Supporting Custom C++ Types

> **CAUTION**
>
> This documentation is still **experimental** and details are subject to changes as we iterate. Feel free to share your feedback on the discussion inside the working group for this page.
>
> Moreover, it contains several **manual steps**. Please note that this won't be representative of the final developer experience once the New Architecture is stable. We're working on tools, templates and libraries to help you get started fast on the New Architecture, without having to go through the whole setup.

By default C++ Turbo Native Modules support bridging functionality for most `std::` standard types.

If you want to add support for new / custom types in your app / library, you only need to provide the necessary `bridging` header file.

This guide continues the previous C++ Turbo Native Modules section.

Example: Int64

C++ Turbo Native Modules don't support `int64_t` numbers yet - because JavaScript doesn't support numbers greater `2^53`.

We can't represent numbers > `2^53` as JavaScript `number`'s - but we can represent them as JavaScript `string`'s and automatically convert (aka. `bridge`) them to C++ `int64_t`'s by creating a custom Bridging header file called `Int64.h` in the `tm` folder:

```
#pragma once

#include <react/bridging/Bridging.h>

namespace facebook::react {
```

```
template <>
struct Bridging<int64_t> {
  static int64_t fromJs(jsi::Runtime &rt, const jsi::String &value) {
    try {
      size_t pos;
      auto str = value.utf8(rt);
      auto num = std::stoll(str, &pos);
      if (pos != str.size()) {
        throw std::invalid_argument("Invalid number"); // don't support alphanumeric
strings
      }
      return num;
    } catch (const std::logic_error &e) {
      throw jsi::JSError(rt, e.what());
    }
  }

  static jsi::String toJs(jsi::Runtime &rt, int64_t value) {
    return bridging::toJs(rt, std::to_string(value));
  }
};

} // namespace facebook::react
```

The key components for your custom `bridging` header are:

- Explicit specialization of the `Bridging` struct for your custom type, `int64_t` in this case
- A `fromJs` function to convert from `jsi::` types to your custom type
- A `toJS` function to convert from your custom type to `jsi:` types

Omitting either `fromJS` or `toJS` would make you `bridging` header either *readonly* or *writeonly*.

Now you can add the following function to your JavaScript spec:

TypeScript Flow

NativeSampleModule.ts

```
  // ...
  readonly cubicRoot: (input: string) => number;
  // ..
```

Declare it in your `NativeSampleModule.h` file and include the `Int64.h` header file:

```
//...
#include "Int64.h"
//...
int32_t cubicRoot(jsi::Runtime& rt, int64_t input);
```

And implement it in `NativeSampleModule.cpp`:

```
//...
#include <cmath>
//...
int32_t NativeSampleModule::cubicRoot(jsi::Runtime& rt, int64_t input) {
    return std::cbrt(input);
}
```

In your app you can call this new native function via:

```
<Section title="Cxx TurboModule">
  NativeSampleModule.cubicRoot(...) ={' '}
  {JSON.stringify(
    NativeSampleModule.cubicRoot('9223372036854775807'),
  )}
</Section>
```

which should return `2097152`.

Any custom type

Similar to the example above you can now write custom `bridging` functionality for any custom C++ type you want to expose to react-native. E.g., you can add support for

`folly::StringPiece`, `QString`, `boost::filesystem::path`, `absl::optional` or any other type you need to support in your C++ Turbo Native Modules.

Path.h

```
#pragma once

#include <react/bridging/Bridging.h>
#include <boost/filesystem.hpp>

namespace facebook::react {

template<>
struct Bridging<boost::filesystem::path> {
  static boost::filesystem::path fromJs(jsi::Runtime& rt, const std::string& value) { //
auto-bridge from jsi::String to std::string
    return boost::filesystem::path(value);
  }

  static jsi::String toJs(jsi::Runtime& rt, boost::filesystem::path value) {
    return bridging::toJs(rt, value.string());
  }
};

} // namespace facebook::react
```

Custom structs

You can use the same approach for you custom types in JavaScript such as this one:

```
export type CustomType = {
  key: string,
  enabled: boolean,
  time?: number,
};
```

which can be exposed to your C++ Turbo Native Module via

TypeScript Flow

NativeSampleModule.ts

```
  // ...
  readonly passCustomType: (input: CustomType) => CustomType;
  // ..
```

Manually typed

To use this custom type in C++, you need to define your custom Struct and `bridging` function e.g., directly in `NativeSampleModule.h`:

```
struct CustomType {
  std::string key;
  bool enabled;
  std::optional<int32_t> time;
};

template <>
struct Bridging<CustomType> {
  static CustomType fromJs(
      jsi::Runtime &rt,
      const jsi::Object &value,
      const std::shared_ptr<CallInvoker> &jsInvoker) {
    return CustomType{
        bridging::fromJs<std::string>(
            rt, value.getProperty(rt, "key"), jsInvoker),
        bridging::fromJs<bool>(
            rt, value.getProperty(rt, "enabled"), jsInvoker),
        bridging::fromJs<std::optional<int32_t>>(
            rt, value.getProperty(rt, "time"), jsInvoker)};
  }

  static jsi::Object toJs(jsi::Runtime &rt, const CustomType &value) {
    auto result = facebook::jsi::Object(rt);
    result.setProperty(rt, "key", bridging::toJs(rt, value.key));
    result.setProperty(rt, "enabled", bridging::toJs(rt, value.enabled));
    if (value.time) {
      result.setProperty(rt, "time", bridging::toJs(rt, value.time.value()));
    }
    return result;
  }
};
```

Declare it in your `NativeSampleModule.h` file:

```
  CustomType passCustomType(jsi::Runtime& rt, CustomType input);
```

Implement it in `NativeSampleModule.cpp`:

```
CustomType NativeSampleModule::passCustomType(jsi::Runtime& rt, CustomType input) {
  input.key = "1909";
  input.enabled = !input.enabled;
  input.time = 42;
  return input;
}
```

In your app you can call this new native function via:

```
<Section title="Cxx TurboModule">
  NativeSampleModule.passCustomType(...) ={' '}
  {JSON.stringify(
    NativeSampleModule.passCustomType({
      key: '123',
      enabled: true,
      time: undefined,
    }),
  )}
</Section>
```

which should return `{"key":"1909","enabled":false","time":42}`.

This works - but is quite complex.

Struct generator

Codegen for C++ Turbo Native Modules does support struct generators, so you can simplify the code above in `NativeSampleModule.h` to:

```
using CustomType = NativeSampleModuleBaseCustomType<std::string, bool,
std::optional<int32_t>>;
template <>
```

```
struct Bridging<CustomType>
    : NativeSampleModuleBaseCustomTypeBridging<std::string, bool, std::optional<int32_t>>
{};
```

With `using CustomType` you declare a name for your concrete struct.

Member types

With `std::string, bool, std::optional<int32_t>` you define the property types of the struct members in the order they were defined in your JavaScript spec. The **order matters**. The *1st* template argument refers to the *1st* data type of the struct, and so forth.

Without any custom conversion functions:

- you can only `bridge` a JS string to a std::string and a JS boolean to a bool.
- but you can choose different JS `number` representations in C++.

Base class

`NativeSampleModuleBaseCustomType` is an auto-generated template in your `AppSpecsJSI.h` which name is generated by:

- `NativeSampleModule` (name of C++ Turbo Native Module in the JavaScript spec) +
- `Base` (constant) +
- `CustomType` (name of type in the JavaScript spec)

The same naming schema applies to the necessary `Bridging` struct which is defined via `struct Bridging<CustomType>.`