

# Eclipse Scout JS

## *Technical Guide*

Scout Team

Version 8.0

# Table of Contents

Introduction .....	1
1. Overview .....	2
2. Widget .....	3
2.1. Lifecycle .....	3
2.2. Creating a Widget .....	3
2.3. Creating a Widget using JSON .....	4
2.4. Finding a Widget .....	6
2.5. Properties .....	7
2.6. Widget Properties .....	7
2.7. Events .....	9
2.8. Icons .....	11
2.9. Parent and Owner .....	11
3. Object Factory .....	13
4. Form .....	14
4.1. Form Lifecycle .....	14
5. Form Field .....	16
6. Value Field .....	17
6.1. Parser, Validator, Formatter .....	17
7. Lookup Call .....	20
8. Styling .....	21
9. Extensibility .....	23
9.1. How to extend Scout objects .....	23
9.2. Export Scout model from a Scout classic (online) application .....	24
10. Widget Reference .....	26
10.1. Smart Field .....	26
11. Browser Support .....	29
12. How-Tos .....	30
12.1. How to Create a Custom field .....	30
Appendix A: Licence and Copyright .....	38
A.1. Licence Summary .....	38
A.2. Contributing Individuals .....	38
A.3. Full Licence Text .....	39

# Introduction

This technical guide documents the Scout JS architecture and describes important concepts.

!

This document is referring to a past Scout release. Please click [here](#) for the recent version.

Looking for something else? Visit <https://eclipsescout.github.io> for all Scout related documentation.

"

This is the guide to Scout JS, the JavaScript part of Eclipse Scout. If you are looking for the Java part please see the technical guide for Scout Classic: <https://eclipsescout.github.io/8.0/technical-guide.html>

!

This document is not complete. Contributions are welcome!  
If you like to help, please create a pull request. Thanks!

Repository:  
<https://github.com/bsi-software/org.eclipse.scout.docs>

# Chapter 1. Overview

A classic Scout application has a client model written in Java, and a UI which is rendered using JavaScript. With this approach you can write your client code using a mature and type safe language. Unfortunately you cannot develop applications which have to run offline because the UI and the Java model need to be synchronized.

With Scout JS this will change. You will be able to create applications running without a UI server because there won't be a Java model anymore. The client code will be written using JavaScript and executed directly in the browser.

You still don't have to care about how the model will be rendered. There isn't a strict separation of the UI and the model anymore, but that doesn't mean you have to write HTML and CSS. You can of course, if you want to. But typically you will be using the Scout widgets you already know.

Scout JS is used in classic Scout applications as well. This means if you understand the concepts of Scout JS, writing custom widgets will be a lot easier.

# Chapter 2. Widget

A widget is a component which may be rendered. It may be simple like a label, or more complex like a tree or table. A form is a widget and a form field, too. A widget contains the model, which represents the state of the widget. In a Scout Classic application, that model will be sent from the UI server to the browser and the Scout UI will use that model to create the widget. In a Scout JS app, the model may be provided using JSON or directly with JavaScript.

## 2.1. Lifecycle

Every widget has a lifecycle. After a widget is instantiated, it has to be initialized using `init`. If you want to display it, you have to call the `render` method. If you want to remove it from the DOM, call the `remove` method. Removing a widget is not the same as destroying it. You can still use it, you can for example change some properties and then render it again. If you really don't need it anymore, call the `destroy` method.

So you see the widget actually has 3 important states:

¥ initialized

¥ rendered

¥ destroyed

The big advantage of this concept is that the model of the widget may be changed any time, even if the widget is not rendered. This means you can prepare a widget like a form, prepare all its child widgets like the form fields, and then render them at once. If you want to hide the form, just remove it. It won't be displayed anymore, but you can still modify it, like changing the label of a field or adding rows to a table. The next time it is rendered the changes will be reflected. If you do such a modification when it is rendered, it will be reflected immediately.

Destroying a widget means it will detach itself from the parent and destroy all children. If you have attached listeners to other widgets at initialization time, now is the time to detach them. After a widget is destroyed it cannot be used anymore. Every attempt will result in a `Widget is destroyed` error.

## 2.2. Creating a Widget

A widget may be created using the constructor function or `scout.create`. Best practice is to always use `scout.create` which gives you two benefits:

1. You don't have to call `init` by yourself.
2. The widget may be extended (see [Chapter 3](#) for details).

The following example creates a `StringField`.

### Listing 1. Creating a string field

```
var field = scout.create('StringField', {  
  parent: groupBox,  
  label: 'hello',  
  value: 'world'  
});
```

The first parameter is the object type, which typically is the name of the constructor function preceded by the name space. `StringField` belongs to the `scout` name space which is the default and may be omitted. If the string field belonged to another name space called `mynamespace`, you would have to write the following:

### Listing 2. Creating a field considering the name space

```
scout.create('myspace.StringField', {})
```

The second parameter of `scout.create` is the model. The model actually is the specification for your widget. In case of the `StringField` you can specify the label, the max length, whether it is enabled and visible and more. If you don't specify them, the defaults are used. The only needed property is the `parent`. To see what the defaults are, have a look at the source code of the widget constructor.

Every widget needs a parent. The parent is responsible to render (and remove) its children. In the example above, the parent is a group box. This group box has a property called `fields`. If the group box is rendered, it will render its fields too.

You don't need a group box to render the string field, you could render it directly onto the desktop. But if you want to use a form, you need a group box and create the form, group box and the field. Doing this programmatically is time consuming, that is why we suggest to use the declarative JSON based approach.

## 2.3. Creating a Widget using JSON

Using the JSON based approach makes it easier to specify multiple widgets in a structured way. The following example defines a form with a group box and a string field.

*Listing 3. A form model defined using JSON*

```
{
  "id": "example.MyForm",
  "type": "model",
  "title": "My first form!",
  "rootGroupBox": {
    "id": "MainBox",
    "objectType": "GroupBox",
    "fields": [
      {
        "id": "MyStringField",
        "objectType": "StringField",
        "label": "hello",
        "value": "world"
      }
    ]
  }
}
```

This description of the form is put in a separate file called `MyForm.json`. Typically you would create a file called `MyForm.js` as well, which contains the logic to interact with the fields. But since we just want to open the form it is not necessary. Instead you can use the following code to create the form:

```
var form = scout.create('Form', scout.models.getModel('example.MyForm', parent));
```

When the form is shown it will look like this:

*Figure 1. A form described using JSON*

### 2.3.1. Using constants in JSON

When you define widget properties in a JSON model, you often want to set a property to a value defined by a constant. When you're writing JavaScript code you'd simply write:

```
field.labelPosition = scout.FormField.labelPosition.ON_FIELD;
```

Since you cannot reference Scout's JavaScript enums/constants in JSON, you'd have to write the following to assign the constant value in the JSON model:

```
{ "labelPosition": 2 }
```

Obviously this makes the JSON code harder to read and understand and the developer has to lookup the enum/constant definition first. This is where the `${const:NAME}` tag comes to help. A widget can define a list of properties that can be used with the const-tag. The widget defines in which enum/constant the name provided by the tag is resolved. Using the const-tag, the JSON example now looks like this:

```
{ "labelPosition": "${const:ON_FIELD}" }
```

When you want to provide const-tag support for a custom widget, you need to call `scout.Widget#resolveConsts` in the `_init` function of your widget and for each supported property specify the property name and the object that contains the enum/constant definition.

## 2.4. Finding a Widget

In the example above we have seen how to create a widget, in that specific case we created a form. Typically it is not sufficient to just create a form, you most likely want to interact with the fields, like reading the values the user entered. In order to do that you need access to the fields. The easiest way is to use the IDs specified in the JSON.

Let's have a look at our example form again:

```
{
  "id": "example.MyForm",
  "type": "model",
  "title": "My first form!",
  "rootGroupBox": {
    "id": "MainBox",
    "objectType": "GroupBox",
    "fields": [
      {
        "id": "MyStringField",
        "objectType": "StringField",
        "label": "hello",
        "value": "world"
      }
    ]
  }
}
```

In this example we have 3 widgets: the form, the root group box and a string field. These widgets are linked to each other which enables us to find the string field starting from the form. This can be done by using the following command:

```
var stringField = form.widget('MyStringField');
```

Now you can read its value, change properties, add event handlers and so on.

## 2.5. Properties

As seen before, every widget has a model representing its state. This model is written onto the widget at the time it is being instantiated. The properties of that model are now available as properties of the widget. So in order to access such a property, just call `widget.yourProperty`. If you want to modify the property, just call `widget.setYourProperty(value)`.

*Listing 4. Accessing and modifying a property*

```
var field = scout.create('scout.StringField', {
  parent: scout.sessions[0].desktop,
  labelVisible: false
});
console.log(field.labelVisible); // prints false

field.setLabelVisible(true);
console.log(field.labelVisible); // prints true
```

It is important to always use the setter to modify a property, because calling it does not just change the value. Instead it will call the method `setProperty(propertyName, value)` which does the following:

1. It will check if the property has changed at all. If the value is still the same, nothing happens. To compare the values `scout.objects.equals` is used, which uses `===` to compare and if that returns false, uses the equals methods of the given objects, if available.
2. If the values are not equal, the model is updated using the method `_setProperty` (notice the `_`). Beside setting the value it also notifies every listener about the property change. So if another widget is interested in that property it may attach a listener and will be informed on every property change (see also the [Events](#) for details).
3. In order to reflect the property change in the UI, the `_render` method is called, if available. The name of this method depends on the property name, it always starts with `_render` and ends with the property name. Example: `_renderLabelVisible`. If the widget does not implement such a method, nothing happens.

It is worth to mention that the behavior of step 2 may be influenced by the widget. If the widget provides a method called `_setPropertyName` (e.g. `_setLabelVisible`, notice the `_`), that method will be called instead of `_setProperty`. This may be useful if something other should be done beside setting the property. If that is the case, that new function is responsible to call `_setProperty` by itself in order to set the property and inform the listeners. That method may also be called by the `_init` method to make sure the additional code is also executed during initialization (calling the public setter in `_init` would not have any effect due to the equals check at the beginning).

## 2.6. Widget Properties

A widget property is a special kind of a property which references another widget.

Defining a property as widget property has the benefit that the widget is created automatically. Lets take the group box as an example. A group box has a widget property called fields. The fields are widgets, namely form fields. If I create a group box, I may specify its fields directly:

*Listing 5. Creating the string field automatically using a widget property*

```
var groupBox = scout.create('GroupBox', {
  parent: scout.sessions[0].desktop,
  label: 'My Group Box',
  fields: [{
    objectType: 'StringField',
    label: 'My String Field'
  }]
});
// check if the string field was created as well
console.log(groupBox.fields[0] instanceof scout.StringField);
```

In the above example the group box is created using `scout.create`. After creating the group box you can access the property fields and you will notice that the string field was created as well, even though `scout.create` has not been called explicitly for the string field. This is because the property `fields` is defined as widget property. During the initialization of the group box it sets the property `fields` and because the value is not a widget yet (resp. the elements in the array), `scout.create` will be called.

This will also happen if you use a setter of a widget property. You can either call the setter with a previously created widget, or just pass the model and the widget will be created automatically.

In addition to creating widgets, calling such a setter will also make sure that obsolete widgets are destroyed. This means if the widget was created using the setter, it will be destroyed when the setter is called with another widget which replaces the previous one. If the widget was created before calling the setter, meaning the `owner` is another widget, it won't be destroyed.

So if a property is defined as widget property, calling a setter will do the following:

1. It checks if the property has changed at all (same as for regular properties).
2. If the values are not equal, `_prepareWidgetProperty` is called which checks if the new value already is a widget and if not creates it. It also destroys the old widget unless the property should not be preserved (see `_preserveOnPropertyChangeProperties`). If the value is an array, it does so for each element in the array (only widgets which are not part of the new array will be destroyed).
3. If the widget is rendered, the old widget is removed unless the property should not be preserved. If there is a custom remove function (e.g. `_removeXY` where XY is the property name), it will be called instead of removing the widgets directly. Note that the widget may have already been removed by the destroy function at the prepare phase.
4. The model is updated (same as for regular properties).
5. The render method is called (same as for regular properties).

## 2.7. Events

Every widget supports event handling by using the class `scout.EventSupport`. This allows the widgets to attach listeners to other widgets and getting informed when an event happens.

The 3 most important methods are the following:

1. `on`: adds a listener
2. `off`: removes a listener
3. `trigger`: triggers an event

So if a widget is interested in an event of another widget, it calls the function `on` with a callback function as parameter. If it is not interested anymore, it uses the function `off` with the same callback function as parameter.

The following example shows how to handle a button click event.

*Listing 6. Handling an event*

```
var button = scout.create('scout.Button', {
  parent: scout.sessions[0].desktop,
  label: 'click me!'
});
button.render();
button.on('click', function(event) {
  // print 'Button "click me!" has been clicked'
  console.log('Button "' + event.source.label + '" has been clicked');
});
```

Every click on the button will execute the callback function. To stop listening, you could call `button.off('click')`, but this would remove every listener listening to the 'click' event. Better is to pass the same reference to the callback used with `on` as parameter for `off`.

*Listing 7. Stop listening for an event*

```
var button = scout.create('scout.Button', {
  parent: scout.sessions[0].desktop,
  label: 'click me!'
});
button.render();
var callback = function(event) {
  // print 'Button "click me!" has been clicked'
  console.log('Button "' + event.source.label + '" has been clicked');

  // stop listening, a second click won't print anything
  button.off('click', callback);
};
button.on('click', callback);
```

”

If the callback function is bound using `bind()`, the bound function has to be used when removing the listener using `off`. This is because `bind()` returns a new function wrapping the original callback.

In order to trigger an event rather than listening to one, you would use the function `trigger`. This is what the button in the above example does. When it is being clicked, it calls `this.trigger('click')` (`this` points to the instance of the button). With the second parameter you may specify additional data which will be copied onto the event. By default the event contains the type (e.g. 'click') and the source which triggered it (e.g. the button).

*Listing 8. Triggering an event with custom event data*

```
trigger('click', {  
  foo: 'bar'  
});  
  
// callback  
function(event) {  
  console.log(event.foo); // prints bar  
}
```

### 2.7.1. Property Change Event

A special kind of event is the property change event. Whenever a property changes, such an event is triggered.

The event has the following properties:

1. `type`: the type of the event which is always `propertyChange`
2. `source`: the widget which triggered the event
3. `name`: the name of the property
4. `newValue`: the new value of the property
5. `oldValue`: the old value of the property

Listening to such an event works in the same way as for other events, just use the type `propertyChange`. The listening below shows how to handle the property change event if the `selected` property of a toggle button changes.

Listing 9. Example of a property change event

```
var button = scout.create('scout.Button', {
  parent: scout.sessions[0].desktop,
  label: 'click me!',
  displayStyle: scout.Button.DisplayStyle.TOGGLE
});
button.render();
button.on('propertyChange', function(event) {
  // prints 'Property selected changed from false to true'
  console.log('Property ' + event.propertyName + ' changed from ' + event.oldValue + '
to ' + event.newValue);
});
button.setSelected(true);
```

## 2.8. Icons

See chapter [Section 2.8](#) for a general introduction to icons in Scout.

Widgets that have a property `iconId` (for instance `Menu`) can display an icon. This `iconId` references an icon which can be either a bitmap image (GIF, PNG, JPEG, etc.) or a character from an icon-font. An example for an icon-font is the *scoutIcons.ttf* which comes shipped with Scout.

Depending on the type (image, font-icon) the `iconId` property references:

¥ Image: `iconId` is an URL which points to an image resource accessible via HTTP.

Example: `/icons/person.png`

¥ Font-icon: `iconId` has the format `font: [UTF-character]`.

Example: `font: \uE043`, references a character in *scoutIcons.ttf*

Example: `font: foolcons \uE109`, references a character in custom font *foolIcons.ttf*

¥ Icon Constants: `iconId` has the format: `${iconId: [constant]}`, where `constant` is a constant in `scout.icons`. This format is especially useful when you configure a Scout widget with a JSON model. The value of the constant is again either an image or a font-icon as described above.

Example: `${iconId: ANGLE_UP}` uses `scout.icons.ANGLE_UP`, icons predefined by Scout

Example: `${iconId: foo.BAR}` uses `foo.icons.BAR`, use this for custom icon constant objects

## 2.9. Parent and Owner

As seen in the previous chapters, the creation of a widget requires a parent. This establishes a link between the child and the parent widget which is necessary for several actions.

1. Rendering a widget into the container of the parent

If you call `widget.render()` you don't have to specify the HTML container to which the widget should be appended. It takes the container of the parent widget by default which is `parent.$container`. You can still pass a custom `$parent` if you like.

## 2. Removing a widget and its children

If `widget.remove()` is called, the widget will be removed from the DOM. Even though removing the parent HTML node would be sufficient in order to make the children disappear from the screen, every child widget will be removed as well. This gives the child widgets the possibility to clean up their rendering state and detach listeners. This is necessary because the widgets still exist after removal and their data can still be updated. Such cleanup actions are typically done by overriding `widget._remove()`. Remember to call the `_remove` of the super class as well.

## 3. Finding child widgets

This is something you will have to do very often if you specified your widgets in a JSON file. If you want to access these widgets from JavaScript you need to get them first. This is easy due to the linkage of the widgets. You can just use the function `widget(id)`. See also chapter [Section 2.4](#) for more details.

## 4. Establishing a link independent of the DOM structure

Normally a child widget contains HTML elements which are children of the parent's HTML elements. Even though this is the regular case it is not a must. Amongst others the following widgets have HTML elements appended to the HTML element of the desktop rather than their actual parents: dialogs, popups, tooltips. The main reason is because these widgets lay on top of every other widget. In order to not get into the z-index hell it is a lot easier to put these widgets on the top level of the DOM structure. But since the HTML elements are now separated, the only link is the parent/child hierarchy.

Imagine you have a string field which displays a tooltip. The parent of the tooltip is the string field but the HTML element is appended to the HTML element of the desktop. Removing the string field will now remove the tooltip as well even though their HTML elements are not linked.

Or think about the following case: there is a popup open which contains a smart field. The smart field shows a popup by itself displaying the proposals. The HTML elements of the main popup and the smart field popup are siblings meaning they are on the same level in the DOM. Normally a popup gets closed when an element not belonging to the popup is clicked. But why is the main popup not closed if the user clicks into the smart field popup even though their HTML elements are not linked? Exactly, because the smart field popup is a child of the smart field and therefore a child of the main popup.

So far we have learned what the parent is. But what is the owner? The owner is the only one which is allowed to destroy its children. Normally, the parent and the owner are the same, that is why you don't have to specify the owner explicitly when creating a widget. The owner will be different if you specify it explicitly or if you use `setParent()` to temporarily change the parent of a widget. In that case the owner points to the old parent. This means if the new parent were destroyed, the newly linked child would not be destroyed, only removed from the DOM.

This comes in handy if you want to temporarily give the control over rendering/removal to another widget (like a popup) but don't let the other widget destroy your widget (e.g. when the popup is closed) because you want to use your widget again (e.g. display it on another popup).

# Chapter 3. Object Factory

As seen in the [Section 2.2](#) a widget may be created using `scout.create`. When using this function, the call is delegated to the `ObjectFactory`.

The `ObjectFactory` is responsible to create and initialize a Scout object. A typical Scout object has an `objectType` and an `init` function. But actually any kind of object with a constructor function in the scout or a custom namespace may be created.

By default, objects are created using naming convention. This means when calling `scout.create('scout.Table', model)`, the `scout` namespace is searched for a class called `Table`. Since `scout` is the default namespace, it may be omitted. So calling `scout.create('Table', model)` has the same effect. If there is such a class found, it will be instantiated and the `init` function called, if there is one. The `model` is passed to that `init` function. So instead of using `scout.create` you could also use the following code:

*Listing 10. Creating an object without the ObjectFactory*

```
var table = new scout.Table();
table.init(model);
```

This will work fine, but you will lose the big benefit of the `ObjectFactory`: the ability to replace existing classes. So if you want to customize the default `Table`, you would likely extend that table and override some functions. In that case you need to make sure every time a table is created, your class is used instead of the default. To do that you have to register your class in the `ObjectFactory` with the `objectType Table`. If `scout.create('Table')` is called the object factory will check if there is a class registered for the type `Table` and, if yes, that class is used. Only if there is no registration found, the default approach using the naming convention is performed.

In order to register your class, you need a file called `objectFactories` and add that to your JavaScript module (e.g. `yourproject-module.js`). The content of that file may look as following:

*Listing 11. Adding a new object factory registration*

```
scout.objectFactories = $.extend(scout.objectFactories, {
  'Table': function() {
    return new yourproject.CustomTable();
  }
});
```

This will simply add a new factory for the type `Table` to the list of existing factories. From now on `yourproject.CustomTable` will be instantiated every time a `Table` should be created.

# Chapter 4. Form

A form is typically used for two purposes:

1. Allowing the user to enter data in a structured way
2. Displaying the data in a structured way

This is achieved by using [Chapter 5s](#). Every form has one root group box (also called main box) which has 1:n form fields. The form fields are layouted using the logical grid layout, unless no custom layout is used. This makes it easy to arrange the fields in a uniform way.

A form may be displayed in various ways, mainly controlled by the property `displayHint`. The following display hints are available by default:

- ¥ view: the form will be opened in a tab and will take full width and height of the bench
- ¥ dialog: the form will be opened as overlaying dialog and will be as width and height as necessary
- ¥ popup-window: the form will be opened in a separate browser window (please note that this feature does not work properly with Internet Explorer)

To display the form, just set one of the above display hints and call `form.open()`.

Beside opening the form as separate dialog or view, you can also embed it into any other widget because it is actually a widget by itself. Just call `form.render()` for that purpose.

## 4.1. Form Lifecycle

When working with forms, you likely want to load, validate and save data as well. The form uses a so called `FormLifecycle` to manage the state of that data. The lifecycle is installed by default, so you don't have to care about it. So whenever the user enters some data and presses the save button, the data is validated and if invalid, a message is shown. If it is valid the data will be saved. The following functions of a form may be used to control that behavior.

- ¥ open: calls `load` and displays the form once the loading is complete.
- ¥ load: calls `_load` and `importData` which you can implement to load the data and then marks the fields as saved to set their initial values. Finally, a `postLoad` event is fired.
- ¥ save: validates the data by checking the mandatory and validation state of the fields. If every mandatory field is filled and every field contains a valid value, the `exportData` and `_save` functions are called which you can implement to save the data. After that every field is marked as saved and the initial value set to the current value.
- ¥ reset: resets the value of every field to its initial value marking the fields as untouched.
- ¥ ok: saves and closes the form.
- ¥ cancel: closes the form if there are no changes made. Otherwise it shows a message box asking to save the changes.
- ¥ close: closes the form and discards any unsaved changes.

✖ `abort`: called when the user presses the "x" icon. It will call `close` if there is a close menu or button, otherwise `cancel`.

If you need to perform form validation which is not related to a particular form-field, you can implement the `_validate` function. This function is always called, even when there is no *touched* field.

If you embed the form into another widget, you probably don't need the functions `open`, `ok`, `close`, `cancel` and `abort`. But `load`, `reset` and `save` may come in handy as well.

Because it is quite common to have a button activating one of these functions (like an 'ok' or 'cancel' button), the following buttons (resp. menus because they are used in the menu bar) are available by default: `OkMenu`, `CancelMenu`, `SaveMenu`, `ResetMenu`, `CloseMenu`.

# Chapter 5. Form Field

A form field is a special kind of a widget. It is mainly used on forms but may actually be added to any other widget.

Every form field contains of the following parts:

*Figure 2. Parts of a form field*

Typical form fields are `StringField`, `DateField` or `TableField`. All these fields have the API of `FormField` in common (like `setLabel()`, `setErrorStatus()`, etc.) but also provide additional API.

Some form fields are actually just a wrapper for another widget. This is for example the case for the `TableField`. The `Table` itself may be used stand-alone, just call `scout.create('Table', {})`. But if you want to use it in a `GroupBox`, which is a form field, you have to use a `TableField` wrapping the `Table`.

# Chapter 6. Value Field

A value field extends the form field by the ability to store a value. Typical form fields are `StringField`, `NumberField`, `DateField` or `SmartField`. All these fields provide a value which is accessible using `field.value` and may be set using `field.setValue(value)`.

## 6.1. Parser, Validator, Formatter

The value always has the target data type of the field. When using a `StringField` the type is `string`, when using a `NumberField` the type is `number`, when using a `DateField` the type is `date`. This means you don't have to care about how to parse the value from the user input, this will be done by the field for you. The field also validates the value, meaning if the user entered an invalid value, an error is shown. Furthermore, if you already have the value and want to show it in the field, you don't have to format the value by yourself.

This process of parsing, validating and formatting is provided by every value field. The responsible functions are `parseValue`, `validateValue` and `formatValue`. If a user enters text, it will be parsed to get the value with the correct type. The value will then be validated to ensure it is allowed to enter that specific value. Afterwards it will be formatted again to make sure the input looks as expected (e.g. if the user enters 2 it may be formatted to 2.0). If you set the value programmatically using `setValue` it is expected that the value already has the correct type, this means parse won't be executed. But the value will be validated, formatted and eventually displayed in the field.

Even though the fields already provide a default implementation of this functionality, you may want to extend or replace it. For that purpose you may set a custom parser and formatter or one or more validators.

### 6.1.1. Custom Parser and Formatter

Typically you don't have to add a custom parser or formatter for a `NumberField` or `DateField`. They work with a `DecimalFormat` or `DateFormat` which means you can specify a pattern how the number or date should be represented. By default, it uses the pattern of the current locale, so you don't even have to specify anything.

For a `StringField` on the other hand, adding a custom parser or formatter could make sense. Let's say you want to group the text into 4 digit blocks, so that if the user inputs 1111222233334444 it should be converted to 1111-2222-3333-4444. This could be done using the following formatter.

*Listing 12. Example of a formatter*

```
function formatter(value, defaultFormatter) {
  Ê var displayText = defaultFormatter(value);
  Ê if (!displayText) {
  Ê   return displayText;
  Ê }
  Ê return displayText.match(/.{4}/g).join('-');
};
```

Keep in mind that you should call the default formatter first unless you want to replace it completely.

To make your formatter active, just use the corresponding setter.

*Listing 13. Setting the formatter*

```
field.setFormatter(formatter);
```

Formatting the value is most of the time only half the job. You probably want to set a parser as well, so that if the user enters the text with the dashes it will be converted to a string without dashes.

*Listing 14. Example of a parser*

```
function parser(displayText, defaultParser) {  
  if (displayText) {  
    return displayText.replace(/-/g, '');  
  }  
  return defaultParser(displayText);  
};
```

Use the corresponding setter to activate the parser.

*Listing 15. Setting the parser*

```
field.setParser(parser);
```

### 6.1.2. Custom Validator

The purpose of a validator is to only allow valid values. This mostly depends on your business rules, this is why the default validators don't do a whole lot of things.

See the following example of a validator used by a `DateField`.

*Listing 16. Example of a validator*

```
function(value) {  
  if (scout.dates.isSameDay(value, new Date())) {  
    throw 'You are not allowed to select the current date';  
  }  
  return value;  
};
```

This validator ensures that the user may not enter the current date. If he does, an error status will be shown on the right side of the date field saying 'You are not allowed to select the current date'.

*Figure 3. Validation error of a date field*

As you can see in the example, in order to mark a value as invalid just throw the error message you want to show to the user. You could also throw an error or a `scout.Status` object. In that case a generic error message will be displayed.

In order to activate your validator, you can either call `setValidator` to replace the existing validator. In that case you should consider calling the default validator first, like you did it for the formatter or parser. Or you can use `addValidator` which adds the validator to the list of validators of the field.

*Listing 17. Adding a validator*

```
field.addValidator(validator);
```

Compared to parse and format you may have multiple validators. When the value is validated, every validator is called and has to agree. If one validation fails, the value is not accepted. This should make it easier to reuse existing validators or separate your validation into tiny validators according to your business rules.

If you now ask yourself, why this is not possible for parsing and formatting, consider the following: `Validate` takes a value and returns a value, the data type is the same for input and output. `Parse` takes a text and creates a value, `format` takes a value and creates a text. The data type is likely not the same (besides for the `StringField`). If you had multiple parsers, the output of the previous parser would be the input of the next one, so depending on the index of your parser you would either get the text or the already parsed value as input. Confusing, isn't it? So in order to keep it simple, there is only one parser and only one formatter for each field.

# Chapter 7. Lookup Call

A **Lookup Call** is used to lookup a single or multiple **Lookup Rows**. Several widgets like **Smart Field**, **List Box** or **Tree Box** take advantage of that concept in order to provide their proposals.

The most important parts of a Lookup Row are the key and the value. The key can be of any type, the text must be of type String. In addition to the key and the text a Lookup Row can also define an icon, a tooltip text, CSS classes and more.

Like in a classic Scout application each Smart Field in Scout JS references a LookupCall class. The lookup call is in charge of querying a data source and returning results for that query. Example: when you type "f" into a Smart Field, a lookup call could return a result which contains lookup rows starting with "F", like "Faz" and "Foo".

The lookup call may return static (hard-coded) data which is already available in the browser, or may fetch an external data-source via HTTP, typically some kind of REST API. Depending on how your Smart Field is configured and used, the Smart Field will call different methods on the LookupCall instance and pass data to that method, like the search text the user has typed into the field. These methods are: **getAll**, **getByText**, **getByKey** and **getByRec**.

- ¥ **getByKey()**: Retrieves a single lookup row for a specific key value. Used by Smart Fields and Smart Columns to get the display text for a given key value.
- ¥ **getByText()**: Retrieve multiple lookup rows which match a certain String. Used by Smart Fields when the user starts to enter some text in the field.
- ¥ **getAll()**: Retrieves all available lookup rows. Used by Smart Fields when the user clicks on the field.
- ¥ **getByRec()**: This can only be used for hierarchical lookup calls. It retrieves all available sub-tree lookup rows for a given parent.

You must implement these methods. Start with creating a sub class of **LookupCall** (.js). Sub class **StaticLookupCall** (.js) when you need a simple lookup call that operates on data that is available locally. Sub class **RemoteLookupCall** (.js) when you must fetch lookup data from a remote server. This class is also used in Scout Classic to start a lookup on the Scout UI Server.

Note that the lookup call works with *Deferreds*. This means the lookup call runs in the background and does not block the UI. When the lookup call is done eventually the Deferred is resolved and the Smart Field will process the result returned by the lookup call.

# Chapter 8. Styling

Beside JS for business logic and JSON for the models, every Scout JS app probably needs some CSS code at some point of time. If you are writing custom widgets, you need it for sure. But also if you are just using the given widgets you might have the need to adjust the look here and there.

Scout uses LESS as CSS preprocessor. It has a lot of advantages to pure CSS: variables, mixins, functions, imports etc. If you use the default build infrastructure provided by Scout, you can not only use LESS easily without getting a headache on how to integrate it in your build system, you get access to all of the LESS constructs used by Scout itself. This means you can use variables for colors, icons and sizes, mixins for animations and to avoid browser specific prefixes, you can import or replace whole Scout LESS files for custom themes and you get a lot of sample code in the Scout code base because Scout itself uses the same concepts.

In order to get started you need a place to put all your LESS code. The recommendation is to create one LESS file for each widget. That means if your widget is called `MyWidget.js` you would create a file called `MyWidget.less`. Even if you adjust the look of an existing widget you can create a file called let's say `StringField.less` and put the rules there. Remember to use a custom namespace (folder) in your project for your JS, JSON and LESS files otherwise it could get name clashes if you name your files the same way as Scout does. (Note: theme files are put in the default namespace, as you will learn in the theme chapter.) If you only have a few rules or non widget specific rules you can create a file called `main.less` and put them there. However, these are only recommendations, you can always do it as you like.

The creation of these files won't make them loaded automatically, you have to register them in your `module` file. The module should be put into your `src/main/js` folder and named according to your namespace, for example `helloWorld-module.less`. Just include your new files using the LESS `@import` keyword.

*Listing 18. Including a LESS file*

```
@import "helloWorld/style/colors.less";
```

This module file is either directly included in your `index.html` or in a `macro` file which is included in the `index.html`. The intention of the macro file is to group individual module files and make one file in order to load all rules at once within one request.

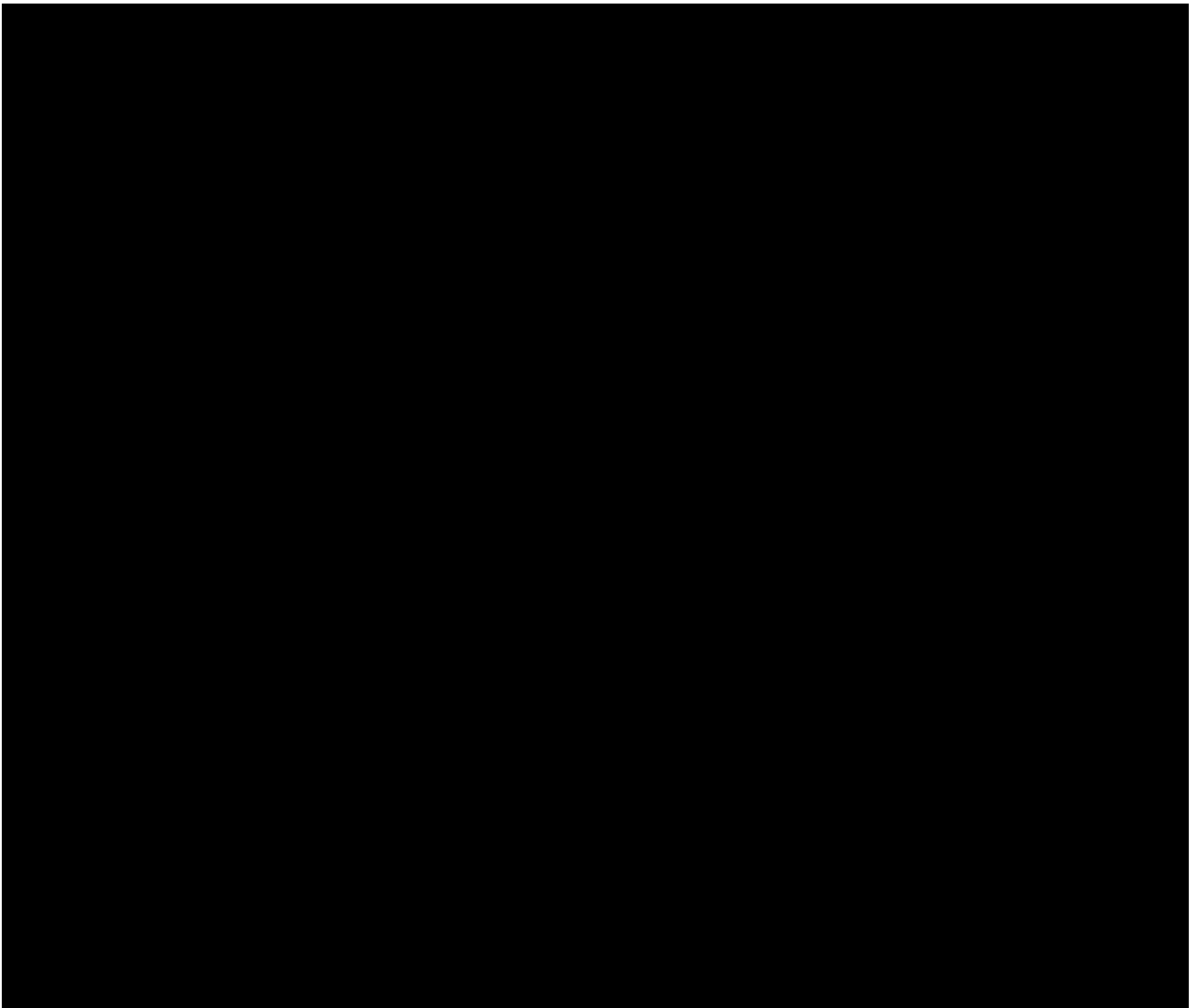
Now that we have all the infrastructure set up we can start adding some rules. As already said you can use all the LESS variables from Scout. The variables can be found in the `scout/style` folder of the `org.eclipse.scout.rt.ui.html` module. If you have a look at the file `colors.less` you find all the colors you can use or customize. Let's say you want to change the background color of the navigation, you can redefine the variable `@navigation-background-color`.

*Listing 19. Changing the background color of the navigation*

```
@navigation-background-color: @palette-red-4;
```

That's it! As you can see, changing this variable not only changes the background color of the

navigation, it also changes the background colors of the view-buttons on the top left and the navigation-handle on the bottom, because they are supposed to have the same color. You could do the same by using CSS rules, but you would have to write several selectors to achieve the same.



*Figure 4. Helloworld default and with a custom navigation background color*

In this example a color from the Scout color palette is used. You can also use a custom hex code instead of `@palette-red-4` or define a custom color palette if you want to use the same colors for different rules.

Beside colors there are some other files in the style folder of the Scout module: `animations.less`, `fonts.less`, `icons.less`, `mixins.less` and `sizes.less`. All these files contain variables or mixins which are used by various Scout widgets and may be used by your project as well.

Redefining a variable is the simplest way of customizing. If you need more control you can always write a custom CSS rule. Keep in mind that these rules need to be more specific than the default CSS rules of Scout, otherwise they won't be executed (see <https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity> for details). Beside that we cannot guarantee that your custom rules will still work when migrating to a newer Scout version, because changing only a small part of the rule might make it more specific so that your rule won't work anymore. That only applies to custom rules which are supposed to change existing look or behavior. If you are writing a custom widget without overriding default Scout rules there shouldn't be any problems regarding migration.

# Chapter 9. Extensibility

## 9.1. How to extend Scout objects

The extension feature in Scout JS works by wrapping functions on the prototype of a Scout object with a wrapper function which is provided by an extension. The extension feature doesn't rely on subclassing, instead we simply register one or more extensions for a single Scout class. When a function is called on an extended object, the functions are called on the registered extensions first. Since a Scout class can have multiple extensions, we speak of an extension chain, where the last element of the chain is the original (extended) object.

The base class for all extensions is `scout.Extension`. This class is used to extend an existing Scout object. In order to use the extension feature you must subclass `scout.Extension` and implement an `init` function, where you register the functions you want to extend. Example:

```
scout.MyExtension.prototype.init = function() {  
  Ê this.extend(scout.MyStringField.prototype, '_init');  
  Ê this.extend(scout.MyStringField.prototype, '_renderProperties');  
};
```

Then you implement functions with the same name and signature on the extension class. Example:

```
scout.MyExtension.prototype._init = function(model) {  
  Ê this.next(model);  
  Ê this.extended.setProperty('bar', 'foo');  
};
```

The extension feature sets two properties on the extension instance before the extended method is called. These two properties are described below. The function scope (`this`) is set to the extension instance when the extended function is called.

**next**

is a reference to the next extended function or the original function of the extended object, in case the current extension is the last extension in the extension chain.

**extended**

is the extended or original object.

All extensions must be registered in the `installExtensions` function of your `scout.App`. Example:

```
scout.MyApp.prototype.installExtensions = function() {
  scout.Extension.install([
    'scout.FooExtension',
    'scout.BarExtension',
    // ...
  ]);
};
```

## 9.2. Export Scout model from a Scout classic (online) application

With the steps described here you can export model-data (e.g. forms, tables, etc.) from an existing classic, online Scout application into JSON format which can be used in a Scout JS application. This is a fast and convenient method to re-use an existing form in a Scout JS application, because you don't have to build the model manually. Here's how to use the export feature:

- ¥ Activate the `TextKeyTextProviderService` in your Scout classic application. You can do this either by calling the static `register` Method at runtime (using the Debugger) or by setting the config property `scout.texts.showKeys`. Once the TextProvider is active, it returns text keys instead of localized texts. The format of the returned string is `${textKey: [text-key]}`. Strings in this format are interpreted browser side by Scout JS and are resolved and localized in `texts.js`.
- ¥ Call the Scout classic web application with the URL parameter `?adapterExportEnabled=true`. This parameter is checked by Scout JS and the effect of it is that Scout JS keeps adapter-data loaded from the server, so we can use it for a later export operation. Usually Scout JS deletes adapter-data right after it has been used to create an adapter instance.
- ¥ Start your browsers developer tools (F12) from your running Scout classic app, inspect the form or another adapter you'd like to export, and search for the ID of that adapter by looking at the `data-id` attribute in the DOM. Then call the following JavaScript function from the console: `JSON.stringify(scout.exportAdapter([id]))`. The console will now output the JSON code for that adapter. Copy/paste that string from the console and tidy it up with a tool of your choice (for instance [jsoneditoronline.org](https://jsoneditoronline.org)).
- ¥ Now the formatted JSON code is ready to be used in your Scout JS project. Simply store the JSON to a .json File and load it in the `_jsonModel` function of your widget by calling `scout.models.getModel(' [ID of json model]')`. Most likely you must edit the JSON file and apply some changes manually. For instance:
  - ! Replace numeric adapter IDs by human-readable, semantic string IDs. You will need these IDs to reference a model element from the JSON model in your JavaScript code, e.g. to register a listener on a Menu.
  - ! Maybe you'll have to adjust objectType and namespace of some widgets in the JSON. This needs to be done for widgets that do not belong to the default `scout` namespace.
  - ! Remove form fields and menus which are never used in your Scout JS app
  - ! Remove unused menu types from the `menuTypes` property of a `scout.Menu`

Note: The function `exportAdapter` can be used not only for forms, but for any other widget/adapter

too.

# Chapter 10. Widget Reference

This chapter describes some of the widgets available in Scout. At the moment most of the widgets are not described yet, but they are already available at the [Scout JS Widgets Application](#). With this application you can try the widgets out and get a feel of what they are capable of and how they are used.

## 10.1. Smart Field

A **Smart Field** provides a list of proposals the user can choose from to pick one single value. In contrast to a common drop down list, the Smart Field provides search as you type which is especially useful for large lists. A very common case is to call a REST service and lookup the proposals while typing. This combination of a drop down list and a search field is the reason why it is called smart.

*Figure 5. Smart Field*

If you don't need the search as you type feature, you can switch it off by setting the property `displayHint` to `dropdown` so that it behaves like a regular drop down list. This means the user cannot filter the values anymore using key board and can choose the values only by mouse / touch.

*Figure 6. Smart Field with display style set to 'dropdown'*

Another type of Smart Field is the so called **Proposal Field**. A Proposal Field does not require the

user to choose from the available proposals but allows him to enter custom text as well.

*Figure 7. Proposal Field*

In order to provide the proposals you can either use a [Chapter 7](#) or a [Code Type](#).

In a Scout JS application you can use SmartFields like in a classic Scout application. Any REST service can be used to provide results for a SmartField lookup call. However, the SmartField expects the result to have a defined structure. If the REST service API is in your hands, you can simply return a JSON response with the right structure. This means less work to do for you in the JavaScript layer, because you don't have to transform the response to a structure the SmartField can process. If you must use a service which API you cannot change, you have no other choice than doing some kind of transformation in JavaScript.

Here's how the response for a lookup call should look like in order to be processed by the SmartField:

```

{
  Ê "queryBy": "ALL|TEXT|KEY|REC", # lookup type, as requested by the client
  Ê "text": "foo", # [optional] only set when queryBy=TEXT, contains the requested
search text
  Ê "key": "123", # [optional] only set when queryBy=KEY, contains the key of the
requested lookup row
  Ê "rec": "234", # [optional] only set when queryBy=REC, contains the key of the
requested parent lookup row
  Ê "lookupRows": [ # the result set of this lookup, contains 0-n lookup rows
  Ê   {
  Ê     # key and text are mandatory properties, all other properties are optional
  Ê     "key": "unique lookup row key",
  Ê     "text": "lookup row text",
  Ê     "iconId": "person.svg",
  Ê     "toolTipText": "foo",
  Ê     "enabled": false,
  Ê     "active": false,
  Ê     "cssClass": "special-format",
  Ê     "backgroundColor": "#cc00ee", # deprecated, use cssClass instead
  Ê     "foregroundColor": "#333333", # deprecated, use cssClass instead
  Ê     "font": "Dialog-PLAIN-12", # deprecated, use cssClass instead
  Ê     "parentKey": "234", # only used for hierarchical smart fields
  Ê     "additionalTableRowData": [ # only used for table like lookup results with
multiple columns
  Ê                                     # contains 0-n objects with arbitrary properties
  Ê       {
  Ê         "foo": "bar"
  Ê       }
  Ê     ]
  Ê   }
  Ê ],
  Ê "exception": "something went wrong" # [optional] only set when an error occurred
during the lookup request
}

```

Here's how the request for a Scout JS SmartField lookup call could look like. Your request to a REST API can look completely different. This example just gives you an idea of how to implement your own LookupCall.

```

{
  Ê "type": "lookupByAll|Text|Key|Rec", # lookup type
  Ê "text": "foo", # [optional] only set when type=lookupByText, contains the requested
search text
  Ê "key": "123", # [optional] only set when type=lookupByKey, contains the key of the
requested lookup row
  Ê "rec": "234", # [optional] only set when type=lookupByRec, contains the key of the
requested parent lookup
}

```

# Chapter 11. Browser Support

The Scout HTML UI requires a web browser with modern built-in technologies: HTML 5, CSS 3, JavaScript (ECMAScript 5). Scout does its best to support all browsers widely in use today by making use of vendor-specific prefixes, polyfills or other workarounds. However, some older or obscure browsers are not supported deliberately, simply because they are lacking basic capabilities or the required effort would be beyond reason.

Here is a non-exhaustive list of supported browsers:

## Desktop

- ¥ Mozilla Firefox  $\geq 35$
- ¥ Google Chrome  $\geq 40$
- ¥ Microsoft Internet Explorer 11 (see table below for known limitations)
- ¥ Microsoft Edge  $\geq 12$  (see table below for known limitations)
- ¥ Apple Safari  $\geq 8$

## Mobile

*(Due to the nature of mobile operating systems, it is hard to specify exact versions of supported browsers. Usually, the screen size and the device speed are the limiting factors.)*

- ¥ iOS  $\geq 8$
- ¥ Android  $\geq 5$
- ¥ Windows Mobile  $\geq 10$

Table 1. Known Limitations

Affected System	Description
Internet Explorer	If the browser is configured to enable the so-called "protected mode", the state of a popup window cannot be determined correctly. This is noticeable when a <code>AbstractBrowserField</code> has the property <i>"show in external window"</i> set to <code>true</code> . Even though the popup window is still open, the method <code>execExternalWindowStateChange()</code> is called immediately, telling you the window was closed (because IE reports so). There is no workaround for this problem, apart from disabling the "protected mode".
Internet Explorer and Edge	Performance in popup windows (e.g. opening a form with <code>DISPLAY_HINT_POPUP_WINDOW</code> ) is very poor. We have <a href="#">filed a bug</a> with the folks at Microsoft in 2015, but unfortunately the issue is still unresolved. To prevent slow forms in IE, they should be using a different display hint. Alternatively, users can use a different browser.

# Chapter 12. How-Tos

This chapter provides various small technical guides to very specific Scout JS subjects.

## 12.1. How to Create a Custom field

This cheat sheet shows how to implement your own custom control for a ScoutJS application. In this example we will write a FlipCard field, which allows you to have a 'card' with a front and back side. Clicking on the card should flip the card from one to the other side.

### 12.1.1. Files

Create an empty JavaScript file and Stylesheet

¥ `src/main/js/hellojs/custom/FlipCard.js` The java script file representing the field.

¥ `src/main/js/hellojs/custom/FlipCard.less` Styling of the field.

### 12.1.2. Add the files to the application

Include the JavaScript file to the module

hellojs-module.js

```
(function(hellojs, scout, $, undefined) {  
  ...  
  __include("hellojs/custom/FlipCard.js");  
  ...  
})(window.hellojs = window.hellojs || {}, scout, jQuery);
```

Include the Stylesheet to the module

hellojs-module.less

```
...  
@import "hellojs/custom/FlipCard.less";  
...
```

### 12.1.3. Minimal Code for a new FormField

Create a minimal FormField

The FlipCard will inherit from FormField which has already a label and field container.

FlipCard.js

```

hellojs.FlipCard = function() {
  Ê hellojs.FlipCard.parent.call(this);
};
scout.inherits(hellojs.FlipCard, scout.FormField);

hellojs.FlipCard.prototype._render = function() {
  Ê // create the field container
  Ê this.addContainer(this.$parent, 'flip-card');
  Ê // create a label
  Ê this.addLabel();

  Ê // create a field
  Ê var $field = this.$parent.appendDiv('content');
  Ê // add the field to the form field.
  Ê this.addField($field);
  Ê this.addMandatoryIndicator();
  Ê this.addStatus();
};

```

Add the FlipCard to the HelloForm

HelloForm.json

```

{
  "id": "hellojs.HelloForm",
  ...
  "rootGroupBox": {
    ...
    "fields": [
      {
        "id": "DetailBox",
        ...
        "fields": [
          {
            "id": "NameField",
            ...

          },
          { !
            "id": "FlipCard",
            "objectType": "hellojs.FlipCard",
            "label": "Flip the card",
            "gridDataHints": {
              "h": 5,
              "weightY": 0
            }
          },
          {
            "id": "GreetButton",
            ...
          }
        ]
      }
    ]
  }
}

```

! The FlipCard field

## Result Minimal Form Field

### 12.1.4. Full featured flip card field

FlipCard.js

```
hellojs.FlipCard = function() {
  Ê hellojs.FlipCard.parent.call(this);
  Ê this.$card = null;
  Ê this.$front = null;
  Ê this.$back = null;

  Ê this.flipped = false;
};
scout.inherits(hellojs.FlipCard, scout.FormField);

hellojs.FlipCard.prototype._render = function() {
  Ê // create the field container
  Ê this.addContainer(this.$parent, 'flip-card');
  Ê // create a label
  Ê this.addLabel();

  Ê // create a field
  Ê var $field = this.$parent.appendDiv('content');
  Ê // add the field to the form field.
  Ê this.addField($field);
  Ê this.addMandatoryIndicator();
  Ê this.addStatus();
}
```

```

Ê // create the field content !
Ê this.$card = $field.appendDiv('card')
Ê .on('mousedown', this._onCardMouseDown.bind(this)); "
Ê this.$front = this.$card.appendDiv('front');
Ê this.$back = this.$card.appendDiv('back');
};

hellojs.FlipCard.prototype._renderProperties = function() { #
Ê hellojs.FlipCard.parent.prototype._renderProperties.call(this);
Ê this._renderFrontImage();
Ê this._renderBackImage();
Ê this._renderFlipped();
};

hellojs.FlipCard.prototype._renderFrontImage = function() {
Ê if (this.frontImage) {
Ê   this.$front.append('<img src=\"' + this.frontImage + '\">');
Ê }
};

hellojs.FlipCard.prototype._renderBackImage = function() {
Ê if (this.backImage) {
Ê   this.$back.append('<img src=\"' + this.backImage + '\">');
Ê }
};

hellojs.FlipCard.prototype._remove = function() { $
Ê hellojs.FlipCard.parent.prototype._remove.call(this);
Ê this.$card = null;
Ê this.$front = null;
Ê this.$back = null;
};

hellojs.FlipCard.prototype._onCardMouseDown = function() { "
Ê this.setFlipped(!this.flipped);
};

hellojs.FlipCard.prototype.setFlipped = function(flipped) {
Ê this.setProperty('flipped', flipped);
};

hellojs.FlipCard.prototype._renderFlipped = function() {
Ê this.$card.toggleClass('flipped', this.flipped);
};

```

! Create the dom elements in the render function.

" Event handling to flip the card. Add/remove the CSS class **flipped** to the card element.

# Initial rendering of the properties. Applies the state to the DOM.

\$ Keep the reference tree clean. Reset DOM references when the field has been removed.

HelloForm.json

```
{
  "id": "hellojs.HelloForm",
  ...
  "rootGroupBox": {
    ...
    "fields": [
      {
        "id": "DetailBox",
        ...
        "fields": [
          {
            "id": "NameField",
            ...

          },
          { !
            "id": "FlipCard",
            "objectType": "hellojs.FlipCard",
            "label": "Flip the card",
            "frontImage": "res/img/card-back.jpg",
            "backImage": "res/img/card-front.jpg",
            "gridDataHints": {
              "h": 5,
              "weightY": 0
            }
          },
          {
            "id": "GreetButton",
            ...
          }
        ]
      }
    ]
  }
}
```

! The FlipCard field

FlipCard.less

```

.flip-card {
  & .card {
    position: absolute;
    cursor: pointer;
    height: 100%;
    width: 152px;
    transition: transform 1s; !
    transform-style: preserve-3d;

    & .flipped {
      transform: rotateY( 180deg );
    }

    & > div {
      display: block;
      height: 100%;
      width: 100%;
      position: absolute;
      backface-visibility: hidden; "

    & .back {
      transform: rotateY( 180deg ); #
    }

    & > img {
      height: 100%;
      width: 100%;
    }
  }
}

```

! Animation of the card.

" Ensure back side is not visible.

# Rotation to back side.

## Result Flip Card

# Appendix A: Licence and Copyright

This appendix first provides a summary of the Creative Commons (CC-BY) licence used for this book. The licence is followed by the complete list of the contributing individuals, and the full licence text.

## A.1. Licence Summary

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit <https://creativecommons.org/licenses/by/3.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

A summary of the license is given below, followed by the full legal text.

You are free:

¥ to Share ---to copy, distribute and transmit the work

¥ to Remix---to adapt the work

¥ to make commercial use of the work

Under the following conditions:

Attribution ---You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

With the understanding that:

Waiver ---Any of the above conditions can be waived if you get permission from the copyright holder.

Public Domain ---Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

Other Rights ---In no way are any of the following rights affected by the license:

¥ Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;

¥ The author's moral rights;

¥ Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice ---For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to <https://creativecommons.org/licenses/by/3.0/>.

## A.2. Contributing Individuals

Copyright (c) 2012-2014.

In the text below, all contributing individuals are listed in alphabetical order by name. For

contributions in the form of GitHub pull requests, the name should match the one provided in the corresponding public profile.

Bresson Jeremie, Fihlon Marcus, Nick Matthias, Schroeder Alex, Zimmermann Matthias

## A.3. Full Licence Text

The full licence text is available online at <http://creativecommons.org/licenses/by/3.0/legalcode>

!

Do you want to improve this document? Have a look at the [sources](#) on GitHub.