**LAMBDA FLOW**

**Technical documentation**

**1. Introduction**

SimpleLambdaFramework is a lightweight C++ library that enables seamless integration between a WebView-based HTML/CSS/JS frontend and an arbitrary backend (e.g., C#, Python, Java) via high-performance binary IPC pipes. It abstracts process creation, WebView hosting, and thread-safe message exchange into an easy-to-use API.

**1.1 Goals**

* **Simplicity**: minimal setup, clear APIs.
* **Performance**: binary IPC over pipes, worker threads.
* **Flexibility**: support any backend language/executable.
* **Cross-platform**: Windows (WebView2), macOS (WebKit), Linux (WebKit2GTK).

**2. Architecture Overview**

The framework comprises three main components:

1. **IPCBridge**: manages IPC channels (reader & writer threads) between host and backend.
2. **WebViewHost**: initializes and controls the WebView window, injects JS-to-C++ hooks.
3. **Host Application** (main.cpp): orchestrates startup, configuration loading, launches backend process, ties IPCBridge and WebViewHost.

**3. Class Reference**

**3.1 class IPCBridge**

Responsible for setting up two uni-directional pipes (frontend → backend, backend → frontend) and launching two dedicated threads:

**Public API**

class IPCBridge {

public:

IPCBridge(const std::string& backendExePath);

~IPCBridge();

// Send a JSON message (Base64-encoded) to the backend

bool sendToBackend(const std::string& jsonMessage);

// Register callback when a message arrives from backend

void onBackendMessage(std::function<void(const std::string&)> callback);

// Start the reader & writer threads and process pipe I/O

bool start();

// Stop threads, close pipes, and terminate backend process

void stop();

};

**Internals**

* **Members**:
  + std::thread readerThread\_, writerThread\_
  + int pipeToBackend\_[2], int pipeFromBackend\_[2]
  + std::mutex queueMutex\_, std::queue<std::string> writeQueue\_
  + std::function<void(const std::string&)> backendCallback\_
  + pid\_t backendPid\_
* **Workflow**:
  1. **start()**: creates pipes, forks/launches backend process, spawns threads.
  2. **writerThread\_**: waits on writeQueue\_, writes binary length + data to pipeToBackend\_.
  3. **readerThread\_**: blocks on pipeFromBackend\_, reads length, reads data, Base64-decodes, invokes backendCallback\_ on main/UI thread via postMessage.
  4. **stop()**: signals threads to exit, closes descriptors, waits on backend exit.

**Design Justification**

* **Two threads** ensure non-blocking RPC: writer decouples senders from OS writes; reader decouples blocking reads from main/UI.
* **Binary length-prefix** maximizes throughput, avoids partial reads.
* **Callbacks** allow asynchronous, event-driven handling of backend responses.

**3.2 class WebViewHost**

Abstracts platform-specific WebView creation and JS bridge:

class WebViewHost {

public:

struct Config { std::string title; int width, height; std::string startUrl; };

WebViewHost(const Config& cfg, IPCBridge& ipc);

~WebViewHost();

// Launch WebView window and message pump

bool run();

};

**Internals**

* **Members**:
  + Platform handle (HWND on Windows, GtkWindow\* on Linux, NSWindow\* on macOS)
  + Event loop integration
  + JS callback registration: intercept JS window.sendMessage(msg) calls, forward to IPCBridge::sendToBackend.
* **Workflow**:
  1. **run()**: initializes WebView control, navigates to startUrl (file:// frontend/index.html).
  2. **JS → C++**: override JS callback inside HTML (window.env.send) to call into C++.
  3. **C++ → JS**: IPCBridge reader invokes WebView's Dispatch API to call window.onBackendMessage(data) in JS.

**Design Justification**

* Decoupling WebViewHost from IPCBridge keeps UI concerns separate from IPC logic.
* Single run() method simplifies lifecycle management.

**4. Message Flow Diagrams**

**4.1 Frontend → Backend**

tsequence

Frontend->>WebViewHost: window.env.send(json)

WebViewHost->>IPCBridge: sendToBackend(json)

IPCBridge->WriterThread: enqueue message

WriterThread->>Backend process pipe: write(len+data)

**4.2 Backend → Frontend**

tsequence

Backend process->>IPCBridge pipe: write(len+data)

IPCBridge ReaderThread->>IPCBridge: read + decode

ReaderThread->>JS Thread: invoke backendCallback\_(data)

JS Thread->>WebViewHost: Dispatch JS callback

WebView->>Frontend: window.onBackendMessage(data)

**4.3 Application Startup**

flowchart

Start --> Load config.json

Load config --> IPCBridge.start()

IPCBridge.start --> launch backend process

launch backend --> init WebViewHost with config

WebViewHost.run --> enter UI event loop

**5. Configuration (config.json)**

Located alongside host-app executable:

{

"title": "My App",

"width": 800,

"height": 600,

"startUrl": "frontend/index.html"

}

Defines window properties and startup page.

**6. Design Justification Summary**

|  |  |  |
| --- | --- | --- |
| **Concern** | **Decision** | **Rationale** |
| Threading | Two dedicated threads (reader & writer) | Non-blocking, isolates I/O from UI |
| IPC Format | Length-prefixed binary + Base64 on JS boundary | Efficient, avoids framing issues |
| Callback model | onBackendMessage & sendToBackend | Asynchronous, event-driven |
| Separation | IPCBridge vs. WebViewHost | Single responsibility, modularity |
| Cross-platform | Abstract WebViewHost impl for Win/macOS/Linux | Code reuse, platform-specific hooks |