xeus

Johan Mabille and Sylvain Corlay

Sep 22, 2023

INSTALLATION

C++ implementation of the Jupyter Kernel protocol

`xeus` is a framework meant to facilitate the implementation of kernels for Project Jupyter. It takes the burden of implementing the Jupyter Kernel protocol so developers can focus on implementing the interpreter part of the kernel.

CHAPTER

ONE

LICENSING

We use a shared copyright model that enables all contributors to maintain the copyright on their contributions.

This software is licensed under the BSD-3-Clause license. See the LICENSE file for details.

1.1 Installation

The framework is made of three libraries:

- `xeus`: The core implementation of the Jupyter protocol
- `xeus-zmq`: A library to author kernels that run in dedicated processes
- `xeus-lite`: A library to author kernels that run in the browser

Both `xeus-zmq` and `xeus-lite` depend on `xeus`.

1.1.1 Installing xeus-zmq

With Mamba or Conda

`xeus` and `xeus-zmq` have been packaged on all platforms for the mamba (or conda) package manager.

```
mamba install xeus-zmq -c conda-forge
```

From Source

`xeus` depends on the following libraries:

- xtl and **nolhmann_json_**

`xeus-zmq` depends on the following libraries:

- libzmq, cppzmq, OpenSSL and *xeus*

On linux platforms, `xeus` also requires libuuid, which is available in all linux distributions.

We have packaged all these dependencies on conda-forge. The simplest way to install them is to run:

```
mamba install cmake zeromq cppzmq OpenSSL xtl nlohmann_json -c conda-forge
```

On Linux platform, you will also need:

```
mamba install libuuid -c conda-forge
```

Once you have installed the dependencies, you can build and install *xeus*:

```
cmake -D CMAKE_BUILD_TYPE=Release .
make
make install
```

You can then build and install *xeus-zmq*:

```
cmake -D CMAKE_BUILD_TYPE=Release
make
make install
```

Installing the Dependencies from Source

The dependencies can also be installed from source. Simply clone the directories and run the following cmake and make instructions.

- libzmq

```
cmake -D WITH_PERF_TOOL=OFF -D ZMQ_BUILD_TESTS=OFF -D ENABLE_CPACK=OFF -D CMAKE_BUILD_
↪TYPE=Release .
make
make install
```

- cppzmq

cppzmq is a header only library:

```
cmake -D CMAKE_BUILD_TYPE=Release .
make install
```

- OpenSSL

OpenSSL has been packaged for most platforms and package manager. It should generally not be required for the user to build it.

- nlohmann_json

nlohmann_jon is a header only library:

```
cmake -DCMAKE_BUILD_TYPE=Release .
make install
```

- xtl

xtl is a header only library:

```
cmake -DCMAKE_BUILD_TYPE=Release .
make install
```

1.1.2 Installing xeus-lite

TODO

1.2 Usage

`xeus` enables custom kernel authors to implement Jupyter kernels more easily. It takes the burden of implementing the Jupyter Kernel protocol so developers can focus on implementing the interpreter part of the Kernel.

Note: In the following documentation:

- `interpreter` refers to the part of the kernel responsible for executing the code, completing the code etc.
- `client` refers to the Jupyter client, which can be Jupyter Notebook/JupyterLab/Jupyter console etc.
- `user` refers to people using the kernel on any Jupyter client.

The easiest way to get started with a new kernel is to create a class inheriting from the base interpreter class `xeus::xinterpreter` and implement the private virtual methods

- `execute_request_impl`: See execute_request Code execution request from the client.
- `complete_request_impl`: See complete_request Code completion request from the client.
- `inspect_request_impl`: See inspect_request Code inspection request (using a question mark on a type for example).
- `is_complete_request_impl`: See is_complete_request Called before code execution (terminal mode) in order to check if the code is complete and can be executed as it is (e.g. when typing a *for* loop on multiple lines in Python, code will be considered complete when the *for* loop has been closed).
- `kernel_info_request_impl`: See kernel_info_request Information request about the kernel: language name (for code highlighting), language version, terminal banner etc.
- `shutdown_request_impl`: Shutdown request from the client, this allows you to do some extra work before the kernel is shut down (e.g. free allocated memory).

A dummy kernel is provided as an example and a more advanced example kernel can be found here. You can also find real kernel implementations based on xeus:

- xeus-cling: C++ kernel
- xeus-python: Python kernel

1.3 Implementing a kernel

In most of the cases, the base kernel implementation is enough, and creating a kernel only means implementing the interpreter part.

The structure of your project should at least look like the following:

```
└── example/
    ├── src/
    │   ├── custom_interpreter.cpp
```

(continues on next page)

(continued from previous page)

```
    │   ├── custom_interpreter.hpp
    │   └── main.cpp
    ├── share/
    │   └── jupyter/
    │       └── kernels/
    │           └── my_kernel/
    │               └── kernel.json.in
    └── CMakeLists.txt
```

The xeus-cookiecutter project provides a template for a xeus-based kernel, and includes the base structure for a xeus-based kernel.

1.3.1 Implementing the interpreter

Let's start by editing the `custom_interpreter.hpp` file, it should contain the declaration of your interpreter class:

```
/***************************************************************************
* Copyright (c) 2016, Johan Mabille, Sylvain Corlay, Martin Renou          *
* Copyright (c) 2016, QuantStack                                           *
*                                                                          *
* Distributed under the terms of the BSD 3-Clause License.                 *
*                                                                          *
* The full license is in the file LICENSE, distributed with this software. *
****************************************************************************/

#ifndef CUSTOM_INTERPRETER
#define CUSTOM_INTERPRETER

#include "xeus/xinterpreter.hpp"

#include "nlohmann/json.hpp"

using xeus::xinterpreter;

namespace nl = nlohmann;

namespace custom
{
    class custom_interpreter : public xinterpreter
    {
    public:

        custom_interpreter() = default;
        virtual ~custom_interpreter() = default;

    private:

        void configure_impl() override;

        nl::json execute_request_impl(int execution_counter,
                                      const std::string& code,
```

(continues on next page)

(continued from previous page)

```
                                            bool silent,
                                            bool store_history,
                                            nl::json user_expressions,
                                            bool allow_stdin) override;

        nl::json complete_request_impl(const std::string& code,
                                             int cursor_pos) override;

        nl::json inspect_request_impl(const std::string& code,
                                            int cursor_pos,
                                            int detail_level) override;

        nl::json is_complete_request_impl(const std::string& code) override;

        nl::json kernel_info_request_impl() override;

        void shutdown_request_impl() override;
    };
}

#endif
```

Note: Almost all custom_interpreter methods return a nl::json instance. This is actually using nlohmann json which is a modern C++ implementation of a JSON datastructure.

In the following sessions we will see details about each one of the methods that need to be implemented in order to have a functional kernel. The user can opt for using the reply API that will appropriately create replies to send to the kernel, or create the replies themselves.

Code Execution

You can implement all the methods described here in the custom_interpreter.cpp file. The main method is of course the execute_request_impl which executes the code whenever the client is sending an execute request.

```
nl::json custom_interpreter::execute_request_impl(int execution_counter, // Typically␣
↪the cell number
                                                  const std::string& /*code*/, // Code␣
↪to execute
                                                  bool /*silent*/,
                                                  bool /*store_history*/,
                                                  nl::json /*user_expressions*/,
                                                  bool /*allow_stdin*/)
{
    // You can use the C-API of your target language for executing the code,
    // e.g. `PyRun_String` for the Python C-API
    //      `luaL_dostring` for the Lua C-API

    // Use this method for publishing the execution result to the client,
    // this method takes the ``execution_counter`` as first argument,
    // the data to publish (mime type data) as second argument and metadata
```

(continues on next page)

(continued from previous page)

```
    // as third argument.
    // Replace "Hello World !!" by what you want to be displayed under the execution cell
    nl::json pub_data;
    pub_data["text/plain"] = "Hello World !!";
    publish_execution_result(execution_counter, std::move(pub_data), nl::json::object());

    // You can also use this method for publishing errors to the client, if the code
    // failed to execute
    // publish_execution_error(error_name, error_value, error_traceback);
    publish_execution_error("TypeError", "123", {"!@#$", "*(*"});

    return xeus::create_successful_reply();
}
```

The result and arguments of the execution request are described in the execute_request documentation.

Note: The other methods are all optional, but we encourage you to implement them in order to have a fully-featured kernel.

Within this method the use of create_error_reply and create_successful_reply might be useful.

Input request

For input request support, you would need to monkey-patch the language functions that prompt for a user input (`input` and `raw_input` in Python, `io.read` in Lua etc) and call `xeus::blocking_input_request` instead. The second parameter should be set to False if what the user is typing should not be visible on the screen.

```
#include "xeus/xinput.hpp"

xeus::blocking_input_request("User name:", true);
xeus::blocking_input_request("Password:", false);
```

Configuration

The `configure_impl` method allows you to perform some operations after the `custom_interpreter` creation and before executing any request. This is optional, but it can be useful, for example it is used in xeus-python for initializing the auto-completion engine.

```
void custom_interpreter::configure_impl()
{
    // Perform some operations
}
```

Code Completion

The `complete_request_impl` method allows you to implement the auto-completion logic for your kernel.

```
nl::json custom_interpreter::complete_request_impl(const std::string& code,
                                                    int cursor_pos)
{
    // Code starts with 'H', it could be the following completion
    if (code[0] == 'H')
    {
        return xeus::create_complete_reply({"Hello", "Hey", "Howdy"}, 5, cursor_pos);
    }
    // No completion result
    else
    {
        return xeus::create_complete_reply({}, cursor_pos, cursor_pos);
    }
}
```

The result and arguments of the completion request are described in the complete_request documentation.

Code Inspection

Allows the kernel user to inspect a variable/class/type in the code. It takes the code and the cursor position as arguments, it is up to the kernel author to extract the token at the given cursor position in the code in order to know for which name the user wants inspection.

```
nl::json custom_interpreter::inspect_request_impl(const std::string& code,
                                                   int /*cursor_pos*/,
                                                   int /*detail_level*/)
{
    nl::json result;

    if (code.compare("print") == 0)
    {
        return xeus::create_inspect_reply(true,
                                          {"text/plain", "Print objects to the text␣
↪stream file, [...]"});
    }
    else
    {
        return xeus::create_inspect_reply();
    }
}
```

The result and arguments of the inspection request are described in the inspect_request documentation and the **create_inspect_reply_** might be useful to create a reply within specifications.

Code Completeness

This request is never called from the Notebook or from JupyterLab clients, but it is called from the Jupyter console client. It allows the client to know if the user finished typing his code, before sending an execute request. For example, in Python, the following code is not considered as complete:

```
def foo:
```

So the kernel should return "incomplete" with an indentation value of 4 for the next line.

The following code is considered as complete:

```
def foo:
    print("bar")
```

So the kernel should return "complete".

```
nl::json custom_interpreter::is_complete_request_impl(const std::string& /*code*/)
{
    return xeus::create_is_complete_reply("complete");
}
```

The result and arguments of the completness request are described in the is_complete_request documentation. Both **create_default_complete_reply_** and **create_is_complete_reply_** methods are recommended.

Kernel info

This request allows the client to get information about the kernel: language, language version, kernel version, etc.

```
nl::json custom_interpreter::kernel_info_request_impl()
{
    return xeus::create_info_reply("",
                                   "my_kernel",
                                   "0.1.0",
                                   "python",
                                   "3.7",
                                   "text/x-python",
                                   ".py");
}
```

The result and arguments of the kernel info request are described in the kernel_info_request documentation. The **create_info_reply_** method will help you to provide complete information about your kernel.

Kernel shutdown

This allows you to perform some operations before shutting down the kernel.

```
void custom_interpreter::shutdown_request_impl()
{
    std::cout << "Bye!!" << std::endl;
}
```

1.3.2 Kernel replies

Error reply

Creates a default error reply to the kernel or allows custom input. The signature of the method is the following:

Where `evalue` is exception value, `ename` is exception name and `trace_back` a vector of strings with the exception stack.

Successful reply

Creates a default success reply to the kernel or allows custom input. The signature of the method is the following:

Where `payload` is a way to trigger frontend actions from the kernel (payloads are deprecated but since there are still no replecement for it you might need to use it). You can find more information about the different kinds of payloads in the official documentation. `data` is a dictionary which the keys is a `MIME_type` (this is the type of data to be shown it must be a valid MIME type, for a list of the possibilities check MDN, note that you're not limited by these types) and the values are the content of the information intended to be displayed in the frontend. And `user_expressions` is a dictionary of strings of arbitrary code, more information about it on the official documentation.

Complete reply

Creates a custom completion reply to the kernel. The signature of the method is the following:

Where `matches` the list of all matches to the completion request, it's a mandatory argument. `cursor_start` and `cursor_end` mark the range of text that should be replaced by the above matches when a completion is accepted, typically `cursor_end` is the same as `cursor_pos` in the request and both these arguments are mandatory for the implementation of the method. `metadata` a dictionary of strings that contains information that frontend plugins might use for extra display information about completions.

In case you do not wish to implement completion in your kernel, instead of creating a complete reply you can use the `create_successful_reply` with its default arguments.

Is complete reply

Creates a default is complete reply to the kernel or allows custom input. The signature of the method is the following:

`status` one of the following 'complete', 'incomplete', 'invalid', 'unknown'. `indent` if status is 'incomplete', indent should contain the characters to use to indent the next line. This is only a hint: frontends may ignore it and use their own autoindentation rules. For other statuses, this field does not exist.

Create info reply

Thorough information about the kernel's infos variables can be found in the Jupyter kernel docs.

1.3.3 Implementing the main entry

Now let's edit the `main.cpp` file which is the main entry for the kernel executable.

```cpp
/***************************************************************************
* Copyright (c) 2016, Johan Mabille, Sylvain Corlay, Martin Renou          *
* Copyright (c) 2016, QuantStack                                           *
*                                                                          *
* Distributed under the terms of the BSD 3-Clause License.                 *
*                                                                          *
* The full license is in the file LICENSE, distributed with this software. *
****************************************************************************/

#include <memory>

#include "xeus/xkernel.hpp"
#include "xeus/xkernel_configuration.hpp"
#include "xeus-zmq/xserver_zmq.hpp"

#include "custom_interpreter.hpp"

int main(int argc, char* argv[])
{
    // Load configuration file
    std::string file_name = (argc == 1) ? "connection.json" : argv[2];
    xeus::xconfiguration config = xeus::load_configuration(file_name);

    auto context = xeus::make_context<zmq::context_t>();

    // Create interpreter instance
    using interpreter_ptr = std::unique_ptr<custom::custom_interpreter>;
    interpreter_ptr interpreter = interpreter_ptr(new custom::custom_interpreter());

    // Create kernel instance and start it
    xeus::xkernel kernel(config, xeus::get_user_name(), std::move(context),
  std::move(interpreter), xeus::make_xserver_zmq);
    kernel.start();

    return 0;
}
```

1.3.4 Kernel file

The `kernel.json` file is a `json` file used by Jupyter in order to retrieve all the available kernels.

It must be installed in the `INSTALL_PREFIX/share/jupyter/kernels/my_kernel` directory, we will see how to do that in the next chapter.

This `json` file contains:

- `display_name`: the name that the Jupyter client should display in its interface (e.g. on the main JupyterLab page).
- `argv`: the command that the Jupyter client needs to run in order to start the kernel. You should leave this value unchanged if you are not sure what you are doing.

- `language`: the target language of your kernel.

You can edit the `kernel.json.in` file as following. This file will be used by cmake for generating the actual `kernel.json` file which will be installed.

```
{
    "display_name": "my_kernel",
    "argv": [
        "@CMAKE_INSTALL_PREFIX@/@CMAKE_INSTALL_BINDIR@/@EXECUTABLE_NAME@",
        "-f",
        "{connection_file}"
    ],
    "language": "python"
}
```

Note: You can provide logos that will be used by the Jupyter client. Those logos should be in files named `logo-32x32.png` and `logo-64x64.png` (`32x32` and `64x64` being the size of the image in pixels), they should be placed next to the `kernel.json.in` file.

1.3.5 Compiling and installing the kernel

Your `CMakeLists.txt` file should look like the following:

```
############################################################################
# Copyright (c) 2016, Johan Mabille, Sylvain Corlay, Martin Renou          #
# Copyright (c) 2016, QuantStack                                           #
#                                                                          #
# Distributed under the terms of the BSD 3-Clause License.                 #
#                                                                          #
# The full license is in the file LICENSE, distributed with this software. #
############################################################################

cmake_minimum_required(VERSION 3.4.3)
project(my_kernel)

set(EXECUTABLE_NAME my_kernel)

# Configuration
# =============

include(GNUInstallDirs)

# We generate the kernel.json file, given the installation prefix and the executable name
configure_file (
    "${CMAKE_CURRENT_SOURCE_DIR}/share/jupyter/kernels/my_kernel/kernel.json.in"
    "${CMAKE_CURRENT_SOURCE_DIR}/share/jupyter/kernels/my_kernel/kernel.json"
)

option(XEUS_STATIC_DEPENDENCIES "link statically with xeus dependencies" OFF)
if (XEUS_STATIC_DEPENDENCIES)
    set(xeus-zmq_target "xeus-zmq-static")
```

(continues on next page)

(continued from previous page)

```
else ()
    set(xeus-zmq_target "xeus-zmq")
endif ()

# Dependencies
# ============

# Be sure to use recent versions
set(xeus-zmq_REQUIRED_VERSION 1.0.2)

find_package(xeus-zmq ${xeus-zmq_REQUIRED_VERSION} REQUIRED)
find_package(Threads)

# Flags
# =====

include(CheckCXXCompilerFlag)

if (CMAKE_CXX_COMPILER_ID MATCHES "Clang" OR CMAKE_CXX_COMPILER_ID MATCHES "GNU" OR␣
↪CMAKE_CXX_COMPILER_ID MATCHES "Intel")
    CHECK_CXX_COMPILER_FLAG("-std=c++14" HAS_CPP14_FLAG)

    if (HAS_CPP14_FLAG)
        set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14")
    else()
        message(FATAL_ERROR "Unsupported compiler -- xeus requires C++14 support!")
    endif()
endif()

# Target and link
# ===============

# my_kernel source files
set(MY_KERNEL_SRC
    src/custom_interpreter.cpp
    src/custom_interpreter.hpp
)

# My kernel executable
add_executable(${EXECUTABLE_NAME} src/main.cpp ${MY_KERNEL_SRC} )
target_link_libraries(${EXECUTABLE_NAME} PRIVATE ${xeus-zmq_target} Threads::Threads)

set_target_properties(${EXECUTABLE_NAME} PROPERTIES
    INSTALL_RPATH_USE_LINK_PATH TRUE
)

# Installation
# ============

# Install my_kernel
install(TARGETS ${EXECUTABLE_NAME}
        RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR})
```

(continues on next page)

(continued from previous page)

```
# Configuration and data directories for jupyter and my_kernel
set(XJUPYTER_DATA_DIR "share/jupyter" CACHE STRING "Jupyter data directory")

# Install Jupyter kernelspecs
set(MY_KERNELSPEC_DIR ${CMAKE_CURRENT_SOURCE_DIR}/share/jupyter/kernels)
install(DIRECTORY ${MY_KERNELSPEC_DIR}
        DESTINATION ${XJUPYTER_DATA_DIR}
        PATTERN "*.in" EXCLUDE)
```

Now you should be able to install your new kernel and use it with any Jupyter client.

For the installation you first need to install dependencies, the easier way is using `conda`:

```
conda install -c conda-forge cmake jupyter xeus xtl nlohmann_json cppzmq
```

Then create a `build` folder in the repository and build the kernel from there:

```
mkdir build
cd build
cmake -D CMAKE_INSTALL_PREFIX=$CONDA_PREFIX ..
make
make install
```

That's it! Now if you run the Jupyter Notebook interface you should be able to create a new Notebook selecting the `my_kernel` kernel. Congrats!

1.3.6 Writing unit-tests for your kernel

For writing unit-tests for you kernel, you can use the jupyter_kernel_test Python library. It allows you to test the results of the requests you send to the kernel.

1.4 Customizing the kernel

While it is possible to create a kernel by focusing on the implementation of the interpreter, `xeus` also offers the possibility to customize some predefined behaviors. This can be done via additional arguments of the `xkernel` consturctor:

```
xkernel(const xconfiguration& config,
        const std::string& user_name,
        interpreter_ptr interpreter,
        history_manager_ptr history_manager = make_in_memory_history_manager(),
        logger_ptr logger = nullptr,
        server_builder sbuilder = make_xserver,
        debugger_builder dbuilder = make_null_debugger);
```

1.4.1 History manager

The `xhistory_manager` class is used to store the `execute_request` messages sent by the frontend. Typical usage is when the console client connects to a kernel that has already executed some code: it asks the `history_manager` for its records and prints them so that the user knows what happened before.

`xeus` provides a default implementation of `xhistory_manager` that stores the messages in memory. It is possible to provide a different history manager by defining a class that inherits from `xhistory_manager` and implements the abstract methods:

```
class file_history_manager : public xeus::xhistory_manager
{
public:

    file_history_manager(const std::string& file_name);
    virtual ~file_history_manager();

private:


    void configure_impl() override;
    void store_inputs_impl(int line_num, const std::string& input) override;
    nl::json get_tail_impl(int n, bool raw, bool output) const override;
    nl::json get_range_impl(int session, int start, int stop, bool raw, bool output)␣
↪const override;
    nl::json search_impl(const std::string& pattern, bool raw, bool output, int n, bool␣
↪unique) const override;
};
```

Then simply pass an instance to the kernel constructor:

```
int main(int argc, char* argv[])
{
    // ....
    // Instantiates interpreter and config
    // ....
    auto hist = std::make_unique<file_history_manager>("my_history_file.txt");
    xeus::xkernel kernel(config,
                         xeus::get_user_name(),
                         interpreter,
                         std::move(hist));
    kernel.start();
    return 0;
}
```

1.4.2 Logger

xeus does not log anything by default. However, it can be useful during the development phase of a new kernel to print the messages that are received by and sent from the kernel. Having a logger that can be enabled on-demand is also useful to track bugs once your new kernel has been released.

xeus provides a flexible logging mechanism that can be easily extended. Two default loggers are available: one that logs to the console, and another one that logs to files. You can add your own by defining a class that inherit from xlogger. Three logging levels are provided, one for message type, one for the content of the message and one for the full message. Loggers can be chained, meaning you can log the message types to the console and the full messages into files:

```
int main(int argc, char* argv[])
{
    // ....
    // Instantiates interpreter and config
    // ....
    auto logger = xeus::make_console_logger(xeus::xlogger::msg_type,
                                            xeus::make_file_logger(xeus::xlogger::full,
↪"my_log_file.log"));
    xeus::xkernel kernel(config,
                         xeus::get_user_name(),
                         interpreter,
                         xeus::make_in_memory_history_manager(),
                         std::move(logger));
    kernel.start();
    return 0;

}
```

To turn on logging, you need to define the variable environment XEUS_LOG before starting the kernel. This way, enabling and disabling the logs do not require to rebuild the kernel.

Defining a new type of logger is as simple as defining a new type of history manager: inherit from xlogger and implement the abstract methods:

```
class my_logger : public xeus::xlogger
{
public:

    my_logger();
    virtual ~mylogger();

private:

    void log_received_message_impl(const xmessage& message, channel c) const override;
    void log_sent_message_impl(const xmessage& message, channel c) const override;
    void log_iopub_message_impl(const xpub_message& message) const override;

    void log_message_impl(const std::string& socket_info,
                          const nl::json& header,
                          const nl::json& parent_header,
                          const nl::json& metadata,
                          const nl::json& content) const override;
};
```

1.4.3 Server

The server is the middleware component responsible for sending and receiving the messages. While you will hardly have to implement your own, you might need to specify a different server that the default one. The core library `xeus` only provides the interface for the server, implementations are provided by `xeus-zmq` and `xeus-lite`.

`xeus-zmq` actually provides three different implementations for the server:

- `xserver_zmq` is the default server implementation, it runs three thread, one for publishing, one for the heartbeat messages, and the main thread handles the shell, control and stdin sockets.
- `xserver_control_main` runs an additional thread for handling the shell and the stdin sockets. Therefore the main thread only listens to the control socket. This allow to easily implement interruption of code execution. This server is required if you want to plug a debugger in the kernel.
- `xserver_shell_main` is similar to `xserver_control_main` except that the main thread handles the shell and the stdin sockets while the additional thread listens to the control socket. This server is required if you want to plug a debugger that does not support native threads and requires the code to be run by the main thread.

`xeus-lite` provides a default implementation only.

1.5 Internals of xeus

xeus is internally architected around four main components:

- The *server* is the middleware component responsible for receiving and sending messages to the Jupyter client and for handling the concurrency model of the application. *xeus* provides two implementations of the *server* component, one built upon *ZeroMQ* and one built for in-memory communication in the browser.
- The kernel core routes the messages to the appropriate method of the *interpreter* and does some book-keeping operations such as storing the message and its answer in the history manager, or sending relevant messages to the server.
- The *interpreter* provides the interface that kernel authors must implement.
- The *debugger* provides the interface for implementing a debugger if the kernel supports debugging.

The *interpreter*, the *server* and the *debugger* are well isolated from each other, only the kernel core can interact with them. The kernel core is also loosely coupled with the server, which makes it easy to replace the server implementation provided by *xeus* with a custom one.

xeus uses the ZeroMQ library which provides the low-level transport layer over which the messages are sent. Before reading more, it is best to familiarize yourself with the concepts of ZeroMQ.

1.6 Server

1.6.1 Public API

The server part of *xeus* provides a public API made in `xserver.hpp`. This file contains the base class `xserver`, which must be inherited from any class implementing a server. This is the unique entry point into the server component used by the kernel core.

xeus-zmq provides the following implementations:

- `xserver_zmq.hpp`: This file contains the interface of the default server implementation, that can be used directly or extended in order to override parts of its behavior.
- `xserver_control_main.hpp`: This file contains the interface of a server that handles the shell and the control socket on different threads. The main thread listens to the control socket.
- `xserver_shell_main.hpp`: This file contains the interface of a server that handlels the shell and the control sockets on different threads. The main threads listens to the shell socket.

Before we dive into the details of the server implementation, let's have a look at the public interface:

```
enum class channel
{
    SHELL,
    CONTROL
};

class XEUS_API xserver
{
public:

    using listener = std::function<void(xmessage)>;
    using internal_listener = std::function<nl::json(nl::json)>;

    virtual ~xserver() = default;

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

    xserver(xserver&&) = delete;
    xserver& operator=(xserver&&) = delete;

    xcontrol_messenger& get_control_messenger();

    void send_shell(xmessage message);
    void send_control(xmessage message);
    void send_stdin(xmessage message);
    void publish(xpub_message message, channel c);

    void start(xpub_message message);
    void abort_queue(const listener& l, long polling_interval);
    void stop();
    void update_config(xconfiguration& config) const;

    void register_shell_listener(const listener& l);
    void register_control_listener(const listener& l);
    void register_stdin_listener(const listener& l);
    void register_internal_listener(const internal_listener& l);

protected:

    xserver() = default;

    void notify_shell_listener(xmessage msg);
    void notify_control_listener(xmessage msg);
```

(continues on next page)

(continued from previous page)

```
    void notify_stdin_listener(xmessage msg);
    nl::json notify_internal_listener(nl::json msg);

private:

    virtual xcontrol_messenger& get_control_messenger_impl() = 0;

    virtual void send_shell_impl(xmessage message) = 0;
```

First thing to notice is the `xserver` class makes use of the Non-Virtual Interface pattern. This allows a clear separation between the client interface (the public methods) and the interface for subclasses (protected non-virtual methods and private virtual methods).

The client interface can be divided into three parts:

- the API to control the server: this is how you configure, start and stop the server. The related methods are `update_config`, `start`, `stop` and `abort_queue`. These methods forward to private pure virtual methods that must be implemented in inheriting classes.
- the API to send message: this is where you decide on which channel you send the message. The related methods are `send_shell`, `send_control`, `send_stdin` and `publish`. These methods also forward to virtual methods that must be implemented in inheriting classes.
- the API to register callbacks: the methods `register_shell_listener`, `register_control_listener` and `register_stdin_listener` allow clients (such as the kernel core component) to register functions that will be called when a message is received by the server. This way, the server component is loosely coupled with its clients, it doesn't need to know anything about them.

The subclass interface contains private virtual methods that must be implemented in inheriting classes to define the behavior of the server, and protected methods to notify the client that a message has been received. This makes inheriting classes independent from the way the `xserver` class stores and uses the callbacks.

1.6.2 xserver_zmq

The `xserver_zmq` class is the default implementation of the server API, its internals are illustrated in the following diagram:

The default server is made of three threads communicating through internal *ZeroMQ* sockets. The main thread is responsible for polling both `shell` and `controller` channels. When a message is received on one of these channels, the corresponding callback is invoked. Any code executed in the interpreter will be executed by the main thread. If the `publish` method is called, the main thread sends a message to the publisher thread.

Having a dedicated thread for publishing messages makes this operation a non-blocking one. When the kernel main thread needs to publish a message, it simply sends it to the publisher thread through an internal socket and continues its execution. The publisher thread will poll its internal socket and forward the messages to the `publisher` channel.

The last thread is the heartbeat. It is responsible for notifying the client that the kernel is still alive. This is done by sending messages on the `heartbeat` channel at a regular rate.

The main thread is also connected to the publisher and the heartbeat threads through internal `controller` channels. These are used to send `stop` messages to the subthread and allow to stop the kernel in a clean way.

1.6.3 xserver_control_main

The `xserver_control_main` class is an alternative implementation of the server API, its internals are illustrated in the following diagram:

This server runs four threads that communicate through internal *ZeroMQ* sockets. The main thread is responsible for polling the `control` channel while a dedicated thread listens on the `shell` channel. Having separated threads for the `control` and `shell` channel makes it possible to send messages on a channel while the kernel is already processing a message on the other channel. For instance one can send on the `control` a request to interrupt a computation running on the `shell`.

The control thread is also connected to the shell, the publisher and the heartbeat threads through internal `controller` channels. These are used to send `stop` messages to the subthread and allow to stop the kernel in a clean way, similarly to the `xserver_zmq`.

The rest of the implementation is similar to that of `xserver_zmq`.

1.6.4 xserver_shell_main

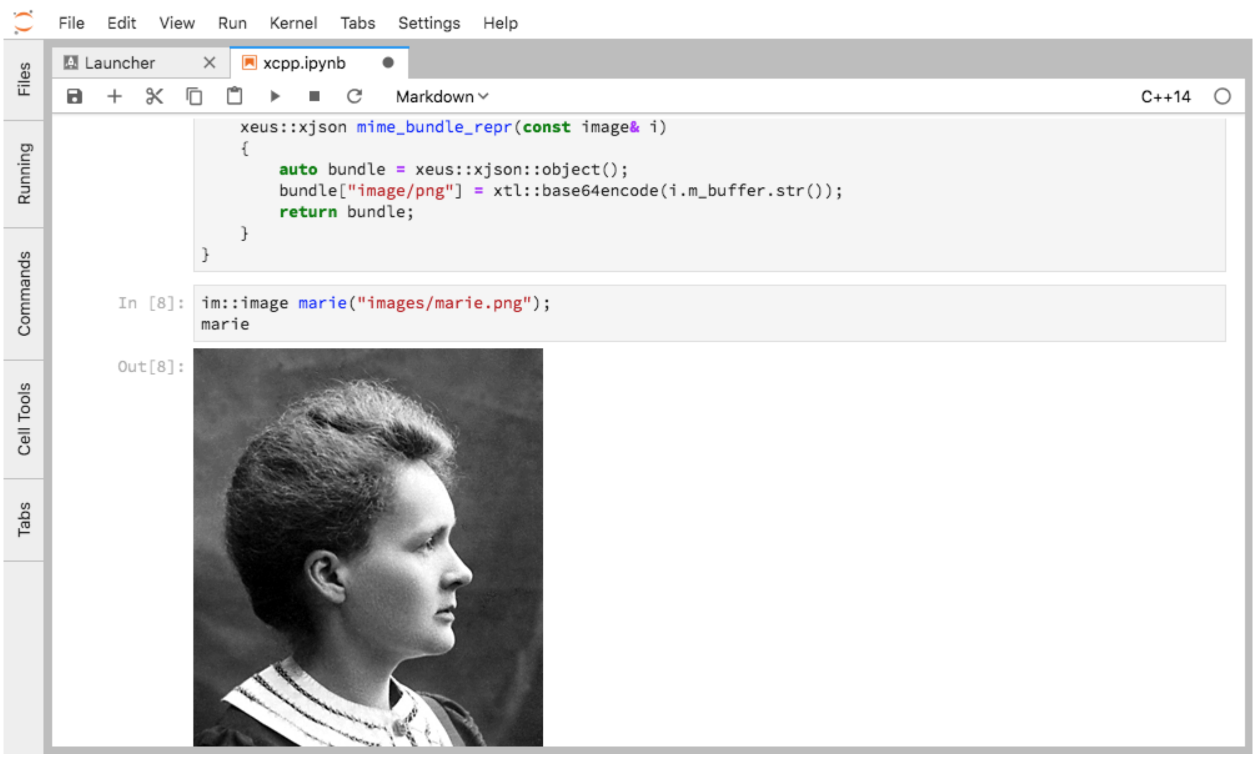
The `xserver_shell_main` class is very similar to the `xserver_control_main` class, except that the main thread listens on the `shell` channel as illustrated in the following diagram:

1.7 Related projects

1.7.1 Xeus-based kernels

xeus-cling

The xeus-cling project is a Jupyter kernel for the C++ programming language based on the Cling C++ interpreter from CERN and Xeus, the native implementation of the Jupyter protocol.



xeus-python

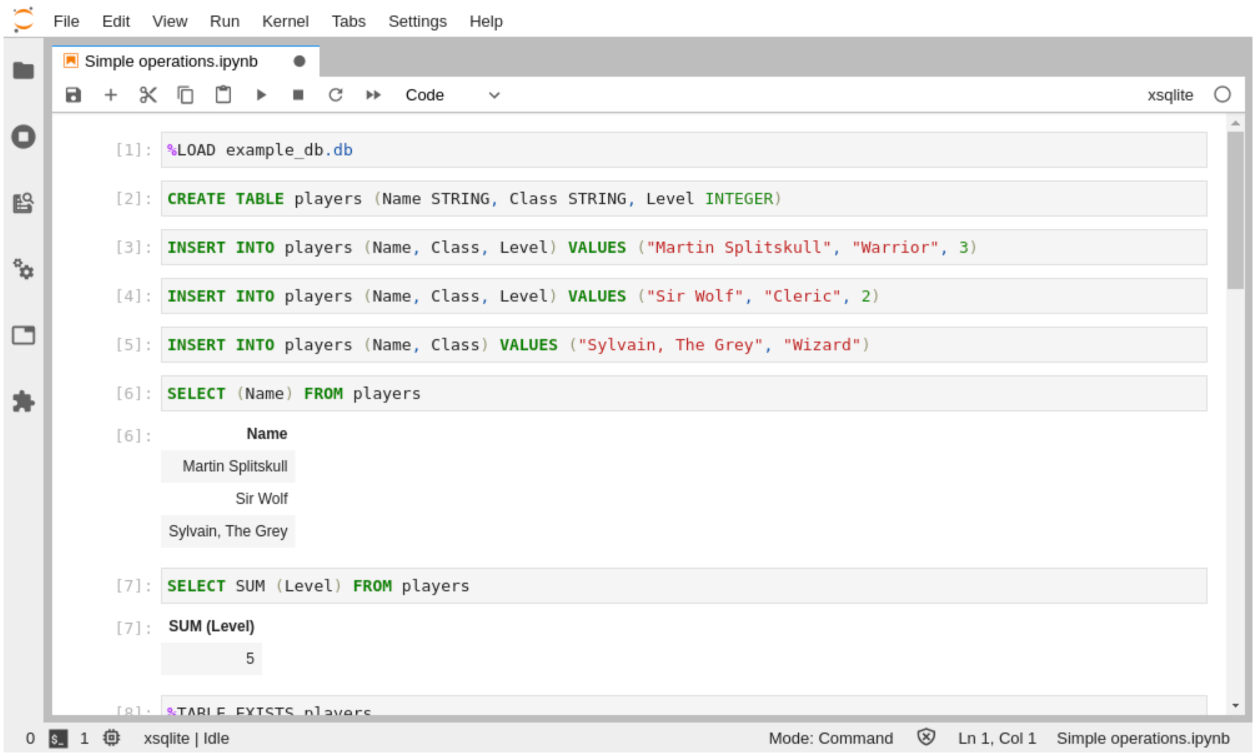
The xeus-python project is a Jupyter kernel for the Python programming language based on the Xeus implementation of the protocol.

xeus-sql

The xeus-sql is a Jupyter kernel for general SQL implementations based on the native implementation of the Jupyter protocol xeus and SOCI.

xeus-sqlite

The xeus-sqlite project is a Jupyter kernel for the SQLite.



xeus-robot

The xeus-robot project is a Jupyter kernel for RobotFramework

xeus-lua

The xeus-lua project is a Jupyter kernel for Lua.

xeus-octave

The xeus-octave project is a Jupyter kernel for GNU Octave.

xeus-wren

The xeus-wren project is a Jupyter kernel for Wren.

1.7.2 xeus-cookiecutter

The xeus-cookiecutter project can be used to generated xeus-based Jupyter kernels. It provides a base template project including an "hello world" kernel.

1.7.3 SlicerJupyter

The SlicerJupyter project is an integration of the xeus-python kernel in the 3D-Slicer desktop application. SlicerJupyter is an example of application taking advantage of xeus to integrate a Jupyter kernel in the event loop of a Desktop application.

1.7.4 xwidgets

The xwidgets project is a C++ implementation of the Jupyter interactive widgets protocol. The Python reference implementation is available in the ipywidgets ipywidgets project.

xwidgets enables the use of the Jupyter interactive widgets in the C++ notebook, powered by the xeus-cling kernel and the cling C++ interpreter from CERN. xwidgets can also be used to create applications making use of the Jupyter interactive widgets without the C++ kernel per se.