Mojom IDL and Bindings Generator

This document is a subset of the Mojo documentation.

Contents

Overview

Mojom is the IDL for Mojo bindings interfaces. Given a `.mojom` file, the bindings generator outputs bindings for all supported languages: **C++**, **JavaScript**, and **Java**.

For a trivial example consider the following hypothetical Mojom file we write to `//services/widget/public/interfaces/frobinator.mojom`:

```
module widget.mojom;

interface Frobinator {
  Frobinate();
};
```

This defines a single interface named `Frobinator` in a module named `widget.mojom` (and thus fully qualified in Mojom as `widget.mojom.Frobinator`.) Note that many interfaces and/or other types of definitions may be included in a single Mojom file.

If we add a corresponding GN target to `//services/widget/public/interfaces/BUILD.gn`:

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

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

and then build this target:

```
ninja -C out/r services/widget/public/interfaces
```

we'll find several generated sources in our output directory:

```
out/r/gen/services/widget/public/interfaces/frobinator.mojom.cc
out/r/gen/services/widget/public/interfaces/frobinator.mojom.h
out/r/gen/services/widget/public/interfaces/frobinator.mojom.js
out/r/gen/services/widget/public/interfaces/frobinator.mojom.srcjar
...
```

Each of these generated source modules includes a set of definitions representing the Mojom contents within the target language. For more details regarding the generated outputs please see documentation for individual target languages.

Mojom Syntax

Mojom IDL allows developers to define **structs**, **unions**, **interfaces**, **constants**, and **enums**, all within the context of a **module**. These definitions are used to generate code in the supported target languages at build time.

Mojom files may **import** other Mojom files in order to reference their definitions.

Primitive Types

Mojom supports a few basic data types which may be composed into structs or used for message parameters.

Type	Description
`bool`	Boolean type (`true` or `false`.)
`int8`, `uint8`	Signed or unsigned 8-bit integer.
`int16`, `uint16`	Signed or unsigned 16-bit integer.
`int32`, `uint32`	Signed or unsigned 32-bit integer.
`int64`, `uint64`	Signed or unsigned 64-bit integer.
`float`, `double`	32- or 64-bit floating point number.
`string`	UTF-8 encoded string.
`array<T>`	Array of any Mojom type *T*; for example, `array<uint8>` or `array<array<string>>`.
`array<T, N>`	Fixed-length array of any Mojom type *T*. The parameter *N* must be an integral constant.
`map<S, T>`	Associated array maping values of type *S* to values of type *T*. *S* may be a `string`, `enum`, or numeric type.
`handle`	Generic Mojo handle. May be any type of handle, including a wrapped native platform handle.
`handle<message_pipe>`	Generic message pipe handle.
`handle<shared_buffer>`	Shared buffer handle.
`handle<data_pipe_producer>`	Data pipe producer handle.
`handle<data_pipe_consumer>`	Data pipe consumer handle.
`InterfaceType`	Any user-defined Mojom interface type. This is sugar for a strongly-typed message pipe handle which should eventually be used to make outgoing calls on the interface.
`InterfaceType&`	An interface request for any user-defined Mojom interface type. This is sugar for a more strongly-typed message pipe handle which is expected to receive request messages and should therefore eventually be bound to an implementation of the interface.
`associated InterfaceType`	An associated interface handle. See Associated Interfaces
`associated InterfaceType&`	An associated interface request. See Associated Interfaces
T?	An optional (nullable) value. Primitive numeric types (integers, floats, booleans, and enums) are not nullable. All other types are nullable.

Modules

Every Mojom file may optionally specify a single **module** to which it belongs.

This is used strictly for aggregaging all defined symbols therein within a common Mojom namespace. The specific impact this has on generated binidngs code varies for each target language. For example, if the following Mojom is used to generate bindings:

```
module business.stuff;

interface MoneyGenerator {
  GenerateMoney();
};
```

Generated C++ bindings will define a class interface `MoneyGenerator` in the `business::stuff` namespace, while Java bindings will define an interface `MoneyGenerator` in the `org.chromium.business.stuff` package. JavaScript bindings at this time are unaffected by module declarations.

NOTE: By convention in the Chromium codebase, **all** Mojom files should declare a module name with at least (and preferrably exactly) one top-level name as well as an inner `mojom` module suffix. *e.g.*, `chrome.mojom`, `business.mojom`, *etc.*

This convention makes it easy to tell which symbols are generated by Mojom when reading non-Mojom code, and it also avoids namespace collisions in the fairly common scenario where you have a real C++ or Java `Foo` along with a corresponding Mojom `Foo` for its serialized representation.

Imports

If your Mojom references definitions from other Mojom files, you must **import** those files. Import syntax is as follows:

```
import "services/widget/public/interfaces/frobinator.mojom";
```

Import paths are always relative to the top-level directory.

Note that circular imports are **not** supported.

Structs

Structs are defined using the **struct** keyword, and they provide a way to group related fields together:

```
struct StringPair {
  string first;
  string second;
};
```

Struct fields may be comprised of any of the types listed above in the Primitive Types section.

Default values may be specified as long as they are constant:

```
struct Request {
  int32 id = -1;
  string details;
};
```

What follows is a fairly comprehensive example using the supported field types:

```
struct StringPair {
  string first;
  string second;
};

enum AnEnum {
  YES,
  NO
};

interface SampleInterface {
  DoStuff();
};

struct AllTheThings {
  // Note that these types can never be marked nullable!
  bool boolean_value;
  int8 signed_8bit_value = 42;
  uint8 unsigned_8bit_value;
  int16 signed_16bit_value;
  uint16 unsigned_16bit_value;
  int32 signed_32bit_value;
  uint32 unsigned_32bit_value;
  int64 signed_64bit_value;
  uint64 unsigned_64bit_value;
  float float_value_32bit;
  double float_value_64bit;
  AnEnum enum_value = AnEnum.YES;

  // Strings may be nullable.
  string? maybe_a_string_maybe_not;

  // Structs may contain other structs. These may also be nullable.
  StringPair some_strings;
  StringPair? maybe_some_more_strings;

  // In fact structs can also be nested, though in practice you must always make
  // such fields nullable -- otherwise messages would need to be infinitely long
  // in order to pass validation!
  AllTheThings? more_things;

  // Arrays may be templated over any Mojom type, and are always nullable:
  array<int32> numbers;
  array<int32>? maybe_more_numbers;

  // Arrays of arrays of arrays... are fine.
  array<array<array<AnEnum>>> this_works_but_really_plz_stop;

  // The element type may be nullable if it's a type which is allowed to be
  // nullable.
  array<AllTheThings?> more_maybe_things;

  // Fixed-size arrays get some extra validation on the receiving end to ensure
  // that the correct number of elements is always received.
  array<uint64, 2> uuid;

  // Maps follow many of the same rules as arrays. Key types may be any
  // non-handle, non-collection type, and value types may be any supported
  // struct field type. Maps may also be nullable.
  map<string, int32> one_map;
  map<AnEnum, string>? maybe_another_map;
  map<StringPair, AllTheThings?>? maybe_a_pretty_weird_but_valid_map;
  map<StringPair, map<int32, array<map<string, string>?>?>?> ridiculous;

  // And finally, all handle types are valid as struct fields and may be
  // nullable. Note that interfaces and interface requests (the "Foo" and
  // "Foo&" type syntax respectively) are just strongly-typed message pipe
  // handles.
  handle generic_handle;
  handle<data_pipe_consumer> reader;
  handle<data_pipe_producer>? maybe_writer;
  handle<shared_buffer> dumping_ground;
  handle<message_pipe> raw_message_pipe;
  SampleInterface? maybe_a_sample_interface_client_pipe;
  SampleInterface& non_nullable_sample_interface_request;
  SampleInterface&? nullable_sample_interface_request;
  associated SampleInterface associated_interface_client;
  associated SampleInterface& associated_interface_request;
  associated SampleInterface&? maybe_another_associated_request;
};
```

For details on how all of these different types translate to usable generated code, see documentation for individual target languages.

Unions

Mojom supports tagged unions using the **union** keyword. A union is a collection of fields which may taken the value of any single one of those fields at a time. Thus they provide a way to represent a variant value type while minimizing storage requirements.

Union fields may be of any type supported by struct fields. For example:

```
union ExampleUnion {
  string str;
  StringPair pair;
  int64 id;
  array<uint64, 2> guid;
  SampleInterface iface;
};
```

For details on how unions like this translate to generated bindings code, see documentation for individual target languages.

Enumeration Types

Enumeration types may be defined using the **enum** keyword either directly within a module or nested within the namespace of some struct or interface:

```
module business.mojom;

enum Department {
  SALES = 0,
  DEV,
};

struct Employee {
  enum Type {
    FULL_TIME,
    PART_TIME,
  };

  Type type;
  // ...
};
```

Similar to C-style enums, individual values may be explicitly assigned within an enum definition. By default, values are based at zero and increment by 1 sequentially.

The effect of nested definitions on generated bindings varies depending on the target language. See documentation for individual target languages

Constants

Constants may be defined using the **const** keyword either directly within a module or nested within the namespace of some struct or interface:

```
module business.mojom;

const string kServiceName = "business";

struct Employee {
  const uint64 kInvalidId = 0;

  enum Type {
    FULL_TIME,
    PART_TIME,
  };

  uint64 id = kInvalidId;
  Type type;
};
```

The effect of nested definitions on generated bindings varies depending on the target language. See documentation for individual target languages

Interfaces

An **interface** is a logical bundle of parameterized request messages. Each request message may optionally define a parameterized response message. Here's an example to define an interface `Foo` with various kinds of requests:

```
interface Foo {
  // A request which takes no arguments and expects no response.
  MyMessage();

  // A request which has some arguments and expects no response.
  MyOtherMessage(string name, array<uint8> bytes);

  // A request which expects a single-argument response.
  MyMessageWithResponse(string command) => (bool success);

  // A request which expects a response with multiple arguments.
  MyMessageWithMoarResponse(string a, string b) => (int8 c, int8 d);
};
```

Anything which is a valid struct field type (see Structs) is also a valid request or response argument type. The type notation is the same for both.

Attributes

Mojom definitions may have their meaning altered by **attributes**, specified with a syntax similar to Java or C# attributes. There are a handle of interesting attributes supported today.

`[Sync]` : The `Sync` attribute may be specified for any interface method which expects a response. This makes it so that callers of the method can wait synchronously for a response. See Synchronous Calls in the C++ bindings documentation. Note that sync calls are not currently supported in other target languages.

`[Extensible]` : The `Extensible` attribute may be specified for any enum definition. This essentially disables builtin range validation when receiving values of the enum type in a message, allowing older bindings to tolerate unrecognized values from newer versions of the enum.

`[Native]` : The `Native` attribute may be specified for an empty struct declaration to provide a nominal bridge between Mojo IPC and legacy `IPC::ParamTraits` or `IPC_STRUCT_TRAITS*` macros. See Using Legacy IPC Traits for more details. Note support for this attribute is strictly limited to C++ bindings generation.

`[MinVersion=N]` : The `MinVersion` attribute is used to specify the version at which a given field, enum value, interface method, or method parameter was introduced. See Versioning for more details.

`[EnableIf=value]` : The `EnableIf` attribute is used to conditionally enable definitions when the mojom is parsed. If the `mojom` target in the GN file does not include the matching `value` in the list of `enabled_features`, the definition will be disabled. This is useful for mojom definitions that only make sense on one platform. Note that the `EnableIf` attribute can only be set once per definition.

Generated Code For Target Languages

When the bindings generator successfully processes an input Mojom file, it emits corresponding code for each supported target language. For more details on how Mojom concepts translate to a given target langauge, please refer to the bindings API documentation for that language:

- C++ Bindings
- JavaScript Bindings
- Java Bindings

Message Validation

Regardless of target language, all interface messages are validated during deserialization before they are dispatched to a receiving implementation of the interface. This helps to ensure consitent validation across interfaces without leaving the burden to developers and security reviewers every time a new message is added.

If a message fails validation, it is never dispatched. Instead a **connection error** is raised on the binding object (see C++ Connection Errors, Java Connection Errors, or JavaScript Connection Errors for details.)

Some baseline level of validation is done automatically for primitive Mojom types.

Non-Nullable Objects

Mojom fields or parameter values (*e.g.*, structs, interfaces, arrays, *etc.*) may be marked nullable in Mojom definitions (see Primitive Types.) If a field or parameter is **not** marked nullable but a message is received with a null value in its place, that message will fail validation.

Enums

Enums declared in Mojom are automatically validated against the range of legal values. For example if a Mojom declares the enum:

```
enum AdvancedBoolean {
  TRUE = 0,
  FALSE = 1,
  FILE_NOT_FOUND = 2,
};
```

and a message is received with the integral value 3 (or anything other than 0, 1, or 2) in place of some `AdvancedBoolean` field or parameter, the message will fail validation.

NOTE: It's possible to avoid this type of validation error by explicitly marking an enum as Extensible if you anticipate your enum being exchanged between two different versions of the binding interface. See Versioning.

Other failures

There are a host of internal validation errors that may occur when a malformed message is received, but developers should not be concerned with these specifically; in general they can only result from internal bindings bugs, compromised processes, or some remote endpoint making a dubious effort to manually encode their own bindings messages.

Custom Validation

It's also possible for developers to define custom validation logic for specific Mojom struct types by exploiting the type mapping system for C++ bindings. Messages rejected by custom validation logic trigger the same validation failure behavior as the built-in type validation routines.

Associated Interfaces

As mentioned in the Primitive Types section above, interface and interface request fields and parameters may be marked as `associated`. This essentially means that they are piggy-backed on some other interface's message pipe.

Because individual interface message pipes operate independently there can be no relative ordering guarantees among them. Associated interfaces are useful when one interface needs to guarantee strict FIFO ordering with respect to one or more other interfaces, as they allow interfaces to share a single pipe.

Currenly associated interfaces are only supported in generated C++ bindings. See the documentation for C++ Associated Interfaces.

Versioning

Overview

NOTE: You don't need to worry about versioning if you don't care about backwards compatibility. Specifically, all parts of Chrome are updated atomically today and there is not yet any possibility of any two Chrome processes communicating with two different versions of any given Mojom interface.

Services extend their interfaces to support new features over time, and clients want to use those new features when they are available. If services and clients are not updated at the same time, it's important for them to be able to communicate with each other using different snapshots (versions) of their interfaces.

This document shows how to extend Mojom interfaces in a backwards-compatible way. Changing interfaces in a non-backwards-compatible way is not discussed, because in that case communication between different interface versions is impossible anyway.

Versioned Structs

You can use the `MinVersion` attribute to indicate from which version a struct field is introduced. Assume you have the following struct:

```
struct Employee {
  uint64 employee_id;
  string name;
};
```

and you would like to add a birthday field. You can do:

```
struct Employee {
  uint64 employee_id;
  string name;
  [MinVersion=1] Date? birthday;
};
```

By default, fields belong to version 0. New fields must be appended to the struct definition (*i.e.*, existing fields must not change **ordinal value**) with the `MinVersion` attribute set to a number greater than any previous existing versions.

Ordinal value refers to the relative positional layout of a struct's fields (and an interface's methods) when encoded in a message. Implicitly, ordinal numbers are assigned to fields according to lexical position. In the example above, `employee_id` has an ordinal value of 0 and `name` has an ordinal value of 1.

Ordinal values can be specified explicitly using `**@**` notation, subject to the following hard constraints:

- For any given struct or interface, if any field or method explicitly specifies an ordinal value, all fields or methods must explicitly specify an ordinal value.
- For an *N*-field struct or *N*-method interface, the set of explicitly assigned ordinal values must be limited to the range *[0, N-1]*.

You may reorder fields, but you must ensure that the ordinal values of existing fields remain unchanged. For example, the following struct remains backwards-compatible:

```
struct Employee {
  uint64 employee_id@0;
  [MinVersion=1] Date? birthday@2;
  string name@1;
};
```

NOTE: Newly added fields of Mojo object or handle types MUST be nullable. See Primitive Types.

Versioned Interfaces

There are two dimensions on which an interface can be extended

Appending New Parameters To Existing Methods : Parameter lists are treated as structs internally, so all the rules of versioned structs apply to method parameter lists. The only difference is that the version number is scoped to the whole interface rather than to any individual parameter list.

```
Please note that adding a response to a message which did not previously
expect a response is a not a backwards-compatible change.
```

Appending New Methods : Similarly, you can reorder methods with explicit ordinal values as long as the ordinal values of existing methods are unchanged.

For example:

```
// Old version:
interface HumanResourceDatabase {
  AddEmployee(Employee employee) => (bool success);
  QueryEmployee(uint64 id) => (Employee? employee);
};

// New version:
interface HumanResourceDatabase {
  AddEmployee(Employee employee) => (bool success);

  QueryEmployee(uint64 id, [MinVersion=1] bool retrieve_finger_print)
      => (Employee? employee,
          [MinVersion=1] array<uint8>? finger_print);

  [MinVersion=1]
  AttachFingerPrint(uint64 id, array<uint8> finger_print)
      => (bool success);
};
```

Similar to versioned structs, when you pass the parameter list of a request or response method to a destination using an older version of an interface, unrecognized fields are silently discarded. However, if the method call itself is not recognized, it is considered a validation error and the receiver will close its end of the interface pipe. For example, if a client on version 1 of the above interface sends an `AttachFingerPrint` request to an implementation of version 0, the client will be disconnected.

Bindings target languages that support versioning expose means to query or assert the remote version from a client handle (*e.g.*, an `InterfacePtr<T>` in C++ bindings.)

See C++ Versioning Considerations and Java Versioning Considerations

Versioned Enums

By default, enums are non-extensible, which means that generated message validation code does not expect to see new values in the future. When an unknown value is seen for a non-extensible enum field or parameter, a validation error is raised.

If you want an enum to be extensible in the future, you can apply the `[Extensible]` attribute:

```
[Extensible]
enum Department {
  SALES,
  DEV,
};
```

And later you can extend this enum without breaking backwards compatibility:

```
[Extensible]
enum Department {
  SALES,
  DEV,
  [MinVersion=1] RESEARCH,
};
```

NOTE: For versioned enum definitions, the use of a `[MinVersion]` attribute is strictly for documentation purposes. It has no impact on the generated code.

With extensible enums, bound interface implementations may receive unknown enum values and will need to deal with them gracefully. See C++ Versioning Considerations for details.

Grammar Reference

Below is the (BNF-ish) context-free grammar of the Mojom language:

```
MojomFile = StatementList
StatementList = Statement StatementList | Statement
Statement = ModuleStatement | ImportStatement | Definition

ModuleStatement = AttributeSection "module" Identifier ";"
ImportStatement = "import" StringLiteral ";"
Definition = Struct Union Interface Enum Const

AttributeSection = "[" AttributeList "]"
AttributeList = <empty> | NonEmptyAttributeList
NonEmptyAttributeList = Attribute
                      | Attribute "," NonEmptyAttributeList
Attribute = Name
          | Name "=" Name
          | Name "=" Literal

Struct = AttributeSection "struct" Name "{" StructBody "}" ";"
       | AttributeSection "struct" Name ";"
StructBody = <empty>
           | StructBody Const
           | StructBody Enum
           | StructBody StructField
StructField = AttributeSection TypeSpec Name Orginal Default ";"

Union = AttributeSection "union" Name "{" UnionBody "}" ";"
UnionBody = <empty> | UnionBody UnionField
UnionField = AttributeSection TypeSpec Name Ordinal ";"

Interface = AttributeSection "interface" Name "{" InterfaceBody "}" ";"
InterfaceBody = <empty>
              | InterfaceBody Const
              | InterfaceBody Enum
              | InterfaceBody Method
Method = AttributeSection Name Ordinal "(" ParamterList ")" Response ";"
ParameterList = <empty> | NonEmptyParameterList
NonEmptyParameterList = Parameter
                      | Parameter "," NonEmptyParameterList
Parameter = AttributeSection TypeSpec Name Ordinal
Response = <empty> | "=>" "(" ParameterList ")"

TypeSpec = TypeName "?" | TypeName
TypeName = BasicTypeName
         | Array
         | FixedArray
         | Map
         | InterfaceRequest
BasicTypeName = Identifier | "associated" Identifier | HandleType | NumericType
NumericType = "bool" | "int8" | "uint8" | "int16" | "uint16" | "int32"
```

```
            | "uint32" | "int64" | "uint64" | "float" | "double"
HandleType = "handle" | "handle" "<" SpecificHandleType ">"
SpecificHandleType = "message_pipe"
                   | "shared_buffer"
                   | "data_pipe_consumer"
                   | "data_pipe_producer"
Array = "array" "<" TypeSpec ">"
FixedArray = "array" "<" TypeSpec "," IntConstDec ">"
Map = "map" "<" Identifier "," TypeSpec ">"
InterfaceRequest = Identifier "&" | "associated" Identifier "&"

Ordinal = <empty> | OrdinalValue

Default = <empty> | "=" Constant

Enum = AttributeSection "enum" Name "{" NonEmptyEnumValueList "}" ";"
     | AttributeSection "enum" Name "{" NonEmptyEnumValueList "," "}" ";"
NonEmptyEnumValueList = EnumValue | NonEmptyEnumValueList "," EnumValue
EnumValue = AttributeSection Name
          | AttributeSection Name "=" Integer
          | AttributeSection Name "=" Identifier

Const = "const" TypeSpec Name "=" Constant ";"

Constant = Literal | Identifier ";"

Identifier = Name | Name "." Identifier

Literal = Integer | Float | "true" | "false" | "default" | StringLiteral

Integer = IntConst | "+" IntConst | "-" IntConst
IntConst = IntConstDec | IntConstHex

Float = FloatConst | "+" FloatConst | "-" FloatConst

; The rules below are for tokens matched strictly according to the given regexes

Identifier = /[a-zA-Z_][0-9a-zA-Z_]*/
IntConstDec = /0|(1-9[0-9]*)/
IntConstHex = /0[xX][0-9a-fA-F]+/
OrdinalValue = /@(0|(1-9[0-9]*))/
FloatConst = ... # Imagine it's close enough to C-style float syntax.
StringLiteral = ... # Imagine it's close enough to C-style string literals, including
```

Additional Documentation

Mojom Message Format : Describes the wire format used by Mojo bindings interfaces over message pipes.

Input Format of Mojom Message Validation Tests : Describes a text format used to facilitate bindings message validation tests.