CineFlux-AutoXML Plugin Architecture

This document describes the plugin architecture for the CineFlux-AutoXML project, which enables extensibility through a modular plugin system.

Overview

The CineFlux-AutoXML plugin architecture is designed to allow for easy extension of the application's functionality through plugins. Plugins can provide various capabilities such as audio analysis, video analysis, subtitle processing, media transcoding, and XML generation.

The architecture consists of the following key components:

1. **Plugin Interfaces**: Define the contract that plugins must implement
2. **Plugin Registry**: Manages plugin registration, loading, and lifecycle
3. **Plugin Types**: Various specialized plugin types for different functionalities
4. **Plugin Loader**: Utility functions for loading and managing plugins

Plugin Types

The system supports the following plugin types:

- **Audio Analysis**: Analyze audio data to extract features, detect speech, etc.
- **Video Analysis**: Analyze video data to detect scenes, objects, motion, etc.
- **Subtitle Analysis**: Parse, analyze, and convert subtitle data
- **Media Transcoder**: Convert media from one format to another
- **XML Generator**: Generate XML output from structured data
- **Utility**: General-purpose utility plugins

Plugin Interfaces

All plugins must implement the `BasePlugin` interface, which defines the core functionality:

```
interface BasePlugin {
  readonly metadata: PluginMetadata;
  initialize(options?: PluginInitOptions): Promise<boolean>;
  process(options: PluginProcessOptions): Promise<PluginResult>;
  getStatus(): Promise<Record<string, any>>;
  dispose(): Promise<void>;
}
```

Specialized plugin types extend this base interface with additional methods specific to their functionality.

Plugin Metadata

Each plugin must provide metadata that describes its capabilities:

```
interface PluginMetadata {
  id: string;                    // Unique identifier
  name: string;                  // Display name
  version: string;               // Semantic version
  author: string;                // Author information
  description: string;           // Brief description
  isWasm: boolean;               // Whether this is a WASM-based plugin
  type: PluginType;              // Type of plugin
  supportedFormats?: string[]; // Supported file formats
  dependencies?: string[];       // Plugin dependencies
}
```

Plugin Registry

The `PluginRegistry` is a singleton class that manages the lifecycle of plugins:

- **Registration**: Register plugins with the system
- **Discovery**: Find plugins by type or ID
- **Loading**: Load plugins from JavaScript modules or WASM binaries
- **Unloading**: Unregister and dispose plugins
- **Events**: Emit and listen for plugin-related events

WASM Support

The plugin system supports WebAssembly (WASM) plugins, allowing for high-performance processing:

- WASM plugins can be loaded from URLs or local files
- The system handles the compilation and instantiation of WASM modules
- A wrapper is created to adapt the WASM module to the plugin interface

Creating a Plugin

To create a plugin:

1. Implement the appropriate plugin interface
2. Define the plugin metadata
3. Implement the required methods
4. Export the plugin as the default export of a module

Example:

```
export class MyPlugin implements AudioAnalysisPlugin {
  public readonly metadata: PluginMetadata = {
    id: 'my-audio-plugin',
    name: 'My Audio Plugin',
    version: '1.0.0',
    author: 'Your Name',
    description: 'A plugin that does something with audio',
    isWasm: false,
    type: PluginType.AudioAnalysis,
    supportedFormats: ['wav', 'mp3']
  };

  // Implement the required methods...
}

export default new MyPlugin();
```

Using Plugins

To use plugins in your application:

1. Initialize the plugin system
2. Register built-in plugins
3. Load external plugins as needed
4. Get plugins by type or ID
5. Process data using the plugins

Example:

```
// Initialize the plugin system
await initializePluginSystem();

// Get audio analysis plugins
const audioPlugins = getPluginsByType(PluginType.AudioAnalysis);
const audioPlugin = audioPlugins[0] as AudioAnalysisPlugin;

// Process audio data
const result = await audioPlugin.process({
  data: audioBuffer,
  format: 'wav',
  options: { detectSpeech: true }
});

// Use the result
if (result.success) {
  console.log(`Audio duration: ${result.data.duration}s`);
}
```

Plugin Events

The plugin system emits events for various lifecycle stages:

- **Registered**: When a plugin is registered
- **Loaded**: When a plugin is loaded
- **Unregistered**: When a plugin is unregistered
- **Error**: When an error occurs with a plugin
- **ProcessStarted**: When processing starts
- **ProcessCompleted**: When processing completes

You can listen for these events to monitor plugin activity.

Directory Structure

The plugin system is organized into the following directories:

- `/src/plugins`: Contains plugin interfaces and implementations
- `/src/plugins/{type}`: Contains plugins of specific types
- `/src/core`: Contains the plugin registry and loader
- `/src/types`: Contains TypeScript types for the plugin system

Future Enhancements

Potential future enhancements to the plugin system:

- Plugin marketplace for discovering and installing plugins
- Plugin versioning and compatibility checking
- Plugin configuration UI
- Plugin sandboxing for security
- Remote plugin execution