API: Easy to use, hard to abuse

- Slim API
  - *Avoids complexity and separates concerns*

- Process calls will only ever get the number of channels to fill that they report during initialisation
  - *This should avoid difficult decisions about what to do with extra channels*

- Process calls will always provide empty buffers so nodes can simply "add" in to them
  - *This makes processing simpler. Multiple sources can all just add in to the buffer without clearing it first and if a node doesn't need to process it can simply return. Clearing buffers is generally a quick CPU operation*

- Optimisations are opt-in
  - *Enables a easy to use and safe API whilst not sacrificing performance*

API: Easy to use, hard to abuse

- Slim API
  - *Avoids complexity and separates concerns*
- Process calls will only ever get the number of channels to fill that they report during initialisation
  - *This should avoid difficult decisions about what to do with extra channels*
- Process calls will always provide empty buffers so nodes can simply "add" in to them
  - *This makes processing simpler. Multiple sources can all just add in to the buffer without clearing it first and if a node doesn't need to process it can simply return. Clearing buffers is generally a quick CPU operation*
- Optimisations are opt-in
  - *Enables a easy to use and safe API whilst not sacrificing performance*

Node Overview

```
class Node
{
public:
    Node() = default;
    virtual ~Node() = default;

    //==============================================================================
    /** Call once after the graph has been constructed to initialise buffers etc. */
    void initialise (const PlaybackInitialisationInfo&);

    /** Call before processing the next block, used to reset the process status. */
    void prepareForNextBlock (juce::Range<int64_t> referenceSampleRange);

    /** Call to process the node, which will in turn call the process method with the
        buffers to fill.
        @param referenceSampleRange The monotonic stream time in samples.
                                    This will be passed to the ProcessContext during the
                                    process callback so nodes can use this to determine file
                                    reading positions etc.
                                    Some nodes may ignore this completely but it should at the
                                    least specify the number to samples to process in this block.
    */
    void process (juce::Range<int64_t> referenceSampleRange);

    /** Returns true if this node has processed and its outputs can be retrieved. */
    bool hasProcessed() const;

    /** Contains the buffers for a processing operation. */
    struct AudioAndMidiBuffer
    {
        juce::dsp::AudioBlock<float> audio;
        tracktion_engine::MidiMessageArray& midi;
    };

    /** Returns the processed audio and MIDI output.
        Must only be called after hasProcessed returns true.
    */
    AudioAndMidiBuffer getProcessedOutput();
```