// Licensed to the Apache Software Foundation (ASF) under one

// or more contributor license agreements. See the NOTICE file

// distributed with this work for additional information

// regarding copyright ownership. The ASF licenses this file

// to you under the Apache License, Version 2.0 (the

// "License"); you may not use this file except in compliance

// with the License. You may obtain a copy of the License at

//

// http://www.apache.org/licenses/LICENSE-2.0

//

// Unless required by applicable law or agreed to in writing,

// software distributed under the License is distributed on an

// "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY

// KIND, either express or implied. See the License for the

// specific language governing permissions and limitations

// under the License.

#include <signal.h>

#include <utility>

#include "arrow/python/flight.h"

#include "arrow/util/io\_util.h"

#include "arrow/util/logging.h"

using arrow::flight::FlightPayload;

namespace arrow {

namespace py {

namespace flight {

const char\* kPyServerMiddlewareName = "arrow.py\_server\_middleware";

PyServerAuthHandler::PyServerAuthHandler(PyObject\* handler,

const PyServerAuthHandlerVtable& vtable)

: vtable\_(vtable) {

Py\_INCREF(handler);

handler\_.reset(handler);

}

Status PyServerAuthHandler::Authenticate(arrow::flight::ServerAuthSender\* outgoing,

arrow::flight::ServerAuthReader\* incoming) {

return SafeCallIntoPython([=] {

const Status status = vtable\_.authenticate(handler\_.obj(), outgoing, incoming);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

}

Status PyServerAuthHandler::IsValid(const std::string& token,

std::string\* peer\_identity) {

return SafeCallIntoPython([=] {

const Status status = vtable\_.is\_valid(handler\_.obj(), token, peer\_identity);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

}

PyClientAuthHandler::PyClientAuthHandler(PyObject\* handler,

const PyClientAuthHandlerVtable& vtable)

: vtable\_(vtable) {

Py\_INCREF(handler);

handler\_.reset(handler);

}

Status PyClientAuthHandler::Authenticate(arrow::flight::ClientAuthSender\* outgoing,

arrow::flight::ClientAuthReader\* incoming) {

return SafeCallIntoPython([=] {

const Status status = vtable\_.authenticate(handler\_.obj(), outgoing, incoming);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

}

Status PyClientAuthHandler::GetToken(std::string\* token) {

return SafeCallIntoPython([=] {

const Status status = vtable\_.get\_token(handler\_.obj(), token);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

}

PyFlightServer::PyFlightServer(PyObject\* server, const PyFlightServerVtable& vtable)

: vtable\_(vtable) {

Py\_INCREF(server);

server\_.reset(server);

}

Status PyFlightServer::ListFlights(

const arrow::flight::ServerCallContext& context,

const arrow::flight::Criteria\* criteria,

std::unique\_ptr<arrow::flight::FlightListing>\* listings) {

return SafeCallIntoPython([&] {

const Status status =

vtable\_.list\_flights(server\_.obj(), context, criteria, listings);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

}

Status PyFlightServer::GetFlightInfo(const arrow::flight::ServerCallContext& context,

const arrow::flight::FlightDescriptor& request,

std::unique\_ptr<arrow::flight::FlightInfo>\* info) {

return SafeCallIntoPython([&] {

const Status status = vtable\_.get\_flight\_info(server\_.obj(), context, request, info);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

}

Status PyFlightServer::GetSchema(const arrow::flight::ServerCallContext& context,

const arrow::flight::FlightDescriptor& request,

std::unique\_ptr<arrow::flight::SchemaResult>\* result) {

return SafeCallIntoPython([&] {

const Status status = vtable\_.get\_schema(server\_.obj(), context, request, result);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

}

Status PyFlightServer::DoGet(const arrow::flight::ServerCallContext& context,

const arrow::flight::Ticket& request,

std::unique\_ptr<arrow::flight::FlightDataStream>\* stream) {

return SafeCallIntoPython([&] {

const Status status = vtable\_.do\_get(server\_.obj(), context, request, stream);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

}

Status PyFlightServer::DoPut(

const arrow::flight::ServerCallContext& context,

std::unique\_ptr<arrow::flight::FlightMessageReader> reader,

std::unique\_ptr<arrow::flight::FlightMetadataWriter> writer) {

return SafeCallIntoPython([&] {

const Status status =

vtable\_.do\_put(server\_.obj(), context, std::move(reader), std::move(writer));

RETURN\_NOT\_OK(CheckPyError());

return status;

});

}

Status PyFlightServer::DoExchange(

const arrow::flight::ServerCallContext& context,

std::unique\_ptr<arrow::flight::FlightMessageReader> reader,

std::unique\_ptr<arrow::flight::FlightMessageWriter> writer) {

return SafeCallIntoPython([&] {

const Status status =

vtable\_.do\_exchange(server\_.obj(), context, std::move(reader), std::move(writer));

RETURN\_NOT\_OK(CheckPyError());

return status;

});

}

Status PyFlightServer::DoAction(const arrow::flight::ServerCallContext& context,

const arrow::flight::Action& action,

std::unique\_ptr<arrow::flight::ResultStream>\* result) {

return SafeCallIntoPython([&] {

const Status status = vtable\_.do\_action(server\_.obj(), context, action, result);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

}

Status PyFlightServer::ListActions(const arrow::flight::ServerCallContext& context,

std::vector<arrow::flight::ActionType>\* actions) {

return SafeCallIntoPython([&] {

const Status status = vtable\_.list\_actions(server\_.obj(), context, actions);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

}

Status PyFlightServer::ServeWithSignals() {

// Respect the current Python settings, i.e. only interrupt the server if there is

// an active signal handler for SIGINT and SIGTERM.

std::vector<int> signals;

for (const int signum : {SIGINT, SIGTERM}) {

ARROW\_ASSIGN\_OR\_RAISE(auto handler, ::arrow::internal::GetSignalHandler(signum));

auto cb = handler.callback();

if (cb != SIG\_DFL && cb != SIG\_IGN) {

signals.push\_back(signum);

}

}

RETURN\_NOT\_OK(SetShutdownOnSignals(signals));

// Serve until we got told to shutdown or a signal interrupted us

RETURN\_NOT\_OK(Serve());

int signum = GotSignal();

if (signum != 0) {

// Issue the signal again with Python's signal handlers restored

PyAcquireGIL lock;

raise(signum);

// XXX Ideally we would loop and serve again if no exception was raised.

// Unfortunately, gRPC will return immediately if Serve() is called again.

ARROW\_UNUSED(PyErr\_CheckSignals());

}

return Status::OK();

}

PyFlightResultStream::PyFlightResultStream(PyObject\* generator,

PyFlightResultStreamCallback callback)

: callback\_(callback) {

Py\_INCREF(generator);

generator\_.reset(generator);

}

arrow::Result<std::unique\_ptr<arrow::flight::Result>> PyFlightResultStream::Next() {

return SafeCallIntoPython(

[=]() -> arrow::Result<std::unique\_ptr<arrow::flight::Result>> {

std::unique\_ptr<arrow::flight::Result> result;

const Status status = callback\_(generator\_.obj(), &result);

RETURN\_NOT\_OK(CheckPyError());

RETURN\_NOT\_OK(status);

return result;

});

}

PyFlightDataStream::PyFlightDataStream(

PyObject\* data\_source, std::unique\_ptr<arrow::flight::FlightDataStream> stream)

: stream\_(std::move(stream)) {

Py\_INCREF(data\_source);

data\_source\_.reset(data\_source);

}

std::shared\_ptr<Schema> PyFlightDataStream::schema() { return stream\_->schema(); }

arrow::Result<FlightPayload> PyFlightDataStream::GetSchemaPayload() {

return stream\_->GetSchemaPayload();

}

arrow::Result<FlightPayload> PyFlightDataStream::Next() { return stream\_->Next(); }

PyGeneratorFlightDataStream::PyGeneratorFlightDataStream(

PyObject\* generator, std::shared\_ptr<arrow::Schema> schema,

PyGeneratorFlightDataStreamCallback callback, const ipc::IpcWriteOptions& options)

: schema\_(schema), mapper\_(\*schema\_), options\_(options), callback\_(callback) {

Py\_INCREF(generator);

generator\_.reset(generator);

}

std::shared\_ptr<Schema> PyGeneratorFlightDataStream::schema() { return schema\_; }

arrow::Result<FlightPayload> PyGeneratorFlightDataStream::GetSchemaPayload() {

FlightPayload payload;

RETURN\_NOT\_OK(ipc::GetSchemaPayload(\*schema\_, options\_, mapper\_, &payload.ipc\_message));

return payload;

}

arrow::Result<FlightPayload> PyGeneratorFlightDataStream::Next() {

return SafeCallIntoPython([=]() -> arrow::Result<FlightPayload> {

FlightPayload payload;

const Status status = callback\_(generator\_.obj(), &payload);

RETURN\_NOT\_OK(CheckPyError());

RETURN\_NOT\_OK(status);

return payload;

});

}

// Flight Server Middleware

PyServerMiddlewareFactory::PyServerMiddlewareFactory(PyObject\* factory,

StartCallCallback start\_call)

: start\_call\_(start\_call) {

Py\_INCREF(factory);

factory\_.reset(factory);

}

Status PyServerMiddlewareFactory::StartCall(

const arrow::flight::CallInfo& info,

const arrow::flight::CallHeaders& incoming\_headers,

std::shared\_ptr<arrow::flight::ServerMiddleware>\* middleware) {

return SafeCallIntoPython([&] {

const Status status = start\_call\_(factory\_.obj(), info, incoming\_headers, middleware);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

}

PyServerMiddleware::PyServerMiddleware(PyObject\* middleware, Vtable vtable)

: vtable\_(vtable) {

Py\_INCREF(middleware);

middleware\_.reset(middleware);

}

void PyServerMiddleware::SendingHeaders(arrow::flight::AddCallHeaders\* outgoing\_headers) {

const Status& status = SafeCallIntoPython([&] {

const Status status = vtable\_.sending\_headers(middleware\_.obj(), outgoing\_headers);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

ARROW\_WARN\_NOT\_OK(status, "Python server middleware failed in SendingHeaders");

}

void PyServerMiddleware::CallCompleted(const Status& call\_status) {

const Status& status = SafeCallIntoPython([&] {

const Status status = vtable\_.call\_completed(middleware\_.obj(), call\_status);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

ARROW\_WARN\_NOT\_OK(status, "Python server middleware failed in CallCompleted");

}

std::string PyServerMiddleware::name() const { return kPyServerMiddlewareName; }

PyObject\* PyServerMiddleware::py\_object() const { return middleware\_.obj(); }

// Flight Client Middleware

PyClientMiddlewareFactory::PyClientMiddlewareFactory(PyObject\* factory,

StartCallCallback start\_call)

: start\_call\_(start\_call) {

Py\_INCREF(factory);

factory\_.reset(factory);

}

void PyClientMiddlewareFactory::StartCall(

const arrow::flight::CallInfo& info,

std::unique\_ptr<arrow::flight::ClientMiddleware>\* middleware) {

const Status& status = SafeCallIntoPython([&] {

const Status status = start\_call\_(factory\_.obj(), info, middleware);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

ARROW\_WARN\_NOT\_OK(status, "Python client middleware failed in StartCall");

}

PyClientMiddleware::PyClientMiddleware(PyObject\* middleware, Vtable vtable)

: vtable\_(vtable) {

Py\_INCREF(middleware);

middleware\_.reset(middleware);

}

void PyClientMiddleware::SendingHeaders(arrow::flight::AddCallHeaders\* outgoing\_headers) {

const Status& status = SafeCallIntoPython([&] {

const Status status = vtable\_.sending\_headers(middleware\_.obj(), outgoing\_headers);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

ARROW\_WARN\_NOT\_OK(status, "Python client middleware failed in StartCall");

}

void PyClientMiddleware::ReceivedHeaders(

const arrow::flight::CallHeaders& incoming\_headers) {

const Status& status = SafeCallIntoPython([&] {

const Status status = vtable\_.received\_headers(middleware\_.obj(), incoming\_headers);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

ARROW\_WARN\_NOT\_OK(status, "Python client middleware failed in StartCall");

}

void PyClientMiddleware::CallCompleted(const Status& call\_status) {

const Status& status = SafeCallIntoPython([&] {

const Status status = vtable\_.call\_completed(middleware\_.obj(), call\_status);

RETURN\_NOT\_OK(CheckPyError());

return status;

});

ARROW\_WARN\_NOT\_OK(status, "Python client middleware failed in StartCall");

}

Status CreateFlightInfo(const std::shared\_ptr<arrow::Schema>& schema,

const arrow::flight::FlightDescriptor& descriptor,

const std::vector<arrow::flight::FlightEndpoint>& endpoints,

int64\_t total\_records, int64\_t total\_bytes,

std::unique\_ptr<arrow::flight::FlightInfo>\* out) {

ARROW\_ASSIGN\_OR\_RAISE(auto result,

arrow::flight::FlightInfo::Make(\*schema, descriptor, endpoints,

total\_records, total\_bytes));

\*out = std::unique\_ptr<arrow::flight::FlightInfo>(

new arrow::flight::FlightInfo(std::move(result)));

return Status::OK();

}

Status CreateSchemaResult(const std::shared\_ptr<arrow::Schema>& schema,

std::unique\_ptr<arrow::flight::SchemaResult>\* out) {

return arrow::flight::SchemaResult::Make(\*schema).Value(out);

}

} // namespace flight

} // namespace py

} // namespace arrow