xwidgets

Johan Mabille and Sylvain Corlay

Jan 26, 2021

INSTALLATION

The C++ backend for Jupyter interactive widgets.

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

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

`xwidgets` and its dependencies require a modern C++ compiler supporting C++14. The following C++ compilers are supported:

- On Windows platforms, Visual C++ 2015 Update 2, or more recent
- On Unix platforms, gcc 4.9 or a recent version of Clang

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

xwidgets is a header-only library but depends on some traditional libraries that need to be installed. On Linux, installation of the dependencies can be done through the package manager, anaconda or manual compilation.

1.1.1 Using the conda package

A package for xwidgets is available on the conda package manager. The package will also pull all the dependencies.

```
conda install xwidgets -c conda-forge
```

1.1.2 From source with cmake

You can also install `xwidgets` from source with cmake. On Unix platforms, from the source directory: However, you need to make sure to have the required libraries available on your machine.

```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/path/to/prefix ..
make install
```

On Windows platforms, from the source directory:

```
mkdir build
cd build
cmake -G "NMake Makefiles" -DCMAKE_INSTALL_PREFIX=/path/to/prefix ..
nmake
nmake install
```

1.2 User Guide

1.2.1 What are Widgets?

Widgets are eventful C++ objects that have a representation in the browser, often a control like a slider, a textbox etc.

Widgets can be used to build interactive GUIs in Jupyter notebooks. They can also be used to synchronize information between C++ and JavaScript.

1.2.2 Using Widgets

Widgets can be used in the Jupyter notebook with the xeus-cling C++ kernel simply by importing headers of the `xwidgets` library. For example, including the `xslider` headers makes the slider widget available.

```
#include "xwidgets/xslider.hpp"
```

Displaying Widgets

Widgets can be displayed using Jupyter's display framework.

```
xw::slider<double> slider;
slider.display();
```

The Model-View-Controller Pattern

If you display the same widget twice, the displayed instances in the front-end will remain in sync with each other. Dragging one slider will modify the value for the other slider. The reason for that is that each time a widget is displayed, a new visual representation is created, reflecting the same underlying object, or model. This Model-View-Controller (MVC) architectural pattern is applied. Each C++ instance of a widget type is a new instance of the model.

1.2.3 The Value Semantics in `xwidgets`

The `xwidgets` framework differs from most common C++ widgets framework in that widgets have a *value semantics* instead of an *entity semantics*. Not dynamic polymorphism is at play, but static polymorphism based on the CRTP pattern.

A consequence, is that if a widget is *copied*, the resulting widget instance will have a new counterpart in the front-end. In the following example, `slider2` is a *copy* of `slider1`. Upon creation, a new front-end widget is created, reflecting the state of that new widget instance. The states of `slider1` and `slider2` are *not* synchronized.

```
xw::slider<double> slider1;
auto slider2 = slider1;
slider2.value = 50.0;
slider1.display();
slider2.display();
```

However, if `slider2` was a reference on `slider1`, or if `slider1` had been moved to `slider2`, `slider2` would still refer to the same widget model in the front-end.

Resource Acquisition is Initialization

The `xwidgets` framework makes use of the `RAII` pattern (Resource Acquisition is Initialization). Creating a new widget instance results in the creation of the counterpart in the HTML/JavaScript frontend. The destructor triggers the destruction of the frontend model and all the views, unless the object being destructed has already been *moved from*.

This architecture ties the lifetime of the front-end object to that of the C++ model.

1.2.4 Naming Conventions and Widget Generators

CRTP bases and final classes

Widget classes with names prefixed by `x` are not meant to be directly instantiated. For example, `xslider` is the top-most CRTP base of `slider`.

Widget classes with names that are not prefixed by `x` can be instantiated, but are *final*, i.e. they cannot be inherited from.

In fact, these final classes are typedefs on a special template parameterized by its base. For example, we have:

```
using button = xmaterialize<xbutton>;
```

Similarly, for template widget types, we also make use of the `xmaterialize` class, which

```
template <class T>
using slider = xmaterialize<xslider, T>;
```

The `xmaterialize` class only implements the final inheritance closing the CRTP, together with the RAII logic, which is to be done at the top-most inheritance level, so that widget creation messages are sent after all the bases have been initialized.

Generator classes

Simple widget types such as `slider` may have a large number of attributes that can be set by the user, such as `handle_color`, `orientation`, `min`, `max`, `value`, `step`, `readout_format`.

Providing a constructor for `slider` with a large number of such attributes would make the use of `xwidgets` very cumbersome, because users would need to know all the positional arguments to modify only one value. Instead, we mimick a keyword argument initialization with a method-chaining mechanism.

```
auto button = xw::slider<double>::initialize()
    .min(1.0)
    .max(9.0)
    .value(4.0)
    .orientation("vertical")
    .finalize();
```

This is a classical approach: calls to `min`, `max`, `value` and `orientation` all return the `slider` instance (by rvalue reference, which is optimized with C++ move semantics and copy ellision). The `finalize()` triggers the creation of the front-end object with the data.

1.2.5 Widget Events

Special Events

Certain widget types such as `button` are not used to represent data types. Instead, the button widget is used to handle mouse clicks. The `on_click` method of the `button` widget can be used to register functions to be called when the button is clicked.

```
xw::button button;

void foo()
{
    std::cout << "Clicked!" << std::endl;
}

button.on_click(foo);
button.display();
```

Xproperty Events

The observer pattern of `xwidgets` relies upon the xproperty library.

`xproperty` can be used to

- register callbacks on changes of widget properties
- register custom validators to only accept certain values
- link properties of different widgets

Registering an Observer

In this example, we register an observer for a slider value, triggering the printing of the new slider value.

```
xw::slider<double> slider;
slider.display()
XOBSERVE(slider, value, [](const auto& s) {
    std::cout << "Observer: New Slider value: " << s.value << std::endl;
});
```

Registering a Validator

In this example, we validate the proposed values for a numerical text. Negative values are rejected.

```
xw::numeral<double> number;
number.min = -100
number.display()
XVALIDATE(number, value, [](const auto&, double proposal) {
    std::cout << "Validator: Proposal: " << proposal << std::endl;
    if (proposal < 0)
    {
        throw std::runtime_error("Only non-negative values are valid.");
    }
```

(continues on next page)

(continued from previous page)

```
    return proposal;
});
```

For more details about the API for `xproperty`, we refer to the `xproperty` documentation.

1.3 Serialization and Deserialization of Widget Attributes

Properties of xobjects are automatically synchronized with the counterpart front-end object. The synchronization involves the serialization and deserialization of the modified properties.

1.3.1 Standard case: JSON serialization of properties

By default, if the type held by the property is JSON-serializable, the behavior of xwidgets will be to make use of the JSON serialization for that type to synchronize the property value between the kernel and the front-end.

By JSON-serializable, we mean that the type has to be convertible from and to the `json` type of the `nlohmann_json` package, a.k.a "JSON for Modern C++". Integral types, floating points, and standard STL containers are supported by `nlohmann_json`.

JSON serialization and deserialization for a user-defined type can be specified by providing an overload of the `to_json` and `from_json` functions in the same namespace where it is defined.

For example, the serialization of a type `ns::person` with attributes "name", "address" and "age" can be specified in the following fashion:

```
using nlohmann::json;

namespace ns
{
    void to_json(json& j, const person& p)
    {
        j = json{{"name", p.name}, {"address", p.address}, {"age", p.age}};
    }

    void from_json(const json& j, person& p)
    {
        p.name = j.at("name").get<std::string>();
        p.address = j.at("address").get<std::string>();
        p.age = j.at("age").get<int>();
    }
}
```

Upon serialization and deserialization of `ns::person` objects, the overloads of `to_json` and `from_json` are picked up by argument-dependent lookup (ADL).

Eventually, patches sent to and received from the front-end are JSON objects whose keys are the names of the attributes, and the values are the JSON representations of the new value.

For example, the JSON patch for a slider widget corresponding to a change of attributes "value", "min", "readout_format" looks like:

```
{
    "value" : 10,
    "min": 0,
```

(continues on next page)

(continued from previous page)

```
    "readout_format": ".2f"
}
```

1.3.2 Advanced use cases: making use of binary serialization

The Jupyter Widgets communication protocol allows for the communication of raw binary buffers to the front-end. This is especially convenient in visualization packages where large numerical arrays may be sent across the wire, or when serializing images and such.

The Jupyter binary serialization protocol

Upon modification of properties in the front-end or the back-end, the content of the comm message sent to or from the front-end is

- a JSON patch holding the JSON-serialized values of the modified attributes
- optionally a number of binary frames holding raw data.

On the JavaScript side, the first stage of the deserialization consists in

1. deserializing the JSON into nested JavaScript objects and arrays.
2. inserting the binary frames in the deserialized JSON in the form of DataView objects at locations specified in a companion `buffer_paths` array sent across the wire alongside the list of buffers.

For example, in the `image` widget, the `value` attribute holds the binary data for the image (encoded in the format specified with the `format` string attribute). A patch for a change if the `value` and `format` attribute will look like

```
// JSON patch:

{
    "format": "png"
}

// buffer paths
[["value"]]

// Buffers
[ { -- Binary png buffer -- } ]
```

On the JavaSCript side, this gets deserialized into

```
{
    "format": "png",
    "value": DataView({ -- Binary png buffer -- })}
}
```

On the C++ side, the internal machinery of `xwidgets` automatically composes this list of paths for the user. Custom widget authors must compose a message of the form

```
// JSON patch:

{
    "format": "png",
    "value": "@buffer_reference@0"
```

(continues on next page)

(continued from previous page)

```
}

// Buffers
[ { -- Binary png buffer -- } ]
```

Instead of specifying the buffer paths in a separate array, the location where the buffer is to be inserted holds a placeholder string indicating the index of the corresponding buffer in the list, prefixed with `@buffer_reference@`.

Making use of the Jupyter serialization protocol in `xwidgets`

Serialization

The serialization is handled by the free function

`xwidgets_serialize(value, patch, buffers);`

where

- the first argument is a const reference to the value,
- the second argument (`patch`) to the JSON object being written.
- the third argument (`buffers`) is a reference to the sequence of buffers of the message.

picked up by argument-dependent lookup, and apply to all xwidgets properties holding values of that type.

Note: The default implementation of `xwidgets_serialize` simply invokes the JSON serialization for that type. In most cases, overloading `xwidgets_serialize` is not necessary. This is mostly relevant for properties for which one wants to bypass JSON serialization or make use of binary serialization.

Deserialization

The deserialization is handled by the free function

`set_property_from_patch(property, patch, buffers);`

where

- the first argument is a reference to the property,
- the second argument (`patch`) holds a const reference to the JSON patch being read.
- the third argument (`buffers`) holds a const reference to the sequence of buffers being read.

`set_property_from_patch` is called for each property of the widget.

The default behavior of `set_property_from_patch` is to invoke the JSON deserialization for each property and it can be specialized for a specific property type.

For example, the overload of `set_property_from_patch` for the `value` property of the image widget reads:

```
inline void set_property_from_patch(decltype(image::value)& property,
                                    const nl::json& patch,
                                    const xeus::buffer_sequence& buffers)
{
    auto it = patch.find(property.name());
    if (it != patch.end())
    {
        using value_type = typename decltype(image::value)::value_type;
```

(continues on next page)

(continued from previous page)

```
        std::size_t index = buffer_index(patch[property.name()].template get
↪<std::string>());
        const auto& value_buffer = buffers[index];
        const char* value_buf = value_buffer.data<const char>();
        property = value_type(value_buf, value_buf + value_buffer.size());
    }
}
```

Note: `decltype(image::value)` is the type of the `value` property of the image widget, which is unique to the image widget, (more specifically, its type is an internal class of the image class).

This specialization is a better match than the default one and is picked-up by argument-dependent lookup, however, this will not apply to properties of other widgets or other properties of this widget also holding a `std::vector<char>`.

Overloading `xwidgets_deserialize`

The default implementation of `set_property_from_patch` reads:

```
template <class P>
inline void set_property_from_patch(P& property,
                                    const nl::json& patch,
                                    const xeus::buffer_sequence& buffers)
{
    auto it = patch.find(property.name());
    if (it != patch.end())
    {
        typename P::value_type value;
        xwidgets_deserialize(value, *it, buffers);
        property = value;
    }
}
```

which means that the default behavior is to call into `xwidgets_deserialize` with the value held by the property. A way to specify a deserialization method for a user-defined type is to overload the `xwidgets_deserialize` method for that type in the same namespace where the type is defined. Then, it will be picked up by argument-dependent lookup, and apply to all xwidgets properties holding values of that type.

Note: The default implementation of `xwidgets_deserialize` simply invokes the JSON deserialization for that type. In most cases, overloading `xwidgets_deserialize` or `set_property_from_patch` is not necessary. This is mostly relevant for properties for which one wants to bypass JSON deserialization or make use of binary deserialization.

1.4 Compiler workarounds

This page tracks the workarounds for the various compiler issues that we encountered in the development. This is mostly of interest for developers interested in contributing to xwidgets.

1.4.1 Visual Studio 2017 and `__declspec(dllexport)`

In `xwidgets.cpp` a number of widget types are precompiled, in order to improve the just-in-time compilation time in the context of the cling C++ interpreter.

However, with Visual Studio 2017, the introduction of `__declspec(dllexport)` instructions for certain widget types causes compilation errors. This is the case for widget types that are used as properties for other widgets such as `xlayout` and style widgets.

The upstream MSVC issue issue appears to have been solved with VS2017 15.7 (Preview 3). The impacted build numbers for Visual Studio are `_MSC_VER==1910`, `_MSC_VER==1911`, `_MSC_VER==1912`, `_MSC_VER==1913`.

1.4.2 Visual Studio and CRTP bases

If we have `template <class T> class Foo : public Bar<Foo<T>>`, then within the implementation of `Foo`, `Bar` should be a template, and not refer to `Bar<Foo<T>>`. However, unlike GCC and Clang, Visual Studio incorrectly makes `Bar` refer to the fully specialized template type.

1.4.3 Visual Studio and ambiguous calls to base constructors in mixins

In `xobject.hpp`, we explicitely define constructors instead of making use of the `using` statement for the base constructor because MSVC wrongly reports that the call to the base class constructor is ambiguous in derived classes.

1.5 Releasing xwidgets

1.5.1 Releasing a new version

From the master branch of xwidgets

- Make sure that you are in sync with the master branch of the upstream remote.
- In file `xwidgets_config.hpp`, set the macros for `XWIDGETS_VERSION_MAJOR`, `XWIDGETS_VERSION_MINOR` and `XWIDGETS_VERSION_PATCH` to the desired values.
- Update the README file w.r.t. dependencies on xwidgets.
- Update the environment.yml file used by binder with the new version of xwidgets and dependencies.
- Stage the changes (`git add`), commit the changes (`git commit`) and add a tag of the form `Major.minor.patch`. It is important to not add any other content to the tag name.
- Push the new commit and tag to the main repository. (`git push`, and `git push --tags`)
- Release xwidgets on conda.
- Update the stable branch to point to the latest tag.

1.6 Related projects

xplot if the C++ backend for the bqplot 2-D plotting library

C++ code

```cpp
#include <algorithm>
#include <random>
#include <vector>

#include "xwidgets/xbox.hpp"

#include "xplot/xtoolbar.hpp"
#include "xplot/xfigure.hpp"
#include "xplot/xmarks.hpp"
#include "xplot/xaxes.hpp"

auto randn(std::size_t n)
{
    std::vector<double> output(n);
    std::random_device rd;
    std::mt19937 gen(rd());
    std::normal_distribution<> dis(5, 2);

    std::for_each(output.begin(), output.end(), [&dis, &gen](auto& v){v = dis(gen);});

    return output;
}

std::size_t data_size = 200;
std::vector<double> data_x(data_size);
std::iota(data_x.begin(), data_x.end(), 0);
std::vector<double> data_y = randn(data_size);
std::vector<double> data_c = randn(data_size);

xpl::linear_scale scale_x, scale_y;
xpl::linear_scale scale_size;

auto scatter = xpl::scatter::initialize(scale_x, scale_y, scale_size)
    .x(data_x)
    .y(data_y)
    .size(data_c)
    .stroke("black")
    .default_size(128)
    .enable_move(true)
    .colors(std::vector<xtl::xoptional<std::string>>{"orangered"})
    .finalize();

xpl::axis axis_x(scale_x), axis_y(scale_y);
axis_x.label = "x";
axis_y.label = "y";
axis_y.orientation = "vertical";
axis_y.side = "left";

xpl::figure fig;
fig.padding_x = 0.025;
```

(continues on next page)

(continued from previous page)

```
fig.add_mark(scatter);
fig.add_axis(axis_x);
fig.add_axis(axis_y);

xpl::toolbar tb(fig);

xw::vbox b = xw::vbox::initialize()
    .children({fig, tb})
    .finalize();
b
```

Output

xleaflet is the C++ backend for the leaflet maps visualization library. The Python reference implementation is available in the ipyleaflet project

C++ code

```
#include "xwidgets/xhtml.hpp"

#include "xleaflet/xmap.hpp"
#include "xleaflet/xmarker.hpp"

auto html = xw::html::initialize()
    .value("Hello from an <b>xwidget</b> in an <b>xmarker</b>!")
    .finalize();

std::array<double, 2> center = {52.204793, 360.121558};

auto map = xlf::map::initialize()
    .center(center)
    .zoom(15)
    .finalize();

auto marker = xlf::marker::initialize()
    .location(center)
    .draggable(false)
    .popup(html)
    .finalize();
map.add_layer(marker);

map
```

Output

xproperty is the C++ implementation of the observer pattern underlying `xwidgets`. It is to `xwidgets` what the traitlets project is to `ipywidgets`.