|  |
| --- |
| Microsoft Corporation |
| RIN JavaScript SDK |
| Developer Documentation v1.0 |
| Microsoft Research |
| **Abstract** |
| This document describes how to extend the JavaScript RIN player by adding new Experience providers. |

Table of Contents:

[1 Introduction 2](#_Toc348097360)

[2 Experience Provider development 2](#_Toc348097361)

[2.1 Sample 1: “Hello World” Experience Provider 2](#_Toc348097362)

[2.2 Sample 2: “Image ES” – A Discrete Keyframe based Experience Provider 4](#_Toc348097363)

[2.3 Communicating with RIN Core Host 5](#_Toc348097364)

[2.4 Building your Experience Provider 6](#_Toc348097365)

[2.5 Replacing or adding Player Controller experience streams 7](#_Toc348097366)

[3 Using Interaction Controls 8](#_Toc348097367)

[3.1 Using the built-in interaction controls 8](#_Toc348097368)

[3.2 Creating custom interaction controls 8](#_Toc348097369)

[4 Deploying your Experience Stream 9](#_Toc348097370)

# Introduction

Rin.js is a JavaScript library that plays Rich Interactive Narratives (RIN). Please see the MSR Technical Report [MSR-TR-2012-78](http://research.microsoft.com/apps/pubs/default.aspx?id=170526) for an introduction to RIN. Experience Providers are pluggable components that are the primary mechanism to extend RIN functionality.

Please refer to the document “RIN JavaScript Player Documentation” for basic instructions on downloading the JS libraries and hosting the RIN player.

# Experience Provider development

Experience providers can be built from scratch or using one of the sample projects as a starting point. It’s best to start by understanding how the sample experience streams work.

## Sample 1: “Hello World” Experience Provider

Hello World Experience Provider is a sample skeleton provider that explicitly implements all the functions required. The implantation is contained in the JavaScript file “lib\HelloWorldExperience.js”.

To use the Hello World Experience Provider, simply add the script reference to the script file in your HTML page. Refer to *Incorporating the Custom Experience Providers into your HTML page* in the *RIN JavaScript Player Documentation* for more details.

Let’s go through the structure of “HelloWorldExperience.js”. The entire script is encapsulated in the body of an anonymous function that creates and registers the provider under the name “HelloWorldExperienceProvider”:

(function () {

// Instantiate the provider object…

var HelloWorldExperienceProvider = ...

// Register the provider with RIN. More precisely, register a

// factory function that knows how to create an instance of the

// provider bound to an Experience Stream (parameter esData)…

**rin.ext.registerFactory**(

rin.contracts.systemFactoryTypes.esFactory, // factory type

"HelloWorldExperienceProvider",

function (orchestrator, esData) {

return new HelloWorldExperienceProvider(orchestrator, esData);

});

// Experience-specific initialization code can go here

...

}());

The script sets up the HelloWorldExperienceProvider object so that it inherits the methods expected by RIN:

HelloWorldExperieneProvider.prototype = {

// Load the experience, with the specified experienceStream

// as default

**load**: function (experienceStreamId) {...},

// Play contents from the given offset & experienceStreamId

**play**: function (offset, experienceStreamId) {...},

// Pause experience stream with the first frame displayed

// at given offset & experienceStreamId

**pause**: function (offset, experienceStreamId) {...},

// Release all resources and unload

**unload**: function () {...},

// Return current state - one of states listed in

// rin.contracts.experienceStreamState

**getState**: function () {...},

// Return html element that displays contents of this

// experience provider.

**getUserInterfaceControl**: function () {...},

// One of the experience streams has raised an ESEvent.

// Take appropriate action (if any). <Optional>

**onESEvent**: function (sender, eventId, eventData) {...},

// Set the base volume. <Optional>

**setVolume**: function (baseVolume) {...},

// Mute or unmute. <Optional>

**setIsMuted**: function (value) {...},

// private prototype fields for the Experience

// may be defined here ...

...

}

This is all that is required to register an Experience Provider factory function with RIN. RIN calls this factory function each time it needs to play a narrative that contains an Experience with the named provider. The function is called with two parameters: orchestrator and esData. The orchestrator parameter is a proxy to the RIN core orchestrator and supports a number of methods that Experience Providers typically need to call while rendering the Experience. These methods are documented in Section 2.3. The esData parameter is the actual Experience data as defined in the RIN data specification.

## Sample 2: “Image ES” – A Discrete Keyframe based Experience Provider

The previous section explained the structure of a skeleton Experience Provider which may be used as a template for writing providers of arbitrary complexity. In many cases, however, a more constrained form of Experience Provider may suffice, in particular, one where continuous rendering of the experience is not required. Instead rendering is only required at discrete points in time that correspond to the occurrence of Keyframes in the Experience Stream. For such Experiences, the SDK has an implementation of “base class”, DiscreteKeyframeESBase”, that does much of the bookkeeping required to render the Experience as it transitions through various player states and switches from one Experience Stream to another.

Image Experience Provider Sample is an example of an Experience Provider that extends DiscreteKeyframeESBase. The provider supports panning, zooming and rotation of an image, either as specified in Keyframes or under the direction of interaction controls that are registered with the player.

The Provider is implemented in JavaScript file “ImageES.js.” We present the outline of the code below.

The ImageES provider object is defined in an anonymous function that takes as its argument the discrete keyframe base class implementation:

var ImageES = (function (\_super) {

...

})(**rin.contracts.DiscreteKeyframeESBase**);

//Register the ES factory so that orchestrator can invoke it

**rin.ext.registerFactory**(

rin.contracts.systemFactoryTypes.esFactory,

"MicrosoftResearch.Rin.ImageExperienceStream",

function (orchestrator, esData) {

return new ImageES(orchestrator, esData);

});

The above code registers a factory function under the name “ImageExperienceStream”. The function creates instances of the ImageES object. Each instance is bound to a specific Experience, which is a specific image in this case.

Refer to the sample provider source code for details of how it implements the displayKeyframe method and other methods the “base class” calls to actually render content and hook into player interaction controls.

## Communicating with RIN Core Host

Experience Providers can communicate with host through “orchestrator” parameter provided in factory. Orchestrator is a proxy object to RIN core and has below members:

|  |  |  |
| --- | --- | --- |
| Name | Type | Comments |
| getCurrentLogicalTimeOffset() | Method | Returns current time offset of player in seconds |
| getRelativeLogicalTime(experienceStreamId, absoluteLogicalTime) | Method | Returns time relative current experience provider. Params: experienceStreamId: optional param to point to specific experience stream. absoluteLogicalTime: optional absolute time, if not specified, assumed to be current player time. |
| play(offset, screenPlayId) | Method |  |
| pause(offset, screenPlayId) | Method |  |
| getIsMuted() | Method |  |
| setIsMuted(value) | Method |  |
| onESEvent(eventId, eventData) | Method | Call this method from experience provider to send a custom event notification to other experience providers. Standard set of event id are available in rin.contracts.ESEventIds enumeration. |
| getPlayerState() | Method | Returns current player state from rin.contracts.PlayerState enumeration. |
| getIsOnStage() | Method | Returns bool indicating if the current experience provider is playing on stage. |
| resolveReference(resourceItemId) | Method | Returns the resolved URL for the given resourceItemId based on current theme and language locale. |
| getCurrentStateSeekUrl() | Method | Returns a URL that indicates current seek state of player. This url can be later passed to seekUrl call to restore the state. |
| seekUrl(seekUrl) | Method | Navigates to the state as described by the seekUrl. |

## Building your Experience Provider

To build your Experience Stream, you can use one of samples as starting point or build your own project. The steps are:

* Add a new js file. If keyframes in your EP simply jumps in state between keyframes, you can save time by inheriting from rin.contracts.DiscreteKeyframeESBase (as described in Sample 2)
* The experience providers should implement below public members.

|  |  |  |
| --- | --- | --- |
| Name | Type | Comments |
| load(experienceStreamId) | method | Load experience data during this call. Called only one. experienceStreamId param indicates the experience that’ll be played. |
| play(offset, experienceStreamId) | method | EP should play the contents at given time offset and experienceStreamId. |
| pause(offset, experienceStreamId) | method | EP should display the contents at given time offset and experienceStreamId and be in paused state. |
| unload() | method | Unload objects loaded in memory and release memory to prepare for removal from stage |
| getState() | method | Return one of valid states from rin.contracts.ExperienceStreamState enum. Values are: Closed, Buffering, Ready, Error. The initial default state should be Closed. The state moves to Buffering when loading is going on. Once the experience provider is ready to play, the state should change to Ready. If any error occurs during loading, the state should change to Error. |
| stateChangedEvent | rin.contracts.Event | Event to be triggered whenever experience provider state changes. The event args should be of type rin.contracts.ESStateChangedEventArgs |
| getUserInterfaceControl() | method | EP should return the HTML element where contents are displayed. If the EP has no user interface, return null. |
| onESEvent(sender, eventId, eventData) | method | This is an optional method to be implemented if communication across experience providers is needed. If implemented, this method is called by RIN core whenever an ES event is raised by other experience providers. |
| setVolume(baseVolume) | Method | This is an optional method to be implemented if the experience provider supports volume control |
| setIsMuted(value) | Method | This is an optional method to be implemented if the experience provider supports mute |

* The experience provider needs to be registered by calling below method:

rin.Ext.registerESFactory("CompanyName.rin.UniqueProviderName", function (orchestrator, esData) { return new rin.MyExperienceName(orchestrator, esData); });

rin.Ext.registerESFactory takes a unique provider ID as first param. To make the provider ID unique, use this suggested format <CompanyName>.Rin.<ProviderName>. The second parameter is factory function.

The factory function has 2 parameters. First one is “orchestrator” – this is a proxy object to host. See [Communicating with RIN Core Host](#_Communicating_with_RIN) for details. The second parameter “esData” is javascript data object specific to the Experience Provider as given in the xrin JSON file.

## Replacing or adding Player Controller experience streams

If you want to change the look & feel of the player controller, below are the options:

1. If the changes required are only image resources for buttons and css style changes, these can be done by changing files under systemResources\themeResources. It is also possible to add a different theme altogether. Details are covered in the RIN themeing document.
2. If totally different controller is required, then a new controller ES can be implemented and registered with RIN core. Look at src\experiences\PlayerControllerES.js for example on how to build a controller ES.
3. If no controller is required or if controllers are located away from RIN player itself, we recommend running without controller using “controls=false” option. The RIN player can be programmatically controlled from a custom controller.

# Using Interaction Controls

Interaction controls are the controls that are displayed in the player’s footer when the user interacts with any experience stream. Interaction controls can be custom-made for any experience stream, or the built-in interaction controls can be used for common scenarios.

## Using the built-in interaction controls

Following interaction controls are available in rin-experiences-1.0.js library.

1. Pan and zoom controls:
2. Selection interaction controls:
3. Audio/video interaction controls:

## Creating custom interaction controls

To create custom interaction controls, you can use one of samples as starting point or build your own project. The steps are:

* Add a html file for adding the UI for the interaction controls
* Add a css file for styling the controls
* The interaction control needs to be registered by calling below method:

rin.ext.registerInteractionControlFactory("MicrosoftResearch.Rin.InteractionControls.RotateControl",function (resourcesResolver, loadedCallback) { $.get(resourcesResolver.resolveSystemResource("interactionControls/RotateControl.html"), null, function (visual) {

var wrap = document.createElement("div"),

systemRoot = resourcesResolver.getSystemRootUrl();

wrap.style["display"] = "inline-block";

wrap.innerHTML = visual.replace(/SYSTEM\_ROOT/g, systemRoot);

loadedCallback(wrap);

});

});

rin.Ext.registerInteractionControlFactory takes a unique interaction control ID as first param. To make the interaction control ID unique, use this suggested format <CompanyName>.Rin.InteractionControls.<InteractionControlName>. The second parameter is factory function.

The factory function has 2 parameters. First one is “resourcesResolver” – this is a proxy object to the resource resolving mechanism used by the player. This can be used to get full path to system resources by passing in relative paths. The second parameter “loadedCallback” is a callback to the experience stream that requested the interactionControl, signaling the completion of load and thus returning the html element.

* In the experience stream that has to display this interaction control, implement the following public method

this.getInteractionControls = function () {

var interactionControls = document.createElement("div");

self.\_orchestrator.getInteractionControls(

["MicrosoftResearch.Rin.InteractionControls.RotateControl", rin.contracts.interactionControlNames.panZoomControl],

function (wrappedInteractionControls) {

interactionControls.innerHTML = wrappedInteractionControls.innerHTML;

ko.applyBindings(self, interactionControls);

});

return interactionControls;

};

* In the getInteractionControls method of the Experiencestream, query the orchestrator’s getInteractionControls method and return the interaction controls ui element to be added to the player footer. The orchestrator’s getInteractionControls method takes two arguments; first argument is an array of the interaction control unique IDs, the same id with which the control was registered to the InteractionControlFactory in the step above, second argument is a callback function from the InteractionControl, after loading its html. Inside the callback function, you can bind or hookup to the UI events and implement the required functionality.

# Deploying your Experience Stream

Your experience stream can be deployed as a js file under web\lib folder or any other place as you desire. This js file can be included in any html page running your RIN using script tag, after rin-core.js is loaded.