The Resource Object

Overview

A Resource object represents a front-end file that is required for any Yellowfin code widgets to function. The Resource class defines the file path, and type, and how it should be handled by Yellowfin. Yellowfin uses this definition to load the Resource into the browser window immediately after loading the related code widget.

Constructors

public Resource(String path, String contentType);

public Resource(String path, String contentType, boolean isLibrary);

public Resource(String path, String contentType, String alias, boolean isLibrary)

Properties

path property

This string is the path to the resource being defined. The path is a relative path from the plugin that is being defined.

contentType property

This string is the type of front-end file to be accessed by the call.

Currently, Yellowfin supports:

- *text/javascript*
- *text/css*
- *text/html*

Yellowfin determines how to handle the file; each type will be handled slightly differently.

File type	What Yellowfin does with it
text/javascript	The file's content is passed as an HTML string to the define callback function.
text/css	The file's content is immediately be inserted into the page.
text/html	The file's content **is** passed a JavaScript Object to the define callback functions.

JavaScript example

```
mySimpleObject.js
define(function() {
     let myObject = {
           objectProperty: ‘hello world’
     }

     return myObject;
});

myClass.js
define(function() {
     //Create a JS class so that we can initialise it in another file.
    let myClass= function() {

    }

    myObject.prototype.hello = function() {
        alert(‘hello world’);
    }

    return myObject;
});

This can then be included in the main.js file:
main.js
define([‘mySimpleObject.js’, ‘myInitialisedableObject.js’], function(MyObject, MyClass) {
    //Alert the value of objectProperty
    alert(MyObject.objectProperty);

    //Initialise an instance of MyClass and then call the hello() function
    let classInstance = new MyClass();
    classInstance.hello();
});
```

alias property

This string enables developers to create a shorthand mapping to refer to a particular file. Whenever this alias is referenced in the define array, it will point to the path variable.

new Resource(“myWidget/js/MyWidget.js”, “text/javascript”, “MyWidget”);

Can then be referred to in the define arrays as:

define([‘MyWidget’], function(MyWidget) { });

isLibrary

This Boolean enables developers to load JavaScript libraries in an unchanged fashion. This can be used for loading third-party libraries into a code widget.

By default, when a file is requested to load, it will load in a certain way so that it can be bundled automatically into a “widget JS file” which means that a single request can be made to load many code widget files. However, not all JS files will successfully load when this is used, so this flag can be used to ensure that the file is loaded separately.

new Resource(“myWidget/js/jquery.js”, “text/javascript”, true);

top

Next steps

When you're ready to get started with code widgets, follow this process:

1. Write the back-end code.
2. Write the front-end code.
3. Customize the Code Widget Properties panel (optional).
4. Bundle the code into a jar file, then upload it to Yellowfin.

To further assist you during code widget development, click on the code widget reference page for samples, API links and detailed descriptions.

top