xeus

Johan Mabille and Sylvain Corlay

May 27, 2020

INSTALLATION

C++ implementation of the Jupyter Kernel protocol

`xeus` is a library 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

1.1.1 With Conda

`xeus` has been packaged on all platforms for the conda package manager.

```
conda install xeus -c conda-forge
```

1.1.2 From Source

`xeus` depends on the following libraries:

- libzmq, cppzmq, OpenSSL and xtl

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

We have packaged all these dependencies for the conda package manager. The simplest way to install them with conda is to run:

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

On Linux platform, you will also need:

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

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

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

If you need the `xeus` library only, you can omit the `BUILD_EXAMPLES` settings.

1.1.3 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.

xtl

xtl is a header only library:

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

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

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. *
****************************************************************************/
```

(continues on next page)

(continued from previous page)

```cpp
#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,
                                      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.

Code Execution

Then, you would need to implement all of those methods one by one 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
    // 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", {"!@#$", "*(*"});

    nl::json result;
    result["status"] = "ok";
    return result;
}
```

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.

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 is 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)
{
    nl::json result;

    // Code starts with 'H', it could be the following completion
    if (code[0] == 'H')
    {
        result["status"] = "ok";
        result["matches"] = {"Hello", "Hey", "Howdy"};
        result["cursor_start"] = 5;
        result["cursor_end"] = cursor_pos;
    }
    // No completion result
    else
    {
        result["status"] = "ok";
        result["matches"] = nl::json::array();
        result["cursor_start"] = cursor_pos;
        result["cursor_end"] = cursor_pos;
    }

    return result;
}
```

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)
    {
        result["found"] = true;
        result["text/plain"] = "Print objects to the text stream file, [...]";
    }
    else
    {
        result["found"] = false;
    }

    result["status"] = "ok";
    return result;
}
```

The result and arguments of the inspection request are described in the inspect_request documentation.

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*/)
{
    nl::json result;

    // if (is_complete(code))
    // {
        result["status"] = "complete";
    // }
    // else
    // {
    //    result["status"] = "incomplete";
    //    result["indent"] = 4;
```

(continues on next page)

(continued from previous page)

```
    //}

    return result;
}
```

The result and arguments of the completness request are described in the is_complete_request documentation.

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()
{
    nl::json result;
    result["implementation"] = "my_kernel";
    result["implementation_version"] = "0.1.0";
    result["language_info"]["name"] = "python";
    result["language_info"]["version"] = "3.7";
    result["language_info"]["mimetype"] = "text/x-python";
    result["language_info"]["file_extension"] = ".py";
    return result;
}
```

The result and arguments of the kernel info request are described in the kernel_info_request documentation.

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 Implementing the main entry

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

```
/***************************************************************************
* 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 "custom_interpreter.hpp"
```

(continues on next page)

(continued from previous page)

```
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);

    // 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(interpreter));
    kernel.start();

    return 0;
}
```

1.3.3 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.

1.3.4 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_target "xeus-static")
else ()
    set(xeus_target "xeus")
endif ()

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

# Be sure to use recent versions
set(xeus_REQUIRED_VERSION 0.19.1)
set(cppzmq_REQUIRED_VERSION 4.3.0)

find_package(xeus ${xeus_REQUIRED_VERSION} REQUIRED)
find_package(cppzmq ${cppzmq_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)
```

(continues on next page)

(continued from previous page)

```
        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_target} Threads::Threads)

if (APPLE)
    set_target_properties(${EXECUTABLE_NAME} PROPERTIES
        MACOSX_RPATH ON
    )
else()
    set_target_properties(${EXECUTABLE_NAME} PROPERTIES
        BUILD_WITH_INSTALL_RPATH 1
        SKIP_BUILD_RPATH FALSE
    )
endif()

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})

# 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!

1.3.5 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;
```

(continues on next page)

(continued from previous page)

```
    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. `xeus` provides two types of server:

- `xserver_zmq` is the default server implementation, it runs three thread, one for publishing, one for the heart-beat 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.

1.5 Internals of xeus

xeus is internally architected around three main components:

- The *server* is the middleware component responsible for receiving and sending messages to the Jupyter client. It is built upon *ZeroMQ* and handles the concurrency model of the application.
- 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 *interpreter* and the *server* 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 of:

- `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.
- `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:

```
lude "xcontrol_messenger.hpp"

space xeus

enum class channel
{
    SHELL,
    CONTROL
};

class XEUS_API xserver
{
public:

    using listener = std::function<void(zmq::multipart_t&)>;
    using internal_listener = std::function<zmq::multipart_t(zmq::multipart_t&)>;

    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(zmq::multipart_t& message);
    void send_control(zmq::multipart_t& message);
    void send_stdin(zmq::multipart_t& message);
    void publish(zmq::multipart_t& message, channel c);

    void start(zmq::multipart_t& message);
    void abort_queue(const listener& l, long polling_interval);
```

(continues on next page)

(continued from previous page)

```
    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(zmq::multipart_t& message);
    void notify_control_listener(zmq::multipart_t& message);
    void notify_stdin_listener(zmq::multipart_t& message);
    zmq::multipart_t notify_internal_listener(zmq::multipart_t& message);

private:
```

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`.

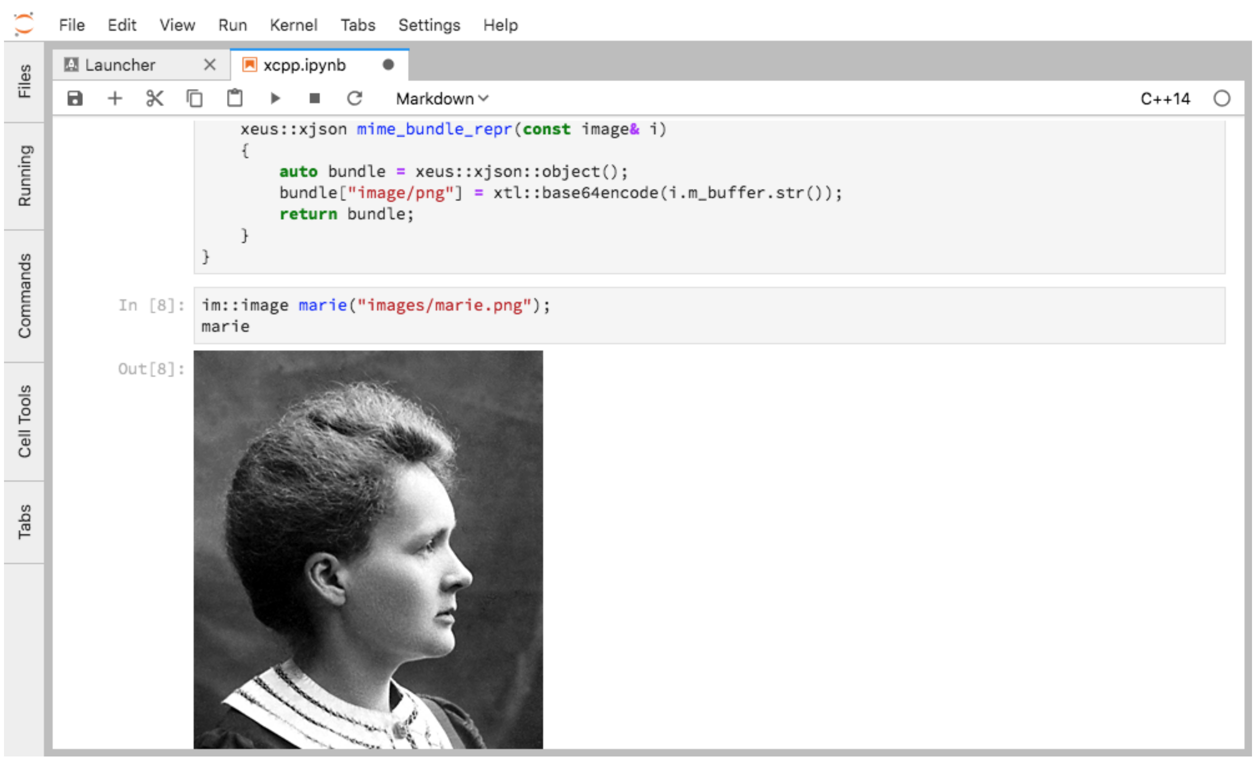
1.6.4 xserver_shell_main

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-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::xjson mime_bundle_repr(const image& i)
    {
        auto bundle = xeus::xjson::object();
        bundle["image/png"] = xtl::base64encode(i.m_buffer.str());
        return bundle;
    }
}
```

```
im::image marie("images/marie.png");
marie
```

1.7.2 xeus-python

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

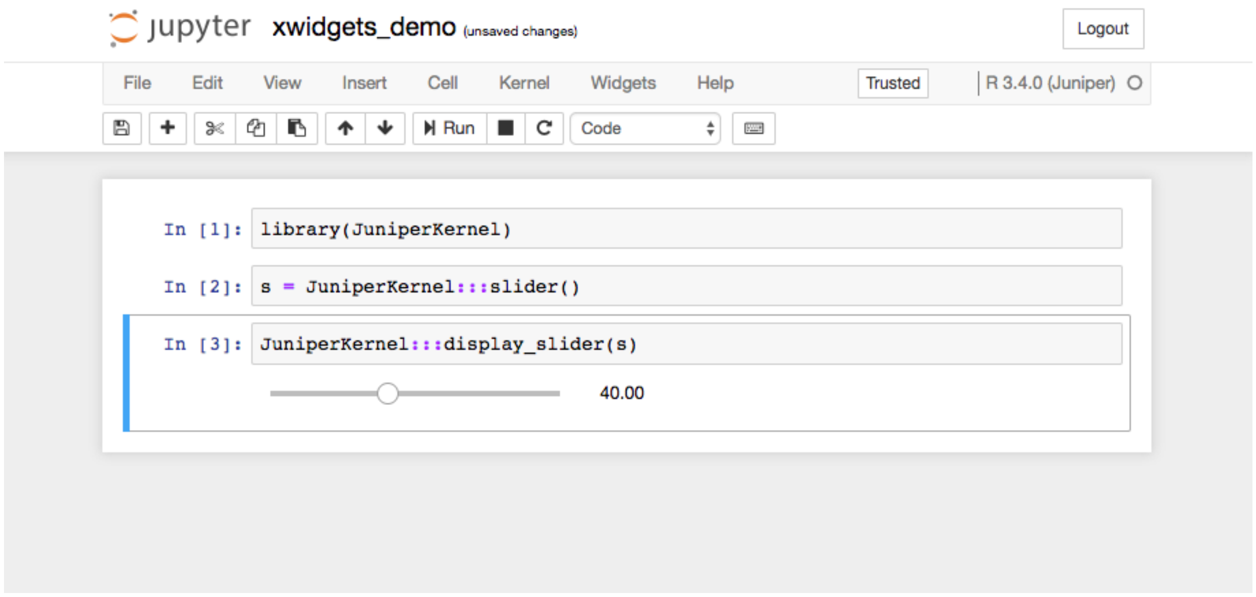
1.7.3 xwidgets

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.

1.7.4 JuniperKernel

The JuniperKernel project is a Jupyter kernel for the R programming language built with Xeus and Rcpp.



```
In [1]: library(JuniperKernel)

In [2]: s = JuniperKernel:::slider()

In [3]: JuniperKernel:::display_slider(s)
```

40.00