

Documentation of jQuery-Widget for Experimental Feature Representation

[Documentation of jQuery-Widget for Experimental Feature Representation](#)

[1 Requirements](#)

[1.1 Purpose](#)

[1.2 Functional Requirements](#)

[1.3 Libraries](#)

[2 Design and Implementation](#)

[2.1 Integration](#)

[2.1.1 Options](#)

[2.2 Managing Data and Calculations](#)

[2.2.1 Feature Type Categories](#)

[2.2.2 Data](#)

[2.2.3 Sorting of Participants](#)

[2.2.4 Arrangement of Features](#)

[2.2.5 Calculations](#)

[2.3 Events](#)

[2.4 Drawing](#)

[2.4.1 Important Structures](#)

[2.4.2 Participant Drawer](#)

[Drawing Participants](#)

[Calculate Colour for “range” range type](#)

[Prepare Drawing of Features and Build Function Name](#)

[Drawing the Legend](#)

[2.4.3 rangeStatusFunctionCollection](#)

[2.4.4 FeatureDrawer](#)

[Drawing](#)

[EventHandling Element](#)

[Tooltip](#)

[2.4.5 ShapeDrawer](#)

[2.5 Public methods](#)

1 Requirements

1.1 Purpose

This jQuery widget will represent protein interactions and should be easy to integrate in the IntAct website detail page (e.g. <http://www.ebi.ac.uk/intact/pages/details/details.xhtml?interactionAc=EBI-1964854>) and DASTY (e.g. http://www.ebi.ac.uk/dasty/client/index.html?q=BRCA1_HUMAN). The aim of the widget is to provide a user with a picture of an interaction to make it more understandable. It is not easy to understand a complex interaction, if the information is provided in a table.

1.2 Functional Requirements

1. Each participant of an interaction is displayed in the image.
2. Features (parts of participants) are assigned to a category by their type.
3. Each category is represented differently.
4. Each range status occurring at the start or end of a feature is represented differently.
5. All interactor types, feature types and range statuses are described in a legend.
6. Two linked features are also connected in the image.
7. Additional information on features is provided.
8. Additional actions can be performed when a feature is clicked.

1.3 Libraries

- jQuery (http://docs.jquery.com/Downloading_jQuery#Download_jQuery)
- jQuery.ui (<http://jqueryui.com/download>)
- Raphaël (<https://raw.github.com/DmitryBaranovskiy/raphael/master/raphael.js>)

2 Design and Implementation

2.1 Integration

To integrate the widget in a .html page the following steps have to be followed:

- Include the file “ui.Interaction.js” in your .html file.
- Download and include all scripts from “InteractionRepresentation/resources/js”.
- Include the file “annotOverlapping.js” in your .html file.
- Include the following javascript-libraries:
 - raphael.js
 - jquery-1.4.3.min.js
 - jquery-ui-1.8.10.custom.min.js
- Create a container (e.g. a <div> for the the widget).
- Create a proxy (e.g. like the [proxy.php](#) in “InteractionRepresentation/resources/proxy”)
- Instantiate the widget like this:

```
<script type="text/javascript">
  $(document).ready(function() {
    var interactionPlugin = $("#myDiv").Interaction(
      {width: 500,
      jsonUrl: 'InteractionRepresentation/resources/data14729917_EBI-1554232.json',
      proxyUrl: 'InteractionRepresentation/resources/proxy/proxy.php'})
    ;});
</script>
```

2.1.1 Options

There are the following options to set, the displayed values are the default values:

```
options: {
  width: 10,
  jsonUrl: '',
  developingMode: false,
  proxyUrl: '',
  useProxyForData: true,
  useProxyForOntology: true,
  useProxyForColours: true,
  legendPosition: '',
  loadingImageUrl: ''},
```

Explanation

width:	the width of the image (height will be calculated)
jsonUrl:	the url to the json-File with interaction data
developingMode:	if set to “true” additional information on errors will be written on the console and ontology data will be loaded from a local file instead of sending a
<u>proxyUrl</u> :	a url for a proxy that can be used in JavaScript Ajax requests since no cross-domain requests are possible with JavaScript
useProxyFor...:	you can decide per resource if the use of the proxy is necessary
legendPosition:	can be set to “left” or “right”, default will be left
loadingImageUrl:	if this property is set the given image will be shown while loading data

2.2 Managing Data and Calculations

2.2.1 Feature Type Categories

With the following JavaScript object categories are created and feature types are assigned to these categories:

```
_typeCategories: {
  "binding site": {
    "identifiers": ['MI:0117'],
    "loadChildren": true,
    "position": "middle",
    "colour": "#FAB875",
    "opacity": "1",
    "symbol": ""
  },
  "tag": {
    "identifiers": ['MI:0507', 'MI:0845', 'MI:0856', 'MI:0373', 'MI:0863',
'MI:0950'],
    "loadChildren": true,
    "position": "top",
    "colour": "#99ccff",
    "opacity": "1",
    "symbol": ""
  },
  "mutation": {
    "identifiers": ['MI:0118'],
    "loadChildren": true,
    "position": "bottom",
    "colour": "#EEEEEE",
    "opacity": 1,
    "symbol": "",
    "stylesheetTerm": "MUTAGEN"
  },
  [...],
  "not recognised": {
    "identifiers": [],
    "loadChildren": false,
    "position": "middle",
    "colour": "#bebebe",
    "opacity": 1,
    "symbol": ""
  }
},
```

The “identifiers” array contains all “MI”-ontology terms of feature types that should be assigned to the category. If the “loadChildren” property is “true” their children terms will be added with a request to the Ontology Lookup Service (see [2.2.2](#)). The specified position must be included in another JavaScript object containing the height of elements at this position in pixels:

```
_positionsOnProtein: {
  "top": 7,
  "middle": 10,
  "bottom": 7
},
```

The properties “colour” and “opacity” are needed to draw the features, as is the “symbol” property, which indicates if the feature is drawn in the “rectangle style” or “line style”. If no symbol name is specified the features will be drawn like this:



Otherwise there must be a function to draw the symbol - named “draw” plus the symbol’s name - in the FeatureDrawer (see [2.3.4](#)) and it will look like this:



If the property “stylesheetTerm” exists, this term is searched in the DAS style sheet and the specified colour is used for this category.

Features where the feature type’s id is found in “_ignoreTerms” will not be represented.

The “identifiers” array of the category “not recognised” will contain all feature types that could not be assigned to any other category and are not listed in the array “_ignoreTerms”.

Most interactors will be treated and represented as a protein (see [2.4.2 Participant Drawer](#)). To represent an interactor type differently the options “identifiers”, “symbol” and “loadChildren” must be specified in the object “_interactorCategories”.

```
_interactorCategories:{
  "bioactive entity": {
    "identifiers" : ["MI:1100"],
    "loadChildren": true,
    "symbol": "BioactiveEntity"
  },
  "gene":{
    "identifiers" : ["MI:0250"],
    "loadChildren": true,
    "symbol": "Gene"
  },
  "nucleic acid":{
    "identifiers" : ["MI:0318"],
    "loadChildren": true,
    "symbol": "NucleicAcid"
  }
}
```

2.2.2 Data

Ontology Lookup Service

To get the children terms of the given identifiers a request is sent to the OLS Webservice with the feature types' parent term "MI:0116" (or any other term assigned to the variable "_MIParent"):

<http://www.ebi.ac.uk/ontology-lookup/json/termchildren?ontology=MI&depth=1000&termId=>

The response is a JSON file hierarchically structured like this:

```
{ "id": "MI:0117",  
  "name": "binding site",  
  "children": [ { "id": "MI:0442", "name": "sufficient binding site" },  
                { "id": "MI:0429", "name": "necessary binding site" } ]  
}
```

By parsing the response all children terms are added to each "identifiers" array containing their parent term.

Interaction Data

To get the data for an interaction in JSON format a request is sent to <http://www.ebi.ac.uk/intact/json?ac=>. The 'ac' parameter expects the EBI-Ac of the desired interaction.

2.2.3 Sorting of Participants

An interaction's participants having features are displayed above participants without features. Participants without features are not sorted further, while participants with features are ordered by the count of their linked features with the protein with the most linked features in the middle and the others placed in descending order alternating above and below it.

2.2.4 Arrangement of Features

Each participant has a collection of features for each possible position. The participant's features are assigned to one of these collections by the position given in their associated category.

Similar Features

Before assigning a feature to a collection it is compared to all other contained features and if all properties (except the "id" and the xref with type "") are equal the new similar feature is not added to the collection. Only the features contained in one of these collection will be drawn in the picture afterwards. To avoid missing links between features, all not-added features' ids are replaced by the similar, added feature's id.

Overlapping Features

All features assigned to one collection have to be organised in different tracks so that overlapping ones are arranged one below the other. This arrangement is done with the "annotOverlapping.js" file, which is also used in Dasty and Karyodas.

2.2.5 Calculations

Height

After the arrangement of all features the height is calculated by summing up the height for each track of features and adding the number given in the variable "`_featureGap`" between all every two tracks. As value for the height of each track the number given in the structure "`_positionsOfProteins`" at the position of the track is used.

Pixels per Amino Acid

To represent image's elements proportionally to the width of it, the number of pixels representing one single amino acid is calculated. The width of one amino acid is calculated by dividing a number of pixels derived from the given width for the whole image by the length of the longest participant. Each position - for example the length of another protein or the start of a feature - will be multiplied with this width to maintain all proportions.

2.3 Events

If a user clicks on a feature or an interactor an event is triggered. This event can easily be caught with jQuery:

```
$(document).bind(eventName, function(event, params){//do something});
```

There are different events for features and interactors with a different JavaScript-Object for the parameter “params”.

Event Name	Parameter		
feature_selected			
	Parameter	Description	
	event	the original click-event	
	interactorId	“identity”-xref of the associated interactor	
	interactorName	name of the associated interactor	
	featureId	id of the clicked feature	
	coordinates	position of the feature on the interactor	
		x	begin of the feature
		x2	end of the feature
positionArray		single positions of non-continuous features	
interactor_selected			
	Parameter	Description	
	event	the original click-event	
	interactorId	“identity”-xref of the interactor	
	interactorName	name of the interactor	

2.4 Drawing

2.4.1 Important Structures

This section lists structures that are used and expected in many different parts of the widget.

Feature Coordinates

After a feature was added to the picture a JavaScript Object with the following properties will be returned for further use:

```
"x": 0,  
"x2": 0,  
"element": Raphaël's object,  
"eventHandlingElement": Raphaël's object  
("positionArray": Raphaël's object)
```

“x” and “x2” provide the start and end of a feature (these values are the original and not the proportional values (see [2.2.5: Pixels per Amino Acid](#))). “element” is the actually visible element representing the associated feature. “eventHandlingElement” is an element that should handle all events for this feature (see [2.3.4: Event Handling Element](#)). “positionArray” will only be defined if the feature is composed of non-continuous positions.

Called Functions

The following structure is used to “remember” all called functions of the rangeStatusFunctionCollection (see [2.3.3](#)):

```
_calledFunctions:{ "draw03350335": [{ "range": { "begin": {"position": 1 },  
                                           "startStatus": ... ,  
                                           "end": {"position": 37 },  
                                           "startStatus": ... }}  
                                ],  
                  ...  
}
```

To remember a function, it's name and the used “rangeList” have to be added to this structure. A new “rangeList” is only added if it differs significantly from the already existing “rangeList”s in this list. These differences are: “beginInterval” instead of “begin” or “endInterval” instead of “end” and “symbol” equals an empty string instead of an in the categories defined symbol.

2.4.2 Participant Drawer

This module is for drawing the interactors and the legend. It prepares the drawing of features, which are actually drawn by the module “Feature Drawer” (see [2.3.4](#))

Drawing Participants

Information on an interaction’s interactor can be found directly at the “participant” property of the interaction data or at the referenced “interactor”. If an “interactor” is referenced its id is provided by the “interactorRef” property. The third possibility is that an “interactionRef” is provided, but this case is ignored at the moment.

If the interactor type is not listed in the object “_typeCategories”, a participant is represented by an colourless rectangle with black borders. A dotted line marks every 100th amino acid.



Length

The length of the participant is calculated by counting the amino acids in the “sequence” property of an interactor. The length’s numeric value is displayed at the end of the participant. If no sequence is provided the participant will be drawn as long as the longest protein in the widget and this fact will be represented by displaying a “?” as length value.

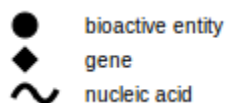
Name

A name for each participant will be also be displayed at the end of each participant’s image. The name will be the

“shortLabel” of the interactor, if this is not provided, the “fullName” is used. If none of these is provided the UniProt-Id provided as “xref” is used as name.

Fusion protein: If an interactor is a fusion protein all identifiers provided in a “multi parent”-“xref” will be combined to one name, for example the name of EBI-1264956 will be “fusion of P10276, P04637”.

If the interactor type is listed in the object “_typeCategories”, the participant is drawn with the function built with “draw” plus the option “symbol”. The following interactor types are drawn differently:



Calculate Colour for “range” range type

Since the “range” range type is represented as a gradient between the two specified positions, a slightly lighter colour than the colour used for the feature is needed. This colour is calculated by converting the HEX colour code to HSL colour code, where the ‘L’ part stands for “lightness”. This value has to be increased to lighten the colour and the resulting HSL colour code is converted to a HEX code again.

Prepare Drawing of Features and Build Function Name

In this module the following things are done to prepare the drawing of features:

- determining their type’s category for drawing details
- extracting information for the display in tooltips (see [2.3.4: Tooltips](#))
- dividing them in features with one range list and features with more than one range lists

More than one range lists

Each range list will be drawn separately in the same manner as features with only one range list (see below) and all drawn elements will be connected by a dotted line afterwards. The smallest position at the begin and the largest position at the end will be used as feature coordinates (see [2.3.1: Feature Coordinates](#)). A specified keyword will be added to the called functions (see [2.3.1: Called Functions](#)) to represent the connected features in the legend.

A single range list - Building the function’s name


To draw a feature’s single range list the function’s name in the rangeStatusFunctionCollection (see [2.3.3](#)) that has to be called to draw this specific range status combination has to be build. To do this, the identifiers of the start and the end status is extracted and the “MI:”-prefix is deleted. These are used to build the two following function names:

- type category + start id + end id, e.g. “tag03350335”
- “draw” + start id + end id, e.g. “draw03350335”

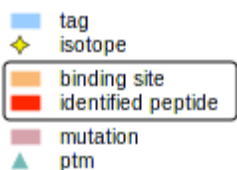
The first function name has to be provided if there is a specific way to draw this range status combination for one category. If one term should be treated exactly like another one (e.g. “ragged n-terminus” and “n-terminal position”) the term has to be added to “_rangeStatusEquivalents” in the main document. Otherwise the default function “draw” + identifiers will be called.

If none of the above functions can be found, one or both identifiers are not recognized. There is a special representation for unrecognized range statuses:

Rectangle  style:

Line  style:

Drawing the Legend



First a picture with examples of all feature type categories is drawn according to their position on the protein and their style attributes, described by their name in “typeCategories” (see [2.2.1](#)). An “not recognised” representation of a feature, will only be included if there is one in the shown interaction.

The second part of the legend describes the range types. Since there are so much different range type combinations, only the ones existing in the shown interaction are displayed in the legend. Therefore all called range status functions called are logged in “calledFunctions” (see [2.3.1 Called Functions](#)). Each “range” in each function name’s list is modified to fit in the legend and the function is called again to draw the element. To make no difference between two categories represented the same way, but with different colours, all legend items are grey and for elements with the “line style” a rectangle is used instead of the original symbol.

Linking Features

The information on linkage of features is contained in the property “inferredInteractionList”:

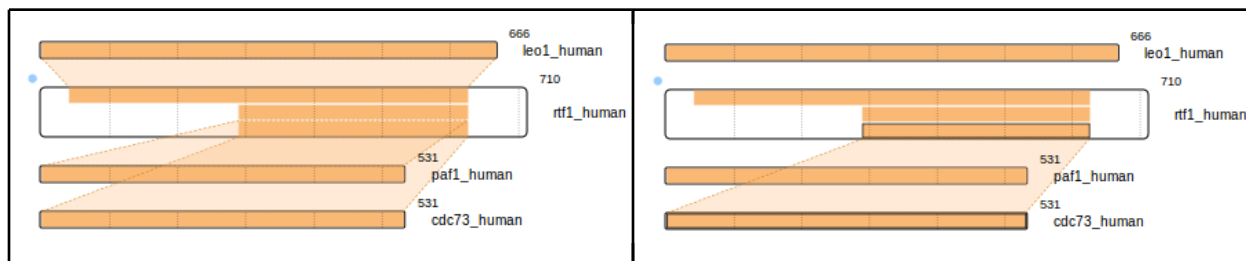
```
"inferredInteractionList":
  {"inferredInteraction":{
    "participant":[{"participantFeatureRef":"72"},
                  {"participantFeatureRef":"74"}],
    "experimentRefList":{"experimentRef":"10"}}}
```

For all “participantFeatureRef”s the coordinates (see [2.3.1: Feature Coordinates](#)) are stored separately and all valid elements are connected according to the information in “inferredInteractionList”. These connections can only exist between two participants, so if there are one or more than two participants in one “inferredInteraction” no connection, but a warning is displayed. A connection is displayed as semi-transparent quadrangle between the bottom of the upper feature and the top of the lower one.



For all elements belonging to one connection an “onClick”- event handler is registered, to highlight and only display a clicked connection.

All connections displayed	Only one connection displayed
---------------------------	-------------------------------



To enable the user to click on all connections the quadrangles are sorted by the area of their enclosing rectangles. Furthermore a click on a connection on top of a feature will be registered as a click on the feature.

2.4.3 rangeStatusFunctionCollection

This module contains a collection of functions for all covered range status combinations. All of these functions have the same parameters to allow a dynamic building of the function name.

An example for a function like this is “draw03350335”, which is the default function to draw a feature with a certain begin and a certain end. All functions with the prefix “draw” are default the functions for these range status combinations.

If a category has a special representation for a specific range status combination, a function with the category name as prefix and the range status id’s as postfix, e.g. “tag03390339”.

If a range status combination is possible, but should not be displayed, a function doing nothing and returning an empty JavaScript Object has to be included in this collection, like:

```
this.draw03390339 = function(range, y, height, featureStart, interactorLength,
                             colour, rangeColour, opacity, sequence, tooltipText,
                             symbol) {
    return {};
};
```

In this collection the functions will not be implemented but call other functions in the Feature Drawer module (see [2.3.4](#)), since it’s purpose is the simplification of adding new range types.

2.4.4 FeatureDrawer

Drawing

This module’s functions extract the values for start and end of features from the expected positions in the data. If there are no value at the expected position a warning is displayed. For example, a “certain” range status can’t have a “beginInterval” whereas a “range” range status can. This module will finally draw the features according to their style attributes.

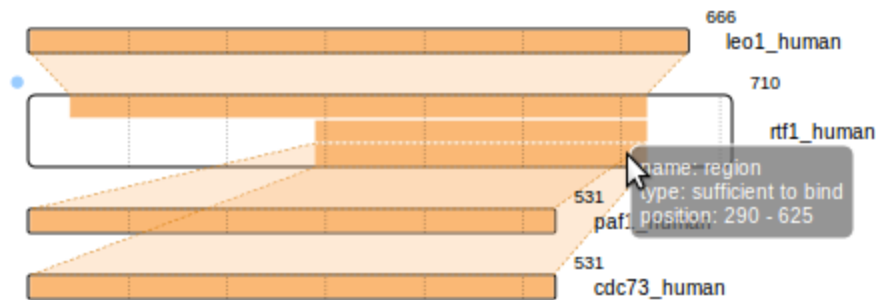
EventHandling Element

The event handling element returned by this module’s function and used in the whole widget,

is the actually displayed feature element plus a transparent element as high and wide as the displayed feature itself. If the height or width is smaller than 1 pixel, the transparent element is heightened or broadened so it is easier to click or hover over a feature for the user. By adding this transparent element it is easier to coordinate the overlapping of displayed elements and the associated event handling elements, since a feature displayed on the top must not be the uppermost event handling element.

Tooltip

As tooltip a semi-transparent rectangle with all the extracted information of a feature is created. It will only be displayed if the user hovers over a feature.



2.4.5 ShapeDrawer

This module is for drawing simple shapes like rectangles, circles, triangles, stars, etc.

2.5 Public methods

There are four public methods for the interaction with the widget:

`highlightRegion(interactorId, x, x2, colour, text):`

Puts a semitransparent rectangle over the interactor identified by the `interactorId`.

If there is more than one interactor identified by this id there will be a rectangle for each one.

The `interactorId` is the “identity” id of the interactor which is also given in the event “`interactor_selected`”.

`x` and `x2` are not pixels but the positions on the interactor.

The `colour` is used at an opacity of 6.25% for the rectangle and at 100% for the text.

The text will be displayed at the top of the rectangle, this parameter is optional.

`unhighlightRegion():`

Hides all highlighting rectangles that are shown.

`highlightFeature(featureId):`

Makes the border of the feature with the given `featureId` black.

The `featureId` is either the source reference `xref` or “`intern`” + the “`id`” of the feature.

It is the same identifier which is given in the event “`feature_selected`”.

`unhighlightFeature():`

Colours the border of the currently highlighted feature in its original colour.