

# Wakanda Studio Extensions

---

## Creating an extension in two minutes

Want to create your first extension in just two minutes? Go to the [My first Extension](#) section.

## What are extensions?

Wakanda Studio Extensions are software programs that can add new functionalities to the Wakanda Studio. For example, an extension can write automatically a set of predefined comment lines at the beginning of scripts.

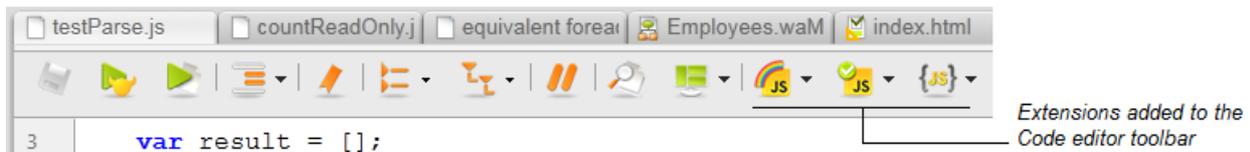
Extensions are written using standard Web technologies such as JavaScript, HTML, CSS, and JPEG. Everyone can write Wakanda Studio extensions, for their own needs or for sharing with the Wakanda community. The Wakanda Development team provides built-in pre-installed extensions such as "Beautifier" that you can use as standard Studio features.

Extensions can add contextual functionalities to various toolbars and menus of the Wakanda Studio, including the main toolbar and the solution explorer contextual menu.

## Where to add extensions?

Commands for executing extensions can be toolbar buttons or contextual menu commands. These interface elements can be added to the following parts of the Wakanda Studio:

- Solution manager toolbar
- Solution explorer contextual menu (Tree view)
- Solution explorer contextual menu (List view)
- Solution explorer contextual menu (Thumbnail view)
- Code editor toolbar
- Code editor contextual menu
  
- Solution explorer global menu (  )



You can combine these locations and create any feature you need:

- a single extension can provide several buttons and/or menu commands in one or several areas
- a single feature can be associated to a button and a menu command

## How to use this manual?

- You want to create an extension in two minutes using a template:  
-> go to [My first Extension](#)
- You want to install an extension in your Wakanda Studio:  
-> go to [Installing Extensions](#)
- You want to see quickly how to write an extension:  
-> go to [Getting started](#)
- You want to access the detailed reference documentation for creating Wakanda studio extensions:  
-> go to [Creating Extensions](#) and [API: Basic](#).

## My first Extension

---

Here are the instructions to make your first Wakanda extension in less than 2 minutes by following these 7 steps:

1. [Download the Extension Template](#) from our server and unzip it in the **Extensions** folder:
  - On Windows: `{Disk}:\Users\{User name}\AppData\Roaming\Wakanda Studio\Extensions\`
  - On Mac OS: `/Users/{User name}/Library/Application Support/Wakanda Studio/Extensions/`You may have to create the **Extensions** folder manually.  
For more information, refer to the [Installing Extensions](#) section.
2. Open `manifest.json` in a text editor and define your extension name by replacing `YOUR_EXTENSION_NAME`.
3. Replace `YOUR_EXTENSION_DESCRIPTION` with a brief description of your extension.
4. Define your extension action name by replacing `YOUR_ACTION` in `manifest.json`.
5. Replace `YOUR_ACTION_TITLE` in `manifest.json` with an easy-to-understand title.
6. Open `index.js` in a code editor and replace `YOUR_ACTION` to rename the action.
7. Write the function body in `index.js` to define your action.

Voilà! Restart your Wakanda Studio and you will see your first extension appear in the main toolbar. You can place your extension icon/menu in other places -- in `manifest.json`, just replace "studioToolbar" with another valid value (please refer to the [senders](#) paragraph).

A good example illustrates the whole picture better than detailed documentation. You can check the Wakanda Studio Extension Demo to learn how to make certain commands more complex.

However, knowledge of Wakanda Studio Extension System is required if you want to accomplish sophisticated extensions. Check the [Wakanda Studio Extension online documentation](#) for more detailed information.

You can use the [Wakanda Studio Extension development forum](#) for any technical questions/answers and for the announcement your new extension.

## Installing Extensions

A Wakanda Studio Extension is a set of files grouped in a single folder. To install the extension in your Wakanda Studio, you just need to copy the extension folder (whose name is free) in the **Extensions** folder at the appropriate location. The **Extensions** folder can exist in two different places:

- In the **Wakanda Studio application folder**:
  - On Windows: next to the *Wakanda Studio.exe* file
  - On Mac OS: at the first level of the *Contents* folder inside the application package.

In this case, extensions are available only in this Wakanda Studio application.

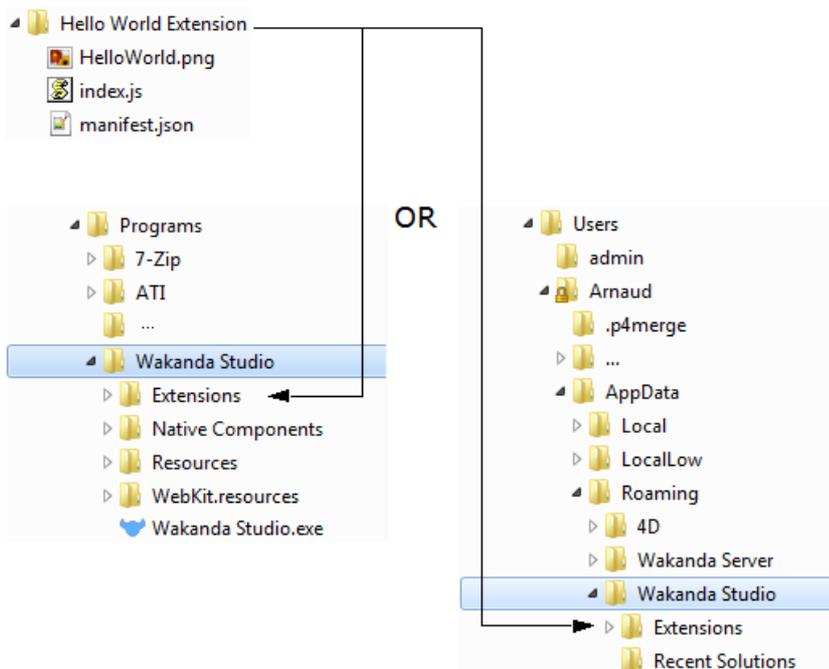
Installing files or folders at this location requires administrator access rights, and can make subsequent updates an issue. It is usually not recommended to install custom extensions in the application.

- In the **user data of Wakanda Studio**:
  - On Windows: `{Disk}:\Users\{User name}\AppData\Roaming\Wakanda Studio\`
  - On Mac OS: `/Users/{User name}/Library/Application Support/Wakanda Studio/`

In this case, extensions are available for any Wakanda Studio application running on the machine in the user session, including subsequent updates. This location does not need specific access rights.

*Note: Under Mac OS 10.7 ("Lion"), you need to expand the Go menu in the Finder while holding down the Option key to reveal the Library command that you must use to open the corresponding folder.*

The following diagram illustrates the installation options (Windows):



Priority is given to the user data location: if the same extension exists at both locations (same folder name), Wakanda Studio will only load the files from the user data.

## Getting started

---

As a first step to discover how to create an extension to the Wakanda Studio, we will write a very classic and basic example: adding a button to the code editor toolbar that displays "Hello, World!".

1. Using any text editor (for example the Wakanda Studio code editor), create a new file named `manifest.json` and write the following code:

```
{
  "extension":
  {
    "name": "Hello World",
    "version": "1.0.0",
    "description": "Hello World Demo for Wakanda Extensions",
    "icon": "HelloWorld.png",
    "senders": [
      {
        "location": "codeEditorToolbar",
        "icon": "HelloWorld.png",
        "actionName": "say_hello"
      }
    ],
    "actions": [
      {
        "name": "say_hello",
        "title": "hello"
      }
    ],
    "lifetime": "action_lifetime"
  }
}
```

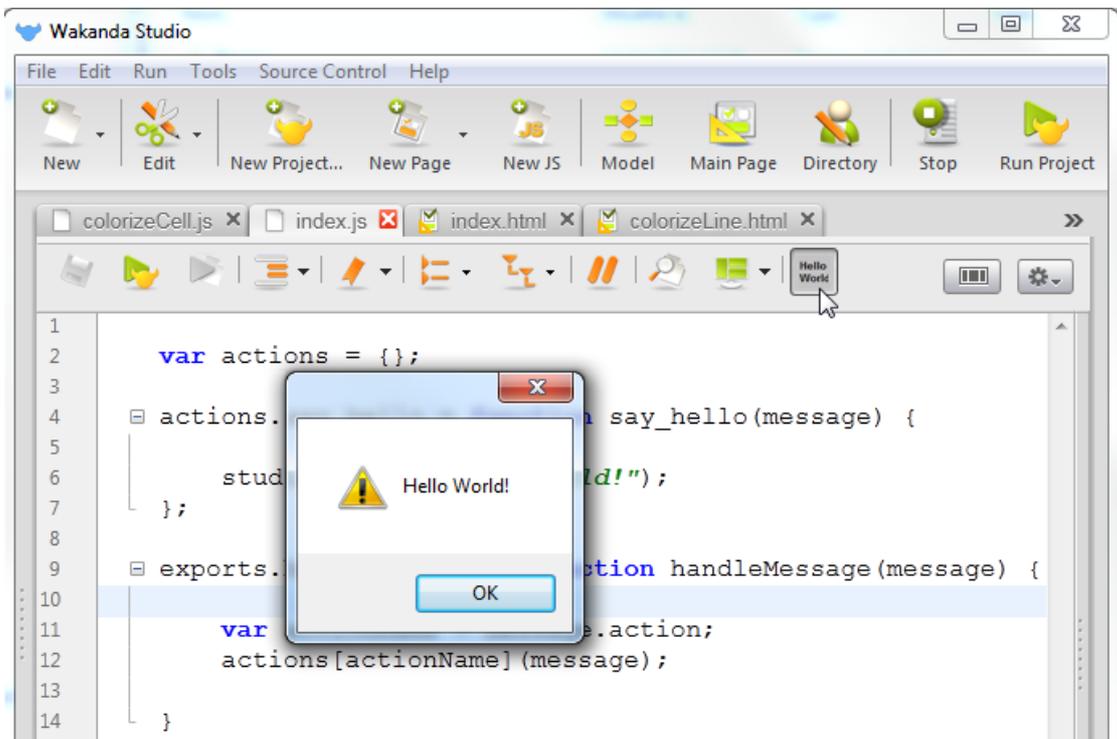
This code describes our extension. For more information on how to write the `manifest.json` file, please refer to the [Configuring the manifest.json file](#) section.

2. Create another file named `index.js` and write the following code:

```
exports.handleMessage = function handleMessage(message) {
  if(message.action == "say_hello")
    studio.alert("Hello World!");
};
```

This file will contain the action(s) to execute and the unique entry point of the extension. For this tutorial, we write the basic contents of the file, but it is generally much faster to use a "template" `index.js` file and add your own actions. For more information on how to write the `index.js` file, please refer to the [Configuring the index.js file](#) section.

3. Create a new folder, name it for example "Hello Word", and save your `manifest.json` and `index.js` files in that folder. Add also a picture button file named "HelloWorld.png" (you can download a little icon [here](#)).
4. Copy the "Hello World" folder in the Wakanda Studio Extensions folder, as described in the [Installing Extensions](#) section (choose the user data folder for more convenience).
5. Relaunch Wakanda Studio if it was already opened and load any file in the Code editor. You should see the new button: click on the button, that's it!



## Creating Extensions

---

A Wakanda Studio extension is defined through two mandatory files:

- **manifest.json**: declares the actions and their location in the Wakanda studio interface. Objects to write in this file are detailed in the [Configuring the manifest.json file](#) section.
- **index.js**: contains the code to execute in response to actions. The [API: Basic](#) is provided for extensions to communicate with Wakanda Studio internal components (for example, the code editor). This file is described in the [Configuring the index.js file](#) section.

An extension can use a unlimited number of additional files (HTML, pictures, scripts...). All the extension files must be gathered in a single folder.

### Configuring the manifest.json file

---

The **manifest.json** file is one of the mandatory pieces of a Wakanda Studio extension: it describes the extension and declares the actions and their locations in the two toolbars and four contextual menus available (see [Where to add extensions?](#)).

In this file, you can define the extension name and properties, the name of each action and the locations where Wakanda Studio should display these action commands. A single extension can add several menu items and buttons in different locations.

The **manifest.json** file is a JSON format file; it only handles strings.

#### extension

"extension" is the main object of the manifest.json file. It contains 7 objects, described below:

- name
- version
- description
- icon
- actions
- senders
- lifetime
- compatibleBuildVersion (optional)

#### name

"name" contains the extension name. Example:

```
"name": "Hello World"
```

#### version

"version" contains the extension version. Example:

```
"version": "1.2.1"
```

#### description

"description" provides a short description of the extension. Example:

```
"description": "My Great Wakanda Studio Extension"
```

#### icon

"icon" contains the path of the default icon file (relative to the extension's folder).

This icon will be used if a single button is defined by the extension. In case of multiple buttons, individual icons can be defined separately through the "senders" object.

Example:

```
"icon": "myIcon.png"
```

#### actions

"actions" contains the action name(s), title(s) and optional elements, as described below:

Object	Mandatory	Type	Description
<i>name</i>	yes	string	Designates the action. Must be unique in "extension"
<i>title</i>	yes	string	Default title for the action (i.e. the item label if the extension is a menu item)

<i>targets</i>	no	array of target objects	<p>Defines for which type of file the action could be proposed (for example, .js or .html files). If omitted, the action can be available for each type of file. Can contain the following objects:</p> <ul style="list-style-type: none"> <li><i>uti</i> <a href="#">Uniform Type Identifier</a></li> <li><i>mimeType</i> to be implemented</li> <li><i>fileExtension</i> to be implemented</li> </ul>
<i>trigger</i>	no	array of event objects	<p>An event object contains the "event" property. When an action subscribes to a studio event, then this action is triggered by this event and sent to the <a href="#">handleMessage Function</a> of the <code>index.js</code> file. Possible values are:</p> <ul style="list-style-type: none"> <li><i>fromSender</i> actions are triggered by the GUI, i.e. buttons and menu items</li> <li><i>onSave</i> the current edited file is saved</li> <li><i>onFileDirty</i> the current file has been altered</li> <li><i>onFilesAddedInSolution</i> a file was added to the solution explorer area. <i>message.source.name</i> will be "fromSolutionExplorer" and <i>message.source.data</i> will contain each added <i>File</i>.</li> <li><i>onFilesRemovedFromSolution</i> a file was removed from the solution explorer area. <i>message.source.name</i> will be "fromSolutionExplorer" and <i>message.source.data</i> will contain each removed <i>File</i>.</li> <li><i>onFolderCollapsed</i> <i>message.source.name</i> will be "fromSolutionExplorer" and <i>message.source.data</i> will contain each operation <i>Folder</i>.</li> <li><i>onFolderExpanded</i> <i>message.source.name</i> will be "fromSolutionExplorer" and <i>message.source.data</i> will contain each operation <i>Folder</i>.</li> <li><i>onSolutionCreated</i> a new solution is created. <i>message.source.name</i> will be "fromSolutionExplorer" and <i>message.source.data</i> will contain the created <i>Folder</i>.</li> <li><i>onProjectCreated</i> a new project is created. <i>message.source.name</i> will be "fromSolutionExplorer" and <i>message.source.data</i> will contain the created project <i>File</i>.</li> <li><i>onSolutionOpened</i> an existing solution is opened. <i>message.source.name</i> will be "fromSolutionExplorer".</li> <li><i>onFilesMovedInSolution(*)</i> file(s) are moved within the solution explorer. <i>message.source.name</i> will be "fromSolutionExplorer". <i>message.source.data</i> will contain the destination <i>Folder</i> as first element and the source path string of each moved item (file or folder) as the following elements.</li> </ul>
<i>shortcut</i>	no	array of shortcut objects	<p>Each action may have its shortcut defined in this field. If the shortcuts defined are in conflict with the Studio's shortcuts, then priority will be given to the Studio. Possible values are:</p> <ul style="list-style-type: none"> <li><i>shortcutKey</i> Possible values: "yes" and "no". Ctrl key for Win, Command key for Mac</li> <li><i>alternateKey</i> Possible values: "yes" and "no".</li> <li><i>shiftKey</i> Possible values: "yes" and "no".</li> <li><i>key</i> Possible values: "A"-"Z", "home", etc.</li> </ul>

(\*) Moving files by drag and drop in a solution will trigger three notifications. Here is the order in which notifications are sent:  
1. *onFilesMovedInSolution*  
2. *onFilesAddedInSolution*

### 3. *onFilesRemovedFromSolution*

However, if an extension action only subscribes to *onFilesMovedInSolution*, then only this action will be triggered by *onFilesMovedInSolution*.

The result (source files moved) is not verified - move operations may fail - so it is the extension author's responsibility to check the presence/absence of files in both source and destination folders.

Example :

```

"actions": [
  {
    "name": "js_if",
    "title": "if-else",
    "targets": [
      { "uti": "com.netscape.javascript-source" }
    ],
    "shortcut": {
      "shortcutKey" : "yes",
      "alternateKey" : "yes",
      "shiftKey" : "no",
      "key" : "i"
    }
  },
],

```

### senders

"senders" defines the location(s) of action commands, i.e. the interface objects that will generate the actions.

This property is an array of sender objects. Each sender object contains the following properties:

Object	Mandatory	Type	Description
<i>location</i>	yes	string	Indicates where to make the extension available. You can pass one or more values. Available strings are: <i>studioToolbar</i> <i>solutionExplorerTreeViewContextMenu</i> <i>solutionExplorerListViewContextMenu</i>  <i>solutionExplorerThumbnailViewContextMenu</i> <i>solutionExplorerGlobalMenu</i> <i>codeEditorToolbar</i> <i>codeEditorContextMenu</i> (to be implemented)
<i>actionName</i>	yes if no "menu" object array is passed	string	Name of an action defined in the <i>name</i> property of the "actions" object. Use this first level <i>actionName</i> when it is available through a single button. Use "menu" property instead if you want to define a menu. "menu" and "actionName" cannot both be present.
<i>title</i>	no	string	Toolbar icon or menu item title. If omitted, the <i>title</i> property of the "actions" object will be used as default value.
<i>menu</i>	yes when the "location" value is <i>solutionExplorerTreeViewContextMenu</i> , <i>solutionExplorerListViewContextMenu</i> , <i>solutionExplorerThumbnailViewContextMenu</i> , <i>solutionExplorerGlobalMenu</i> or <i>codeEditorContextMenu</i> or if no "actionName" object is passed	array of menu item objects	Array of menu item objects, which can be repeated recursively down until the 2nd level. Use this property if you want to define a menu. Use the first level "actionName" instead if you want to define a single button. "menu" and "actionName" cannot be both present. Each "menu item" object contains the following properties:
		<i>actionName</i>	Mandatory if no "menu" sub-object array is passed. "menu" and "actionName" cannot both be present.
			Name of an action defined in the <i>name</i> property of the "actions" object

			<i>menu</i>	Mandatory if no "actionName" sub-object array is passed. "menu" and "actionName" cannot both be present.	Title of a menu
			<i>title</i>	Mandatory if a "menu" sub-object array is passed	Menu item title. If omitted, the <i>title</i> property of the "action" object is used
		A menu array can contain an object separator, which has the following syntax: "separator":{}			
<i>icon</i>	yes for buttons (i.e. first-level <i>actionName</i> and <i>location</i> in <i>studioToolbar</i> or <i>codeEditorToolbar</i> )	string		Path of the picture file used as icon for the button (relative to the extension's folder).	
<i>tips</i>	no	string		Additional information to display when the cursor moves over the button. Used only for buttons (i.e. first-level <i>actionName</i> and <i>location</i> in <i>studioToolbar</i> or <i>codeEditorToolbar</i> ).	
<i>alternateTitle</i>	no	string		Title to display if the action's alternative state is turned on. Used only for buttons (i.e. first-level <i>actionName</i> and <i>location</i> in <i>studioToolbar</i> or <i>codeEditorToolbar</i> ).	
<i>alternateIcon</i>	no	string		Name of the icon file to display if the action's alternative state is turned on. Used only for buttons (i.e. first-level <i>actionName</i> and <i>location</i> in <i>studioToolbar</i> or <i>codeEditorToolbar</i> ).	
<i>alternateTips</i>	no	string		Tip to display if the action's alternative state is turned on. Used only for buttons (i.e. first-level <i>actionName</i> and <i>location</i> in <i>studioToolbar</i> or <i>codeEditorToolbar</i> ).	

Example of senders in both the code editor toolbar and contextual menu. They are associated with the same actions:

```
"senders": [
  {
    "location": "codeEditorToolbar",
    "tips": "Check Javascript Error"
    "menu":
      [
        {
          "actionName": "checkError"
        },
        {
          "actionName": "cleanErrors"
        }
      ]
  },
  {
    "location": "codeEditorContextMenu"
    "menu":
      [
        {
          "actionName": "checkError"
        },
        {
          "actionName": "cleanErrors"
        }
      ]
  }
]
```

```

    }
  ],
}
],

```

## lifetime

"lifetime" allows you to define the lifetime of the JavaScript context. Two values are available:

application_lifetime	Keep JavaScript context alive among the actions
action_lifetime	A new JavaScript context is created for each action and released after execution

Using "application\_lifetime" allows the writing and reading of global variables in the unique context that is shared by all the action calls in an extension. Each extension will have its own context.

Example:

```
"lifetime": "application_lifetime"
```

## compatibleBuildVersion

"compatibleBuildVersion" indicates the lowest Wakanda Studio build version compatible with the extension. Note that it's a build version, not a major version. The build version can be found in the "About Wakanda Studio" dialog box.

If the Wakanda Studio build version is smaller than the indicated version, the extension will not be loaded. The extension will always be loaded if this property is omitted. It must be a digital value (not a string type).

Example:

```
"compatibleBuildVersion": 105605
```

## Configuring the index.js file

---

The `index.js` file is the entry point of an extension for Wakanda Studio. All features (actions) provided by the extension are defined in this JavaScript file. You can use standard JavaScript code as well as a specific [Extensions API](#). All the Wakanda Studio components are available through this API in `index.js`.

## handleMessage Function

The main entry function in `index.js` is named `handleMessage`. All the actions you declared in `manifest.json` will be passed to this callback function and should be processed here.

The `handleMessage` should be set as the `handleMessage` property of an `exports` object.

The `handleMessage` function receives a `message` object as parameter. The `message` object has three properties, "action", "event" and "source":

- `message.action` contains the name of the action declared in `manifest.json` (for example, "js-if").
- `message.event` indicates the source of triggered `message` object. It can contain:
  - "fromSender" if the message is triggered by the Wakanda Studio interface (ie. user clicks on a button or menu item).
  - "onSave", "onFileDirty", "onFilesAddedInSolution", "onFilesRemovedFromSolution", "onFolderCollapsed", "onFolderExpanded", "onSolutionCreated", "onProjectCreated" or "onSolutionOpened" if the action is defined through a trigger and the user triggered the action.
  - "fromExtension" if the message is triggered by another extension (see `sendCommand()`).
- `message.source` contains an object with two properties, "name" and "data".
  - "name" value is the event source name (string). Possible values are:
    - `fromSender`: the message is triggered by the Wakanda GUI (ie. user clicks on a button or menu item).
    - `fromExtension`: the message is triggered by another extension (see `sendCommand()`).
    - `fromCodeEditor`: the message is triggered by the Code editor.
    - `fromWebDesigner`: the message is triggered by the Web Designer.
    - `fromSolutionExplorer`: the message is triggered by Solution Explorer.
    - `fromSolutionList`: the message is triggered by Solution List.
    - `fromSolutionThumbnails`: the message is triggered by Solution Thumbnails.
  - "data" is an array which can contain one or more element(s) of string, `File` or `Folder` type. It depends on the event.
 

For example, when the event is `onFilesAddedInSolution`, "data" is an array of `File` objects representing all files added to the Solution Explorer.

Within this entry function, you will usually call any appropriate function depending on the `message.action` value.

## Example

Here is a typical `handleMessage` function:

```
exports.handleMessage = function handleMessage(message) { // main entry point
```

```
var actionName;  
actionName = message.action; // get the action name  
actions[actionName](message); // execute the actionName function  
    // stored in the actions object  
    // and pass the 'message' parameter as is  
};
```

## Using the Extension API

In the `index.js` file, you can use a dedicated set of API. This API gives access to the Wakanda Studio components and allows you to benefit from all the features and capacities of the Studio.

The following API themes are available:

- [API: Basic](#)
- [API: Code Editor](#)
- [API: GUI](#)
- [API: Storage](#)
- [API: Studio](#)
- [API: Web Zone Dialog](#)

## API: Basic

---

The "basic" theme methods allow you to display standard JavaScript dialogs.

### Using 'studio' Object

All Wakanda Extension APIs are available through the "studio" object. Thus, you must prefix each API call with 'studio.'  
For example, to call the `alert()` method, you should write:

```
studio.alert("Hello World!");
```

### alert()

---

void **alert**( String *message* )

Parameter	Type	Description
message	String	Alert message

### Description

The `alert()` method displays a warning text in a standard alert dialog box.

### Example

The following code, executed from the `index.js` file of an extension:

```
studio.alert("Hello World!");
```

Displays:



### confirm()

---

Boolean **confirm**( String *message* )

Parameter	Type	Description
message	String	Confirmation message
Returns	Boolean	true if the answer is Yes, false otherwise

### Description

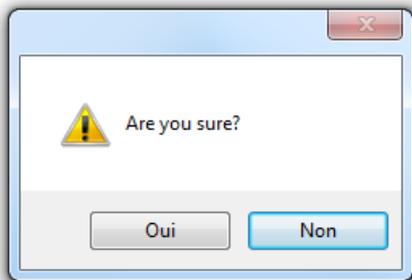
The `confirm()` method displays a confirmation dialog box and returns `true` if the user clicked on the `yes` button, and `false` if the user clicked `no`. Yes and No labels are based on the current system language.

### Example

The following code, executed from the `index.js` file of an extension:

```
var isok = studio.confirm("Are you sure?");
```

displays:



## prompt()

---

String **prompt**( String *message* [, String *defaultAnswer* ] )

Parameter	Type	Description
<i>message</i>	String	Prompt message
<i>defaultAnswer</i>	String	Pre-entered string for the reply
Returns	String	String entered by the user

### Description

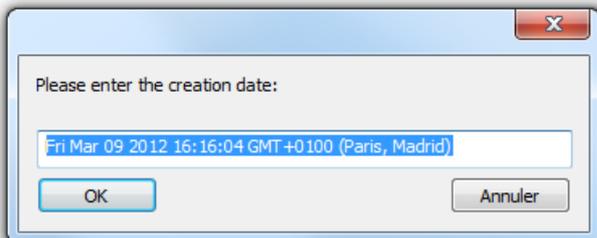
The **prompt()** method prompts the user to enter a value in response to a *message* and returns the entered value. You can pass in *defaultAnswer* a string showing an example of the value to enter or proposing a standard answer.

### Example

The following code, executed from the *index.js* file of an extension:

```
var vaDate = new Date();  
var userDate = studio.prompt("Please enter the creation date:",vaDate);
```

displays:



## API: Code Editor

---

Methods in this theme allows reading and modifying the text displayed in the Wakanda Studio Code Editor. Methods support JavaScript, HTML, XML, and any source code displayed in the editor.

### Using 'studio' Object

All Wakanda Extension APIs are available through the "studio" object. Thus, you must prefix each API call with 'studio.'  
For example, to call the `alert()` method, you should write:

```
studio.alert("Hello World!");
```

### currentEditor.clearAnnotations()

---

```
void currentEditor.clearAnnotations()
```

#### Description

The `currentEditor.clearAnnotations()` method removes all warning symbols from the annotation bar of the open document. This method will clear symbols added by any Wakanda extension using `.` However, it will not remove system warnings indicating, for example, syntax errors.

### currentEditor.getContent()

---

```
String currentEditor.getContent()
```

Returns                      String                      Contents of the edited document

#### Description

The `currentEditor.getContent()` method returns the whole content of the document currently displayed in the Code editor.

#### Example

You want to store temporarily a specific version of your code and be able to view it at any moment. You add two buttons to the code editor associated with the "store\_copy" and "show\_copy" actions. In the `index.js` file, you can write:

```
actions.store_copy= function store_copy() {  
  
    var content = studio.currentEditor.getContent(); // gets the current content  
    studio.extension.currentDialog.setItem("codeCopy" , content); // put it in the storage  
};  
  
actions.show_copy= function show_copy() {  
  
    var copied = studio.extension.currentDialog.getItem("codeCopy"); //read the storage  
    studio.alert(copied); // show the contents of the codeCopy attribute  
};
```

### currentEditor.getEditingFile()

---

```
File | Null currentEditor.getEditingFile()
```

Returns                      File, Null                      Edited document

#### Description

The `currentEditor.getEditingFile()` method returns a *File* object referencing the document currently opened in the Code editor.

If there is no current document in the Code editor (for example if the window in the foreground is not a Code editor window), the method returns null.

#### Example

```
var docPath = studio.currentEditor.getEditingFile().path;  
// docPath returns, for example  
// 'C:/Wakanda Solutions/My Solution/MyProject/MyScript.js'
```

## currentEditor.getSelectedText()

String | Null **currentEditor.getSelectedText()**

Returns Null, String Currently selected text

### Description

The `currentEditor.getSelectedText()` method returns the text selected in the document currently displayed in the Code editor. If nothing is selected in the document, `currentEditor.getSelectedText()` returns Null.

## currentEditor.getSelectionInfo()

Object **currentEditor.getSelectionInfo()**

Returns Object Definition of the selection in the document

### Description

The `currentEditor.getSelectionInfo()` method returns information about the selection in the document currently displayed in the Code editor.

Information depends on the number of line(s) selected as well as the cursor position.

You must also consider the following specificities:

- the Code editor line numbering starts at 1, although the `currentEditor.getSelectionInfo()` method line numbering starts at 0;
- collapsed or expanded code structures need to be taken into account. This is the reason why the returned object contain different properties for *selected lines* (includes all lines, whatever their expand/collapse status) and *"visible" selected lines* (counts a single line for a collapsed block).

The method returns an object containing the following properties:

Property	Type	Description
<i>firstLineIndex</i>	Number	Index of starting selection line
<i>firstVisibleLine</i>	Number	Index of "visible" starting selection line
<i>firstLineOffset</i>	Number	Starting selection position in the first selection line
<i>lastLineIndex</i>	Number	Index of ending selection line
<i>lastVisibleLine</i>	Number	Index of "visible" ending selection line
<i>lastLineOffset</i>	Number	Position of the last selected character in the last selection line
<i>isLeftToRightSection</i>	Boolean	true if the selection direction is from left to right, false otherwise
<i>offsetFromStartOfText</i>	Boolean	Position of the first selected character from the beginning of text (0)
<i>selectionLength</i>	Number	Length of selection

### Example

Considering the following selection in the code editor:

```
1 0 include("scripts/jshint.js"); 0
2 1 1
3 2 var 2
4 3 actions; 3
5 4 4
6 5 actions = {}; 5
7 6
8 7 actions.cleanErrors = function cleanErrors(message) { 7 8-9-10
12 8
13 9 actions.checkErrorWithWarningDlg = function checkErrorWithWarningDlg() { 12
14 10 var 13
15 11 fileContent; 14
16 12 var 15
17 13 result; 16
18 14 var 17
19 15 option; 18
20 16
21 studio.currentEditor.clearAnnotations();
22 option = getOptFromPref();
23 fileContent = studio.currentEditor.getContent();
24 result = JSLINT(fileContent, option);
25
```

Line Index  
Visible line index

```

var selObj = studio.currentEditor.getSelectionInfo();
var s1 = selObj.firstLineIndex;           // s1 contains 5
var s2 = selObj.firstVisibleLine;        // s2 contains 5
var s3 = selObj.firstLineOffset;        // s3 contains 10
var s4 = selObj.lastLineIndex;           // s4 contains 18
var s5 = selObj.lastVisibleLine;        // s5 contains 15
var s6 = selObj.lastLineOffset;         // s6 contains 13
var s7 = selObj.offsetFromStartOfText;   // s7 contains 61
var s8 = selObj.selectionLength;         // s8 contains 251
var isLR = selObj.isLeftToRightSection   // isLR contains true

```

## currentEditor.insertText()

---

```
void currentEditor.insertText( String textToInsert )
```

Parameter	Type	Description
textToInsert	String	Text to insert in the open document

### Description

The `currentEditor.insertText()` method inserts *textToInsert* into the document currently displayed in the Code editor, at the current cursor position.

If text was selected in the document, it is replaced by *textToInsert*.

### Example

You want to be able to insert the current date in your code. You add a button to the code editor associated with the "add\_date" action. In the `index.js` file, you can write:

```

actions.add_date= function add_date() {
    var vadata = new Date();
    studio.currentEditor.insertText(vadata);
};

```

## currentEditor.selectByLineIndex()

---

```
void currentEditor.selectByLineIndex( Number start, Number end, Number firstLineIndex, Number lastLineIndex, Boolean fromLeftToRight )
```

Parameter	Type	Description
start	Number	Start line offset
end	Number	End line offset
firstLineIndex	Number	Starting line index
lastLineIndex	Number	Ending line index
fromLeftToRight	Boolean	true for left-to-right selection, otherwise false

### Description

The `currentEditor.selectByLineIndex()` method allows you to change the selection of text in the document currently displayed in the Code editor using line index parameters, that is, without taking the collapsed/expanded status of lines into account. If you want to set the selection of text with respect to the collapsed/expanded status of lines, you should consider using the `currentEditor.selectByVisibleLine()` method.

Pass in *start*, *end*, *firstLineIndex*, *lastLineIndex* and *fromLeftToRight* parameters the new selection definition. For more information about these parameters, please refer to the `currentEditor.getSelectionInfo()` method description.

## currentEditor.selectByVisibleLine()

---

```
void currentEditor.selectByVisibleLine( Number start, Number end, Number firstVisibleLineIndex, Number lastVisibleLineIndex, Boolean fromLeftToRight )
```

Parameter	Type	Description
start	Number	Start line offset
end	Number	End line offset
firstVisibleLineIndex	Number	Starting visible line index
lastVisibleLineIndex	Number	Ending visible line index
fromLeftToRight	Boolean	true for left-to-right selection, otherwise false

### Description

The `currentEditor.selectByVisibleLine()` method allows you to change the selection of text in the document currently displayed in the Code editor using visible line index parameters, that is, by taking the collapsed/expanded status of lines into account. If you want to set the selection of text without worrying about the collapsed/expanded status of lines, you should consider using the `currentEditor.selectByLineIndex()` or `currentEditor.selectFromStartOfText` methods.

Pass in *start*, *end*, *firstVisibleLineIndex*, *lastVisibleLineIndex* and *fromLeftToRight* parameters the new selection definition. For more information about these parameters, please refer to the [currentEditor.getSelectionInfo\(\)](#) method description.

## currentEditor.selectFromStartOfText

---

void **currentEditor.selectFromStartOfText**( Number *offset*, Number *length*, Boolean *fromLeftToRight* )

Parameter	Type	Description
offset	Number	Starting selection offset
length	Number	Selection length
fromLeftToRight	Boolean	true to select from left to right, false otherwise

### Description

The `currentEditor.selectFromStartOfText` method allows you to change the selection of text in the document currently displayed in the Code editor by selecting the *offset* character to *offset+length* character. You can pass a negative value in *length*, so that the text before the *offset* character will be selected. The *offset* character will be evaluated from the beginning of the text and includes collapsed blocks. If the new selection overlaps a collapsed block, the block is automatically expanded.

Pass `true` in the *fromLeftToRight* parameter to select text from left to right, and `false` to select from right to left.

### Example

Considering the following content:

```
1
2  WAF.onAfterInit = function onAfterInit() {
3
4  // @region namespaceDeclaration ...
11
```

If you execute the following code:

```
studio.currentEditor.selectFromStartOfText(45,200,true)
```

The new selection will be:

```
1
2  WAF.onAfterInit = function onAfterInit() {
3
4  // @region namespaceDeclaration
5      var go = {}; // @button
6      var documentEvent = {}; // @document
7      var button2 = {}; // @button
8      var textField5 = {}; // @textField
9      var button1 = {}; // @button
10 // @endregion
11
```

But, if you execute the following code:

```
studio.currentEditor.selectFromStartOfText(45,205,true)
```

The new selection will be:

```
1
2  WAF.onAfterInit = function onAfterInit() {
3
4  // @region namespaceDeclaration ...
11
```

In this case, there is no need to expand the block, it is entirely selected.

## currentEditor.setAnnotation()

---

void **currentEditor.setAnnotation**( Number *lineIndex*, String *errorMsg* )

Parameter	Type	Description
lineIndex	Number	Line index where to add a warning symbol
errorMsg	String	Tip to display when the mouse hovers on the warning symbol

### Description

The `currentEditor.setAnnotation()` method allows you to add a warning symbol in the vertical annotation bar at the *lineIndex* line in the open document. Keep in mind that Wakanda's Code editor line numbering starts at 1, but JavaScript

indexes document lines starting at 0.

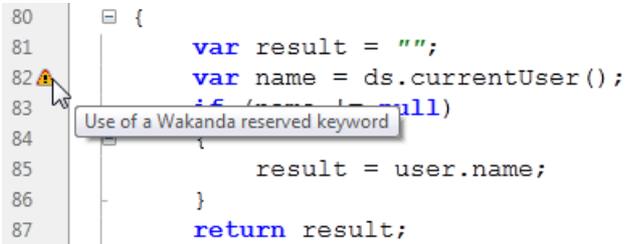
Pass in *errorMsg* the message to display as tip when the mouse hovers on the set symbol.

### Example

The following code:

```
studio.currentEditor.setAnnotation(81,"Use of a Wakanda reserved keyword");
```

... will add a warning symbol associated with a message in the open document:



```
80 {
81     var result = "";
82     var name = ds.currentUser();
83     if (name !== null)
84     {
85         result = user.name;
86     }
87     return result;
```

### currentEditor.setCaretPosition()

---

void **currentEditor.setCaretPosition**( Number *offset* )

Parameter	Type	Description
offset	Number	New position for the caret

### Description

The `currentEditor.setCaretPosition()` method moves the caret ( | ) to the defined *offset* position in the document currently opened in the Code editor.

The character position you pass in *offset* will be evaluated from the beginning of the text, including collapsed blocks. If the new caret position is within a collapsed block, it is automatically expanded.

## API: Extension

---

### Using 'studio' Object

All Wakanda Extension APIs are available through the "studio" object. Thus, you must prefix each API call with 'studio.'  
For example, to call the `alert()` method, you should write:

```
studio.alert("Hello World!");
```

### extension.getFolder()

---

Folder `extension.getFolder()`

Returns

Folder

Extension folder

### Description

The `extension.getFolder()` method returns a *Folder* object referencing the folder of the extension.

### Example

You can call this code in the `index.js` file to get the extension folder path:

```
var fold = studio.extension.getFolder().path;
```

If your extension is installed in the user data as described in the [Installing Extensions](#) section, the `fold` string will contain:

```
C:/Users/Arnaud/AppData/Roaming/Wakanda Studio/Extensions/Hello World Extension/
```

## API: GUI

---

- Each extension action associated to a **button** has two graphical properties:
  - alternative property (Boolean): the extension can change button icon, button title, or button tips by changing the associated action's alternative state.
  - enabled property (Boolean): the extension can make button enabled or disabled by setting enabled state to **true** or **false** respectively.
- Each extension action associated with a **menu item** has two graphical properties as well:
  - checked property (Boolean): the extension can check/uncheck a menu item by changing the associated action's checked state to **true** or **false**.
  - enabled property (Boolean): the extension can show or hide the item by setting the enabled state to **true** or **false** respectively.

### Using 'studio' Object

All Wakanda Extension APIs are available through the "studio" object. Thus, you must prefix each API call with 'studio.'  
For example, to call the `alert()` method, you should write:

```
studio.alert("Hello World!");
```

### checkMenuItem()

---

void **checkMenuItem**( String *actionName*, Boolean *isChecked* )

Parameter	Type	Description
actionName	String	actionName defined in the manifest.json file
isChecked	Boolean	True to check the menu item, false otherwise

### Description

The `checkMenuItem()` method allows you to set the checked state of the menu item associated to the *actionName*.  
Pass **true** in the *isChecked* parameter to check the menu item button and **false** to uncheck it.

### isActionAlternated()

---

Boolean **isActionAlternated**( String *actionName* )

Parameter	Type	Description
actionName	String	actionName defined in the manifest.json file
Returns	Boolean	True if the alternated button action state is on, false otherwise

### Description

The `isActionAlternated()` method returns **true** if the alternated state for the *actionName* of a button is on.  
The method returns **false** if the alternated state is off.

### isActionEnabled()

---

Boolean **isActionEnabled**( String *actionName* )

Parameter	Type	Description
actionName	String	actionName defined in the manifest.json file
Returns	Boolean	True if the enabled button action state is on, false otherwise

### Description

The `isActionEnabled()` method returns **true** if the enabled state for the *actionName* of a button is on.  
The method returns **false** if the enabled state is off.

### isMenuItemChecked()

---

Boolean **isMenuItemChecked**( String *actionName* )

Parameter	Type	Description
actionName	String	actionName defined in the manifest.json file
Returns	Boolean	True if the actionName menu item is checked, false otherwise

### Description

The `isMenuItemChecked()` method returns `true` if the menu item associated to the `actionName` is checked.  
The method returns `false` if the menu item is not checked.

## setActionAlternated()

---

void **setActionAlternated**( String *actionName*, Boolean *isAlternated* )

Parameter	Type	Description
<code>actionName</code>	String	<code>actionName</code> defined in the <code>manifest.json</code> file
<code>isAlternated</code>	Boolean	True to set the alternate state of the button, false otherwise

### Description

The `setActionAlternated()` method allows you to set the alternate state of the button associated to the `actionName`.  
Pass `true` in the `isAlternated` parameter to set the alternated state and `false` to remove it.

## setActionEnabled()

---

void **setActionEnabled**( String *actionName*, Boolean *isEnabled* )

Parameter	Type	Description
<code>actionName</code>	String	<code>actionName</code> defined in the <code>manifest.json</code> file
<code>isEnabled</code>	Boolean	True to enable the button action, false otherwise

### Description

The `setActionEnabled()` method allows you to set the enabled state of the button associated to the `actionName`.  
Pass `true` in the `isEnabled` parameter to enable the button and `false` to disable it.

## API: Preferences

---

This set of APIs allows the extension author to read or write extension settings, called preferences. A preference is a combination of a key and a value. You can use two different sets of extension preferences: general preferences and solution preferences.

### General preferences

General extension preferences are used by the Wakanda Studio application. They are shared by all solutions. General extension preferences are saved in the following file (optional):

- On Windows: *{Disk}:\Users\{User name}\AppData\Roaming\Wakanda Studio\ExtensionPreference\EXT\_FOLDER\_NAME\extension.prefs*
- On Mac OS: */Users/{User name}/Library/Application Support/Wakanda Studio/ExtensionPreference/EXT\_FOLDER\_NAME\extension.prefs*

... where *EXT\_FOLDER\_NAME* is the extension folder name.

### Solution extension preferences

Solutions extension preferences are set separately for each solution. They are designed to store solution-relative preferences, such as specific paths. Solution extension preferences are saved in the following file (optional):

- On Windows: *{Disk}:\Users\{User name}\Documents\Wakanda\{solution name}\{solution name} Solution\ExtensionPreference\EXT\_FOLDER\_NAME\extension.prefs*
- On Mac OS: */Users/{User name}/Documents/Wakanda/{solution name}/{solution name} Solution/ExtensionPreference/EXT\_FOLDER\_NAME\extension.prefs*

... where *EXT\_FOLDER\_NAME* is the extension folder name.

### Using 'studio' Object

All Wakanda Extension APIs are available through the "studio" object. Thus, you must prefix each API call with 'studio.'  
For example, to call the `alert()` method, you should write:

```
studio.alert("Hello World!");
```

### extension.deletePrefFile()

---

Boolean `extension.deletePrefFile()`

Returns Boolean True if the preference file was successfully deleted, false otherwise

#### Description

The `extension.deletePrefFile()` method removes the preference file from the disk. If the file was successfully deleted, the method returns `True`, otherwise (for example, if the file is locked), it returns `False`.

### extension.deleteSolutionPrefFile()

---

Boolean `extension.deleteSolutionPrefFile()`

Returns Boolean True if the solution preference file was successfully deleted, false otherwise

#### Description

The `extension.deleteSolutionPrefFile()` method removes the solution preference file from the disk. If the file was successfully deleted, the method returns `True`. Otherwise (for example, if the file is locked), it returns `False`.

### extension.getPref()

---

String `extension.getPref( String keyName )`

Parameter	Type	Description
keyName	String	Name of the preference key to read
Returns	String	Current value of the preference key

#### Description

The `extension.getPref()` method returns the current value of the *keyName* preference key in the extension preference file.

If the *keyName* key does not exist in the file, an empty string is returned.

---

## extension.getPrefFolderPath()

String **extension.getPrefFolderPath()**

Returns                      String                      Extension preference folder path

### Description

The `extension.getPrefFolderPath()` method returns the extension preference folder full path, where the extension can add its files.

If the extension preference folder does not exist yet when the method is called, it is created.

### Example

```
var prefPath = studio.extension.getPrefFolderPath();
// returns for example under Windows:
// 'C:\Users\{Name}\AppData\Roaming\Wakanda Studio\ExtensionPreference\Hello World Extension\
```

---

## extension.getSolutionPref()

String **extension.getSolutionPref( String keyName )**

Parameter	Type	Description
keyName	String	Name of the solution preference key to read
Returns	String	Current value of the solution preference key

### Description

The `extension.getSolutionPref()` method returns the current value of the *keyName* preference key in the solution extension preference file.

If the *keyName* key does not exist in the file, an empty string is returned.

---

## extension.getSolutionPrefFolder()

Folder **extension.getSolutionPrefFolder()**

Returns                      Folder                      Solution extension preference folder

### Description

The `extension.getSolutionPrefFolder()` method returns a reference to the solution extension preference folder, where the extension solution can add its files. The method returns a Folder object, that you can handle through the various properties and methods of the Folder class.

If the extension solution preference folder does not already exist when this method is called, it is created.

### Example

You want to display the current solution extension preference folder path:

```
var prefs = studio.extension.getSolutionPrefFolder();
studio.alert(prefs.path);
```

---

## extension.isPrefFileExisting()

Boolean **extension.isPrefFileExisting()**

Returns                      Boolean                      True if a preference file exists, False otherwise

### Description

The `extension.isPrefFileExisting()` method returns **true** if a preference file exists for the extension, and **false** otherwise. It can be useful for example to restore the factory default settings.

---

## extension.isSolutionPrefFileExisting()

Boolean **extension.isSolutionPrefFileExisting()**

Returns Boolean True if a solution preference file exists, False otherwise

## Description

The `extension.isSolutionPrefFileExisting()` method returns `true` if a preference file exists for the solution extension, and `false` otherwise.

It can be useful, for example, to restore the factory default settings.

## `extension.setPref()`

---

Boolean `extension.setPref( String keyName, String keyValue )`

Parameter	Type	Description
<code>keyName</code>	String	Name of the preference key to write
<code>keyValue</code>	String	New value for the preference key
Returns	Boolean	True if the value was successfully set, false otherwise

## Description

The `extension.setPref()` method writes a `keyName/keyValue` preference pair in the extension preference file. If the `keyName` preference was already defined in the file, its value is replaced by `keyValue`. If it was not defined, a new `keyName/keyValue` preference pair is added to the file.

The method returns `true` if it was successful and `false` otherwise.

## `extension.setSolutionPref()`

---

Boolean `extension.setSolutionPref( String keyName, String keyValue )`

Parameter	Type	Description
<code>keyName</code>	String	Name of the solution preference key to write
<code>keyValue</code>	String	New value for the solution preference key
Returns	Boolean	True if the value was successfully set, false otherwise

## Description

The `extension.setSolutionPref()` method writes a `keyName/keyValue` preference pair in the solution extension preference file. If the `keyName` preference was already defined in the file, its value is replaced by `keyValue`. If it was not defined, a new `keyName/keyValue` preference pair is added to the file.

The method returns `true` if it was successful and `false` otherwise.

## Example

You want to set a value to a "color" key:

```
var isOK = studio.extension.setSolutionPref("color", "blue");
if (isOK)
    studio.alert("Preference successfully saved");
```

## API: Solution

---

The "Solution" theme methods allow you to get information from the Solution level.

### Using 'studio' Object

All Wakanda Extension APIs are available through the "studio" object. Thus, you must prefix each API call with 'studio.'  
For example, to call the `alert()` method, you should write:

```
studio.alert("Hello World!");
```

### currentSolution.getExpandedFolders()

---

Array `currentSolution.getExpandedFolders()`

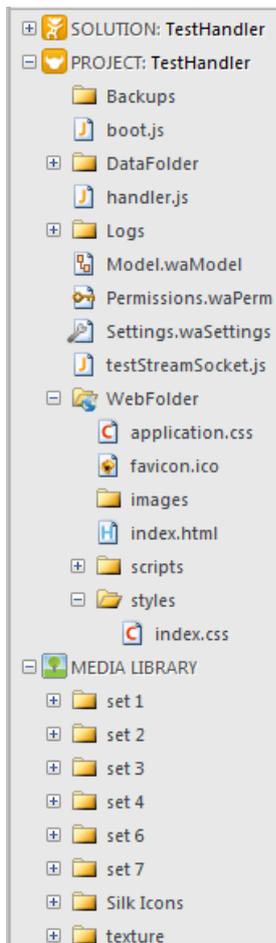
Returns Array Expanded folder(s)

### Description

The `currentSolution.getExpandedFolders()` method returns the list of folders which are currently expanded in the solution explorer window. The returned value is an array of *Folder* objects.

### Example

Given the following items in the solution explorer, if your solution is located at the root folder:



```
var arrExpand = studio.currentSolution.getExpandedFolders();  
// arrExpand[0].path contains "C:/TestHandler/TestHandler/"  
// arrExpand[1].path contains "C:/TestHandler/TestHandler/WebFolder/"  
// arrExpand[2].path contains "C:/TestHandler/TestHandler/WebFolder/styles/"  
// arrExpand[3].path contains "C:/Wakanda/Wakanda Studio/Resources/Web Components/walib/WAF/n
```

### currentSolution.getSelectedItems()

---

Array `currentSolution.getSelectedItems()`

Returns Array Selected item(s)

## Description

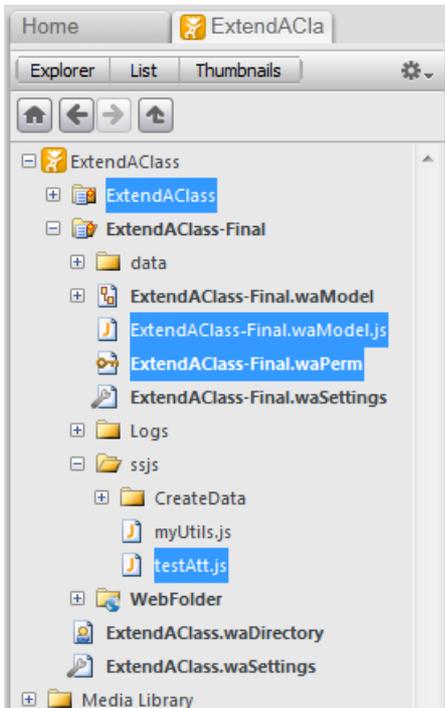
The `currentSolution.getSelectedItems()` method returns an array of selected files and folders in the Solution Explorer window. This information is useful when you need to execute an action on the selected items.

The array order is based on the user selection sequence: first items selected are in the first positions of the array. If no item is selected in the Solution Explorer, `currentSolution.getSelectedItems()` returns an empty array.

`currentSolution.getSelectedItems()` returns an array of objects of the *File* and/or *Folder* type.

## Example

Given the following items selected in the Solution Explorer, if your solutions are located at the root folder:



```
var arrSel = studio.currentSolution.getSelectedItems();  
// arrSel[0].path contains "C:/ExtendAClass/ExtendAClass/"  
// arrSel[1].path contains "C:/ExtendAClass/ExtendAClass-Final/ExtendAClass-Final.waPerm"  
// arrSel[2].path contains "C:/ExtendAClass/ExtendAClass-Final/ExtendAClass-Final.waModel.js"  
// arrSel[3].path contains "C:/ExtendAClass/ExtendAClass-Final/ssjs/testAtt.js"  
// in Wakanda v2 you get the path directly in arrSel[n]
```

## `currentSolution.getSolutionFile()`

---

File `currentSolution.getSolutionFile()`

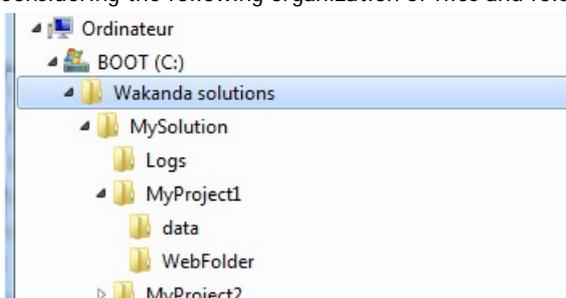
Returns File Current solution file

## Description

The `currentSolution.getSolutionFile()` method returns a *File* object referencing the current solution file (named '*SolutionName.waSolution*').

## Example

Considering the following organization of files and folders on disk:



```
var solPath=studio.currentSolution.getSolutionFile().path;
// returns C:\Wakanda solutions\MySolution\MySolution.waSolution
```

## currentSolution.restoreItemsIcon()

Boolean **currentSolution.restoreItemsIcon**( Array *filePathsToSet* )

Parameter	Type	Description
filePathsToSet	Array	Array of file full paths
Returns	Boolean	true if method executed successfully, false otherwise

### Description

The **currentSolution.restoreItemsIcon()** method removes any overlay icon added to icons of files referenced by the *filePathsToSet* parameter. Overlay icons can be added with the **currentSolution.setItemsOverlayIcon()** method.

In *filePathsToSet*, pass an array of strings (file full path names) to designate file icons whose overlay icon should be removed. If any item designated does not have an overlay icon or is a folder, it is ignored.

## currentSolution.setItemsOverlayIcon()

void **currentSolution.setItemsOverlayIcon**( Array *filePathsToSet* , String *iconFilePath* [, String *position*] )

Parameter	Type	Description
filePathsToSet	Array	Array of file full paths
iconFilePath	String	Path to the icon file
position	String	LowerRight (default), LowerLeft, UpperLeft or UpperRight

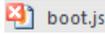
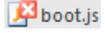
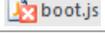
### Description

The **currentSolution.setItemsOverlayIcon()** method allows you to add an overlay icon to a set of file icons in the solution explorer area. This method can be used to 'mark' files that are involved in a specific action.

In *filePathsToSet*, pass an array of strings (file full path names) to designate files to receive an overlay icon. If a path designates a folder, it is ignored.

In *iconFilePath*, pass the full path of the icon file to use. The file must be in the *png* format. It is recommended that you use an 11x11 pixel icon.

The optional *position* parameter will define the position of the overlay icon on the solution file. You can pass one of the following strings:

position	Result
'UpperLeft'	
'UpperRight'	
'LowerLeft'	
'LowerRight' (default if omitted)	

An overlay icon will remain displayed during the entire session, unless you call **currentSolution.setItemsOverlayIcon()** again or the **currentSolution.restoreItemsIcon()** method on the item.

### Example

You want to add a specific overlay icon to the currently selected items. In the *index.js* file of the extension, you can add the following code:

```
var icons=[] // in;
var sel = studio.currentSolution.getSelectedItems();
for (var i = 0 ; i<sel.length;i++)
    icons[i] = sel[i].path;
var isOK = studio.currentSolution.setItemsOverlayIcon(icons,"C:/Graphic/mark.png","LowerLeft"
```

If the items selected include any folders, they are simply ignored.

## API: Storage

---

Storage features are useful when an extension needs to share information between `index.js` and the Web Zone Dialog. The Wakanda Studio Extension proposes a *Storage* object simply named `storage`, thus available through:

```
studio.extension.storage //extension storage object
```

*Note: For more information about Storage objects in Wakanda, please refer to the [Storage](#) section.*

### Using 'studio' Object

All Wakanda Extension APIs are available through the "studio" object. Thus, you must prefix each API call with 'studio.'  
For example, to call the `alert()` method, you should write:

```
studio.alert("Hello World!");
```

### extension.storage.clear()

---

```
void extension.storage.clear()
```

#### Description

The `extension.storage.clear()` method removes all the key/value pairs defined in the *storage* object.

### extension.storage.getItem()

---

```
String | Null extension.storage.getItem( String keyName )
```

Parameter	Type	Description
keyName	String	Name of key to get the value
Returns	Null, String	Value associated to the key

#### Description

The `extension.storage.getItem()` method returns the current value associated with the given *keyName*.  
If *keyName* is not an existing key in the *storage* object, the method returns **Null**.

### extension.storage.key()

---

```
String extension.storage.key( Number keyIndex )
```

Parameter	Type	Description
keyIndex	Number	Key index number
Returns	String	Key name

#### Description

The `extension.storage.key()` method returns the key name for a given *keyIndex* in the *storage* object.

### extension.storage.removeItem()

---

```
void extension.storage.removeItem( String keyName )
```

Parameter	Type	Description
keyName	String	Name of the key to remove

#### Description

The `extension.storage.removeItem()` method removes the *keyName* key and its associated value from the *storage* object.

### extension.storage.setItem()

---

```
void extension.storage.setItem( String keyName, String keyValue )
```

Parameter	Type	Description
keyName	String	Name of the key to set
keyValue	String	Value of the key to set

#### Description

The `extension.storage.setItem()` method associates the *keyValue* to the given *keyName* in the *storage* object.

## API: Studio

---

### Using 'studio' Object

All Wakanda Extension APIs are available through the "studio" object. Thus, you must prefix each API call with 'studio.'  
For example, to call the `alert()` method, you should write:

```
studio.alert("Hello World!");
```

### Buffer()

---

void **Buffer**( Number | Array | String *definition* [, String *encoding*] )

Parameter	Type	Description
definition	Number, Array, String	Size (number or array) of the buffer or string to set to the buffer
encoding	String	Encoding method (if a string is passed in definition)

### Description

The `Buffer()` constructor method allows you to create and handle a SSJS *Buffer* type object from your extension code.  
For more information about *Buffer* objects, please refer to the [Buffers](#) class description.

### Example

We want to create a new Buffer containing the selected items in the Solution Explorer:

```
var arrSel = studio.currentSolution.getSelectedItems();  
var myBuffer = new studio.Buffer(arrSel);
```

### File()

---

File **File**( String | Folder *path* [, String *fileName*] )

Parameter	Type	Description
path	String, Folder	Path of the file to reference
fileName	String	Name of the file to reference
Returns	File	New File object

### Description

The `File()` constructor method allows you to create and handle a SSJS *File* type object from your extension code.  
For more information about *File* objects, please refer to the [File](#) description.

### Example

We want to create a new *File* object referencing the current opened document:

```
var fileRef = studio.File(studio.currentEditor.getPath());
```

### Folder()

---

Folder **Folder**( String *path* )

Parameter	Type	Description
path	String	Path of the folder to reference
Returns	Folder	New Folder object

### Description

The `Folder()` constructor method allows you to create and handle a SSJS *Folder* type object from your extension code.  
For more information about *Folder* objects, please refer to the [Folder](#) description.

### Example

We want to create a new *Folder* object referencing the preferences folder:

```
var folderRef = studio.Folder(studio.extension.getPrefFolderPath());
```

### sendCommand()

---

Boolean **sendCommand**( String *commandName* )

Parameter	Type	Description
commandName	String	Action to execute
<b>Returns</b>	Boolean	true if the method executed successfully, false otherwise

## Description

The `sendCommand()` method runs the Wakanda Studio menu command or another extensions' action defined in the `commandName` parameter. The method returns `true` if the command was called with success, and `false` otherwise (for example, if `commandName` does not exist).

- To execute a command from the Wakanda Studio, pass one of the following strings in `commandName`:

```
About
Close
CloseSolution
NewCatalog
NewCSS
NewFile
NewFolder
NewGUI
NewHTML
NewJavascript
NewJSON
NewMobileGUI
NewPHP
NewProject
NewSolution
NewTabletGUI
NewTXT
NewWebComponent
NewXML
OpenFile
OpenSolution
Save
SaveAll
SaveAs
```

- To execute an action from another extension, use the following format in `commandName`:

```
EXTENSIONNAME.ACTIONNAME
```

where `EXTENSIONNAME` is the folder name of extension and `ACTIONNAME` is the action message name. When running `sendCommand()` to call the action of another extension, the destination extension will receive "fromExtension" as `message.event` in the `handleMessage` function. For more information, please refer to [handleMessage Function](#) paragraph.

## `studio.SystemWorker()`

---

```
void studio.SystemWorker ( commandLine , executionPath )
```

Parameter	Type	Description
<code>commandLine</code>	String	Command line to execute
<code>executionPath</code>	String, Folder	Directory where command is executed

## Description

The `studio.SystemWorker()` constructor method allows you to create and handle a SSJS `SystemWorker` type object from your extension code.

For more information about `SystemWorker` objects, please refer to the [SystemWorker Instances](#) description.

## API: Web Zone Dialog

---

Wakanda Studio API provides ways to launch modal or non modal Web zones. It could be useful when an extension needs a customizable dialog box.

Use the Wakanda Studio extension *Storage* object (*studio.extension.storage*) to share information between modal/modless dialog boxes and *index.js*. If you want to get values from the dialog in *index.js*, the extension lifetime should be set as *application\_lifetime*.

*Note: For more information about studio.extension.storage, please refer to the [API: Storage](#) chapter.*

### extension.quitDialog()

---

void **extension.quitDialog()**

#### Description

The **extension.quitDialog()** method closes the dialog box opened by or .

After having opened an HTML dialog, it is recommended that you attach this method to an OK or a Cancel button (or both) in your HTML page code.

### extension.showModalDialog()

---

Boolean **extension.showModalDialog**( String *htmlPage* [,String *arguments* [,Object *params* [,String *callback*]]) )

Parameter	Type	Description
htmlPage	String	Relative file path to the HTML page to load
arguments	String	Arguments to process
params	Object	Window parameters: {title (string), dialogwidth (number), dialogheight (number), resizable (boolean)}
callback	String	Callback function
Returns	Boolean	True if the dialog box was validated, false otherwise

#### Description

The **extension.showModalDialog()** method opens a modal dialog box displaying the *htmlPage*.

Pass in the *htmlPage* parameter a file path, relative to the extension folder, indicating which HTML page to load.

*arguments* is an object or a valid javascript value containing any parameters to pass to the HTML page.

On the HTML page side, you will access these *arguments* through the *userArguments* key of the Studio *Storage* object. For example, you can use an instruction such as:

```
var myArgs = studio.extension.storage.getItem( 'userArguments' );
```

You can pass in *params* an object containing title and size parameters as properties:

- *title* (string): title for the dialog box. Example {title: "Select Settings"}. By default if this parameter is omitted, the title area is empty.
- *dialogwidth* (number): width of the dialog box in pixels. By default if this parameter is omitted, the width is 640 pixels.
- *dialogheight* (number): height of the dialog box in pixels. By default if this parameter is omitted, the height is 400 pixels.
- *resizable* (boolean): true if the dialog box must be resizable, false otherwise. By default if this parameter is omitted, the dialog is resizable.

The HTML modal dialog is executed asynchronously. If you want to get a result from the HTML dialog, you need to define a *callback* function, that will be called when the dialog is closed.

Again, you can use the Studio *Storage* object. For example, you could put the result value into the **studio.extension.storage.returnValue** key and get this value in *callback* function. When the HTML dialog is closed, the *callback* function is executed, then you get back in the *index.js* file any result from your dialog.

Note that the *callback* function should be defined in the same way as the other actions.

#### Example

We want to open a custom Settings dialog box to allow the user to set parameters.

- In the *index.js* file, we added the following actions:

```
//the settings action is called when the user clicks a button
actions.settings = function settings(message) {
  var option;
  option = DefaultOption;
  option = getOptFromPref(option); // gets current values from existing preferences
  studio.extension.showModalDialog(
    "settingsDialog.html",
```

```

    option,
    {title:"My Settings", dialogwidth:470, dialogheight:380, resizable:false},
    'writeOptions');
};

//The "writeOptions" callback function
actions.writeOptions = function writeOptions(message) {
    var newOption = studio.extension.storage.returnValue; // gets values from the dialog
    if (newOption) //if there are new values
    {
        studio.extension.setPref("pref1", newOption.pref1);
        studio.extension.setPref("pref2", newOption.pref2);
        //...
    }
}

```

- In the HTML page named "settingsDialog.html", you should have defined the corresponding functions, for example:

```

function init() {
    document.getElementById('pref1').value = studio.extension.storage.dialogArguments.pref1;
    document.getElementById('pref2').value = studio.extension.storage.dialogArguments.pref2;
    setValidation();
}

function getValueAndQuitHtmlPage() {
    var hpref1;
    var hpref2;

    hpref1= document.getElementById('pref1').value;
    hpref2= document.getElementById('pref2').value;

    studio.extension.storage.returnValue = {
        "pref1":hpref1,
        "pref2":hpref2
    };
    studio.extension.quitDialog();
}

```

## extension.showModelessDialog()

---

Boolean **extension.showModelessDialog**( String *htmlPage* [,String *arguments* [,Object *params* [,String *callback*]])

Parameter	Type	Description
htmlPage	String	Relative file path to the HTML page to load
arguments	String	Arguments to process
params	Object	Window parameters
callback	String	Callback function name
Returns	Boolean	True if the dialog box was validated, false otherwise

### Description

The **extension.showModelessDialog()** method opens a non modal dialog box displaying the *htmlPage*.

This method is similar to the **extension.showModalDialog()** method, except that it opens a non modal dialog box.