Fields

The concepts behind fields and how to configure them
v1.26

Sirenia

September 20, 2019

Contents

When configuring an application for automation purposes it is often necessary to interact with the user-interface of the application in some manner. A *field* as concept in Cuesta represents an element in the user-interface which can be interacted with.

This can be a button, a dropdown, a table or any other type of user-interface element. Once defined a field can be manipulated in a flow, e.g. clicking a button named Ok would look like the following in a flow:

```
Fields.Ok.click();
```

What happens in that statement is that we get the `Ok` field from the `Field` object. If the field name is not a valid Javascript variable name, then use the object indexing scheme instead, e.g.:

```
Fields['Ok'].click();
```

1 Defining a field

A field can be identified from its *path* or using a *screenshot* of the field. The path approach utilizes structural information in the user-interface while the screenshot is purely visual making it more brittle wrt changes in application appearance. The Cuesta form for defining a field is given below:

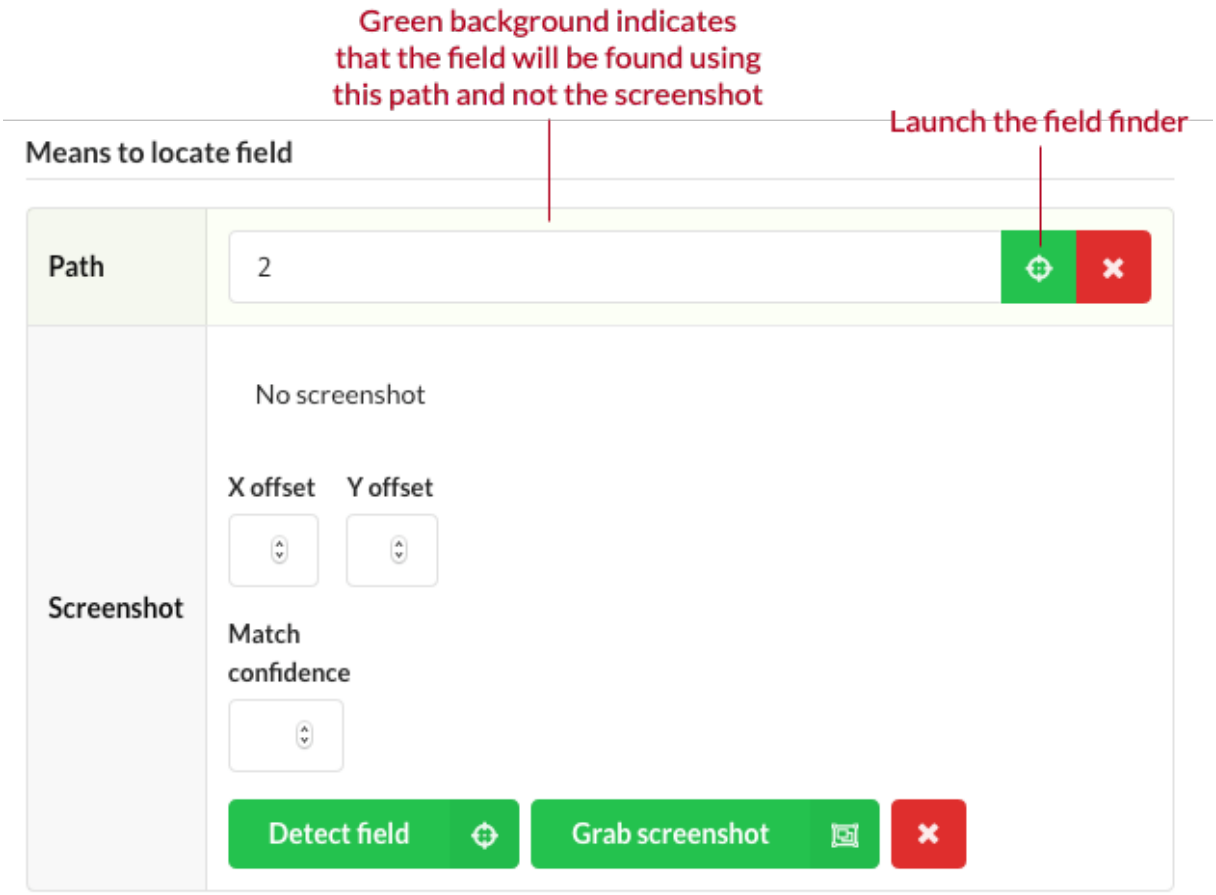
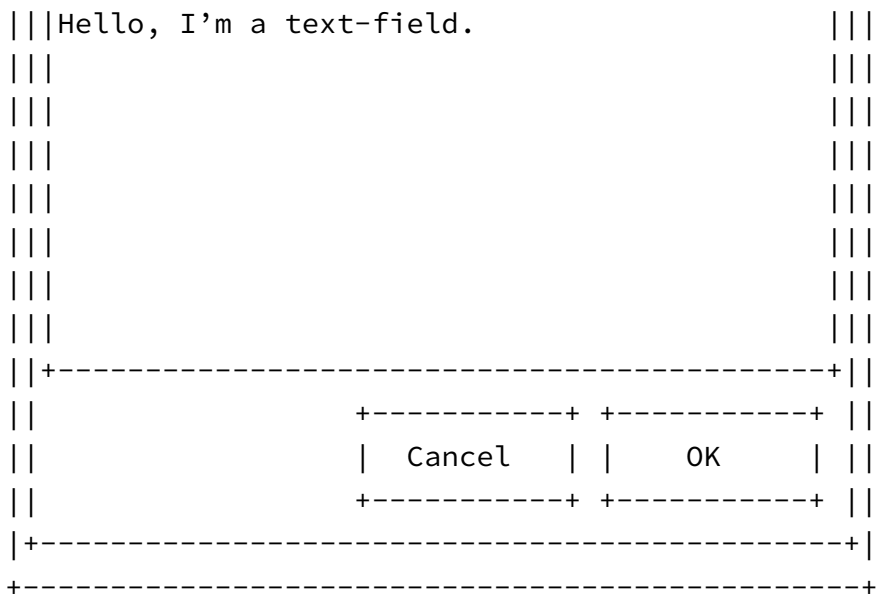


Figure 1: Defining a field in Cuesta

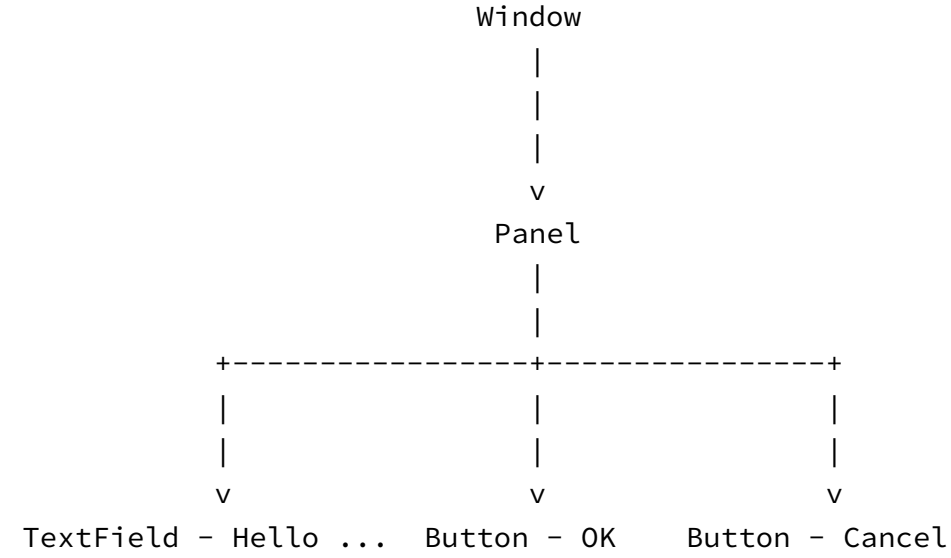
1.1 Paths to fields

A user-interface has a structure like a tree with the root of the tree being the window and the elements the branching structure. For instance the following application layout:

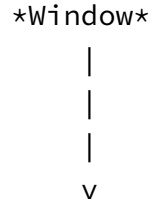
```
+-----------------------------------------------+
|+---------------------------------------------+|
||+-------------------------------------------+||
```

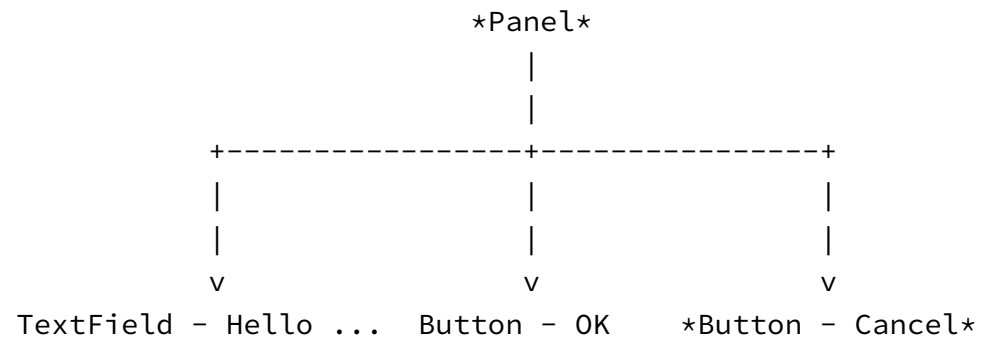


The structure of the user-interface above can be mapped to a tree like so:



To identify e.g. the *Cancel* button we use a scheme where you provide the *path* from the root of the application to the element to be identified. In the example above an identifying could be (marked with ⋆):





And in a textual form this translates to:

```
Panel/Button - Cancel
```

1.1.1 Path segments

Paths are compromised of a number of segments; one for each step down in the tree structure. Each segment matches itself against a user-interface element, checking a set of predefined properties on the element. These are commonly:

- The *type* of the element (e.g. `TextField`, `Label`)
- The *automation-id* if present
- The *textual content* of the element
- The given *name* of the element
- and more ...

Thus the path in the previous example can be shortened to:

```
Panel/Cancel
```

The default matching algorithm for each segment simply checks if the given string is a substring of the extracted element property. Using a few simple operatorswe provide more flexibility and greater accuracy:

- `*ancel` matches any string ending with ”ancel”
- `Can*` matches any string starting with ”Can”
- `*an*` matches any string containing ”an”
- `^.+an.+^` matches the string against the regular expression `.+an.+`

Furthermore the segment `**` can be used to match deep into the structure, e.g. the path:

```
**/Cancel
```

Will match the first element matching `Cancel` anywhere in the structure. This can also be used like so:

```
Panel/**/Cancel
```

Which matches any `Cancel` element with a `Panel` ancestor.

1.1.2 Special and advanced techniques

The following is a list of semi-rarely used techniques which can prove useful in tricky and non-accessible user-interfaces.

Locating via relative position

If it is not possible to locate a field directly then it may be possible to use another more easily locatable field as a sort of anchor and then using the relative (according to the anchor) position of the desired field as a guide. This is often the case with fields that have easily locatable *labels*. Using the *label* as the anchor and then specifying e.g. that the desired field is the textfield below the anchor is then possible. A positional path looks like:

```
**/Panel/AnchorElement<above>DesiredElement
```

This path will cause the `AnchorElement` to be found and then a search for the nearest `DesiredElement` **below** the anchor. It should be read as ”look for a `Panel` element somewhere in the tree, then find a child `AnchorElement` which is positioned *above* a `DesiredElement` which is our target”.

The possible positional hints are:

- `<above>` the anchor element must be found above the target
- `<below>` the anchor element must be found below the target
- `<left-of>` the anchor element must be found left of the target
- `<right-of>` the anchor element must be found right of the target
- `<nearest>` the nearest matching element to the anchor is selected

The *closest* match is the one selected if there are more matching elements in all the above cases.

Skipping matches

In case the user-interface contains ”twins” i.e. undistinguishable elements in the same level of the tree then then the *skip* operator (#) can be used to select the *i’th* matching sibling. Consider the following tree:

```
Panel
  |- Button
  |- Button
```

```
  '- Button
```

In order to target the 2nd button we might use the following path:

```
Panel/Button#1
```

The skip operator can only be used with the last element in the path and will thus only apply to the siblings of the targeted element.

Restricting path property types (native applications only)

To increase path resolution speed in native applications you can specify which property on the UI element should be used for matching each path segment by prefixing the segment with `(<type-goes-here>)`. A button with the text "Ok" can then be specified with `**/(text)Ok`. The type can appear on all segments e.g.

```
**/(type)Panel/(text)Ok
```

Which resolves any UI element with the text "Ok" directly inside a panel.

The available property-types are:

- `id` the (automation)id of the element
- `text` the text representation of the element (normally the text you can see in the UI)
- `class` the class of the UI element
- `type` the type of the UI element
- `name` the name of the UI element

The `type` can be "button", "panel", "menu", "textbox" etc.

Backtracking (in web-applications only)

Another rarely occurring case is when the target can only be uniquely matched by targeting one of its descendents. In the following example we have a clearly locatable *Ok* `Button` but we are really only interested in an anonymous `Panel` locate two levels up in the tree.

```
Panel                  <- this is our target
  '- Panel
       '- Button - Ok  <- this is the Button we can locate
```

Here we can target the desired panel by way of the following path:

```
[Panel]/Panel/Ok
```

The `[` and `]` effectively tells the path targeting mechanism to do the full path resolution but return the element contained within.

Using CSS selectors (in web-applications only)

An alternative path format for use in web-applications is using CSS selectors. In some cases using CSS selectors is easier and faster. E.g. for finding an element with a specific id:

```
#the-id
```

vs the normal path format:

```
**/the-id
```

The latter being faster.

Searching a specific embedded window

Given a multi-window application or an application with many embedded ”windows”, it is sometimes useful to limit the search for a given element to a specific window. This is done by prefixing the path with `{title-of-window}` and thus limiting the search to any windows whose title matches the given. E.g.:

```
{MyWindow}**/Panel/Ok
```

1.2 Optical fields

Optical fields are simply small screenshots of the user-interface element with an optional offset which Manatee will try to find visually and translate to a proper element. The offset is used e.g. when clicking s.t. the actual click is offset from the found location of the element.

1.2.1 Using the built-in screenshot-taker

If the field can only be identified by a screenshort press the Grab screenshot button. A red square will appear which can be move and resized to fit the field. When the red square fits the field click on the square once. The square turns green and is now fixed to the field. In order to be able to click on the field (if applicable) click on the position on the screen where the click must be done. A red dot will show where the click will be done.

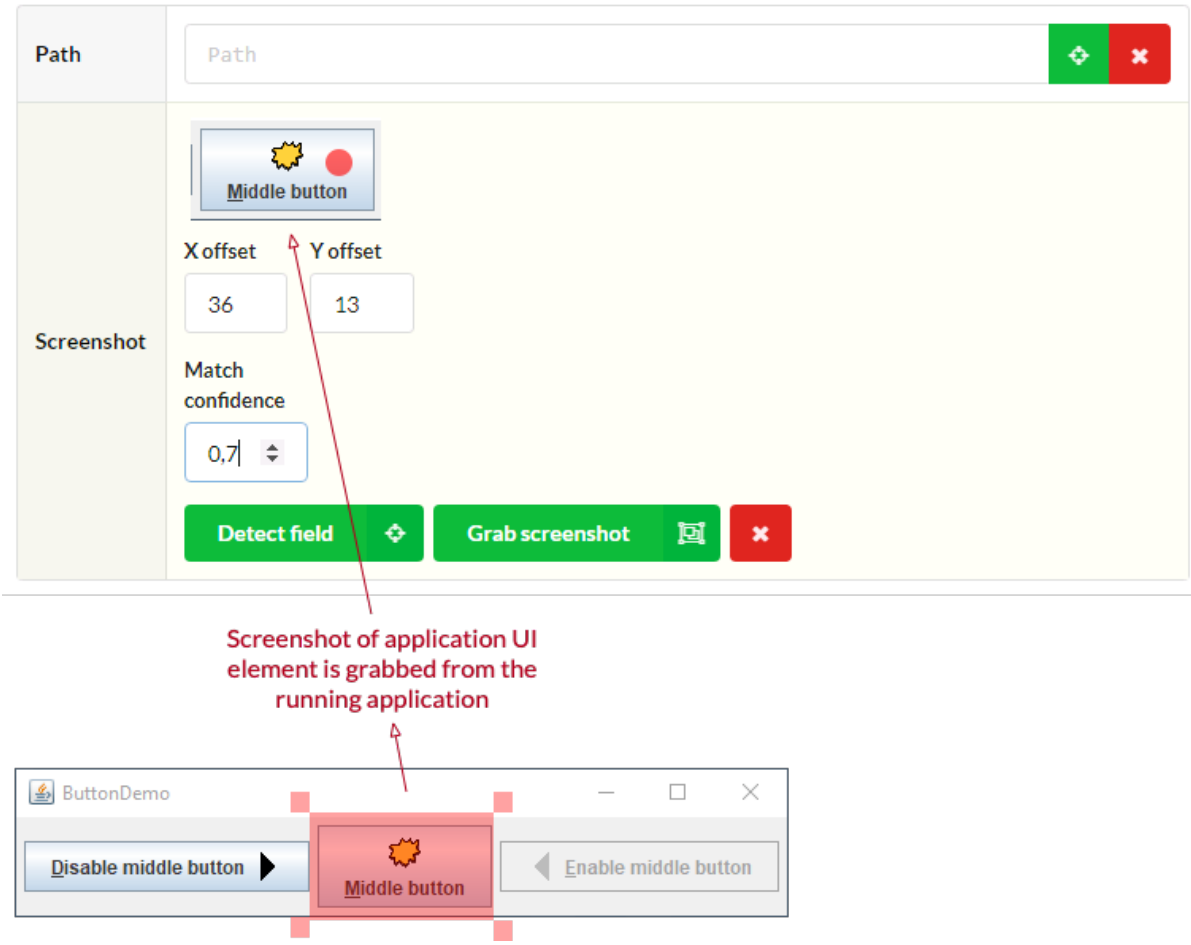


Figure 2: The built-in screenshot taker in action

The screenshort is shown in Cuesta. The click position can be adjusted in the X and Y offset fields. Match confidence can be set to reduce how acurately the screenshort should match the graphics on the screen in the application in order to have a match on the field. It can typically be set to 0.7.

1.3 Testing a path

Given a path it is useful to be able to see that the element found when the path resolution is done in Manatee is the correct element found. This can be done directly from Cuesta or by using the field in a flow.

1.3.1 Using Cuesta

Activating the locate button in Cuesta will cause a local Manatee to *highlight* the field found. This is a quick and easy way to check whether the path is correct or not.

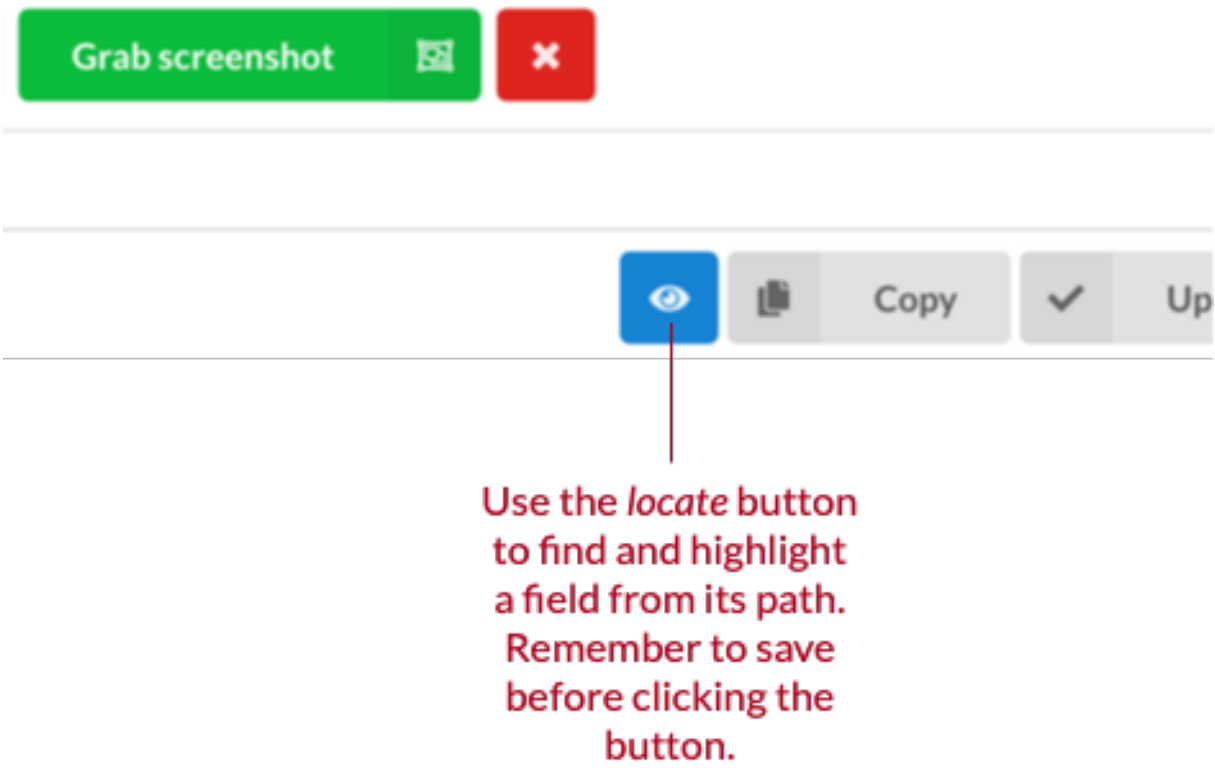


Figure 3: Click the locate button to find and highlight the field

1.3.2 Using a flow or the debugger

It is also possible to use a flow or the REPL in the `Debug.ger()` to *highlight*, *inspect* or otherwise manipulate and test a path. Fields can be created on-the-fly in a flow meaning that the following code is a quick way to try out a path:

```
var f = new Field('**/Panel/Ok');
f.highlight(); // to try and highlight the element
f.click(); // to try and click the element
f.inspect(); // to gain more info about the element
```

```
// and other field methods
```

2 Fields API

Once a field has been defined it can be used in a flow. Depending on the type of field (i.e. whether it represents a button, a panel or something else) the following methods are available.

2.1 Click

Will click on the given field.

2.1.1 Parameters

- `options` an optional options object, supports;
- `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller. The use-case for the `deadline` parameter is for example if the click launches a dialog which blocks the thread, then setting a deadline allows the flow to continue even though the click is technically not done.
- `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to `false` (underlying model traversal).

2.1.2 Example

```
Fields[”mybutton”].click();

// With an optional 500 ms deadline
Fiels[”mybutton”].click({ deadline: 500 });
```

2.1.3 Support

- ChromeDriver
- IEDriver
- JavaDriver
- NativeDriver

2.2 Click with offset

Will click on the given field offset by the amount given. It allows you to e.g. click in the middle of a table row or the corner of a button.

2.2.1 Parameters

- `options` an optional options object, supports;
- `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller. The use-case for the `deadline` parameter is for example if the click launches a dialog which blocks the thread, then setting a deadline allows the flow to continue even though the click is technically not done.

2.2.2 Example

```
// Click myButton 10px from top and 10px from left
Fields["mybutton"].clickWithOffset(10, 10);
```

2.2.3 Support

- JavaDriver
- NativeDriver

2.3 Simulated Click

Will simulate a mouse-click on the given field. The difference between simulate-click and click is only relevant for Java applications where mouse-events can be generated directly (click) or as a series of injected events - mousedown, mouseclicked, mouseup (simulateClick).

2.3.1 Example

```
Fields["mybutton"].simulateClick();
```

2.3.2 Support

- ChromeDriver
- JavaDriver

2.4 Simulated click with offset

Will click on the given field offset by the amount given. It allows you to e.g. click in the middle of a table row or the corner of a button.

2.4.1 Example

```
// Click myButton 10px from top and 10px from left
Fields["mybutton"].simulateClickWithOffset(10, 10);
```

2.4.2 Support

- JavaDriver
- NativeDriver

2.5 Right click

Will right-click on the given field.

2.5.1 Parameters

- `options` an optional options object, supports;
- `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller. The use-case for the `deadline` parameter is for example if the click launches a dialog which blocks the thread, then setting a deadline allows the flow to continue even though the click is technically not done.

2.5.2 Example

```
Fields["mybutton"].rightClick();
```

2.5.3 Support

- ChromeDriver
- JavaDriver
- NativeDriver

2.6 Right-click with offset

Will click on the given field offset by the amount given. It allows you to e.g. click in the middle of a table row or the corner of a button.

2.6.1 Parameters

- `options` an optional options object, supports;
- `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller. The use-case for the `deadline` parameter is for example if the click launches a dialog which blocks the thread, then setting a deadline allows the flow to continue even though the click is technically not done.

2.6.2 Example

```
// Click myButton 10px from top and 10px from left
Fields["mybutton"].rightClickWithOffset(10, 10);
```

2.6.3 Support

- JavaDriver
- NativeDriver

2.7 Double click

Will double-click on the given field.

2.7.1 Parameters

- `options` an optional options object, supports;
- `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller. The use-case for the `deadline` parameter is for example if the click launches a dialog which blocks the thread, then setting a deadline allows the flow to continue even though the click is technically not done.

2.7.2 Example

```
Fields["mybutton"].doubleClick();
```

2.7.3 Support

- ChromeDriver
- JavaDriver
- NativeDriver

2.8 Double-click with offset

Will click on the given field offset by the amount given. It allows you to e.g. click in the middle of a table row or the corner of a button.

2.8.1 Parameters

- `options` an optional options object, supports;
- `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller. The use-case for the `deadline` parameter is for example if the click launches a dialog which blocks the thread, then setting a deadline allows the flow to continue even though the click is technically not done.

2.8.2 Example

```
// Click myButton 10px from top and 10px from left
Fields["mybutton"].doubleClickWithOffset(10, 10);
```

2.8.3 Support

- JavaDriver
- NativeDriver

2.9 Click cell

Click in a cell in table (only applicable for tables). Clicking a cell has the following variants:

- `clickCell(...)` left-click a cell,
- `rightClickCell(...)`, and
- `doubleClickCell(...)`.

All with the following parameters:

2.9.1 Parameters

- `rowMatch` a text to match in the row - if an integer is supplied then that is used to select the row number
- `colMatch` a text to match in a column header
- `options` an options object on which the follow properties can be set;
  - `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller.
  - `reflectionDepth` indicates how deep to do the search for the `rowMatch` value (also see Reflection depth)
  - `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to `false` (underlying model traversal).

2.9.2 Example

```
// Click in the cell defined by its row containing 'A' and its column (header) c
Fields["myTable"].clickCell('A', 'B');
// The same command but use reflection depth to do a deeper search
Fields["myTable"].clickCell('A', 'B', { reflectionDepth: 2 });
// Click row 10 in column with header 'B'
Fields["myTable"].clickCell(10, 'B', { reflectionDepth: 2 });
```

2.9.3 Support

- NativeDriver
- JavaDriver

2.10 Read

Will read the value of the field. Depending on the type of the field the behavior will differ, e.g. on a label it will return the text content of the label, for a text-field it will return the contents of the text-field. For a more complex container type it will return a JSON representation of the control (which can be natively accessed in the flow as an object). See JSON serialisation for details on how different types are serialised.

2.10.1 Parameters

- `options` an optional options object with details regarding the inspection.
  - `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to `false` (underlying model traversal).

2.10.2 Example

```
var contents = Fields["mytextfield"].read();
```

2.10.3 Support

- ChromeDriver
- IEDriver (complex content *not* supported)
- JavaDriver
- NativeDriver

2.11 Exists

Returns true if the field could be found.

2.11.1 Example

```
if(Fields["mytextfield"].exists()) {
  ...
}
```

2.11.2 Support

- ChromeDriver
- IEDriver
- JavaDriver
- NativeDriver

2.12 Inspect

Inspect a given field. The returned object will contain misc information about the field - the type of information depends on the type of the field.

2.12.1 Parameters

- `options` an optional options object with details regarding the inspection.
  - `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to `false` (underlying model traversal).
  - `reflectionDepth` (see below)

2.12.2 Reflection depth

You can optionally obtain more detailed information about the data in eg treeviews. To do this, pass a positive `reflectionDepth` value as shown in the examples below.

As an example, reflectionDepth: 3 means the result includes fields such as `arrival.date.day` (3 steps) but not eg `patient.eyes.left.tla` (4 steps).

The `reflectionDepth` paramater affects the data available in the output under the objects in the control in question (eg treeview nodes). The main use of this feature is to determine which patterns to use with `Field['field'].select()` when simply selecting the rendered text doesn't work.

2.12.3 Example

```
var info = Fields["mytextfield"].inspect();
// See which information was returned
Debug.showDialog(JSON.stringify(info));
// If info has a 'text' property, then this will show the text
Debug.showDialog(info.text);

var detailedInfo = Fields["myTreeView"].inspect({ reflectionDepth: 2});
// This object includes extra data under the nodes of 'myTreeView'.
Debug.showDialog(JSON.stringify(detailedInfo));
```

2.12.4 Support

- JavaDriver
- NativeDriver

2.13 Input

Input a text value into a textfield/textbox/etc.

2.13.1 Parameters

- `text` the text to input
- `options` an optional options object.
  - `useCachedUI` an optional boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to `false` (underlying model traversal).
  - `file` an optional boolean indicating if the field is an html file input, which requires special treatment. If this is set to true, then the value must be a valid path that points to a file or an exception may be thrown. Only applicable to web apps.

2.13.2 Example

```
Fields["mytextfield"].input("some text");

Fields["myFileField"].input("C:\\some\\file.txt", { file: true });
```

2.13.3 Support

- ChromeDriver
- IEDriver
- JavaDriver
- NativeDriver

2.14 Native input

Inputs text into a field using native events, i.e. simulating keyboard input. This is useful for fields which does validation (e.g. date-fields or similar). Use only if the `input` method does not work.

2.14.1 Parameters

- `text` the text to input - you can use `<backspace>` to indicate a backspace/delete action, as well as `<enter>` and `<tab>`.

2.14.2 Example

```
Fields["mydatefield"].inputNative("11112011");
Fields["mydatefield"].inputNative("123<backspace>"); // field will contain '12'
```

2.14.3 Support

- ChromeDriver
- NativeDriver
- JavaDriver

2.15 Native input with delay

Inputs text into a field using native events with a given delay between each keystroke simulating keyboard input. This is useful for fields which does validation (e.g. date-fields or similar). Use only if the `input` method does not work.

2.15.1 Parameters

- `text` the text to input
- `delay` the number of *milliseconds* to wait between each "keystroke"

2.15.2 Example

```
Fields["mydatefield"].inputNativeWithDelay("some text", 100);
```

2.15.3 Support

- ChromeDriver
- JavaDriver

2.16 Select

Select a value. This only works for dropdowns, listboxes and tree-views.

Note that for tree-views the value given to this function may be an expression which matches the path to a leaf. E.g. for the following tree:

```
tree
├── a
│   └── b
│       └── c
├── d
└── x
    └── y
```

The node c may be selected by:

```
Fields["tree"].select("a/b/c");
```

2.16.1 Parameters

- `value` the value to select. By default `value` is treated as a regular expression, where characters like `.`, `*` and `(` have special meaning. If you want a literal match you need to surround `value` with << and >>, e.g. `select('<<'+v+'>>')` where v is the literal value to match.
- `options` an optional options object with details regarding the selection.

- `deadline` the time in ms to wait for the select to fail/succeed. If the select takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller.
- `reflectionDepth` an option indicating how far the select matching should dive into java objects (eg treeview nodes). Setting this too high may negatively affect performance. Defaults to 0. Use the `inspect` method to determine how to match against this information and what an appropriate (minimal) reflection depth is.
- `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to `false` (underlying model traversal).

2.16.2 Example

```
// Select ”option1” and use reflectionDepth to to try and find ”option1”
Fields[”mytree”].select(”option1”, { reflectionDepth: 2 });
```

2.16.3 Support

- ChromeDriver
- IEDriver
- JavaDriver
- NativeDriver

2.17 Select with index

Select a value based in an index. This only works for dropdowns, listboxes and tree-views.

2.17.1 Parameters

- `index` is the index in the combo, listbox or tree to select.
- `options` an optional options object with details regarding the selection.
  - `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to `false` (underlying model traversal).

2.17.2 Example

```
Fields["mycombo"].selectIndex(5);
```

2.17.3 Support

- JavaDriver
- NativeDriver

2.18 Select with offset

Select a value (with an offset). This only works for dropdowns, listboxes and tree-views.

2.18.1 Parameters

- `value` the value to base selection on. The value needs only to partially match the shown option to be selected, e.g. using "utte" in a list containing the item "butter" will select it.
- `offset` (int) the offset which will be used to do actual selection. E.g. if "1" then the next element (which was found using value will be selected).

2.18.2 Example

```
Fields["mytree"].selectWithOffset("option1", 1);
```

2.18.3 Support

- JavaDriver

2.19 Select with offset and skip

Select a value (with an offset and skip). This only works for dropdowns, listboxes and tree-views.

2.19.1 Parameters

- `value` the value to base selection on. The value needs only to partially match the shown option to be selected, e.g. using "utte" in a list containing the item "butter" will select it.

- `offset` (int) the offset which will be used to do actual selection. E.g. if ”1” then the next element (which was found using value will be selected).
- `skip` will select the N’th match to start from. E.g. `1` will skip the first match and select the 2nd.

2.19.2 Example

```
Fields[”mytree”].selectWithOffsetAndSkip(”option1”, 1, 1);
```

If used on e.g. a combobox with options; [”option1”, ”option2”, ”option1”, ”option3”] the code-fragment above will select ”option3”. This is done by first looking for all ”option1”s. Then skip 1 this will get you the 2nd ”option1”, then offset by 1 which will get you ”option3”.

2.19.3 Support

- JavaDriver

2.20 Edit cell

Can be used in a table to edit a given cell.

2.20.1 Parameters

- `row` the row in which to find the cell (match any cell in the row)
- `column` the column in which to find the cell (must match a single column)
- `value` the value to put into the cell (works with textfield and dropdowns)
- `options` is an optional argument, which can contain:
  - `reflectionDepth` used to finding the value if there is e.g. a combobox in the cell to edit (also see Reflection depth)
- `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to `false` (underlying model traversal).

2.20.2 Example

Given the following table:

header 1	header 2
cell 1	cell 2
cell 3	cell 4

This command:

```
Fields["mytable"].editcell("cell 3", "header 2", "boom");
```

Will result in this table:

header 1	header 2
cell 1	cell 2
cell 3	*boom*

2.21 Highlight

Highlight the given field with the default color.

2.21.1 Example

```
Fields["myfield"].highlight();
```

2.21.2 Support

- ChromeDriver
- IEDriver
- JavaDriver
- NativeDriver

2.22 Highlight with color

Highlight the given field with the given color. Available colors are `red`, `green` and `blue`.

2.22.1 Parameters

- `color` the highlighting color - `red`, `green` or `blue`.

2.22.2 Example

```
Fields["myfield"].highlightWithColor("blue");
```

2.22.3 Support

- ChromeDriver
- IEDriver
- JavaDriver
- NativeDriver

2.23 Lowlight

Cancel a highlight on a field.

2.23.1 Example

```
Fields["myfield"].lowlight();
```

2.23.2 Support

- ChromeDriver
- IEDriver
- JavaDriver