Advanced Debugging of Maestro Forms

Occasionally while developing Maestro forms you may see a behavior that doesn't seem right and you can't figure out why. This usually occurs in relation to rules that you've created using the script editor.

The following sections describe some advanced debugging techniques that you can use to find out what is going on in your script and in the form data.

Topics covered:

This article is focused on functional aspects of the form - debugging of CSS and styling issues is not covered.

Recommended Skill Set

Debugging Maestro forms in the browser is a technical task and requires some understanding of web application development in general, specifically the web standards of HTML, CSS and JavaScript.

Additionally, forms produced by Maestro run on AngularJS. AngularJS is a JavaScript based open-source front-end web application framework backed by Google. There is a very strong community supporting this technology and while it is not required that you are familiar with AngularJS, it would be a benefit for you to have some basic knowledge of the framework. We would recommend you review the tutorials to develop your understanding of this technology:

- AngularJS Tutorial

Debugging Tools

Before getting started, ensure you have appropriate tools for debugging JavaScript in the browser. These include:

1. **Modern Browser**
   Most modern browsers come with good developer tools these days. We prefer to use Google Chrome, but Firefox and Microsoft Edge can also be used. This article assumes you are using Google Chrome which we recommend.

2. **Developer Tools Enabled**
   All these browsers come with developer tools out of the box and allow you to install plugins/extensions to provide additional capabilities. There are a number of ways to enable these developer tools but universally, hitting the **F12** key while viewing a page will bring up the built-in developer tools.

3. **AngularJS Browser Plugin**
   Maestro forms are built on the AngularJS framework and plugins are available to provide specific support for AngularJS web applications. In Chrome, we use AngularJS Batarang, Firefox also has AngularJS plugins that may provide the same capability.

Maestro Form Architecture

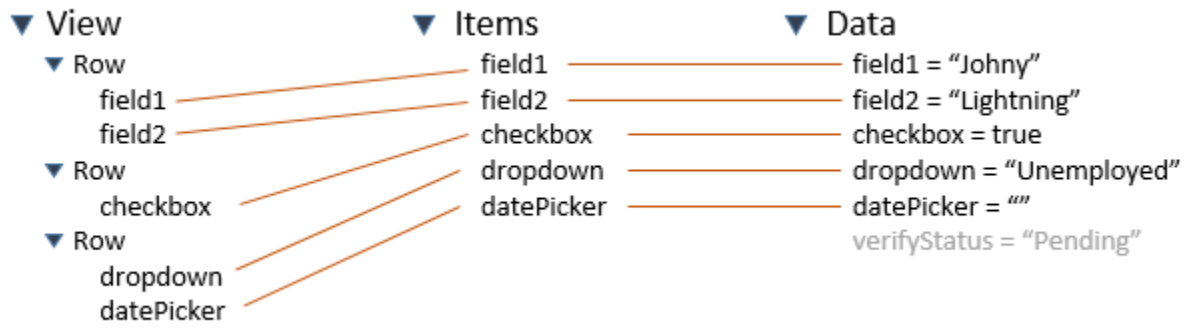
The architecture of a form consists of 3 main elements:

1. **Form View**
   A hierarchical structure that defines the visual layout of the form.

2. **Form Items**
   Each component (e.g. Field, Button) or container (e.g. Block, Section) within the form structure is considered an **item**. Form items is a list containing every item in the form.

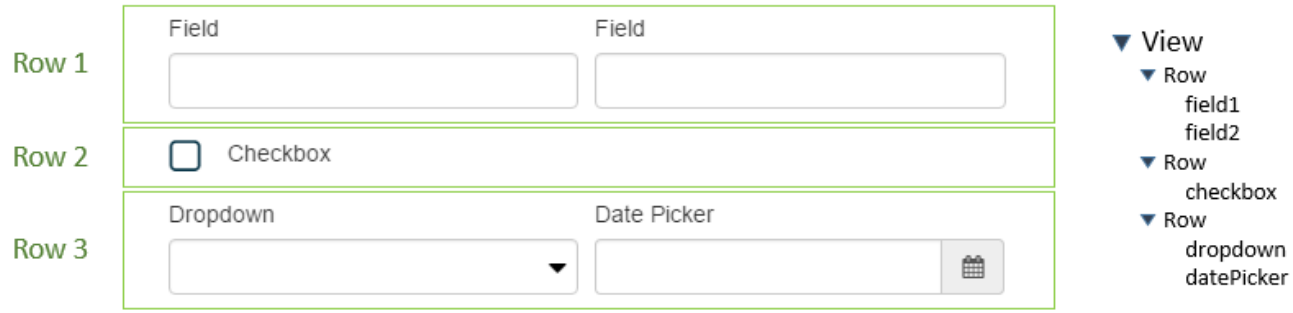
3. **Form Data**
   Think of form data simply as a map of key / value pairs. Most data elements are linked to a form item (e.g. a field) but data elements can also exist autonomously.

These 3 elements are all closely related. Form items appear in the Form View and have Data elements associated with them:



Form View

The form view is a hierarchical structure that consists primarily of horizontal rows, each containing one or more components. The hierarchical structure represents the layout of the form and all components within it.



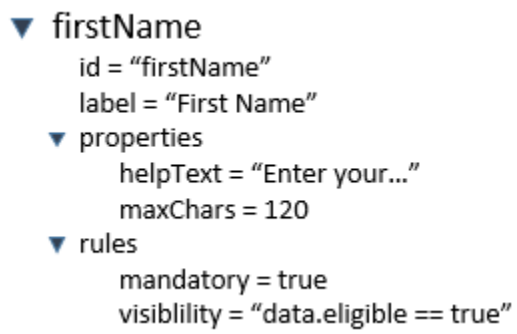
Being hierarchical, rows in the view can be containers for lists of more rows in a recursive manner.

Form Items

Each form item in Maestro is uniquely identified by a field Id. Maestro generates these field Ids based on the name of the field but also allows you to override the generated value and manually specify a field Id so long as they are unique in the context of the form.

Having guaranteed unique field Ids for all items within a form allows us to have a flat list (array) representation containing all items in the form, making access to these items much simpler than having to navigate a view hierarchy to find them.

Items can contain UI elements, but also have properties and rules associated with them. By way of demonstration, consider a mandatory first name field that has a maximum length of 120 characters and a visibility rule:



Form Data

All data in the form is stored in a single object called the Form Data object. Like form items, each form data element must have an id that is unique in the context of the form. If a data element exists with the same Id as a form item, that form item will read and write its data to that data element. Data elements can exist without being bound to a form item.

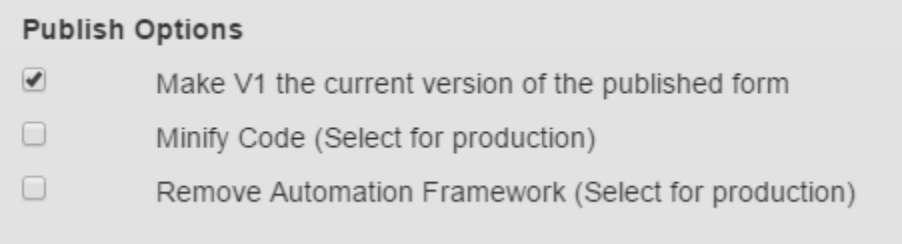
Depicted below is 4 data elements, 3 of which are bound to fields (items) and one that is un-bound (verifyStatus). The middleName data element is blank suggesting that nothing has been entered into the middleName field.

Data elements that are not bound to form items (verifyStatus in this example) will not be Saved/Submitted with the form data and so only exist for the duration of the session.

Note that the **Data Field** component type can be used to bind data elements without having to display them in the visual layout so that they get persisted with the form data.

Publishing Forms for Debug

When you publish a Maestro form there are a number of options available to you.



Some of these are very relevant to debugging, these are:

Publish Option	Description
Minify Code	**Deselect** this option. Code minification is the process of compressing code to reduce the size and therefore load time, but does not affect the operation of the code. The process removes unnecessary characters from the code,white space, new line, comments etc... The result of minification is a single line of code that is very long and difficult to read and debug so ensure that this option is deselected for debugging.
Remove Automation Framework	**Deselect** this option. The automation framework built into Maestro forms facilitates UI driven automated tests such as those that would be run from test tools like Selenium. This framework also contains some features that will assist in debugging JavaScript so it is handy to keep the automation framework for debugging.

Activating a Debug Session

in order to start a debugging session you should:

1. Load the published Maestro form in a modern browser of your choice
    a. You must access the published version of the form, **DO NOT** try and debug a form in Preview mode in the Maestro designer.

b. If you are intending to debug JavaScript (e.g. Maestro rules) then ensure you have deselected the **Minify Code** and **Remove Automation Framework** publish options as described above.

2. Once the published form is loaded in the browser you can activate the browser developer tools by hitting the **F12** key.

> **Shortcuts for Republishing Forms**
>
> While debugging forms you may need to republish your form with changes multiple times and you need to refresh the page with your published form on it each time. The following keyboard shortcuts are available for you to use to streamline this process:
>
> - CTRL-P: When used from the Maestro Design page will bring up the Publish dialog. (Command-P on Mac)
> - CTRL-ENTER: Will trigger the publish function. Remember to deselect the Minify Code publish option first.
> - CTRL-TAB: This will switch you to the last active browser tab which in most cases will be the published form.
> - CTRL-R: Will refresh the currently active browser window.
>
> Refreshing the page with the republished form on it should leave the browser developer tools active so you should not need to reactivate them each time.

Inspecting the Form Object

In the Maestro framework, the Form object contains everything else you might need to access for debugging including the View, Items and Data (see Maestro Form Architecture), so to access any of these elements you need to scope the Form object. There are a number of ways to do this from the **Console** tab in the browser developer tools:

- Using JQuery you can scope the form object in this manner:

```
$("Form").scope().Form
```

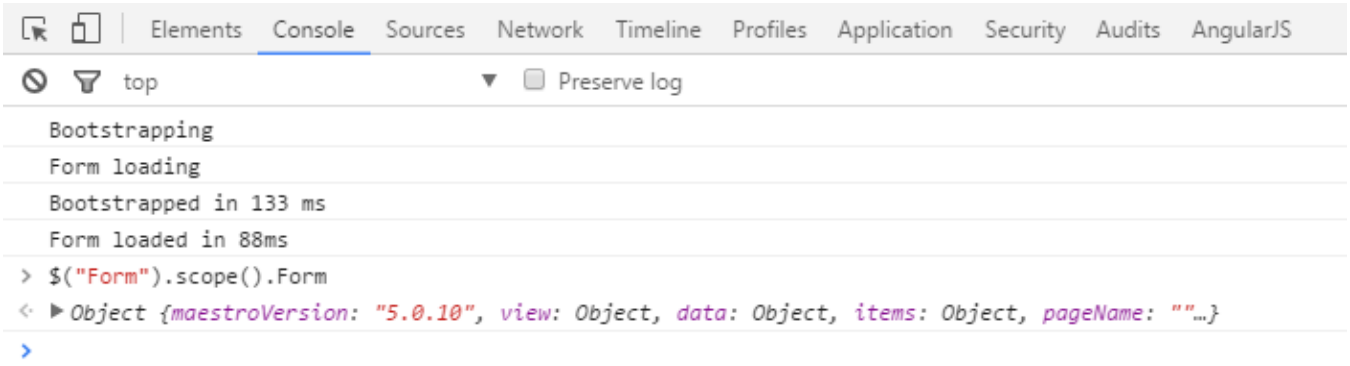
- If you have included the Automation Framework (see Publishing Options) you can short cut this access by using the **maestro** root element included with that framework:

```
maestro.Form
```

- If you have AngularJS Batarang (or similar) installed, they may also allow you to access the Form object by way of the $scope object (You need to select an element in the Elements tab first - e.g. body or form):

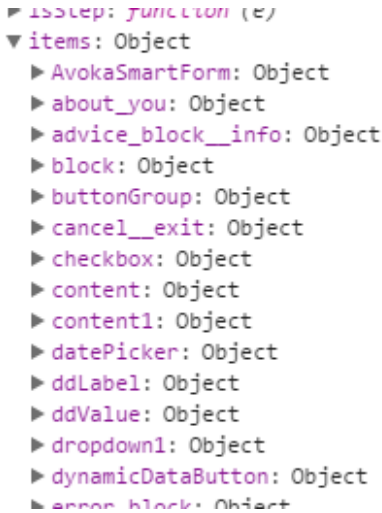
```
$scope.Form
```

Enter one of these directives into the Command prompt in the Console tab and press Enter to scope the Form object:



Opening up this object you will see a long list of sub elements but among those you will find the 3 key elements to the Maestro Form Architecture, the Form Data object,

the Form Items,



and the Form View.

Of course, if you have a JavaScript debugging session active (as described below) you will have access to the Form Data object in the function scope and can readily inspect its contents.

Logging an Item to the Console

There is a handy Util function to generate a printable representation of an item and log it to the console as follows:

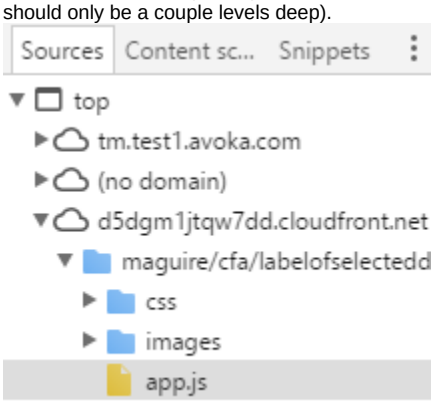
```
$scope.Util.logItem($scope.Form.items.personalDetailsBlock)

"personalDetailsBlock
 firstName
 middleNames
 lastName
 dateOfBirth
  day
  month
  year
  date
  daysData
 emailAddress
 mobileNumber
"
```

Locating a JavaScript Rule for Debug

All rules in Maestro are implemented as JavaScript so to locate the logic implemented in a Maestro rule you must find the JavaScript code that implements that rule. You can do this as follows:

1. In the developer tools window select the tab that lists the page source files (in Chrome this is titled **Sources**).
2. Open the **app.js** file by either using CTRL-O (CMD-O on Mac) and typing in the name of the file, or by searching in for it the file sources hierarchy then double-click to open it (the location of this file will vary depending on your installation and the existence of a CDN in your environment, but it should only be a couple levels deep).



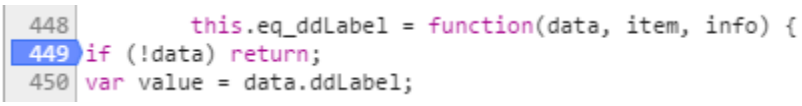
3. In the app.js file, search for and locate the JavaScript function that contains your rule in the app.js file. There are several ways to do this depending on the capabilities of the browser dev tools, but most commonly you will search by a code snippet or by the function name:

    a. To find the function using a code snippet, select a snippet of code from your rule script by clicking in the app.js file and using CTRL-F to activate the find feature. You may need to review your rule script in the Maestro designer and select a snippet that is likely to be unique. For example, below I have searched for the snippet *Form.getItemFromPath('data.buttonGroup')* and located the function **eq_ddLabel** that contains my rule script:

```
447
448         this.eq_ddLabel = function(data, item, info) {
449 if (!data) return;
450 var value = data.ddLabel;
451
452 var item = Form.getItemFromPath('data.buttonGroup');
453 var selectionObj = Util.find(item.properties.options, "value", data.buttonGroup);
454
455 return (selectionObj && selectionObj.label) || "";
456 };
457
```

    b. To search by JavaScript function name you will need to know the naming conventions for these functions (see the section on JavaScript Function Naming below). If you know the function name you can either search for the function name as a snippet (as above) or if your dev tools support it you can use the JavaScript Member search capabilities - in Chrome this can be activated using **CTRL-SHIFT-O:**

4. To debug your rule, set a break-point on one of the lines before the code you want to step through by clicking in the left column area and confirming the break-point indicator:



5. Now you are ready to debug the rule. You can return to your form window and perform the action that will trigger the rule to execute and step through your rule script using the standard JavaScript debug functions.

> A word of caution against modifying JavaScript in-line in the browser debug tools. Results of this sort of change are often unreliable. While it is less convenient to republish a form each time you want to change a rule - this is the recommended approach.

Rule Types and JavaScript Function Naming

Every rule that you put into your Maestro form will have a corresponding JavaScript function in the app.js file. The name of this function will be generated based on a standard naming convention that consists of a rule type code and the field ID with an underscore separator as follows:

```
<rule-type-code>_<field-id>
```

The following codes represent the types of rules used by the Maestro foundation widgets:

Rule Type Code	Description
sh	Short for **sh**ow, this is the code for a **visibility** rule that controls whether the field is presented on screen. The result of this rule should be a boolean value where true = visible, false = read-only.
us	Short for **us**able, this is the code for an **editability** rule that controls the read-only status of a field. The result of this rule should be a boolean value where true = editable, false = read-only.
md	Short for mandatory, this is the code for a **mandatory if** rule that controls the required state of an item. The result of this rule should be a boolean value where true = mandatory, false = optional.
ok	Interpreted as **OK**, this is the code for a **validation** rule that controls the error state of the field. The result of this rule should be a string that is either blank (for valid values) or contains the error message (for invalid values).
chok	Interpreted as Change OK, this is the code for a **validate after change** rule that augments the standard validation rule to provide support for dynamic data validations. The result of this rule can either contain the error message (like the standard validation rule) or a promise.
eq	Short for **eq**uals, this is the code for a **calculation** rule. The result of this rule should be the calculated value of the field. Warning: In Maestro, calculation rules are not triggered if the field they belong to is not visible.
click	A **click rule** is an action rule triggered by clicking on a clickable UI item.
blur	A **blur rule** is an action rule triggered when the focus leaves a field.
focus	A **focus rule** is an action rule triggered when a field acquires the focus in the browser window.
change	A **change rule** is an action rule triggered when the value of a field changes.
dc	An acronym for **d**ynamic **c**lass, this rule can be used to apply a css class to a field dynamically based on logic.
load	An **action rule** that is triggered at the time the form is loaded.

presubmit	An **action rule** that is triggered prior to form submit.
postsubmit	An **action rule** that is triggered after the form submit event.
onSuccess	This rule type is specific to the Dynamic Data Button and is an action rule that is executed upon successful completion of the dynamic data function.
onFailure	This rule type is specific to the Dynamic Data Button and is an action rule that is executed upon failure from a dynamic data call.

Note: widget developers can create additional rule types for their widgets.

Examples

Function Name	Rule
sh_previousAddress	A visibility rule on the previousAddress block
eq_totalIncome	A calculation rule on the totalIncome currency field
ok_emailAddress	A validation rule on the emailAddress field
click_verifyIdentity	A click rule on the verifyIdentity button
load_AvokaSmartForm	A load rule on the form