Program Listing for File fragment.hpp

Return to documentation for file (include/holoscan/core/fragment.hpp)

/* * SPDX-FileCopyrightText: Copyright (c) 2022-2024 NVIDIA CORPORATION & AFFILIATES. All rights reserved. * SPDX-License-Identifier: Apache-2.0 * * Licensed 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. */ #ifndef HOLOSCAN_CORE_FRAGMENT_HPP #define HOLOSCAN_CORE_FRAGMENT_HPP #include <future> *// for std::future* #include <iostream> *// for std::cout* #include <memory> *// for std::shared_ptr* #include <set> *// for std::set* #include <string> *// for std::string* #include <type_traits> *// for std::enable_if_t, std::is_constructible* #include <unordered_map> #include <unordered_set> #include <tuple> #include <utility> *// for std::pair* #include "common.hpp" #include "config.hpp" #include "dataflow_tracker.hpp" #include "executor.hpp" #include "graph.hpp" #include "network_context.hpp" #include "scheduler.hpp" namespace holoscan { namespace gxf { *// Forward declarations* class GXFExecutor; } *// namespace gxf // key = operator name, value = (input port names, output port names, multi-receiver names)* using FragmentPortMap = std::unordered_map<std::string, std::tuple<std::unordered_set<std::string>, std::unordered_set<std::string>, std::unordered_set<std::string>>>; *// Data structure containing port information for multiple fragments. Fragments are composed by // the workers and port information is sent back to the driver for addition to this map. // The keys are the fragment names.* using MultipleFragmentsPortMap = std::unordered_map<std::string, FragmentPortMap>; class Fragment { public: Fragment() = default; virtual ~Fragment() = default; Fragment(Fragment&&) = default; Fragment& operator= (Fragment&&) = default; Fragment& name(const std::string& name) &; Fragment&& name(const std::string& name) &&; const std::string& name() const; Fragment& application(Application* app); Application* application() const; void config(const std::string& config_file, const std::string& prefix = ""); void config(std::shared_ptr<Config>& config); Config& config(); OperatorGraph& graph(); Executor& executor(); std::shared_ptr<Scheduler> scheduler(); *// /** // * @brief Set*

*the scheduler used by the executor // * // * @param scheduler The scheduler to be added. // */* void scheduler(const std::shared_ptr<Scheduler>& scheduler); std::shared_ptr<NetworkContext> network_context(); *// /** // * @brief Set the network context used by the executor // * // * @param network_context The network context to be added. // */* void network_context(const std::shared_ptr<NetworkContext>& network_context); ArgList from_config(const std::string& key); std::unordered_set<std::string> config_keys(); template <typename OperatorT, typename StringT, typename... ArgsT, typename = std::enable_if_t<std::is_constructible_v<std::string, StringT>>> std::shared_ptr<OperatorT> make_operator(StringT name, ArgsT&&... args) { HOLOSCAN_LOG_DEBUG("Creating operator '{}'", name); auto op = std::make_shared<OperatorT>(std::forward<ArgsT>(args)...); op->name(name); op->fragment(this); auto spec = std::make_shared<OperatorSpec>(this); op->setup(*spec.get()); op->spec(spec); *// We used to initialize operator here, but now it is initialized in initialize_fragment // function after a graph of a fragment has been composed.* return op; } template <typename OperatorT, typename... ArgsT> std::shared_ptr<OperatorT> make_operator(ArgsT&&... args) { HOLOSCAN_LOG_DEBUG("Creating operator"); auto op = make_operator<OperatorT>("noname_operator", std::forward<ArgsT>(args)...); return op; } template <typename ResourceT, typename StringT, typename... ArgsT, typename = std::enable_if_t<std::is_constructible_v<std::string, StringT>>> std::shared_ptr<ResourceT> make_resource(StringT name, ArgsT&&... args) { HOLOSCAN_LOG_DEBUG("Creating resource '{}'", name); auto resource = std::make_shared<ResourceT>(std::forward<ArgsT>(args)...); resource->name(name); resource->fragment(this); auto spec = std::make_shared<ComponentSpec>(this); resource->setup(*spec.get()); resource->spec(spec); *// Skip initialization. `resource->initialize()` is done in GXFOperator::initialize()* return resource; } template <typename ResourceT, typename... ArgsT> std::shared_ptr<ResourceT> make_resource(ArgsT&&... args) { HOLOSCAN_LOG_DEBUG("Creating resource"); auto resource = make_resource<ResourceT>("noname_resource", std::forward<ArgsT>(args)...); return resource; } template <typename ConditionT, typename StringT, typename... ArgsT, typename = std::enable_if_t<std::is_constructible_v<std::string, StringT>>> std::shared_ptr<ConditionT> make_condition(StringT name, ArgsT&&... args) { HOLOSCAN_LOG_DEBUG("Creating condition '{}'", name); auto condition = std::make_shared<ConditionT>(std::forward<ArgsT>(args)...); condition-

>name(name); condition->fragment(this); auto spec = std::make_shared<ComponentSpec>(this); condition->setup(*spec.get()); condition->spec(spec); *// Skip initialization. `condition->initialize()` is done in GXFOperator::initialize()* return condition; } template <typename ConditionT, typename... ArgsT> std::shared_ptr<ConditionT> make_condition(ArgsT&&... args) { HOLOSCAN_LOG_DEBUG("Creating condition"); auto condition = make_condition<ConditionT>("noname_condition", std::forward<ArgsT>(args)...); return condition; } template <typename SchedulerT, typename StringT, typename... ArgsT, typename = std::enable_if_t<std::is_constructible_v<std::string, StringT>>> std::shared_ptr<SchedulerT> make_scheduler(StringT name, ArgsT&&... args) { HOLOSCAN_LOG_DEBUG("Creating scheduler '{}'", name); auto scheduler = std::make_shared<SchedulerT>(std::forward<ArgsT>(args)...); scheduler->name(name); scheduler->fragment(this); auto spec = std::make_shared<ComponentSpec>(this); scheduler->setup(*spec.get()); scheduler->spec(spec); *// Skip initialization. `scheduler->initialize()` is done in GXFExecutor::run()* return scheduler; } template <typename SchedulerT, typename... ArgsT> std::shared_ptr<SchedulerT> make_scheduler(ArgsT&&... args) { HOLOSCAN_LOG_DEBUG("Creating scheduler"); auto scheduler = make_scheduler<SchedulerT>("", std::forward<ArgsT>(args)...); return scheduler; } template <typename NetworkContextT, typename StringT, typename... ArgsT, typename = std::enable_if_t<std::is_constructible_v<std::string, StringT>>> std::shared_ptr<NetworkContextT> make_network_context(StringT name, ArgsT&&... args) { HOLOSCAN_LOG_DEBUG("Creating network context '{}'", name); auto network_context = std::make_shared<NetworkContextT>(std::forward<ArgsT>(args)...); network_context->name(name); network_context->fragment(this); auto spec = std::make_shared<ComponentSpec>(this); network_context->setup(*spec.get()); network_context->spec(spec); *// Skip initialization. `network_context->initialize()` is done in GXFExecutor::run()* return network_context; } template <typename NetworkContextT, typename... ArgsT> std::shared_ptr<NetworkContextT> make_network_context(ArgsT&&... args) { HOLOSCAN_LOG_DEBUG("Creating network_context"); auto network_context = make_network_context<NetworkContextT>("", std::forward<ArgsT>(args)...); return network_context; } virtual void add_operator(const std::shared_ptr<Operator>& op); virtual void add_flow(const std::shared_ptr<Operator>& upstream_op, const std::shared_ptr<Operator>& downstream_op); virtual void add_flow(const std::shared_ptr<Operator>& upstream_op, const std::shared_ptr<Operator>&

```
downstream_op, std::set<std::pair<std::string, std::string>> port_pairs); virtual void compose(); virtual void run(); virtual std::future<void> run_async(); DataFlowTracker& track(uint64_t num_start_messages_to_skip = kDefaultNumStartMessagesToSkip, uint64_t num_last_messages_to_discard = kDefaultNumLastMessagesToDiscard, int latency_threshold = kDefaultLatencyThreshold); DataFlowTracker* data_flow_tracker() { return data_flow_tracker_.get(); } virtual void compose_graph(); FragmentPortMap port_info() const; protected: friend class Application; // to access 'scheduler_' in Application friend class AppDriver; friend class gxf::GXFExecutor; template <typename ConfigT, typename... ArgsT> std::shared_ptr<Config> make_config(ArgsT&&... args) { return std::make_shared<ConfigT> (std::forward<ArgsT>(args)...); } template <typename GraphT> std::unique_ptr<GraphT> make_graph() { return std::make_unique<GraphT>(); } template <typename ExecutorT> std::shared_ptr<Executor> make_executor() { return std::make_shared<ExecutorT>(this); } template <typename ExecutorT, typename... ArgsT> std::unique_ptr<Executor> make_executor(ArgsT&&... args) { return std::make_unique<ExecutorT>(std::forward<ArgsT>(args)...); } void reset_graph_entities(); // Note: Maintain the order of declarations (executor_ and graph_) to ensure proper destruction // of the executor's context. std::string name_; Application* app_ = nullptr; std::shared_ptr<Config> config_; std::shared_ptr<Executor> executor_; std::unique_ptr<OperatorGraph> graph_; std::shared_ptr<Scheduler> scheduler_; std::shared_ptr<NetworkContext> network_context_; std::shared_ptr<DataFlowTracker> data_flow_tracker_; bool is_composed_ = false; }; } // namespace holoscan #endif/* HOLOSCAN_CORE_FRAGMENT_HPP */
```