



POLITECNICO
MILANO 1863



JUCE framework for VST plugins

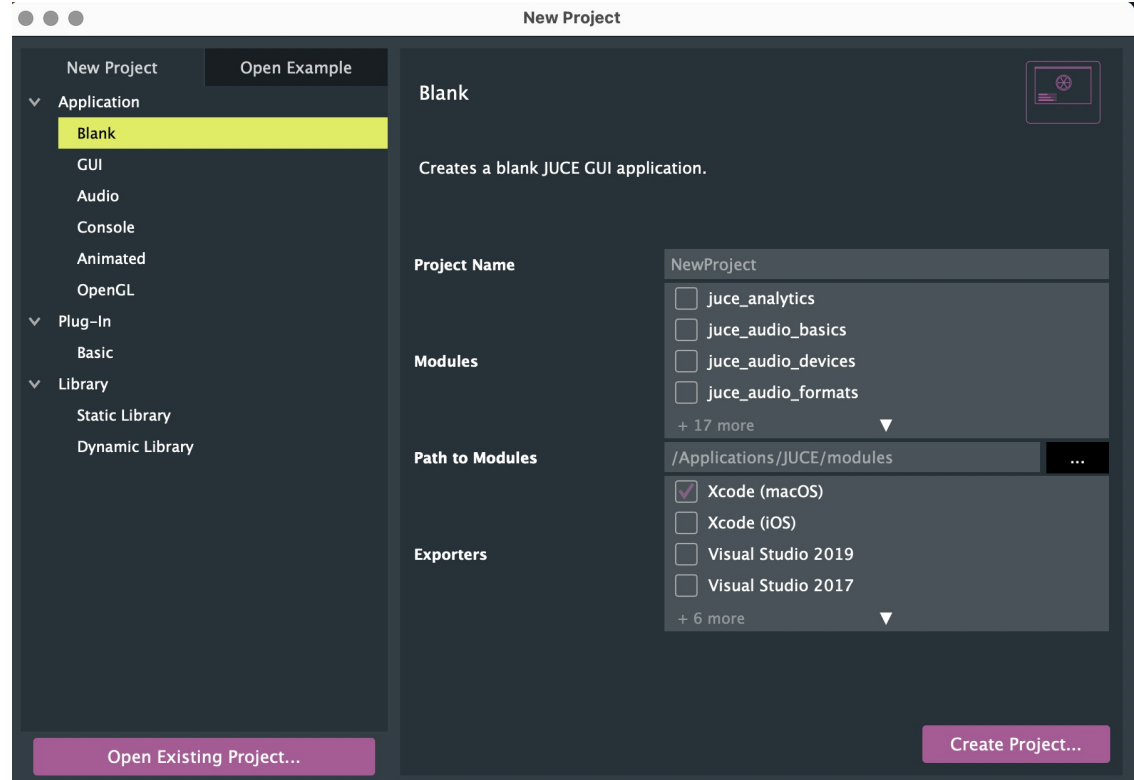
Project creation

Open **Projucer**:

- File
- New Project

and the new project wizard appears.

For each project type JUCE will generate all project files and add minimal setup code.



JUCE project types

Project type	Description
Application/Blank	This creates a blank JUCE application.
Application/GUI	This creates a minimal JUCE application with an empty application window. You can start here and add more functionality, such as more GUI components, using the various classes that JUCE offers.
Application/Audio	This creates a minimal JUCE application, like Application/GUI, but automatically adds all the setup code that you need to easily get audio input and output. You can use this for games, multimedia applications, and much more.
Application/Console	JUCE is also very useful for developing command-line applications that do not have any GUI at all. Use this project type to create such an application
Application/Animated	This creates an application which draws an animated graphical display. You can start here to create an animated mobile app, for example.
Application/OpenGL	This creates a blank JUCE application, like Application/GUI, but adds support for OpenGL to draw features including 3D model import and GLSL shaders.
Plug-In/Basic	This creates a basic audio plug-in. All the code to support VST, AudioUnit and AAX plug-in formats, is added automatically. Depending on your setup, this project type may require some additional preparation steps to work correctly. See Tutorial: Create a basic Audio/MIDI plugin, Part 1: Setting up for more information.
Library/Static, Library/Dynamic	This project type is useful to create re-usable software libraries that build on top of JUCE. The Projucer supports creating libraries for both static and dynamic linking.

Project Creation

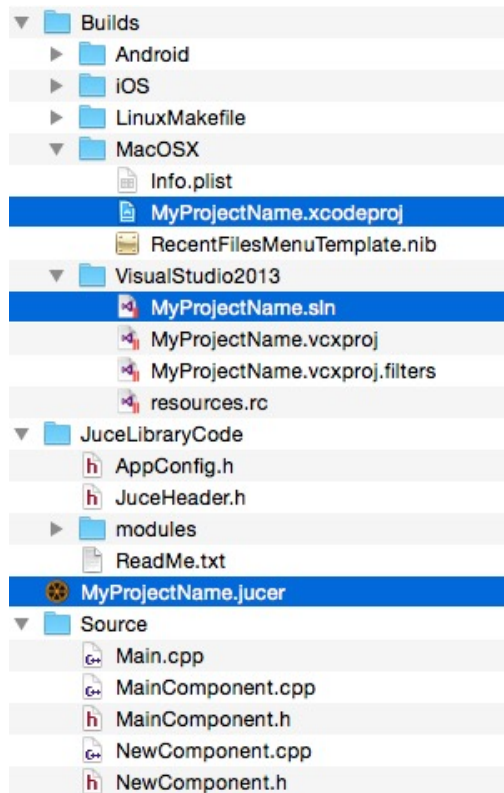
After you have selected the appropriate project type, you can fill out some additional project settings on the right-hand side of the window:

- **Project Name:** set your name for the project.
- **Modules:** The JUCE framework code is organised into *modules*. Here, you can select which modules are included in your project and in the section below you can specify the location of the modules subfolder located inside the JUCE folder you installed earlier.
- **Path to Modules:** check that the specified path is where the JUCE installation has put the modules
- **Exporters** - Here, you select which native IDEs you want to use to build and debug your app. This also defines the desktop and mobile platforms that your app will support. Don't worry, this is not a final choice — with the Projucer, you can add additional platforms and IDEs later.

Projuce Project structure

After creating a new JUCE Project, in the folder specified you will find:

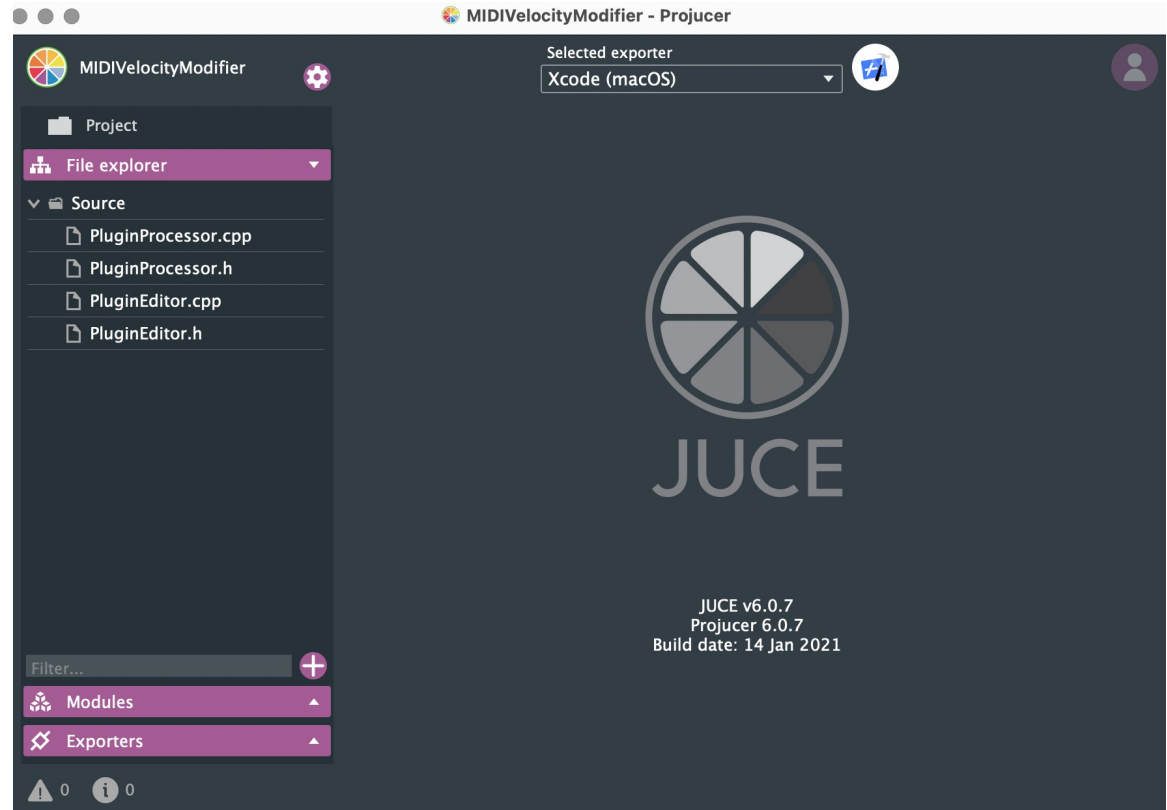
- **.juicer file**: contains all project setting, double-click for opening the project in Projucer
- **Source folder**: contains the C++ code for the project
- **Builds folder**: contains the export targets generated by Projucer; here are highlighted the Xcode project file and the Visual studio solution file
- **JuceLibraryCode folder**: contains auto-generated header files that include the JUCE library code via the JUCE modules. Note that the actual JUCE library code is not located here, but inside your global JUCE folder that you installed earlier.






Project Window

The main Projucer project window is separated into three different sections:

- toolbar
- side panel
- editor



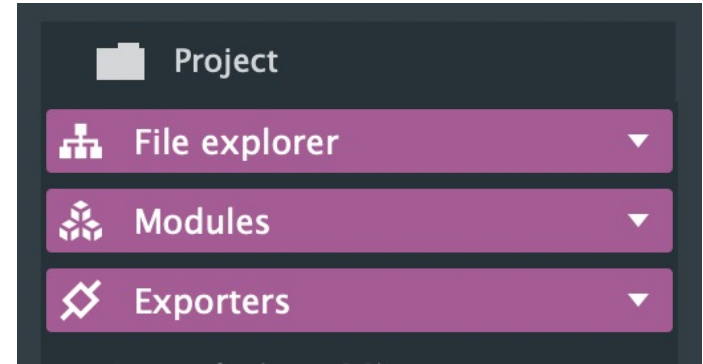
Toolbar

Action	Description
Project settings 	Opens the project settings panel.
Save and open in IDE 	Saves the project and opens it in the default IDE for the chosen exporter. Icon varies depending on IDE.
User settings 	Opens the user settings popup window.

Side Panel

The Project side panel of the Projucer is divided into three sections:

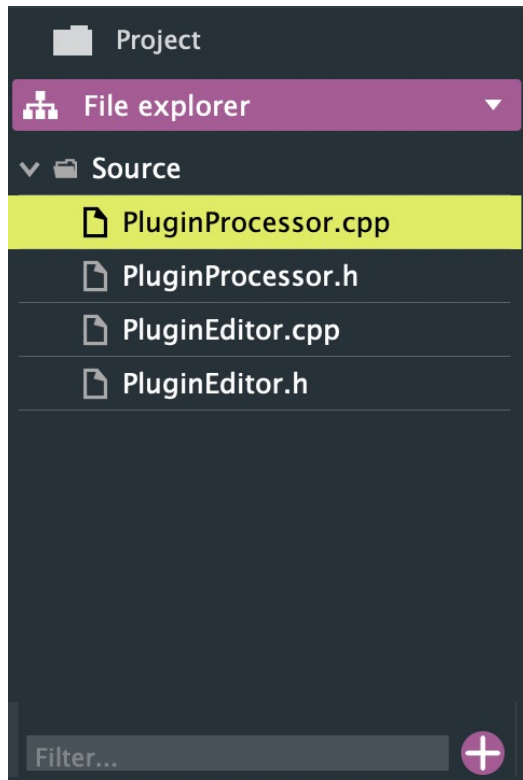
- **File explorer**
- **Modules**
- **Exporters**



File Explorer

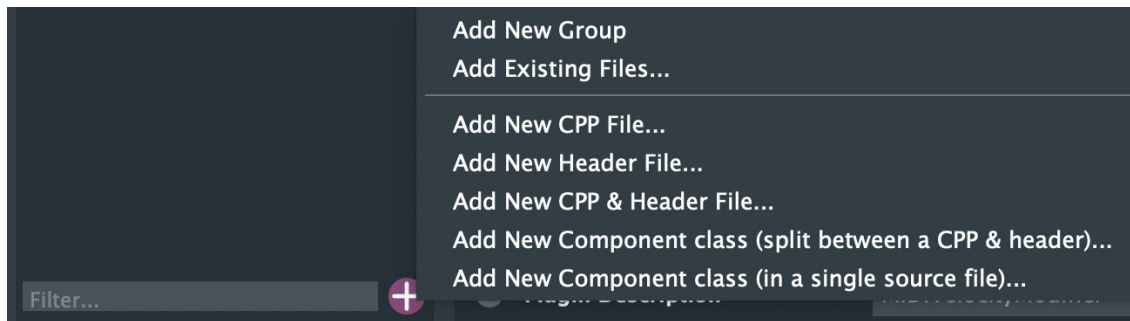
The File Explorer shows all the files of the project and Projucers offers a basic code editor for C++ source files → this is the editor we are going to use! (of course, others editors can be used)

You can add new source files to your project and delete or rename existing ones, as well as organise files into groups. To do this (and access other functionality as well), click on the "+" symbol in the bottom or right-click on the file or group, and select what you want to do from the pop-up menu.



File Explorer

Note: groups are not the same as folders. If you create a file in a group in Projucer, you will also be asked in which folder to save the file. With this, it is possible to have a folder structure that is different from the group structure. However, it is highly recommended and good practice to have your groups follow the structure and names of the folders.



Warning: you should never add, rename, and/or remove source files from JUCE projects inside your native IDE (such as Xcode, Visual Studio). These changes would be overwritten the next time you save the project in Projucer (which re-generates the native IDE projects every time). Instead, always use the Projucer itself to make such changes.

Modules

The JUCE library code is organised into different *modules*. By default, all modules that are needed for the type of your project are added.

Most JUCE modules require other JUCE modules to properly compile. If you ever remove a module on which other modules of your project depend on, then the Projucer will highlight the now "broken" modules in red. You must then either remove the "broken" modules or add the missing modules.



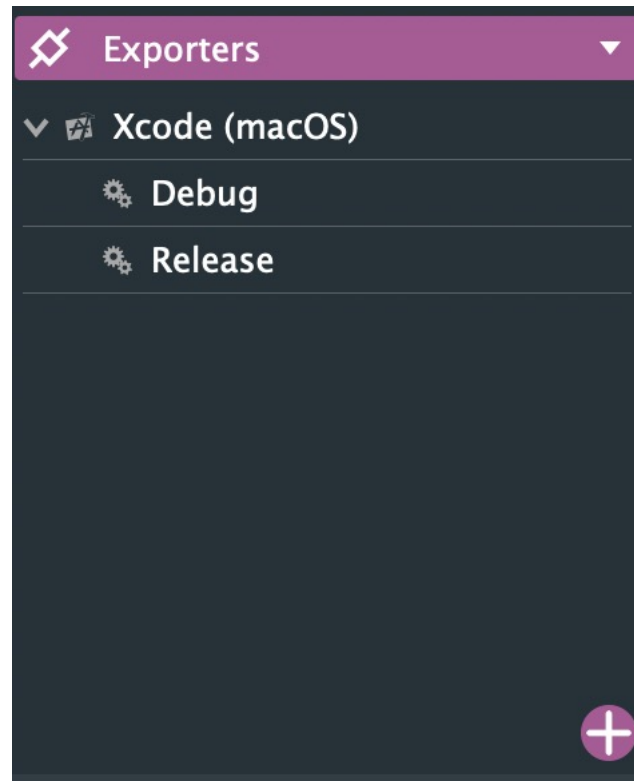
Modules

- If you click on the settings icon in the Modules tab, the module settings page opens. It provides an overview over all currently used modules and allows you to set the paths to them. To change the paths of all modules at once, enter the correct path into one of the modules, select it, and click on **Set paths for all modules...**
- Alternatively, you can choose to use global search paths for modules by clicking on **Enable/disable global path for modules...** To set your global search paths, navigate to menu item **Projuicer > Global Search Paths** on MacOS or **File > Global Search Paths** on Windows and Linux.
- If you click on an individual module in the browser, you get access to module-specific settings and properties.
- Usually, the default settings work just fine and you should rarely need to go into this page.

Modules			
Module	Version	Make Local Copy	Paths
juce_audio_basics	6.0.7	No	Global
juce_audio_devices	6.0.7	No	Global
juce_audio_formats	6.0.7	No	Global
juce_audio_plugin_client	6.0.7	No	Global
juce_audio_processors	6.0.7	No	Global
juce_audio_utils	6.0.7	No	Global
juce_core	6.0.7	No	Global
juce_data_structures	6.0.7	No	Global
juce_events	6.0.7	No	Global
juce_graphics	6.0.7	No	Global
juce_gui_basics	6.0.7	No	Global
juce_gui_extra	6.0.7	No	Global
<div>Set copy-mode for all modules... Set paths for all modules... Enable/disable global path for modules...</div>			

Exporters

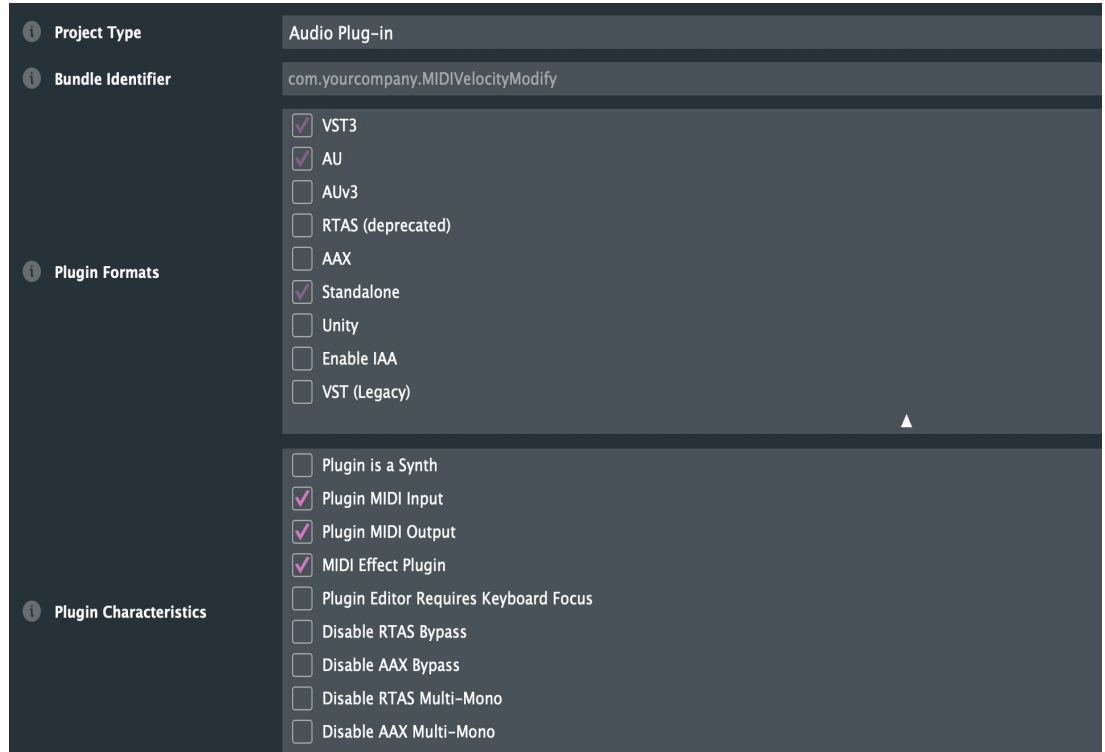
- In the Exporter tab you will find all the exporter targets specified during project creation.
- If you click on these in the browser, you get access to the build settings for this export target. This is the place to specify additional compiler and linker flags and other platform-specific build settings.
- To create a new export target for an additional IDE and platform, click on the "+" symbol. You can do this at any time.
- For example, if you originally created a JUCE application for Windows and OSX, but then later you decided to also add support for Linux, you can do it with ease.



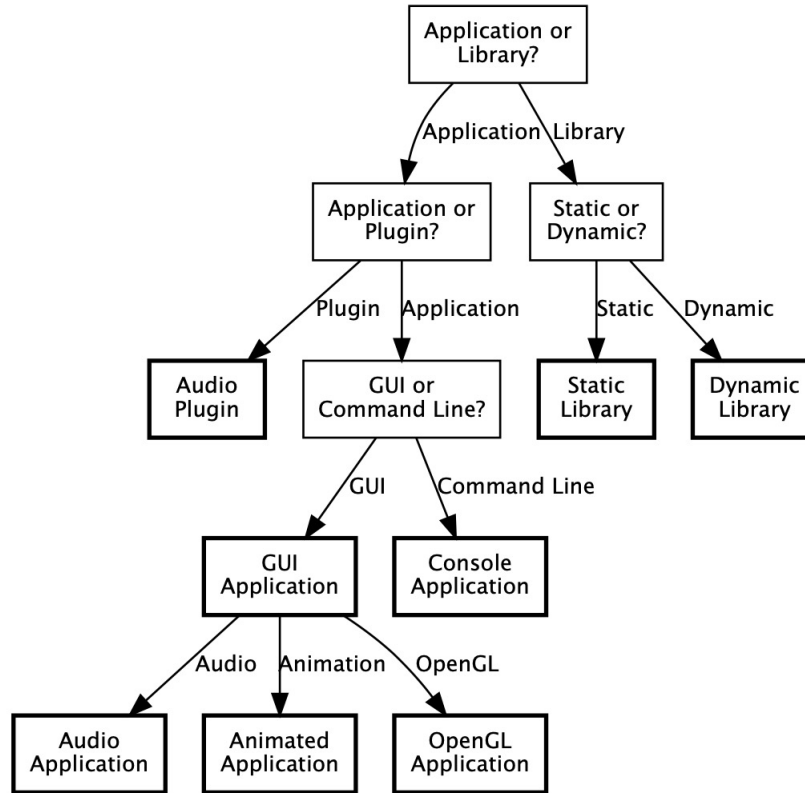
Project Setting

Among all the project settings, these are the ones that maybe you want to check:

- **Plugin Formats**
- **Plugin Characteristics**, enabling MIDI Input, Ouput, ecc depending on the project



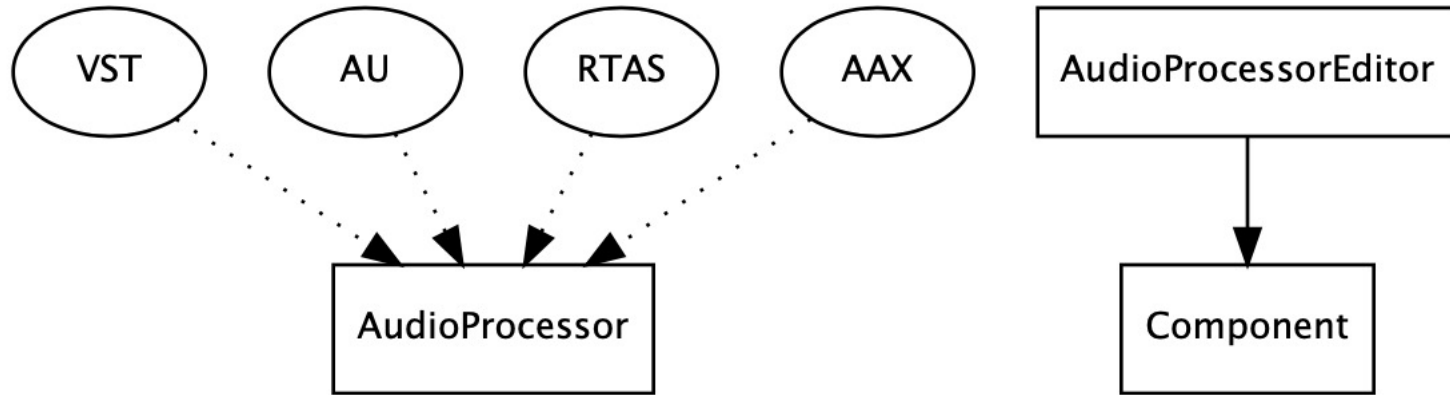
How do we choose the right project template?



Ex 1: a MIDI processor

- In this first example we will create a very simple plugin that is able to react to incoming MIDI messages by modifying their velocity value.
- We will start from an empty JUCE project and we will learn where to modify it to achieve the desired behaviour.
- Let's create a new **audio plug-in** in Projucer, and call it `MIDIVelocityModify`.
- In the project setting be sure that **Plugin MIDI Input** and **Plugin MIDI Output** are enabled.

Audio Plugin Structure



Audio Plugin Structure

The **AudioProcessor** class is an abstract base class that handles the audio processing of your audio plugin. It represents an instance of a loaded plugin and is wrapped by the different plugin formats such as VST, AU, RTAS and AAX.

The **AudioProcessorEditor** class is a base class derived from the **Component** class and holds the GUI functionalities of your audio plugin.

When an Audio Plugin project is created you will find the following files in the source folder:

- PluginProcessor.h: Header file for the PluginProcessor derived from the **AudioProcessor** class.
- PluginProcessor.cpp: Implementation file for the PluginProcessor derived from the **AudioProcessor** class.
- PluginEditor.h: Header file for the PluginEditor derived from the **AudioProcessorEditor** class.
- PluginEditor.cpp: Implementation file for the PluginEditor derived from the **AudioProcessorEditor** class.

Audio Plugin Structure

- In our project we use two derived (child) classes of these two base (father) classes:
`nameOfTheProjectProcessor` and `nameOfTheProjectProcessorEditor` (in this specific example
`nameOfTheProject = MIDIVelocityModify`)
- Hence, in the 4 files we will find:
 - `PluginProcessor.cpp`, containing the class `MIDIVelocityModifyAudioProcessor`
 - `PluginProcessor.h`, containing declaration of the class in the previous file;
 - `PluginEditor.cpp` containing the class `MIDIVelocityModifyAudioProcessorEditor`
 - `PluginEditor.h` containing the declaration of the class in the previous file.

Audio Plugin Structure

- The Processor should be considered as the parent of the Editor: there is only one Processor in a plugin but there may be multiple Editors.
- The Editor stores a reference to the Processor which it refers to, so that it can edit and access information from the audio thread.
- It is the Editor's job to set and get information on this Processor thread and not the other way around.
- The main function we will be editing in the `PluginProcessor.cpp` file is the `processBlock()` method. This receives and produces both audio and MIDI data to the plug-in output. We are going to modify also `prepareToPlay()` method too in the next exercises.
- The main function we will change in the `PluginEditor.cpp` file is the constructor, where we initialize and set up our window and GUI objects

MIDI processor

Which are the steps needed for implementing our plugin?

1. Create the GUI control
2. Pass control information, retrieved by the GUI, to the processor class
3. In the processor class, actually modify the MIDI notes

Let's see this step by step

1) Create the GUI control:

- 1 • In the declaration, create a `Slider` object as a private member of the class `MIDIVelocityModifyAudioProcessorEditor` (let's call it `Editor` from now on);

```
juce::Slider midiVolume;
```

- 2 • set the properties of this slider with various functions in the Editor constructor: to attach our slider to it , we will call `addAndMakeVisible(&midiVolume)` + modify the size of the window;

```
midiVolume.setSliderStyle (juce::Slider::LinearBarVertical);  
midiVolume.setRange(0.0, 127.0, 1.0);  
midiVolume.setTextBoxStyle (juce::Slider::NoTextBox, false, 90, 0);  
midiVolume.setPopupDisplayEnabled (true, false, this);  
midiVolume.setTextValueSuffix ("Volume");  
midiVolume.setValue(1.0);  
addAndMakeVisible (&midiVolume);
```

- 3 • modify the size and the location of the slider relative to the window whenever the window is resized, inside the function `resized()` of the `Editor` class;

```
midiVolume.setBounds (40, 30, 20, getHeight() - 60);
```


2) Pass control information to the processor

- 4 • to what variable this slider will be linked? Let's create a public variable `noteOnVel` in the `MIDIVelocityModifyAudioProcessor` declaration (let's call it `Processor` from now on)

```
public  
    float noteOnVel;
```

pass control information

- 5 • add the inheritance of `Slider::Listener` class to the `Editor` → add it as base class, so that we can register our class to receive slider changes

```
class MidivelocityModifyAudioProcessorEditor : public  
juce::AudioProcessorEditor,  
public juce::Slider::Listener
```

- 6 • add the declaration of the default callback function `sliderValueChanged` to the `Editor` class in the header file

```
void sliderValueChanged (juce::Slider* slider) override;
```

pass control information

- 7 • add the slider listener to our volume slider in the Editor constructor

```
midiVolume.addListener (this);
```

- 8 • define the actual listener function `sliderValueChanged`: this function will set our public processor volume variable `noteOnVel` depending on the value of the slider

```
void MIDIVelocityModifyAudioProcessorEditor::sliderValueChanged  
(juce::Slider* slider)  
{audioProcessor.noteOnVel = midiVolume.getValue();}
```

3) Modify the MIDI notes in the Processor class

- 9 • The `processBlock()` method in the `Processor` class receives and produces both MIDI and audio buffers in real time. We are going **to iterate through the midi buffer** to intercept signals of `noteOnType` and set their velocity to the value of our slider.
- All the modified MIDI messages will be collected in a MIDI buffer that will be swapped with the original one.
- The default code in the `processBlock()` function should be deleted.

modify MIDI notes

```
buffer.clear();
juce::MidiBuffer processedMidi;
int time;
juce::MidiMessage m;
for (juce::MidiBuffer::Iterator i (midiMessages); i.getNextEvent (m, time);) {
    if (m.isNoteOn())
    {
        juce::uint8 newVel = (juce::uint8)noteOnVel;
        m = juce::MidiMessage::noteOn(m.getChannel(), m.getNoteNumber(), newVel);
    }
    else if (m.isNoteOff()){}
    else if (m.isAftertouch()){}
    else if (m.isPitchWheel()){}
    processedMidi.addEvent (m, time);
}
midiMessages.swapWith (processedMidi);
```

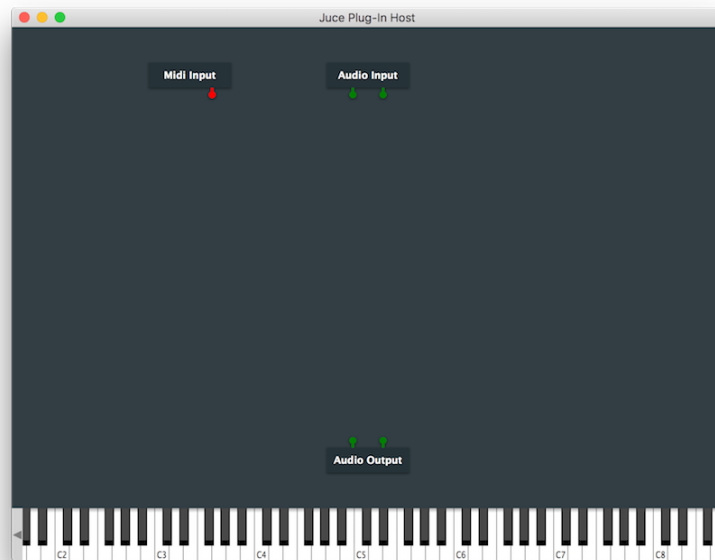
MIDI processor: test with Audio Plugin Host

AudioPluginHost is a utility distributed with JUCE that helps in testing the developed plugins;

go in the JUCE installation folder

go to extras/AudioPluginHost/ and open the .jucer file with the Projucer

- click Save Project and open in IDE
- inside your IDE build the project
- this will create a binary (on Mac OS X you will find it at extras/AudioPluginHost/Builds/MacOSX/build)
- when you run the application the window will be like this:



MIDI processor: test with Audio Plugin Host

Take care of linking all the components,
add a MIDI virtual source
(<https://github.com/flit/MidiKeys/releases>)
and a MIDI synth for testing without real
MIDI devices connected.

