-step.

Build the dojo widget for the editor

Firstly, you need to create a dojo widget that suits the UI requirements. Depending on your business needs, you can choose to extend some of the built-in editor and override some attributes of it, or you can build one from the ground. Please refer to dojo's documentation on how to build a dojo widget and how to extend a widget. In this example, we build our own widget using dojo's template mechanism. We create an HTML template as below. Some part of the template is required by the properties view; please refer to the remarks in the code.

```
<div>
    <!-- Label and star indicator. Do not change this part. -->
    <div class="idxLabel dijitInline dijitHidden" dojoAttachPoint="labelWrap">
        <label dojoAttachPoint="compLabelNode"></label>
    </div>

    <!-- Custom code goes in the div with stateNode and oneuiBaseNode attach point. -->
    <div dojoAttachPoint="stateNode,oneuiBaseNode">
```

```
        <!-- Yes and No check boxes. -->
        <form action="" dojoAttachPoint="checkBoxNode,focusNode">
            <label><input type="checkbox" data-dojo-attach-point="yesNode" data-dojo-attach-
event="onclick:_onYesClick">Yes</input></label>
            <label><input type="checkbox" data-dojo-attach-point="noNode" data-dojo-attach-
event="onclick:_onNoClick">No</input></label>
        </form>
    </div>

    <!-- Error icon. Do not change this part. -->
    <div class='dijitReset dijitInline dijitValidationContainer' dojoAttachPoint="iconNode">
        <div class="dijitValidationIcon">
            <input class="dijitReset dijitInputField dijitValidationInner" value="&#935; " type="text"
tabIndex="-1" readonly="readonly" role="presentation"/>
        </div>
    </div>
</div>
```

We then create a dojo class for the widget. In addition to the standard dojo practices, properties view requires some methods and properties to be provided. Please refer to the remarks in the code.

```
define([
    // Standard dojo modules for widget.
    "dojo/_base/declare",
    "dijit/_WidgetBase",
    "dijit/_TemplatedMixin",
    // The template.
    "dojo/text!./templates/YesNoCheckBoxEditor.html",
    // Modules required by properties view.
    "idx/form/_CssStateMixin",
    "idx/form/_CompositeMixin",
    "idx/form/_ValidationMixin",
    "pvr/widget/editors/mixins/_EditorMixin"
], function(declare, _WidgetBase, _TemplatedMixin, template,
    _CssStateMixin, _CompositeMixin, _ValidationMixin, _EditorMixin)
{
    // Define the custom editor class.
    return declare("custom.editor.YesNoCheckBoxEditor", [
            _WidgetBase, _TemplatedMixin,
            _CssStateMixin, _CompositeMixin, _ValidationMixin,
            _EditorMixin],
    {
        // Standard dojo practices.
        // Loading the template.
        templateString: template,

        // Custom mouse click handlers to avoid both options are checked.
        _onYesClick: function(nodeBeingChecked) {
            this.noNode.checked = false;
        },
        _onNoClick: function(nodeBeingChecked) {
            this.yesNode.checked = false;
        },

        // Properties view required methods.
        // The class to apply CSS. Must be set even if you don't want your custom CSS.
        oneuiBaseClass: "customYesNoCheckBoxEditor",

        // Register the basic events for the widget.
        postCreate: function(){
            this._event = {
                "input" : "onChange",
                "blur"     : "_onBlur",
                "focus" : "_onFocus"
            }
            this.inherited(arguments);
```

```
        },

        // Provide the value attribute for controller to get and set the value from the editor.
        _setValueAttr: function(_val) {
            if (typeof _val === "undefined" || _val == null) {
                // The input value is neither yes nor no, uncheck both boxes.
                this.yesNode.checked = false;
                this.noNode.checked = false;
            } else if (_val) {
                // The input value is yes, check the yes box and uncheck the no box.
                this.yesNode.checked = true;
                this.noNode.checked = false;
            } else {
                // The input value is no, check the no box and uncheck the yes box.
                this.yesNode.checked = false;
                this.noNode.checked = true;
            }
        },
        _getValueAttr: function() {
            if (this.yesNode.checked) {
                return true;
            } else if (this.noNode.checked) {
                return false;
            } else {
                return null;
            }
        }
    });
});
```

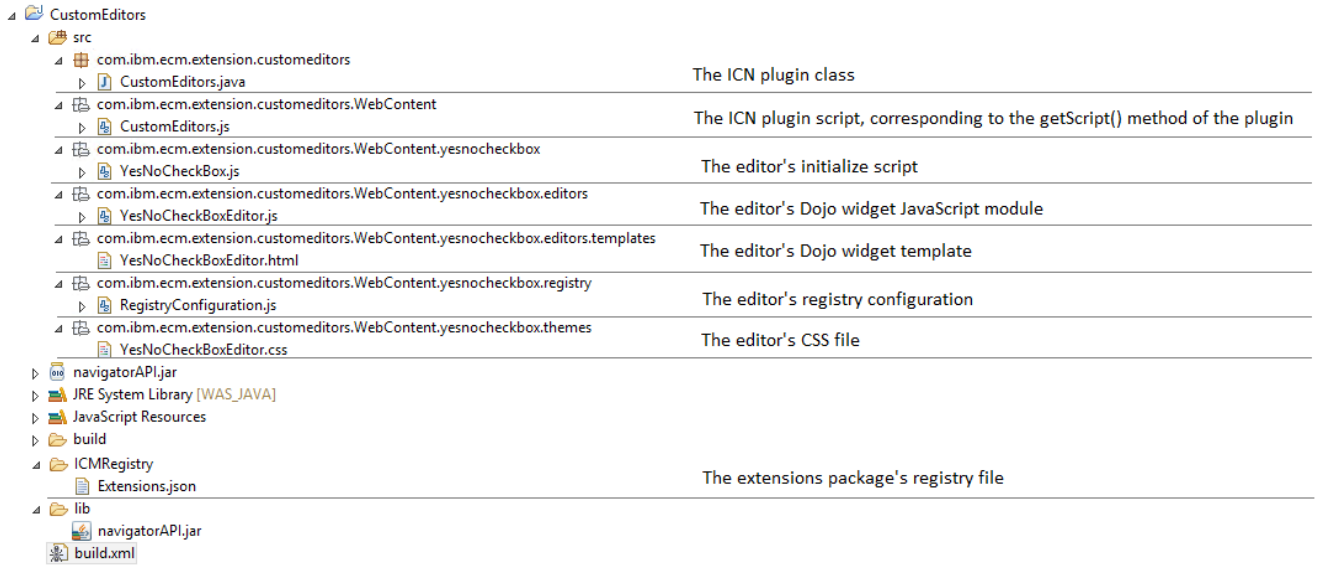
Optionally you can create CSS to beautify your widget. If you want the widget to inherit the system's theme, just create an empty CSS file. Test your widget in browser following the standard dojo practice. You can refer to dojo's documentation on how to run a widget in browser.

Create the extensions project

When your dojo widget is ready, the next step is to create an extensions package project and put the widget in it. We recommend you follow the directory structure below to create your project.

Folder	**Content**
ProjectName/*ProjectName*	Contains the files that are used to create the JAR file for the IBM Content Navigator plug-in.
ProjectName/*ProjectName*/src/*PackageName*	Contains the files that are used to create the JAR file for the IBM Content Navigator plug-in.
ProjectName/*ProjectName*/src/*PackageName*/WebContent/	Contains the main JavaScript plug-in file and the root folder of your custom editors and controller code. It can have subfolder structures to organize the code packages.
ProjectName/ICMRegistry	Contains the Extension.json file that are used to register the extensions package
ProjectName/lib	Contains navigatorAPI.jar and other jar's depended by your code

In this example, we create the project structure and put the files in as below. Please see the source code attached for the details. We also provide a sample build script so that you can build the code easily.



The ICMRegistry/Extensions.json and the ICN plugin script (WebContent/CustomEditor.js) serve as manifest of the extensions in the extensions package for Case Builder and Case Client respectively. They consists of the similar information, and you should always keep them in sync during your development. In this example, both files have the information for the custom Yes/No check box editor as below

Extensions.json:

```
{
    "Name": "Custom Editors",
    "Description": "A set of custom editor(s).",
    "Locale": "",
    "Version": "5.2.1",
    "Extensions": [{
        "id": "CustomYesNoCheckBox",
        "title": "Custom Yes/No Check Box Editor",
        "description": "Use with Boolean properties; can be set to Yes, No, or neither.",
        "type": "ViewDesignerExtensions",
        "packages": {
            "yesnocheckbox": "/navigator/plugin/CustomEditors/getResource/yesnocheckbox"
        },
        "css": [
            "/navigator/plugin/CustomEditors/getResource/yesnocheckbox/themes/YesNoCheckBoxEditor.css"
        ],
        "bootstrapModule": "yesnocheckbox/YesNoCheckBox",
        "bootstrapMethod": "initialize"
    }]
}
```

CustomEditor.js:

```
require([
    "dojo/_base/lang", "dojo/_base/array", "dojo/dom-construct",
    "dojo/sniff", "dojo/promise/all", "dojo/Deferred"
], function(lang, array, domConstruct, sniff, all, Deferred) {
    // Define the extensions packages to be loaded.
    var extensions = [{
```

```
        packages: {
            yesnocheckbox: "/navigator/plugin/CustomEditors/getResource/yesnocheckbox"
        },
        css: [
            "/navigator/plugin/CustomEditors/getResource/yesnocheckbox/themes/YesNoCheckBoxEditor.css"
        ],
        bootstrapModule: "yesnocheckbox/YesNoCheckBox",
        bootstrapMethod: "initialize"
    }];

    // !!! DO NOT MODIFY !!! Common code to load the extension packages.
    var promises = [];
    array.forEach(extensions, function(extension) {
        // Code to load the the extension
        // ...
    });
    all(promises);
});
```

The two files are loaded when initializing properties view designer in Case Builder and loading Case Client desktop, respectively, and the bootstrap modules specified in the files will be loaded inconsequentially. For each bootstrap module, the bootstrap method will be invoked when the module is loaded. In this example, we merge the Registry configuration for the custom Yes/No editor as below. You can add more initialization for your editor in this module if needed.

YesNoCheckBox.js:

```
define([
        "dojo/_base/declare",
        "dojo/has!icm-builder?
icm/propsdesign/ui/registry/RegistryManager:icm/widget/properties/registry/RegistryManager",
        "yesnocheckbox/registry/RegistryConfiguration"
], function(declare, RegistryManager, registryConfig) {
        return declare("custom.YesNoCheckBox", null, {
                // Bootstrap method.
                initialize: function() {
                        // Merge the registry for the custom editor.
                        RegistryManager.mergeConfiguration(registryConfig);
                        // More initialization code as you need..
                }
        });
});
```

Now let's see the registry configuration for the custom Yes/No check box editor. We create the file according to the schema we provided previously. The editor is defined in the editors node, and we map it to single boolean properties in the mappings node.

RegistryConfiguration.js:

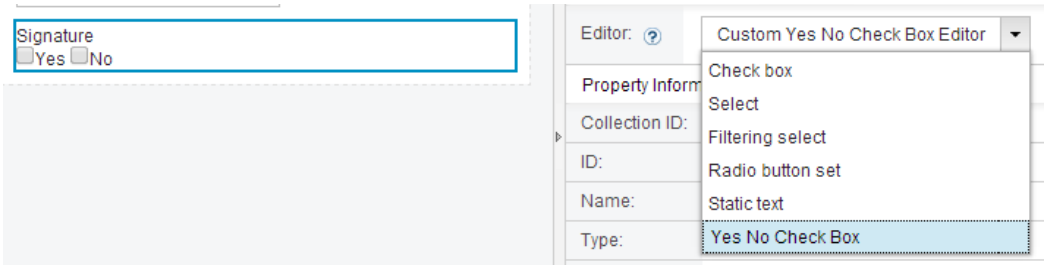
```
define([
        "dojo/_base/declare",
        "yesnocheckbox/editors/YesNoCheckBoxEditor"
], function(declare, YesNoCheckBoxEditor) {

        return {
                editors: {
                        editorConfigs: {
                                "customYesNoCheckBoxEditor": {
                                        label: "Yes No Check Box",
                                        editorClass: YesNoCheckBoxEditor
```

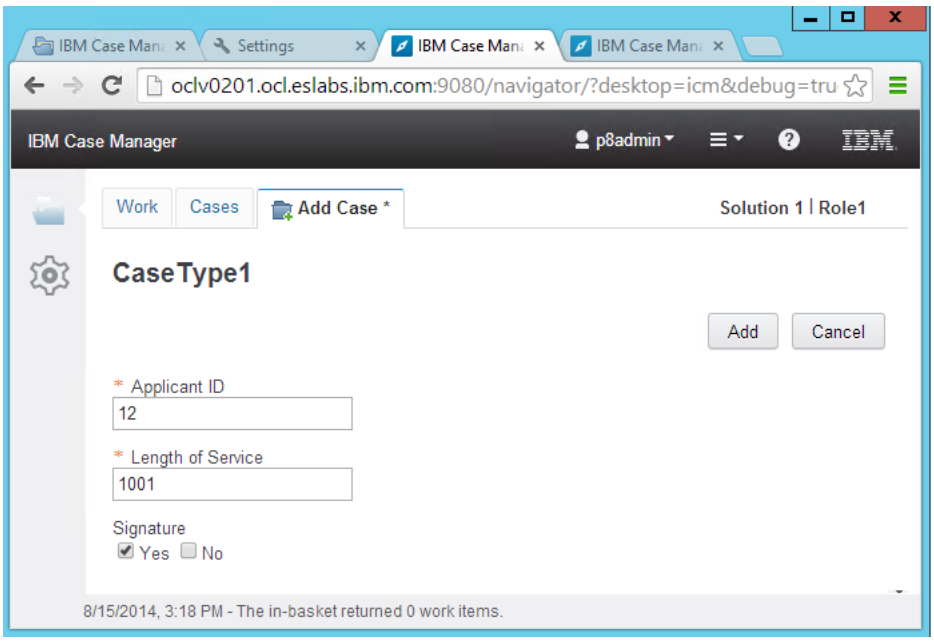
```
                        }
                },
                mappings: {
                        types: {
                                "boolean": {
                                        single: {
                                                editorConfigs: [
                                                        "customYesNoCheckBoxEditor"
                                                ]
                                        }
                                }
                        }
                }
        }
});
```

Deploy and test the extensions package

When you have the code build successfully into an extensions package in a zip file, you can deploy the package using the “Deploy and Register Extensions Package” task in IBM Case Manage Configuration tool. After that, the custom editor is ready to test. Login to Case Builder and open the properties view designer, you will see the custom editor is available when you choose a boolean property, as below.



Deploy the view, and you will be able to use it in Case Client.



Please note that the editor resides in Case Client as a IBM Navigator plugin, you need to keep Case Client up and running during you use properties view designer in Case Builder. There is a chance that you get an error for the very first time you open the properties view designer after you restart the application server of Case Client. If you meet this error, please login to Case Client, and then you can continue to use properties view designer in Case Builder.

Appendix

We provide the schema of the registry configuration for your reference. Some of the configurations are out of the scope of this article. There will be another article for advanced customization scenarios and the comprehensive customization features supported.

Schema of Registry configuration

Registry configuration is a JavaScript object conforming to the following schema. You should follow this schema to create the registry configuration for your custom editor. The configuration is merged into the system built-in configuration and you only need to provide the delta structure of your configuration, but not the whole structure. (e.g., you don't have to provide the choices structure in the configuration if your editor does not use choice list.)

```
editors: {
	editorConfigs: {
		"<Editor name>": {
			// Required attributes.
			label: "<Editor label displayed in properties view designer>",
			editorClass: <Editor class. Must be loaded in require>,
			// Optional attributes.
			defaultFieldWidth: "16px",
			defaultColumnWidth: "60px"
			... // See the attributes list in the next section for details.
		},
		...
	},
	mappings: {
		// Editors mapping by types
		types: {
			"<type>": {
				// Editors to work with single-valued properties
				single: {
					editorConfigs: [
						"<Editor name 1>",
						"<Editor name 2>",
						...
					],
					defaultEditorConfig: "<Default Editor name>"
				},
				// Editors to work with multi-valued properties
				multi: {
					editorConfigs: [
						"<Editor name 1>",
```

```
                                "<Editor name 2>",
                                ...
                        ],
                        defaultEditorConfig: "<Default Editor name>"
                }
        },
        ...
},
// Editors that work with choice lists
choices: {
        // Flat choices (choice lists that do not have inter-dependency on each other)
        flat: {
                // Editors to work with single-valued properties
                single: {
                        editorConfigs: [
                                "<Editor name 1>",
                                "<Editor name 2>",
                                ...
                        ],
                        defaultEditorConfig: "<Default Editor name>"
                },
                ...
                // Editors to work with multi-valued properties
                multi: {
                        editorConfigs: [
                                "<Editor name 1>",
                                "<Editor name 2>",
                                ...
                        ],
                        defaultEditorConfig: "<Default Editor name>"
                }
                ...
        },
        hierarchical: {
                single: {
                        editorConfigs: [
                                "dropDownTreeSelect"
                        ],
                        defaultEditorConfig: "dropDownTreeSelect"
                },
                multi: {
                        editorConfigs: [
                                "dropDownTreeSloshBucket"
                        ],
                        defaultEditorConfig: "dropDownTreeSloshBucket"
                }
        }
},
// Outer wrapper editors for multi-valued properties
multiEditors: {
        editorConfigs: [
                "<Multi-valued editor 1>",
                "<Multi-valued editor 2>",
```

```
                        ...
                    ],
                    defaultEditorConfig: "<Default Editor name>"
                }
            }
}
```

Attributes in the editorConfigs

Attribute	Value	Default value
label	Editor label displayed in properties view designer. The value for the attribute is required.	<Value required>
editorClass	The dojo class of the editor. The class must be loaded ahead in the require statement. The value for the attribute is required	<Value required>
formatterClass	The JavaScript module to format the value in the editor. The module must be a sub class of pvr/widget/editors/formatters/Formatter.	
falseIfNull	Indicates whether this editor class unconditionally coerces null values to false because it is unable to model the null value. The CheckBoxEditor is an example of an editor which supports this attribute. Note: This setting should not be confused with the falseIfNull attribute of a property controller or the falseIfNull setting of other editor classes that allow the falseIfNull functionality to be configurable.	false
zeroIfNull	Indicates whether this editor class unconditionally coerces null values to zero because it is unable to model the null value. There are currently no examples of editor classes with this attribute. Note: This setting should not be confused with the zeroIfNull attribute of a property controller or the zeroIfNull setting of other editor classes that allow the zeroIfNull functionality to be configurable.	false
minimumFieldWidth	The minimum input field width of the editor. If there is insufficient room for the field width configured in the view definition, the editor will shrink as required to fit its container. However, it will never shrink below this minimumFieldWidth setting.	"16px"
defaultFieldWidth	The default input field width of the editor. This field width will be applied if a specific field width was not configured in the view definition.	"16px"
defaultColumnWidth	The default container column width of the editor. This field width will be applied in property table columns, multiple value editor controls and in horizonatally aligned Layout containers if a specific field width is not configured in the view definition.	"60px"
settings	A list of setting definitions supported by the editor. A setting control for each setting definition is rendered in the settings panel of the view designer	

	and can be used to modify the specified setting in the view definition.
	
	You can specify the name of the setting, the control to present and edit the setting, and the parameters for the control. The control class must be loaded ahead in the require statement.
	[
	{
	name: <setting name 1>,
	controlClass: <The control to present/edit the setting>,
	controlParams: {
	//parameters for the setting's control.
	}
	},
	{
	name: <setting name 2>,
	controlClass: <The control to present/edit the setting>,
	controlParams: {
	//parameters for the setting's control.
	}
	},
	...
	]
	
	Specific details will be provided in a later article.	

supportsLabel	Indicates whether the editor itself should typically display the property's label. CheckBoxEditor is an example of an editor that displays the label.	false