
xeus
Release

Jan 10, 2018

INSTALLATION

1	Licensing	3
1.1	Installation	3
1.2	Usage	4

C++ implementation of the Jupyter Kernel protocol

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

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 -c conda-forge xeus
```

1.1.2 From Source

`xeus` depends on the following libraries:

- `libzmq`, `cppzmq`, `cryptopp` and `xtl`

On linux platforms, `xeus` also requires `libuuid`, which is available in all linux distributions (`uuid-dev` on Debian).

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 cryptopp 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
```

cryptopp

cryptopp must be built as a static library, building *cryptopp* as a shared library is not supported on Windows.

```
cmake -D BUILD_SHARED=OFF -D BUILD_TESTING=OFF -D CMAKE_BUILD_TYPE=Release .
make
make install
```

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.

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

- `execute_request_impl`

- `complete_request_impl`
- `inspect_request_impl`
- `history_request_impl`
- `is_complete_request_impl`
- `input_reply_impl`

as seen in the echo kernel provided as an example.

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

using xeus::xinterpreter;
using xeus::xjson;
using xeus::xhistory_arguments;

namespace echo_kernel
{
    class echo_interpreter : public xinterpreter
    {
    public:
        echo_interpreter() = default;
        virtual ~echo_interpreter() = default;

    private:
        void configure() override;

        xjson execute_request_impl(int execution_counter,
                                   const std::string& code,
                                   bool silent,
                                   bool store_history,
                                   const xjson::node_type* user_expressions,
                                   bool allow_stdin) override;

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

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

        xjson history_request_impl(const xhistory_arguments& args) override;

        xjson is_complete_request_impl(const std::string& code) override;

        xjson kernel_info_request_impl() override;

        void input_reply_impl(const std::string& value) override;
    };
}
```

Kernel authors can then rebind to the native APIs of the interpreter that is being interfaced, providing richer information than with the classical approach of a wrapper kernel capturing textual output.