Mojo C++ Bindings API

This document is a subset of the Mojo documentation.

Contents

Overview

The Mojo C++ Bindings API leverages the C++ System API to provide a more natural set of primitives for communicating over Mojo message pipes. Combined with generated code from the Mojom IDL and bindings generator, users can easily connect interface clients and implementations across arbitrary intra- and inter-process bounaries.

This document provides a detailed guide to bindings API usage with example code snippets. For a detailed API references please consult the headers in //mojo/public/cpp/bindings.

For a simplified guide targeted at Chromium developers, see this link.

Getting Started

When a Mojom IDL file is processed by the bindings generator, C++ code is emitted in a series of `.h` and `.cc` files with names based on the input `.mojom` file. Suppose we create the following Mojom file at `//services/db/public/mojom/db.mojom`:

```
module db.mojom;

interface Table {
  AddRow(int32 key, string data);
};

interface Database {
  CreateTable(Table& table);
};
```

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

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

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

Ensure that any target that needs this interface depends on it, e.g. with a line like:

```
   deps += [ '//services/db/public/mojom' ]
```

If we then build this target:

```
ninja -C out/r services/db/public/mojom
```

This will produce several generated source files, some of which are relevant to C++ bindings. Two of these files are:

```
out/gen/services/db/public/mojom/db.mojom.cc
out/gen/services/db/public/mojom/db.mojom.h
```

You can include the above generated header in your sources in order to use the definitions therein:

```cpp
#include "services/business/public/mojom/factory.mojom.h"

class TableImpl : public db::mojom::Table {
  // ...
};
```

This document covers the different kinds of definitions generated by Mojom IDL for C++ consumers and how they can effectively be used to communicate across message pipes.

NOTE: Using C++ bindings from within Blink code is typically subject to special constraints which require the use of a different generated header. For details, see Blink Type Mapping.

Interfaces

Mojom IDL interfaces are translated to corresponding C++ (pure virtual) class interface definitions in the generated header, consisting of a single generated method signature for each request message on the interface. Internally there is also generated code for serialization and deserialization of messages, but this detail is hidden from bindings consumers.

Basic Usage

Let's consider a new `//sample/logger.mojom` to define a simple logging interface which clients can use to log simple string messages:

```
module sample.mojom;

interface Logger {
  Log(string message);
};
```

Running this through the bindings generator will produce a `logging.mojom.h` with the following definitions (modulo unimportant details):

```
namespace sample {
namespace mojom {

class Logger {
  virtual ~Logger() {}

  virtual void Log(const std::string& message) = 0;
};

using LoggerPtr = mojo::InterfacePtr<Logger>;
using LoggerRequest = mojo::InterfaceRequest<Logger>;

}  // namespace mojom
}  // namespace sample
```

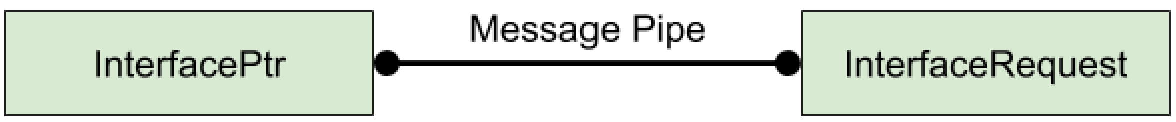
Makes sense. Let's take a closer look at those type aliases at the end.

InterfacePtr and InterfaceRequest

You will notice the type aliases for `LoggerPtr` and `LoggerRequest` are using two of the most fundamental template types in the C++ bindings library: **`InterfacePtr<T>`** and **`InterfaceRequest<T>`**.

In the world of Mojo bindings libraries these are effectively strongly-typed message pipe endpoints. If an `InterfacePtr<T>` is bound to a message pipe endpoint, it can be dereferenced to make calls on an opaque `T` interface. These calls immediately serialize their arguments (using generated code) and write a corresponding message to the pipe.

An `InterfaceRequest<T>` is essentially just a typed container to hold the other end of an `InterfacePtr<T>`'s pipe -- the receiving end -- until it can be routed to some implementation which will **bind** it. The `InterfaceRequest<T>` doesn't actually *do* anything other than hold onto a pipe endpoint and carry useful compile-time type information.



So how do we create a strongly-typed message pipe?

Creating Interface Pipes

One way to do this is by manually creating a pipe and wrapping each end with a strongly-typed object:

```
#include "sample/logger.mojom.h"

mojo::MessagePipe pipe;
sample::mojom::LoggerPtr logger(
    sample::mojom::LoggerPtrInfo(std::move(pipe.handle0), 0));
sample::mojom::LoggerRequest request(std::move(pipe.handle1));
```

That's pretty verbose, but the C++ Bindings library provides a more convenient way to accomplish the same thing. interface_request.h defines a `MakeRequest` function:

```
sample::mojom::LoggerPtr logger;
auto request = mojo::MakeRequest(&logger);
```

This second snippet is equivalent to the first one.

NOTE: In the first example above you may notice usage of the `LoggerPtrInfo` type, which is a generated alias for `mojo::InterfacePtrInfo<Logger>`. This is similar to an `InterfaceRequest<T>` in that it merely holds onto a pipe handle and cannot actually read or write messages on the pipe. Both this type and `InterfaceRequest<T>` are safe to move freely from sequence to sequence, whereas a bound `InterfacePtr<T>` is bound to a single sequence.

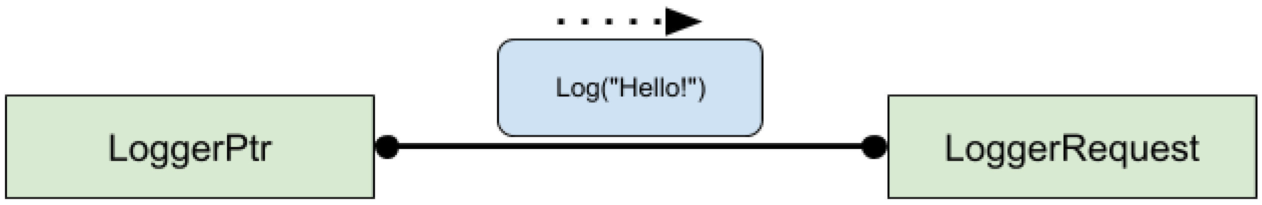
An `InterfacePtr<T>` may be unbound by calling its `PassInterface()` method, which returns a new `InterfacePtrInfo<T>`. Conversely, an `InterfacePtr<T>` may bind (and thus take ownership of) an `InterfacePtrInfo<T>` so that interface calls can be made on the pipe.

The sequence-bound nature of `InterfacePtr<T>` is necessary to support safe dispatch of its message responses and connection error notifications.

Once the `LoggerPtr` is bound we can immediately begin calling `Logger` interface methods on it, which will immediately write messages into the pipe. These messages will stay queued on the receiving end of the pipe until someone binds to it and starts reading them.

```
logger->Log("Hello!");
```

This actually writes a `Log` message to the pipe.



But as mentioned above, `InterfaceRequest` *doesn't actually do anything*, so that message will just sit on the pipe forever. We need a way to read messages off the other end of the pipe and dispatch them. We have to **bind the interface request**.

Binding an Interface Request

There are many different helper classes in the bindings library for binding the receiving end of a message pipe. The most primitive among them is the aptly named `mojo::Binding<T>`. A `mojo::Binding<T>` bridges an implementation of `T` with a single bound message pipe endpoint (via a `mojo::InterfaceRequest<T>`), which it continuously watches for readability.

Any time the bound pipe becomes readable, the `Binding` will schedule a task to read, deserialize (using generated code), and dispatch all available messages to the bound `T` implementation. Below is a sample implementation of the `Logger` interface. Notice that the implementation itself owns a `mojo::Binding`. This is a common pattern, since a bound implementation must outlive any `mojo::Binding` which binds it.

```
#include "base/logging.h"
#include "base/macros.h"
#include "sample/logger.mojom.h"

class LoggerImpl : public sample::mojom::Logger {
 public:
  // NOTE: A common pattern for interface implementations which have one
  // instance per client is to take an InterfaceRequest in the constructor.

  explicit LoggerImpl(sample::mojom::LoggerRequest request)
      : binding_(this, std::move(request)) {}
  ~Logger() override {}

  // sample::mojom::Logger:
  void Log(const std::string& message) override {
    LOG(ERROR) << "[Logger] " << message;
  }

 private:
  mojo::Binding<sample::mojom::Logger> binding_;

  DISALLOW_COPY_AND_ASSIGN(LoggerImpl);
};
```

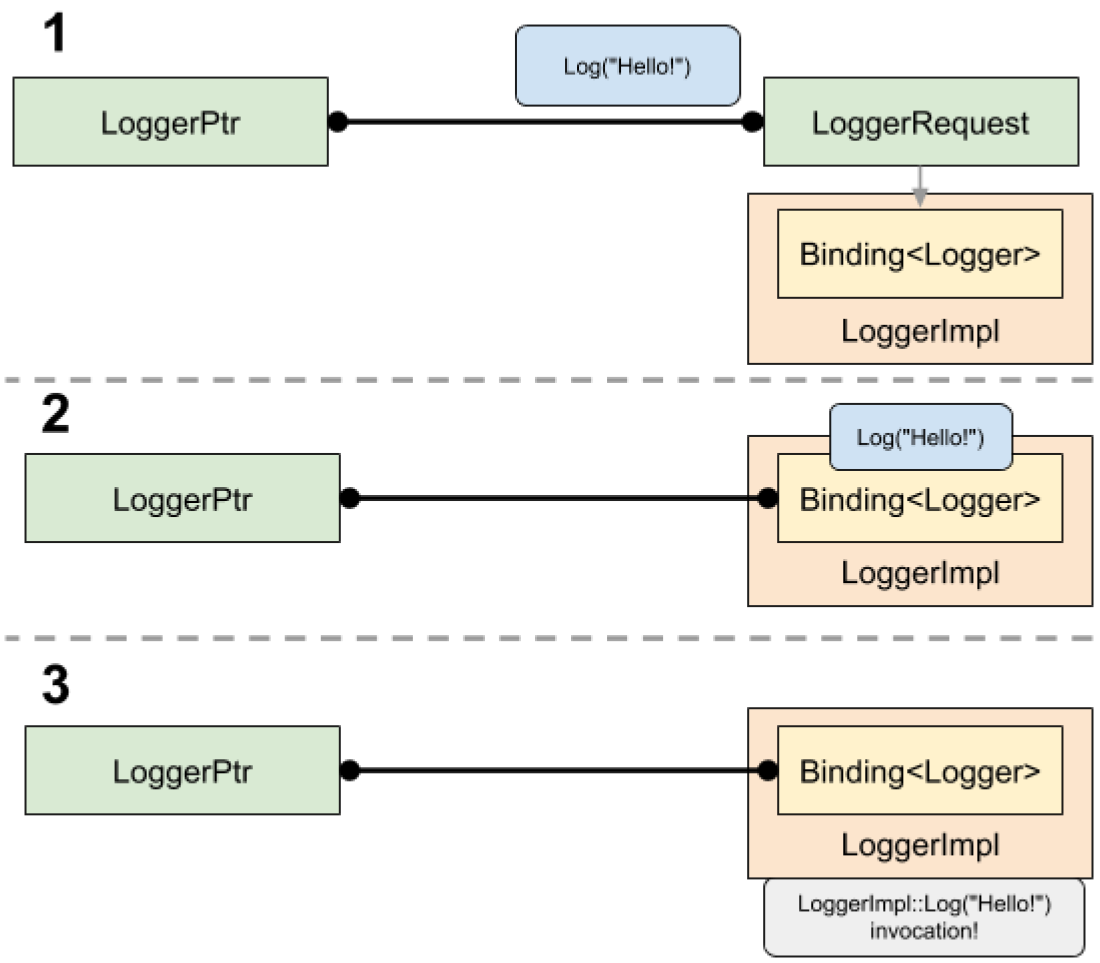
Now we can construct a `LoggerImpl` over our pending `LoggerRequest`, and the previously queued `Log` message will be dispatched ASAP on the `LoggerImpl`'s sequence:

```
LoggerImpl impl(std::move(request));
```

The diagram below illustrates the following sequence of events, all set in motion by the above line of code:

1. The `LoggerImpl` constructor is called, passing the `LoggerRequest` along to the `Binding`.
2. The `Binding` takes ownership of the `LoggerRequest`'s pipe endpoint and begins watching it for readability. The pipe is readable immediately, so a task is scheduled to read the pending `Log` message from the pipe ASAP.

3. The `Log` message is read and deserialized, causing the `Binding` to invoke the `Logger::Log` implementation on its bound `LoggerImpl`.



As a result, our implementation will eventually log the client's `"Hello!"` message via `LOG(ERROR)`.

NOTE: Messages will only be read and dispatched from a pipe as long as the object which binds it (*i.e.* the `mojo::Binding` in the above example) remains alive.

Receiving Responses

Some Mojom interface methods expect a response. Suppose we modify our `Logger` interface so that the last logged line can be queried like so:

```
module sample.mojom;

interface Logger {
  Log(string message);
  GetTail() => (string message);
};
```

The generated C++ interface will now look like:

```
namespace sample {
namespace mojom {

class Logger {
 public:
  virtual ~Logger() {}

  virtual void Log(const std::string& message) = 0;

  using GetTailCallback = base::OnceCallback<void(const std::string& message)>;

  virtual void GetTail(GetTailCallback callback) = 0;
}

}  // namespace mojom
}  // namespace sample
```

As before, both clients and implementations of this interface use the same signature for the `GetTail` method: implementations use the `callback` argument to *respond* to the request, while clients pass a `callback` argument to asynchronously `receive` the response. Here's an updated implementation:

```
class LoggerImpl : public sample::mojom::Logger {
 public:
  // NOTE: A common pattern for interface implementations which have one
  // instance per client is to take an InterfaceRequest in the constructor.

  explicit LoggerImpl(sample::mojom::LoggerRequest request)
      : binding_(this, std::move(request)) {}
  ~Logger() override {}

  // sample::mojom::Logger:
  void Log(const std::string& message) override {
    LOG(ERROR) << "[Logger] " << message;
    lines_.push_back(message);
  }

  void GetTail(GetTailCallback callback) override {
    std::move(callback).Run(lines_.back());
  }

 private:
  mojo::Binding<sample::mojom::Logger> binding_;
  std::vector<std::string> lines_;

  DISALLOW_COPY_AND_ASSIGN(LoggerImpl);
};
```

And an updated client call:

```
void OnGetTail(const std::string& message) {
  LOG(ERROR) << "Tail was: " << message;
}

logger->GetTail(base::BindOnce(&OnGetTail));
```

Behind the scenes, the implementation-side callback is actually serializing the response arguments and writing them onto the pipe for delivery back to the client. Meanwhile the client-side callback is invoked by some internal logic which watches the pipe for an incoming response message, reads and deserializes it once it arrives, and then invokes the callback with the deserialized parameters.

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 `mojo::InterfacePtr<T>` 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 is a simple `base::Closure` and 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 -- particularly if the error was unexpected -- create a new pipe and attempt to establish a new connection with it.

All message pipe-binding C++ objects (*e.g.*, `mojo::Binding<T>`, `mojo::InterfacePtr<T>`, *etc.*) support setting their connection error handler via a `set_connection_error_handler` method.

We can set up another end-to-end `Logger` example to demonstrate error handler invocation:

```
sample::mojom::LoggerPtr logger;
LoggerImpl impl(mojo::MakeRequest(&logger));
impl.set_connection_error_handler(base::BindOnce([] { LOG(ERROR) << "Bye."; }));
logger->Log("OK cool");
logger.reset();  // Closes the client end.
```

As long as `impl` stays alive here, it will eventually receive the `Log` message followed immediately by an invocation of the bound callback which outputs `"Bye."`. Like all other bindings callbacks, a connection error handler will **never** be invoked once its corresponding binding object has been destroyed.

In fact, suppose instead that `LoggerImpl` had set up the following error handler within its constructor:

```
LoggerImpl::LoggerImpl(sample::mojom::LoggerRequest request)
    : binding_(this, std::move(request)) {
  binding_.set_connection_error_handler(
      base::BindOnce(&LoggerImpl::OnError, base::Unretained(this)));
}

void LoggerImpl::OnError() {
  LOG(ERROR) << "Client disconnected! Purging log lines.";
  lines_.clear();
}
```

The use of `base::Unretained` is *safe* because the error handler will never be invoked beyond the lifetime of `binding_`, and `this` owns `binding_`.

A Note About Endpoint Lifetime and Callbacks

Once a `mojo::InterfacePtr<T>` is destroyed, it is guaranteed that pending callbacks as well as the connection error handler (if registered) won't be called.

Once a `mojo::Binding<T>` is destroyed, it is guaranteed that no more method calls are dispatched to the implementation and the connection error handler (if registered) won't be called.

Best practices for dealing with process crashes and callbacks

A common situation when calling mojo interface methods that take a callback is that the caller wants to know if the other endpoint is torn down (e.g. because of a crash). In that case, the consumer usually wants to know if the response callback won't be run. There are different solutions for this problem, depending on how the `InterfacePtr<T>` is held:

1. The consumer owns the `InterfacePtr<T>`: `set_connection_error_handler` should be used.
2. The consumer doesn't own the `InterfacePtr<T>`: there are two helpers depending on the behavior that the caller wants. If the caller wants to ensure that an error handler is run, then `mojo::WrapCallbackWithDropHandler` should be used. If the caller wants the callback to always be run, then `mojo::WrapCallbackWithDefaultInvokeIfNotRun` helper should be used. With both of these helpers, usual callback care should be followed to ensure that the callbacks don't run after the consumer is destructed (e.g. because the owner of the `InterfacePtr<T>` outlives the consumer). This includes using `base::WeakPtr` or `base::RefCounted`. It should also be noted that with these helpers, the callbacks could be run synchronously while the InterfacePtr is reset or destroyed.

A Note About Ordering

As mentioned in the previous section, closing one end of a pipe will eventually trigger a connection error on the other end. However it's important to note that this event is itself ordered with respect to any other event (*e.g.* writing a message) on the pipe.

This means that it's safe to write something contrived like:

```
void GoBindALogger(sample::mojom::LoggerRequest request) {
  LoggerImpl impl(std::move(request));
  base::RunLoop loop;
  impl.set_connection_error_handler(loop.QuitClosure());
  loop.Run();
}

void LogSomething() {
  sample::mojom::LoggerPtr logger;
  bg_thread->task_runner()->PostTask(
      FROM_HERE, base::BindOnce(&GoBindALogger, mojo::MakeRequest(&logger)));
  logger->Log("OK Computer");
}
```

When `logger` goes out of scope it immediately closes its end of the message pipe, but the impl-side won't notice this until it receives the sent `Log` message. Thus the `impl` above will first log our message and *then* see a connection error and break out of the run loop.

Types

Enums

Mojom enums translate directly to equivalent strongly-typed C++11 enum classes with `int32_t` as the underlying type. The typename and value names are identical between Mojom and C++. Mojo also always defines a special enumerator `kMaxValue` that shares the value of the highest enumerator: this makes it easy to record Mojo enums in histograms and interoperate with legacy IPC.

For example, consider the following Mojom definition:

```
module business.mojom;

enum Department {
  kEngineering,
  kMarketing,
  kSales,
};
```

This translates to the following C++ definition:

```
namespace business {
namespace mojom {

enum class Department : int32_t {
  kEngineering,
  kMarketing,
  kSales,
  kMaxValue = kSales,
};

}  // namespace mojom
}  // namespace business
```

Structs

Mojom structs can be used to define logical groupings of fields into a new composite type. Every Mojom struct elicits the generation of an identically named, representative C++ class, with identically named public fields of corresponding C++ types, and several helpful public methods.

For example, consider the following Mojom struct:

```
module business.mojom;

struct Employee {
  int64 id;
  string username;
  Department department;
};
```

This would generate a C++ class like so:

```
namespace business {
namespace mojom {

class Employee;

using EmployeePtr = mojo::StructPtr<Employee>;

class Employee {
 public:
  // Default constructor - applies default values, potentially ones specified
  // explicitly within the Mojom.
  Employee();

  // Value constructor - an explicit argument for every field in the struct, in
  // lexical Mojom definition order.
  Employee(int64_t id, const std::string& username, Department department);

  // Creates a new copy of this struct value
  EmployeePtr Clone();

  // Tests for equality with another struct value of the same type.
  bool Equals(const Employee& other);

  // Equivalent public fields with names identical to the Mojom.
  int64_t id;
  std::string username;
  Department department;
};

}  // namespace mojom
}  // namespace business
```

Note when used as a message parameter or as a field within another Mojom struct, a `struct` type is wrapped by the move-only `mojo::StructPtr` helper, which is roughly equivalent to a `std::unique_ptr` with some additional utility methods. This allows struct values to be nullable and struct types to be potentially self-referential.

Every genereated struct class has a static `New()` method which returns a new `mojo::StructPtr<T>` wrapping a new instance of the class constructed by forwarding the arguments from `New`. For example:

```
mojom::EmployeePtr e1 = mojom::Employee::New();
e1->id = 42;
e1->username = "mojo";
e1->department = mojom::Department::kEngineering;
```

is equivalent to

```
auto e1 = mojom::Employee::New(42, "mojo", mojom::Department::kEngineering);
```

Now if we define an interface like:

```
interface EmployeeManager {
  AddEmployee(Employee e);
};
```

We'll get this C++ interface to implement:

```
class EmployeeManager {
 public:
  virtual ~EmployeManager() {}

  virtual void AddEmployee(EmployeePtr e) = 0;
};
```

And we can send this message from C++ code as follows:

```
mojom::EmployeeManagerPtr manager = ...;
manager->AddEmployee(
    Employee::New(42, "mojo", mojom::Department::kEngineering));

// or
```

```
auto e = Employee::New(42, "mojo", mojom::Department::kEngineering);
manager->AddEmployee(std::move(e));
```

Unions

Similarly to structs, tagged unions generate an identically named, representative C++ class which is typically wrapped in a `mojo::StructPtr<T>`.

Unlike structs, all generated union fields are private and must be retrieved and manipulated using accessors. A field `foo` is accessible by `foo()` and settable by `set_foo()`. There is also a boolean `is_foo()` for each field which indicates whether the union is currently taking on the value of field `foo` in exclusion to all other union fields.

Finally, every generated union class also has a nested `Tag` enum class which enumerates all of the named union fields. A Mojom union value's current type can be determined by calling the `which()` method which returns a `Tag`.

For example, consider the following Mojom definitions:

```
union Value {
  int64 int_value;
  float float_value;
  string string_value;
};

interface Dictionary {
  AddValue(string key, Value value);
};
```

This generates the following C++ interface:

```
class Value {
 public:
  ~Value() {}
};

class Dictionary {
 public:
  virtual ~Dictionary() {}

  virtual void AddValue(const std::string& key, ValuePtr value) = 0;
};
```

And we can use it like so:

```
ValuePtr value = Value::New();
value->set_int_value(42);
CHECK(value->is_int_value());
```

```
CHECK_EQ(value->which(), Value::Tag::INT_VALUE);

value->set_float_value(42);
CHECK(value->is_float_value());
CHECK_EQ(value->which(), Value::Tag::FLOAT_VALUE);

value->set_string_value("bananas");
CHECK(value->is_string_value());
CHECK_EQ(value->which(), Value::Tag::STRING_VALUE);
```

Finally, note that if a union value is not currently occupied by a given field, attempts to access that field will DCHECK:

```
ValuePtr value = Value::New();
value->set_int_value(42);
LOG(INFO) << "Value is " << value->string_value();  // DCHECK!
```

Sending Interfaces Over Interfaces

We know how to create interface pipes and use their Ptr and Request endpoints in some interesting ways. This still doesn't add up to interesting IPC! The bread and butter of Mojo IPC is the ability to transfer interface endpoints across other interfaces, so let's take a look at how to accomplish that.

Sending Interface Requests

Consider a new example Mojom in `//sample/db.mojom`:

```
module db.mojom;

interface Table {
  void AddRow(int32 key, string data);
};

interface Database {
  AddTable(Table& table);
};
```

As noted in the Mojom IDL documentation, the `Table&` syntax denotes a `Table` interface request. This corresponds precisely to the `InterfaceRequest<T>` type discussed in the sections above, and in fact the generated code for these interfaces is approximately:

```
namespace db {
namespace mojom {

class Table {
 public:
  virtual ~Table() {}

  virtual void AddRow(int32_t key, const std::string& data) = 0;
}

using TablePtr = mojo::InterfacePtr<Table>;
using TableRequest = mojo::InterfaceRequest<Table>;

class Database {
 public:
  virtual ~Database() {}

  virtual void AddTable(TableRequest table);
};

using DatabasePtr = mojo::InterfacePtr<Database>;
using DatabaseRequest = mojo::InterfaceRequest<Database>;

}  // namespace mojom
}  // namespace db
```

We can put this all together now with an implementation of `Table` and `Database`:

```
#include "sample/db.mojom.h"

class TableImpl : public db::mojom:Table {
 public:
  explicit TableImpl(db::mojom::TableRequest request)
      : binding_(this, std::move(request)) {}
  ~TableImpl() override {}

  // db::mojom::Table:
  void AddRow(int32_t key, const std::string& data) override {
    rows_.insert({key, data});
  }

 private:
  mojo::Binding<db::mojom::Table> binding_;
  std::map<int32_t, std::string> rows_;
};

class DatabaseImpl : public db::mojom::Database {
 public:
  explicit DatabaseImpl(db::mojom::DatabaseRequest request)
      : binding_(this, std::move(request)) {}
  ~DatabaseImpl() override {}

  // db::mojom::Database:
```

```
  void AddTable(db::mojom::TableRequest table) {
    tables_.emplace_back(std::make_unique<TableImpl>(std::move(table)));
  }

 private:
  mojo::Binding<db::mojom::Database> binding_;
  std::vector<std::unique_ptr<TableImpl>> tables_;
};
```

Pretty straightforward. The `Table&` Mojom paramter to `AddTable` translates to a C++ `db::mojom::TableRequest`, aliased from `mojo::InterfaceRequest<db::mojom::Table>`, which we know is just a strongly-typed message pipe handle. When `DatabaseImpl` gets an `AddTable` call, it constructs a new `TableImpl` and binds it to the received `TableRequest`.

Let's see how this can be used.

```
db::mojom::DatabasePtr database;
DatabaseImpl db_impl(mojo::MakeRequest(&database));

db::mojom::TablePtr table1, table2;
database->AddTable(mojo::MakeRequest(&table1));
database->AddTable(mojo::MakeRequest(&table2));

table1->AddRow(1, "hiiiiiiii");
table2->AddRow(2, "heyyyyyy");
```

Notice that we can again start using the new `Table` pipes immediately, even while their `TableRequest` endpoints are still in transit.

Sending InterfacePtrs

Of course we can also send `InterfacePtr`s:

```
interface TableListener {
  OnRowAdded(int32 key, string data);
};

interface Table {
  AddRow(int32 key, string data);

  AddListener(TableListener listener);
};
```

This would generate a `Table::AddListener` signature like so:

```
  virtual void AddListener(TableListenerPtr listener) = 0;
```

and this could be used like so:

```
db::mojom::TableListenerPtr listener;
TableListenerImpl impl(mojo::MakeRequest(&listener));
table->AddListener(std::move(listener));
```

Other Interface Binding Types

The Interfaces section above covers basic usage of the most common bindings object types: `InterfacePtr`, `InterfaceRequest`, and `Binding`. While these types are probably the most commonly used in practice, there are several other ways of binding both client- and implementation-side interface pipes.

Strong Bindings

A **strong binding** exists as a standalone object which owns its interface implementation and automatically cleans itself up when its bound interface endpoint detects an error. The `MakeStrongBinding` function is used to create such a binding. .

```
class LoggerImpl : public sample::mojom::Logger {
 public:
  LoggerImpl() {}
  ~LoggerImpl() override {}

  // sample::mojom::Logger:
  void Log(const std::string& message) override {
    LOG(ERROR) << "[Logger] " << message;
  }

 private:
  // NOTE: This doesn't own any Binding object!
};

db::mojom::LoggerPtr logger;
mojo::MakeStrongBinding(std::make_unique<LoggerImpl>(),
                        mojo::MakeRequest(&logger));

logger->Log("NOM NOM NOM MESSAGES");
```

Now as long as `logger` remains open somewhere in the system, the bound `LoggerImpl` on the other end will remain alive.

Binding Sets

Sometimes it's useful to share a single implementation instance with multiple clients. `BindingSet` makes this easy. Consider the Mojom:

```
module system.mojom;

interface Logger {
  Log(string message);
};

interface LoggerProvider {
  GetLogger(Logger& logger);
};
```

We can use `BindingSet` to bind multiple `Logger` requests to a single implementation instance:

```
class LogManager : public system::mojom::LoggerProvider,
                   public system::mojom::Logger {
 public:
  explicit LogManager(system::mojom::LoggerProviderRequest request)
      : provider_binding_(this, std::move(request)) {}
  ~LogManager() {}

  // system::mojom::LoggerProvider:
  void GetLogger(LoggerRequest request) override {
    logger_bindings_.AddBinding(this, std::move(request));
  }

  // system::mojom::Logger:
  void Log(const std::string& message) override {
    LOG(ERROR) << "[Logger] " << message;
  }

 private:
  mojo::Binding<system::mojom::LoggerProvider> provider_binding_;
  mojo::BindingSet<system::mojom::Logger> logger_bindings_;
};
```

InterfacePtr Sets

Similar to the `BindingSet` above, sometimes it's useful to maintain a set of `InterfacePtr`s for *e.g.* a set of clients observing some event. `InterfacePtrSet` is here to help. Take the Mojom:

```
module db.mojom;

interface TableListener {
  OnRowAdded(int32 key, string data);
};
```

```
interface Table {
  AddRow(int32 key, string data);
  AddListener(TableListener listener);
};
```

An implementation of `Table` might look something like like this:

```
class TableImpl : public db::mojom::Table {
 public:
  TableImpl() {}
  ~TableImpl() override {}

  // db::mojom::Table:
  void AddRow(int32_t key, const std::string& data) override {
    rows_.insert({key, data});
    listeners_.ForEach([key, &data](db::mojom::TableListener* listener) {
      listener->OnRowAdded(key, data);
    });
  }

  void AddListener(db::mojom::TableListenerPtr listener) {
    listeners_.AddPtr(std::move(listener));
  }

 private:
  mojo::InterfacePtrSet<db::mojom::Table> listeners_;
  std::map<int32_t, std::string> rows_;
};
```

Associated Interfaces

Associated interfaces are interfaces which:

- enable running multiple interfaces over a single message pipe while preserving message ordering.
- make it possible for the bindings to access a single message pipe from multiple sequences.

Mojom

A new keyword `associated` is introduced for interface pointer/request fields. For example:

```
interface Bar {};

struct Qux {
  associated Bar bar3;
};

interface Foo {
  // Uses associated interface pointer.
  SetBar(associated Bar bar1);
  // Uses associated interface request.
  GetBar(associated Bar& bar2);
  // Passes a struct with associated interface pointer.
  PassQux(Qux qux);
  // Uses associated interface pointer in callback.
  AsyncGetBar() => (associated Bar bar4);
};
```

It means the interface impl/client will communicate using the same message pipe over which the associated interface pointer/request is passed.

Using associated interfaces in C++

When generating C++ bindings, the associated interface pointer of `Bar` is mapped to `BarAssociatedPtrInfo` (which is an alias of `mojo::AssociatedInterfacePtrInfo<Bar>`); associated interface request to `BarAssociatedRequest` (which is an alias of `mojo::AssociatedInterfaceRequest<Bar>`).

```
// In mojom:
interface Foo {
  ...
  SetBar(associated Bar bar1);
  GetBar(associated Bar& bar2);
  ...
};

// In C++:
class Foo {
  ...
  virtual void SetBar(BarAssociatedPtrInfo bar1) = 0;
  virtual void GetBar(BarAssociatedRequest bar2) = 0;
  ...
};
```

Passing associated interface requests

Assume you have already got an `InterfacePtr<Foo> foo_ptr`, and you would like to call `GetBar()` on it. You can do:

```
BarAssociatedPtrInfo bar_ptr_info;
BarAssociatedRequest bar_request = MakeRequest(&bar_ptr_info);
foo_ptr->GetBar(std::move(bar_request));

// BarAssociatedPtr is an alias of AssociatedInterfacePtr<Bar>.
BarAssociatedPtr bar_ptr;
```

```
bar_ptr.Bind(std::move(bar_ptr_info));
bar_ptr->DoSomething();
```

First, the code creates an associated interface of type `Bar`. It looks very similar to what you would do to setup a non-associated interface. An important difference is that one of the two associated endpoints (either `bar_request` or `bar_ptr_info`) must be sent over another interface. That is how the interface is associated with an existing message pipe.

It should be noted that you cannot call `bar_ptr->DoSomething()` before passing `bar_request`. This is required by the FIFO-ness guarantee: at the receiver side, when the message of `DoSomething` call arrives, we want to dispatch it to the corresponding `AssociatedBinding<Bar>` before processing any subsequent messages. If `bar_request` is in a subsequent message, message dispatching gets into a deadlock. On the other hand, as soon as `bar_request` is sent, `bar_ptr` is usable. There is no need to wait until `bar_request` is bound to an implementation at the remote side.

A `MakeRequest` overload which takes an `AssociatedInterfacePtr` pointer (instead of an `AssociatedInterfacePtrInfo` pointer) is provided to make the code a little shorter. The following code achieves the same purpose:

```
BarAssociatedPtr bar_ptr;
foo_ptr->GetBar(MakeRequest(&bar_ptr));
bar_ptr->DoSomething();
```

The implementation of `Foo` looks like this:

```
class FooImpl : public Foo {
  ...
  void GetBar(BarAssociatedRequest bar2) override {
    bar_binding_.Bind(std::move(bar2));
    ...
  }
  ...

  Binding<Foo> foo_binding_;
  AssociatedBinding<Bar> bar_binding_;
};
```

In this example, `bar_binding_`'s lifespan is tied to that of `FooImpl`. But you don't have to do that. You can, for example, pass `bar2` to another sequence to bind to an `AssociatedBinding<Bar>` there.

When the underlying message pipe is disconnected (e.g., `foo_ptr` or `foo_binding_` is destroyed), all associated interface endpoints (e.g., `bar_ptr` and `bar_binding_`) will receive a connection error.

Passing associated interface pointers

Similarly, assume you have already got an `InterfacePtr<Foo> foo_ptr`, and you would like to call `SetBar()` on it. You can do:

```
AssociatedBinding<Bar> bar_binding(some_bar_impl);
BarAssociatedPtrInfo bar_ptr_info;
BarAssociatedRequest bar_request = MakeRequest(&bar_ptr_info);
foo_ptr->SetBar(std::move(bar_ptr_info));
bar_binding.Bind(std::move(bar_request));
```

The following code achieves the same purpose:

```
AssociatedBinding<Bar> bar_binding(some_bar_impl);
BarAssociatedPtrInfo bar_ptr_info;
bar_binding.Bind(&bar_ptr_info);
foo_ptr->SetBar(std::move(bar_ptr_info));
```

Performance considerations

When using associated interfaces on different sequences than the master sequence (where the master interface lives):

- Sending messages: send happens directly on the calling sequence. So there isn't sequence hopping.
- Receiving messages: associated interfaces bound on a different sequence from the master interface incur an extra sequence hop during dispatch.

Therefore, performance-wise associated interfaces are better suited for scenarios where message receiving happens on the master sequence.

Testing

Associated interfaces need to be associated with a master interface before they can be used. This means one end of the associated interface must be sent over one end of the master interface, or over one end of another associated interface which itself already has a master interface.

If you want to test an associated interface endpoint without first associating it, you can use `mojo::MakeIsolatedRequest()`. This will create working associated interface endpoints which are not actually associated with anything else.

Read more

- Design: Mojo Associated Interfaces

Synchronous Calls

See this document

TODO: Move the above doc into the repository markdown docs.

Type Mapping

In many instances you might prefer that your generated C++ bindings use a more natural type to represent certain Mojom types in your interface methods. For one example consider a Mojom struct such as the `Rect` below:

```
module gfx.mojom;

struct Rect {
  int32 x;
  int32 y;
  int32 width;
  int32 height;
};

interface Canvas {
  void FillRect(Rect rect);
};
```

The `Canvas` Mojom interface would normally generate a C++ interface like:

```
class Canvas {
 public:
  virtual void FillRect(RectPtr rect) = 0;
};
```

However, the Chromium tree already defines a native `gfx::Rect` which is equivalent in meaning but which also has useful helper methods. Instead of manually converting between a `gfx::Rect` and the Mojom-generated `RectPtr` at every message boundary, wouldn't it be nice if the Mojom bindings generator could instead generate:

```
class Canvas {
 public:
  virtual void FillRect(const gfx::Rect& rect) = 0;
}
```

The correct answer is, “Yes! That would be nice!” And fortunately, it can!

Global Configuration

While this feature is quite powerful, it introduces some unavoidable complexity into build system. This stems from the fact that type-mapping is an inherently viral concept: if `gfx::mojom::Rect` is mapped to `gfx::Rect` anywhere, the mapping needs to apply *everywhere*.

For this reason we have a few global typemap configurations defined in chromium_bindings_configuration.gni and blink_bindings_configuration.gni. These configure the two supported variants of Mojom generated bindings in the repository. Read more on this in the sections that follow.

For now, let's take a look at how to express the mapping from `gfx::mojom::Rect` to `gfx::Rect`.

Defining `StructTraits`

In order to teach generated bindings code how to serialize an arbitrary native type `T` as an arbitrary Mojom type `mojom::U`, we need to define an appropriate specialization of the `mojo::StructTraits` template.

A valid specialization of `StructTraits` MUST define the following static methods:

- A single static accessor for every field of the Mojom struct, with the exact same name as the struct field. These accessors must all take a const ref to an object of the native type, and must return a value compatible with the Mojom struct field's type. This is used to safely and consistently extract data from the native type during message serialization without incurring extra copying costs.
- A single static `Read` method which initializes an instance of the the native type given a serialized representation of the Mojom struct. The `Read` method must return a `bool` to indicate whether the incoming data is accepted (`true`) or rejected (`false`).

There are other methods a `StructTraits` specialization may define to satisfy some less common requirements. See Advanced StructTraits Usage for details.

In order to define the mapping for `gfx::Rect`, we want the following `StructTraits` specialization, which we'll define in `//ui/gfx/geometry/mojo/geometry_struct_traits.h`:

```
#include "mojo/public/cpp/bindings/struct_traits.h"
#include "ui/gfx/geometry/rect.h"
#include "ui/gfx/geometry/mojo/geometry.mojom.h"

namespace mojo {

template <>
class StructTraits<gfx::mojom::RectDataView, gfx::Rect> {
 public:
  static int32_t x(const gfx::Rect& r) { return r.x(); }
  static int32_t y(const gfx::Rect& r) { return r.y(); }
  static int32_t width(const gfx::Rect& r) { return r.width(); }
  static int32_t height(const gfx::Rect& r) { return r.height(); }

  static bool Read(gfx::mojom::RectDataView data, gfx::Rect* out_rect);
};

}  // namespace mojo
```

And in `//ui/gfx/geometry/mojo/geometry_struct_traits.cc`:

```
#include "ui/gfx/geometry/mojo/geometry_struct_traits.h"

namespace mojo {
```

```
// static
template <>
bool StructTraits<gfx::mojom::RectDataView, gfx::Rect>::Read(
    gfx::mojom::RectDataView data,
  gfx::Rect* out_rect) {
  if (data.width() < 0 || data.height() < 0)
    return false;

  out_rect->SetRect(data.x(), data.y(), data.width(), data.height());
  return true;
};

}  // namespace mojo
```

Note that the `Read()` method returns `false` if either the incoming `width` or `height` fields are negative. This acts as a validation step during deserialization: if a client sends a `gfx::Rect` with a negative width or height, its message will be rejected and the pipe will be closed. In this way, type mapping can serve to enable custom validation logic in addition to making callsites and interface implemention more convenient.

Enabling a New Type Mapping

We've defined the `StructTraits` necessary, but we still need to teach the bindings generator (and hence the build system) about the mapping. To do this we must create a **typemap** file, which uses familiar GN syntax to describe the new type mapping.

Let's place this `geometry.typemap` file alongside our Mojom file:

```
mojom = "//ui/gfx/geometry/mojo/geometry.mojom"
public_headers = [ "//ui/gfx/geometry/rect.h" ]
traits_headers = [ "//ui/gfx/geometry/mojo/geometry_struct_traits.h" ]
sources = [
  "//ui/gfx/geometry/mojo/geometry_struct_traits.cc",
  "//ui/gfx/geometry/mojo/geometry_struct_traits.h",
]
public_deps = [ "//ui/gfx/geometry" ]
type_mappings = [
  "gfx.mojom.Rect=gfx::Rect",
]
```

Let's look at each of the variables above:

- `mojom`: Specifies the `mojom` file to which the typemap applies. Many typemaps may apply to the same `mojom` file, but any given typemap may only apply to a single `mojom` file.
- `public_headers`: Additional headers required by any code which would depend on the Mojom definition of `gfx.mojom.Rect` now that the typemap is applied. Any headers required for the native target type definition should be listed here.

- `traits_headers`: Headers which contain the relevant `StructTraits` specialization(s) for any type mappings described by this file.
- `sources`: Any implementation sources and headers needed for the `StructTraits` definition. These sources are compiled directly into the generated C++ bindings target for a `mojom` file applying this typemap.
- `public_deps`: Target dependencies exposed by the `public_headers` and `traits_headers`.
- `deps`: Target dependencies exposed by `sources` but not already covered by `public_deps`.
- `type_mappings`: A list of type mappings to be applied for this typemap. The strings in this list are of the format `"MojomType=CppType"`, where `MojomType` must be a fully qualified Mojom typename and `CppType` must be a fully qualified C++ typename. Additional attributes may be specified in square brackets following the `CppType`:

  - `move_only`: The `CppType` is move-only and should be passed by value in any generated method signatures. Note that `move_only` is transitive, so containers of `MojomType` will translate to containers of `CppType` also passed by value.
  - `copyable_pass_by_value`: Forces values of type `CppType` to be passed by value without moving them. Unlike `move_only`, this is not transitive.
  - `nullable_is_same_type`: By default a non-nullable `MojomType` will be mapped to `CppType` while a nullable `MojomType?` will be mapped to `base::Optional<CppType>`. If this attribute is set, the `base::Optional` wrapper is omitted for nullable `MojomType?` values, but the `StructTraits` definition for this type mapping must define additional `IsNull` and `SetToNull` methods. See Specializing Nullability below.
  - `force_serialize`: The typemap is incompatible with lazy serialization (e.g. consider a typemap to a `base::StringPiece`, where retaining a copy is unsafe). Any messages carrying the type will be forced down the eager serailization path.

Now that we have the typemap file we need to add it to a local list of typemaps that can be added to the global configuration. We create a new `//ui/gfx/typemaps.gni` file with the following contents:

```
typemaps = [
  "//ui/gfx/geometry/mojo/geometry.typemap",
]
```

And finally we can reference this file in the global default (Chromium) bindings configuration by adding it to `_typemap_imports` in chromium_bindings_configuration.gni:

```
_typemap_imports = [
  ...,
  "//ui/gfx/typemaps.gni",
  ...,
]
```

StructTraits Reference

Each of a `StructTraits` specialization's static getter methods -- one per struct field -- must return a type which can be used as a data source for the field during serialization. This is a quick reference mapping

Mojom field type to valid getter return types:

Mojom Field Type	C++ Getter Return Type
`bool`	`bool`
`int8`	`int8_t`
`uint8`	`uint8_t`
`int16`	`int16_t`
`uint16`	`uint16_t`
`int32`	`int32_t`
`uint32`	`uint32_t`
`int64`	`int64_t`
`uint64`	`uint64_t`
`float`	`float`
`double`	`double`
`handle`	`mojo::ScopedHandle`
`handle<message_pipe>`	`mojo::ScopedMessagePipeHandle`
`handle<data_pipe_consumer>`	`mojo::ScopedDataPipeConsumerHandle`
`handle<data_pipe_producer>`	`mojo::ScopedDataPipeProducerHandle`
`handle<shared_buffer>`	`mojo::ScopedSharedBufferHandle`
`FooInterface`	`FooInterfacePtr`
`FooInterface&`	`FooInterfaceRequest`
`associated FooInterface`	`FooAssociatedInterfacePtr`
`associated FooInterface&`	`FooAssociatedInterfaceRequest`
`string`	Value or reference to any type `T` that has a `mojo::StringTraits` specialization defined. By default this includes `std::string`, `base::StringPiece`, and `WTF::String` (Blink).
`array<T>`	Value or reference to any type `T` that has a `mojo::ArrayTraits` specialization defined. By default this includes `std::vector<T>`, `mojo::CArray<T>`, and `WTF::Vector<T>` (Blink).
`map<K, V>`	Value or reference to any type `T` that has a `mojo::MapTraits` specialization defined. By default this includes `std::map<T>`, `mojo::unordered_map<T>`, and `WTF::HashMap<T>` (Blink).

Mojom Field Type	C++ Getter Return Type
`FooEnum`	Value of any type that has an appropriate `EnumTraits` specialization defined. By default this inlcudes only the generated `FooEnum` type.
`FooStruct`	Value or reference to any type that has an appropriate `StructTraits` specialization defined. By default this includes only the generated `FooStructPtr` type.
`FooUnion`	Value of reference to any type that has an appropriate `UnionTraits` specialization defined. By default this includes only the generated `FooUnionPtr` type.

Using Generated DataView Types

Static `Read` methods on `StructTraits` specializations get a generated `FooDataView` argument (such as the `RectDataView` in the example above) which exposes a direct view of the serialized Mojom structure within an incoming message's contents. In order to make this as easy to work with as possible, the generated `FooDataView` types have a generated method corresponding to every struct field:

- For POD field types (*e.g.* bools, floats, integers) these are simple accessor methods with names identical to the field name. Hence in the `Rect` example we can access things like `data.x()` and `data.width()`. The return types correspond exactly to the mappings listed in the table above, under StructTraits Reference.
- For handle and interface types (*e.g* `handle` or `FooInterface&`) these are named `TakeFieldName` (for a field named `field_name`) and they return an appropriate move-only handle type by value. The return types correspond exactly to the mappings listed in the table above, under StructTraits Reference.
- For all other field types (*e.g.*, enums, strings, arrays, maps, structs) these are named `ReadFieldName` (for a field named `field_name`) and they return a `bool` (to indicate success or failure in reading). On success they fill their output argument with the deserialized field value. The output argument may be a pointer to any type with an appropriate `StructTraits` specialization defined, as mentioned in the table above, under StructTraits Reference.

An example would be useful here. Suppose we introduced a new Mojom struct:

```
struct RectPair {
  Rect left;
  Rect right;
};
```

and a corresponding C++ type:

```
class RectPair {
 public:
  RectPair() {}
  const gfx::Rect& left() const { return left_; }
  const gfx::Rect& right() const { return right_; }

  void Set(const gfx::Rect& left, const gfx::Rect& right) {
    left_ = left;
    right_ = right;
  }

  // ... some other stuff

 private:
  gfx::Rect left_;
  gfx::Rect right_;
};
```

Our traits to map `gfx::mojom::RectPair` to `gfx::RectPair` might look like this:

```
namespace mojo {

template <>
class StructTraits
 public:
  static const gfx::Rect& left(const gfx::RectPair& pair) {
    return pair.left();
  }

  static const gfx::Rect& right(const gfx::RectPair& pair) {
    return pair.right();
  }

  static bool Read(gfx::mojom::RectPairDataView data, gfx::RectPair* out_pair) {
    gfx::Rect left, right;
    if (!data.ReadLeft(&left) || !data.ReadRight(&right))
      return false;
    out_pair->Set(left, right);
    return true;
  }
}  // namespace mojo
```

Generated `ReadFoo` methods always convert `multi_word_field_name` fields to `ReadMultiWordFieldName` methods.

Variants

By now you may have noticed that additional C++ sources are generated when a Mojom is processed. These exist due to type mapping, and the source files we refer to throughout this docuemnt (namely `foo.mojom.cc` and `foo.mojom.h`) are really only one **variant** (the *default* or *chromium* variant) of the C++ bindings for a given Mojom file.

The only other variant currently defined in the tree is the *blink* variant, which produces a few additional files:

```
out/gen/sample/db.mojom-blink.cc
out/gen/sample/db.mojom-blink.h
```

These files mirror the definitions in the default variant but with different C++ types in place of certain builtin field and parameter types. For example, Mojom strings are represented by `WTF::String` instead of `std::string`. To avoid symbol collisions, the variant's symbols are nested in an extra inner namespace, so Blink consumer of the interface might write something like:

```
#include "sample/db.mojom-blink.h"

class TableImpl : public db::mojom::blink::Table {
 public:
  void AddRow(int32_t key, const WTF::String& data) override {
    // ...
  }
};
```

In addition to using different C++ types for builtin strings, arrays, and maps, the global typemap configuration for default and “blink” variants are completely separate. To add a typemap for the Blink configuration, you can modify blink_bindings_configuration.gni.

All variants share some definitions which are unaffected by differences in the type mapping configuration (enums, for example). These definitions are generated in *shared* sources:

```
out/gen/sample/db.mojom-shared.cc
out/gen/sample/db.mojom-shared.h
out/gen/sample/db.mojom-shared-internal.h
```

Including either variant's header (`db.mojom.h` or `db.mojom-blink.h`) implicitly includes the shared header, but may wish to include *only* the shared header in some instances.

Finally, note that for `mojom` GN targets, there is implicitly a corresponding `mojom_{variant}` target defined for any supported bindings configuration. So for example if you've defined in `//sample/BUILD.gn`:

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

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

Code in Blink which wishes to use the generated Blink-variant definitions must depend on `"//sample:mojom_blink"`.

Versioning Considerations

For general documentation of versioning in the Mojom IDL see Versioning.

This section briefly discusses some C++-specific considerations relevant to versioned Mojom types.

Querying Interface Versions

`InterfacePtr` defines the following methods to query or assert remote interface version:

```
void QueryVersion(const base::Callback<void(uint32_t)>& callback);
```

This queries the remote endpoint for the version number of its binding. When a response is received `callback` is invoked with the remote version number. Note that this value is cached by the `InterfacePtr` instance to avoid redundant queries.

```
void RequireVersion(uint32_t version);
```

Informs the remote endpoint that a minimum version of `version` is required by the client. If the remote endpoint cannot support that version, it will close its end of the pipe immediately, preventing any other requests from being received.

Versioned Enums

For convenience, every extensible enum has a generated helper function to determine whether a received enum value is known by the implementation's current version of the enum definition. For example:

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

generates the function in the same namespace as the generated C++ enum type:

```
inline bool IsKnownEnumValue(Department value);
```

Using Mojo Bindings in Chrome

See Converting Legacy Chrome IPC To Mojo.

Additional Documentation

Calling Mojo From Blink : A brief overview of what it looks like to use Mojom C++ bindings from within Blink code.