Using JSValue

Edit

> **This documentation and the underlying platform code is a work in progress.**

`JSValue` is a native, immutable invariant value type, and is meant to hold any of the commonly used JS types: `bool`s, `int`s, `double`s, `string`s, arrays, and objects. It is provided for native developers (writing native modules or view managers) who want an equivalent to the `folly::dynamic` type that is compatible with the WinRT ABI surface provided by `Microsoft.ReactNative`.

Two `JSValue` implementations are provided: one for C++ developers in the `Microsoft.ReactNative.Cxx` shared project, and one for C# developers in the `Microsoft.ReactNative.SharedManaged` project.

> This purpose of this document is to provide equivalency to the scenarios supported by `folly/dynamic.h`.

C# C++/WinRT

Overview

Here are some code samples to get started (assumes a `using Microsoft.ReactNative.Managed;` was used):

```
JSValue nul = JSValue.Null;
JSValue boolean = false;

// Arrays can be initialized with JSValueArray.
JSValueArray array = new JSValueArray() { "array ", "of", 4, " elements" };
Debug.Assert(array.Count == 4);
JSValueArray emptyArray = new JSValueArray();
Debug.Assert(emptyArray.Count == 0);

// JSValueArrays can be implicitly converted to JSValues, however
// like all JSValues they will be immutable. Accessing the contents
// via the AsArray() method will return an IReadOnlyList<JSValue>.
JSValue array2 = array;
Debug.Assert(array2.AsArray().Count == 4);
JSValue emptyArray2 = emptyArray;
Debug.Assert(emptyArray2.AsArray().Count == 0);

// Maps from strings to JSValues are called objects. The
// JSValueObject type is how you make an empty map from strings
// to JSValues.
JSValueObject map = new JSValueObject();
map["something"] = 12;
map["another_something"] = map["something"].AsInt32() * 2;

// JSValueObjects may be initialized this way
JSValueObject map2 = new JSValueObject()
{
    { "something", 12 },
    { "another_something", 24 },
};

// Like JSValueArrays, JSValueObjects can also be implicitly
// converted to JSValues, and likewise they will be immutable.
// Accessing the contents via the AsObject() method will return an
// IReadOnlyDictionary<string, JSValue>.
JSValue map3 = map;
Debug.Assert(map3.AsObject().Count == 2);
JSValue map4 = map2;
Debug.Assert(map4.AsObject().Count == 2);
```

`JSValue` s require checking at runtime that the stored type is compatible with the operation. Some operations may throw runtime exceptions or produce unexpected behavior as type conversions fail and default values are returned.

More examples should hopefully clarify this:

```
JSValue dint = 42;

JSValue str = "foo";
JSValue anotherStr = str + "something"; // fine
JSValue thisDoesNotCompile = str + dint; // compilation error
```

Explicit type conversions can be requested for some of the basic types:

```
JSValue dint = 12345678;
JSValue doub = dint.AsDouble(); // doub will hold 12345678.0
JSValue str = dint.AsString(); // str == "12345678"

JSValue hugeInt = long.MaxValue; // hugeInt = 9223372036854775807
JSValue hugeDoub = hugeInt.AsDouble(); // hugeDoub = 9.2233720368547758E+18
```

Iteration and Lookup

You can iterate over `JSValueArray` s as you would over any C# enumerable.

```
JSValueArray array = new JSValueArray() { 2, 3, "foo" };

foreach (var val in array)
{
    doSomethingWith(val);
}
```

You can iterate over `JSValueObject` s just like any other `IDictionary<string, JSValue>` .

```
        // Key is kvp.Key, value is kvp.Value
        processKey(kvp.Key);
        processValue(kvp.Value);
    }

    foreach (var key in obj.Keys)
    {
        processKey(key);
    }

    foreach (var value in obj.Values)
    {
        processValue(value);
    }
```

You can find an element by key in a `JSValueObject` using the `TryGetValue()` method, which takes the key and returns `true` if a key is present and provides the value as an out variable. If the key is not preset, it returns `false` and the out variable will be null.

```
    JSValueObject obj = new JSValueObject() { { "2", 3}, { "hello", "world" }, { "x", } };

    if (obj.TryGetValue("hello", out JSValue value))
    {
        // value is "world"
    }

    if (obj.TryGetValue("no_such_key", out JSValue value2))
    {
        // this block will not be executed
    }
    // value2 is null
```

Use for JSON

Performance

`JSValue` s can be useful for manipulating large and complex JS objects in your native code, giving you random access to just the values you need. However, note that there is a performance penalty to doing this, as the entirety of the JS object will be parsed into the `JSValue` before it is passed to your code.

The performance hit of using `JSValue` in your external native code is in addition to the performance hit of `Microsoft.ReactNative` 's own internal use of `folly::dynamic` . Data is marshaled out of `Microsoft.ReactNative` through a high-performance serialization interface, however reading that data back into a `JSValue` means taking the time and memory to completely re-construct the original object structure.

For more information, see Marshaling Data.

‹ Using Asynchronous Windows APIs

Compile time code generation for C# ›