Widget API

Overview of Widget API

The functions described here are typically used when working with code widgets, which further extend the functionality and flexibility of Yellowfin. These can be added to the dashboard by importing them through the plugin manager. They will then appear in the list of code widgets for selection during the dashboard building process. Take a look at the guide to creating a code widget for further information and resources.

Passing data with WidgetAPI.serverRequest and doRequest

At certain times during code widget development, you may need to pass data to the Yellowfin server. This can be achieved by using the *WidgetAPI.serverRequest* function. This makes a call to the Java class associated with the code widget.

myWidget.js

```
let widgetAPI = options.api.widget;
widgetAPI.serverRequest('hello').then(result => {
    alert(result.hello);
});
```

MyWidgetImplementation.java

```
  public void doRequest(CodeWidgetActionRequest request, CodeWidgetActionResponse response) {
    if("hello".equals(request.getAction())) {
        response.addData("hello", "Hello JavaScript!");
    }
}
```

CodeWidgetActionRequest

When a request is made from the front-end component of a code widget, a CodeWidgetActionRequest object is created. This object contains the data which was added to that request.

Functions

String getAction()

This method will match the string that was passed to the WidgetAPI.serverRequest function. It should be used to determine which actions your code widget should be undertaking.

For example, when a request from the front end is made like this:

```
widgetAPI.serverRequest(“hello”);
```

getAction() will return the string “hello”.

String getParameter(String parameterName)

This method returns a string version of a parameter included in the JavaScript *actionData* object. If the passed parameterName does not match any parameters in the request, null will be returned.

```
widgetAPI.serverRequest(“hello”, { name: “Code Widget” });
```

When *request.getParameter(“name”)* is called, it will return “Code Widget”.

<T> getParameterAsObject(String parameterName, Class<T> clazz)

This function returns the value of the passed *parameterName* as the passed Class type. For example:

```
Integer counter = getParameterAsObject(“counter”, Integer.class);
```

Calling the above code returns an integer. If the value cannot be parsed, null will be returned.

Or, for example,

```
WidgetAPI.serverRequest(‘example’, {
    counter: 1,
    myClass: {
        ‘one’: ‘one’,
        ‘two’: ‘two’
    }
});
```

When the request above is made from the front end, these can be retrieved directly as objects by doing the following in Java:

```
class MyClass {
        private String one = null;
        private String two = null;
        public MyClass(String one, String two) {
              this.one = one;
              this.two = two;
        }
        public String getOne() {
            return one;
        }
        public String getTwo() {
            return two;
       }
}
Integer counter = request.getParameterAsObject(‘counter’, Integer.class);
MyClass myClass = request.getParameterAsObject(‘myClass’, MyClass.class);
System.out.println(myClass.getTwo()); //Prints “two”
```

In this example, the variable *counter* will be parsed as an integer and *MyClass* will have the values mapped from the JSON to its corresponding private member variables.

Using this method has a limitation when generics are used, as type safety cannot be guaranteed. If you are attempting to parse an object using generics, we recommend you use *getParameterAsObject(String, Type)* instead.

<T> getParameterAsObject(String parameter, Type type)

This function attempts to parse the passed parameter key using the passed Type parameter.

This function works similarly to *<T> getParameterAsObject(String parameterName, Class<T> clazz)*, but it lets you define a Type object, which is more suitable when using generics because you can use classes that support generics.

```
List<String> myList = getParameterAsObject(“myList”, new TypeToken<List<String>>() {}.getType());
```

⚠ Using this function introduces a dependency on gson, which is shipped with Yellowfin, but is not documented on this wiki.

If you're not familiar with gson and wish to avoid using it, an alternative option is to get the String version and then parse it.

CodeWidgetResponse

This object represents the response that will be sent back to the JavaScript component of the code widget.

Any data that you wish to return to the JavaScript can be added to this, and the JavaScript will receive this response as a simple JSON object.

Functions

addData(String parameter, Object data)

This function adds the passed parameter and value to an Object to be returned to the JavaScript. For example:

```
response.addData('greeting', 'Hello World');
```

If the Java text above is used, the parameter and value can be accessed by using the JavaScript:

```
widget.serverRequest('greeting').then(result => {
    alert(result.greeting);
})
```

The objects that are passed in can be much more complex than simple strings. Any Java class can be added to this. If you add an implementation of a Java class to any publicly-accessible getters, like *getProperty* or *isProperty*, they will be added to the JSON data. When this happens, the first letter after "get" will be changed to lowercase in the generated JSON. For example:

getProperty becomes *property*

isAProperty becomes *aProperty*

The following class:

```
public Class MyExampleClass {
    public String getSayHello() {
        Return "Hello"
    }
    public String helloWorld() {
        return "Hello World";
    }
    public boolean isCodeWidget() {
        return true;
    }
}
```

...when added to the response:

```
response.addData("exampleClass", new MyExampleClass());
```

...would create create the following object:

```
{
    sayHello: "Hello",
    codeWidget: true
}
```

WidgetAPI

The widget API provides the functionality to allow the JavaScript component of a code widget to send messages to the back-end component.

WidgetAPI.serverRequest(action, actionData)

Returns

Promise - A promise that is resolved when the code widget request completes.

The promise will be passed an Object containing a JSON representation of the *CodeWidgetResponse* that is created by the backend component.

Parameters

action - String - The action you wish the server to perform

actionData - Object - An object containing any data you wish to pass along with the request.

Example

myWidget.js

```
define(function() {

    function MyWidget(options) {
        let widgetAPI = options.apis.widgetAPI;
        //Trigger a "hello" request to the backend of the widget
        widgetAPI.serverRequest('hello').then(result => {
            alert(result.hello);
        });
    }
    return MyWidget;
}
```

MyWidgetImplementation.java

```
    public void doRequest(CodeWidgetActionRequest request, CodeWidgetActionResponse response) {

    if("hello".equals(request.getAction())) {
        response.addData("hello", "Hello JavaScript!");
    }
}
```

In this example, when *MyWidget.js* is initialized, it will immediately trigger a “hello” request to the code widget’s Java component. It responds by adding the key-value pair “hello”: “Hello JavaScript” to the response. Which is then alerted by the JavaScript.

Back to top