**Summary of What Enables a PostProcessor (e.g. WeatherPostProcessor)**

**✅ Step 1: You specify the name of the processor in your Access database**

* The PostProcessor column contains the string:  
  ➤ "WeatherPostProcessor"

**✅ Step 2: ApiServiceConfigLoader.cs reads it and links the implementation**

csharp

CopyEdit

var postProcessorName = reader["PostProcessor"]?.ToString();

var processor = ApiPostProcessorRegistry.GetProcessor(postProcessorName);

if (processor != null)

{

definition.PostProcess = (result, item, keyword) => processor.Process(result, item, keyword);

}

What happens here:

* It reads the PostProcessor string from the database (e.g., "WeatherPostProcessor").
* It uses ApiPostProcessorRegistry.GetProcessor(...) to resolve that name into an actual **object that implements** IApiPostProcessor.
* Then it assigns the Process(...) method to the PostProcess delegate in the ApiServiceDefinition.

**✅ Step 3: ApiPostProcessorRegistry.cs acts like a factory/lookup table**

csharp

CopyEdit

private static readonly Dictionary<string, IApiPostProcessor> \_processors =

new(StringComparer.OrdinalIgnoreCase)

{

{ "WeatherPostProcessor", new WeatherPostProcessor() }

};

* This hardcoded dictionary tells your app:

"If someone asks for WeatherPostProcessor, return a new WeatherPostProcessor() instance."

**✅ Step 4: WeatherPostProcessor.cs is a concrete implementation**

This class **implements the shared interface** and contains the logic specific to weather data:

csharp

CopyEdit

public class WeatherPostProcessor : IApiPostProcessor

{

public AggregatedResult Process(AggregatedResult result, JsonElement item, string keyword)

{

// custom weather parsing

return result;

}

}

**🧩 How the Following Classes Work Together**

**✅ 1. IApiPostProcessor.cs**

* This defines the **contract** that all processors must follow:

csharp

CopyEdit

public interface IApiPostProcessor

{

AggregatedResult Process(AggregatedResult result, JsonElement item, string keyword);

}

* It ensures consistency across all post-processors (weather, Spotify, news, etc.).

**✅ 2. ApiPostProcessorRegistry.cs**

* A **central registry** that maps processor names (from the DB) to actual processor **instances**.
* Allows you to dynamically plug in different logic per service based on what’s declared in the DB.

**🧠 Why This Design Works Well**

* ✅ Fully **declarative**: just put "WeatherPostProcessor" in your DB — no code changes.
* ✅ Extensible: you can add "SpotifyPostProcessor" or "NewsPostProcessor" later the same way.
* ✅ Decoupled: data config and business logic are separated.
* ✅ Testable: each processor is isolated, easy to unit test.