Program Listing for File gxf_operator.hpp

Return to documentation for file (include/holoscan/core/gxf/gxf_operator.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_GXF_GXF_OPERATOR_HPP #define HOLOSCAN_CORE_GXF_GXF_OPERATOR_HPP #include <gxf/core/gxf.h> #include <iostream> #include <string> #include <utility> #include "../executors/gxf/gxf_parameter_adaptor.hpp" #include "../operator.hpp" #include "./gxf_utils.hpp" namespace holoscan::ops { class GXFOperator : public holoscan::Operator { public: HOLOSCAN_OPERATOR_FORWARD_TEMPLATE() explicit GXFOperator(ArgT&& arg, ArgsT&&... args) : Operator(std::forward<ArgT>(arg), std::forward<ArgsT>(args)...) { operator_type_ = holoscan::Operator::OperatorType::kGXF; } GXFOperator() : Operator() { operator_type_ = holoscan::Operator::OperatorType::kGXF; } void initialize() override; virtual const char* gxf_typename() const = 0; gxf_context_t gxf_context() const { return gxf_context_; } *// Note: now can get eid() from graph_entity_.eid()* void gxf_eid(gxf_uid_t gxf_eid) { gxf_eid_ = gxf_eid; } gxf_uid_t gxf_eid() const { return gxf_eid_; } void gxf_cid(gxf_uid_t gxf_cid) { gxf_cid_ = gxf_cid; } gxf_uid_t gxf_cid() const { return gxf_cid_; } template <typename typeT> static void register_converter() { ::holoscan::Operator::register_argument_setter<typeT>(); register_parameter_adaptor<typeT>(); } protected: gxf_uid_t add_codelet_to_graph_entity() override; void set_parameters() override; template <typename typeT> static void register_parameter_adaptor() { ::holoscan::gxf::GXFParameterAdaptor::get_instance().add_param_handler<typeT>([](gxf_context_t context, gxf_uid_t uid, const char* key, const ArgType& arg_type, const std::any& any_value) { try { auto& param = *std::any_cast<Parameter<typeT>*>(any_value); param.set_default_value(); *// set default value if not set.* if (param.has_value()) { auto& value = param.get(); switch

```
(arg_type.container_type()) { case ArgContainerType::kNative: case
ArgContainerType::kVector: { if (arg_type.element_type() ==
ArgElementType::kCustom) { YAML::Node value_node =
YAML::convert<typeT>::encode(value); return
GxfParameterSetFromYamlNode(context, uid, key, &value_node, ""); } break; } case
ArgContainerType::kArray: { HOLOSCAN_LOG_ERROR("Unable to handle
ArgContainerType::kArray type for key '{}'", key); break; } } HOLOSCAN_LOG_WARN(
"Unable to get argument for key '{}' with type '{}'", key, typeid(typeT).name()); } }
catch (const std::bad_any_cast& e) { HOLOSCAN_LOG_ERROR( "Bad any cast
exception caught for argument '{}': {}", key, e.what()); } return GXF_FAILURE; }); }
gxf_context_t gxf_context_ = nullptr; gxf_uid_t gxf_eid_ = 0; gxf_uid_t gxf_cid_ = 0;
nvidia::gxf::Handle<nvidia::gxf::Codelet> codelet_handle_; std::string gxf_typename_
= "unknown_gxf_typename"; }; } // namespace holoscan::ops #endif/*
HOLOSCAN_CORE_GXF_GXF_OPERATOR_HPP */
```