**Architecture Document: Modular DSP Toolkit (App + Plugin)**

**1. Overview**

This document outlines the high-level architecture of the Modular DSP Toolkit, including both the standalone JUCE-based application and the REAPER plugin. The goal is to describe how DSP tools and the framework work together to build, configure, and execute dynamic signal processing pipelines.

**2. System Components**

**2.1 DSP Designer Application (JUCE App)**

A graphical interface for users to:

* Assemble DSP effect chains using drag-and-drop modules.
* Process audio files to audition chains in real time.
* Save configurations to reusable files.

**2.2 REAPER Plugin (JUCE-based VST/AU)**

A lightweight runtime component for loading and applying DSP chains built in the app using a selected configuration file.

**2.3 Configuration File Format**

* JSON-based structure representing the DSP chain.
* Describes modules, order, and parameter values.
* Serves as the single source of truth for REAPER plugin processing.

**3. App Architecture**

**3.1 Main Components**

* **ToolboxPanel:** Left-side panel containing available DSP module icons.
* **SignalGridView:** Center area with columns representing DSP stages (left-to-right order).
  + Each column contains a stack of modules (processed top-to-bottom).
* **ControlPanel:** Bottom section showing editable parameters for the selected module.
* **AudioEngine:** Handles playback of loaded audio files and applies current DSP chain in real time.
* **ConfigManager:** Reads/writes JSON configuration files.

**3.2 Module Runtime Management**

* Each DSP module is implemented as a JUCE AudioProcessor subclass.
* Modules are wrapped and arranged dynamically into an AudioProcessorGraph.
* Graph nodes correspond to user stack order; connections are wired based on user layout.
* Any UI changes immediately update the graph for real-time auditioning.

**3.3 Signal Flow**

1. Audio file is loaded.
2. Audio is passed through each module in top-down order per column.
3. The resulting signal from each column is routed left-to-right through subsequent columns.
4. Final output is sent to the audio device for monitoring.

**3.4 Configuration Saving**

* JSON structure includes:
  + Chain metadata (name, description)
  + Module list (type, parameter map)
  + Column and order data
  + Optional macro/automation metadata

**4. REAPER Plugin Architecture**

**4.1 Main Components**

* **ConfigLoader:** Reads and parses configuration file selected by the user.
* **ProcessorGraphBuilder:** Builds an AudioProcessorGraph to match the loaded configuration.
* **MinimalUI:** Provides dropdown selector for config file, plus Wet/Dry and volume controls.
* **MacroMapper (optional):** Applies macro values to one or more module parameters.

**4.2 Runtime Flow**

1. User selects a config file.
2. Plugin parses file and instantiates defined DSP modules.
3. Modules are connected in a graph according to file-defined order.
4. Wet/Dry blend and master I/O volume are applied to final output.
5. Audio from REAPER track is processed through the assembled graph.

**4.3 Memory and Performance**

* Only modules required by the config are instantiated.
* Effect parameters are baked into the config and do not require real-time editing (aside from macros).
* Minimal UI footprint for efficiency and portability.

**5. DSP Module Interface**

All DSP tools/modules implement a shared interface:

class DSPTool : public juce::AudioProcessor

{

public:

virtual String getModuleType() const = 0;

virtual void setParameters(const juce::NamedValueSet&) = 0;

virtual NamedValueSet getParameters() const = 0;

};

This ensures each module can:

* Be dynamically created by type
* Load and save parameters cleanly
* Integrate easily into the processor graph

**6. Future Considerations**

* Plugin support for modulation (e.g., internal LFO routing)
* User tagging or categorization of chains
* Plugin macro link editor within REAPER UI (advanced users)
* OSC/MIDI binding to app or plugin parameters

**7. Summary**

This architecture provides a robust, modular framework for creating, testing, and deploying dynamic DSP chains across both standalone and DAW-integrated workflows. The separation between the design app and plugin runtime ensures creative freedom without sacrificing performance or usability in a production setting.