// Copyright 2013 The Flutter Authors. All rights reserved.

// Use of this source code is governed by a BSD-style license that can be

// found in the LICENSE file.

#ifndef FLUTTER\_SHELL\_PLATFORM\_COMMON\_CLIENT\_WRAPPER\_INCLUDE\_FLUTTER\_EVENT\_CHANNEL\_H\_

#define FLUTTER\_SHELL\_PLATFORM\_COMMON\_CLIENT\_WRAPPER\_INCLUDE\_FLUTTER\_EVENT\_CHANNEL\_H\_

#include <iostream>

#include <memory>

#include <string>

#include "binary\_messenger.h"

#include "engine\_method\_result.h"

#include "event\_sink.h"

#include "event\_stream\_handler.h"

namespace flutter {

class EncodableValue;

// A named channel for communicating with the Flutter application using

// asynchronous event streams. Incoming requests for event stream setup are

// decoded from binary on receipt, and C++ responses and events are encoded into

// binary before being transmitted back to Flutter. The MethodCodec used must be

// compatible with the one used by the Flutter application. This can be achieved

// by creating an EventChannel

// ("https://api.flutter.dev/flutter/services/EventChannel-class.html")

// counterpart of this channel on the Dart side.

// The C++ type of stream configuration arguments, events, and error details are

// templated, but only values supported by the specified MethodCodec can be

// used.

template <typename T = EncodableValue>

class EventChannel {

public:

// Creates an instance that sends and receives event handler on the channel

// named |name|, encoded with |codec| and dispatched via |messenger|.

EventChannel(BinaryMessenger\* messenger,

const std::string& name,

const MethodCodec<T>\* codec)

: messenger\_(messenger), name\_(name), codec\_(codec) {}

~EventChannel() = default;

// Prevent copying.

EventChannel(EventChannel const&) = delete;

EventChannel& operator=(EventChannel const&) = delete;

// Registers a stream handler on this channel.

// If no handler has been registered, any incoming stream setup requests will

// be handled silently by providing an empty stream.

//

// Note that the EventChannel does not own the handler and will not

// unregister it on destruction. The caller is responsible for unregistering

// the handler if it should no longer be called.

void SetStreamHandler(std::unique\_ptr<StreamHandler<T>> handler) {

if (!handler) {

messenger\_->SetMessageHandler(name\_, nullptr);

return;

}

// std::function requires a copyable lambda, so convert to a shared pointer.

// This is safe since only one copy of the shared\_pointer will ever be

// accessed.

std::shared\_ptr<StreamHandler<T>> shared\_handler(handler.release());

const MethodCodec<T>\* codec = codec\_;

const std::string channel\_name = name\_;

const BinaryMessenger\* messenger = messenger\_;

BinaryMessageHandler binary\_handler =

[shared\_handler, codec, channel\_name, messenger,

// Mutable state to track the handler's listening status.

is\_listening = bool(false)](const uint8\_t\* message,

const size\_t message\_size,

const BinaryReply& reply) mutable {

constexpr char kOnListenMethod[] = "listen";

constexpr char kOnCancelMethod[] = "cancel";

std::unique\_ptr<MethodCall<T>> method\_call =

codec->DecodeMethodCall(message, message\_size);

if (!method\_call) {

std::cerr

<< "Unable to construct method call from message on channel: "

<< channel\_name << std::endl;

reply(nullptr, 0);

return;

}

const std::string& method = method\_call->method\_name();

if (method.compare(kOnListenMethod) == 0) {

if (is\_listening) {

std::unique\_ptr<StreamHandlerError<T>> error =

shared\_handler->OnCancel(nullptr);

if (error) {

std::cerr << "Failed to cancel existing stream: "

<< (error->error\_code) << ", "

<< (error->error\_message) << ", "

<< (error->error\_details);

}

}

is\_listening = true;

std::unique\_ptr<std::vector<uint8\_t>> result;

auto sink = std::make\_unique<EventSinkImplementation>(

messenger, channel\_name, codec);

std::unique\_ptr<StreamHandlerError<T>> error =

shared\_handler->OnListen(method\_call->arguments(),

std::move(sink));

if (error) {

result = codec->EncodeErrorEnvelope(error->error\_code,

error->error\_message,

error->error\_details.get());

} else {

result = codec->EncodeSuccessEnvelope();

}

reply(result->data(), result->size());

} else if (method.compare(kOnCancelMethod) == 0) {

std::unique\_ptr<std::vector<uint8\_t>> result;

if (is\_listening) {

std::unique\_ptr<StreamHandlerError<T>> error =

shared\_handler->OnCancel(method\_call->arguments());

if (error) {

result = codec->EncodeErrorEnvelope(error->error\_code,

error->error\_message,

error->error\_details.get());

} else {

result = codec->EncodeSuccessEnvelope();

}

is\_listening = false;

} else {

result = codec->EncodeErrorEnvelope(

"error", "No active stream to cancel", nullptr);

}

reply(result->data(), result->size());

} else {

reply(nullptr, 0);

}

};

messenger\_->SetMessageHandler(name\_, std::move(binary\_handler));

}

private:

class EventSinkImplementation : public EventSink<T> {

public:

EventSinkImplementation(const BinaryMessenger\* messenger,

const std::string& name,

const MethodCodec<T>\* codec)

: messenger\_(messenger), name\_(name), codec\_(codec) {}

~EventSinkImplementation() = default;

// Prevent copying.

EventSinkImplementation(EventSinkImplementation const&) = delete;

EventSinkImplementation& operator=(EventSinkImplementation const&) = delete;

private:

const BinaryMessenger\* messenger\_;

const std::string name\_;

const MethodCodec<T>\* codec\_;

protected:

void SuccessInternal(const T\* event = nullptr) override {

auto result = codec\_->EncodeSuccessEnvelope(event);

messenger\_->Send(name\_, result->data(), result->size());

}

void ErrorInternal(const std::string& error\_code,

const std::string& error\_message,

const T\* error\_details) override {

auto result =

codec\_->EncodeErrorEnvelope(error\_code, error\_message, error\_details);

messenger\_->Send(name\_, result->data(), result->size());

}

void EndOfStreamInternal() override { messenger\_->Send(name\_, nullptr, 0); }

};

BinaryMessenger\* messenger\_;

const std::string name\_;

const MethodCodec<T>\* codec\_;

};

} // namespace flutter

#endif // FLUTTER\_SHELL\_PLATFORM\_COMMON\_CLIENT\_WRAPPER\_INCLUDE\_FLUTTER\_EVENT\_CHANNEL\_H\_