// 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\_PLUGIN\_REGISTRAR\_H\_

#define FLUTTER\_SHELL\_PLATFORM\_COMMON\_CLIENT\_WRAPPER\_INCLUDE\_FLUTTER\_PLUGIN\_REGISTRAR\_H\_

#include <flutter\_plugin\_registrar.h>

#include <map>

#include <memory>

#include <set>

#include <string>

#include "binary\_messenger.h"

#include "texture\_registrar.h"

namespace flutter {

class Plugin;

// A object managing the registration of a plugin for various events.

//

// Currently this class has very limited functionality, but is expected to

// expand over time to more closely match the functionality of

// the Flutter mobile plugin APIs' plugin registrars.

class PluginRegistrar {

public:

// Creates a new PluginRegistrar. |core\_registrar| and the messenger it

// provides must remain valid as long as this object exists.

explicit PluginRegistrar(FlutterDesktopPluginRegistrarRef core\_registrar);

virtual ~PluginRegistrar();

// Prevent copying.

PluginRegistrar(PluginRegistrar const&) = delete;

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

// Returns the messenger to use for creating channels to communicate with the

// Flutter engine.

//

// This pointer will remain valid for the lifetime of this instance.

BinaryMessenger\* messenger() { return messenger\_.get(); }

// Returns the texture registrar to use for the plugin to render a pixel

// buffer.

TextureRegistrar\* texture\_registrar() { return texture\_registrar\_.get(); }

// Takes ownership of |plugin|.

//

// Plugins are not required to call this method if they have other lifetime

// management, but this is a convenient place for plugins to be owned to

// ensure that they stay valid for any registered callbacks.

void AddPlugin(std::unique\_ptr<Plugin> plugin);

protected:

FlutterDesktopPluginRegistrarRef registrar() const { return registrar\_; }

// Destroys all owned plugins. Subclasses should call this at the beginning of

// their destructors to prevent the possibility of an owned plugin trying to

// access destroyed state during its own destruction.

void ClearPlugins();

private:

// Handle for interacting with the C API's registrar.

FlutterDesktopPluginRegistrarRef registrar\_;

std::unique\_ptr<BinaryMessenger> messenger\_;

std::unique\_ptr<TextureRegistrar> texture\_registrar\_;

// Plugins registered for ownership.

std::set<std::unique\_ptr<Plugin>> plugins\_;

};

// A plugin that can be registered for ownership by a PluginRegistrar.

class Plugin {

public:

virtual ~Plugin() = default;

};

// A singleton to own PluginRegistrars. This is intended for use in plugins,

// where there is no higher-level object to own a PluginRegistrar that can

// own plugin instances and ensure that they live as long as the engine they

// are registered with.

class PluginRegistrarManager {

public:

static PluginRegistrarManager\* GetInstance();

// Prevent copying.

PluginRegistrarManager(PluginRegistrarManager const&) = delete;

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

// Returns a plugin registrar wrapper of type T, which must be a kind of

// PluginRegistrar, creating it if necessary. The returned registrar will

// live as long as the underlying FlutterDesktopPluginRegistrarRef, so

// can be used to own plugin instances.

//

// Calling this multiple times for the same registrar\_ref with different

// template types results in undefined behavior.

template <class T>

T\* GetRegistrar(FlutterDesktopPluginRegistrarRef registrar\_ref) {

auto insert\_result =

registrars\_.emplace(registrar\_ref, std::make\_unique<T>(registrar\_ref));

auto& registrar\_pair = \*(insert\_result.first);

FlutterDesktopPluginRegistrarSetDestructionHandler(registrar\_pair.first,

OnRegistrarDestroyed);

return static\_cast<T\*>(registrar\_pair.second.get());

}

// Destroys all registrar wrappers created by the manager.

//

// This is intended primarily for use in tests.

void Reset() { registrars\_.clear(); }

private:

PluginRegistrarManager();

using WrapperMap = std::map<FlutterDesktopPluginRegistrarRef,

std::unique\_ptr<PluginRegistrar>>;

static void OnRegistrarDestroyed(FlutterDesktopPluginRegistrarRef registrar);

WrapperMap\* registrars() { return &registrars\_; }

WrapperMap registrars\_;

};

} // namespace flutter

#endif // FLUTTER\_SHELL\_PLATFORM\_COMMON\_CLIENT\_WRAPPER\_INCLUDE\_FLUTTER\_PLUGIN\_REGISTRAR\_H\_