Integration Service Documentation

Release 3.1.0

eProsima

Jun 23, 2021

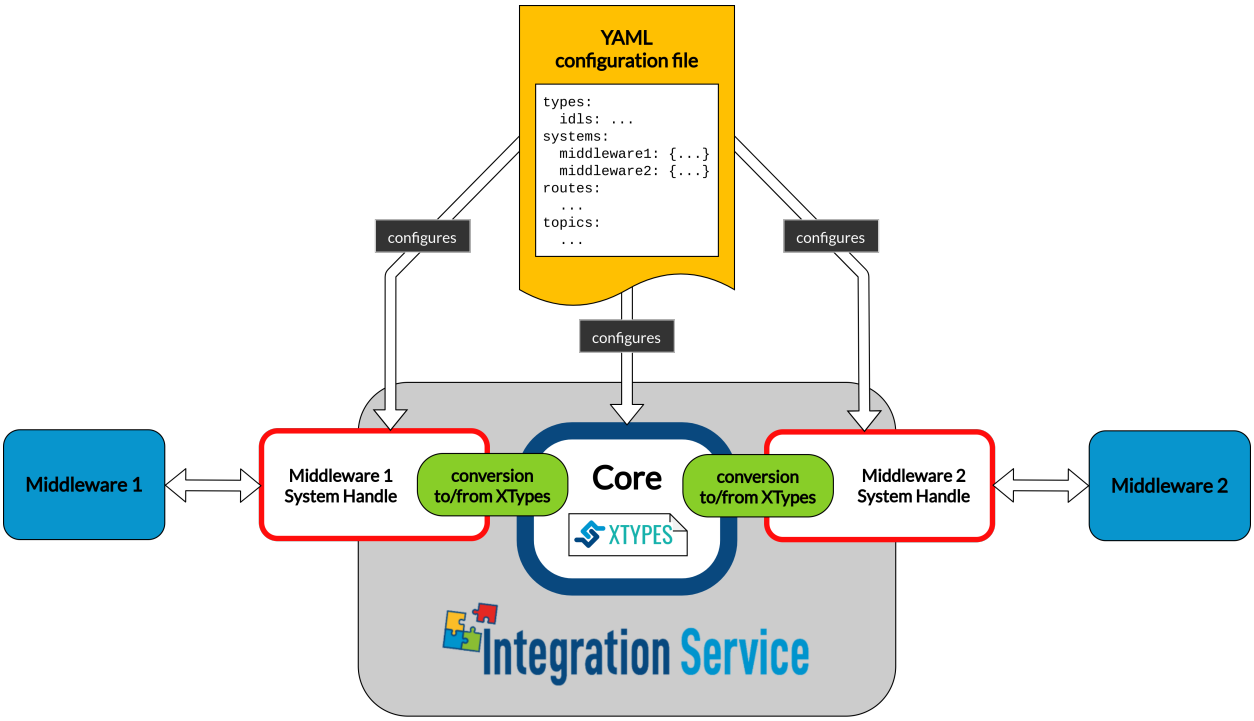
INTRODUCTION

eProsima Integration Service is a tool that enables intercommunicating an arbitrary number of protocols that speak different languages.

If one has a number of complex systems and wills to combine them to create a larger, even more complex system, *Integration Service* can act as an intermediate message-passing tool that, by speaking a common language, centralizes and mediates the integration.

The communication between the different protocols is made possible by system-specific plugins, or *System Handles*. These provide the necessary conversion between the target protocols and the common representation language spoken by *Integration Service*, based on an implementation of the xTypes specification. Once a system is communicated with the core, it enters the *Integration Service* world and can straightforwardly reach out to any other system that already exists in this world.

Integration Service is configured by means of a **YAML** text file, through which the user can provide a mapping between the topics and services handled by the middlewares of the systems involved.



Integration Service comprises the following elements:

1. The *Integration Service Core* engine.
2. The *System Handles* or plugins, for each supported protocol.
3. A *YAML Configuration* file, which follows a specific syntax.

Integration Service provides a plugin-based platform that is easily and intuitively configurable. An *Integration Service* instance can connect *N* middlewares through dedicated plugins that speak the same language as the core. This common language is eProsima xTypes; a fast and lightweight OMG DDS-XTYPES standard C++17 header-only implementation. Find more information on the core and on the *xTypes* representation language in the *Integration Service*

Core user manual of this documentation.

CHAPTER

ONE

SYSTEM HANDLES

The plugins, or **System Handles**, are discovered by *Integration Service* at runtime after they have been installed.

Available *System Handles* up-to-date are listed below:

System Handle	Repository
Fast DDS System Handle	https://github.com/eProsima/FastDDS-SH
FIWARE System Handle	https://github.com/eProsima/FIWARE-SH
ROS 1 System Handle	https://github.com/eProsima/ROS1-SH
ROS 2 System Handle	https://github.com/eProsima/ROS2-SH
WebSocket System Handle	https://github.com/eProsima/WebSocket-SH

New *System Handles* for additional protocols can be easily created, automatically allowing communication of the new protocol with the middlewares that are already supported.

The plugin-based framework is specially advantageous when it comes to integrating a new component into a complex system where the rest of sub-systems use incompatible protocols. Indeed, once all protocols of interest are communicated with the core, each via a dedicated *System Handle*, the integration happens straightforwardly. The great advantage of using *Integration Service* is that it relies on centralization rather than on the creation of dedicated bridges for each pair of components. For a system made of N components, this means that the number of new software parts to add grows as N rather than N^2.

For further information, please refer to the *System Handle specific user manual* of the documentation.

CHAPTER

TWO

YAML CONFIGURATION FILES

Integration Service is configured by means of a **YAML** file that specifies a set of compulsory fields, plus some optional ones.

This configuration approach is especially profitable when it comes to integrating large systems, since a single *YAML* file is needed no matter how many protocols are being communicated.

The strength of this approach is that different translations are possible by only changing the configuration file. This means that no compilation steps are required between each *Integration Service* instantiation, as it is configured at runtime.

Detailed information on how to configure an *Integration Service*-mediated communication via a *YAML* file can be found in the *YAML configuration user manual* of this documentation.

CHAPTER

THREE

MAIN FEATURES

1. **Free and Open Source:** The Integration Service Core, and all *System Handles available to date* are free and open source.
2. **Easily configurable:** As detailed above, an *Integration Service* instance is easily configurable by means of a *YAML* file. For more information on how to do so, please consult the *YAML Configuration* user manual of this documentation.
3. **Easy to extend to new platforms:** New platforms can easily enter the *Integration Service* world by generating the plugin, or *System Handle* needed by the core to integrate them. For more information on **System-Handles**, please consult the *System Handle user manual* of this documentation.
4. **Easy to use:** Installing and running *Integration Service* is intuitive and straightforward. Please refer to the *installation manual* to be guided through the installation process.

CHAPTER

FOUR

TYPICAL USE-CASES

Integration Service comes in handy for a varied set of application scenarios, such as:

- **Communication among systems** using different protocols which handle incompatible types, topics, and services. A complete list of the available examples described for this use-case scenario can be found *here*.
- **Integration of systems under the same protocol** which are isolated per specific protocol features. A complete list of the available examples described for this use-case scenario can be found *here*.
- **Communication through the Internet** between systems hosted by logically separated WANs located in different geographical regions. A complete list of the available examples described for this use-case scenario can be found *here*.

CHAPTER

FIVE

STRUCTURE OF THE DOCUMENTATION

This documentation is organized into the sections listed below:

- Installation Manual
- User Manual
- API Reference
- Examples
- *Release Notes*

CHAPTER

SIX

CONTACT AND COMMERCIAL SUPPORT

Find more about us at eProsima's webpage.

Support available at:

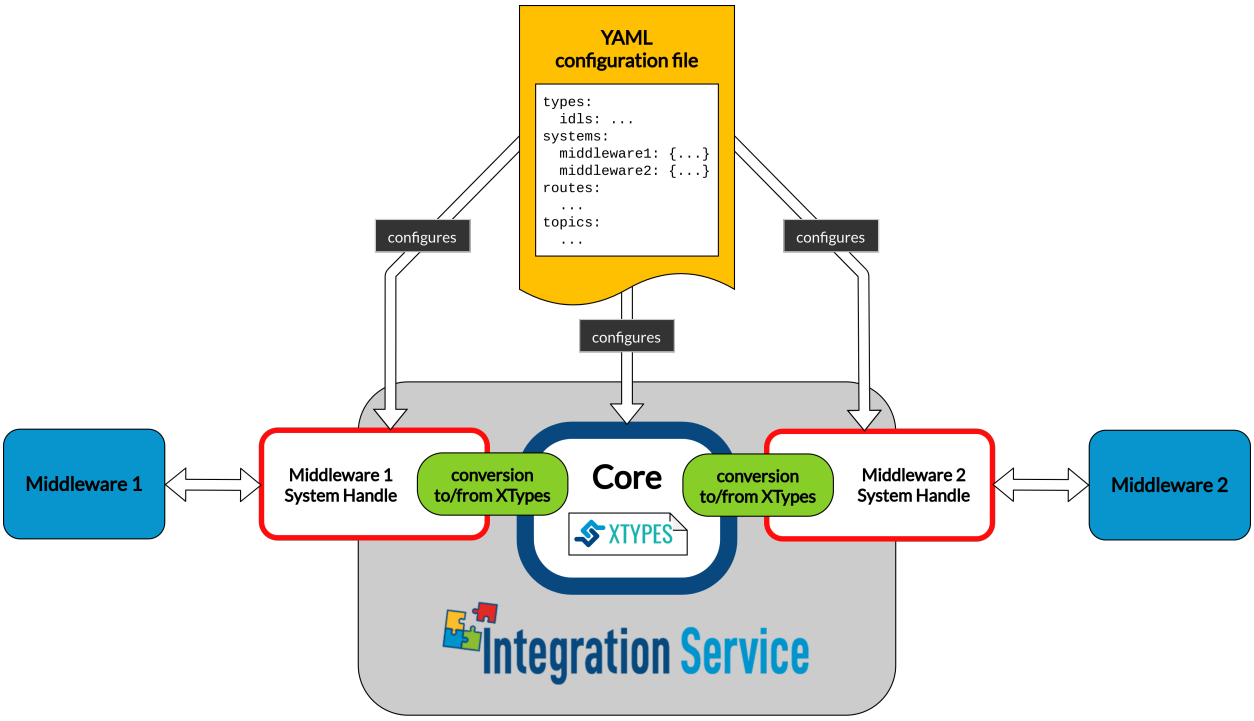
- Email: support@eprosima.com
- Phone: +34 91 804 34 48

eProsima Integration Service is a tool that enables intercommunicating an arbitrary number of protocols that speak different languages.

If one has a number of complex systems and wills to combine them to create a larger, even more complex system, *Integration Service* can act as an intermediate message-passing tool that, by speaking a common language, centralizes and mediates the integration.

The communication between the different protocols is made possible by system-specific plugins, or *System Handles*. These provide the necessary conversion between the target protocols and the common representation language spoken by *Integration Service*, based on an implementation of the xTypes specification. Once a system is communicated with the core, it enters the *Integration Service* world and can straightforwardly reach out to any other system that already exists in this world.

Integration Service is configured by means of a **YAML** text file, through which the user can provide a mapping between the topics and services handled by the middlewares of the systems involved.



Integration Service comprises the following elements:

1. The *Integration Service Core* engine.
2. The *System Handles* or plugins, for each supported protocol.
3. A *YAML Configuration* file, which follows a specific syntax.

6.1 Integration Service Core

Integration Service provides a plugin-based platform that is easily and intuitively configurable. An *Integration Service* instance can connect *N* middlewares through dedicated plugins that speak the same language as the core. This common language is eProsima xTypes; a fast and lightweight OMG DDS-XTYPES standard C++17 header-only implementation. Find more information on the core and on the *xTypes* representation language in the *Integration Service Core* user manual of this documentation.

6.2 System Handles

The plugins, or **System Handles**, are discovered by *Integration Service* at runtime after they have been installed.

Available *System Handles* up-to-date are listed below:

System Handle	Repository
Fast DDS System Handle	https://github.com/eProsima/FastDDS-SH
FIWARE System Handle	https://github.com/eProsima/FIWARE-SH
ROS 1 System Handle	https://github.com/eProsima/ROS1-SH
ROS 2 System Handle	https://github.com/eProsima/ROS2-SH
WebSocket System Handle	https://github.com/eProsima/WebSocket-SH

New *System Handles* for additional protocols can be easily created, automatically allowing communication of the new protocol with the middlewares that are already supported.

The plugin-based framework is specially advantageous when it comes to integrating a new component into a complex system where the rest of sub-systems use incompatible protocols. Indeed, once all protocols of interest are communicated with the core, each via a dedicated *System Handle*, the integration happens straightforwardly. The great advantage of using *Integration Service* is that it relies on centralization rather than on the creation of dedicated bridges for each pair of components. For a system made of N components, this means that the number of new software parts to add grows as N rather than N^2.

For further information, please refer to the *System Handle specific user manual* of the documentation.

6.3 YAML configuration files

Integration Service is configured by means of a **YAML** file that specifies a set of compulsory fields, plus some optional ones.

This configuration approach is especially profitable when it comes to integrating large systems, since a single *YAML* file is needed no matter how many protocols are being communicated.

The strength of this approach is that different translations are possible by only changing the configuration file. This means that no compilation steps are required between each *Integration Service* instantiation, as it is configured at runtime.

Detailed information on how to configure an *Integration Service*-mediated communication via a *YAML* file can be found in the *YAML configuration user manual* of this documentation.

6.4 Main features

1. **Free and Open Source:** The Integration Service Core, and all *System Handles available to date* are free and open source.
2. **Easily configurable:** As detailed above, an *Integration Service* instance is easily configurable by means of a *YAML* file. For more information on how to do so, please consult the *YAML Configuration* user manual of this documentation.
3. **Easy to extend to new platforms:** New platforms can easily enter the *Integration Service* world by generating the plugin, or *System Handle* needed by the core to integrate them. For more information on **System-Handles**, please consult the *System Handle user manual* of this documentation.
4. **Easy to use:** Installing and running *Integration Service* is intuitive and straightforward. Please refer to the *installation manual* to be guided through the installation process.

6.5 Typical use-cases

Integration Service comes in handy for a varied set of application scenarios, such as:

- **Communication among systems** using different protocols which handle incompatible types, topics, and services. A complete list of the available examples described for this use-case scenario can be found *here*.
- **Integration of systems under the same protocol** which are isolated per specific protocol features. A complete list of the available examples described for this use-case scenario can be found *here*.

- **Communication through the Internet** between systems hosted by logically separated WANs located in different geographical regions. A complete list of the available examples described for this use-case scenario can be found *here*.

6.6 Structure of the documentation

This documentation is organized into the sections listed below:

- Installation Manual
- User Manual
- API Reference
- Examples
- *Release Notes*

6.7 Contact and commercial support

Find more about us at eProsima's webpage.

Support available at:

- Email: support@eprosima.com
- Phone: +34 91 804 34 48

6.8 Dependencies

On this page, we provide a list of the dependencies required for an *Integration Service* instance to function. To do so, we distinguish between those requirements that are common to all the repositories, the ones regarding the *Integration Service Core* and those of each *System Handle*.

Dependency	Description	Installation
CMake	At least version *3.5* is required to build the project files.	`apt install cmake`
C++	*eProsima Integration-Service* uses standard C++14.	`apt install build-essential`
colcon	Command line tool to build and test multiple software packages.	Colcon installation guide

6.8.1 Core

The *Integration Service Core* has the following requirements:

Dependency	Description	Installation
YAML-cpp	*YAML* parser and emitter in C++.	`apt install libyaml-cpp-dev`
Boost program options	Allows obtaining name-value pairs from the config file.	`apt install libboost-program-options-dev`

Note: eProsima xTypes is an additional dependency but it is not necessary to install it, since if the Integration Service Core repository is cloned using the `--recursive` option, it is downloaded automatically.

6.8.2 System Handles

Beyond the dependencies of the core, each **System Handle** has its own specific dependencies.

The *Fast DDS System Handle* has the following requirements:

Dependency	Description	Installation
Fast DDS (v2.0.0 or superior)	eProsima C++ implementation for *DDS*.	Binaries installation guide Sources installation guide

The *FIWARE System Handle* has the following requirements:

Dependency	Description	Installation
Asio C++ Library	C++ library for network and low-level I/O programming.	`apt install libasio-dev`
cURLpp library	C++ wrapper for *libcURL*.	`apt install libcurlpp-dev`
cURL library	Command-line tool for getting or sending data using *URL* syntax.	`apt install libcurl4-openssl-dev`

The *ROS 1 System Handle* has the following requirements:

Dependency	Description	Installation
ROS 1	*Melodic/Noetic ROS 1* distribution.	Melodic installation guide Noetic installation guide

The *ROS 2 System Handle* has the following requirements:

Dependency	Description	Installation
ROS 2	*Foxy/Galactic ROS 2* distribution.	Foxy installation guide Galactic installation guide

The *WebSocket System Handle* has the following requirements:

Dependency	Description	Installation
OpenSSL	Toolkit for *TLS* and *SSL* protocols.	`apt install libssl-dev`
WebSocket++	*WebSocket* Protocol C++ library implementation.	`apt install libwebsocketpp-dev`

6.9 Installation

This section provides the user with an easy-to-follow installation guide of both the *Integration Service* and of the *System Handles*, and an explanation of how to launch and deploy an *Integration Service* project.

Important: Before following the installation steps, check that you have all the necessary *dependencies* installed.

6.9.1 Workspace Setup

This section explains step by step the workspace configuration required to use *Integration Service*. It is divided into two subsections, which describe the configuration of the *Integration Service Core* and the *System Handles* respectively.

Core

The *Integration Service* core consist of many *CMake* packages which can be configured and built manually, but we recommend to use colcon, as it makes the job much smoother.

The starting point is to create a *colcon workspace* and clone the Integration-Service repository, containing the core. To do so, follow the next instructions:

```
mkdir ~/is-workspace
cd ~/is-workspace
git clone https://github.com/eProsima/Integration-Service.git src/Integration-Service␣
↪--recursive
```

At this point, you have the *Integration Service* library correctly cloned into your `is-workspace/src/Integration-Service` folder.

Note: The `--recursive` flag is needed to correctly initialize the *xTypes* library as a submodule.

System Handles

As discussed in the *Introduction* section, *Integration Service* allows to bring an arbitrary number of middlewares into communication, each integrated into the core with a dedicated *System Handle*.

The workflow is thus dependent on the middlewares involved in the desired communication. The up-to-date list of the available *System Handles* and the repositories hosting them is provided in the *Built-in System Handles* section.

Depending on the use-case, you might need to have either one, two, or more *System Handles* installed. In the examples section, you can find a collection of relevant examples clarifying how to use these plugins according to your needs.

You will have to clone the repositories of the desired *System Handles* into the previously created `is-workspace`:

```
cd ~/is-workspace
git clone https://github.com/eProsima/<middleware_1-SH>.git src/middleware_1-SH
...
git clone https://github.com/eProsima/<middleware_N_SH>.git src/middleware_2-SH
```

Where `<middleware_i-SH>`, with *i = 1, .., N* refers to the *i*-th *System Handle* needed for carrying out the integration, chosen among the ones listed in the *Built-in System Handles* section. Each such *System Handle* will be cloned in a dedicated `src/middleware_i-SH` folder inside your `is-workspace`.

Note: If using a custom *System Handle* which is not present in the *eProsima* GitHub organization, clone the dedicated repository into the `is-workspace`.

6.9.2 Build

Once all the necessary packages have been cloned, they need to be built. To do so, execute from the `is-workspace`:

```
colcon build <COMPILATION_FLAGS>
```

Note: `<COMPILATION_FLAGS>` refers to the optional flags used to configure *Integration Service*. For further details refers to the *Global compilation flags* section.

Once that's finished building and before launching your *Integration Service* project, you need to source the new *colcon* overlay:

```
source install/setup.bash
```

Warning: If the *ROS 1* and *ROS 2 System Handles* coexist in your *Integration Service* workspace, please notice that the building process must be split into two steps, due to incompatibility between *ROS* distros:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
colcon build --packages-skip-regex is-ros1 <COMPILATION_FLAGS>
source /opt/ros/$<ROS1_DISTRO>/setup.bash
colcon build <COMPILATION_FLAGS>
```

Global compilation flags

Integration Service uses *CMake* for building and packaging the project. There are several *CMake* flags, which can be tuned during the configuration step:

- `BUILD_LIBRARY`: This compilation flag can be used to completely disable the compilation of the *Integration Service* set of libraries, that is, the *Integration Service Core* and all the existing *System Handles* existing in the *colcon* workspace. It is enabled by default.

  This flag is useful, for example, to speed up the documentation generation process, when building the api_reference from the *Doxygen* source code comments.

  ```
  ~/is_ws$ colcon build --cmake-args -DBUILD_LIBRARY=OFF
  ```

- `BUILD_API_REFERENCE`: It is used to generate all the necessary files for building the api_reference section of this documentation, starting from the source code comments written using a *Doxygen*-like format. It is disabled by default; to use it:

  ```
  ~/is_ws$ colcon build --cmake-args -DBUILD_API_REFERENCE=ON
  ```

- `BUILD_TESTS`: When compiling *Integration Service*, use the `-DBUILD_TESTS=ON` CMake option to compile both the unitary tests for the *Integration Service Core* and the unitary and integration tests for all the *System Handles* present in the *colcon* workspace:

```
~/is_ws$ colcon build --cmake-args -DBUILD_TESTS=ON
```

- `BUILD_EXAMPLES`: Allows to compile all the utilities that can be used for the several provided usage examples for *Integration Service*, located under the examples/utils folder of the core repository. These applications can be used to test the *Integration Service* with some of the provided *YAML* configuration files, which are located under the examples/basic directory of the core repository:

```
~/is_ws$ colcon build --cmake-args -DBUILD_EXAMPLES=ON
```

Note: To use this flag, all the examples dependencies need to be installed.

- `BUILD_FASTDDS_EXAMPLES`: Allows to compile the *Fast DDS* utilities that can be used for several of the provided usage examples for *Integration Service*, located under the examples/utils/dds folder. These applications can be used to test the *Integration Service* with some of the provided *YAML* configuration files, which are located under the examples/basic directory of the core repository:

```
~/is_ws$ colcon build --cmake-args -DBUILD_FASTDDS_EXAMPLES=ON
```

Note: To compile these examples you need to have Fast DDS (v.2.0.0 or superior) and its dependencies installed.

To date, the following *Fast DDS* user application examples and utility packages are available:

  - `DDSHelloWorld`: A simple publisher/subscriber C++ application, running under *Fast DDS*. It publishes or subscribes to a simple string topic, named *HelloWorldTopic*.

    The resulting executable will be located inside the `build/is-examples/dds` folder, and named `DDSHelloWorld`. Please execute `DDSHelloWorld -h` to see a full list of supported input parameters.

  - `DDSAddTwoInts`: A simple server/client C++ application, running under *Fast DDS*. It allows performing service requests and replies to a service named *AddTwoIntsService*, which consists of two integer numbers as request type and answers with a single value, indicating the sum of them.

    The resulting executable will be located inside the `build/is-examples/dds` folder, and named `DDSAddTwoInts`. Please execute `DDSAddTwoInts -h` to see a full list of supported input parameters.

- `BUILD_ROS1_EXAMPLES`: Allows to compile the *ROS 1* utilities that can be used for several of the provided usage examples for *Integration Service*, located under the examples/utils/ros1 folder. These applications can be used to test the *Integration Service* with some of the provided *YAML* configuration files, which are located under the examples/basic directory of the core repository:

```
~/is_ws$ colcon build --cmake-args -DBUILD_ROS1_EXAMPLES=ON
```

Note: In order to compile this example you need to have *ROS 1* (Melodic or superior) installed and sourced, and the *Integration Service* `example_interfaces` ROS 1 package compiled.

To date, the following *ROS 1* user application examples and utility packages are available:

  - `add_two_ints_server`: A simple C++ server application, running under *ROS 1*. It listens to requests coming from *ROS 1* clients and produces an appropriate answer for them; specifically, it is capable of

listening to a *ROS 1* service called `add_two_ints`, which consists of two integer numbers as request type and answers with a single value, indicating the sum of them.

The resulting executable will be located inside the `build/devel/lib/add_two_ints_server` folder, and named `add_two_ints_server_node`.

- `example_interfaces`: *ROS 1* package containing the service type definitions for the *AddTwoInts* services examples, for which the *ROS 1* type support files will be automatically generated. As specified in the *services examples tutorials*, it must be compiled and installed in the system, using `catkin`:

```
~/is_ws$ cd examples/utils/ros1/catkin_ws/
~/is_ws/examples/utils/ros1/catkin_ws$ catkin_make -DBUILD_EXAMPLES=ON -
↪DCMAKE_INSTALL_PREFIX=/opt/ros/$<ROS1_DISTRO> install
```

- `BUILD_WEBSOCKET_EXAMPLES`: Allows to compile the *WebSocket* utilities that can be used for several of the provided usage examples for *Integration Service*, located under the examples/utils/websocket folder. These applications can be used to test the *Integration Service* with some of the provided *YAML* configuration files, which are located under the examples/basic directory of the core repository:

```
~/is_ws$ colcon build --cmake-args -DBUILD_WEBSOCKET_EXAMPLES=ON
```

Note: In order to compile this example you need to have *OpenSSL* and *WebSocket++* installed.

To date, the following *WebSocket* user application examples and utility packages are available:

- `WebSocketAddTwoInts`: A simple server/client C++ application, running under *WebSocket++*. It allows performing service requests and replies to a service named *add_two_ints*, which consists of two integer numbers as request type and answers with a single value, indicating the sum of them.

  The resulting executable will be located inside the `build/is-examples/websocket` folder, and named `DDSAddTwoInts`. Please execute `WebSocketAddTwoInts -h` to see a full list of supported input parameters.

6.9.3 Deployment

The `is-workspace` is now prepared for running an *Integration Service* instance.

The communication can be configured using a *YAML* file as explained in section *YAML Configuration*. Once created, it is passed to *Integration Service* with the following instruction:

```
integration-service <config.yaml>
```

As soon as *Integration Service* is initiated, the desired protocols can be communicated by launching them in independent terminal windows. To get a better taste of how to do so, refer to the examples section, which provides several examples of how to connect instances of systems that are already integrated into the *Integration Service* ecosystem.

Note: The sourcing of the local *colcon* overlay is required every time the *colcon* workspace is opened in a new shell environment. As an alternative, you can copy the source command with the full path of your local installation to your `.bashrc` file as:

```
source /PATH-TO-YOUR-IS-WORKSPACE/is-workspace/install/setup.bash
```

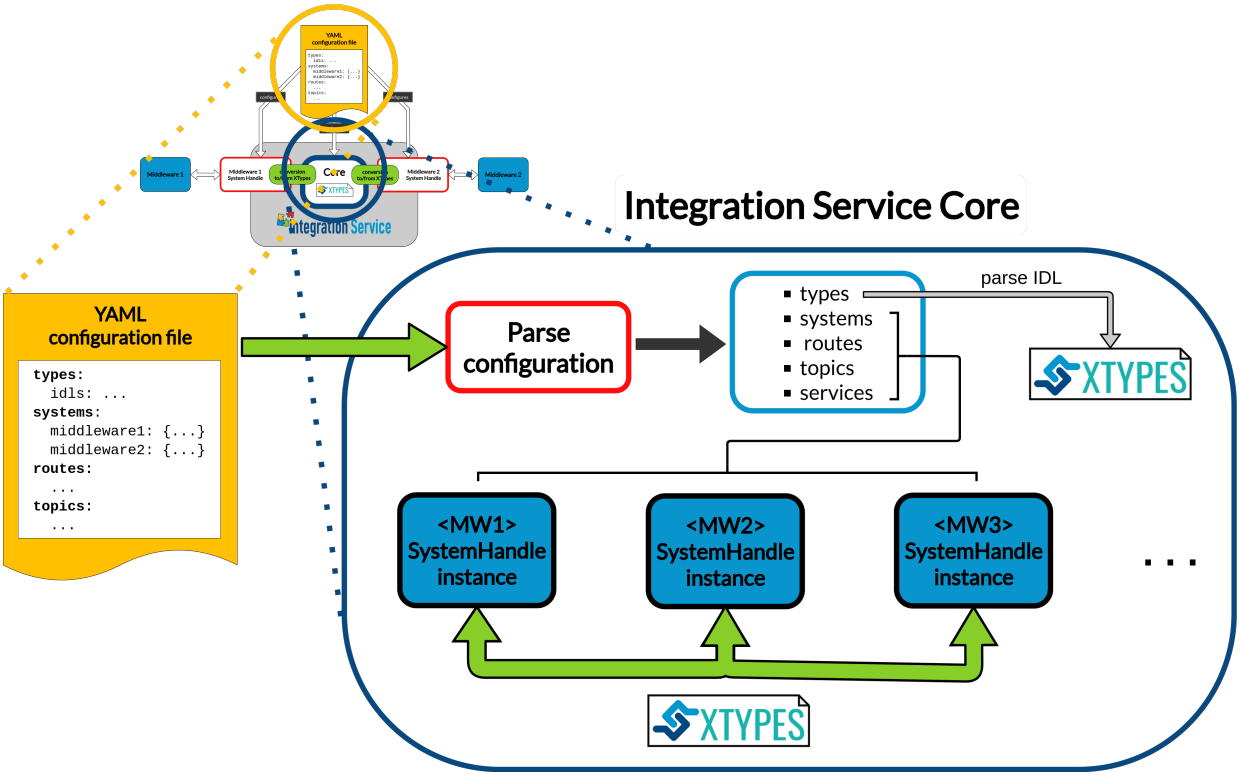
6.10 Integration Service Core

The `is-core` library defines a set of abstract interfaces and provides some utility classes that form a plugin-based framework.

A single `integration-service` executable instance can connect *N* middlewares, where each middleware has a plugin, or *System Handle* associated with it.

The *System Handle* for a middleware is a lightweight wrapper around that middleware (e.g. a *ROS* node or a *WebSocket* server/client). The `is-core` library provides *CMake* functions that allow these middleware *System Handles* to be discovered by the `integration-service` executable at runtime after the *System Handle* has been installed.

A single `integration-service` instance can route any number of topics or services to/from any number of middlewares. Because of this, downstream users can extend *Integration Service* to communicate with any middleware.



According to the diagram depicted above, the *Integration Service Core* executes the following steps:

1. **Parse** the *YAML* configuration file. This file must contain everything needed to successfully launch an *Integration Service* instance. To get a detailed view on how to write a configuration file for the *Integration Service*, please refer to the *YAML Configuration* section of this documentation.

2. If the configuration parsing process ended successfully, **IDL types are parsed** and registered within the *Integration Service Core* type registry database. This will allow to later define the required *topics* and *services* types that will take part in the intercommunication process.

   The *IDL* parsing procedure is fulfilled thanks to the built-in parser provided with the xTypes library. More information about how this library works and why it is extremely useful for *Integration Service* can be found in the section *below*.

3. According to the specified **systems**, the corresponding *System Handles* dynamic libraries are loaded.

   Please take into account that each *system* type must match on of those supported by *Integration Service*. A table with every built-in provided *System Handle* and their corresponding source code GitHub repositories can be

found *here*.

If a user wants to incorporate a new protocol into the *Integration Service* ecosystem to use it in his or her *Integration Service* application instance, the specific *System Handle* must be implemented first. Please refer to the *System Handle implementation* section of the documentation.

4. Next, the **routes** are processed, and links between the routed *systems* are established.

   For example, if a route consists on establishing a link `from` *Middleware_1* `to` *Middleware_2*, the *Integration Service Core* will internally register:

   - A *Middleware_1* **TopicSubscriber**.
   - A *Middleware_2* **TopicPublisher**.

   The *Middleware_1 TopicSubscriber* will be listening to a certain *Middleware_1 topic publisher*. This subscriber registers a callback that internally converts the middleware-specific data type instance into a *Dynamic Data*, using *xTypes* for that purpose, and forwards the converted data instance to the *Middleware_2 TopicPublisher*, which will then publish it so that it can be consumed by the final endpoint destination, that is, a *Middleware_2 subscriber*.

5. For the defined **topics** and **services**, the topic/service name are registered within the *Integration Service Core*, prior to check that the specified **type** exists and has been previously registered within the type registry. Each topic/service must use one of the provided *routes* in the configuration file.

If all these steps are correctly fulfilled, an *Integration Service* instance is launched and starts listening for incoming messages to translate them into the specified protocols. The green arrow on the diagram depicts this behavior. Notice that *xTypes* is the common language representation used for transmitting the data among *System Handles*, so we will introduce this library right away.

6.10.1 The xTypes library

eProsima xTypes is a fast and lightweight C++17 header-only implementation of the OMG DDS-xTypes standard.

This library allows to create *Dynamic Type* representations at runtime, by means of feeding the provided parser with an IDL type definition. For example, given the following *IDL*:

```
struct Inner {
    long a;
};

struct Outer {
    long b;
    Inner c;
};
```

xTypes provides with an easy and intuitive API to retrieve the structured dynamic type:

```
xtypes::idl::Context context = idl::parse(my_idl);
const xtypes::StructType& inner = context.module().structure("Inner");
const xtypes::StructType& outer = context.module().structure("Outer");
```

The *Integration Service Core* uses xTypes as the common representation language for transmitting information between each *System Handle* instance that is desired to establish a communication between. To do so, *System Handles* must provide a way to convert their specific data types instances into/from *xTypes*. An example on how this procedure would look like for a *System Handle*, that is, the *FastDDS System Handle*, can be found here.

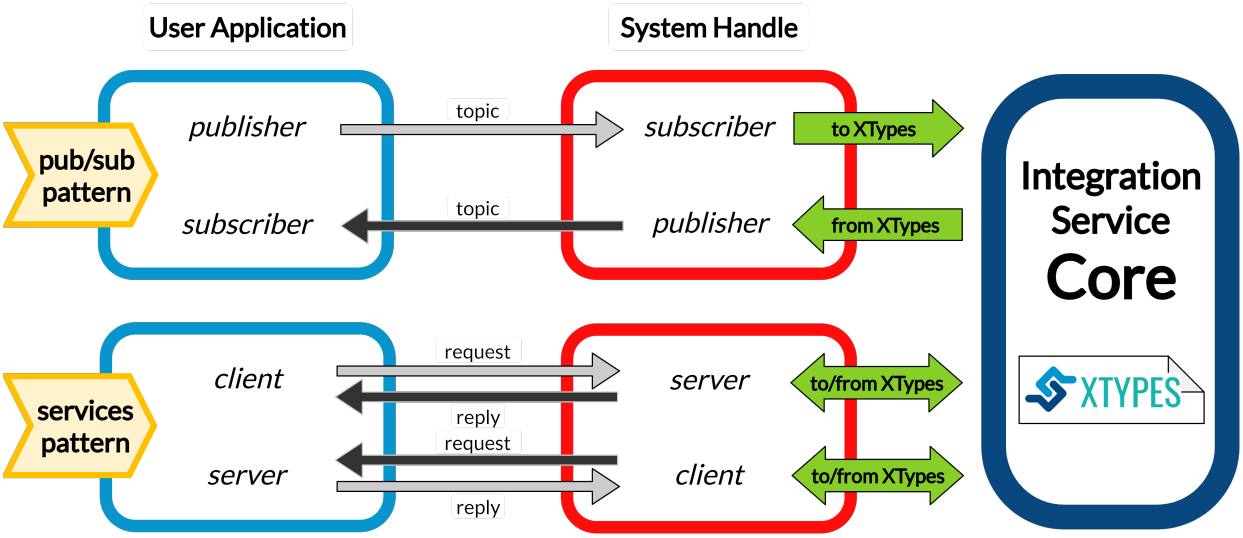
6.10.2 API Reference

The *Integration Service API Reference* constitutes an independent section within this documentation. To access the *Integration Service Core* subsection, use this *link*.

6.11 System Handle

As explained in the *Introduction*, a single `integration-service` instance can route any number of topics or services to/from any number of middlewares.

This occurs through the use of *System Handles*, which are system-specific plugins that allows a certain middleware or communication protocol to speak the same language used by the *Integration Service*, that is, Extensible and Dynamic Topic Types for DDS (xTypes).



6.11.1 Built-in System Handles

This section provides an insight over the existing built-in *System Handles* provided along with *Integration Service* for connecting the core with the following middlewares or communication protocols:

Fast DDS System Handle

The *Fast DDS System Handle* can be used for three main purposes:

- Connection between a *DDS* application and an application running over a different middleware implementation. This is the classic use-case for *Integration Service*.
- Connecting two *DDS* applications running under different Domain IDs.
- Creating a *TCP tunnel*, by running an *Integration Service* instance on each of the machines you want to establish a communication between.

Dependencies

The only dependency of this *System Handle* is to have a Fast DDS installation (v2.0.0 or superior) in your system.

Note: The *Fast DDS System Handle* requires an installation of *Fast DDS* to work. The *System Handle* first looks into the system for a previous installation of *Fast DDS* v2.0.0 or superior. If it doesn't find one, it downloads and installs its own version.

Configuration

Regarding the *Fast DDS System Handle*, there are several specific parameters which can be configured for the *DDS* middleware. All of these parameters are optional, and are suboptions of the main five sections:

- `systems`: The system `type` must be `fastdds`. In addition to the `type` and `types-from` fields, the *Fast DDS System Handle* accepts the following specific configuration fields:

```
systems:
    dds:
    type: fastdds
    participant:
        domain_id: 3
        file_path: <path_to_xml_profiles_file>.xml
        profile_name: fastdds-sh-participant-profile
```

  - `participant`: Allows to add a specific configuration for the Fast DDS DomainParticipant:
    - `domain_id`: Provides an easy way to change the *Domain ID* of the DDS entities created by the *Fast DDS System Handle*.
    - `file_path`: Path to an *XML* file, containing a configuration profile for the *System Handle* participant. More information about Fast DDS *XML* profiles and how to fully customize the properties of DDS entities through them is available here.
    - `profile_name`: Within the provided *XML* file, the name of the *XML* profile associated to the *Integration Service Fast DDS System Handle* participant.

Examples

There are several examples that you can find in this documentation in which the *Fast DDS System Handle* is employed in the communication process. Some of them are presented here:

- *DDS - ROS 2 bridge*
- *DDS Service Server*
- *DDS Domain ID change*
- *WAN-TCP tunneling over DDS*

Compilation flags

Besides the *Global compilation flags* available for the whole *Integration Service* product suite, there are some specific flags which apply only to the *Fast DDS System Handle*. They are listed below:

- `BUILD_FASTDDS_TESTS`: Allows to specifically compile the *Fast DDS System Handle* unitary and integration tests. It is useful to avoid compiling each *System Handle*'section test suite present in the `colcon` workspace, which is what would happen if using the `BUILD_TESTS` flag, with the objective of minimizing building time. To use it, after making sure that the *Fast DDS System Handle* is present in your `colcon` workspace, execute the following command:

```
~/is_ws$ colcon build --cmake-args -DBUILD_FASTDDS_TESTS=ON
```

API Reference

The *Integration Service API Reference* constitutes an independent section within this documentation. To access the *Fast DDS System Handle* subsection, use this *link*.

FIWARE System Handle

This repository contains the source code of the *Integration Service System Handle* for the FIWARE middleware protocol, widely used in the robotics field.

The main purpose of the *FIWARE System Handle* is that of establishing a connection between a *FIWARE's Context Broker* and an application running over a different middleware implementation. This is the classic use-case for *Integration Service*.

Dependencies

The dependencies of the *FIWARE System Handle* are:

- Asio C++ Library
- cURLpp library
- cURL library

Configuration

Regarding the *FIWARE System Handle*, there are several specific parameters which must be configured for the *FIWARE* middleware. These parameters are mandatory, and are suboptions of the main five sections:

- `systems`: The system `type` must be `fiware`. In addition to the `type` and `types-from` fields, the *FIWARE System Handle* accepts some specific configuration fields:

```
systems:
   fiware:
       type: fiware
       host: localhost
       port: 1026
```

- `port`: The specific port where the *FIWARE's Context Broker* will listen for incoming connections. This field is required.

- `host`: The IP address of the *FIWARE's Context Broker*. This field is required.

Examples

There is one example that you can find in this documentation in which the *FIWARE System Handle* is employed in the communication process:

- *FIWARE - ROS 2 bridge*

Compilation flags

Besides the *Global compilation flags* available for the whole *Integration Service* product suite, there are some specific flags which apply only to the *FIWARE System Handle*. They are listed below:

- `BUILD_FIWARE_TESTS`: Allows to specifically compile the *FIWARE System Handle* unitary and integration tests. It is useful to avoid compiling each *System Handle*'section test suite present in the `colcon` workspace, which is what would happen if using the `BUILD_TESTS` flag, with the objective of minimizing building time. To use it, after making sure that the *FIWARE System Handle* is present in your `colcon` workspace, execute the following command:

```
~/is_ws$ colcon build --cmake-args -DBUILD_FIWARE_TESTS=ON
```

ROS 1 System Handle

The main purpose of the *ROS 1 System Handle* is that of establishing a connection between a *ROS 1* application and an application running over a different middleware implementation. This is the classic use-case for *Integration Service*.

Dependencies

The only dependency of this *System Handle* is to have a ROS 1 installation (Melodic or Noetic) in your system.

Configuration

Regarding the *ROS 1 System Handle*, there are several specific parameters which can be configured for the *ROS 1* middleware. All of these parameters are optional, and are suboptions of the main five sections:

- `systems`: The system `type` must be `ros1`. In addition to the `type` and `types-from` fields, the *ROS 1 System Handle* accepts the following specific configuration fields:

```
systems:
  ros1:
    type: ros1
    node_name: "my_ros1_node"
```

  - `node_name`: The *ROS 1 System Handle* node name.

- `topics`: The topic `route` must contain `ros1` within its `from` or `to` fields. Additionally, the *ROS 1 System Handle* accepts the following topic specific configuration parameters, within the `ros1` specific middleware configuration tag:

```
routes:
  ros2_to_ros1: { from: ros2, to: ros1 }
  ros1_to_dds: { from: ros1, to: dds }

topics:
  hello_ros1:
    type: std_msgs/String
    route: ros2_to_ros1
    ros1: { queue_size: 10, latch: false }
  hello_dds:
    type: std_msgs/String
    route: ros1_to_dds
    ros1: { queue_size: 5 }
```

  - `queue_size`: The maximum message queue size for the *ROS 1* publisher or subscription.
  - `latch`: Enable or disable latching. When a connection is latched, the last message published is saved and sent to any future subscribers that connect. This configuration parameter only makes sense for *ROS 1* publishers, so it is only useful for routes where the *ROS 1 System Handle* acts as a publisher, that is, for routes where `ros1` is included in the `to` list.

Examples

There are several examples that you can find in this documentation in which the *ROS 1 System Handle* is employed in the communication process. Some of them are presented here:

- *ROS 1 - ROS 2 bridge*
- *ROS 1 Service Server*

Compilation flags

Besides the *Global compilation flags* available for the whole *Integration Service* product suite, there are some specific flags which apply only to the *ROS 1 System Handle*; they are listed below:

- `BUILD_ROS1_TESTS`: Allows to specifically compile the *ROS 1 System Handle* unitary and integration tests. It is useful to avoid compiling each *System Handle*'section test suite present in the `colcon` workspace, which is what would happen if using the `BUILD_TESTS` flag, with the objective of minimizing building time. To use it, after making sure that the *ROS 1 System Handle* is present in your `colcon` workspace, execute the following command:

```
~/is_ws$ colcon build --cmake-args -DBUILD_ROS1_TESTS=ON
```

- `MIX_ROS_PACKAGES`: It accepts as an argument a list of ROS packages, such as `std_msgs`, `geometry_msgs`, `sensor_msgs`, `nav_msgs`... for which the required transformation library to convert the specific *ROS 1* type definitions into *xTypes*, and the other way around, will be built. This list is shared with the ROS 2 System Handle, meaning that the ROS packages specified in the *MIX_ROS_PACKAGES* variable will also be built for *ROS 2* if the corresponding *System Handle* is present within the *Integration Service* workspace. To avoid possible errors, if a certain package is only present in *ROS 1*, the *MIX_ROS1_PACKAGES* flag must be used instead.

  These transformation libraries are also known within the *Integration Service* context as `Middleware Interface Extension` or `mix` libraries.

By default, only the `std_msgs_mix` library is compiled, unless the `BUILD_TESTS` or `BUILD_ROS1_TESTS` is used, case in which some additional ROS 1 packages `mix` files required for testing will be built.

If the user wants to compile some additional packages to use them with *Integration Service*, the following command must be launched to compile it, adding as much packages to the list as desired:

```
~/is_ws$ colcon build --cmake-args -DMIX_ROS_PACKAGES="std_msgs geometry_msgs␣
↪sensor_msgs nav_msgs"
```

- `MIX_ROS1_PACKAGES`: It is used just as the *MIX_ROS_PACKAGES* flag, but will only affect *ROS 1*; this means that the *mix* generation engine will not search within the *ROS 2* packages, allowing to compile specific *ROS 1* packages independently.

  For example, if a user wants to compile a certain package *dummy_msgs* independently from *ROS 1*, but compiling *std_msgs* and *geometry_msgs* for both the *ROS 1* and *ROS 2 System Handles*, the following command should be executed:

```
~/is_ws$ colcon build --cmake-args -DMIX_ROS_PACKAGES="std_msgs geometry_msgs" -
↪DMIX_ROS2_PACKAGES="dummy_msgs"
```

API Reference

The *Integration Service API Reference* constitutes an independent section within this documentation. To access the *ROS 1 System Handle* subsection, use this *link*.

ROS 2 System Handle

The *ROS 2 System Handle* can be used for two main purposes:

- Connection between a *ROS 2* application and an application running over a different middleware implementation. This is the classic use-case for *Integration Service*.
- Connecting two *ROS 2* applications running under different Domain IDs.

Dependencies

The only dependency of this *System Handle* is to have a ROS 2 installation (Foxy or superior) in your system.

Configuration

Regarding the *ROS 2 System Handle*, there are several specific parameters which can be configured for the *ROS 2* middleware. All of these parameters are optional, and are suboptions of the main five sections:

- `systems`: The system `type` must be `ros2`. In addition to the `type` and `types-from` fields, the *ROS 2 System Handle* accepts the following specific configuration fields:

```
systems:
  ros2:
    type: ros2
    namespace: "/"
    node_name: "my_ros2_node"
    domain: 4
```

- `namespace`: The *namespace* of the ROS 2 node created by the *ROS 2 System Handle*.
- `node_name`: The *ROS 2 System Handle* node name.
- `domain`: Provides with an easy way to change the *Domain ID* of the ROS 2 entities created by the *ROS 2 System Handle*.

Examples

There are several examples that you can find in this documentation in which the *ROS 2 System Handle* is employed in the communication process. Some of them are presented here:

- *ROS 1 - ROS 2 bridge*
- *DDS - ROS 2 bridge*
- *ROS 2 - WebSocket bridge*
- *FIWARE - ROS 2 bridge*
- *ROS 2 Service Server*
- *ROS 2 Domain ID change*

Compilation flags

Besides the *Global compilation flags* available for the whole *Integration Service* product suite, there are some specific flags which apply only to the *ROS 2 System Handle*; they are listed below:

- `BUILD_ROS2_TESTS`: Allows to specifically compile the *ROS 2 System Handle* unitary and integration tests. It is useful to avoid compiling each *System Handle*'section test suite present in the `colcon` workspace, which is what would happen if using the `BUILD_TESTS` flag, with the objective of minimizing building time. To use it, after making sure that the *ROS 2 System Handle* is present in your `colcon` workspace, execute the following command:

```
~/is_ws$ colcon build --cmake-args -DBUILD_ROS2_TESTS=ON
```

- `IS_ROS2_DISTRO`: This flag is intended to select the *ROS 2* distro that should be used to compile the *ROS 2 System Handle*. If not set, the version will be retrieved from the last *ROS distro* sourced in the compilation environment; this means that if the last *ROS* environment sourced corresponds to *ROS 1*, the compilation process will stop and warn the user about it.
- `MIX_ROS_PACKAGES`: It accepts as an argument a list of ROS packages, such as `std_msgs`, `geometry_msgs`, `sensor_msgs`, `nav_msgs`... for which the required transformation library to convert the specific *ROS 2* type definitions into *xTypes*, and the other way around, will be built. This list is shared with the ROS 1 System Handle, meaning that the ROS packages specified in the *MIX_ROS_PACKAGES* variable will also be built for *ROS 1* if the corresponding *System Handle* is present within the *Integration Service* workspace. To avoid possible errors, if a certain package is only present in *ROS 2*, the *MIX_ROS2_PACKAGES* flag must be used instead.

  These transformation libraries are also known within the *Integration Service* context as `Middleware Interface Extension` or `mix` libraries.

  By default, only the `std_msgs_mix` library is compiled, unless the `BUILD_TESTS` or `BUILD_ROS2_TESTS` is used, case in which some additional ROS 2 packages `mix` files required for testing will be built.

  If the user wants to compile some additional packages to use them with *Integration Service*, the following command must be launched to compile it, adding as much packages to the list as desired:

```
~/is_ws$ colcon build --cmake-args -DMIX_ROS_PACKAGES="std_msgs geometry_msgs␣
↪sensor_msgs nav_msgs"
```

- `MIX_ROS2_PACKAGES`: It is used just as the *MIX_ROS_PACKAGES* flag, but will only affect *ROS 2*; this means that the *mix* generation engine will not search within the *ROS 1* packages, allowing to compile specific *ROS 2* packages independently.

  For example, if a user wants to compile a certain package *dummy_msgs* independently from *ROS 2*, but compiling *std_msgs* and *geometry_msgs* for both the *ROS 1* and *ROS 2 System Handles*, the following command should be executed:

```
~/is_ws$ colcon build --cmake-args -DMIX_ROS_PACKAGES="std_msgs geometry_msgs" -
↪DMIX_ROS2_PACKAGES="dummy_msgs"
```

API Reference

The *Integration Service API Reference* constitutes an independent section within this documentation. To access the *ROS 2 System Handle* subsection, use this *link*.

WebSocket System Handle

This repository contains the source code of *Integration Service System Handle* for the WebSocket middleware protocol, widely used in the robotics field. The main purpose of the *WebSocket System Handle* is that of establishing a connection between a *WebSocket* application and an application running over a different middleware implementation. This is the classic use-case for *Integration Service*.

Dependencies

The dependencies of the *WebSocket System Handle* are:

- OpenSSL
- WebSocket++

Configuration

Regarding the *WebSocket System Handle*, there are several specific parameters which can be configured for the WebSocket middleware. All of these parameters are suboptions of the main five sections:

- `systems`: The system `type` must be `websocket_server` or `websocket_client`. In addition to the `type` and `types-from` fields, the *WebSocket System Handle* accepts a wide variety of specific configuration fields, depending on the selected operation mode (*Client* or *Server*).

  For the `websocket_server` *System Handle*, there are two possible configuration scenarios: the former one uses a TLS endpoint, and the latter uses a TCP endpoint.

  TLS

```
systems:
  websocket:
    type: websocket_server
    port: 80
```

(continues on next page)

(continued from previous page)

```
    cert: path/to/cert/file.crt
    key: path/to/key/file.key
    authentication:
      policies: [
        { secret: this-is-a-secret, algo: HS256, rules: {example: "*regex*
↪"} }
      ]
```

TCP

```
systems:
  websocket:
    type: websocket_server
    port: 80
    security: none
    encoding: json
```

- `port`: The specific port where the *server* will listen for incoming connections. This field is required.
- `security`: If this field is not present, a secure TLS endpoint will be created. If the special value `none` is written, a TCP *WebSocket server* will be set up.
- `cert`: The *X.509* certificate that the *server* should use. This field is mandatory if `security` is enabled.
- `key`: A path to the file containing the public key used to verify credentials with the specified certificate. If `security` is enabled, this field must exist and must be filled in properly.
- `authentication`: It is a list of `policies`. Each policy accepts the following keys: * `secret`: When using **MAC** *(Message Authentication Code)* method for verification, this field allows to set the secret used to authenticate the client requesting a connection to the server. * `pubkey`: Path to a file containing a **PEM** encoded public key.

  NOTE: Either a *secret* or a *pubkey* is required.

  * `rules`: List of additional claims that should be checked. It should contain a map with keys corresponding to the claim identifier, and values corresponding to regex patterns that should match the payload's value. In the example above, the rule will check that the payload contains an `example` claim and that its value contains the *regex* keyword in any position of the message. This field is optional.
  * `algo`: The algorithm that should be used for encrypting the connection token. If the incoming token is not encrypted with the same algorithm, it will be discarded. If not specified, the HS256 algorithm will be used.
- `encoding`: Specifies the protocol, built over JSON, that allows users to exchange useful information between the client and the server, by means of specifying which keys are valid for the JSON sent/received messages and how they should be formatted for the server to accept and process these messages. By default, `json` encoding is provided in the *WebSocket System Handle* and used if not specified otherwise. Users can implement their own encoding by implementing the Encoding class.

  For the `websocket_client` *System Handle*, there are also two possible configuration scenarios: using TLS or TCP.

  TLS

```
systems:
  websocket:
    type: websocket_client
    host: localhost
```

(continues on next page)

(continued from previous page)

```
    port: 80
    cert_authorities: [my_cert_authority.ca.crt]
    authentication:
        token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey...
```

TCP

```
systems:
  websocket:
    type: websocket_client
    port: 80
    security: none
    encoding: json
    authentication:
        token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey...
```

* `port`: The specific port where the *client* will attempt to establish a connection to a *WebSocket server*. This field is mandatory.
* `host`: Address where the *WebSocket server* is hosted. If not specified, it will use `localhost` as the default value.
* `security`: If this field is not present, a secure TLS endpoint will be created. If the special value `none` is written, a TCP *WebSocket client* will be set up.
* `cert_authorities`: List of *certificate authorities* used to validate the client against the server. This field is optional and only applicable if `security` is not disabled.
* `authentication`: allows to specify the public `token` used to perform the secure authentication process with the server. This field is mandatory.
* `encoding`: Specifies the protocol, built over JSON, that allows users to exchange useful information between the client and the server, by means of specifying which keys are valid for the JSON sent/received messages and how they should be formatted for the server to accept and process these messages. By default, `json` encoding is provided in the *WebSocket System Handle* and used if not specified otherwise. Users can implement their own encoding by implementing the Encoding class.

JSON encoding protocol

In order to communicate with the *WebSocket System Handle* using the JSON encoding, the messages should follow a specific pattern. This pattern will be different depending on the paradigm used for the connection (*pub/sub* or *client/server*) and the communication purpose.

Several fields can be used in those messages, but not all of them are mandatory. All of them will be described in this section, as well as in which cases they are optional:

- `op`: The *Operation Code* is mandatory in every communication as it specifies the purpose of the message. This field can assume nine different values, which are the ones detailed below.
  - `advertise`: It notifies that there is a new publisher that is going to publish messages on a specific topic. The fields that can be set for this operation are: `topic`, `type` and optionally the `id`.

```
{"op": "advertise", "topic": "helloworld", "type": "HelloWorld", "id": "1"}
```

  - `unadvertise`: It states that a publisher is not going to publish any more messages on a specific topic. The fields that can be set for this operation are: `topic` and optionally the `id`.

```
{"op": "unadvertise", "topic": "helloworld", "id": "1"}
```

  - `publish`: It identifies a message that wants to be published over a specific topic. The fields that can be set for this operation are: `topic` and `msg`.

```
{"op": "publish", "topic": "helloworld", "msg": {"data": "Hello"}}
```

  - `subscribe`: It notifies that a subscriber wants to receive the messages published under a specific topic. The fields that can be set for this operation are: `topic` and optionally the `id` and `type`.

```
{"op": "subscribe", "topic": "helloworld", "type": "HelloWorld", "id": "1"}
```

  - `unsubscribe`: It states that a subscriber doesn't want to receive messages from a specific topic anymore. The fields that can be set for this operation are: `topic` and optionally the `id`.

```
{"op": "unsubscribe", "topic": "helloworld", "id": "1"}
```

  - `call_service`: It identifies a message request that wants to be published on a specific service. The fields that can be set for this operation are: `service`, `args` and optionally the `id`.

```
{"op": "call_service", "service": "hello_serv", "args": {"req": "req"}, "id":
↪"1"}
```

  - `advertise_service`: It notifies that a new server is going to attend to the requests done on a specific service. The fields that can be set for this operation are: `request_type`, `reply_type` and `service`.

```
{"op": "advertise_service", "service": "hello_serv", "request_type":
↪"HelloRequest", "reply_type": "HelloReply"}
```

  - `unadvertise_service`: It states that a server is not going to attend any more the requests done on a specific service. The fields that can be set for this operation are: `type` and `service`.

```
{"op": "unadvertise_service", "service": "hello_serv", "type": "HelloReply"}
```

  - `service_response`: It identifies a message reply that wants to be published as response to a specific request. The fields that can be set for this operation are: `service`, `values` and optionally the `id`.

```
{"op": "service_response", "service": "hello_serv", "values": {"resp": "resp"}
↪, "id": "1"}
```

- `id`: Code that identifies the message.
- `topic`: Name that identifies a specific topic.
- `type`: Name of the type that wants to be used for publishing messages on a specific topic.
- `request_type`: Name of the type that wants to be used for the service requests.
- `reply_type`: Name of the type that wants to be used for the service responses.
- `msg`: Message that is going to be published under a specific topic.
- `service`: Name that identifies a specific service.
- `args`: Message that is going to be published under a specific service as a request.
- `values`: Message that is going to be published under a specific service as a response.
- `result`: Value that states if the request has been successful.

Examples

There are several examples that you can find in this documentation in which the *WebSocket System Handle* is employed in the communication process. Some of them are presented here:

- *ROS 2 - WebSocket bridge*
- *WebSocket Service Server*

Compilation flags

Besides the *Global compilation flags* available for the whole *Integration Service* product suite, there are some specific flags which apply only to the *WebSocket System Handle*; they are listed below:

- `BUILD_WEBSOCKET_TESTS`: Allows to specifically compile the *WebSocket System Handle* unitary and integration tests. It is useful to avoid compiling each *System Handle*'section test suite present in the `colcon` workspace, which is what would happen if using the `BUILD_TESTS` flag, with the objective of minimizing building time. To use it, after making sure that the *WebSocket System Handle* is present in your `colcon` workspace, execute the following command:

```
~/is_ws$ colcon build --cmake-args -DBUILD_WEBSOCKET_TESTS=ON
```

API Reference

The *Integration Service API Reference* constitutes an independent section within this documentation. To access the *WebSocket System Handle* subsection, use this *link*.

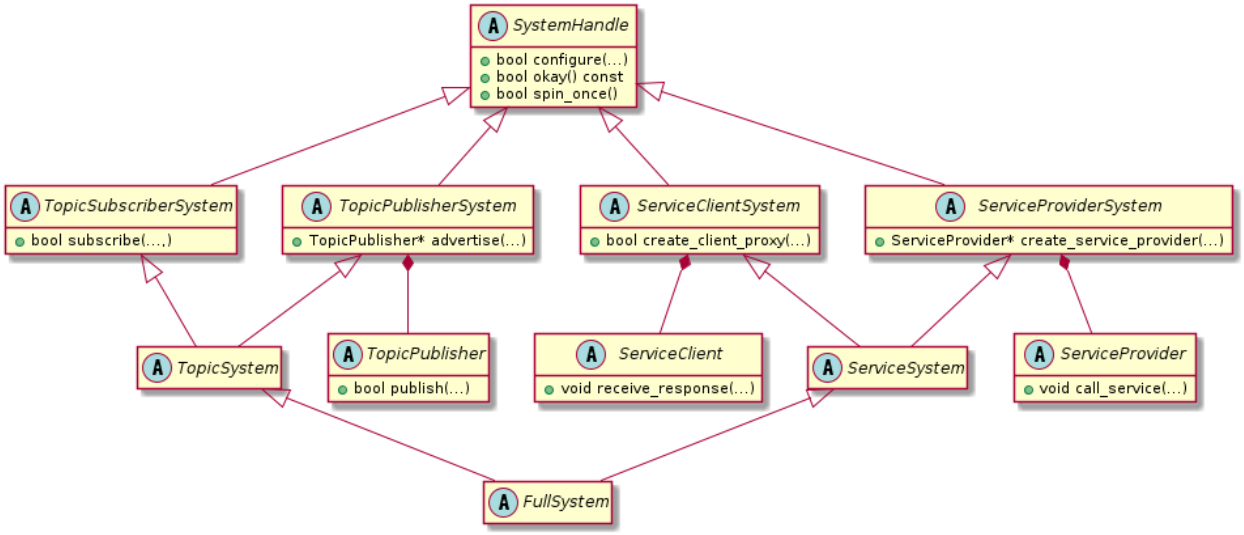
Protocol	*System Handle* overview
Fast DDS	*Fast DDS System Handle*
FIWARE	*FIWARE System Handle*
ROS 1	*ROS 1 System Handle*
ROS 2	*ROS 2 System Handle*
WebSocket	*WebSocket System Handle*

Additional *System Handles* can be implemented by users, in order to have the desired middlewares joining the *Integration Service* world. Adding a new *System Handle* automatically allows communication with the rest of the protocols already available in this ecosystem.

6.11.2 Implementation

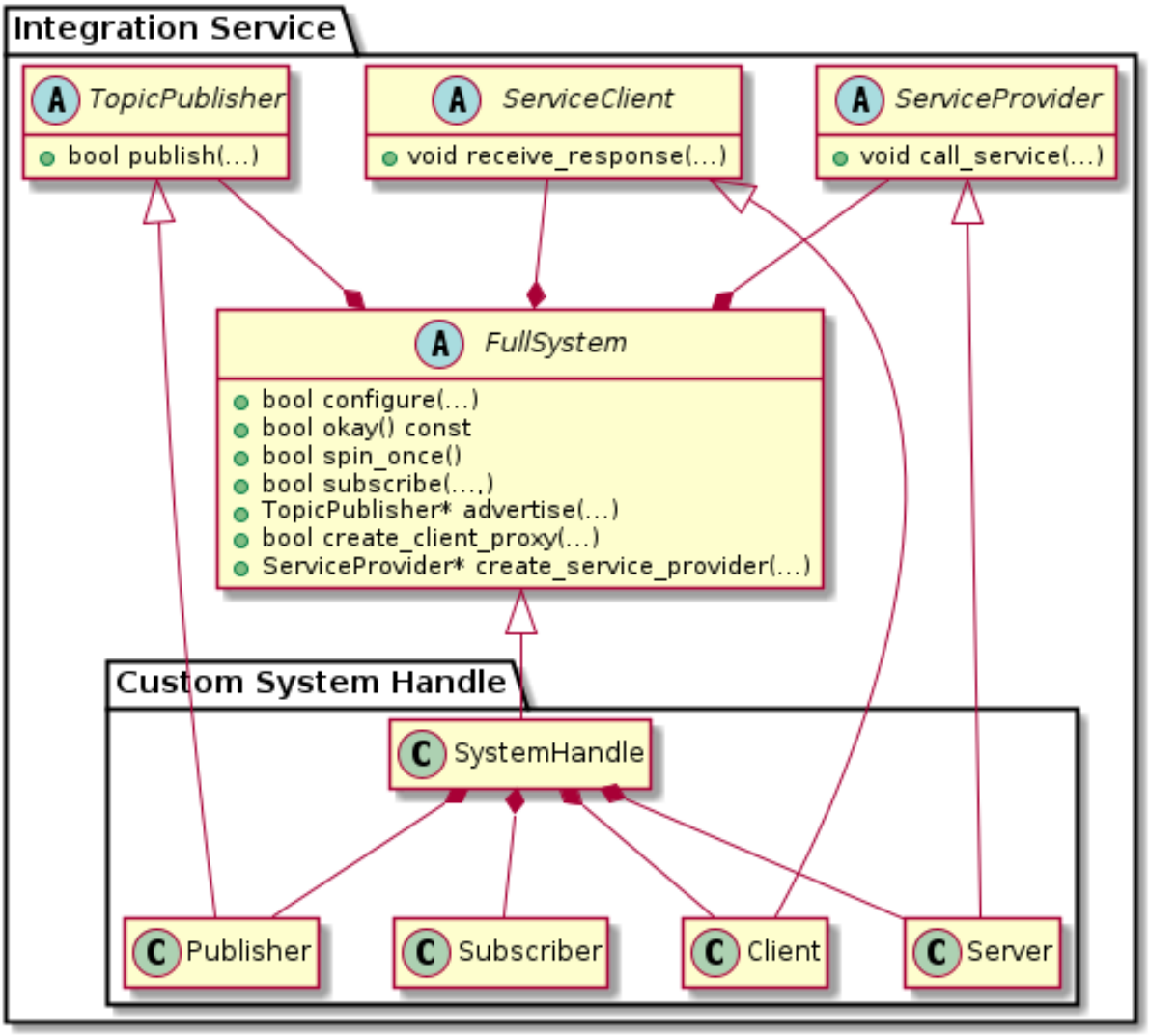
This section provides an overview of the architecture of a *System Handle*, by depicting the class inheritance structure and specifying the methods which need to be implemented in order to create a custom *System Handle*.

Here you can find a diagram of a *System Handle* class inheritance structure.



Each *System Handle* must inherit, directly or indirectly, from the `SystemHandle` superclass. Depending on the nature of each protocol, it should implement the derived classes using multiple inheritance from `TopicSubscriberSystem`, `TopicPublisherSystem`, `ServiceClientSystem`, and/or `ServiceProviderSystem`. To simplify this inheritance, classes `TopicSystem`, `ServiceSystem`, and `FullSystem` are available to inherit from.

In the diagram below, the architecture of a generic "Full" *System Handle* and its integration into *Integration Service* is shown.



To ease the implementation, the new `system::SystemHandle` will inherit from `FullSystem`. The following sections will explain the methods to be implemented.

To implement the `TopicPublisher`, `ServiceClient`, and `ServiceProvider` interfaces, the most direct way is to create child classes, respectively `system::Publisher`, `system::Client`, and `system::Server`. An additional class `system::Subscriber` may be useful to manage the subscribers created. In the example shown in the diagram above, the `system::SystemHandle` will contain the needed instances of these classes, but any approach may be valid if the interfaces are met.

SystemHandle Class

All *System Handles* must implement the `configure`, `okay`, and `spin_once` methods that belong to the superclass:

```
bool configure(
    const RequiredTypes& types,
    const YAML::Node& configuration,
    TypeRegistry& type_registry);

bool okay() const = 0;

bool spin_once();
```

The `configure` method is called to setup the *System Handle* with the associated `configuration`, defined in the *YAML* file that is passed to it. The types that the SH needs to manage to implement the communication are passed to this method via the `types` argument, whereas the new types created by the *System Handle* are expected to be filled in the `type_registry`.

The `okay` method is called by *Integration Service* to check if the *System Handle* is working. This method will verify internally if the middleware has any problem.

The `spin_once` method is called by *Integration Service* to allow spinning to those middlewares that need it.

TopicSubscriberSystem Class

This kind of system must implement the `subscribe` and the `is_internal_message` method:

```
using SubscriptionCallback = std::function<void(const xtypes::DynamicData& message,␣
↪void* filter_handle)>;

bool subscribe(
    const std::string& topic_name,
    const xtypes::DynamicType& message_type,
    SubscriptionCallback callback,
    const YAML::Node& configuration);

bool is_internal_message(
    void* filter_handle);
```

Integration Service will call the `subscribe` method in order to create a new subscriber to the topic `topic_name` using `message_type` type, plus an optional `configuration`. Once the middleware system receives a message from the subscription, the message must be translated into the `message_type` and the *System Handle* must invoke the `callback` with the translated message.

The `callback` will be called only if the `is_internal_message` method returns `false`. This prevents *Integration Service* from recursively send messages within itself, for example, if a publisher and a subscriber are created pointing to the same topic. Users must define, for each middleware, the type of the *filter_handle* parameter, and cast it accordingly. Some protocols, such as *WebSocket*, might not need to filter its messages at all; in that case, this method can be simply implemented as a `return false;` clause.

TopicPublisherSystem Class

This kind of system must implement the `advertise` method:

```
std::shared_ptr<TopicPublisher> advertise(
    const std::string& topic_name,
    const xtypes::DynamicType& message_type,
    const YAML::Node& configuration);
```

Integration Service will call this method in order to create a new `TopicPublisher` to the topic `topic_name` using `message_type` type, and optional `configuration`.

The `TopicPublisher` is an interface that must be implemented by a `Publisher` in order to allow *Integration Service* to publish messages to the target middleware. This interface defines a single method `publish`:

```
bool publish(const xtypes::DynamicData& message);
```

When *Integration Service* needs to publish to the middleware system it will call the `TopicPublisher::publish` method, with a message that must be translated from the `message_type` parameter by the `advertise` method above.

ServiceClientSystem Class

This kind of system must implement the `create_client_proxy` method:

```
using RequestCallback =
    std::function<void(
        const xtypes::DynamicData& request,
        ServiceClient& client,
        std::shared_ptr<void> call_handle)>;

bool create_client_proxy(
    const std::string& service_name,
    const xtypes::DynamicType& service_type,
    RequestCallback callback,
    const YAML::Node& configuration);
```

Integration Service will call this method in order to create a new `ServiceClient` to the service `service_name` using the `service_type` type, plus an optional `configuration`. This `ServiceClient` will be provided as an argument in the `callback` invocation when a response is received.

The `ServiceClient` is an interface that must be implemented by a `Client` in order to allow *Integration Service* to relate a *request* with its *reply*. This is done by providing a `call_handle` both in the `call_service` method from `ServiceProvider` and in the `callback` from `create_client_proxy` method. When the *reply* is received by another *System Handle*, its `ServiceProvider` will call the `receive_response` method from the `Client`:

```
void receive_response(
    std::shared_ptr<void> call_handle,
    const xtypes::DynamicData& response);
```

The `receive_response`:

- Translates the `response` from `service_type` and relate the `call_handle`, if needed, to its middleware's request;
- Replies to its middleware.

ServiceProviderSystem Class

This kind of system must implement the `create_service_proxy` method:

```
std::shared_ptr<ServiceProvider> create_service_proxy(
    const std::string& service_name,
    const xtypes::DynamicType& service_type,
    const YAML::Node& configuration);
```

Integration Service will call this method in order to create a new `ServiceProvider` to the service `service_name` using the `service_type` type, plus an optional `configuration`.

The `ServiceProvider` is and interface that must be implemented by a `Server` in order to allow *Integration Service* to *request* (or call) a service from the target middleware.

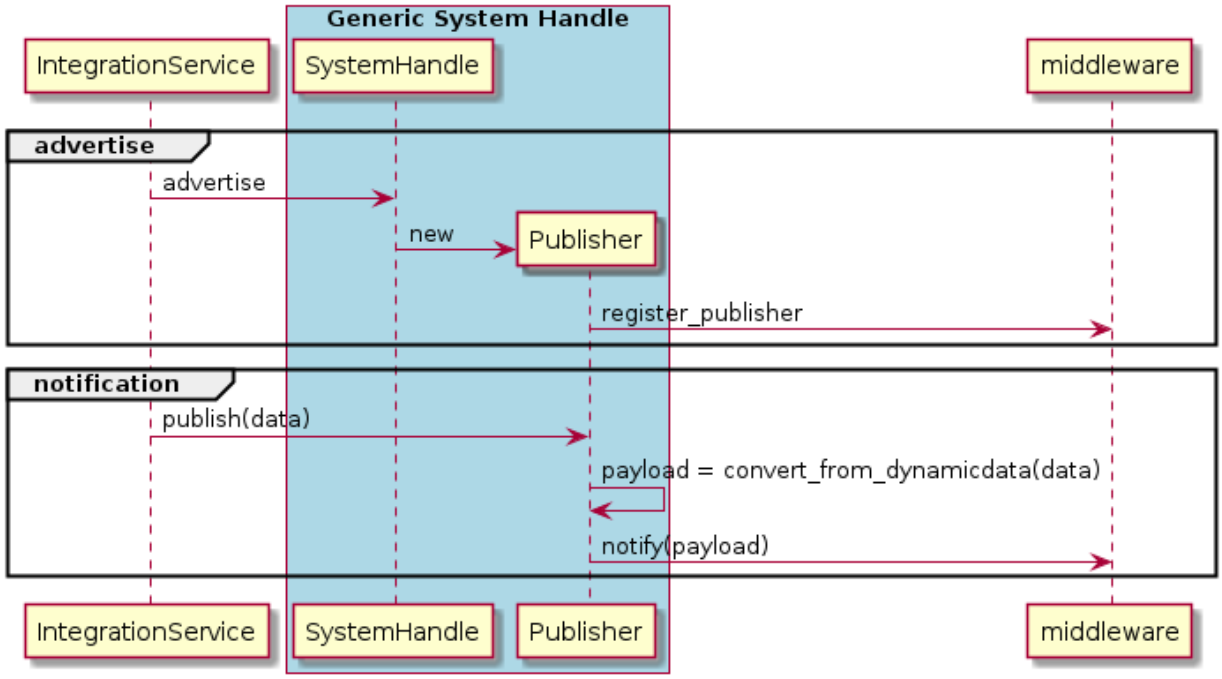
```
void call_service(
    const xtypes::DynamicData& request,
    ServiceClient& client,
    std::shared_ptr<void> call_handle);
```

This `call_service` method will translate the `request` from `service_type` and will call its middleware service, which stores the related `call_handle` and `client`. Once it receives the response from its middleware, it must translate back the response and retrieve the `call_handle` and `client` related. Then, it will invoke the `receive_response` method from the client using the `call_handle` as argument.

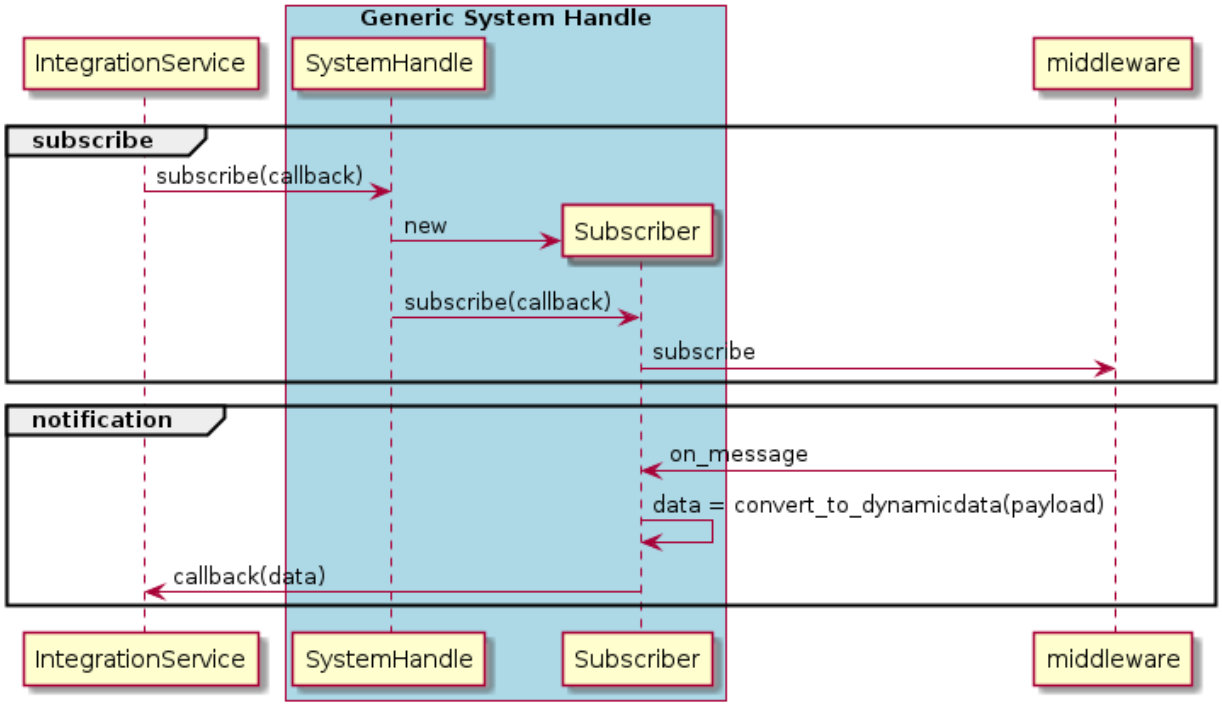
6.11.3 Sequence diagrams

The following diagrams illustrate the previous sections using a *generic System Handle*.

TopicPublisher flow

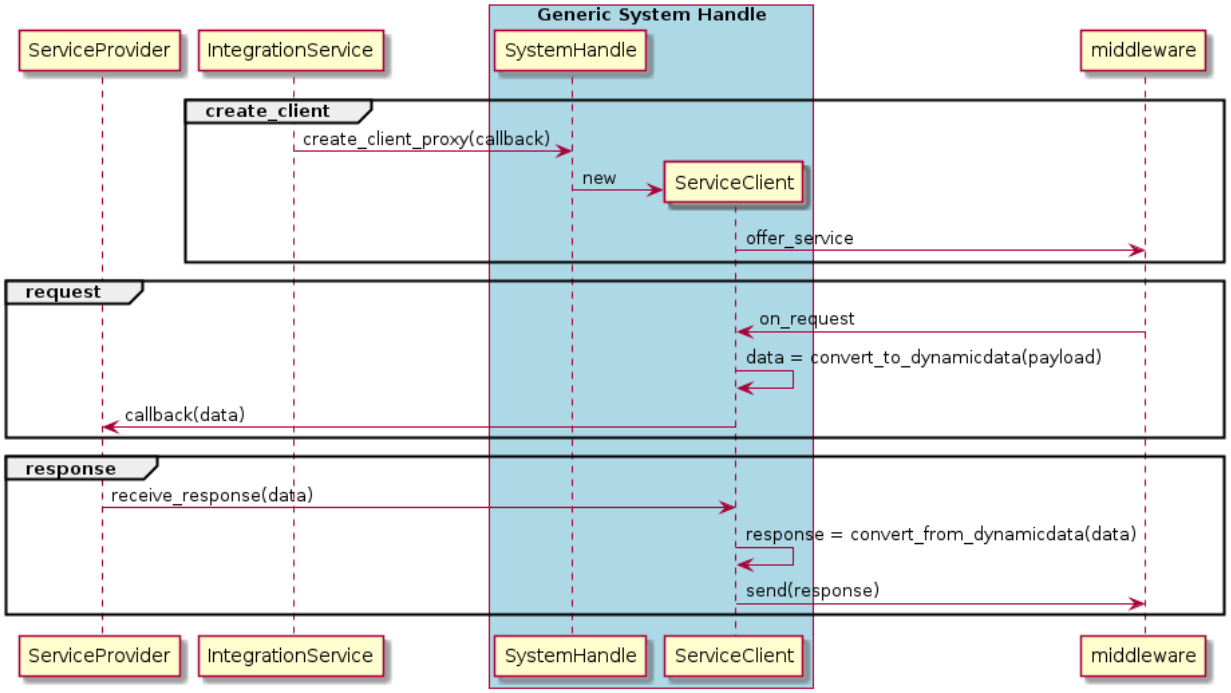


TopicSubscriber flow



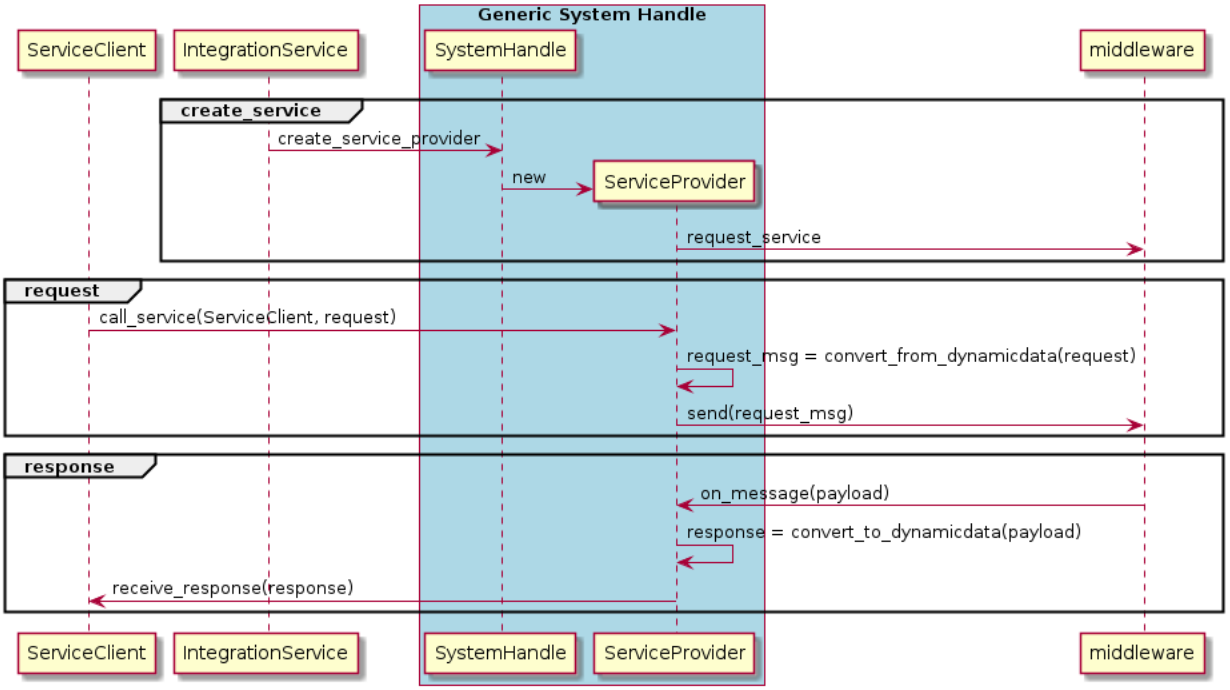
ServiceClient flow

Note that a `ServiceClient` acts as a client for *Integration Service* and as a server for the **middleware**.



ServiceProvider flow

Note that a `ServiceProvider` acts as a server for *Integration Service* and as a client for the **middleware**.



6.12 YAML Configuration

The first part of this section provides a general overview of all the parameters available to configure and launch an *Integration Service* instance. To get more detailed information on every subsection, please have a look at the list below:

- *Types definition*
- *Systems definition*
- *Routes definition*
- *Topics definition*
- *Services definition*
- *Remapping*

The *Integration Service* can be configured during runtime by means of a dedicated **YAML** file. This configuration file must follow a specific syntax, meaning that it is required that a number of compulsory sections are opportunely filled for it to successfully configure and launch an *Integration Service* instance, while others are optional. Both kinds are listed and reviewed below:

- `types`: *(optional)*: It allows to list the IDL types used by the *Integration Service* to later define the topics and services types which will take part in the communication process.

  This field can be omitted for certain *Integration Service* instances where one or more *System Handles* already include(s) static type definitions and their corresponding transformation libraries (*Middleware Interface Extension* or *mix* files).

```
types:
  idls:
    - >
      #include <GoodbyeWorld.idl>
      struct HelloWorld
      {
        string data;
        GoodbyeWorld bye;
      };
  paths: [ "/home/idl_files/goodbyeworld/" ]
```

Several parameters can be configured within this section:

- `idls`: List of *IDL* type definitions that can be directly embedded within the configuration file. If the `types` section is defined, this subsection is mandatory. The type can be entirely defined within the *YAML* file, or can be included from a preexisting *IDL* file; for the latter, the system path containing where the *IDL* file is stored must be placed into the `paths` section described below.
- `paths` *(optional):* Using this parameter, an existing *IDL* type written in a separate file can be included within the *Integration Service* types section. If the *IDL* path is not listed here, the previous subsection `#include` preprocessor directive will fail.

For more details on this section, please refer to the *Types definition* subsection of this page.

- `systems`: Specifies which middlewares will be involved in the communication process, allowing to configure them individually.

  Some configuration parameters are common for all the supported middlewares within the *Integration Service* ecosystem; while others are specific of each middleware. To see which parameters are relevant for a certain middleware, please refer to its dedicated subsection in the *Built-in System Handles* page.

```
systems:
  foo: { type: foo }
  bar: { type: bar, types-from: foo }
```

  In relation to the common parameters, their behavior is explained in the following section:

  - `type`: Middleware or protocol kind. To date, the supported middlewares are: *fastdds*, *ros1*, *ros2*, *fiware*, *websocket_server* and *websocket_client*. There is also a *mock* option, mostly used for testing purposes.
  - `types-from` *(optional)*: Configures the types inheritance from a given system to another. This allows to use types defined within *Middleware Interface Extension* files for a certain middleware into another middleware, without the need of duplicating them or writing an equivalent *IDL* type for the rest of systems.

  For more details on this section, please refer to the *Systems definition* subsection of this page.

- `routes`: In this section, a list must be introduced, corresponding to which bridges are needed by *Integration Service* in order to fulfill the intercommunication requirements for a specific use case.

  At least one route is required; otherwise, running *Integration Service* would be useless.

```
routes:
  foo_to_bar: { from: foo, to: bar }
  bar_to_foo: { from: bar, to: foo }
  foo_server: { server: foo, clients: bar }
  bar_server: { server: bar, clients: foo }
```

  There are two kinds of routes, corresponding to either a publication/subscription paradigm or a server/client paradigm:

  - `from-to`: Defines a route **from** one (or several) system(s) **to** one (or several) system(s). A `from` system expects to connect a publisher user application with a subscriber user application in the `to` system.
  - `server-clients`: Defines a route for a request/reply architecture in which there are one or several **clients** which forward request petitions and listen to responses coming from a **server**, which must be unique for each service route.

  For more details on this section, please refer to the *Routes definition* subsection of this page.

- `topics`: Specifies the topics exchanged over the `routes` listed above corresponding to the publication-subscription paradigm. The topics must be specified in the form of a *YAML* dictionary, meaning that two topics can never have the same name.

  For each topic, some configuration parameters are common for all the supported middlewares within the *Integration Service* ecosystem; while others are specific of each middleware. To see which topic parameters must/can be configured for a certain middleware, please refer to its dedicated subsection in the *Built-in System Handles* page.

```
topics:
  hello_foo:
    type: HelloWorld
    route: bar_to_foo
  hello_bar:
    type: HelloWorld
    route: foo_to_bar
    remap: { bar: { topic: HelloBar } }
```

  In relation to the common parameters, their behavior is explained below:

  - `type`: The topic type name. This type must be defined in the `types` section of the YAML configuration file, or it must be loaded by means of a `Middleware Interface Extension` file by any of the middleware plugins or *System Handles* involved in the communication process.
  - `route`: Communication bridge to be used for this topic. The route must be one among those defined in the `routes` section described above.
  - `remap` *(optional):* Allows to establish equivalences between the **topic** name and its **type**, for any of the middlewares defined in the used route. This means that the topic name and type name may vary in each user application endpoint that is being bridged, but, as long as the type definition is equivalent, the communication will still be possible.

  For more details on this section, please refer to the *Topics definition* subsection of this page.

- `services`: Allows to define the services that *Integration Service* will be in charge of bridging, according to the service `routes` listed above for the client/server paradigm. The services must be specified in the form of a *YAML* dictionary, meaning that two services can never have the same name.

  For each service, some configuration parameters are common for all of the supported middlewares within the *Integration Service* ecosystem; while others are specific of each middleware. To see which parameters must/can be configured for a certain middleware in the context of a service definition, please refer to its dedicated subsection in the *Built-in System Handles* page.

```
services:
  serve_foo:
    request_type: FooRequest
    reply_type: FooReply
    route: foo_server
  serve_bar:
    request_type: BarRequest
    reply_type: BarReply
```

(continues on next page)

(continued from previous page)

```
    route: bar_server
    remap: { foo: { request_type: bar_req, reply_type: bar_repl, topic: ServeBar }
↪ }
```

Regarding the common parameters, they differ slightly from the `topics` section:

- `type` *(optional):* The service type. As services usually are composed of a request and a reply, this field only makes sense for those services which consist solely of a request action with no reply. Usually, within the `services` context, it is not used at all.
- `request_type`: The service request type. This type must be defined in the `types` section of the YAML configuration file, or must be loaded by means of a `Middleware Interface Extension` file by any of the middleware plugins, or *System Handles*, involved in the communication process.
- `reply_type`: The service reply type. This type must be defined in the `types` section of the YAML configuration file, or must be loaded by means of a `Middleware Interface Extension` file by any of the middleware plugins, or *System Handles*, involved in the communication process.
- `route`: Communication bridge to be used for this service. The route must be one among those defined in the `routes` section described above and must be a route composed of a *server* and one or more *clients*.
- `remap` *(optional):* Allows to establish equivalences between the **service** name (*topic* field) and its **request and reply type**, for any of the middlewares defined in the used route. This means that the service name and types names may vary in each user application endpoint that is being bridged, but, as long as the type definition is equivalent, the communication will still be possible.

For more details on this section, please refer to the *Services definition* subsection of this page.

6.12.1 Types definition

Some *System Handles* have the ability to inform *Integration Service* of the types definition (using xTypes) that they can use. The *System Handles* of *ROS 1* and *ROS 2* are examples of this. Nevertheless, there are cases where the *System Handle* is not able to retrieve the type specification (*websocket*, *mock*, *dds*, *fiware*, ...) that it needs for the communication.

In those cases, there are two ways to pass this information to the *System Handle*:

- Using the `types-from` property, that *imports* the types specification from another system.
- Specifying the type yourself by embedding an *IDL* into the YAML.

Regarding the second option, the *IDL* content can be provided in the *YAML* either directly, as follows:

```
types:
    idls:
        - >
            struct name
            {
                idl_type1 member_1_name;
                idl_type2 member_2_name;
            };
```

or by inclusion of a `paths` field, that can be used to provide the preprocessor with a list of paths where to search for *IDL* files to include into the *IDL* content. The syntax in this case would be:

```
types:
    idls:
        - >
```

(continues on next page)

(continued from previous page)

```
            #include <idl_file_to_parse.idl>

        paths: [ idl_file_to_parse_path ]
```

Notice that these two approaches can be mixed.

The name for each type can be whatever the user wants, with the two following rules:

1. The name cannot have spaces in it.
2. The name must be formed only by letters, numbers and underscores.

Note: A minimum of a structure type is required for the communication.

For more details about *IDL* definition, please refer to the IDL specification documentation.

The following is an example of a full configuration defining a `dds-fiware` communication using the types definition contained in the `idls` block.

```
types:
    idls:
        - >
            struct Stamp
            {
                int32 sec;
                uint32 nanosec;
            };

            struct Header
            {
                string frame_id;
                stamp stamp;
            };

systems:
    dds: { type: dds }
    fiware: { type: fiware, host: 192.168.1.59, port: 1026 }

routes:
    fiware_to_dds: { from: fiware, to: dds }
    dds_to_fiware: { from: dds, to: fiware }

topics:
    hello_dds:
        type: "Header"
        route: fiware_to_dds
    hello_fiware:
        type: "Header"
        route: dds_to_fiware
```

6.12.2 Systems definition

A *System Handle* may need additional configuration that should be defined in its `systems` entry as a *YAML* map. Each entry of this section represents a middleware involved in the communication, and corresponds to an instance of a *System Handle*. All *System Handles* accept the `type` and `types-from` options in their `systems` entry. If `type` is omitted, the key of the *YAML* entry will be used as `type`.

```
systems:
    dds:
    ros2_domain5: { type: ros2, domain: 5, node_name: "ros_node_5" }
    fiware: { host: 192.168.1.59, port: 1026 }
```

The snippet above will create three *System Handles* instances:

- A *DDS System Handle* instance, with default configuration.
- A *ROS 2 System Handle* instance, named `ros2_domain` with `domain = 5` and `node_name = "is_5"`.
- A *FIWARE System Handle* instance, with `host = 192.168.1.59` and `port = 1026`.

The *System Handles* currently available for *Integration Service* are listed in a table that you can find in the *Built-in System Handles* section of this documentation.

A new *System Handle* can be created by implementing the desired `SystemHandle` subclasses to add support to any other protocol or system. For more information consult the *System Handle* section.

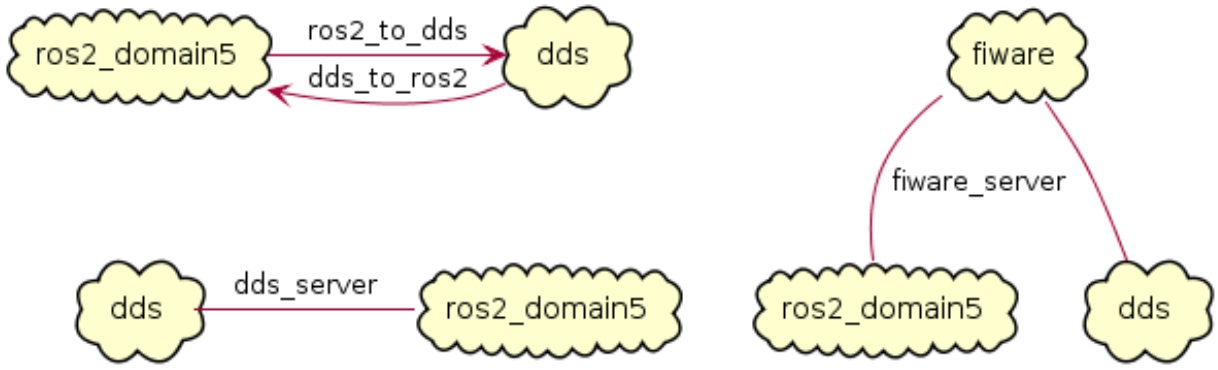
6.12.3 Routes definition

This section allows enumerating the bridges between the systems that *Integration Service* must manage. To achieve bidirectional communication, both ways must be specified.

`routes` definition keywords are specific depending on whether the route is defining a *publisher/subscriber* path (`from-to`) or a *service/client* communication path (`server-client`). For example:

```
routes:
    ros2_to_dds: { from: ros2_domain5, to: dds }
    dds_to_ros2: { from: dds, to: ros2_domain5 }
    dds_server: { server: dds, clients: ros2_domain5 }
    fiware_server: { server: fiware, clients: [ dds, ros2_domain5 ] }
```

This *YAML* defines the following routes:



- The route `ros2_to_dds` defines a `ros2_domain5` publisher with a `dds` subscriber.
- The route `dds_to_ros2` defines a `dds` publisher with a `ros2_domain5` subscriber.

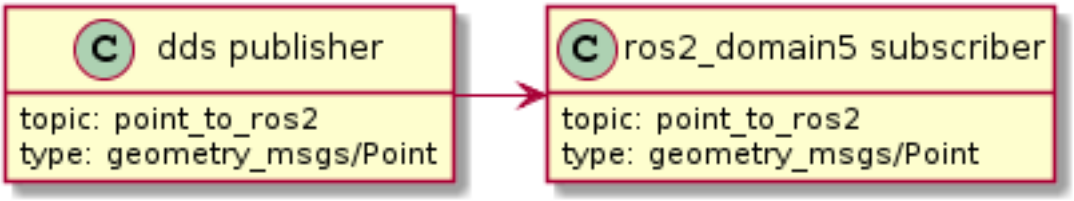
- Having the routes `ros2_to_dds` and `dds_to_ros2` results in a bidirectional communication between the `ros2_domain5` and `dds` systems.
- The route `dds_server` defines a `dds` server with only one client: `ros2_domain5`.
- The route `fiware_server` defines a `fiware` server with two clients: `ros2_domain5` and `dds`.

6.12.4 Topics definition

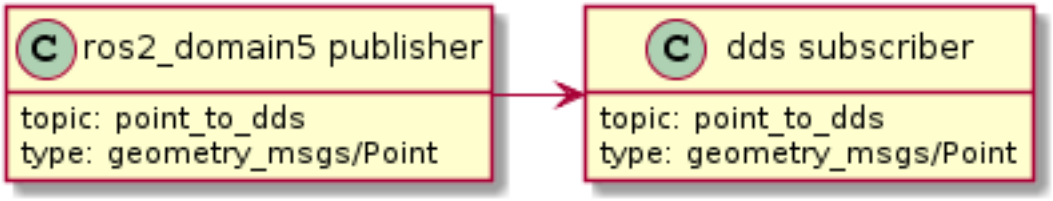
Each `system` is able to *publish/subscribe* to each other's `topics`. These *publish/subscription* policies are set directly in the YAML configuration file by specifying the topic `type` and its `route` (which `system` is the publisher and which is the subscriber) as the main parameters:

```
topics:
    point_to_ros2:
        type: "geometry_msgs/Point"
        route: dds_to_ros2
    point_to_dds:
        type: "geometry_msgs/Point"
        route: ros2_to_dds
```

- The topic `point_to_ros2` will create a `dds` publisher and a `ros2_domain5` subscriber.



- The topic `point_to_dds` will create a `ros2_domain5` publisher and a `dds` subscriber.



If a custom *System Handle* needs additional configuration regarding the `topics`, it can be added to the topic definition as new map entries.

6.12.5 Services definition

`service` definition is very similar to `topics` definition, with the difference that in this case `routes` can only be chosen among the ones specified with the *server/client* syntax; also, the `type` entry for these fields usually follows the *request/response* model, pairing each of them with the corresponding `route`, depending on which `system` acts as the server and which as the client(s).

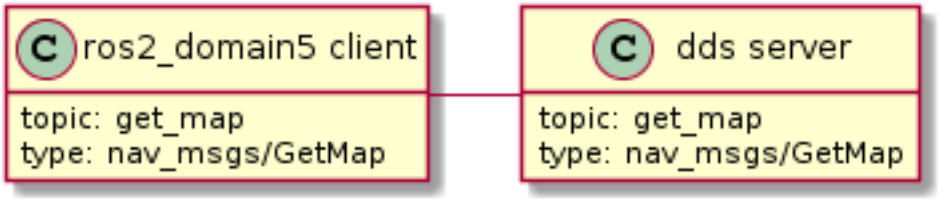
```
services:
    get_map:
        type: "nav_msgs/GetMap"
        route: dds_server
```

(continues on next page)

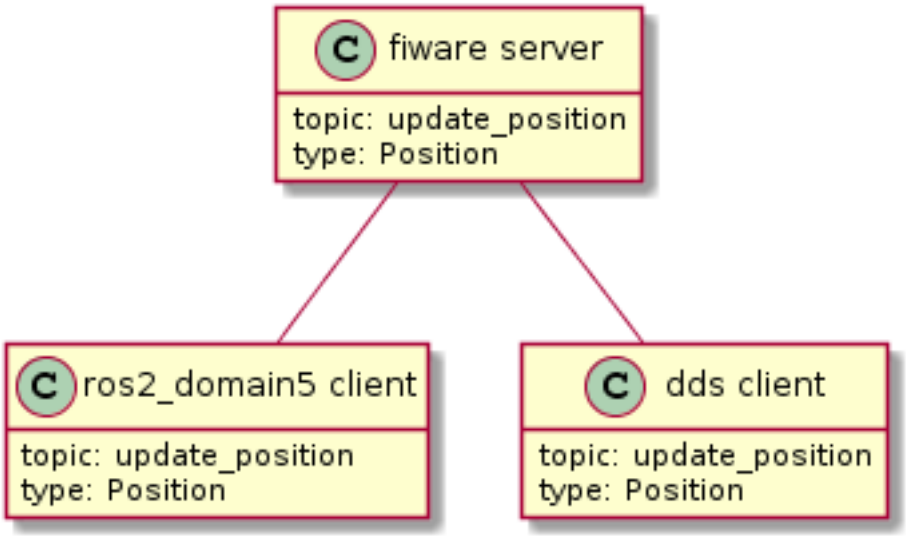
(continued from previous page)

```
    update_position:
        type: "Position"
        route: fiware_server
```

- The service `get_map` will create a `dds` server and a `ros2_domain5` client.



- The service `update_position` will create a `fiware` server, and `dds` and `ros2_domain5` clients.



If a custom *System Handle* needs additional configuration regarding the `services`, it can be added in the service definition as new map entries.

Note: If the `type` field is defined, as in the example above, this `type` will be taken into consideration as the **request type**. If a certain service needs to distinguish between **request** and **reply** types, the fields `request_type` and `reply_type` must be used instead.

6.12.6 Remapping

Sometimes, `topics` or `types` from one system are different from those managed by the systems with which it is being bridged. To solve this, *Integration Service* allows to remap `types` and `topics` in the *Topics definition* and in the *Services definition*.

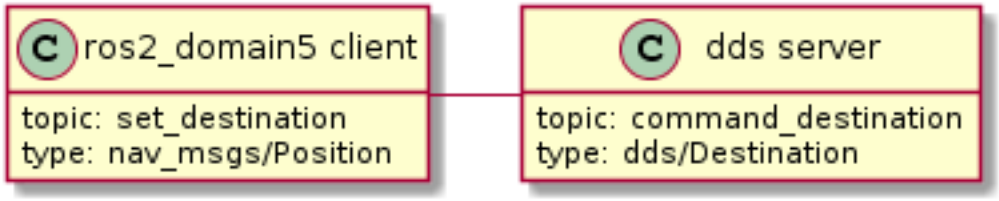
```
services:
    set_destination:
        type: "nav_msgs/Position"
        route: dds_server
```

(continues on next page)

(continued from previous page)

```
        remap:
            dds:
                type: "dds/Destination"
                topic: "command_destination"
```

In this `services` entry, the `remap` section defines the `type` and the `topic` that must be used in the `dds` system, instead of the ones defined by the service definition, which will be used by the `ros2_domain5` system.



6.13 Integration Service Core

This section presents the API provided by the *Integration Service* `is-core` library.

6.13.1 Core

This section of the API reference corresponds to the include/is/core folder of the *Integration Service* main repository.

This folder contains several files that can be divided into two different categories:

- Those located in the `include/is/core` folder, which are intended for parsing, configuring and executing an *Integration Service* instance.

Config

class `eprosima::is::core::internal::`**Config**

Internal representation of the configuration provided to the *Integration Service* instance, by means of a *YAML* file.

Public Types

using SubscriptionCallbacks = std::vector<std::unique_ptr<is::*TopicSubscriberSystem*::*SubscriptionCallba*

Signature for the container used to store the subscription callbacks for a certain *Integration Service* instance.

using RequestCallbacks = std::vector<std::unique_ptr<is::*ServiceClientSystem*::*RequestCallback*>>

Signature for the container used to store the service request callbacks for a certain *Integration Service* instance.

Public Functions

Config(**const** YAML::Node &*node* = YAML::Node(), **const** std::string &*filename* = "<text>")

Constructor.

Parameters

- `[in] node`: Parsed representation of the *YAML* configuration file. It defaults to empty.
- `[in] filename`: The path of the *YAML* configuration file.

bool **parse**(**const** YAML::Node &*node*, **const** std::string &*filename* = "<text>")

Parses the provided configuration, according to the configuration file scheme defined for *Integration Service*.

Configuration files typically contain the following sections:

i. `types`: Specifies the `IDL` types used by *Integration Service* to transmit messages. These `IDL` definitions will be parsed using `eprosima::xtypes` parser for `IDL` files, and the resulting *Dynamic Types* will be added to the available types database.

   The following subsections are permitted:

   1.1. `idl`: IDL content.

   1.2. `paths`: includes paths containing IDL definitions that will also be parsed and added to the types database.

ii. `systems`: Lists the middlewares involved in the communication, allowing to configure them. Custom aliases can be given to any system.

   The following subsections are permitted:

   2.1. `type`: to be selected among the middlewares supported by *Integration Service*: `ros2`, `dds`, `websocket`, `ros1`...

   2.2. `types-from`: allows to inherit type definitions from one system to another. In this way, users do not have to redefine types for each system.

   2.3. Custom configuration parameters, such as `domain_id` (for ROS 2). Each *SystemHandle* may define its own configuration fields, please refer to their documentation for more details.

iii. `routes`: Lists the communication bridges that *Integration Service* needs to establish among `systems`. Each route has a specific name.

   The following subsections are permitted:

   3.1. `from/to`: publisher/subscriber communication.

   3.2. `server/clients`: server/client communication.

iv. `topics/services`: Allows to configure the topics exchanged over the `routes` described above, in either publisher/subscriber or client/server communication, and provides detailed information about them. Each topic or service must have a unique name in the *YAML* file.

   The following subsections are permitted:

   4.1. `type`: Type involved in the communication. It can be a built-in type, usually coming from a `mix` library; or a custom user-defined type, by means of an IDL definition.

   4.2. `route`: Communication bridge, of the ones defined above, that must perform the communication.

   4.3. `remap`: allows to establish equivalences between `topic` names and `types` for each system involved in the communication.

4.4. Custom configuration parameters, which are specific for each middleware. Please refer to the specific *SystemHandle* documentation.

Return `True` if the parsing was correct, `false` if some error occurred.
Parameters
- `[in] node`: The parsed YAML representation of the configuration file provided.
- `[in] filename`: The path of the configuration file.

bool **`okay`**`()` **`const`**
Checks if everything is okay with the configuration process.

Return The `_okay` boolean parameter.

`operator bool``()` **`const`**
bool operator overload.

Return The *okay()* parameter.

bool **`load_middlewares`**(is::internal::*SystemHandleInfoMap* &*info_map*) **`const`**
Performs a search and loads the dynamic libraries required for each middleware, that is, the *SystemHandle* entities.

After the *SystemHandle* shared library is loaded successfully, the required types to be found during the *SystemHandle* configuration phase are registered, according to what was specified in the *YAML* configuration.

Next, the `types-from` parameter is checked, which specifies the middleware from which each *SystemHandle* wants to inherit types from, as declared in the configuration.

Finally, for each *SystemHandle*, the `configure` method is called. If one of the middlewares listed is not properly configured, the whole process fails.

Return Boolean value indicating whether the load process was successful or not.
Parameters
- `[out] info_map`: Map between the middlewares and their *SystemHandle* instances information (handle pointer, topic publisher and subscriber and service client and provider systems, as well as its type registry). This map should be filled with the information for all the *SystemHandle* defined in the configuration, once this method succeeds.

bool **`configure_topics`**(**`const`** is::internal::*SystemHandleInfoMap* &*info_map*, *SubscriptionCallbacks* &*subscription_callbacks*) **`const`**
Configures topics communication, according to the specified route, type and remapping parameters.

First, compatibility between the type defined in the `from` endpoint and the `to` (destination) endpoint is checked, because it could happen that, because of a remapping, the type definition in the source and destination systems is slightly (or completely) different. To do that, `check_topic_compatibility` is called. Please refer to its documentation for more details.

If the types are compatible, the next step is to check the publishing capabilities of the destination endpoint, and if everything is correct, that is, if the system has an associated *TopicPublisherSystem*, the *SystemHandle* advertises the topic. This publication will transmit the data to the user application, which must define a subscriber capable of receiving and processing the data.

Then, in the source endpoint, the existence of subscribing capabilities is checked, and, if so, the subscriber defines a SubscriptionCallback lambda, that iterates through the previous constructed list of publishers and ensures that each defined destination endpoint gets the data

published. This callback is used to call to *TopicSubscriberSystem::subscribe* method.

If any of the defined topics cannot find publishing or subscription capabilities (i.e. invalid routes), the returned value will be false and the process will fail.

Return `true` if all the topics were successfully configured, `false` otherwise.
Parameters

- `[in] info_map`: Map filled during the `load_middlewares` phase and containing the information for each loaded *SystemHandle*, in terms of its instance, supported types and publish/subscribe or client/server capabilities.
- `[in] subscription_callbacks`: Reference to the map used to store all of the active subscription callbacks for a certain *SystemHandle* instance.

bool **configure_services**(**const** is::internal::*SystemHandleInfoMap* &*info_map*, *RequestCallbacks* &*request_callbacks*) **const**

Configures services, according to the specified route, type and remapping parameters.

First, compatibility between the request and reply types defined in the `server` endpoint and the `clients` endpoints is checked, because it could happen that, because of a remapping, the types defined for request/reply is slightly (or completely) different in the server and client endpoints. To do that, `check_service_compatibility` is called. Please refer to its documentation for more details.

If the types are compatible, the next step is to check the service providing capabilities of the server endpoint, and if everything is correct, that is, if the system has an associated *ServiceProviderSystem*, the *SystemHandle* creates the corresponding service provider proxy.

Then, for all the defined clients, *ServiceClientSystem* capabilities are checked, and a request callback is defined, which basically executes the *ServiceProvider::call_service* method from the associated provider. This `call_service` is then responsible of sending back the response to the client, if applicable (that is, if a `reply_type` has been defined in the *YAML* configuration.)

If any of the defined services cannot find server or client capabilities (i.e. invalid routes), the returned value will be false and the process will fail.

Return `true` if all the services were successfully configured, `false` otherwise.
Parameters

- `[in] info_map`: Map filled during the `load_middlewares` phase and containing the information for each loaded *SystemHandle*, in terms of its instance, supported types and publish/subscribe and client/server capabilities.
- `[in] request_callbacks`: Reference to the map used to store all of the active request callbacks for a certain *SystemHandle* instance.

bool **check_topic_compatibility**(**const** is::internal::*SystemHandleInfoMap* &*info_map*, **const** std::string &*topic_name*, **const** *TopicConfig* &*config*) **const**

Checks compatibility between the *TopicInfo* registered in the endpoints responsible for a topic publish/subscribe communication in *Integration Service*.

This compatibility check is ensured thanks to eProsima `xtypes` library and its `TypeConsistency` definition. If types are not equal, some policies might be automatically applied to try to make them compatible, such as ignoring member names, type signs, etc.

Return `true` if the topic is compatible among the defined systems, `false` otherwise.
Parameters

- `[in] info_map`: Map filled during the `load_middlewares` phase and containing the information for each loaded *SystemHandle*, in terms of its instance, supported types and publish/subscribe or client/server capabilities.
- `[in] topic_name`: The topic whose compatibility will be checked between endpoints.
- `[in] config`: *TopicConfig* structure containing information such as the type, the source/destination defined route and the remappings, as well as the specific middleware configurations for this topic.

bool **check_service_compatibility**(**const** is::internal::*SystemHandleInfoMap* *&info_map*, **const** std::string *&service_name*, **const** *ServiceConfig* *&config*) **const**

Checks compatibility between the *ServiceConfig* registered in the endpoints responsible of a server/client communication in the *Integration Service*.

This compatibility check is ensured thanks to eProsima `xtypes` library and its `TypeConsistency` definition. If types are not equal, some policies might be automatically applied to try to make them compatible, such as ignoring member names, type signs, etc.

The check is performed both for request and reply types.

Return `true` if the service is compatible among the defined systems, `false` otherwise.
Parameters

- `[in] info_map`: Map filled during the `load_middlewares` phase and containing the information for each loaded *SystemHandle*, in terms of its instance, supported types and publish/subscribe and client/server capabilities.
- `[in] service_name`: The service whose compatibility will be checked between endpoints.
- `[in] config`: *ServiceConfig* structure containing information such as the request and reply types, the server and clients defined route and the remappings, as well as the specific middleware configurations for this service.

const eprosima::xtypes::DynamicType ***resolve_type**(**const** *TypeRegistry* *&types*, **const** std::string *&path*) **const**

This function allows to retrieve a type member from an externally defined type containing it, to use it as the type for a certain configuration.

The used syntax when retrieving the inner type must be `<outer_type>.<type_member_name>`.

For example, if a type is defined like this in an *IDL*:

```
union MyUnion (switch uint8)
{
      case 0: int32 _zero;
      case 1: int64 _one;
      default: int128 _default;
};
```

You can define the following topic: `ExampleTopic: { route: "a_to_b", type: MyUnion._zero }`

Return A pointer to the inner DynamicType representing the type requested by the user.
Parameters

- `[in] types`: TypeRegistry containing all the available types where the search of the parent type will be performed.

- `[in] path`: The whole type definition, as specified by the user. In the example, it would be `MyUnion._zero`.

Public Static Functions

Config **`from_file`**`(`**`const`** std::string &*file*`)`
Helper static constructor to retrieve a *Config* instance from a file path.

Parameters
- `[in] file`: A string containing the configuration file path.

`struct` `eprosima::is::core::internal::`**`MiddlewareConfig`**
Holds information relative to each middleware configuration.

Public Members

std::string **`type`**
The name of the middleware.

std::vector<std::string> **`types_from`**
The name of middleware whose types want to be used.

YAML::Node **`config_node`**
YAML configuration associated with the specific middleware.

`struct` `eprosima::is::core::internal::`**`TopicRoute`**
Stores information relative to topic routes:

Public Functions

std::set<std::string> **`all`**`()` **`const`**
Helper method to retrieve at once *from* and *to* sets.

Return A set containing all endpoints for this *TopicRoute*.

Public Members

std::set<std::string> **`from`**
Source middleware endpoint.

std::set<std::string> **`to`**
Destination middleware endpoint.

`struct` `eprosima::is::core::internal::`**`ServiceRoute`**
Stores information relative to service routes:

Public Functions

std::set<std::string> **all**() **const**
Helper method to retrieve at once *server* and *clients* sets.

Return A set containing all endpoints for this *ServiceRoute*.

Public Members

std::set<std::string> **clients**
Client endpoints.

std::string **server**
Server endpoint.

struct eprosima::is::core::internal::**TopicInfo**
Struct containing information about a certain topic.

Public Members

std::string **name**
The name of the topic.

std::string **type**
The name of the type for the specific topic.

using eprosima::is::core::internal::**ServiceInfo** = *TopicInfo*
Struct containing information about a certain service.

- **std::string name** The name of the service.
- **std::string type** The name of the request type for the specific service.
- **std::string reply_type** The name of the reply type for the specific service.

struct eprosima::is::core::internal::**TopicConfig**
Holds the configuration provided for a certain topic.

Public Members

std::string **message_type**
The name of the type for the specific topic.

TopicRoute **route**
The route followed by the specific topic.

std::map<std::string, *TopicInfo*> **remap**
A map with the remaps needed for the specific topic.

std::map<std::string, YAML::Node> **middleware_configs**
A map with the YAML configuration for the specific topic.

struct eprosima::is::core::internal::**ServiceConfig**
This struct stores the configuration provided for a certain service.

Public Members

std::string **request_type**
The name of the request type for the specific service.

std::string **reply_type**
The name of the reply type for the specific service.

ServiceRoute **route**
The route followed by the specific service.

std::map<std::string, *ServiceInfo*> **remap**
A map with the remaps needed for the specific service.

std::map<std::string, YAML::Node> **middleware_configs**
A map with the YAML configuration for the specific service.

Instance

class `eprosima::is::core::Instance`
Base class for creating an *Integration Service* instance. It can be called directly, or under the wrapping methods `run_instance`.

Public Functions

Instance(int *argc*, char **argv*[])
Creates an *Integration Service* instance which receives the arguments fed by the user from the command line.

Parameters
- `[in] argc`: Number of arguments given.
- `[in] argv`: String representation list of arguments provided, to be parsed before launching the instance.

Instance(**const** YAML::Node &*config_node*, **const** std::vector<std::string> &*is_prefixes*, **const** *MiddlewarePrefixPathMap* &*middleware_prefixes*)
Creates an *Integration Service* instance explicitly indicating the configuration of the *Integration Service* core and of the dedicated middleware *SystemHandle* plugins, and setting their relevant properties.

Parameters
- `[in] config_node`: The *YAML* configuration file, structured as defined in the *`is::core::internal::Config::parse()`* method documentation, that should be provided to *Integration Service* to successfully start a communication between two or more applications using different communication protocols.
- `[in] is_prefixes`: Global prefix paths for *Integration Service* to search for configuration files or `mix` files. These act as a complement to the already existing environment variables created during compilation/installation steps by CMake.
- `[in] middleware_prefixes`: Prefix paths specific to a certain middleware. Used when loading a middleware's plugin, that is, its *SystemHandle* implementation.

Instance(**const** std::string &*config_file_path*, **const** std::vector<std::string> &*is_prefixes*, **const** *MiddlewarePrefixPathMap* &*middleware_prefixes*)
Creates an *Integration Service* instance explicitly indicating the configuration of the *Integration Service* core and of the dedicated middleware *SystemHandle* plugins, and setting their relevant properties.

Parameters
- `[in] config_file_path`: The *YAML* configuration file, structured as defined in the *is::core::internal::Config::parse()* method documentation, that should be provided to *Integration Service* to successfully start a communication between two or more applications using different communication protocols.
- `[in] is_prefixes`: Global prefix paths for *Integration Service* to search for configuration files or `mix` files. These act as a complement to the already existing environment variables created during compilation/installation steps by CMake.
- `[in] middleware_prefixes`: Prefix paths specific to a certain middleware. Used when loading a middleware's plugin, that is, its *SystemHandle* implementation.

~Instance()
Destructor.

InstanceHandle **run**()
Runs the *Integration Service* instance in its own thread.

The handle allows to wait on that thread or instruct it to quit.

The handle uses RAII, so the instance will stop running automatically if the *InstanceHandle* dies.

If *run()* is called again while another instance handle is still alive and running, the new instance handle will still refer to the previously started and still running instance. Calling `quit()` on any of the handles will make them all quit. The automatic RAII shutdown of the instance will become effective once all handles have died.

If the existing handles are still alive but no longer running, they will become detached from this instance, and calling *run()* will initiate a new set of instance handles.

In most cases, simply calling one of the *run_instance()* functions and not worrying about how *InstanceHandle* entities might interact is more than enough.

Return An *InstanceHandle* to manage the running *Integration Service* instance.

using eprosima::is::core::**MiddlewarePrefixPathMap** = std::unordered_map<std::string, std::vector<std
MiddlewarePrefixPathMap contains a map of the prefixes that are available for each middleware. These prefixes are used to look for the dynamic libraries of either the *SystemHandle* plugin or the *MiddlewareInterfaceExtension* files and, once they are located, to load them.

See *Search*

InstanceHandle

class eprosima::is::core::**InstanceHandle**
This is the class responsible of handling an *Integration Service* instance.

It allows to perform several actions on the *Integration Service* instance, such as asking whether it is running or not or handling the threads that are launched each time a *SystemHandle* is launched from the core.

It also allows to quit the instance in a safe way, waiting for the pending jobs to finish.

Public Functions

InstanceHandle(**const** *InstanceHandle* &*other*)
Copy constructor.

Parameters
- `[in] other`: The *InstanceHandle* to be copied.

InstanceHandle(*InstanceHandle* &&*other*)
Move constructor.

Parameters
- `[in] other`: Movable reference to another *InstanceHandle* object.

~InstanceHandle()
Destructor.

The destructor will call *quit()* and then *wait()*, because the *Integration Service* instance cannot run without the handle active.

bool **running**() **const**
It allows to check if the instance is still running.

Return `true` if the *Integration Service* instance is still running, `false` otherwise.

operator bool() **const**
`bool()` operator overload. It performs an implicit cast to *running()*.

Return `true` if the *Integration Service* instance is still running, `false` otherwise.

int **wait**()
Waits for the instance to stop running.

The instance may be stopped by calling *quit()* or by sending `SIGINT` (`ctrl+C` from the terminal).

Return The return code for the execution process of this instance.

InstanceHandle &**wait_for**(**const** std::chrono::nanoseconds &*max_time*)
Waits for the instance to stop running, or for the max time to be reached.

Return A reference to this instance handle, so that it can be chained with *quit()* or *wait()*.

Parameters
- `[in] max_time`: Time, in nanoseconds, to wait for the instance to finish running.

InstanceHandle &**quit**()
Instructs the node handle to quit (this will not occur instantly).

After this, it calls *wait()* in order to wait until the instance has finished running, and retrieves the return code.

Return A reference to this instance handle so that it can be chained with *wait_for()* or *wait()*.

const *TypeRegistry* ***type_registry**(**const** std::string &*middleware_name*)
Requests the TypeRegitry for a given middleware.

Return A pointer to the TypeRegistry, or `nullptr` if the middleware does not exist.

Parameters
- `[in] middleware_name`: The middleware whose TypeRegistry is to be retrieved.

core::*InstanceHandle* `eprosima::is::`**`run_instance`**(int *argc*, char **argv*[])

Creates an *Integration Service* instance and runs it in its own thread. This is a wrapper for the *core::Instance* constructor and for the *core::Instance::run()* method.

Return An InstanceHandle to manage the running *Integration Service* instance.

Parameters

- `[in] argc`: Number of given arguments.
- `[in] argv`: String representation list of the provided arguments, to be parsed before launching the instance.

core::*InstanceHandle* `eprosima::is::`**`run_instance`**(**const** std::string *&config_file_path*, **const** std::vector<std::string> *&is_prefixes* = {}, **const** core::*MiddlewarePrefixPathMap* *&middleware_prefixes* = {})

Creates an *Integration Service* instance and runs it in its own thread. This is a wrapper for the *core::Instance* constructor and the `run()` method.

Return An InstanceHandle to manage the running *Integration Service* instance.

Parameters

- `[in] config_file_path`: The *YAML* configuration file, structured as defined in the *is::core::internal::Config::parse()* method documentation, that should be provided to *Integration Service* to successfully start a communication between two or more applications using different communication protocols.
- `[in] is_prefixes`: Global prefix paths for *Integration Service* to search for configuration files or `mix` files. These act as a complement to the already existing environment variables created during compilation/installation steps by CMake.
- `[in] middleware_prefixes`: Prefix paths specific to a certain middleware. Used when loading a middleware's plugin, that is, its *SystemHandle* implementation.

core::*InstanceHandle* `eprosima::is::`**`run_instance`**(**const** YAML::Node *&config_node*, **const** std::vector<std::string> *&is_prefixes* = {}, **const** core::*MiddlewarePrefixPathMap* *&middleware_prefixes* = {})

Creates an *Integration Service* instance and runs it in its own thread. This is a wrapper for the *core::Instance* constructor and the `run()` method.

Return An InstanceHandle to manage the running *Integration Service* instance.

Parameters

- `[in] config_node`: The *YAML* configuration file, structured as defined in the *is::core::internal::Config::parse()* method documentation, that should be provided to *Integration Service* to successfully start a communication between two or more applications using different communication protocols.
- `[in] is_prefixes`: Global prefix paths for *Integration Service* to search for configuration files or `mix` files. These act as a complement to the already existing environment variables created during compilation/installation steps by CMake.
- `[in] middleware_prefixes`: Prefix paths specific to a certain middleware. Used when loading a middleware's plugin, that is, its *SystemHandle* implementation.

- Those located in the `include/is/core/runtime` folder, which corresponds with the runtime necessary tools. To date, these include the search tool to load the .mix associated with the *System Handles* that will be used during execution and the tool that allows accessing the fields specified in the *YAML* configuration file.

FieldToString

class eprosima::is::core::**FieldToString**

Convenience class for converting simple field types into strings. It is useful to discern between the different Dynamic Types that may be requested to be replaced in a certain *StringTemplate*, and perform the conversion accordingly.

Public Functions

FieldToString(**const** std::string &*usage_details*)

Constructor.

Parameters

- `[in] usage_details`: Sets the details for how the conversion should be used.

FieldToString(**const** *FieldToString* &*other*)

Copy Constructor.

Parameters

- `[in] other`: The instance to be copied.

FieldToString(*FieldToString* &&*other*)

Move Constructor.

Parameters

- `[in] other`: The instance to be moved.

~FieldToString() = default

Destructor.

const std::string **to_string**(eprosima::xtypes::ReadableDynamicDataRef *field*, **const** std::string &*field_name*) **const**

Converts a certain field to a string, given the field name.

Return A const string representation of the requested field.

Parameters

- `[in] field`: Reference to the Dynamic Data instance representing the field's values.
- `[in] field_name`: The specific field whose value should be retrieved.

const std::string &**details**() **const**

Gets a const reference to the details attribute.

Return A const string reference to "details".

std::string &**details**()

Gets a mutable reference to the details attribute.

Return A non-const string reference to "details".

class eprosima::is::core::**UnknownFieldToStringCast** : **public** runtime_error

Exception that gets thrown by *FieldToString* when it's unknown how to convert a given field type into a string.

Public Functions

UnknownFieldToStringCast(**const** std::string &*type*, **const** std::string &*field_name*, **const** std::string &*details*)
Constructor.

Parameters
- `[in] type`: The type kind that should have been cast to a string.
- `[in] field_name`: The field whose conversion to string was unsuccessfully attempted.
- `[in] details`: The details on how the conversion is being done.

~UnknownFieldToStringCast() = default
Destructor.

const std::string &**type**() **const**
Getter method for `_type` parameter.

Return A const reference to the field type string.

const std::string &**field_name**() **const**
Getter method for the field's name.

Return A const reference to the field's name string.

Mix

using eprosima::is::core::**Mix** = *MiddlewareInterfaceExtension*

class eprosima::is::core::**MiddlewareInterfaceExtension**
Abbreviated as "Mix", allows to generate `mix` files which contain the required dynamic libraries for a certain *Integration Service* instance to be loaded.

Also, when talking about a specific *SystemHandle*, `mix` files are used to list the necessary dynamic libraries, containing information about conversion from the specific middleware data type definition (such as ROS 2 *msg*) to xtypes, and viceversa.

Libraries within the `mix` file are listed using the following structure:
- For Linux systems: `"dl" : "../../<relative_path_to_dl>"`
- On Windows platforms: `"dll" : "../../<relative/path_to_dll>"`

These extensions are automatically generated by `is_mix_generator` CMake function and they contain information about specific types, such as their conversion methods to/from `xtypes`.

Public Functions

MiddlewareInterfaceExtension(YAML::Node &&*mix_content*, **const** std::string &*absolute_file_directory_path*)
Constructor.

Parameters
- `[in] mix_content`: Movable reference representing the content of the `mix` file.
- `[in] absolute_file_directory_path`: Absolute path where the `mix` file is stored. This path will be later on used as a starting point to navigate to the relative paths defined by the `mix` file, so that dynamic libraries can be loaded.

MiddlewareInterfaceExtension(**const** *MiddlewareInterfaceExtension* &*other*) = delete
MiddlewareInterfaceExtension shall not be copy constructible.

MiddlewareInterfaceExtension(*MiddlewareInterfaceExtension &&other*)
Move constructor.

Parameters
- `[in] other`: Movable reference to another *MiddlewareInterfaceExtension* instance.

~MiddlewareInterfaceExtension()
Destructor.

bool **load**()
Performs the load operation of the dynamic libraries defined in the `mix` file.

Return `true` if the dynamic libraries were loaded successfully, `false` otherwise.

Public Static Functions

MiddlewareInterfaceExtension **from_file**(**const** std::string *&filename*)
Creates a *MiddlewareInterfaceExtension* representation from a `mix` file path.

Return A properly initialized *MiddlewareInterfaceExtension* object.
Parameters
- `[in] filename`: Path to the `mix` file.

MiddlewareInterfaceExtension **from_string**(**const** std::string *&mix_text*, **const** std::string *&absolute_file_directory_path*)
Creates a *MiddlewareInterfaceExtension* representation from a text YAML representation and an absolute file directory path.

Return A properly initialized *MiddlewareInterfaceExtension* object.
Parameters
- `[in] mix_text`: Content of the `mix` file, in text format.
- `[in] absolute_file_directory_path`: Path from where to start looking for dynamic libraries defined in the `mix` content.

MiddlewareInterfaceExtension **from_node**(YAML::Node *&&node*, **const** std::string *&absolute_file_directory_path*)
Creates a *MiddlewareInterfaceExtension* representation from a YAML node representation and an absolute file directory path.

Return A properly initialized *MiddlewareInterfaceExtension* object.
Parameters
- `[in] node`: Content of the `mix` file, in YAML format.
- `[in] absolute_file_directory_path`: Path from where to start looking for the dynamic libraries defined in the `mix` content.

Search

class `eprosima::is::core::`**Search**
This class searches for *Integration Service* message/service plugin resource files, called *MiddlewareInterfaceExtension* (.mix) files.

These files will be searched based on a fixed lookup scheme. This lookup scheme comprises two phases:

- First, it searchs based on the middleware prefixes `mw_prefix`.
- Second, it searchs based on the *Integration Service* prefixes `<is_prefix>`.

The middleware prefixes and *Integration Service* prefixes can be passed in as command line arguments or set as environment variables. Command line arguments will take precedence over environment variables. The environment variable named `IS_PREFIX_PATH` will be added to the *Integration Service* prefixes `<is_prefix>`, and `IS_<MIDDLEWARE_NAME>_PREFIX_PATH` will be added to the middleware prefixes `mw_prefix`. The environment variables should be a colon-separated list of absolute paths.

Additionally, the contents of the LD_LIBRARY_PATH variable will be added to the *Integration Service* prefixes `<is_prefix>`, because the *resource* directory is expected to be inside a *lib* directory. Finally, `/usr/local/lib/<arch>`, `/usr/local/lib`, `/usr/lib/<arch>`, and `/usr/lib` will be added to the *Integration Service* prefixes `<is_prefix>` in this same order of precedence.

Lookup Scheme

The lookup scheme is described below, where `mw_prefix` and `<is_prefix>` are defined above. `<middleware>` refers to the name of the middleware (as given to the constructor of the *Search* class). `type` is the type name of the message or service. In cases of ROS `<msg|srv|*>`, messages will use `msg` while services use `srv`; when searching for things other than messages or services, a custom string can be substituted for *.

- `<mw_prefix>/<msg|srv|*>/<type>.mix`
- `<mw_prefix>/<type>.mix`
- `<is_prefix>/<middleware>/<msg|srv|*>/<type>.mix`
- `<is_prefix>/<middleware>/<type>.mix`
- `<is_prefix>/is/<middleware>/<msg|srv|*>/<type>.mix`
- `<is_prefix>/is/<middleware>/<type>.mix`

The `type` value will usually look like *package_name/MessageType*. Any slashes within the `type` name will be used as a directory delimiters while searching.

Lookup Pattern

Similarly, the lookup pattern for the base *Integration Service* middleware interface extension file `(<middleware>.mix)` will be:

- `<mw_prefix>/<middleware>.mix`
- `<is_prefix>/<middleware>.mix`
- `<is_prefix>/is/<middleware>.mix`
- `<is_prefix>/is/<middleware>/<middleware>.mix`

Public Functions

Search(**const** std::string &*middleware_name*)
Creates a *Search* utility instance for the specified middleware.

Parameters
- `[in] middleware_name`: The middleware for which a *Search* utility will be created.

Search(**const** *Search* &*other*)
Copy constructor.

Parameters
- `[in] other`: Const reference to the *Search* object to be copied.

Search(*Search* &&*other*)
Move constructor.

Parameters

- `[in] other`: Movable reference of the *Search* object to be moved.

Search &**operator=**(**const** *Search* &*other*)
Copy assignment operator.

Return A reference to *this Search* instance.
Parameters
- `[in] other`: Right assignment operand to be copied to *this Search* object.

Search &**operator=**(*Search* &&*other*)
Copy assignment operator.

Return A reference to *this Search* instance.
Parameters
- `[in] other`: Right assignment operand to be moved to *this Search* object.

~Search()
Destructor.

void **add_priority_middleware_prefix**(**const** std::string &*path*)
Adds priority to the specified path. The paths given here will be used as the first option during the search.

Parameters
- `[in] path`: The path to prioritize.

void **add_fallback_middleware_prefix**(**const** std::string &*path*)
Adds a custom middleware prefix path. The paths given here will be used as `mw_prefix` path prefixes, and will be checked after all other middleware prefixes have been exhausted. The prefix paths passed to this function will be evaluated starting from the path most recently passed in to the first one passed in (i.e. in reverse order).

Parameters
- `[in] path`: An absolute path to use as a middleware prefix search path.

const std::string **find_message_mix**(**const** std::string &*msg_type*, std::vector<std::string> **checked_paths* = nullptr) **const**
Looks for a `mix` file that provides information for a message type.

Return The full path to the .mix file if found. If not found, it will return an empty string.
Parameters
- `[in] msg_type`: This type will be used for `type` in the search scheme.
- `[out] checked_paths`: If given a non-nullptr, it will be filled with a list of the paths that were searched. It may be useful for debugging purposes.

const std::string **find_service_mix**(**const** std::string &*srv_type*, std::vector<std::string> **checked_paths* = nullptr) **const**
Looks for a `mix` file that provides information for a service type.

Return The full path to the .mix file if found. If not found, it will return an empty string.
Parameters
- `[in] srv_type`: This type will be used for `type` in the search scheme.
- `[out] checked_paths`: If given a non-nullptr, it will be filled with a list of the paths that were searched. It may be useful for debugging purposes.

const std::string **find_generic_mix**(**const** std::string &*type*, **const** std::string &*subdir* = "", std::vector<std::string> **checked_paths* = nullptr) **const**
Looks for a `mix` file that provides information for a type which is not a message nor a

service.

Return The full path to the .mix file if found. If not found, it will return an empty string.
Parameters
- `[in] type`: This type will be used for `type` in the search scheme.
- `[in] subdir`: This will replace `<msg|srv|*>` in the search scheme. Leave this as an empty string to not search in a `<msg|srv|*>` subdirectory.
- `[out] checked_paths`: If given a non-nullptr, it will be filled with a list of the paths that were searched. It may be useful for debugging purposes.

const std::string **find_file**(**const** std::string &*filename*, **const** std::string &*subdir* = "", std::vector<std::string> **checked_paths* = nullptr) **const**

Looks for any file (with any extension, not just .mix) that may be residing in an *Integration Service* or middleware subdirectory.

Return The full path to the file if found. If not found, it will return an empty string.
Parameters
- `[in] filename`: The name of the file, including its extension. This should include any nested directories that it may contain relative to the *Integration Service* or middleware directories.
- `[in] subdir`: A subdirectory that might or might not be nested into the *Integration Service* or middleware directories.
- `[out] checked_paths`: If given a non-nullptr, it will be filled with a list of the paths that were searched. It may be useful for debugging purposes.

const std::string **find_middleware_mix**(std::vector<std::string> **checked_paths* = nullptr) **const**

Looks for a `mix` file for the middleware specified during the construction of this *Search* instance.

Parameters
- `[out] checked_paths`: If given a non-nullptr, it will be filled with a list of the paths that were searched. It may be useful for debugging purposes.

Search &**relative_to_config**(bool *toggle* = true)

It can be used to toggle the *Search* to check for files relative to the directory of the config file that was used to launch the *Integration Service*.

By default, this behavior is turned off.

The config-file directory will be treated as a middleware prefix, whose priority comes directly before the "fallback" middleware prefixes. It will be searched after "priority" middleware prefixes, and after any prefix passed in as command line argument or given as environment variables.

Return A reference to this very *Search* instance.
Parameters
- `[in] toggle`: Boolean to enable or disable this behavior.

Search &**relative_to_home**(bool *toggle* = true)

It can be used to toggle the *Search* to check for files relative to the user's home directory.

By default this behavior is turned off.

The home directory will be treated as a middleware prefix, whose priority is the same as the *relative_to_config()* priority, except *relative_to_config()* will have higher priority if both are activated at the same time.

Return A reference to this very *Search* instance.
Parameters

- `[in] toggle`: Boolean to enable or disable this behavior.

Search &**ignore_system_prefixes**(bool *toggle* = true)
It can be used to toggle whether the system prefixes are ignored or not. Note that this has some overlap with the `ignore_is_prefixes` option.

By default these prefixes are not ignored.

Return A reference to this very *Search* instance.
Parameters
- `[in] toggle`: Boolean to enable or disable this behavior.

Search &**ignore_is_prefixes**(bool *toggle* = true)
It can be used to toggle whether all *Integration Service* prefixes are ignored or not. Note that this has some overlap with the `ignore_system_prefixes` option.

By default these prefixes are not ignored.

Return A reference to this very *Search* instance.
Parameters
- `[in] toggle`: Boolean to enable or disable this behavior.

Search &**ignore_middleware_prefixes**(bool *toggle* = true)
It can be used to toggle whether the middleware prefixes are ignored or not.

By default these prefixes are not ignored.

Return A reference to this very *Search* instance.
Parameters
- `[in] toggle`: Boolean to enable or disable this behavior.

Public Static Functions

void **add_cli_is_prefix**(**const** std::string &*path*)
Used by the *Instance* class to set *Integration Service* prefixes that were specified from the command line.

Parameters
- `[in] path`: The path to be added as the *Integration Service* prefix.

void **add_cli_middleware_prefix**(**const** std::string &*middleware*, **const** std::string &*path*)
Used by the *Instance* class to set middleware prefixes that were specified from the command line.

Parameters
- `[in] middleware`: The middleware to which the prefix will be added.
- `[in] path`: The path to be added as the *Integration Service* prefix.

void **set_config_file_directory**(**const** std::string &*path*)
Used by the *Instance* class to set the path where the configuration file is stored.

Parameters
- `[in] path`: The path to be set as the configuration directory.

const std::string **to_env_format**(**const** std::string &*str*)
Convert a given string to environment format.

Return A properly formatted string to the env format.
Parameters
- `[in] str`: The string to be converted.

StringTemplate

class eprosima::is::core::**StringTemplate**

Allows to create a partially filled string with certain parameterizable fields that can be replaced during runtime. It is also possible to specify some details on how the template should be used.

More information about how to construct and properly use it is available on the *StringTemplate* constructor.

Public Functions

StringTemplate(**const** std::string &*template_string*, **const** std::string &*usage_details*)

Constructor.

Parameters

- `[in] template_string`: A string that describes the desired template. Varying components of the string must be wrapped in curly braces `{}`. Currently only `{message.<field>}` variables are supported. The varying components of the string will be replaced by the value of the requested field when *`compute_string()`* is called.
- `[in] usage_details`: A string that describes how this *StringTemplate* is being used.

StringTemplate(**const** *StringTemplate* &*other*)

Copy constructor.

Parameters

- `[in] other`: const reference to a *StringTemplate* instance to be copied.

StringTemplate(*StringTemplate* &&*other*)

Move constructor.

Parameters

- `[in] other`: Movable reference to a *StringTemplate* instance.

~StringTemplate()

Destructor.

const std::string **compute_string**(**const** eprosima::xtypes::DynamicData &*message*) **const**

Computes the desired output string, given the input message.

Return The computed string with the required substitutions properly made.

Parameters

- `[in] message`: The message used to compute the string template parameters.

std::string &**usage_details**()

Gets a mutable reference to the usage_details for this *StringTemplate*.

Return The mutable reference to the `usage_details` string.

const std::string &**usage_details**() **const**

Gets a const reference to the usage_details for this *StringTemplate*.

Return A const reference to the `usage_details` string.

class eprosima::is::core::**InvalidTemplateFormat** : **public** runtime_error

Runtime error that gets thrown when a certain runtime substitution string template is malformed in the *YAML* file.

Public Functions

InvalidTemplateFormat(**const** std::string &*template_string*, **const** std::string &*details*)
Constructor.

Parameters
- `[in] template_string`: The source string containing the malformed template.
- `[in] details`: Correct usage details of this template.

~InvalidTemplateFormat() = default
Destructor.

const std::string &**template_string**() **const**
Gets a const reference to the malformed *StringTemplate*.

Return A const reference to the template.

class eprosima::is::core::**UnavailableMessageField** : **public** runtime_error
Runtime error that gets thrown when a certain field, required to perform the substitution in a *StringTemplate*, is unavailable within the provided Dynamic Data.

Public Functions

UnavailableMessageField(**const** std::string &*field_name*, **const** std::string &*details*)
Constructor.

Parameters
- `[in] field_name`: The field which was not found during the substitution.
- `[in] details`: Details on how to use this template.

const std::string &**field_name**() **const**
Gets a const reference to the field's name.

Return A const reference to the string representing the field's name.

6.13.2 System Handle

This section of the API reference corresponds to the include/is/systemhandle folder of the *Integration Service* main repository.

This folder contains the files that allow the creation of new *System Handles* and their registry within the system.

RegisterSystem

class eprosima::is::internal::**Register**
Static class that contains a static map of *is::detail::SystemHandleFactoryBuilder* instances.

is::detail::SystemHandleFactoryBuilder is nothing but a function signature that helps in the creation of a `std::unique_ptr<SystemHandle>` object.

In this way, each time a given *SystemHandle* instance is required, it will be created from the factory map.

Public Static Functions

void **insert**(std::string &&*middleware*, detail::*SystemHandleFactoryBuilder* &&*handle*)
Inserts a new *is::detail::SystemHandleFactoryBuilder* element in the factory map.

Parameters

- `[in] middleware`: The middleware's name.
- `[in] handle`: The handle function responsible for creating the *SystemHandle* instance.

SystemHandleInfo **get**(**const** std::string &*middleware*)
Gets the *SystemHandleInfo* object associated to a given middleware.

Return A *SystemHandleInfo* object which is properly initialized if the middleware exists and it is registered within the *Register*, or pointing to `nullptr` otherwise.

Parameters

- `[in] middleware`: The middleware from which we want to obtain a *SystemHandleInfo* instance.

using eprosima::is::internal::**SystemHandleInfoMap** = std::map<std::string, *SystemHandleInfo*>

class eprosima::is::internal::**SystemHandleInfo**
Storage class that holds all the information relative to a certain *SystemHandle* instance.

This class will retrieve the corresponding *TopicPublisherSystem*, *TopicSubscriberSystem*, *ServiceClientSystem* and *ServiceProviderSystem* instances associated to the *SystemHandle* instance, if applicable.

If not applicable, these instances will just be cast to `nullptr`. Later on, this will allow to know whether a certain *SystemHandle* comes or not with any of these four working capabilities.

Also, a *is::TypeRegistry* is defined, where all the types that the *SystemHandle* instance must know prior to start performing any conversion are defined.

Public Functions

SystemHandleInfo(std::unique_ptr<*SystemHandle*> *input*)
Constructor.

Parameters

- `[in] input`: The *SystemHandle* instance which we want to obtain information from.

SystemHandleInfo(**const** *SystemHandleInfo* &*other*) = delete
SystemHandleInfo shall not be copy constructible.

SystemHandleInfo(*SystemHandleInfo* &&*other*)
Move constructor.

Parameters

- `[in] other`: A movable reference to other *SystemHandleInfo* instances.

~SystemHandleInfo() = default
Destructor.

operator bool() const
`bool()` operator overload.

Return `true` if the pointer to the handle is not `nullptr`, `false` otherwise.

Public Members

std::unique_ptr<*SystemHandle*> **handle**
Class members.

System Handle

class `eprosima::is::SystemHandle`
It is the base interface class for all middleware systems.

All middleware systems that want to interact with *Integration Service* must implement, at least, this interface.

Depending on the type of middleware, it should also implement the derived classes, using multiple virtual inheritance:

- *TopicSubscriberSystem*: provides subscribing capabilities.
- *TopicPublisherSystem*: provides publishing capabilities.
- *ServiceClientSystem*: allows to manage middleware service clients.
- *ServiceProviderSystem*: allows to manage middleware service servers.

A *SystemHandle* implementing the four interfaces described above is called a *FullSystem*, and it is usually the base class used for implementing a middleware plugin for the *Integration Service*.

Subclassed by *eprosima::is::ServiceClientSystem*, *eprosima::is::ServiceProviderSystem*, *eprosima::is::TopicPublisherSystem*, *eprosima::is::TopicSubscriberSystem*

Public Functions

SystemHandle() = default
Default constructor.

SystemHandle(**const** *SystemHandle*&) = delete
SystemHandle shall not be copy constructible.

SystemHandle &**operator=**(**const** *SystemHandle*&) = delete
SystemHandle shall not be copy assignable.

SystemHandle(*SystemHandle*&&) = delete
SystemHandle shall not be move constructible.

SystemHandle &**operator=**(*SystemHandle*&&) = delete
SystemHandle shall not be move assignable.

~SystemHandle() = default
Destructor.

bool **configure**(**const** core::*RequiredTypes* &*types*, **const** YAML::Node &*configuration*, *TypeRegistry* &*type_registry*) = 0
Configures the *Integration Service* handle for this middleware's system.

Return `true` if the configuration process was successful, `false` otherwise.

Parameters

- `[in] types`: The set of types (messages and services) that this middleware needs to support. The *SystemHandle* must register this type into the *is::TypeRegistry*, using for that the storage class *is::internal::SystemHandleInfo*.
- `[in] configuration`: The configuration specific for this *SystemHandle*, as described in the user-provided *YAML* input file. See the specific *SystemHandle* implementation documentation for a list of accepted configuration parameters for each middleware.
- `[in] type_registry`: The set of type definitions that this middleware is able to support.

bool **`okay`**`()` **`const`** = 0
Method that allows to check if a *SystemHandle* is correctly working.

Return `true` if the *SystemHandle* is under normal behavior, `false` otherwise.

`operator bool``()` **`const`**
`bool()` operator overload. Implicit conversion, same as *`okay()`*.

Return `true` if the *SystemHandle* is under normal behavior, `false` otherwise.

bool **`spin_once`**`()` = 0
Tell the *SystemHandle* to spin once, e.g. read through its subscriptions.

Return `true` if the *SystemHandle* is still working; `false` otherwise.

`class` `eprosima::is::`**`TopicSubscriberSystem`** : **`public virtual`** eprosima::is::*SystemHandle*
Extends the *SystemHandle* class with subscription capabilities.

Subclassed by *eprosima::is::TopicSystem*

Public Types

`using SubscriptionCallback` = std::function<void(**`const`** xtypes::DynamicData &message, void *filter_handle)>
Signature of the callback that gets triggered when a subscriber receives some data.

Public Functions

`TopicSubscriberSystem``()` = default
Constructor.

`~TopicSubscriberSystem``()` = default
Destructor.

bool **`subscribe`**`(`**`const`** std::string &*topic_name*, **`const`** xtypes::DynamicType &*message_type*, *SubscriptionCallback* **callback*, **`const`** YAML::Node &*configuration*`)` = 0
Has this *SystemHandle* instance subscribed to a topic.

Return `true` if subscription was successfully established, `false` otherwise.

Parameters

- `[in] topic_name`: Name of the topic to get subscribed to.

- `[in] message_type`: Message type that this topic should expect to receive.
- `[in] callback`: The callback which should be triggered when a message comes in.
- `[in] configuration`: A *YAML* node containing any middleware-specific configuration information for this subscription. This may be an empty node.

bool **is_internal_message**(void **filter_handle*) = 0
Check if a certain message in a subscriber comes from a middleware publisher created by *Integration Service* in the same *SystemHandle* instance.

This method must be implemented by each *SystemHandle* according to its middleware and protocol intricacies and particularities. Some protocols might not need this at all. This method is called, during the SubscriptionCallback function, to avoid sending messages indefinitely, thus creating an infinite loop.

Parameters

- `[in] filter_handle`: Opaque pointer to entity containing the information used to perform the filtering; this is usually meta-information regarding the just received message instance in the middleware's subscriber side.

class `eprosima::is::`**TopicPublisher**
This is the abstract interface for objects that can act as publisher proxies.

These objects will be created by *Integration Service* as bridges between the common data representation (`eprosima::xtypes`) and the *user subscription applications*, when data are to be published from one middleware to another.

These objects should be generated by the *TopicPublisherSystem* `advertise()` method.

Public Functions

TopicPublisher() = default
Constructor.

~TopicPublisher() = default
Destructor.

bool **publish**(**const** xtypes::DynamicData &*message*) = 0
Publishes to a topic.

Return `true` if the data was correctly published, `false` otherwise.

Parameters

- `[in] message`: DynamicData that is being published.

class `eprosima::is::`**TopicPublisherSystem** : **public virtual** eprosima::is::*SystemHandle*
This class extends the *SystemHandle* class by providing it with publishing capabilities.

Subclassed by *eprosima::is::TopicSystem*

Public Functions

TopicPublisherSystem() = default
Constructor.

~TopicPublisherSystem() = default
Destructor.

std::shared_ptr<*TopicPublisher*> **advertise**(**const** std::string &*topic_name*, **const** xtypes::DynamicType &*message_type*, **const** YAML::Node &*configuration*) = 0
Advertises the ability to publish to a topic.

Return `true` if the advertisement was successful, `false` otherwise.

Parameters

- `[in] topic_name`: Name of the topic to advertise.
- `[in] message_type`: Message type that this entity will publish.
- `[in] configuration`: A *YAML* node containing any middleware-specific configuration information for this publisher. This may be an empty node.

class eprosima::is::**TopicSystem** : **public virtual** eprosima::is::*TopicPublisherSystem*, **public virtual** eprosima
It is the conjunction of *TopicPublisherSystem* and *TopicSubscriberSystem*. It allows to create a middleware library for *Integration Service* fully compatible with the publish/subscribe paradigm.

Subclassed by *eprosima::is::FullSystem*

Public Functions

TopicSystem() = default
Constructor.

~TopicSystem() = default
Destructor.

class eprosima::is::**ServiceClient**
This is the abstract interface for objects that can act as client proxies.

This class is different from *ServiceClientSystem*, because *ServiceClientSystem* is the interface for *SystemHandle* libraries that are able to support client proxies, whereas *ServiceClient* is the interface for the client proxy objects themselves.

This class, when overridden by the specific middleware implementation, will typically contain a `middleware::server` object, so that `receive_response` can fetch the response sent by the *user server application* (usually, implemented using a different middleware) and pass this response to the *target user client application*, which will receive the final response by means of the internal server created by this *ServiceClient*.

Public Functions

ServiceClient() = default
Constructor.

~ServiceClient() = default
Destructor.

void **receive_response**(std::shared_ptr<void> *call_handle*, **const** xtypes::DynamicData &*response*) = 0
Receives the response of a service request.

Attention Services are assumed to all be asynchronous (non-blocking), so this function may be called by multiple threads at once. developers implementing a *ServiceClient* derived class must make sure that they can handle multiple simultaneous calls to this function.

Parameters

- `[in] call_handle`: The handle that was given to the call by this *ServiceClient*. The usage of the handle is determined by the *ServiceClient* implementation. Typically, `receive_response` will cast this handle into a useful object type that contains information on where to send the service response message.
- `[in] response`: The message that represents the response from the service.

class eprosima::is::**ServiceClientSystem** : **public virtual** eprosima::is::*SystemHandle*
This class extends the *SystemHandle* class with service client handling capabilities.

Subclassed by *eprosima::is::ServiceSystem*

Public Types

using RequestCallback = std::function<void(**const** xtypes::DynamicData &request, *ServiceClient* &client, std::shared_ptr<void> call_handle)>
Signature of the callback that gets triggered when a client has made a request.

Public Functions

ServiceClientSystem() = default
Constructor.

~ServiceClientSystem() = default
Destructor.

bool **create_client_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*service_type*, *RequestCallback* *callback, **const** YAML::Node &*configuration*)
Create a proxy for a client application.

Return `true` if a client proxy could be created, `false` otherwise.

Parameters

- `[in] service_name`: Name of the service this client proxy shall listen to.
- `[in] service_type`: Service request and reply type to expect.

- `[in] callback`: The callback that should be used when a request comes in from the middleware.
- `[in] configuration`: A *YAML* node containing any middleware-specific configuration information for this service client. This may be an empty node.

bool **`create_client_proxy`**(**`const`** std::string &*service_name*, **`const`** xtypes::DynamicType &*request_type*, **`const`** xtypes::DynamicType &*reply_type*, *RequestCallback* **callback*, **`const`** YAML::Node &*configuration*)

Create a proxy for a client application.

Return `true` if a client proxy could be created, `false` otherwise.

Parameters

- `[in] service_name`: Name of the service for this client to listen to.
- `[in] request_type`: Type of service request to expect.
- `[in] reply_type`: Type of service reply to expect.
- `[in] callback`: The callback that should be used when a request comes in from the middleware.
- `[in] configuration`: A *YAML* node containing any middleware-specific configuration information for this service client. This may be an empty node.

`class` `eprosima::is::`**`ServiceProvider`**

This is the abstract interface for objects that can act as service server proxies.

This class is different from *ServiceProviderSystem*, because *ServiceProviderSystem* is the interface for *SystemHandle* libraries that are able to support service server proxies, whereas *ServiceProvider* is the interface for the service server proxy objects themselves.

This class, when overridden by the specific middleware implementation, will typically contain a `middleware::client` object that will actually send the request to the *user server application*. After processing the request by means of the `call_service` method, thanks to the associated *ServiceClient* entity, `receive_response` will be called, to pass the response to the *user client application* (typically, implemented using a different middleware, which justifies the use of the *Integration Service* to interconnect them).

Public Functions

`ServiceProvider``()` = default

Constructor.

`~ServiceProvider``()` = default

Destructor.

void **`call_service`**(**`const`** xtypes::DynamicData &*request*, *ServiceClient* &*client*, std::shared_ptr<void> *call_handle*) = 0

Call a service.

Attention It is important that this function:

i. Is **non-blocking**.

ii. Calls `client.receive_response()` when the service finishes.

Parameters

- `[in] request`: Request message for the service.

- `[inout] client`: The proxy for the client that is making the request.
- `[in] call_handle`: A handle for the call. The usage of this handle is determined by the *ServiceClient* implementation. The *ServiceProvider* should not attempt to cast or modify it in any way; it should only be passed back to the *ServiceClient* later on, when `receive_response()` is called.

class `eprosima::is::` **ServiceProviderSystem** : **public virtual** eprosima::is::*SystemHandle*

This class extends the *SystemHandle* class with service server handling capabilities.

Subclassed by *eprosima::is::ServiceSystem*

Public Functions

ServiceProviderSystem`()` = default

Constructor.

~ServiceProviderSystem`()` = default

Destructor.

std::shared_ptr<*ServiceProvider*> **create_service_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*service_type*, **const** YAML::Node &*configuration*)

Create a proxy for a service server.

Return `true` if the middleware's *SystemHandle* can offer this service, `false` otherwise.

Parameters

- `[in] service_name`: Name of the service to offer.
- `[in] service_type`: Type of service being offered.
- `[in] configuration`: A *YAML* node containing any middleware-specific configuration information for this service provider. This may be an empty node.

std::shared_ptr<*ServiceProvider*> **create_service_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*request_type*, **const** xtypes::DynamicType &*reply_type*, **const** YAML::Node &*configuration*)

Creates a proxy for a service server.

Return `true` if the middleware's *SystemHandle* can offer this service, `false` otherwise.

Parameters

- `[in] service_name`: Name of the service to offer.
- `[in] request_type`: Type of service request being offered.
- `[in] reply_type`: Type of service reply being offered.
- `[in] configuration`: A *YAML* node containing any middleware-specific configuration information for this service provider. This may be an empty node.

class `eprosima::is::` **ServiceSystem** : **public virtual** eprosima::is::*ServiceClientSystem*, **public virtual** eprosim

It is the conjunction of *ServiceProviderSystem* and *ServiceClientSystem*. Allows to create a middleware library for *Integration Service* fully compatible with the request/reply paradigm.

Subclassed by *eprosima::is::FullSystem*

Public Functions

ServiceSystem() = default
Constructor.

~ServiceSystem() = default
Destructor.

class eprosima::is::**FullSystem** : **public virtual** eprosima::is::*TopicSystem*, **public virtual** eprosima::is::*Servic*
It is the conjunction of *ServiceSystem* and *TopicSystem*. It allows to define a whole middleware, in terms of both publish/subscribe and request/reply paradigms.

Usually, most middleware plugins for *Integration Service* will inherit from this class.

Public Functions

FullSystem() = default
Constructor.

~FullSystem() = default
Destructor.

struct eprosima::is::core::**RequiredTypes**
Contains the set of topics and services types required in order to successfully create an *Integration Service* instance, based on the configuration provided.

Public Members

std::set<std::string> **messages**
Set of topic types stated within the configuration file.

std::set<std::string> **services**
Set of service types stated within the configuration file.

using eprosima::is::**TypeRegistry** = std::map<std::string, xtypes::DynamicType::Ptr>
Map used to store the DynamicType name mapped to its representation.

IS_REGISTER_SYSTEM(*middleware_name_str*, *SystemType*)
Call this macro in a .cpp file of your middleware's plugin library, so that the *Integration Service* can find your *eprosima::is::SystemHandle* implementation when your plugin library gets dynamically loaded. For example:

```
IS_REGISTER_SYSTEM("my_middleware", my::middleware::SystemHandle)
```

The first argument should be a string representing the name of the middleware. This should match the name in the `system:` dictionary of your *Integration Service* configuration file. Each middleware should have a unique name.

The second argument should be the literal type (not a string) of the class that implements *eprosima::is::SystemHandle* in your plugin library.

SystemHandleFactory

template<typename **SystemHandleImplType**>
class eprosima::is::detail::**SystemHandleRegistrar**

Builder class to help register any *SystemHandle* kind, during runtime.

Template Parameters

- `SystemHandleImplType`: The *is::SystemHandle* overridden implementation kind, for a certain middleware.

Public Functions

SystemHandleRegistrar(std::string &&*middleware*)

Constructor.

Parameters

- `[in] middleware`: The middleware name to be registered into the factory.

using eprosima::is::detail::**SystemHandleFactoryBuilder** = std::function<std::unique_ptr<*SystemHandle*>()>

Signature of the function that gets triggered when a new *SystemHandle* instance is created.

void eprosima::is::detail::**register_system_handle_factory**(std::string &&*middleware*, *SystemHandleFactoryBuilder* &&*handle*)

Wrapper method for *is::internal::Register::insert*.

Parameters

- `[in] middleware`: The middleware's name.
- `[in] handle`: The handle function responsible for creating the *SystemHandle* instance.

6.13.3 Utils

This section of the API reference corresponds to the include/is/utils folder of the *Integration Service* main repository.

This folder contains the logger tool and the conversion library used for those middlewares that use static types, such as *ROS 2* and *ROS 1*.

Convert

template<typename **Type**>
struct eprosima::is::utils::**Convert**

A utility to help with converting data between generic DynamicData field objects and middleware-specific data structures.

This struct will work as-is on primitive types (a.k.a. arithmetic types or strings), but a template specialization for converting to or from any complex class types should be created.

Public Types

using native_type = *Type*
Alias for the Type.

Public Static Functions

void **from_xtype_field**(**const** xtypes::ReadableDynamicDataRef &*from*, *native_type* &*to*)
Move data from a xTypes DynamicData field to a native middleware data structure.

Parameters

- `[in] from`: A readable reference to the DynamicData field to be transferred.
- `[in] to`: The destination native middleware data structure.

void **to_xtype_field**(**const** *native_type* &*from*, xtypes::WritableDynamicDataRef *to*)
Move data from a native middleware data structure to a xTypes DynamicData field.

Parameters

- `[in] from`: A readable reference to the middleware native data structure to be transferred.
- `[in] to`: A writable reference to the target DynamicData field.

Public Static Attributes

constexpr bool **type_is_primitive** = std::is_arithmetic<*Type*>::value || std::is_same<std::string, *Type*>::value || std::is_s
Const expression to check if the type is primitive or not.

struct `eprosima::is::utils::`**CharConvert**
A class that helps create a Convert<> specialization for managing some char issues.

'rosidl' parse 'char' types as 'signed' values from 'msg' files and parse as 'unsigned' from idl files. This create a mismatched between types. This patch solves this issue when the native type differs from the DynamicData type for this specific case.

Note This specialization can be removed safety if rosidl modifies its behavior.

Subclassed by eprosima::is::utils::Convert< char >

Public Types

using native_type = char
Alias for the Type.

Public Static Functions

void **from_xtype_field**(**const** xtypes::ReadableDynamicDataRef &*from*, *native_type* &*to*)
Documentation inherited from *Convert*.

void **to_xtype_field**(**const** *native_type* &*from*, xtypes::WritableDynamicDataRef *to*)
Documentation inherited from *Convert*.

template<typename **ElementType**, template<typename, typename> class **NativeType**, typename **Allocator**, std::size_t **Upper**
struct eprosima::is::utils::**ResizableUnboundedContainerConvert**
A class that helps create a Convert<> specialization for resizable unbounded container message types.

To create a specialization for a native middleware message type, do the following:

```
namespace eprosima {
namespace is {
namespace utils {

template<typename ElementType, typename Allocator>
struct Convert<native::middleware::type<ElementType, Allocator> >
    : ResizableUnboundedContainerConvert<
        ElementType,
        native::middleware::type,
        Allocator,
        size_t::upperbound::limit
    > { };

} // namespace utils
} // namespace is
} // namespace eprosima
```

Note The UpperBound limit could typically be calculated as

```
std::numeric_limits<typename native::middleware::type<ElementType, Allocator>
↪::size_type>::max()
```

Public Static Functions

void **from_xtype**(**const** xtypes::ReadableDynamicDataRef &*from*, std::vector<bool>::reference *to*)
This template specialization is needed to deal with the edge case produced by vectors of bools. std::vector<bool> is specialized to be implemented as a bitmap, and as a result its operator[] cannot return its bool elements by reference. Instead it returns a "reference" proxy object.

void **from_xtype_field**(**const** xtypes::ReadableDynamicDataRef &*from*, native_type &*to*)
Documentation inherited from *Convert*.

void **to_xtype_field**(**const** native_type &*from*, xtypes::WritableDynamicDataRef *to*)
Documentation inherited from *Convert*.

template<typename **ElementType**, template<typename, std::size_t, typename> class **NativeType**, typename **Allocator**, std::siz
struct eprosima::is::utils::**ResizableBoundedContainerConvert**
A class that helps create a Convert<> specialization for resizable bounded container message types.

To create a specialization for a native middleware message type, do the following:

```
namespace eprosima {
namespace is {
```

(continues on next page)

(continued from previous page)

```
namespace utils {

template<typename ElementType, std::size_t N, typename Allocator, template
↪<typename, std::size_t, typename> class VectorImpl>
struct Convert<VectorImpl<ElementType, N, Allocator> >
    : ResizableBoundedContainerConvert<
        ElementType,
        VectorImpl,
        Allocator,
        N
    > { };

} // namespace utils
} // namespace is
} // namespace eprosima
```

Public Static Functions

void **from_xtype_field**(**const** xtypes::ReadableDynamicDataRef &*from*, native_type &*to*)
Documentation inherited from *Convert*.

void **to_xtype_field**(**const** native_type &*from*, xtypes::WritableDynamicDataRef *to*)
Documentation inherited from *Convert*.

template<typename **ElementType**, template<typename, std::size_t> class **NativeType**, std::size_t **UpperBound**, std::enable_if_t
struct eprosima::is::utils::**NonResizableContainerConvert**

A class that helps create a Convert<> specialization for non resizable container message types.

To create a specialization for a native middleware message type, do the following:

```
namespace eprosima {
namespace is {
namespace utils {

template<template <typename, std::size_t> class Array, typename ElementType,␣
↪std::size_t N>
struct Convert<Array<ElementType, N> >
    : NonResizableContainerConvert<
        ElementType,
        Array,
        N
    > { };

} // namespace utils
} // namespace is
} // namespace eprosima
```

Public Static Functions

void **from_xtype_field**(**const** xtypes::ReadableDynamicDataRef &*from*, native_type &*to*)
Documentation inherited from *Convert*.

void **to_xtype_field**(**const** native_type &*from*, xtypes::WritableDynamicDataRef *to*)
Documentation inherited from *Convert*.

Log

class eprosima::is::utils::**Logger**
Allows to easily log information into the standard output. It should be used as the preferred method for printing information within the whole Integration Service suite (core and *SystemHandle*).

Public Functions

Logger() = default
Default constructor.

Logger(**const** std::string &*header*)
Constructor.

Parameters

- `[in] header`: The user-defined headed that will be printed at the beginning of every logger's message.

Logger(**const** *Logger*&) = default
Copy constructor.

Logger(**const** *Logger*&&) = delete
Logger shall not be move constructible.

~Logger() = default
Destructor.

const *Level* &**get_level**() **const**
Get the maximum logging level for this *Logger* instance.

Return A non-mutable reference to the maximum permitted logging level: `DEBUG`, `INFO`, `WARN`, `ERROR`.

Logger &**operator<<**(**const** *Level* &*level*)
Operator << overload for a certain logging *Level*. Sets the logging level for the char/string messages streamed afterwards, until std::endl is received.

Return A reference to this object.

Parameters

- `[in] level`: The logging *Level* of a new upcoming message.

Logger &**operator<<**(**const** char **message*)
Operator << overload for a certain message.

Return A reference to this object.

Parameters

- `[in] message`: The message to be printed to stdout.

Logger &**operator<<**(**const** std::string &*message*)
Operator << overload for a certain message.

Return A reference to this object.

Parameters

- `[in] message`: The message to be printed to stdout.

template<typename **T**>
Logger &**operator<<**(**const** *T* &*value*)
Operator << overload for arithmetic types.

Return A reference to this object.

Parameters

- `[in] value`: A const reference to the numeric value.

Logger &**operator<<**(std::basic_ostream<char, std::char_traits<char>> &(**func*)) std::basic_ostream<char, std::char_traits<char>>&
Operator << overload for ostream function pointer. Useful for streaming special operations, such as std::endl.

Return A reference to this object.

Parameters

- `[in] func`: Pointer to std::ostream function.

class Level
Enumeration holding all the possible logging values. Messages logged with a logging priority level lower than the configured maximum level will not be displayed. This level is configurable via CMake parameters and can also be set using the provided `set_logging_level` method API.

- **Values**:
  - *Level*::**ERROR**
  - *Level*::**WARN**
  - *Level*::**INFO**
  - *Level*::**DEBUG**

class CurrentLevelStatus
Enumeration class which stores all the possible statuses for the current operation in the logger.

- **Values**:
  - *CurrentLevelStatus*::**NON_SPECIFIED**
  - *CurrentLevelStatus*::**SPECIFIED**
  - *CurrentLevelStatus*::**SPECIFIED_BUT_HIDDEN**

6.14 Fast DDS System Handle

This section presents the API provided by the *Integration Service* `is-fastdds` library.

6.14.1 Client

class `eprosima::is::sh::fastdds::Client` : **public virtual** eprosima::is::*ServiceClient*, **private** DataWriterLis

This class represents a DDS *Client*, built over the publisher/subscriber layer of *Fast DDS* using the DDS-RPC paradigm, within the *Integration Service* framework.

It is composed of a *Fast DDS Subscriber*, to listen for requests coming from the DDS dataspace; plus a *Fast DDS Publisher*, to send replies from the *Integration Service* back to the DDS service client application.

Its topic type definition and data instances for request and reply types are represented by means of the Fast DDS Dynamic Types API, which allows to get rid of TypeSupport for each used type and eases users the task of defining and using their own custom types on the go, by means of a valid *IDL* definition.

This class inherits from Fast DDS Data Reader Listener and from Fast DDS Data Writer Listener for reacting to datawriter and datareader events, such as matching with subscribers and publishers or receiving new data from them.

The request petitions are associated with each received reply by means of the sample identity and the related sample identity attributes.

Public Functions

Client (eprosima::is::sh::fastdds::*Participant* *participant, **const** std::string &*service_name*, **const** xtypes::DynamicType &*request_type*, **const** xtypes::DynamicType &*reply_type*, *ServiceClientSystem*::*RequestCallback* *callback, **const** YAML::Node &*config*)

Construct a new *Client* object.

Parameters

- `[in] participant`: The associated *Integration Service Participant*, which holds the DDS entities that compose this *Client*.
- `[in] service_name`: The service name. It will produce two topics: `<service_name>_Request` and `<service_name>_Reply`.
- `[in] request_type`: A dynamic type definition of the request topic's type.
- `[in] reply_type`: A dynamic type definition of the reply topic's type.
- `[in] callback`: Callback that gets triggered when a client has made a request.
- `[in] config`: Additional configuration that might be required to configure this *Client*.

~Client () **override**

Destroy the *Client* object.

Client (**const** *Client* &*rhs*) = delete

Client shall not be copy constructible.

Client &**operator=** (**const** *Client* &*rhs*) = delete

Client shall not be copy assignable.

Client (*Client* &&*rhs*) = delete

Client shall not be move constructible.

Client &**operator=**(*Client* &&*rhs*) = delete
Client shall not be move assignable.

void **receive_response**(std::shared_ptr<void> *call_handle*, **const** xtypes::DynamicData &*response*) **override**
Inherited from *ServiceClient*.

bool **add_config**(**const** YAML::Node &*configuration*, *ServiceClientSystem*::*RequestCallback* **callback*)
Handle type remappings for DDS request and reply types. It allows to resolve complex type remappings, which remap to a specific type member, for example, an UnionType member, by means of the dot **.** operator.

Parameters

- `[in] configuration`: The *YAML* configuration containing the remapping to be applied.
- `[in] callback`: The callback that gets triggered when a client has made a request.

6.14.2 DDSMiddlewareException

class `eprosima::is::sh::fastdds::`**DDSMiddlewareException** : **public** runtime_error
Launches a runtime error every time an unexpected behavior occurs related to *Fast DDS* middleware, when configuring or using this *is::SystemHandle*.

Public Functions

DDSMiddlewareException(**const** utils::*Logger* &*logger*, **const** std::string &*message*)
Construct a new *DDSMiddlewareException* object.

Parameters

- `[in] logger`: The logging tool.
- `[in] message`: The message to throw the runtime error with.

6.14.3 Participant

class `eprosima::is::sh::fastdds::`**Participant**
This class represents a FastDDS DomainParticipant within the *Integration Service* framework.

It includes a mapping of the topic names to their corresponding Dynamic Type representation, and also mappings to identify each topic with its type.

This class inherits from Fast DDS DomainParticipantListener class to scan for state changes on the DDS participant created by this *Integration Service* *is::SystemHandle*.

Public Functions

Participant()
Construct a new *Participant*, with default values.

Exceptions

- `DDSMiddlewareException`: If the *DomainParticipant* could not be created.

Participant(**const** YAML::Node &*config*)
Construct a new *Participant* object with the user-provided parameters in the *YAML* configuration file.

- `file_path`: Specifies the path to the XML profile that will be used to configure the *DomainParticipant*. More information on how to write these XML profiles can be found here.
- `profile_name`: Provide a name to search for within the profiles defined in the XML that corresponds to the configuration profile that we want this *Participant* to be configured with.

Parameters

- `[in] config`: The configuration provided by the user. It must contain two keys in the *YAML* map:

Exceptions

- `DDSMiddlewareException`: If the XML profile was incorrect and, thus, the *DomainParticipant* could not be created.

~Participant()
Destroy the *Participant* object.

void **build_participant**(**const** fastdds::dds::DomainId_t &*domain_id* = 0)
Construct a *Fast DDS DomainParticipant*, given its DDS domain ID.

Parameters

- `[in] domain_id`: The DDS domain ID for this participant.

Exceptions

- `DDSMiddlewareException`: If the *DomainParticipant* could not be created.

fastdds::dds::DomainParticipant ***get_dds_participant**() **const**
Get the associate *FastDDS DomainParticipant* attribute.

Return The DDS participant.

void **register_dynamic_type**(**const** std::string &*topic_name*, **const** std::string &*type_name*, fastrtps::types::DynamicTypeBuilder **builder*)
Register a *Dynamic Type* within the types map. Also, register the associated DDS topic.

Parameters

- `[in] topic_name`: The topic name to be associated to the *Dynamic Type*.
- `[in] type_name`: The type name to be registered in the factory.
- `[in] builder`: A class that represents a builder for the desired *Dynamic Type*.

Exceptions

- *DDSMiddlewareException*: If the type could not be registered.

fastrtps::types::DynamicData ***create_dynamic_data**(**const** std::string &*topic_name*) **const**
Create an empty dynamic data object for the specified topic.

Return The empty DynamicData for the required topic.

Parameters

- `[in] topic_name`: The topic name.

Exceptions

- *DDSMiddlewareException*: if the topic was not found or the type was not registered previously.

void **delete_dynamic_data**(fastrtps::types::DynamicData **data*) **const**
Delete a certain dynamic data from the *DomainParticipant* database.

Parameters

- `[in] data`: The dynamic data to be deleted.

const fastrtps::types::DynamicType ***get_dynamic_type**(**const** std::string &*name*) **const**
Get the dynamic type pointer associated to a certain key.

Return The pointer to the dynamic type if found, or `nullptr` otherwise.

Parameters

- `[in] name`: The key to find within the types map.

const std::string &**get_topic_type**(**const** std::string &*topic*) **const**
Get the type name associated to a certain topic.

Return A const reference to the topic type's name.

Parameters

- `[in] topic`: The topic whose type is wanted to be retrieved.

void **associate_topic_to_dds_entity**(fastdds::dds::Topic **topic*, fastdds::dds::DomainEntity **entity*)
Register a topic into the topics map.

Note This method is a workaround until `fastdds::dds::DomainParticipant::find_topic` gets implemented.

Parameters

- `[in] topic`: The name of the topic to register.
- `[in] entity`: A pointer to the entity to be registered.

bool **dissociate_topic_from_dds_entity**(fastdds::dds::Topic **topic*, fastdds::dds::DomainEntity **entity*)
Unregister a topic from the topics map.

Note This method is a workaround until `fastdds::dds::DomainParticipant::find_topic` gets implemented.

Parameters

- `[in] topic`: The name of the topic to unregister.
- `[in] entity`: A pointer to the entity to be unregistered.

6.14.4 Publisher

class `eprosima::is::sh::fastdds::Publisher` : **public virtual** eprosima::is::*TopicPublisher*, **private** DataWri

This class represents a Fast DDS Publisher within the *Integration Service* framework.

Its topic type definition and data instances are represented by means of the Fast DDS Dynamic Types API, which allows to get rid of TypeSupport for each used type and eases users the task of defining and using their own custom types on the go, by means of a valid *IDL* definition.

This class inherits from Fast DDS Data Writer Listener for reacting to datawriter events, such as matching with subscribers.

Public Functions

Publisher(*Participant* *participant, **const** std::string &topic_name, **const** xtypes::DynamicType &message_type, **const** YAML::Node &config)

Construct a new *Publisher* object.

Parameters

- `[in] participant`: The associated *Integration Service Participant*, that holds this *Publisher*.
- `[in] topic_name`: The topic that this DDS publisher will send data to.
- `[in] message_type`: A dynamic type definition of the topic's type.
- `[in] config`: Specific configuration regarding this publisher, in *YAML* format. Allowed fields are:
  - `service_instance_name`: Specify the DDS RPC service instance name property.

Exceptions

- *`DDSMiddlewareException`*: if some error occurs while creating the *Fast DDS* publisher.

~Publisher() **override**

Destroy the *Publisher* object.

Publisher(**const** *Publisher*&) = delete

Publisher shall not be copy constructible.

Publisher &**operator=**(**const** *Publisher*&) = delete

Publisher shall not be copy assignable.

Publisher(*Publisher*&&) = delete

Publisher shall not be move constructible.

Publisher &**operator=**(*Publisher*&&) = delete

Publisher shall not be move assignable.

bool **publish**(**const** xtypes::DynamicData &message) **override**

Inherited from *TopicPublisher*.

const std::string &**topic_name**() **const**
Get the topic name where this publisher sends data to.

Return The topic name.

const fastrtps::rtps::InstanceHandle_t **get_dds_instance_handle**() **const**
Get the DDS instance handle object for the associated datawriter.

Return The datawriter instance handle.

6.14.5 Server

class eprosima::is::sh::fastdds::**Server** : **public virtual** eprosima::is::*ServiceProvider*, **private** DataWriterI
This class represents a DDS *Server*, built over the publisher/subscriber layer of *Fast DDS* using the DDS-RPC paradigm, within the *Integration Service* framework.

It is composed of a *Fast DDS Publisher*, to send the request to the DDS dataspace; plus a *Fast DDS Subscriber*, to receive replies from the DDS application server and send them back to the *Integration Service*.

Its topic type definition and data instances for request and reply types are represented by means of the Fast DDS Dynamic Types API, which allows to get rid of TypeSupport for each used type and eases users the task of defining and using their own custom types on the go, by means of a valid *IDL* definition.

This class inherits from Fast DDS Data Reader Listener and from Fast DDS Data Writer Listener for reacting to datawriter and datareader events, such as matching with subscribers and publishers or receiving new data from them.

The request petitions are associated with each received reply by means of the sample identity and the related sample identity attributes.

Public Functions

Server(eprosima::is::sh::fastdds::*Participant* **participant*, **const** std::string &*service_name*, **const** xtypes::DynamicType &*request_type*, **const** xtypes::DynamicType &*reply_type*, **const** YAML::Node &*config*)
Construct a new *Server* object.

Parameters

- `[in] participant`: The associated *Integration Service Participant*, which holds the DDS entities that compose this *Server*.
- `[in] service_name`: The service name. It will produce two topics: `<service_name>_Request` and `<service_name>_Reply`.
- `[in] request_type`: A dynamic type definition of the request topic's type.
- `[in] reply_type`: A dynamic type definition of the reply topic's type.
- `[in] config`: Additional configuration that might be required to configure this *Server*.

Exceptions

- `DDSMiddlewareExeption`: if some error occurs while creating the *Fast DDS* entities.

~Server() **override**
Destroy the *Server* object.

Server(**const** *Server* &*rhs*) = delete
Server shall not be copy constructible.

Server &**operator=**(**const** *Server* &*rhs*) = delete
Server shall not be copy assignable.

Server(*Server* &&*rhs*) = delete
Server shall not be move constructible.

Server &**operator=**(*Server* &&*rhs*) = delete
Server shall not be move assignable.

void **call_service**(**const** xtypes::DynamicData &*is_request*, *ServiceClient* &*client*, std::shared_ptr<void> *call_handle*) **override**
Inherited from *ServiceProvider*.

bool **add_config**(**const** YAML::Node &*configuration*)
Handle type remappings for DDS request and reply types. It allows to resolve complex type remappings, which remap to a specific type member, for example, an UnionType member, by means of the dot `.` operator.

Return `true` if success.

Parameters

- `[in] configuration`: The *YAML* configuration containing the remapping to be applied.

6.14.6 Subscriber

class `eprosima::is::sh::fastdds::`**Subscriber** : **private** DataReaderListener
This class represents a Fast DDS Subscriber within the *Integration Service* framework.

Its topic type definition and data instances are represented by means of the Fast DDS Dynamic Types API, which allows to get rid of TypeSupport for each used type and eases users the task of defining and using their own custom types on the go, by means of a valid *IDL* definition.

This class inherits from Fast DDS Data Reader Listener for reacting to datareader events, such as matching with publishers or receiving new data from them.

Public Functions

Subscriber(*Participant* *participant, **const** std::string &*topic_name*, **const** xtypes::DynamicType &*message_type*, *TopicSubscriberSystem*::*SubscriptionCallback* *is_callback*)
Construct a new *Subscriber* object.

Parameters

- `[in] participant`: The associated *Integration Service* *Participant*, which holds this *Subscriber*.
- `[in] topic_name`: The topic that this DDS subscriber will attach to.
- `[in] message_type`: A dynamic type definition of the topic's type.
- `[in] is_callback`: Callback function signature defined by the *Integration Service*, triggered each time a new data arrives to the DDS *Subscriber*.

Exceptions

- *`DDSMiddlewareException`*: if some error occurs while creating the *Fast DDS* subscriber.

~Subscriber()
Destroy the *Subscriber* object.

Subscriber(**const** *Subscriber*&) = delete
Subscriber shall not be copy constructible.

Subscriber &**operator=**(**const** *Subscriber*&) = delete
Subscriber shall not be copy assignable.

Subscriber(*Subscriber*&&) = delete
Subscriber shall not be move constructible.

Subscriber &**operator=**(*Subscriber*&&) = delete
Subscriber shall not be move assignable.

void **receive**(**const** fastrtps::types::DynamicData **dds_message*, fastdds::dds::SampleInfo *sample_info*)
Handle the receiving of a new message from the DDS dataspace.

Parameters

- `[in] dds_message`: The incoming message.
- `[in] sample_info`: Structure containing the relevant information regarding the incoming message.

6.15 ROS 1 System Handle

This section presents the API provided by the *Integration Service* `is-ros1` library.

6.15.1 Factory

class `eprosima::is::sh::ros1::`**Factory**
This is a singleton class that allows to gain access to the specific publisher, subscriber, client and server conversion functions, callbacks and other tasks, for each topic and service type.

Coming from the ROS 1 `msg` and `srv` files, the *Integration Service* `genmsg` plugin will generate the conversion library files for each of them, after calling the `is_ros1_genmsg_mix` macro in the CMakeLists.txt file of the `ros1_mix_generator` CMake project.

The generated conversion files will be compiled into a dynamic library that will be registered to a `mix` file, using the *is::core::MiddlewareInterfaceExtension* API provided by the *Integration Service Core*. This ROS 1 conversion `mix` libraries will use this *Factory* class to register the conversion functions from/to ROS 1 types to xTypes, as well as the subscription, publisher, service server and service client factories, that will be used later to create the necessary links in the *core* to bridge ROS 1 with another middleware supported by the *Integration Service*.

Public Types

using RegisterTypeToFactory = std::function<xtypes::DynamicType::Ptr ( ) >
Signature for the method that will be used to register a dynamic type within the types factory.

using RegisterSubscriptionToFactory = std::function<std::shared_ptr<void> (ros::NodeHandle &node, **const** std::string &topic_name, **const** xtypes::DynamicType &message_type, *TopicSubscriberSystem*::*SubscriptionCallback* *callback, uint32_t queue_size, **const** ros::TransportHints &transport_hints) >
Signature for the method that will be used to create a ROS 1 subscription to a certain topic, within the subscriptions factory.

It allows to specify the associated ROS 1 node, the topic name and type, as well as the callback function called every time a new message data arrives to this subscription.

This *Factory* method returns an opaque pointer containing the subscription object created by the *Integration Service* to manage a subscription. This subscription object is dependent on every ROS 1 type and it is autogenerated in the template available in resources/convert__msg.cpp.em and resources/convert__msg.hpp.em.

using RegisterPublisherToFactory = std::function<std::shared_ptr<*TopicPublisher*> (ros::NodeHandle &node, **const** std::string &topic_name, uint32_t queue_size, bool latch) >
Signature for the method that will be used to create a ROS 1 publisher to a certain topic, within the publishers factory.

It allows to specify the associated ROS 1 node, the topic name to publish to, the queue size and enabling/disabling message latching.

This *Factory* method returns a pointer to an *Integration Service* *TopicPublisher* object, holding the created ROS 1 publisher. This publisher object is dependent on every ROS 1 type and it is autogenerated in the template available in resources/convert__msg.cpp.em and resources/convert__msg.hpp.em.

using RegisterServiceClientToFactory = std::function<std::shared_ptr<*ServiceClient*> (ros::NodeHandle &node, **const** std::string &service_name, *ServiceClientSystem*::*RequestCallback* *callback)>

Signature for the method that will be used to create a ROS 1 service client to a certain service, within the service clients factory.

It allows to specify the associated ROS 1 node, the service name, as well as the callback function called every time a new request data arrives to this service client.

This *Factory* method returns a pointer containing the *Integration Service ServiceClient* object created by the *Integration Service* to manage a service client. This service client object is dependent on every ROS 1 type and it is autogenerated in the template available in resources/convert__srv.cpp.em.

using RegisterServiceProviderToFactory = std::function<std::shared_ptr<*ServiceProvider*> (ros::NodeHandle &node, **const** std::string &service_name)>

Signature for the method that will be used to create a ROS 1 service server to a certain service, within the service servers factory.

It allows to specify the associated ROS 1 node and the service name.

This *Factory* method returns a pointer containing the *Integration Service ServiceProvider* object created by the *Integration Service* to manage a service server. This service server object is dependent on every ROS 1 type and it is autogenerated in the template available in resources/convert__srv.cpp.em.

Public Functions

void **register_type_factory**(**const** std::string &*type_name*, *RegisterTypeToFactory* *register_type_func*)

Register a dynamic type within the types *Factory*.

Parameters

- `[in] type_name`: The type name, used as key in the *Factory* types map.
- `[in] register_type_func`: The function used to create the type.

xtypes::DynamicType::Ptr **create_type**(**const** std::string &*type_name*)

Create a dynamic type instance using the types registered previously in the *Factory*.

Return A pointer to the created type, or `nullptr` if the type was not registered in the *Factory*.

Parameters

- `[in] type_name`: The name of the type to be created.

void **register_subscription_factory**(**const** std::string &*topic_type*, *RegisterSubscriptionToFactory* *register_sub_func*)

Register a ROS 1 subscription builder within the *Factory*.

Parameters

- `[in] topic_type`: The name of the topic type, used to index the subscription factory map.
- `[in] register_sub_func`: The function used to create the subscription.

std::shared_ptr<void> **create_subscription**(**const** xtypes::DynamicType &*topic_type*, ros::NodeHandle &*node*, **const** std::string &*topic_name*, *TopicSubscriberSystem*::*SubscriptionCallback* **callback*, uint32_t *queue_size*, **const** ros::TransportHints &*transport_hints*)

Create a ROS 1 subscription handler for the *Integration Service*, using the subscriptions registered previously in the *Factory*.

Return An opaque pointer to the created *Integration Service* subscription entity.

Parameters

- `[in] topic_type`: A reference to the dynamic type representation of the topic type.
- `[in] node`: The ROS 1 node that will hold this subscription.
- `[in] topic_name`: The topic name to be subscribed to.
- `[in] callback`: The callback function called every time the ROS 1 subscription receives a new data.
- `[in] queue_size`: The maximum message queue size for the ROS 1 subscription.
- `[in] transport_hints`: Provides the subscriber with specific transport information.

void **register_publisher_factory**(**const** std::string &*topic_type*, *RegisterPublisherToFactory* *register_pub_func*)

Register a ROS 1 publisher builder within the *Factory*.

Parameters

- `[in] topic_type`: The name of the topic type, used to index the publisher factory map.
- `[in] register_pub_func`: The function used to create the publisher.

std::shared_ptr<*TopicPublisher*> **create_publisher**(**const** xtypes::DynamicType &*topic_type*, ros::NodeHandle &*node*, **const** std::string &*topic_name*, uint32_t *queue_size*, bool *latch*)

Create a ROS 1 publisher handler for the *Integration Service*, using the publisher registered previously in the *Factory*.

Return A pointer to the created *Integration Service* *TopicPublisher* entity.

Parameters

- `[in] topic_type`: A reference to the dynamic type representation of the topic type.
- `[in] node`: The ROS 1 node that will hold this publisher.
- `[in] topic_name`: The topic name to publish to.

- `[in] queue_size`: The maximum message queue size for the ROS 1 publisher.
- `[in] latch`: Enable/disable latching. When a connection is latched, the last message published is saved and sent to any future subscribers that connect.

void **register_client_proxy_factory**(**const** std::string &*service_response_type*, *RegisterServiceClientToFactory* *register_service_client_func*)

Register a ROS 1 service client builder within the *Factory*.

Parameters

- `[in] service_response_type`: The name of the service response type, used as index in the service client factory map.
- `[in] register_service_client_func`: The function used to create the service client.

std::shared_ptr<*ServiceClient*> **create_client_proxy**(**const** std::string &*service_response_type*, ros::NodeHandle &*node*, **const** std::string &*service_name*, *ServiceClientSystem*::*RequestCallback* *callback*)

Create a ROS 1 service client handler for the *Integration Service*, using the service client registered previously in the *Factory*.

Return A pointer to the created *Integration Service* *ServiceClient* entity.

Parameters

- `[in] service_response_type`: A reference to the dynamic type representation of the service response type.
- `[in] node`: The ROS 1 node that will hold this service client.
- `[in] service_name`: The service name to forward petitions to.
- `[in] callback`: The callback function called every time the ROS 1 service client receives a new request.

void **register_server_proxy_factory**(**const** std::string &*service_request_type*, *RegisterServiceProviderToFactory* *register_service_server_func*)

Register a ROS 1 service server builder within the *Factory*.

Parameters

- `[in] service_request_type`: The name of the service server type to be registered.
- `[in] register_service_server_func`: The function used to create the service server.

std::shared_ptr<*ServiceProvider*> **create_server_proxy**(**const** std::string &*service_request_type*, ros::NodeHandle &*node*, **const** std::string &*service_name*)

Create a ROS 1 service server handler for the *Integration Service*, using the service server registered previously in the *Factory*.

Return A pointer to the created *Integration Service* *ServiceProvider* entity.

Parameters

- `[in] service_request_type`: A reference to the dynamic type representation of the service request type.

- `[in] node`: The ROS 1 node that will hold this service server.
- `[in] service_name`: The name of the service.

Public Static Functions

Factory &**instance**()
Get a reference to the singleton instance of this *Factory*.

Return A mutable reference to the *Factory* singleton object instance.

class Implementation
Defines the actual implementation of the *Factory* class.

Allows to use the *pimpl* procedure to separate the implementation from the interface of *Factory*.

Methods named equal to some *Factory* method will not be documented again. Usually, the interface class will call `_pimpl->method()`, but the functionality and parameters are exactly the same.

6.15.2 MetaPublisher

std::shared_ptr<is::*TopicPublisher*> `eprosima::is::sh::ros1::`**make_meta_publisher**(**const** eprosima::xtypes::DynamicType &*message_type*, ros::NodeHandle &*node*, **const** std::string &*topic_name*, uint32_t *queue_size*, bool *latch*, **const** YAML::Node &*configuration*)
Produces a *is::TopicPublisher* that allows to use runtime substitution parameters in the *YAML* configuration file.

See *is::core::StringTemplate*

Return A pointer to the created *Integration Service* *TopicPublisher* entity.

Parameters

- `[in] message_type`: A reference to the dynamic type representation of the topic type.
- `[in] node`: The ROS 1 node that will hold this publisher.
- `[in] topic_name`: The topic name to publish to.
- `[in] queue_size`: The maximum message queue size for the ROS 1 publisher.
- `[in] latch`: Enable/disable latching. When a connection is latched, the last message published is saved and sent to any future subscribers that connect.

- `[in] configuration`: The configuration specific for this *SystemHandle*, as described in the user-provided *YAML* input file.

6.15.3 SystemHandle

class `eprosima::is::sh::ros1::`**SystemHandle** : **public virtual** eprosima::is::*FullSystem*

Implements all the interface defined for the *Integration Service FullSystem*, for the ROS 1 ecosystem.

Note This *SystemHandle* is currently prepared to support the latest LTS distribution of ROS 1, that is, Noetic Ninjemys.

Public Functions

SystemHandle()

Construct a new *SystemHandle* object.

bool **configure**(**const** core::*RequiredTypes* &*types*, **const** YAML::Node &*configuration*, *TypeRegistry* &*type_registry*) **override**

Inherited from *SystemHandle*.

bool **okay**() **const override**

Inherited from *SystemHandle*.

bool **spin_once**() **override**

Inherited from *SystemHandle*.

~SystemHandle() **override**

Inherited from *SystemHandle*.

bool **subscribe**(**const** std::string &*topic_name*, **const** xtypes::DynamicType &*message_type*, SubscriptionCallback **callback*, **const** YAML::Node &*configuration*) **override**

Inherited from *SystemHandle*.

bool **is_internal_message**(void **filter_handle*) **override**

Inherited from *TopicSubscriberSystem*.

std::shared_ptr<*TopicPublisher*> **advertise**(**const** std::string &*topic_name*, **const** xtypes::DynamicType &*message_type*, **const** YAML::Node &*configuration*) **override**

Inherited from *SystemHandle*.

bool **create_client_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*service_type*, RequestCallback **callback*, **const** YAML::Node &*configuration*) **override**

Inherited from *SystemHandle*.

bool **create_client_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType&, **const** xtypes::DynamicType &*reply_type*, RequestCallback **callback*, **const** YAML::Node &*configuration*) **override**

Inherited from *ServiceClientSystem*.

std::shared_ptr<*ServiceProvider*> **create_service_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*service_type*, **const** YAML::Node &*configuration*) **override**

Inherited from *SystemHandle*.

std::shared_ptr<*ServiceProvider*> **create_service_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*request_type*, **const** xtypes::DynamicType&, **const** YAML::Node &*configuration*) **override**

Inherited from *ServiceProviderSystem*.

6.16 ROS 2 System Handle

This section presents the API provided by the *Integration Service* `is-ros2` library.

6.16.1 Factory

class `eprosima::is::sh::ros2::`**Factory**

This is a singleton class that allows to gain access to the specific publisher, subscriber, client and server conversion functions, callbacks and other tasks, for each topic and service type.

Coming from the ROS 2 `msg` and `srv` files, the *Integration Service* `rosidl` plugin will generate the conversion library files for each of them, after calling the `is_ros2_rosidl_mix` macro in the CMakeLists.txt file of the `ros2_mix_generator` CMake project.

The generated conversion files will be compiled into a dynamic library that will be registered to a `mix` file, using the *is::core::MiddlewareInterfaceExtension* API provided by the *Integration Service Core*. This ROS 2 conversion `mix` libraries will use this *Factory* class to register the conversion functions from/to ROS 2 types to xTypes, as well as the subscription, publisher, service server and service client factories, that will be used later to create the necessary links in the *core* to bridge ROS 2 with another middleware supported by the *Integration Service*.

Public Types

using RegisterTypeToFactory = std::function<xtypes::DynamicType::Ptr()>

Signature for the method that will be used to register a dynamic type within the types factory.

using RegisterSubscriptionToFactory = std::function<std::shared_ptr<void>(rclcpp::Node &node, **const** std::string &topic_name, **const** xtypes::DynamicType &message_type, *TopicSubscriberSystem*::*SubscriptionCallback* *callback, **const** rmw_qos_profile_t &qos_profile)>

Signature for the method that will be used to create a ROS 2 subscription to a certain topic, within the subscriptions factory.

It allows to specify the associated ROS 2 node, the topic name and type, as well as the callback function called every time a new message data arrives to this subscription.

This *Factory* method returns an opaque pointer containing the subscription object created by the *Integration Service* to manage a subscription. This subscription object is dependent on every ROS 2 type and it is autogenerated in the template available in resources/convert__msg.cpp.em and resources/convert__msg.hpp.em.

using RegisterPublisherToFactory = std::function<std::shared_ptr<*TopicPublisher*> (rclcpp::Node &node, **const** std::string &topic_name, **const** rmw_qos_profile_t &qos_profile) >

Signature for the method that will be used to create a ROS 2 publisher to a certain topic, within the publishers factory.

It allows to specify the associated ROS 2 node, the topic name to publish to, and the QoS profile for the publisher.

This *Factory* method returns a pointer to an *Integration Service TopicPublisher* object, holding the created ROS 2 publisher. This publisher object is dependent on every ROS 2 type and it is autogenerated in the template available in resources/convert__msg.cpp.em and resources/convert__msg.hpp.em.

using RegisterServiceClientToFactory = std::function<std::shared_ptr<*ServiceClient*> (rclcpp::Node &node, **const** std::string &service_name, *ServiceClientSystem*::*RequestCallback* *callback, **const** rmw_qos_profile_t &qos_profile) >

Signature for the method that will be used to create a ROS 2 service client to a certain service, within the service clients factory.

It allows to specify the associated ROS 2 node, the service name, as well as the callback function called every time a new request data arrives to this service client.

This *Factory* method returns a pointer containing the *Integration Service ServiceClient* object created by the *Integration Service* to manage a service client. This service client object is dependent on every ROS 2 type and it is autogenerated in the template available in resources/convert__srv.cpp.em.

using RegisterServiceProviderToFactory = std::function<std::shared_ptr<*ServiceProvider*> (rclcpp::Node &node, **const** std::string &service_name, **const** rmw_qos_profile_t &qos_profile)>

Signature for the method that will be used to create a ROS 2 service server to a certain service, within the service servers factory.

It allows to specify the associated ROS 2 node and the service name.

This *Factory* method returns a pointer containing the *Integration Service ServiceProvider* object created by the *Integration Service* to manage a service server. This service server object is dependent on every ROS 2 type and it is autogenerated in the template available in resources/convert__srv.cpp.em.

Public Functions

void **register_type_factory**(**const** std::string &*type_name*, *RegisterTypeToFactory* *register_type_func*)

Register a dynamic type within the types *Factory*.

Parameters

- `[in] type_name`: The type name, used as key in the *Factory* types map.
- `[in] register_type_func`: The function used to create the type.

xtypes::DynamicType::Ptr **create_type**(**const** std::string &*type_name*)

Create a dynamic type instance using the types registered previously in the *Factory*.

Return A pointer to the created type, or `nullptr` if the type was not registered in the *Factory*.

Parameters

- `[in] type_name`: The name of the type to be created.

void **register_subscription_factory**(**const** std::string &*topic_type*, *RegisterSubscriptionToFactory* *register_sub_func*)

Register a ROS 2 subscription builder within the *Factory*.

Parameters

- `[in] topic_type`: The name of the topic type, used to index the subscription factory map.
- `[in] register_sub_func`: The function used to create the subscription.

std::shared_ptr<void> **create_subscription**(**const** xtypes::DynamicType &*topic_type*, rclcpp::Node &*node*, **const** std::string &*topic_name*, *TopicSubscriberSystem*::*SubscriptionCallback* **callback*, **const** rmw_qos_profile_t &*qos_profile*)

Create a ROS 2 subscription handler for the *Integration Service*, using the subscriptions registered previously in the *Factory*.

Return An opaque pointer to the created *Integration Service* subscription entity.

Parameters

- `[in] topic_type`: A reference to the dynamic type representation of the topic type.
- `[in] node`: The ROS 2 node that will hold this subscription.
- `[in] topic_name`: The topic name to be subscribed to.
- `[in] callback`: The callback function called every time the ROS 2 subscription receives a new data.
- `[in] qos_profile`: The QoS used to create the subscription.

void **register_publisher_factory**(**const** std::string &*topic_type*, *RegisterPublisherToFactory* *register_pub_func*)

Register a ROS 2 publisher builder within the *Factory*.

Parameters

- `[in] topic_type`: The name of the topic type, used to index the publisher factory map.
- `[in] register_pub_func`: The function used to create the publisher.

std::shared_ptr<*TopicPublisher*> **create_publisher**(**const** xtypes::DynamicType &*topic_type*, rclcpp::Node &*node*, **const** std::string &*topic_name*, **const** rmw_qos_profile_t &*qos_profile*)

Create a ROS 2 publisher handler for the *Integration Service*, using the publisher registered previously in the *Factory*.

Return A pointer to the created *Integration Service* *TopicPublisher* entity.

Parameters

- `[in] topic_type`: A reference to the dynamic type representation of the topic type.
- `[in] node`: The ROS 2 node that will hold this publisher.
- `[in] topic_name`: The topic name to publish to.
- `[in] qos_profile`: The QoS used to create the publisher.

void **register_client_proxy_factory**(**const** std::string &*service_response_type*, *RegisterServiceClientToFactory* *register_service_client_func*)

Register a ROS 2 service client builder within the *Factory*.

Parameters

- `[in] service_response_type`: The name of the service response type, used as index in the service client factory map.
- `[in] register_service_client_func`: The function used to create the service client.

std::shared_ptr<*ServiceClient*> **create_client_proxy**(**const** std::string &*service_response_type*, rclcpp::Node &*node*, **const** std::string &*service_name*, *ServiceClientSystem*::*RequestCallback* **callback*, **const** rmw_qos_profile_t &*qos_profile*)

Create a ROS 2 service client handler for the *Integration Service*, using the service client registered previously in the *Factory*.

Return A pointer to the created *Integration Service* *ServiceClient* entity.

Parameters

- `[in] service_response_type`: A reference to the dynamic type representation of the service response type.
- `[in] node`: The ROS 2 node that will hold this service client.
- `[in] service_name`: The service name to forward petitions to.
- `[in] callback`: The callback function called every time the ROS 2 service client receives a new request.
- `[in] qos_profile`: The QoS used to create the service client.

void **register_server_proxy_factory**(**const** std::string &*service_request_type*, *RegisterServiceProviderToFactory* *register_service_server_func*)

Register a ROS 2 service server builder within the *Factory*.

Parameters

- `[in] service_request_type`: The name of the service server type to be registered.
- `[in] register_service_server_func`: The function used to create the service server.

std::shared_ptr<*ServiceProvider*> **create_server_proxy**(**const** std::string &*service_request_type*, rclcpp::Node &*node*, **const** std::string &*service_name*, **const** rmw_qos_profile_t &*qos_profile*)

Create a ROS 2 service server handler for the *Integration Service*, using the service server registered previously in the *Factory*.

Return A pointer to the created *Integration Service* *ServiceProvider* entity.

Parameters

- `[in] service_request_type`: A reference to the dynamic type representation of the service request type.
- `[in] node`: The ROS 2 node that will hold this service server.
- `[in] service_name`: The service name to process petitions from.
- `[in] qos_profile`: The QoS used to create the service server.

Public Static Functions

Factory &**instance**()

Get a reference to the singleton instance of this *Factory*.

Return A mutable reference to the *Factory* singleton object instance.

class Implementation

Defines the actual implementation of the *Factory* class.

Allows to use the *pimpl* procedure to separate the implementation from the interface of *Factory*.

Methods named equal to some *Factory* method will not be documented again. Usually, the interface class will call `_pimpl->method()`, but the functionality and parameters are exactly the same.

6.16.2 MetaPublisher

std::shared_ptr<is::*TopicPublisher*> `eprosima::is::sh::ros2::`**`make_meta_publisher`**(**const** eprosima::xtypes::DynamicType *&message_type*, rclcpp::Node *&node*, **const** std::string *&topic_name*, **const** rmw_qos_profile_t *&qos_profile*, **const** YAML::Node *&configuration*)

Produces a *is::TopicPublisher* that allows to use runtime substitution parameters in the *YAML* configuration file.

See *is::core::StringTemplate*

Return A pointer to the created *Integration Service TopicPublisher* entity.

Parameters

- `[in] message_type`: A reference to the dynamic type representation of the topic type.
- `[in] node`: The ROS 1 node that will hold this publisher.
- `[in] topic_name`: The topic name to publish to.
- `[in] qos_profile`: The QoS used to create the publisher.
- `[in] configuration`: The configuration specific for this *SystemHandle*, as described in the user-provided *YAML* input file.

6.16.3 System Handle

`class` `eprosima::is::sh::ros2::`**`SystemHandle`** : **`public virtual`** eprosima::is::*FullSystem*

Implements all the interface defined for the *Integration Service FullSystem*, for the ROS 2 ecosystem.

Some changes might be needed to support ROS 2 Galactic, the forthcoming version of ROS 2. This will be mainly related to the use of the new API for setting the DOMAIN ID within every ROS 2 node, instead of using the `ROS_DOMAIN_ID` environment variable.

Note This *SystemHandle* is currently prepared to support the latests distributions of ROS 2, that is, Foxy Fitzroy and Galactic Geochelone.

Public Functions

SystemHandle()
Construct a new *SystemHandle* object.

bool **configure**(**const** core::*RequiredTypes* &*types*, **const** YAML::Node &*configuration*, *TypeRegistry* &*type_registry*) **override**
Inherited from *SystemHandle*.

bool **okay**() **const** **override**
Inherited from *SystemHandle*.

bool **spin_once**() **override**
Inherited from *SystemHandle*.

~SystemHandle() **override**
Inherited from *SystemHandle*.

bool **subscribe**(**const** std::string &*topic_name*, **const** xtypes::DynamicType &*message_type*, SubscriptionCallback **callback*, **const** YAML::Node &*configuration*) **override**
Inherited from *TopicSubscriberSystem*.

bool **is_internal_message**(void **filter_handle*) **override**
Inherited from *TopicSubscriberSystem*.

std::shared_ptr<*TopicPublisher*> **advertise**(**const** std::string &*topic_name*, **const** xtypes::DynamicType &*message_type*, **const** YAML::Node &*configuration*) **override**
Inherited from *TopicPublisherSystem*.

bool **create_client_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*service_type*, RequestCallback **callback*, **const** YAML::Node &*configuration*) **override**
Inherited from *ServiceClientSystem*.

bool **create_client_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType&, **const** xtypes::DynamicType &*reply_type*, RequestCallback **callback*, **const** YAML::Node &*configuration*) **override**
Inherited from *ServiceClientSystem*.

std::shared_ptr<*ServiceProvider*> **create_service_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*service_type*, **const** YAML::Node &*configuration*) **override**
Inherited from *ServiceProviderSystem*.

std::shared_ptr<*ServiceProvider*> **create_service_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*request_type*, **const** xtypes::DynamicType&, **const** YAML::Node &*configuration*) **override**
Inherited from *ServiceProviderSystem*.

6.17 WebSocket System Handle

This section presents the API provided by the *Integration Service* `is-websocket` library.

6.17.1 Encoding

class `eprosima::is::sh::websocket::`**Encoding**

This interface class defines all the methods that must be implemented in order to create an encoding to be used to construct and interpret raw *WebSocket* messages.

eprosima::is::sh::websocket::JsonEncoding: *Encoding* implementation for message exchanging using JSON format.

Subclassed by eprosima::is::sh::websocket::JsonEncoding

Public Functions

void **interpret_websocket_msg**(**const** std::string &*msg*, *Endpoint* &*endpoint*, std::shared_ptr<void> *connection_handle*) **const** = 0

Interpret an incoming *WebSocket* message.

Parameters

- `[in] msg`: The message to be interpreted.
- `[in] endpoint`: The target endpoint which will perform the actions specified by the message.
- `[in] connection_handle`: Opaque pointer which identifies the current connection.

std::string **encode_publication_msg**(**const** std::string &*topic_name*, **const** std::string &*topic_type*, **const** std::string &*id*, **const** xtypes::DynamicData &*msg*) **const** = 0

Encode a publish message.

Return A string representation of the encoded publication message, ready to be sent using *WebSocket*.

Parameters

- `[in] topic_name`: The name of the topic where the message will be published to.
- `[in] topic_type`: The type name of the topic where the message will be published to.
- `[in] id`: The publisher ID.
- `[in] msg`: The message data to be published. This will be transformed to *JSON* format beforehand.

std::string **encode_service_response_msg**(**const** std::string &*service_name*, **const** std::string &*service_type*, **const** std::string &*id*, **const** xtypes::DynamicData &*response*, bool *result*) **const** = 0

Encode a service response message.

Return A string representation of the encoded service response message, ready to be sent using *WebSocket*.

Parameters

- `[in] service_name`: The name of the service which is answering.
- `[in] service_type`: The type name of the service reply.
- `[in] id`: The service ID.
- `[in] response`: The message data containing the service response. This will be transformed to *JSON* format beforehand.
- `[in] result`: Indicates if the response was received or not from the service server.

std::string **encode_subscribe_msg**(**const** std::string &*topic_name*, **const** std::string &*message_type*, **const** std::string &*id*, **const** YAML::Node &*configuration*) **const** = 0

Encode a subscription message.

Return A string representation of the encoded subscription message, ready to be sent using *WebSocket*.

Parameters

- `[in] topic_name`: The name of the topic to which the subscription will be performed.
- `[in] message_type`: The type name of the topic to which the subscription will be performed.
- `[in] id`: The subscriber ID.
- `[in] configuration`: Additional configuration that might be required for the subscription operation.

std::string **encode_advertise_msg**(**const** std::string &*topic_name*, **const** std::string &*message_type*, **const** std::string &*id*, **const** YAML::Node &*configuration*) **const** = 0

Encode an advertisement message. This step is required prior to publish operation.

Return A string representation of the encoded advertise message, ready to be sent using *WebSocket*.

Parameters

- `[in] topic_name`: The name of the topic to which the advertisement will be performed.
- `[in] message_type`: The type name of the topic to which the advertisement will be performed.
- `[in] id`: The publisher ID.
- `[in] configuration`: Additional configuration that might be required for the advertise operation.

std::string **encode_call_service_msg**(**const** std::string &*service_name*, **const** std::string &*service_type*, **const** xtypes::DynamicData &*service_request*, **const** std::string &*id*, **const** YAML::Node &*configuration*) **const** = 0

Encode a call service message.

Return A string representation of the encoded call service message, ready to be sent using *WebSocket*.

Parameters

- `[in] service_name`: The name of service to be called.
- `[in] service_type`: The type name of the service to be called.

- `[in] service_request`: The data of the request message. This will be transformed to *JSON* format beforehand.
- `[in] id`: The service ID.
- `[in] configuration`: Additional configuration that might be required for the call service operation.

std::string **encode_advertise_service_msg**(**const** std::string &*service_name*, **const** std::string &*request_type*, **const** std::string &*reply_type*, **const** std::string &*id*, **const** YAML::Node &*configuration*) **const** = 0

Encode an advertise service message. This step is required prior to service call operations.

Return A string representation of the encoded service advertise message, ready to be sent using *WebSocket*.

Parameters

- `[in] service_name`: The name of the service to which the advertisement will be performed.
- `[in] request_type`: The request type name of the service to which the advertisement will be performed.
- `[in] reply_type`: The reply type name of the service to which the advertisement will be performed.
- `[in] id`: The service ID.
- `[in] configuration`: Additional configuration that might be required for the advertise operation.

bool **add_type**(**const** xtypes::DynamicType &*type*, **const** std::string &*type_name* = "")

Add a type to the types database.

Return `true` if the type was correctly added, or `false` otherwise.

Parameters

- `[in] type`: The dynamic type to be added.
- `[in] type_name`: The type name.

6.17.2 Endpoint

class `eprosima::is::sh::websocket::`**Endpoint** : **public** eprosima::is::*FullSystem*, **public** eprosima::is::*ServiceCl*

Represents a *WebSocket* endpoint for the *Integration Service*. The *Endpoint* class will be later specialized for client and server applications.

Subclassed by *eprosima::is::sh::websocket::Client*, *eprosima::is::sh::websocket::Server*

Public Functions

Endpoint(**const** std::string &*name*)
Constructor.

Parameters

- `name`: The name given to this *Endpoint* instance. It will be used to identify logging traces.

bool **configure**(**const** core::*RequiredTypes* &*types*, **const** YAML::Node &*configuration*, *TypeRegistry* &*type_registry*) **override**
Inherited from *SystemHandle*.

bool **okay**() **const** = 0
Inherited from *SystemHandle*.

bool **spin_once**() = 0
Inherited from *SystemHandle*.

~Endpoint() = default
Destructor.

bool **subscribe**(**const** std::string &*topic_name*, **const** xtypes::DynamicType &*message_type*, *TopicSubscriberSystem*::*SubscriptionCallback* *callback*, **const** YAML::Node &*configuration*) **final override**
Inherited from *TopicSubscriberSystem*.

bool **is_internal_message**(void **filter_handle*) **final override**
Inherited from *TopicSubscriberSystem*.

std::shared_ptr<*TopicPublisher*> **advertise**(**const** std::string &*topic_name*, **const** xtypes::DynamicType &*message_type*, **const** YAML::Node &*configuration*) **final override**
Inherited from *TopicPublisherSystem*.

bool **create_client_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*service_type*, *ServiceClientSystem*::*RequestCallback* *callback*, **const** YAML::Node &*configuration*) **final override**
Inherited from *ServiceClientSystem*.

bool **create_client_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*request_type*, **const** xtypes::DynamicType &*reply_type*, *ServiceClientSystem*::*RequestCallback* *callback*, **const** YAML::Node &*configuration*) **final override**
Inherited from *ServiceClientSystem*.

std::shared_ptr<*ServiceProvider*> **create_service_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*service_type*, **const** YAML::Node &*configuration*) **final override**
Inherited from *ServiceProviderSystem*.

std::shared_ptr<*ServiceProvider*> **create_service_proxy**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*request_type*, **const** xtypes::DynamicType &*reply_type*, **const** YAML::Node &*configuration*) **final override**
Inherited from *ServiceProviderSystem*.

void **startup_advertisement**(**const** std::string &*topic*, **const** xtypes::DynamicType &*message_type*, **const** std::string &*id*, **const** YAML::Node &*configuration*)

Send out an advertisement the next time a connection is made.

Parameters

- `[in] topic`: The topic name.
- `[in] message_type`: The Dynamic Type message representation.
- `[in] id`: The publisher ID.
- `[in] configuration`: Additional configuration, in *YAML* format, required to advertise the topic.

void **runtime_advertisement**(**const** std::string &*topic*, **const** xtypes::DynamicType &*message_type*, **const** std::string &*id*, **const** YAML::Node &*configuration*) = 0

Send out an advertisement to all existing connections right away. This is for publication topics that are determined at runtime by topic templates.

Parameters

- `[in] topic`: The topic name.
- `[in] message_type`: The Dynamic Type message representation.
- `[in] id`: The message ID.
- `[in] configuration`: Additional configuration, in *YAML* format, required to advertise the topic.

bool **publish**(**const** std::string &*topic*, **const** xtypes::DynamicData &*message*)

Publish a message to a certain topic.

See *is::TopicPublisher*.

Return `true` if the publication was made, `false` otherwise.

Parameters

- `[in] topic`: The topic name to publish to.
- `[in] message`: The message data instance to be published.

void **call_service**(**const** std::string &*service*, **const** xtypes::DynamicData &*request*, *ServiceClient* &*client*, std::shared_ptr<void> *call_handle*)

Call a service.

See *is::ServiceProvider*.

Parameters

- `[in] service`: The name of the service to be called.
- `[in] request`: Request message for the service.
- `[inout] client`: The proxy for the client that is making the request.
- `[in] call_handle`: A handle for the call.

void **receive_response**(std::shared_ptr<void> *call_handle*, **const** xtypes::DynamicData &*response*) **final override**

Inherited from *ServiceClient*.

void **receive_topic_advertisement_ws**(**const** std::string &*topic_name*, **const** xtypes::DynamicType &*message_type*, **const** std::string &*id*, std::shared_ptr<void> *connection_handle*)

Process an advertisement message. This step is required prior to publish operation.

Parameters

- `[in] topic_name`: The name of the topic to be advertised.
- `[in] message_type`: The type name of the topic to be advertised.
- `[in] id`: The publisher ID.
- `[in] connection_handle`: Opaque pointer which identifies the current connection.

void **receive_topic_unadvertisement_ws**(**const** std::string &*topic_name*, **const** std::string &*id*, std::shared_ptr<void> *connection_handle*)

Process an unadvertisement message.

Parameters

- `[in] topic_name`: The name of the topic to be unadvertised.
- `[in] id`: The publisher ID.
- `[in] connection_handle`: Opaque pointer which identifies the current connection.

void **receive_publication_ws**(**const** std::string &*topic_name*, **const** xtypes::DynamicData &*message*, std::shared_ptr<void> *connection_handle*)

Process an publication.

Parameters

- `[in] topic_name`: The name of the topic where the message will be published to.
- `[in] message`: The message published.
- `[in] connection_handle`: Opaque pointer which identifies the current connection.

void **receive_subscribe_request_ws**(**const** std::string &*topic_name*, **const** xtypes::DynamicType **message_type*, **const** std::string &*id*, std::shared_ptr<void> *connection_handle*)

Process a request for subscribing to a certain topic.

Parameters

- `[in] topic_name`: The name of the topic where the subscription will be performed to.
- `[in] message_type`: The dynamic type of the topic to get subscribed to.
- `[in] id`: The identifier of the message.
- `[in] connection_handle`: Opaque pointer which identifies the current connection.

void **receive_unsubscribe_request_ws**(**const** std::string &*topic_name*, **const** std::string &*id*, std::shared_ptr<void> *connection_handle*)

Process a request for unsubscribing to a certain topic.

Parameters

- `[in] topic_name`: The name of the topic where the subscription will be stopped.
- `[in] id`: The identifier of the message.
- `[in] connection_handle`: Opaque pointer which identifies the current connection.

void **receive_service_request_ws**(**const** std::string &*service_name*, **const** xtypes::DynamicData &*request*, **const** std::string &*id*, std::shared_ptr<void> *connection_handle*)

Process a service request.

Parameters

- `[in] service_name`: The name of the service.
- `[in] request`: The request data.
- `[in] id`: The service ID.
- `[in] connection_handle`: Opaque pointer which identifies the current connection.

void **receive_service_advertisement_ws**(**const** std::string &*service_name*, **const** xtypes::DynamicType &*req_type*, **const** xtypes::DynamicType &*reply_type*, std::shared_ptr<void> *connection_handle*)

Process a service advertisement. This is required prior to calling a service.

Parameters

- `[in] service_name`: The name of the service.
- `[in] req_type`: The request data type.
- `[in] reply_type`: The reply data type.
- `[in] connection_handle`: Opaque pointer which identifies the current connection.

void **receive_service_unadvertisement_ws**(**const** std::string &*service_name*, **const** xtypes::DynamicType **service_type*, std::shared_ptr<void> *connection_handle*)

Process a service unadvertisement. The service will no longer be available.

Parameters

- `[in] service_name`: The name of the service.
- `[in] service_type`: The service type. Usually refers to the request type.
- `[in] connection_handle`: Opaque pointer which identifies the current connection.

void **receive_service_response_ws**(**const** std::string &*service_name*, **const** xtypes::DynamicData &*response*, **const** std::string &*id*, std::shared_ptr<void> *connection_handle*)

Process a service response.

Parameters

- `[in] service_name`: The name of the service.
- `[in] response`: The response data.

- `[in] id`: The service ID.
- `[in] connection_handle`: Opaque pointer which identifies the current connection.

6.17.3 Client

class Client : **public** eprosima::is::sh::websocket::*Endpoint*

This class represents a *WebSocket* Client, which can be defined as an application that sends requests to a specific port and waits for the Server response.

It implements some of the *Endpoint* class methods.

6.17.4 Server

class Server : **public** eprosima::is::sh::websocket::*Endpoint*

This class represents a *WebSocket* Server, which can be defined as an application that listens to a specific port waiting for Client's requests.

It implements some of the *Endpoint* class methods.

6.17.5 JWTValidator

class `eprosima::is::sh::websocket::`**JwtValidator**

Class that validates the received JSON Web Token according to the *VerificationPolicy* specified on the configuration file.

Public Functions

void **verify**(**const** std::string &*token*)

Verifies the token.

Parameters

- `[in] token`: String containing the *JSON Web Token*.

Exceptions

- `VerificationError`:

void **add_verification_policy**(**const** *VerificationPolicy* &*policy*)

Adds a policy to resolve the verification strategy to use.

The *VerificationPolicy* should set the VerificationStrategy and returns true if it is able to provide a strategy. If there are multiple policies that can process a token, the 1st policy that matches is used. VerificationPolicyFactory contains some simple predefined policies.

Remark The idea is that *JwtValidator* should support verifying in multiple use cases. For example, choosing a secret based on the issuer or other claims and any custom strategy as required. There is no way to open up such flexibility from within the class so the conclusion is to have a handler that the consumer supplies to choose the verification method.

Parameters

- `[in] policy`: The policy to be added.

class eprosima::is::sh::websocket::**VerificationPolicy**
Class containing all the relevant information about the verification policy, which includes the public key or the secret key used for generating the token.

Public Types

using Rule = std::pair<std::string, std::string>
Rule signature.

Public Functions

VerificationPolicy(std::vector<*Rule*> *rules*, std::vector<*Rule*> *header_rules*, std::string *secret_or_pubkey*)
Constructor.

const std::string &**secret_or_pubkey**() **const**
Retrieves the public key or secret.

class ServerConfig
Loads from the *YAML* configuration file the authentication policy that will be used by the *JwtValidator*.

6.18 Different Protocols

This page gathers all the existing examples for *Integration Service* that connect **different protocols** which handle incompatible types.

6.18.1 Publisher - Subscriber

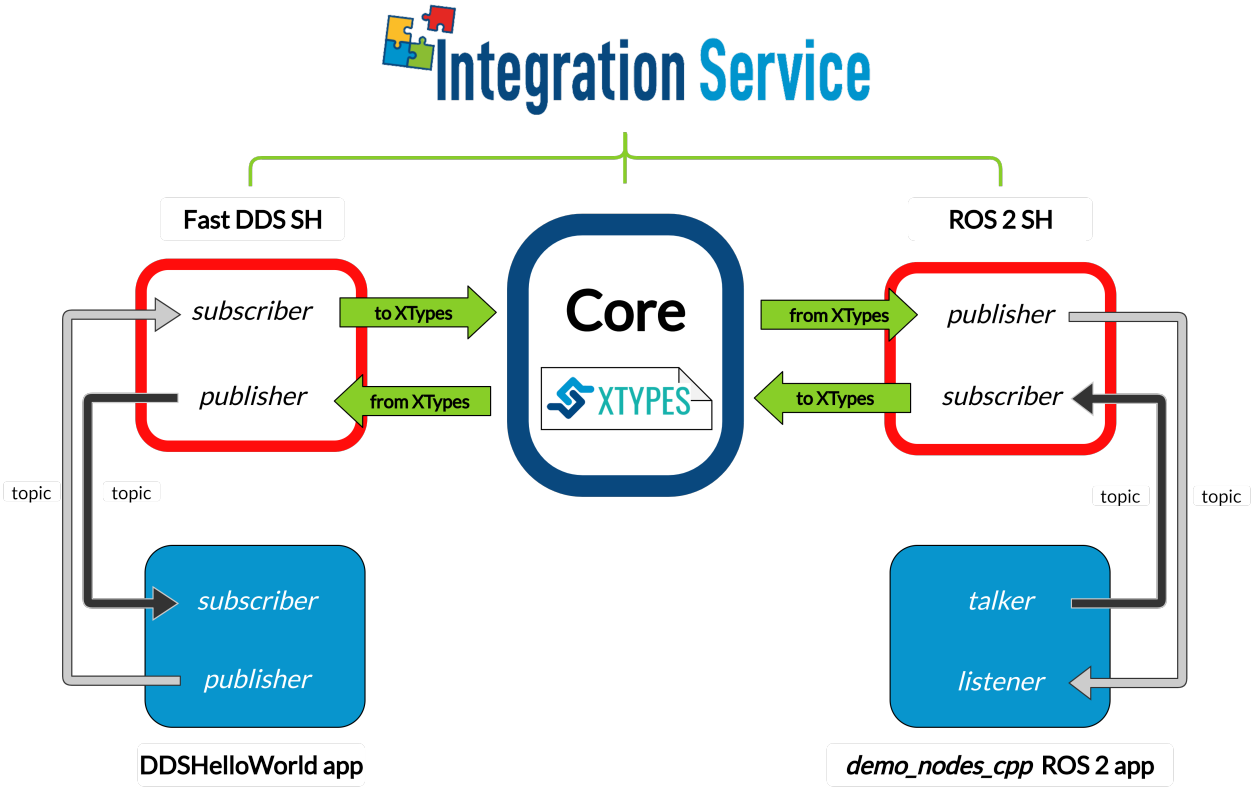
This page gathers all the *Integration Service* existing examples based on the **publisher - subscriber** paradigm that connect different protocols which handle incompatible types.

DDS - ROS 2 bridge

In this example we address a very common situation faced in the robotics world: that of bridging *DDS* and *ROS 2*. Specifically, we discuss how to do so with the *Fast DDS* implementation.

A user with knowledge of both systems may be aware that *ROS 2* uses *DDS* as a middleware but hides some of *DDS*' configuration details, thus making a direct communication between the two problematic. By using *Integration Service*, this task can be eased, and achieved with minimal effort from the user's side.

The steps described below address such a situation, by putting into communication a *ROS 2* `talker-listener` example with a *Fast DDS* `DDSHelloWorld` example.



Requirements

To prepare the deployment and setup the environment, you need to have *Integration Service* correctly installed in your system. To do so, please follow the steps delineated in the *Installation* section.

Also, to get this example working, the following requirements must be met:

- Having **ROS 2** (*Foxy* or superior) installed, with the `talker-listener` example working.
- Having the **ROS 2 System Handle** installed. You can download it from the ROS2-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS2-SH.git src/ROS2-SH src/ros2-sh
```

- Having **Fast DDS** (*v.2.0.0* or superior) installed and the *Integration Service* `DDSHelloWorld` example working. This example can be found in the main *Integration Service* repository, under the examples/utils/dds/DDSHelloWorld folder; to compile it, you can either compile the whole *Integration Service* project using `colcon` with the *CMake* flag `BUILD_EXAMPLES` enabled; or execute the following steps:

```
cd ~/is-workspace/src/Integration-Service/examples/utils/dds/DDSHelloWorld
mkdir build && cd build
cmake .. -DBUILD_EXAMPLES=ON && make
```

- Having the **Fast DDS System Handle** installed. You can download it from the FastDDS-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/FastDDS-SH.git src/FastDDS-SH
```

After you have everything correctly installed in your `is-workspace`, build the packages by running:

```
colcon build --cmake-args -DBUILD_FASTDDS_EXAMPLES=ON
```

Deployment

Below we explain how to deploy an example of this communication in both directions allowed.

ROS 2 talker to DDS subscriber

To enable communication from *ROS 2* to *Fast DDS*, open three terminals:

- In the first terminal, source your *ROS 2* installation and execute a *ROS 2* `talker`:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 run demo_nodes_cpp talker
```

- In the second terminal, execute a *Fast DDS* HelloWorld `subscriber` from within the `is-workspace`:

```
cd ~/is-workspace
source install/setup.bash
./build/is-examples/dds/DDSHelloWorld/DDSHelloWorld -m subscriber
```

At this point, the two applications cannot communicate due to the incompatibility of their *topics* and *types*. This is where *Integration Service* comes into play to make the communication possible.

- In the third terminal, go to the `is-workspace` folder, source the *ROS 2* and local installations, and execute *Integration Service* with the `integration-service` command followed by the fastdds_ros2__helloworld.yaml configuration file located in the `src/Integration-Service/examples/basic` folder:

```
cd ~/is-workspace
source /opt/ros/$<ROS2_DISTRO>/setup.bash
source install/setup.bash
integration-service src/Integration-Service/examples/basic/fastdds_ros2__
↪helloworld.yaml
```

Once the last command is executed, the two applications will start communicating.

DDS publisher to ROS 2 listener

To enable communication from *Fast DDS* to *ROS 2*, open three terminals:

- In the first terminal, execute a *Fast DDS* HelloWorld `publisher` from within the `is-workspace`:

```
cd ~/is-workspace
source install/setup.bash
./build/is-examples/dds/DDSHelloWorld/DDSHelloWorld -m publisher
```

- In the second terminal, source your *ROS 2* installation and execute a *ROS 2* `listener`:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 run demo_nodes_cpp listener
```

At this point, the two applications cannot communicate due to the incompatibility of their *topics* and *types*. This is where *Integration Service* comes into play to make the communication possible.

- In the third terminal, go to the `is-workspace` folder, source the *ROS 2* and local installations, and execute *Integration Service* with the `integration-service` command followed by the fastdds_ros2__helloworld.yaml configuration file located in the `src/Integration-Service/examples/basic` folder:

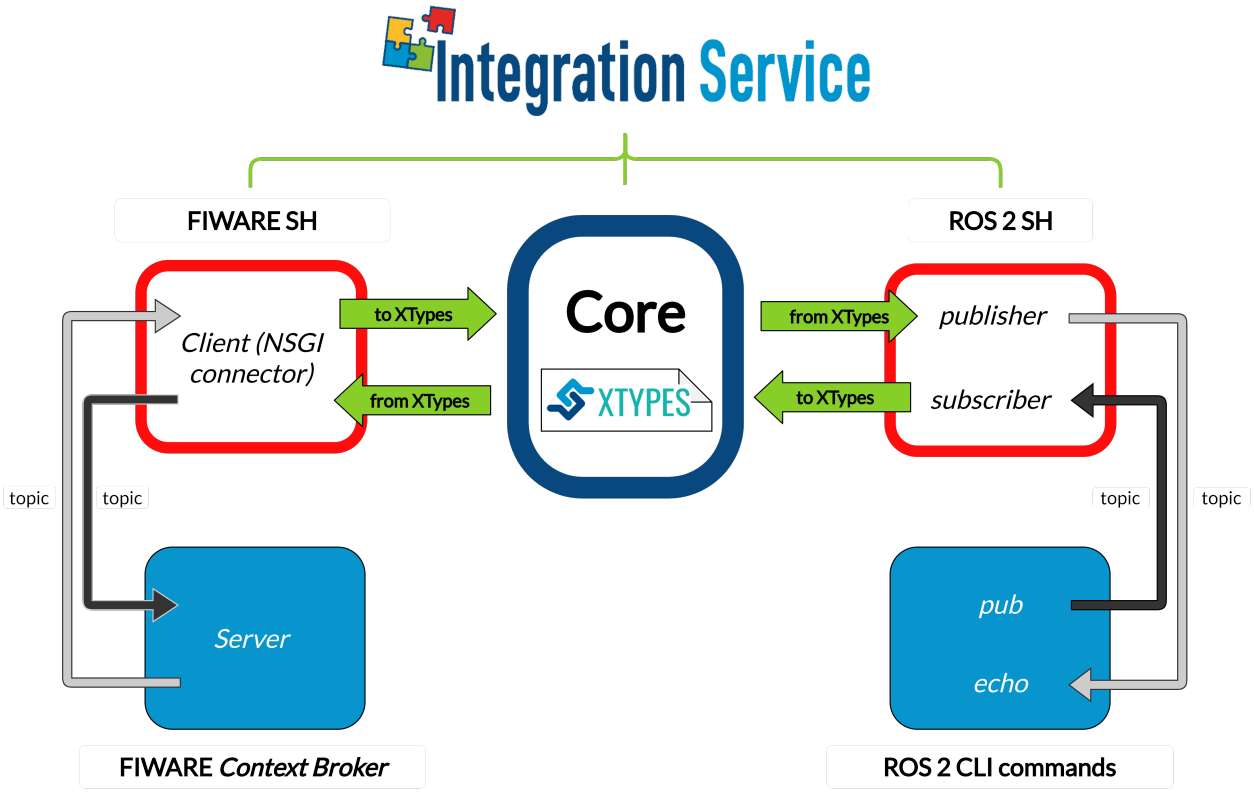
```
cd ~/is-workspace
source /opt/ros/$<ROS2_DISTRO>/setup.bash
source install/setup.bash
integration-service src/Integration-Service/examples/basic/fastdds_ros2__
↪helloworld.yaml
```

Once the last command is executed, the two applications will start communicating.

FIWARE - ROS 2 bridge

An interesting use case is the one of bringing information coming from the *ROS 2* world into the *FIWARE* ecosystem, so that it can be used to translate information coming from physically operated *ROS 2* robots into its *FIWARE's* digital twin models.

The steps described below aim to provide an easy way to translate the information coming from a *ROS 2* publisher into the *FIWARE's Orion Context Broker*; and viceversa.



Requirements

To prepare the deployment and setup the environment, you need to have *eProsima Integration Service* correctly installed in your system. To do so, please follow the steps delineated in the *Installation* section.

Also, to get this example working, the following requirements must be met:

- Having **ROS 2** (*Foxy* or superior) installed, with the `talker-listener` example working.
- Having the **ROS 2 System Handle** installed. You can download it from the ROS2-SH dedicated repository into the `is-workspace` where you have *eProsima Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS2-SH.git src/ROS2-SH
```

- Having a **FIWARE's Context Broker** correctly set up. To do so: * Set up a *MongoDB* database image:

```
docker run --rm --name mongodb -d mongo:3.4
```

  - Create a container for the *FIWARE's Orion Context Broker*, linked to the previously created *MongoDB* docker:

```
docker run --rm -d --name orion1 --link mongodb:mongodb -p 1026:1026 fiware/
↪orion -dbhost mongodb
```

  It is very important to retrieve the `fiware/orion` docker container IP, because it will be later placed in the *Integration Service YAML* configuration file. To do so, simply check the output of the following command:

```
ifconfig docker0 | grep "inet "
```

- Having the **FIWARE System Handle** installed. You can download it from the FIWARE-SH dedicated repository into the `is-workspace` where you have *eProsima Integration Service* installed:

```
cd ~/dds-is-workspace
git clone https://github.com/eProsima/FIWARE-SH.git src/FIWARE-SH
```

After you have everything correctly installed, build the packages by running:

```
colcon build
```

Deployment

Below we explain how to deploy an example of this communication in both directions allowed.

ROS 2 pub to FIWARE

To enable communication from *ROS 2* to *FIWARE*, open three terminals:

- In the first terminal, go to the `is-workspace` folder, source the *ROS 2* and local installations, and execute *eProsima Integration Service* with the `integration-service` command followed by the ros2_fiware__helloworld.yaml configuration file located in the `src/Integration-Service/examples/basic` folder.

Important: **Please notice** that the *YAML* may have a different IP address for the `host` file that the one you retrieved from your docker container bridge, if so, replace it properly. From now on, in this example, the host will be located at IP address `172.17.0.1`:

```
cd ~/is-workspace
source /opt/ros/$<ROS2_DISTRO>/setup.bash
source install/setup.bash
integration-service src/Integration-Service/examples/basic/ros2_fiware__
↪helloworld.yaml
```

- In the second terminal, create the corresponding entities in the *FIWARE's Context Broker*:

```
curl 172.17.0.1:1026/v2/entities -s -S -H 'Content-Type: application/json' -d @* <
↪<EOF
{
"id": "hello_fiware",
"type": "HelloWorld",
"data": {
    "value": "",
    "type": "String"
}
}
EOF
```

Now, in your browser, go to http://172.17.0.1:1026/v2/entities. You should see the context broker entity named **hello_fiware** previously created.

- In the third terminal, source the *ROS 2* installation and launch the *ROS 2* `pub`:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 topic pub /hello_fiware std_msgs/msg/String "{data: Hello FIWARE}"
```

Now, if you press *F5* in the browser, you should see that the entity has been correctly updated.

FIWARE to ROS 2 echo

To enable communication from *FIWARE* to *ROS 2*, open three terminals:

- In the first terminal, go to the `is-workspace` folder, source the *ROS 2* and local installations, and execute *eProsima Integration Service* with the `integration-service` command followed by the ros2_fiware__helloworld.yaml configuration file located in the `src/Integration-Service/examples/basic` folder.

Important: **Please notice** that the *YAML* may have a different IP address for the `host` file that the one you retrieved from your docker container bridge, if so, replace it properly. From now on, in this example, the host will be located at IP address `172.17.0.1`:

```
cd ~/is-workspace
source /opt/ros/$<ROS2_DISTRO>/setup.bash
source install/setup.bash
integration-service src/Integration-Service/examples/basic/ros2_fiware__
↪helloworld.yaml
```

- In the second terminal, create the corresponding entities in the *FIWARE's Context Broker*:

```
curl 172.17.0.1:1026/v2/entities -s -S -H 'Content-Type: application/json' -d @- <
↪<EOF
{
"id": "hello_ros2",
"type": "HelloWorld",
"data": {
    "value": "",
    "type": "String"
}
}
EOF
```

Now, in your browser, go to http://172.17.0.1:1026/v2/entities. You should see the context broker entity named **hello_fiware** previously created.

- In the third terminal, source the *ROS 2* installation and launch the *ROS 2* `echo`:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 topic echo /hello_ros2
```

- Again in the second terminal, update the *FIWARE* entity hosted in the *Context Broker*:

```
curl 172.17.0.1:1026/v2/entities/hello_ros2/attrs?type=HelloWorld -s -S -H
↪'Content-Type: application/json' -X PUT -d @- <<EOF
{
"data": {
    "value": "Hello, ROS 2",
    "type": "String"
}
}
EOF
```

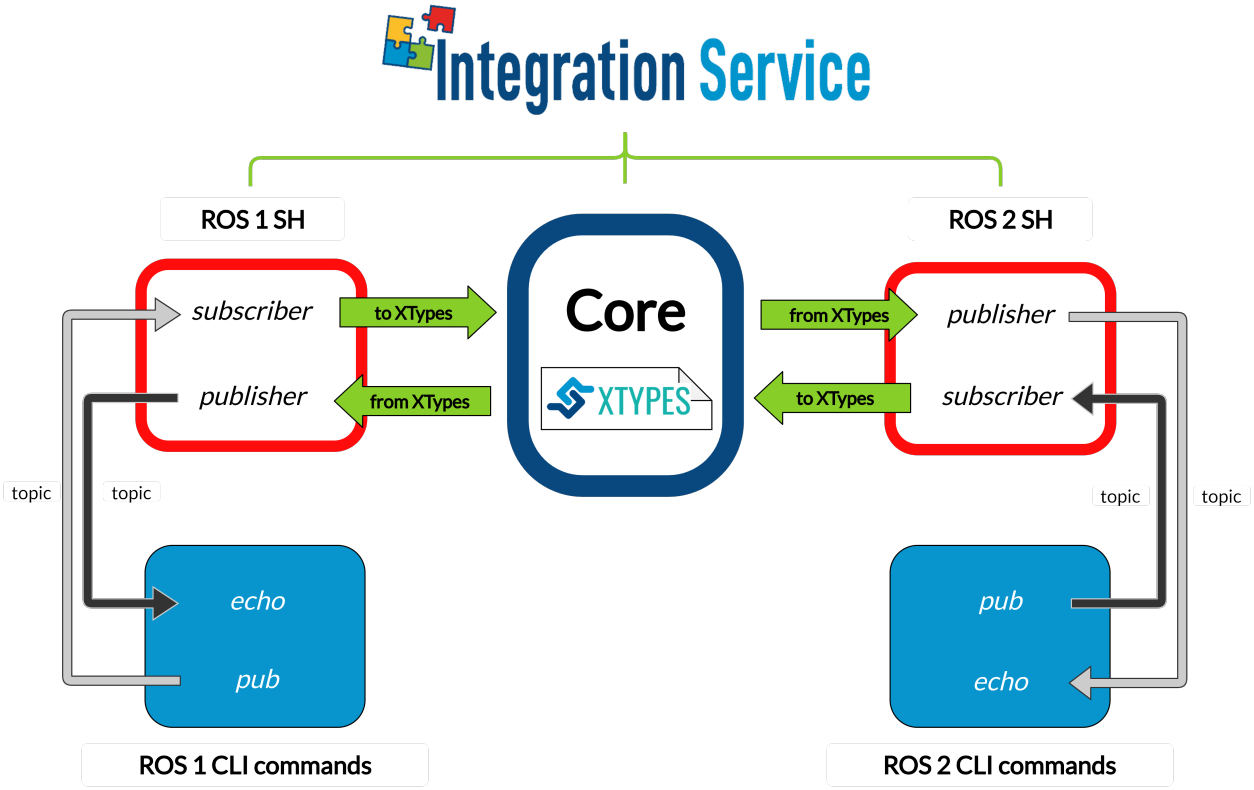
You should see the message echoed in the ROS 2 terminal.

ROS 1 - ROS 2 bridge

Another typical situation of systems using different protocols is that of *ROS 1* and *ROS 2*.

By using *Integration Service*, this communication can be achieved with minimum user's effort. As both a **ROS 1 System Handle** and a *ROS 2 System Handle* already exist, the communication is straightforward.

In the example below, we show how *Integration Service* puts into communication two `pub-echo` examples, one from *ROS 2*, and the other from *ROS 1*.



Requirements

To prepare the deployment and setup the environment, you need to have *Integration Service* correctly installed in your system. To do so, please follow the steps delineated in the *Installation* section.

Also, to get this example working, the following requirements must be met:

- Having **ROS 2** (*Foxy* or superior) installed, with the `talker-listener` example working.
- Having the **ROS 2 System Handle** installed. You can download it from the ROS2-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS2-SH.git src/ROS2-SH
```

- Having **ROS 1** (*Melodic* or superior) installed, with the `pub-echo` example working.
- Having the **ROS 1 System Handle** installed. You can download it from the ROS1-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/dds-is-workspace
git clone https://github.com/eProsima/ROS1-SH.git src/ROS1-SH
```

After you have everything correctly installed, build the packages by running:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
colcon build --packages-skip-regex is-ros1
source /opt/ros/$<ROS1_DISTRO>/setup.bash
colcon build
```

Deployment

Below we explain how to deploy an example of this communication in both directions allowed.

ROS 1 pub to ROS 2 echo

To enable communication from *ROS 1* to *ROS 2*, open four terminals:

- In the first terminal, source the *ROS 1* installation and run the `roscore`:

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
roscore
```

- In the second terminal, source the *ROS 1* installation and launch the *ROS 1* `pub`:

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
rostopic pub -r 1 /hello_ros2 std_msgs/String "Hello, ros2"
```

- In the third terminal, source the *ROS 2* installation and launch the *ROS 2* `echo`:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 topic echo /hello_ros2 std_msgs/String
```

- In the fourth terminal, go to the `is-workspace` folder, source the *ROS 2*, the *ROS 1*, and local installations, and execute *Integration Service* with the `integration-service` command followed by the ros1_ros2__helloworld.yaml configuration file located in the `src/Integration-Service/examples/basic` folder:

```
cd ~/is-workspace
source /opt/ros/$<ROS2_DISTRO>/setup.bash
source /opt/ros/$<ROS1_DISTRO>/setup.bash
source install/setup.bash
integration-service src/Integration-Service/examples/basic/ros1_ros2__helloworld.
↪yaml
```

Once *Integration Service* is launched, the *ROS 1* `pub` and the *ROS 2* `echo` will start communicating.

ROS 2 pub to ROS 1 echo

To enable communication from *ROS 2* to *ROS 1*, open four terminals:

- In the first terminal, source the *ROS 1* installation and run the `roscore`:

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
roscore
```

- In the second terminal, source the *ROS 2* installation and launch the *ROS 2* `pub`:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 topic pub -r 1 /hello_ros1 std_msgs/String "{data: \"Hello, ros1\"}"
```

- In the third terminal, source the *ROS 1* installation and launch the *ROS 1* `echo`:

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
rostopic echo /hello_ros1
```

- In the fourth terminal, go to the `is-workspace` folder, source the *ROS 2*, the *ROS 1*, and local installations, and execute *Integration Service* with the `integration-service` command followed by the ros1_ros2__helloworld.yaml configuration file located in the `src/Integration-Service/examples/basic` folder:

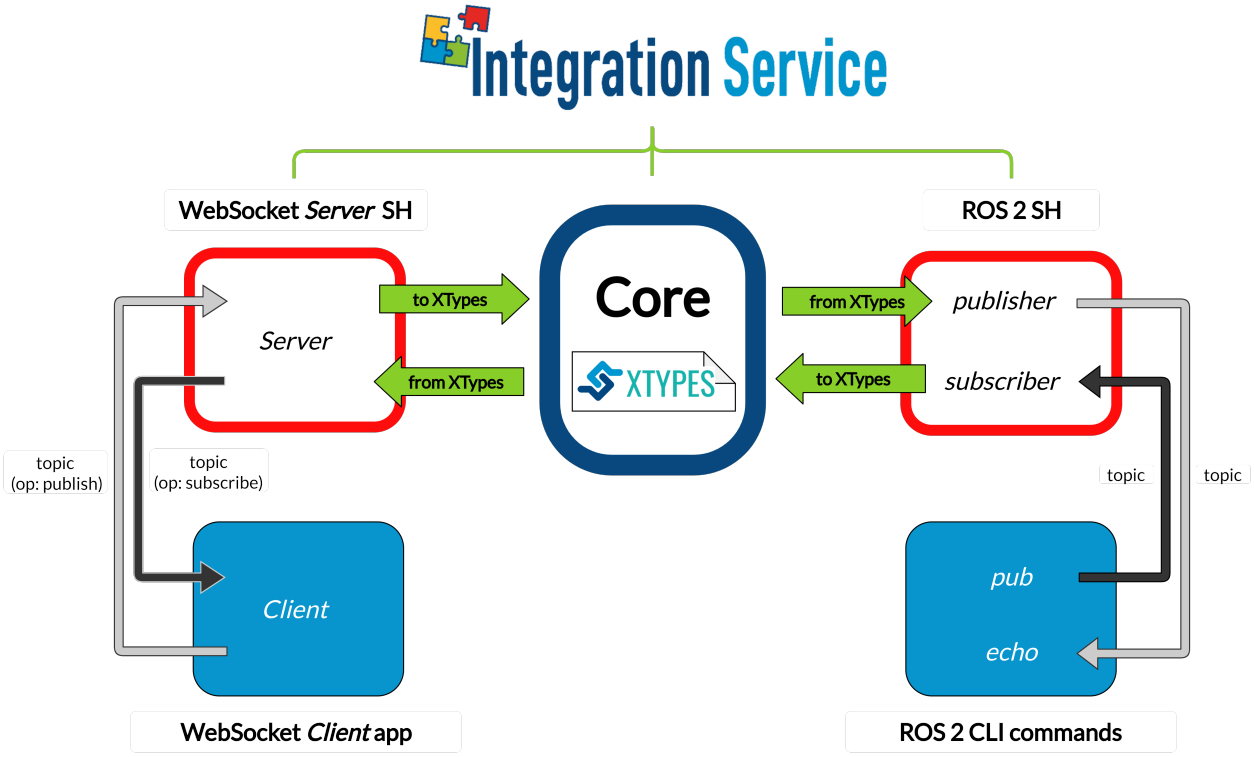
```
cd ~/is-workspace
source /opt/ros/$<ROS1_DISTRO>/setup.bash
source /opt/ros/$<ROS2_DISTRO>/setup.bash
source install/setup.bash
integration-service src/Integration-Service/examples/basic/ros1_ros2__helloworld.
↪yaml
```

Once *Integration Service* is launched, the *ROS 2* `pub` and the *ROS 1* `echo` will start communicating.

ROS 2 - WebSocket bridge

Another relevant use-case for *Integration Service* is that of connecting a *WebSocket* and a *ROS 2* application

The examples detailed below addresses the situation of a *ROS 2* `talker-listener` example communicating with a *WebSocket* `client`.



Requirements

To prepare the deployment and setup the environment, you need to have *Integration Service* correctly installed in your system. To do so, please follow the steps delineated in the *Installation* section.

Also, to get this example working, the following requirements must be met:

- Having **ROS 2** (*Foxy* or superior) installed, with the `talker-listener` example working.
- Having the **ROS 2 System Handle** installed. You can download it from the ROS2-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS2-SH.git src/ROS2-SH
```

- Having OpenSSL and WebSocket++ installed:

```
apt install libssl-dev libwebsocketpp-dev
```

- Having the **WebSocket System Handle** installed. You can download it from the WebSocket-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/WebSocket-SH.git src/WebSocket-SH
```

After you have everything correctly installed in your `is-workspace`, build the packages by running:

```
colcon build
```

Deployment

Below we explain how to deploy an example of this communication in both directions allowed.

ROS 2 pub to WebSocket client

To enable communication from *ROS 2* to a *WebSocket client*, open two terminals:

- In the first terminal, source your *ROS 2* installation and execute a *ROS 2* `pub`:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 topic pub /hello_websocket std_msgs/msg/String "{data: Hello WebSocket}"
```

- In the second terminal, go to the `is-workspace` folder, source the *ROS 2* and local installations, and execute *Integration Service* with the `integration-service` command followed by the ros2_websocket__helloworld.yaml configuration file located in the `src/Integration-Service/basic` folder:

```
cd ~/is-workspace
source /opt/ros/$<ROS2_DISTRO>/setup.bash
source install/setup.bash
integration-service src/Integration-Service/examples/basic/ros2_websocket__
↪helloworld.yaml
```

Up to this point, the *Integration Service* should have created a *WebSocket server* application within the *WebSocket System Handle*, to listen and handle petitions coming from a *WebSocket client*.

We will now explain how to simply test the intercommunication between *ROS 2* and a demo *WebSocket client* application, which can be found in websocket.org/echo webpage:

- First, under the **Location** section, connect to the *WebSocket server* automatically deployed by the *Integration Service*. To do so, and since the example is being run without SSL security, copy and paste the following URL into the *Location* field text box, and press **Connect**:

```
ws://localhost:80
```

  After this, you should see two WebSocket messages received automatically, due to the fact that the *WebSocket Server* hosted in the *Integration Service* detected an incoming connection: a *subscribe* operation message for the `hello_ros2` topic; and an *advertise* operation for the `hello_websocket` topic.

- Since the ROS 2 talker to WebSocket client example is being tested, we must first send a subscribe operation request for the `hello_websocket` topic. To do so, under the *Message* text box, enter the following and press *Send*:

```
{"op": "subscribe", "topic": "hello_websocket", "type": "std_msgs/String"}
```

After this, in the *Log* you should receive the following message from *ROS 2*:

```
RECEIVED: {"msg":{"data":"Hello WebSocket"},"op":"publish","topic":"hello_websocket"}
```

WebSocket client to ROS 2 echo

To enable communication from a *WebSocket client* to *ROS 2*, open two terminals:

- In the first terminal, source your *ROS 2* installation and execute a *ROS 2* `echo`:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 topic echo /hello_ros2 std_msgs/msg/String
```

- In the second terminal, go to the `is-workspace` folder, source the *ROS 2* and local installations, and execute *Integration Service* with the `integration-service` command followed by the ros2_websocket__helloworld.yaml configuration file located in the `src/Integration-Service/basic` folder:

```
cd ~/is-workspace
source /opt/ros/$<ROS2_DISTRO>/setup.bash
source install/setup.bash
integration-service src/Integration-Service/examples/basic/ros2_websocket__
↪helloworld.yaml
```

Up to this point, the *Integration Service* should have created a *WebSocket server* application within the *WebSocket System Handle*, to listen and handle petitions coming from a *WebSocket client*.

We will now explain how to simply test the intercommunication between *ROS 2* and a demo *WebSocket client* application, which can be found in websocket.org/echo webpage:

- First, under the **Location** section, connect to the *WebSocket server* automatically deployed by the *Integration Service*. To do so, and since the example is being run without SSL security, copy and paste the following URL into the *Location* field text box, and press **Connect**:

```
ws://localhost:80
```

After this, you should see two WebSocket messages received automatically, due to the fact that the *WebSocket Server* hosted in the *Integration Service* detected an incoming connection: a *subscribe* operation message for the `hello_ros2` topic; and an *advertise* operation for the `hello_websocket` topic.

- Since the WebSocket client to ROS 2 echo example is being tested, we must first send an advertise operation request for the `hello_ros2` topic. To do so, under the *Message* text box, enter the following and press *Send*:

```
{"op": "advertise", "topic": "hello_ros2", "type": "std_msgs/String"}
```

After this, we can send individual messages from the *WebSocket client*, using the *publish* operation:

```
{"op": "publish", "topic": "hello_ros2", "msg": {"data": "Hello ROS 2"}}
```

The messages should be shown in the *ROS 2* echo terminal.

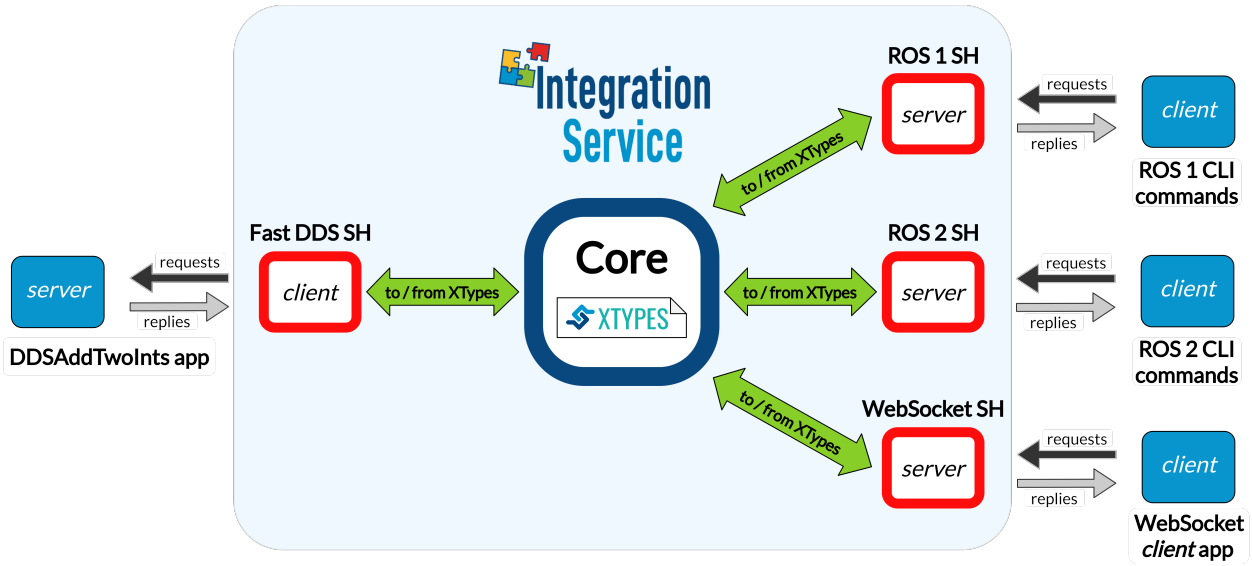
6.18.2 Server - Client

This page gathers all the *Integration Service* existing examples based on the **server - client** paradigm that connect different protocols which handle incompatible types.

DDS Service Server

This example tackles the task of bridging a *DDS* server with one or more client applications, implemented using a wide variety of protocols.

Specifically, we discuss how to forward petitions coming from *ROS 1*, *ROS 2* and a *WebSocket* service client applications to a *Fast DDS* `DDSAddTwoInts` server application, so that it can process them and fulfill each request with a proper answer message.



Note: If you are looking for an example on how to perform a service request from a *DDS* client to another protocol, please refer to any of the remaining examples in the *server/client examples* section.

Requirements

To prepare the deployment and setup the environment, you need to have *Integration Service* correctly installed in your system. To do so, please follow the steps delineated in the *Installation* section.

Also, to get this example working, the following requirements must be met:

- Having **Fast DDS** (*v.2.0.0* or superior) installed and the *Integration Service* `DDSAddTwoInts` example working. This example can be found in the main *Integration Service* repository, under the examples/utils/dds/DDSAddTwoInts folder; to compile it, you can either compile the whole *Integration Service* project using `colcon` with the CMake flag `BUILD_EXAMPLES` enabled; or execute the following steps:

```
cd ~/is-workspace/src/Integration-Service/examples/utils/dds/DDSAddTwoInts
mkdir build && cd build
cmake .. -DBUILD_EXAMPLES=ON && make
```

- Having the **Fast DDS System Handle** installed. You can download it from the FastDDS-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/FastDDS-SH.git src/FastDDS-SH
```

- Having **ROS 1** (*Melodic* or superior) installed and the *Integration Service* `example_interfaces` *ROS 1* package compiled. This package can be found in the main *Integration Service* repository, under the examples/utils/ros1/src/example_interfaces folder. To compile and install it:

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
cd ~/is-workspace/src/Integration-Service/example/utils/ros1/catkin_ws
catkin_make -DBUILD_EXAMPLES=ON -DCMAKE_INSTALL_PREFIX=/opt/ros/$<ROS1_DISTRO>␣
↪install
```

- Having the **ROS 1 System Handle** installed. You can download it from the ROS1-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS1-SH.git src/ROS1-SH
```

- Having **ROS 2** (*Foxy* or superior) installed, along with the `example_interfaces` types package. To install it:

```
apt install ros-$<ROS2_DISTRO>-example-interfaces
```

- Having the **ROS 2 System Handle** installed. You can download it from the ROS2-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS2-SH.git src/ROS2-SH src/ros2-sh
```

- Having OpenSSL and WebSocket++ installed:

```
apt install libssl-dev libwebsocketpp-dev
```

- Having the **WebSocket System Handle** installed. You can download it from the WebSocket-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/WebSocket-SH.git src/WebSocket-SH
```

After you have everything correctly installed in your `is-workspace`, build the packages by running:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
colcon build --packages-skip-regex is-ros1 -DMIX_ROS_PACKAGES="example_interfaces"
source /opt/ros/$<ROS1_DISTRO>/setup.bash
colcon build --cmake-args -DBUILD_EXAMPLES=ON -DMIX_ROS_PACKAGES="example_interfaces"
```

Deployment

Below we explain how to deploy a full example of this communication, calling the *DDS* service from each of the available clients.

Launch the DDS AddTwoInts server

To do so, open a terminal, go to the `is-workspace` folder and execute the following command:

```
cd ~/is-workspace
./build/is-examples/dds/DDSAddTwoInts/DDSAddTwoInts -m server
```

The server will start running under the default *DDS* domain ID 0 listening for incoming petitions.

Execute Integration Service

Open two terminals:

- In the first terminal, source the *ROS 1* installation and run the `roscore`:

  ```
  source /opt/ros/$<ROS1_DISTRO>/setup.bash
  roscore
  ```

- In the second terminal, go to the `is-workspace` folder, source the *ROS 1*, *ROS 2* and local installations, and execute *Integration Service* with the `integration-service` command followed by the fastdds_server__addtwoints.yaml configuration file located in the `src/Integration-Service/examples/basic` folder.

  ```
  source /opt/ros/$<ROS1_DISTRO>/setup.bash
  source /opt/ros/$<ROS2_DISTRO>/setup.bash
  source install/setup.bash
  integration-service src/Integration-Service/examples/basic/fastdds_server__
  ↪addtwoints.yaml
  ```

Call the service from ROS 1

In a new terminal, source your *ROS 1* installation and invoke the service by executing the following instructions:

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
rosservice call /add_two_ints 3 4
```

You should receive the following output from the *DDS* server processing the petition:

```
sum: 7
```

Call the service from ROS 2

In a new terminal, source your *ROS 2* installation and invoke the service by executing the following instruction:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts "{a: 5, b: 17}"
```

You should receive the following output from the *DDS* server processing the petition:

```
waiting for service to become available...
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=5, b=17)

response:
example_interfaces.srv.AddTwoInts_Response(sum=22)
```

Call the service from WebSocket

The *WebSocket client* demo application used for this example can be found in the websocket.org/echo webpage:

- First, under the **Location** section, connect to the *WebSocket server* automatically deployed by the *Integration Service*. To do so, and since the example is being run without SSL security, copy and paste the following URL into the *Location* field text box, and press **Connect**:

  ```
  ws://localhost:80
  ```

- Now it is time to advertise the service we want to use; to do so, under the *Message* text box, enter the following and press *Send*:

  ```
  {"op": "advertise_service", "service": "add_two_ints", "request_type":
  ↪"AddTwoInts_Request", "reply_type": "AddTwoInts_Response"}
  ```

- Finally, after the service has been advertised, call it by sending the following message from the *WebSocket* echo:

  ```
  {"op": "call_service", "service": "add_two_ints", "args": {"a": 14, "b": 25}}
  ```

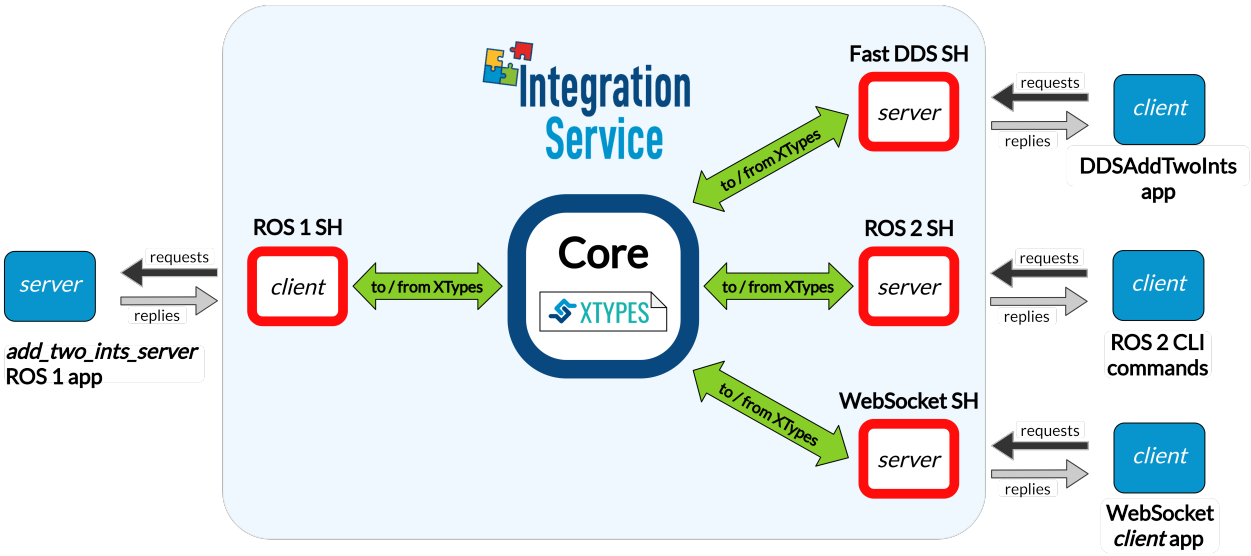
After this, in the *Log*, you should receive the following response from the *DDS* server:

```
RECEIVED: {"op":"service_response","result":true,"service":"add_two_ints","values":{
↪"sum":39}}
```

ROS 1 Service Server

This example tackles the task of bridging a *ROS 1* server with one or more client applications, implemented using a wide variety of protocols.

Specifically, we discuss how to forward petitions coming from *Fast DDS*, *ROS 2* and a *WebSocket* service client applications to a *ROS 1* `add_two_ints_server` server application, from a provided *Integration Service* custom *ROS 1* package called `add_two_ints_server`; so that it can process them and fulfill each request with a proper answer message.



Note: If you are looking for an example on how to perform a service request from a *ROS 1* client to another protocol, please refer to any of the remaining examples in the *server/client examples* section.

Requirements

To prepare the deployment and setup the environment, you need to have *Integration Service* correctly installed in your system. To do so, please follow the steps delineated in the *Installation* section.

Also, to get this example working, the following requirements must be met:

- Having **Fast DDS** (*v.2.0.0* or superior) installed and the *Integration Service* `DDSAddTwoInts` example working. This example can be found in the main *Integration Service* repository, under the examples/utils/dds/DDSAddTwoInts folder; to compile it, you can either compile the whole *Integration Service* project using `colcon` with the CMake flag `BUILD_EXAMPLES` enabled; or execute the following steps:

```
cd ~/is-workspace/src/Integration-Service/examples/utils/dds/DDSAddTwoInts
mkdir build && cd build
cmake .. -DBUILD_EXAMPLES=ON && make
```

- Having the **Fast DDS System Handle** installed. You can download it from the FastDDS-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/FastDDS-SH.git src/FastDDS-SH
```

- Having **ROS 1** (*Melodic* or superior) installed and the *Integration Service* `example_interfaces` and `add_two_ints_server` *ROS 1* packages compiled. These package can be found in the main *Integration Service* repository, under the examples/utils/ros1 folder. The former one needs to be compiled and installed before the whole the rest of the *ROS 1*-related *Integration Service* packages; to do so:

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
cd ~/is-workspace/src/Integration-Service/example/utils/ros1/catkin_ws
catkin_make -DBUILD_EXAMPLES=ON -DCMAKE_INSTALL_PREFIX=/opt/ros/$<ROS1_DISTRO>␣
↪install
```

- Having the **ROS 1 System Handle** installed. You can download it from the ROS1-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS1-SH.git src/ROS1-SH
```

- Having **ROS 2** (*Foxy* or superior) installed, along with the `example_interfaces` package. To install it:

```
apt install ros-$<ROS2_DISTRO>-example-interfaces
```

- Having the **ROS 2 System Handle** installed. You can download it from the ROS2-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS2-SH.git src/ROS2-SH src/ros2-sh
```

- Having OpenSSL and WebSocket++ installed:

```
apt install libssl-dev libwebsocketpp-dev
```

- Having the **WebSocket System Handle** installed. You can download it from the WebSocket-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/WebSocket-SH.git src/WebSocket-SH
```

After you have everything correctly installed in your `is-workspace`, build the packages by running:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
colcon build --packages-skip-regex is-ros1 -DMIX_ROS_PACKAGES="example_interfaces"
source /opt/ros/$<ROS1_DISTRO>/setup.bash
colcon build --cmake-args -DBUILD_EXAMPLES=ON -DMIX_ROS_PACKAGES="example_interfaces"
```

Deployment

Below we explain how to deploy a full example of this communication, calling the *ROS 1* service from each of the available clients.

Launch the ROS 1 *add_two_ints_server* node

Open two terminals:

- In the first terminal, source the *ROS 1* installation and run the `roscore`:

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
roscore
```

- In the second terminal, go to the `is-workspace` directory. Then, source the *ROS 1* and local installations, and execute the `add_two_ints_server` *ROS 1* node:

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
rosrun add_two_ints_server add_two_ints_server_node
```

The server will start running as an independent *ROS 1* node, listening for incoming petitions.

Execute Integration Service

To launch *Integration Service* open a terminal and go to the `is-workspace` folder. Then, source the *ROS 1*, *ROS 2* and local installations, and execute *Integration Service* with the `integration-service` command followed by the ros1_server__addtwoints.yaml configuration file located in the `src/Integration-Service/examples/basic` folder.

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
source /opt/ros/$<ROS2_DISTRO>/setup.bash
source install/setup.bash
integration-service src/Integration-Service/examples/basic/ros1_server__
↪addtwoints.yaml
```

Call the service from Fast DDS

In a new terminal, go to the `is-workspace` folder and execute the following command:

```
./build/is-examples/dds/DDSAddTwoInts/DDSAddTwoInts -m client -c <number_of_requests>
```

The *DDSAddTwoInts* example application will request to add two numbers an specific amount of times, specified with the `-c` flag; if not present, ten requests will be performed by default.

For instance, if `-c 4`, should see something like this in your screen, indicating that the *ROS 1* server is processing the requests:

```
AddTwoIntsService client running under DDS Domain ID: 0
AddTwoIntsService client performing 4 requests.
AddTwoIntsService client:
        - Request 1 + 3
        - Received response: 4
AddTwoIntsService client:
        - Request 2 + 4
        - Received response: 6
AddTwoIntsService client:
        - Request 3 + 5
        - Received response: 8
AddTwoIntsService client:
        - Request 4 + 6
        - Received response: 10
```

Call the service from ROS 2

In a new terminal, source your *ROS 2* installation and invoke the service by executing the following instructions:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts "{a: 5, b: 17}"
```

You should receive the following output from the *ROS 1* server processing the petition:

```
waiting for service to become available...
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=5, b=17)

response:
example_interfaces.srv.AddTwoInts_Response(sum=22)
```

Call the service from WebSocket

The *WebSocket client* demo application used for this example can be found in the websocket.org/echo webpage:

- First, under the **Location** section, connect to the *WebSocket server* automatically deployed by the *Integration Service*. To do so, and since the example is being run without SSL security, copy and paste the following URL into the *Location* field text box, and press **Connect**:

```
ws://localhost:80
```

- Now it is time to advertise the service we want to use; to do so, under the *Message* text box, enter the following and press *Send*:

```
{"op": "advertise_service", "service": "add_two_ints", "request_type":
↪"AddTwoInts_Request", "reply_type": "AddTwoInts_Response"}
```

- Finally, after the service has been advertised, call it by sending the following message from the *WebSocket* echo:

```
{"op": "call_service", "service": "add_two_ints", "args": {"a": 14, "b": 25}}
```

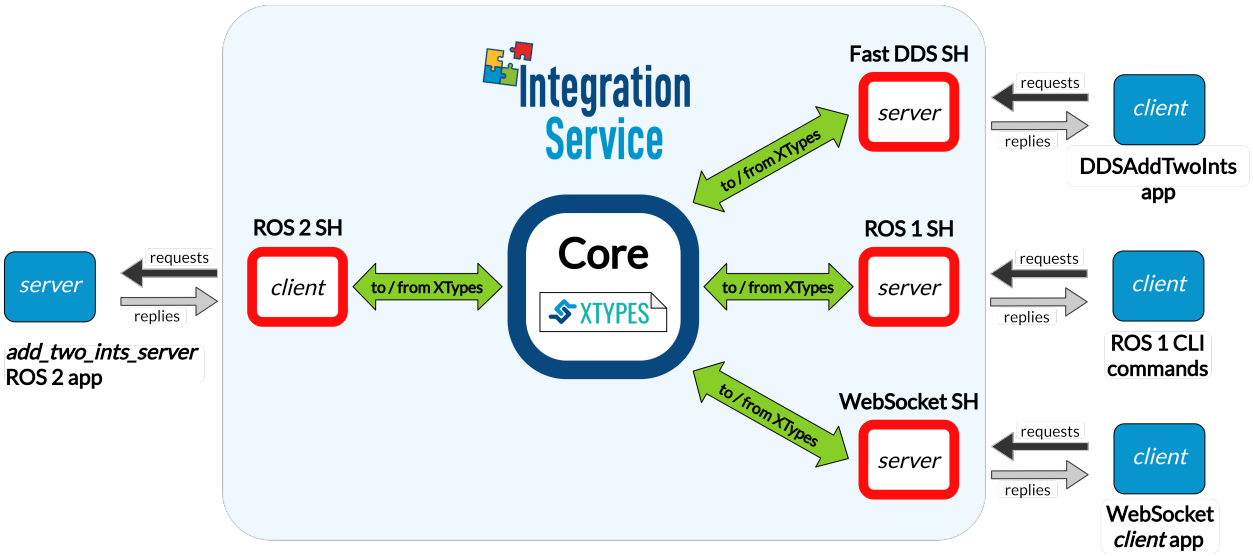
After this, in the *Log*, you should receive the following response from the *ROS 1* server:

```
RECEIVED: {"op":"service_response","result":true,"service":"add_two_ints","values":{
↪"sum":39}}
```

ROS 2 Service Server

This example tackles the task of bridging a *ROS 2* server with one or more client applications, implemented using a wide variety of protocols.

Specifically, we discuss how to forward petitions coming from *Fast DDS*, *ROS 1* and a *WebSocket* service client applications to a *ROS 2* `add_two_ints_server` server application, from the built-in *ROS 2* package `demo_nodes_cpp`; so that it can process them and fulfill each request with a proper answer message.



Note: If you are looking for an example on how to perform a service request from a *ROS 2* client to another protocol,

please refer to any of the remaining examples in the *server/client examples* section.

Requirements

To prepare the deployment and setup the environment, you need to have *Integration Service* correctly installed in your system. To do so, please follow the steps delineated in the *Installation* section.

Also, to get this example working, the following requirements must be met:

- Having **Fast DDS** (*v.2.0.0* or superior) installed and the *Integration Service* `DDSAddTwoInts` example working. This example can be found in the main *Integration Service* repository, under the examples/utils/dds/DDSAddTwoInts folder; to compile it, you can either compile the whole *Integration Service* project using `colcon` with the CMake flag `BUILD_EXAMPLES` enabled; or execute the following steps:

```
cd ~/is-workspace/src/Integration-Service/examples/utils/dds/DDSAddTwoInts
mkdir build && cd build
cmake .. -DBUILD_EXAMPLES=ON && make
```

- Having the **Fast DDS System Handle** installed. You can download it from the FastDDS-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/FastDDS-SH.git src/FastDDS-SH
```

- Having **ROS 1** (*Melodic* or superior) installed and the *Integration Service* `example_interfaces` *ROS 1* package compiled. This package can be found in the main *Integration Service* repository, under the examples/utils/ros1/src/example_interfaces folder. To compile and install it:

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
cd ~/is-workspace/src/Integration-Service/example/utils/ros1/catkin_ws
catkin_make -DBUILD_EXAMPLES=ON -DCMAKE_INSTALL_PREFIX=/opt/ros/$<ROS1_DISTRO>␣
↪install
```

- Having the **ROS 1 System Handle** installed. You can download it from the ROS1-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS1-SH.git src/ROS1-SH
```

- Having **ROS 2** (*Foxy* or superior) installed, along with the `demo_nodes_cpp` package. To install it:

```
apt install ros-$<ROS2_DISTRO>-demo-nodes-cpp
```

- Having the **ROS 2 System Handle** installed. You can download it from the ROS2-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS2-SH.git src/ROS2-SH src/ros2-sh
```

- Having OpenSSL and WebSocket++ installed:

```
apt install libssl-dev libwebsocketpp-dev
```

- Having the **WebSocket System Handle** installed. You can download it from the WebSocket-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/WebSocket-SH.git src/WebSocket-SH
```

After you have everything correctly installed in your `is-workspace`, build the packages by running:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
colcon build --packages-skip-regex is-ros1 -DMIX_ROS_PACKAGES="example_interfaces"
source /opt/ros/$<ROS1_DISTRO>/setup.bash
colcon build --cmake-args -DBUILD_EXAMPLES=ON -DMIX_ROS_PACKAGES="example_interfaces"
```

Deployment

Below we explain how to deploy a full example of this communication, calling the *ROS 2* service from each of the available clients.

Launch the ROS 2 *demo_nodes_cpp* add_two_ints_server

To do so, open a terminal and execute the following command:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 run demo_nodes_cpp add_two_ints_server
```

The server will start running as an independent *ROS 2* node, listening for incoming petitions.

Execute Integration Service

Open two terminals:

- In the first terminal, source the *ROS 1* installation and run the `roscore`:

  ```
  source /opt/ros/$<ROS1_DISTRO>/setup.bash
  roscore
  ```

- In the second terminal, go to the `is-workspace` folder, source the *ROS 1*, *ROS 2* and local installations, and execute *Integration Service* with the `integration-service` command followed by the ros2_server__addtwoints.yaml configuration file located in the `src/Integration-Service/examples/basic` folder.

  ```
  source /opt/ros/$<ROS1_DISTRO>/setup.bash
  source /opt/ros/$<ROS2_DISTRO>/setup.bash
  source install/setup.bash
  integration-service src/Integration-Service/examples/basic/ros2_server__
  ↪addtwoints.yaml
  ```

Call the service from Fast DDS

In a new terminal, go to the `is-workspace` folder and execute the following command:

```
./build/is-examples/dds/DDSAddTwoInts/DDSAddTwoInts -m client -c <number_of_requests>
```

The *DDSAddTwoInts* example application will request to add two numbers an specific amount of times, specified with the `-c` flag; if not present, ten requests will be performed by default.

For instance, if `-c 4`, should see something like this in your screen, indicating that the *ROS 2* server is processing the requests:

```
AddTwoIntsService client running under DDS Domain ID: 0
AddTwoIntsService client performing 4 requests.
AddTwoIntsService client:
        - Request 1 + 3
        - Received response: 4
AddTwoIntsService client:
        - Request 2 + 4
        - Received response: 6
AddTwoIntsService client:
        - Request 3 + 5
        - Received response: 8
AddTwoIntsService client:
        - Request 4 + 6
        - Received response: 10
```

Call the service from ROS 1

In a new terminal, source your *ROS 1* installation and invoke the service by executing the following instructions:

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
rosservice call /add_two_ints 3 4
```

You should receive the following output from the *ROS 2* server processing the petition:

```
sum: 7
```

Call the service from WebSocket

The *WebSocket client* demo application used for this example can be found in the websocket.org/echo webpage:

- First, under the **Location** section, connect to the *WebSocket server* automatically deployed by the *Integration Service*. To do so, and since the example is being run without SSL security, copy and paste the following URL into the *Location* field text box, and press **Connect**:

  ```
  ws://localhost:80
  ```

- Now it is time to advertise the service we want to use; to do so, under the *Message* text box, enter the following and press *Send*:

  ```
  {"op": "advertise_service", "service": "add_two_ints", "request_type":
  ↪"AddTwoInts_Request", "reply_type": "AddTwoInts_Response"}
  ```

- Finally, after the service has been advertised, call it by sending the following message from the *WebSocket* echo:

```
{"op": "call_service", "service": "add_two_ints", "args": {"a": 14, "b": 25}}
```

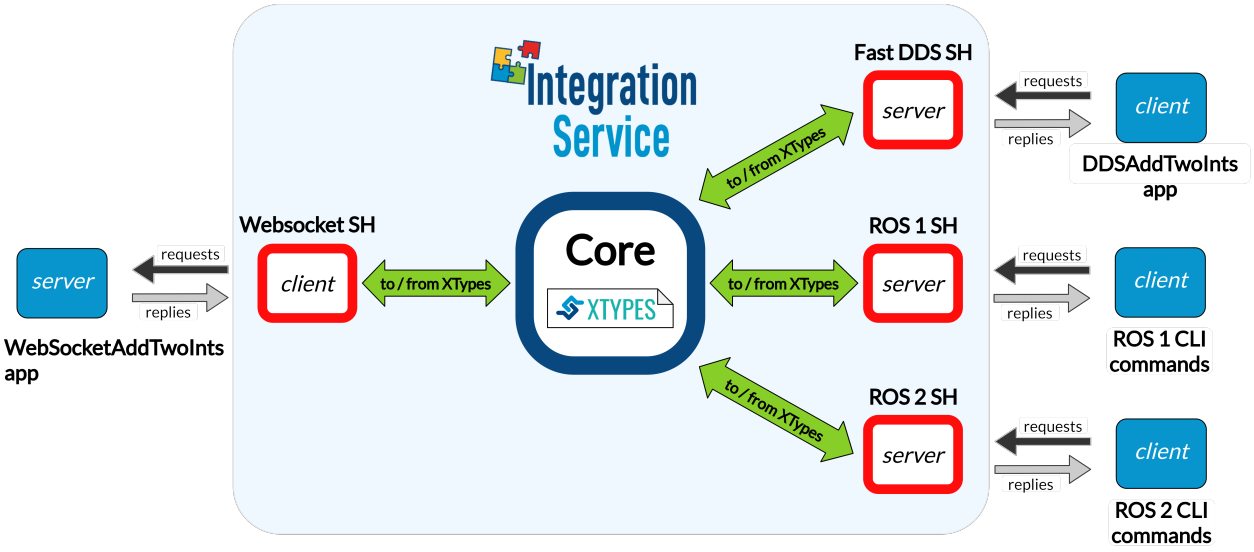
After this, in the *Log*, you should receive the following response from the *ROS 2* server:

```
RECEIVED: {"op":"service_response","result":true,"service":"add_two_ints","values":{
↪"sum":39}}
```

WebSocket Service Server

This example tackles the task of bridging a *WebSocket* server with one or more client applications, implemented using a wide variety of protocols.

Specifically, we discuss how to forward petitions coming from *Fast DDS*, *ROS 1* and *ROS 2* service client applications to a *WebSocket* `WebSocketAddTwoInts` server application, so that it can process them and fulfill each request with a proper answer message.



Note: If you are looking for an example on how to perform a service request from a *WebSocket* client to another protocol, please refer to any of the remaining examples in the *server/client examples* section.

Requirements

To prepare the deployment and setup the environment, you need to have *Integration Service* correctly installed in your system. To do so, please follow the steps delineated in the *Installation* section.

Also, to get this example working, the following requirements must be met:

- Having **Fast DDS** (*v.2.0.0* or superior) installed and the *Integration Service* `DDSAddTwoInts` example working. This example can be found in the main *Integration Service* repository, under the examples/utils/dds/DDSAddTwoInts folder; to compile it, you can either compile the whole *Integration Service* project using `colcon` with the CMake flag `BUILD_EXAMPLES` enabled; or execute the following steps:

```
cd ~/is-workspace/src/Integration-Service/examples/utils/dds/DDSAddTwoInts
mkdir build && cd build
cmake .. -DBUILD_EXAMPLES=ON && make
```

- Having the **Fast DDS System Handle** installed. You can download it from the FastDDS-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/FastDDS-SH.git src/FastDDS-SH
```

- Having **ROS 1** (*Melodic* or superior) installed and the *Integration Service* `example_interfaces` *ROS 1* package compiled. This package can be found in the main *Integration Service* repository, under the examples/utils/ros1/src/example_interfaces folder. To compile and install it:

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
cd ~/is-workspace/src/Integration-Service/example/utils/ros1/catkin_ws
catkin_make -DBUILD_EXAMPLES=ON -DCMAKE_INSTALL_PREFIX=/opt/ros/$<ROS1_DISTRO>␣
↪install
```

- Having the **ROS 1 System Handle** installed. You can download it from the ROS1-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS1-SH.git src/ROS1-SH
```

- Having **ROS 2** (*Foxy* or superior) installed, along with the `example_interfaces` types package. To install it:

```
apt install ros-$<ROS2_DISTRO>-example-interfaces
```

- Having the **ROS 2 System Handle** installed. You can download it from the ROS2-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS2-SH.git src/ROS2-SH src/ros2-sh
```

- Having OpenSSL and WebSocket++ installed:

```
apt install libssl-dev libwebsocketpp-dev
```

  Also, the *Integration Service* `WebSocketAddTwoInts` example will be needed for the tutorial. This example application can be found in the main *Integration Service* repository, under the examples/utils/websocket/WebSocketAddTwoInts folder. To compile it, you can either compile the whole *Integration Service* project using `colcon` with the CMake flag `BUILD_EXAMPLES` enabled; or execute the following steps:

```
cd ~/is-workspace/src/Integration-Service/examples/utils/websocket/
↪WebSocketAddTwoInts
mkdir build && cd build
cmake .. -DBUILD_EXAMPLES=ON && make
```

- Having the **WebSocket System Handle** installed. You can download it from the WebSocket-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/WebSocket-SH.git src/WebSocket-SH
```

After you have everything correctly installed in your `is-workspace`, build the packages by running:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
colcon build --packages-skip-regex is-ros1 -DMIX_ROS_PACKAGES="example_interfaces"
```

(continues on next page)

(continued from previous page)

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
colcon build --cmake-args -DBUILD_EXAMPLES=ON -DMIX_ROS_PACKAGES="example_interfaces"
```

Deployment

Below we explain how to deploy a full example of this communication, calling the *WebSocket* service from each of the available clients.

Launch the WebSocket AddTwoInts server

To do so, open a terminal, go to the `is-workspace` folder and execute the following command:

```
cd ~/is-workspace
./build/is-examples/websocket/WebSocketAddTwoInts/WebSocketAddTwoInts
```

The *WebSocket* server will start running, listening for incoming client connection petitions; after that, it will we able to dispatch service request petitions with a proper answer message.

Execute Integration Service

Open two terminals:

- In the first terminal, source the *ROS 1* installation and run the `roscore`:

  ```
  source /opt/ros/$<ROS1_DISTRO>/setup.bash
  roscore
  ```

- In the second terminal, go to the `is-workspace` folder, source the *ROS 1*, *ROS 2* and local installations, and execute *Integration Service* with the `integration-service` command followed by the websocket_server__addtwoints.yaml configuration file located in the `src/Integration-Service/examples/basic` folder.

  ```
  source /opt/ros/$<ROS1_DISTRO>/setup.bash
  source /opt/ros/$<ROS2_DISTRO>/setup.bash
  source install/setup.bash
  integration-service src/Integration-Service/examples/basic/websocket_server__
  ↪addtwoints.yaml
  ```

Call the service from Fast DDS

In a new terminal, go to the `is-workspace` folder and execute the following command:

```
./build/is-examples/dds/DDSAddTwoInts/DDSAddTwoInts -m client -c <number_of_requests>
```

The *DDSAddTwoInts* example application will request to add two numbers an specific amount of times, specified with the `-c` flag; if not present, ten requests will be performed by default.

For instance, if `-c 4`, should see something like this in your screen, indicating that the *WebSocket* server is processing the requests:

```
AddTwoIntsService client running under DDS Domain ID: 0
AddTwoIntsService client performing 4 requests.
AddTwoIntsService client:
        - Request 1 + 3
        - Received response: 4
AddTwoIntsService client:
        - Request 2 + 4
        - Received response: 6
AddTwoIntsService client:
        - Request 3 + 5
        - Received response: 8
AddTwoIntsService client:
        - Request 4 + 6
        - Received response: 10
```

Call the service from ROS 1

In a new terminal, source your *ROS 1* installation and invoke the service by executing the following instructions:

```
source /opt/ros/$<ROS1_DISTRO>/setup.bash
rosservice call /add_two_ints 3 4
```

You should receive the following output from the *WebSocket* server processing the petition:

```
sum: 7
```

Call the service from ROS 2

In a new terminal, source your *ROS 2* installation and invoke the service by executing the following instruction:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts "{a: 5, b: 17}"
```

You should receive the following output from the *WebSocket* server processing the petition:

```
waiting for service to become available...
requester: making request: example_interfaces.srv.AddTwoInts_Request(a=5, b=17)

response:
example_interfaces.srv.AddTwoInts_Response(sum=22)
```

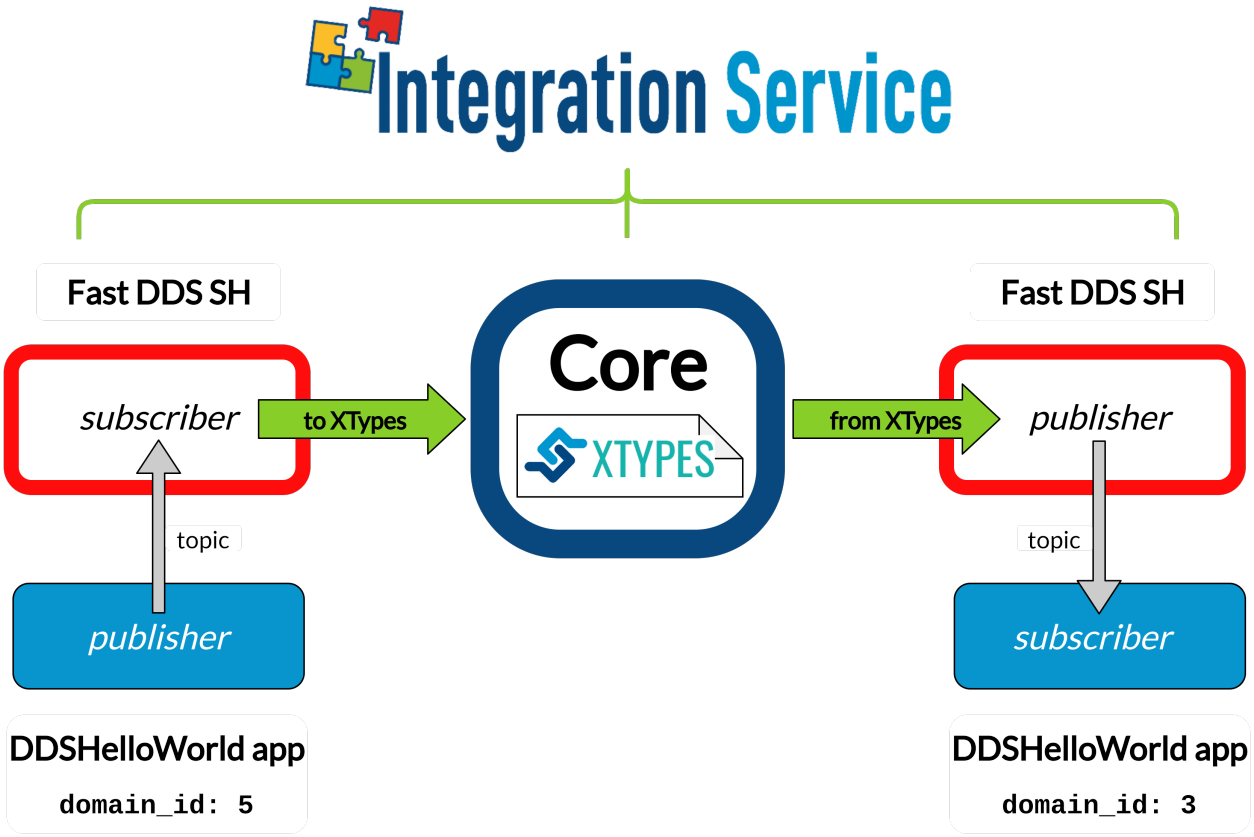
6.19 Same Protocol

This page gathers all the existing examples for *Integration Service* that connect applications written under the **same protocol**. They are not focused on showing how effective *Integration Service* is capable of translate one protocol's types into another's (see *Different Protocols* for that purpose); instead, this section tries to depict how easy is for *Integration Service* to bridge logically isolated applications written under the same protocol.

6.19.1 DDS Domain ID change

A very typical scenario within the *DDS* ecosystem is that of two applications running under different *DDS* domain IDs, so that they are isolated from each other; however, it might be required to bridge some of the topics published by one of the applications, so that a subscriber on a different domain ID can consume this information. This is where the *Integration Service* plays a fundamental role, by allowing to bridge two *DDS* dataspaces easily.

The steps described below allows a *Fast DDS HelloWorld* publisher application, running under a certain domain ID, to communicate with a *Fast DDS HelloWorld* subscriber application, which is running under a different domain ID.



Requirements

To prepare the deployment and setup the environment, you need to have *Integration Service* correctly installed in your system. To do so, please follow the steps delineated in the *Installation* section.

Also, to get this example working, the following requirements must be met:

- Having **Fast DDS** (*v.2.0.0* or superior) installed and the *Integration Service* `DDSHelloWorld` example working. This example can be found in the main *Integration Service* repository, under the examples/utils/dds/DDSHelloWorld folder; to compile it, you can either compile the whole *Integration Service* project using `colcon` with the CMake flag `BUILD_EXAMPLES` enabled; or execute the following steps:

```
cd ~/is-workspace/src/Integration-Service/examples/utils/dds/DDSHelloWorld
mkdir build && cd build
cmake .. -DBUILD_EXAMPLES=ON && make
```

- Having the **Fast DDS System Handle** installed. You can download it from the dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/FastDDS-SH.git src/FastDDS-SH
```

After you have everything correctly installed in your `is-workspace`, build the packages by running:

```
colcon build --cmake-args -DBUILD_FASTDDS_EXAMPLES=ON
```

Deployment

Below we explain how to deploy an example of this use case. To do so, open three terminals:

- In the first terminal, execute the `DDSHelloWorld` example, as a subscriber running under *DDS* domain ID **3** and subscribed to the topic `hello_domain_3`:

```
cd ~/is-workspace/build/is-examples/dds/DDSHelloWorld
./DDSHelloWorld -m subscriber -n hello_domain_3 -d 3
```

- In the second terminal, execute the `DDSHelloWorld` example, as a publisher running under *DDS* domain ID **5** and publishing data to the topic `hello_domain_3`:

```
cd ~/is-workspace/build/is-examples/dds/DDSHelloWorld
./DDSHelloWorld -m publisher -n hello_domain_3 -d 5
```

Up to this point, no communication should be seen between the publisher and the subscriber, due to the domain ID change. This is where *Integration Service* comes into play to make the communication possible.

- In the third terminal, go to the `is-workspace` folder, source the local installations, and execute *Integration Service* with the `integration-service` command followed by the fastdds__domain_id_change.yaml configuration file located in the `src/Integration-Service/examples/basic` folder:

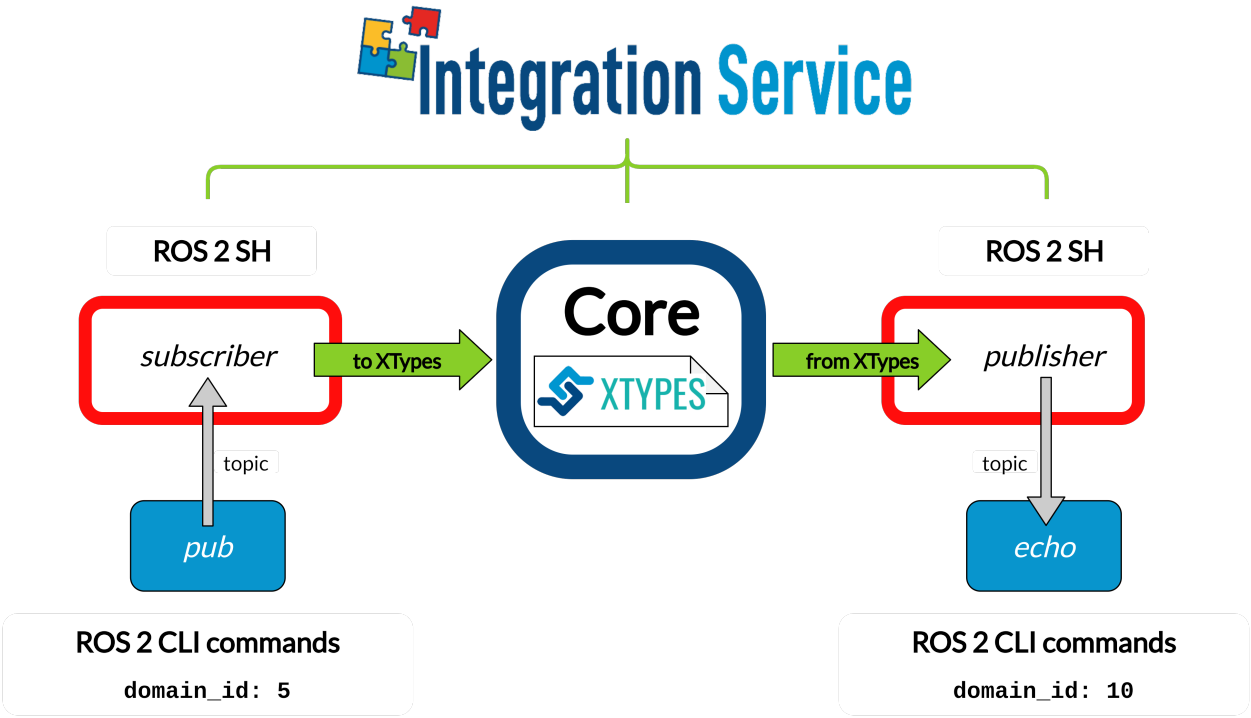
```
cd ~/is-workspace
source install/setup.bash
integration-service src/Integration-Service/examples/basic/fastdds__domain_id_
↪change.yaml
```

Once the last command is executed, the two applications will start communicating.

6.19.2 ROS 2 Domain ID change

A very typical scenario within the *ROS 2* ecosystem is that of two applications running under different *ROS 2* domain IDs, so that they are isolated from each other; however, it might be required to bridge some of the published topics by the first application, so that a subscriber on the second application, running on a different domain ID can consume this information. This is where the *Integration Service* plays a fundamental role, by allowing to bridge two *ROS 2* dataspaces easily.

The steps described below allow a *ROS 2* publisher application, running under a certain domain ID, to communicate with a *ROS 2* subscriber (echo) application, which is running under a different domain ID.



Requirements

To prepare the deployment and setup the environment, you need to have *Integration Service* correctly installed in your system. To do so, please follow the steps delineated in the *Installation* section.

Also, to get this example working, the following requirements must be met:

- Having **ROS 2** (*Foxy* or superior) installed, with the `talker-listener` example working.
- Having the **ROS 2 System Handle** installed. You can download it from the dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS2-SH.git src/ROS2-SH
```

After you have everything correctly installed in your `is-workspace`, build the packages by running:

```
colcon build
```

Deployment

Below we explain how to deploy an example of this use case. To do so, open three terminals:

- In the first terminal, source the *ROS 2* installation and launch the *ROS 2* `pub` application, under domain ID **5**:

```
ROS_DOMAIN_ID=5 ros2 topic pub -r 1 /string_topic std_msgs/String "{data: \"Hello,
↪ ROS 2 domain 10\"}"
```

- In the second terminal, source the *ROS 2* installation and launch the *ROS 2* `echo` application, under domain ID **10**:

```
ROS_DOMAIN_ID=10 ros2 topic echo /string_topic std_msgs/msg/String
```

Up to this point, no communication should be seen between the publisher and the subscriber, due to the domain ID change. This is where *Integration Service* comes into play to make the communication possible.

- In the third terminal, go to the `is-workspace` folder, source the local installations, and execute *Integration Service* with the `integration-service` command followed by the ros2__domain_id_change.yaml configuration file located in the `src/Integration-Service/basic` folder:

```
cd ~/is-workspace
source install/setup.bash
integration-service src/Integration-Service/examples/basic/ros2__domain_id_change.
↪yaml
```

Once the last command is executed, the two applications will start communicating.

6.20 WAN Communication

This page gathers all the existing examples for *Integration Service* that allow to bridge through the Internet systems hosted by logically separated WANs, which could be even located in different geographical regions.

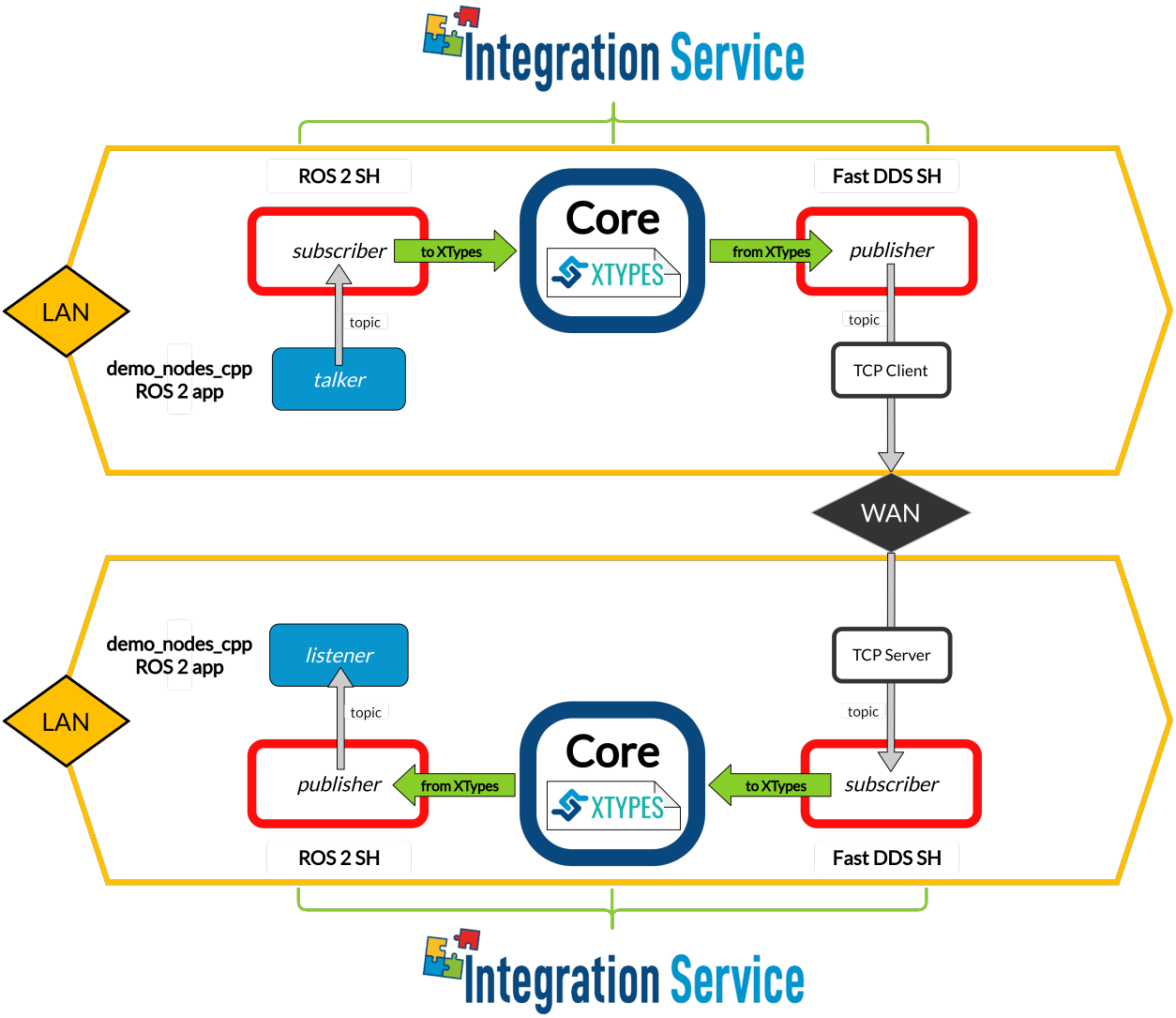
6.20.1 WAN-TCP tunneling over DDS

One of the most critical and powerful use-cases of *Integration Service* is that of two systems located in different geographical regions which need to communicate through the Internet, using a *WAN* connection.

Using a pair of *Integration Service* instances, one for each system, this scenario can be addressed with a secure TCP tunnel thanks to the *SSL-TCP* capabilities of *Fast DDS*.

Integration Service acts as a gateway to translate each system to *DDS*, which then makes the tunneling over *SSL-TCP* possible. A proper configuration of the destination router and firewalls allows the communication.

The example discussed here illustrates, specifically, how to configure *Integration Service* to achieve WAN communication between two separated *ROS 2* instances. Notice, however, that any other applications from systems integrated in the *Integration Service* ecosystem could be bridged across the *WAN*, thanks to the *Fast DDS System Handle* TCP tunneling capabilities.



Requirements

To prepare the deployment and setup the environment, you need to have *Integration Service* correctly installed in your system. To do so, please follow the steps delineated in the *Installation* section.

Also, to test this example properly, you need two separate subnets that are not connected but both with internet access, or a testing environment simulating this scenario (for example, two routers, with one of them acting as an ISP for the second).

Notice that both the route tables and the NAT must be configured so as to ensure proper port redirection before starting the test.

Note: The IP addresses shown here only serve the purpose of illustrating the example. The important information is the **real** public IP of the *server* machine. Also, its router must enable the NAT to forward the listening port to the *server*.

Also, to get this example working, the following requirements must be met in both machines:

- Having **ROS 2** (*Foxy* or superior) installed, with the `talker-listener` example working.
- Having the **ROS 2 System Handle** installed. You can download it from the ROS2-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/ROS2-SH.git src/ROS2-SH
```

- Having **Fast DDS** (v.2.0.0 or superior) installed.
- Having the **Fast DDS System Handle** installed. You can download it from the FastDDS-SH dedicated repository into the `is-workspace` where you have *Integration Service* installed:

```
cd ~/is-workspace
git clone https://github.com/eProsima/FastDDS-SH.git src/FastDDS-SH
```

After you have everything correctly installed, build the packages by running:

```
colcon build
```

Once the environment is prepared and tested (for example, using a port-scanner), modify the file `wan_config.xml` inside the folder `src/FastDDS-SH/examples/wan/` to match the IP address and port of with the WAN IP address and forwarded port of your environment.

Deployment

This examples launches a *ROS 2* `talker` in the *server* machine, and a *ROS 2* `listener` in the *client* machine. An *Integration Service* instance will communicate these two applications by translating the *types* and *topics* of *ROS 2* to those of *Fast DDS*, and then use the WAN-TCP communication capabilities of the latter to operate the tunneling.

To test it, open two terminals in each machine.

On the server side:

- In the first terminal, source the *ROS 2* installation and launch the *ROS 2* `talker` example:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 run demo_nodes_cpp talker
```

- In the second terminal, go to the `is-workspace` folder, source the *ROS 2*, *Fast DDS*, and local installations, and execute *Integration Service* with the `integration-service` command followed by the the server YAML configuration file located in the `src/Integration-Service/examples/wan_tunneling/ros2__wan_helloworld` folder:

```
cd ~/is-workspace
source /opt/ros/$<ROS2_DISTRO>/setup.bash
source install/setup.bash
integration-service src/Integration-Service/examples/wan_tunneling/ros2__wan_
↪helloworld/wan_server_talker.yaml
```

On the client side:

- In the first terminal, launch the *ROS 2* `listener` example:

```
source /opt/ros/$<ROS2_DISTRO>/setup.bash
ros2 run demo_nodes_cpp listener
```

- In the second terminal, go to the `is-workspace` folder, source the *ROS 2*, *Fast DDS*, and local installations, and execute *Integration Service* with the `integration-service` command followed by the the client YAML configuration file located in the `src/Integration-Service/examples/wan_tunneling/ros2__wan_helloworld` folder:

```
cd ~/dds-is-workspace
source /opt/ros/$<ROS2_DISTRO>/setup.bash
source install/setup.bash
integration-service src/Integration-Service/examples/wan_tunneling/ros2__wan_
↪helloworld/wan_client_listener.yaml
```

Once the two *Integration Service* instances match, the *ROS 2* `talker-listener` example will start to communicate.

Warning: If the example doesn't work, review carefully your NAT configuration.

6.21 Latest version

6.21.1 v3.1.0

This release includes the following *bugfixes* and *improvements*:

Common

- Added *new compilation flags* to compile each middleware examples independently.
- Document in more detail the *dependencies* required for the *Integration Service Core* and each *System Handle*.
- Fixed infinite loop problem produced when there are internal publishers and subscribers over the same topic.
- Avoid creation of empty folders when compiling using *colcon*.

ROS 1 System Handle

- Fixed bug producing a high CPU usage.

ROS 2 System Handle

- Cross-compatibility with *ROS 2 Galactic*.
- Added *new compilation flag* to select which ROS 2 version will be used.

WebSocket System Handle

- Handle properly *asio* and *websocketpp* exceptions.

6.22 Previous versions

6.22.1 v3.0.0

This release comprises a whole restructuring and renaming of the project, formerly known as SOSS, into **Integration Service**.

The *ROS 2* and *WebSocket System Handles*, which were previously included as subfolders of the main project, have been separated into independent repositories.

Important: **Previous** versions are considered **deprecated**, and it is not recommended to use them from now on.

There are some global changes that affect to all repositories; they are listed here:

- Completed migration from SOSS to **Integration Service**. This, code wise, included changing the **C++** namespace convention of the whole project, from `soss::core` to `eprosima::is::core` and from `soss::<SH_NAME>` to `eprosima::is::sh::<SH_NAME>`.
- Used *eProsima xTypes* as the common language for the *Core* to speak to each *System Handle*.
- Created unique pool for `SubscriptionCallback` and `RequestCallback` lambda functions. Prior to this version, they were being copied multiple among the *Core* and involved *System Handles*, leading to unnecessary copies and entities destruction problems.
- Added new *Logger* class, with different logging levels: *DEBUG*, *INFO*, *WARN*, *ERROR*.
- Debug logging traces are automatically enabled if the project is compiled in debug mode.
- Add much more traces and unify the logging style.
- Full *API reference* <*api_reference*> documentation using *Doxygen*.
- Migrated whole test suite from `Catch` to `Google Test`.
- Applied uncrustify rules.
- Added brand new *README* section for each repository, with detailed information about the project status and features.
- Added *global compilation flags* for building tests, examples, libraries and the API reference.

In relation to each repository, these are the most relevant changes for this major release:

Core

- Add an optional *YAML* configuration file *types* section, with an `idl` subsection. This allows users to introduce their own data type definitions at runtime, following the IDL specification.
- Inclusion of a *types-from* option in the *YAML* configuration file, to allow type inheritance among *System Handles*.
- Created a Core GitHub action automated task for unitary and integration tests.
- Skip blank services names.
- *JSON* conversion library: handle special double/float values (`Inf`, `NaN`...).
- *JSON* conversion library: boolean type support.
- *JSON* conversion library: sequences and arrays.
- Moved all example configuration files into a common folder.
- Created utility packages and applications for testing all the examples tutorials available in the documentation.
- Fix non-resizable containers for conversion to/from *ROS 1* and *ROS 2* static types definitions.

Fast DDS System Handle

- Migrated from *Fast RTPS* to *Fast DDS*.

  Important: From now on, only *2.X.X* versions of *Fast DDS* will be supported.

- Got rid of separate *CMake* testing project; now, it is included in the same project as the System Handle, under the `test` folder.

- Created a FastDDS-SH GitHub action automated task for unitary and integration tests.
- Added support for setting a custom *DDS* participant *domain ID*. This option can be set in the *YAML* specific configuration section for the `fastdds` middleware.
- Added the possibility of creating the *Fast DDS DomainParticipant* entity with a custom XML profile.
- Added *specific compilation flags*.
- *UDPv4* is now used as default transport if the user does not set a custom one.
- Treat services request and reply types properly, even if no remap if present.
- Fix concurrency problem detected in the client/server integration tests.

FIWARE System Handle

- Created a FIWARE-SH GitHub action automated task for unitary and integration tests.
- Added *specific compilation flags*.

ROS 1 System Handle

- Created a ROS1-SH GitHub action automated task for unitary and integration tests.
- Added *specific compilation flags*.
- Fixed `SystemHandle::configure` return value logic.
- Rearranged project folders with a more consistent structure.
- Created a new **MIX generator** project to ease users the task of compiling custom *ROS 1 packages* transformation libraries.
- Fixed service server/client not taking into account request and reply types separately.

ROS 2 System Handle

- Created a ROS2-SH GitHub action automated task for unitary and integration tests.
- Added *specific compilation flags*.
- Rearranged project folders to a more consistent structure.
- Created a new **MIX generator** project to ease users the task of compiling custom *ROS 2 packages* transformation libraries.
- Fixed service server/client not taking into account request and reply types separately.

WebSocket System Handle

- Created a WebSocket-SH GitHub action automated task for unitary and integration tests.
- Add *specific compilation flags*.
- Added support for *TCP* (non-secure) connections.
- Add the option in the *YAML* configuration file to *disable security*.
- Add tests for server/client interaction.
- Fixed concurrency problem when handling connections.
- Add specific traces with more information about the connection ID. Keep track of all the opened connections and tag them with a unique ID.

6.22.2 v2.1.0

This release includes the following *bugfixes* and *improvements*:

ROS 2 System Handle

- Cross-compatibility with *ROS 2 Eloquent* and *Foxy*.

WebSocket System Handle

- Verify token with public key.

6.22.3 v2.0.0

This release includes the following *bugfixes* and *improvements*:

Core

- Fix segfault when converting integer types to/from the *Core* language to the specific protocol.
- *JSON* conversion library: fix vector-of-bool conversions.
- *JSON* conversion library: allow more flexibility in converting JSON values into strings.
- Support for *ROS 1* boolean type being cast as uint8_t.
- Fix template substitution for string templates that have an ending substring.
- Support dispatch templates for topic names.

Fast DDS System Handle

- Added *Fast DDS System Handle*, with complete support for publication/subscription and services.

ROS 1 System Handle

- Avoid infinite waiting if a request do not reach the remote service.
- Fixed file separator trouble with topic names containing a `/`.
- Include the *Core* as a dependency of the *genmsg* package.
- Allow, via *YAML* configuration parameters, to specify custom names for *ROS 1* nodes.
- Remove *package.xml* files, because they confuse *colcon*.
- Added latching and queue_size specific topic configuration parameters.

ROS 2 System Handle

- Avoid infinite waiting if a request do not reach the remote service.
- Fixed file separator trouble with topic names containing a `/`.
- Include the *Core* as a dependency of the *rosidl* package.
- Support for both *ROS 2 Crystal* and *Dashing*.
- Have *CMake* throw a fatal error when dependencies are missing.
- Corrected format.
- Support for *ROS 2* domain change.

WebSocket System Handle

- Update *WebSocket System Handle* tests, so the TLS handshake works.

- Have *CMake* throw a fatal error when dependencies are missing.

6.22.4 v1.0.0

Initial version:

- Developed basic *Core* structure and support for *Middleware Interface Extension* files.
- Developed initial version of *ROS 1*, *ROS 2* and *WebSocket System Handles* with publisher/subscriber and server/client support; and *FIWARE System Handle*, with publisher/subscriber support.
- Created custom dynamic type `Message` class for handling content exchange among *System Handles*.
- Created *mock System Handle*, for testing purposes.
- Support for basic types: ints, floats, strings...

INDEX

C

E

I

L