Mojo JavaScript Bindings API

This document is a subset of the Mojo documentation.

Contents

Getting Started

The bindings API is defined in the `mojo` namespace and implemented in `mojo_bindings.js`, which could be generated by the GN target `//mojo/public/js:bindings`.

When a Mojom IDL file is processed by the bindings generator, JavaScript code is emitted in a `.js` file with the name based on the input `.mojom` file. Suppose we create the following Mojom file at `//services/echo/public/interfaces/echo.mojom`:

```
module test.echo.mojom;

interface Echo {
  EchoInteger(int32 value) => (int32 result);
};
```

And a GN target to generate the bindings in `//services/echo/public/interfaces/BUILD.gn`:

```
import("//mojo/public/tools/bindings/mojom.gni")

mojom("interfaces") {
  sources = [
    "echo.mojom",
  ]
}
```

Bindings are generated by building one of these implicitly generated targets (where “foo” is the target name):

- `foo_js` JavaScript bindings; used as compile-time dependency.
- `foo_js_data_deps` JavaScript bindings; used as run-time dependency.

If we then build this target:

```
ninja -C out/r services/echo/public/interfaces:interfaces_js
```

This will produce several generated source files. The one relevant to JavaScript bindings is:

```
out/gen/services/echo/public/interfaces/echo.mojom.js
```

In order to use the definitions in `echo.mojom`, you will need to include two files in your html page using `<script>` tags:

- `mojo_bindings.js` **Note: This file must be included before any `.mojom.js` files.**
- `echo.mojom.js`

```
<!DOCTYPE html>
<script src="URL/to/mojo_bindings.js"></script>
<script src="URL/to/echo.mojom.js"></script>
<script>

var echoPtr = new test.echo.mojom.EchoPtr();
var echoRequest = mojo.makeRequest(echoPtr);
// ...

</script>
```

Interfaces

Similar to the C++ bindings API, we have:

- `mojo.InterfacePtrInfo` and `mojo.InterfaceRequest` encapsulate two ends of a message pipe. They represent the client end and service end of an interface connection, respectively.
- For each Mojom interface `Foo`, there is a generated `FooPtr` class. It owns an `InterfacePtrInfo`; provides methods to send interface calls using the message pipe handle from the `InterfacePtrInfo`.
- `mojo.Binding` owns an `InterfaceRequest`. It listens on the message pipe handle and dispatches incoming messages to a user-defined interface implementation.

Let's consider the `echo.mojom` example above. The following shows how to create an `Echo` interface connection and use it to make a call.

```
<!DOCTYPE html>
<script src="URL/to/mojo_bindings.js"></script>
<script src="URL/to/echo.mojom.js"></script>
<script>

function EchoImpl() {}
EchoImpl.prototype.echoInteger = function(value) {
  return Promise.resolve({result: value});
};

var echoServicePtr = new test.echo.mojom.EchoPtr();
var echoServiceRequest = mojo.makeRequest(echoServicePtr);
var echoServiceBinding = new mojo.Binding(test.echo.mojom.Echo,
                                          new EchoImpl(),
                                          echoServiceRequest);
echoServicePtr.echoInteger({value: 123}).then(function(response) {
  console.log('The result is ' + response.value);
});

</script>
```

Interface Pointers and Requests

In the example above, `test.echo.mojom.EchoPtr` is an interface pointer class. `EchoPtr` represents the client end of an interface connection. For method `EchoInteger` in the `Echo` Mojom interface, there is a corresponding `echoInteger` method defined in `EchoPtr`. (Please note that the format of the generated method name is `camelCaseWithLowerInitial`.)

There are some control methods shared by all interface pointer classes. For example, binding/extracting `InterfacePtrInfo`, setting connection error handler, querying version information, etc. In order to avoid name collision, they are defined in `mojo.InterfacePtrController` and exposed as the `ptr` field of every interface pointer class.

In the example above, `echoServiceRequest` is an `InterfaceRequest` instance. It represents the service end of an interface connection.

`mojo.makeRequest` creates a message pipe; populates the output argument (which could be an `InterfacePtrInfo` or an interface pointer) with one end of the pipe; returns the other end wrapped in an `InterfaceRequest` instance.

Binding an InterfaceRequest

A `mojo.Binding` bridges an implementation of an interface and a message pipe endpoint, dispatching incoming messages to the implementation.

In the example above, `echoServiceBinding` listens for incoming `EchoInteger` method calls on the messsage pipe, and dispatches those calls to the `EchoImpl` instance.

Receiving Responses

Some Mojom interface methods expect a response, such as `EchoInteger`. The corresponding JavaScript method returns a Promise. This Promise is resolved when the service side sends back a response. It is rejected if the interface is disconnected.

Connection Errors

If a pipe is disconnected, both endpoints will be able to observe the connection error (unless the disconnection is caused by closing/destroying an endpoint, in which case that endpoint won't get such a notification). If there are remaining incoming messages for an endpoint on disconnection, the connection error won't be triggered until the messages are drained.

Pipe disconnecition may be caused by:

- Mojo system-level causes: process terminated, resource exhausted, etc.
- The bindings close the pipe due to a validation error when processing a received message.
- The peer endpoint is closed. For example, the remote side is a bound interface pointer and it is destroyed.

Regardless of the underlying cause, when a connection error is encountered on a binding endpoint, that endpoint's **connection error handler** (if set) is invoked. This handler may only be invoked *once* as long as the endpoint is bound to the same pipe. Typically clients and implementations use this handler to do some kind of cleanup or recovery.

```
// Assume echoServicePtr is already bound.
echoServicePtr.ptr.setConnectionErrorHandler(function() {
  DoImportantCleanUp();
});

// Assume echoServiceBinding is already bound:
echoServiceBinding.setConnectionErrorHandler(function() {
  DoImportantCleanUpToo();
});
```

Note: Closing one end of a pipe will eventually trigger a connection error on the other end. However it's ordered with respect to any other event (*e.g.* writing a message) on the pipe. Therefore, it is safe to make an `echoInteger` call on `echoServicePtr` and reset it immediately (which results in disconnection), `echoServiceBinding` will receive the `echoInteger` call before it observes the connection error.

Associated Interfaces

An associated interface connection doesn't have its own underlying message pipe. It is associated with an existing message pipe (i.e., interface connection).

Similar to the non-associated interface case, we have:

- `mojo.AssociatedInterfacePtrInfo` and `mojo.AssociatedInterfaceRequest` encapsulate a *route ID*, representing a logical connection over a message pipe.

- For each Mojom interface `Foo`, there is a generated `FooAssociatedPtr` class. It owns an `AssociatedInterfacePtrInfo`. It is the client side of an interface.
- `mojo.AssociatedBinding` owns an `AssociatedInterfaceRequest`. It listens on the connection and dispatches incoming messages to a user-defined interface implementation.

See this document for more details.

Automatic and Manual Dependency Loading

By default, generated `.mojom.js` files automatically load Mojom dependencies. For example, if `foo.mojom` imports `bar.mojom`, loading `foo.mojom.js` will insert a `<script>` tag to load `bar.mojom.js`, if it hasn't been loaded.

The URL of `bar.mojom.js` is determined by:

- the path of `bar.mojom` relative to the position of `foo.mojom` at build time;
- the URL of `foo.mojom.js`.

For exmple, if at build time the two Mojom files are located at:

```
a/b/c/foo.mojom
a/b/d/bar.mojom
```

The URL of `foo.mojom.js` is:

```
http://example.org/scripts/b/c/foo.mojom.js
```

Then the URL of `bar.mojom.js` is supposed to be:

```
http://example.org/scripts/b/d/bar.mojom.js
```

If you would like `bar.mojom.js` to live at a different location, you need to set `mojo.config.autoLoadMojomDeps` to `false` before loading `foo.mojom.js`, and manually load `bar.mojom.js` yourself. Similarly, you need to turn off this option if you merge `bar.mojom.js` and `foo.mojom.js` into a single file.

```
<!-- Automatic dependency loading -->
<script src="http://example.org/scripts/mojo_bindings.js"></script>
<script src="http://example.org/scripts/b/c/foo.mojom.js"></script>


<!-- Manual dependency loading -->
<script src="http://example.org/scripts/mojo_bindings.js"></script>
<script>
  mojo.config.autoLoadMojomDeps = false;
</script>
```

```
<script src="http://example.org/scripts/b/d/bar.mojom.js"></script>
<script src="http://example.org/scripts/b/c/foo.mojom.js"></script>
```

Performance Tip: Avoid Loading the Same .mojom.js File Multiple Times

If `mojo.config.autoLoadMojomDeps` is set to `true` (which is the default value), you might accidentally load the same `.mojom.js` file multiple times if you are not careful. Although it doesn't cause fatal errors, it hurts performance and therefore should be avoided.

```
<!-- Assume that mojo.config.autoLoadMojomDeps is set to true: -->

<!-- No duplicate loading; recommended. -->
<script src="http://example.org/scripts/b/c/foo.mojom.js"></script>

<!-- No duplicate loading, although unnecessary. -->
<script src="http://example.org/scripts/b/d/bar.mojom.js"></script>
<script src="http://example.org/scripts/b/c/foo.mojom.js"></script>

<!-- Load bar.mojom.js twice; should be avoided. -->
<!-- when foo.mojom.js is loaded, it sees that bar.mojom.js is not yet loaded,
     so it inserts another <script> tag for bar.mojom.js. -->
<script src="http://example.org/scripts/b/c/foo.mojom.js"></script>
<script src="http://example.org/scripts/b/d/bar.mojom.js"></script>
```

If a `.mojom.js` file is loaded for a second time, a warnings will be showed using `console.warn()` to bring it to developers' attention.

Name Formatting

As a general rule, Mojom definitions follow the C++ formatting style. To make the generated JavaScript bindings conforms to our JavaScript style guide, the code generator does the following conversions:

In Mojom	In generated .mojom.js
MethodLikeThis	methodLikeThis
parameter_like_this	parameterLikeThis
field_like_this	fieldLikeThis
name_space.like_this	nameSpace.likeThis