TPL Dataflow

Problem: Linking Blocks

Problem: Create a pipeline of data processing steps.

Solution: Use LinkTo to connect blocks.

Important Notes:

Allows for complex data processing workflows.

Consider using PropagateCompletion for signaling pipeline completion.

Problem: Propagating Errors

Problem: Ensure errors in dataflow are handled appropriately.

Solution: Use PropagateCompletion and handle exceptions in blocks or at pipeline completion.

Important Notes:

Helps in maintaining the integrity of dataflow chains.

Aggregating exceptions might be necessary for complex pipelines.

Problem: Unlinking Blocks

Problem: Dynamically manage data flow by unlinking blocks.

Solution: Use UnlinkAllOutput or manage IDisposable from LinkTo.

Important Notes:

Useful for runtime adjustments to data processing.

Requires careful management to avoid data loss or unintended behavior.

Problem: Throttling Blocks

Problem: Limit data processing speed.

Solution: Configure MaxDegreeOfParallelism and BoundedCapacity.

Important Notes:

Prevents overwhelming of resources or downstream systems.

Balancing throughput with system capabilities is key.

Problem: Parallel Processing

Problem: Process data in parallel through multiple instances of a block.

Solution: Utilize MaxDegreeOfParallelism on blocks.

Important Notes:

Increases throughput but can complicate debugging.

Ensure thread safety when dealing with shared state.

Problem: Creating Custom Blocks

Problem: Custom processing not covered by standard blocks.

Solution: Combine or extend existing blocks or implement custom block logic.

Important Notes:

Offers flexibility but increases complexity.

Proper encapsulation ensures blocks can be reused.

Problem: Handling Completion of Dataflow Meshes

Problem: Ensure all parts of a dataflow network complete.

Solution: Use Completion property and PropagateCompletion.

Important Notes:

Critical for knowing when a complex workflow has finished.

Can help in managing resources or triggering subsequent operations.

Conclusion

Both Rx and TPL Dataflow provide powerful tools for handling concurrency and asynchronous programming in C#, but they address different aspects:

Rx focuses on reactive programming, dealing with streams of data or events in a more declarative way.

TPL Dataflow provides a more procedural approach to managing workflows, where data explicitly moves from one processing step to another.

System.Reactive (Rx)

Problem: Converting .NET Events

Problem: Turn traditional .NET events into observable sequences.

Solution: Use Observable.FromEventPattern.

Important Notes:

Enhances event composition with other Rx operators.

Proper management of event subscriptions is necessary to prevent memory leaks.

Adaptable to any .NET event with the correct handler signature.

Problem: Sending Notifications to a Context

Problem: Ensure notifications are processed on a specific context, like the UI thread.

Solution: Use ObserveOn with the appropriate scheduler.

Important Notes:

Scheduler choice affects performance and thread safety.

Critical for applications with UI components to avoid thread exceptions.

Problem: Grouping Event Data

Problem: Group events by count or time for batch processing.

Solution:

Buffer for count-based grouping.

Window for time-based grouping.

Important Notes:

Be cautious with memory use, especially with large buffers or frequent windows.

Window creates multiple observables, which might be resource-intensive.

Problem: Taming Event Streams

Problem: Control event processing rate.

Solution:

Throttle for debouncing.

Sample for periodic sampling.

Important Notes:

Throttle waits for inactivity, useful for input debounce.

Sample can miss intermediate values, focusing only on the latest.

Problem: Timeouts

Problem: Manage operations that might take too long.

Solution: Apply Timeout for specific time constraints.

Important Notes:

Prevents system hangs; crucial for operations with external dependencies.

Requires handling of potential timeout exceptions.

Problem: Deferred Evaluation

Problem: Delay observable creation until subscription.

Solution: Use Observable.Defer.

Important Notes:

Ensures you always work with the latest data on subscription.

Reduces unnecessary work if subscriptions are not made.