Nata Overview

Jonathan Leaver

August 16, 2025

Contents

1 Finite State Machines

Think of a finite state machine in the context of many family board games. Usually, everyone sits around a table, and on the table is a game map of circles and arrows. Each person in the family has a token that they place inside one of the circles on the map to show where they are. This position represents the `State` that each player is in. Outbound from that circle are a number of arrows that represent the places the player can go (often only if certain conditions are met). As events happen during the game, these act as an `Input` to allow the player to move to a new position - a different `State`. For example, rolling some dice or drawing a card allows the player to move to a new circle (hopefully one closer to the goal).

1.1 Functional Definition

If we wanted to model each player's unique strategy, we might define a function in F# corresponding to the following signature:

```
type Strategy = State -> Input -> State
```

In essence, the player's strategy first depends on what `State` they're in. If they're at the start, they will respond to a roll of the dice differently than if they're in a difficult place on the map. Then, once the state has been established, we match with the possible `Input` values to determine a new `State`.

Suppose we have a simple game where everyone begins in the start state, let's call it `State.Start`. Then the first person to roll a 6, represented as `Input.RolledASix`, is the winner. It's a simple game, and it's exceedingly easy to model the player's strategy:

```
let strategy : Strategy =
    function
    | State.Start ->
        function
        // if we roll a six at the start, we win
        | Input.RolledASix -> State.Goal
        // otherwise we wait for the next roll to try again
        | _ -> Start
    | State.Goal ->
        function
        | _ -> Goal // once we've won, we're done
```

At this point, we've modeled our first finite state machine. It contains a set of defined states and inputs, along with the transitions between them.

2 Service.Consumer

A consumer service is an entity consisting of some state of its own (e.g. an accumulation), and an index relative to an input channel. It is defined as:

```
type Consumer<'StateOrAccumulator,'InputIndex> = {
    State:'StateOrAccumulator
    Index:'InputIndex
}
```

2.1 Optimistic Concurrency

Nata.Service aims to provide reliable, event-based micro-services using optimistic concurrency. Behind the scenes, this is implemented using the Nata.IO.Competitor<'Data> capability. At a minimum, the storage technology used for persisting consumer data needs to support either Competitor directly, or provide both Nata.IO.ReaderFrom<'Data,'Index> and Nata.IO.WriterTo<'Data,'Index>.

Since the input stream's 'InputIndex is recorded by the Consumer along with the current 'StateOrAccumulator, our service can be stopped and started at will, each time resuming from the last position. In order to maintain high-availability, multiple instances of the Consumer can also be started, operating on the same data. This can enable continuous operation during deployments, as well as potentially provide the shortest possible latency in a real-time system.

2.2 Module

2.2.1 consume - *compete to handle input*

The simplest primitive related to optimistic concurrency is a consumer that subscribes to some input channel, maintains a checkpoint of where it is in the input stream, and calls a handle function at-least-once for each input.

```
let subscribeFrom : SubscriberFrom<'Input,'InputIndex> =
    Channel.subscriberFrom input
let checkpoint : Competitor<Consumer<unit,'InputIndex>> =
    Channel.competitor checkpoints

let handle (input:'Input, index:'InputIndex) : unit =
    // cause some side-effect at-least-once,
    // e.g. a sink, an observer, etc.

Consumer.consume subscribeFrom checkpoint handle
|> Consumer.start
```

As you may observe, this Consumer uses a unit value for State.

2.2.2 consumeEvent

An overload is available to consume that exposes Nata.IO.Event<'Input> metadata from the underlying channel, which might include: date, time, partition key, correlation identifiers, etc. It differs only in the signature of its handle function:

```
let handle (input:Event<'Input>, index:'InputIndex) : unit =
    // cause some side-effect at-least-once,
    // e.g. a sink, an observer, logger, etc.

Consumer.consumeEvent subscribeFrom checkpoint handle
|> Consumer.start
```

2.2.3 fold - *compete for input using the fold pattern*

Building upon `consume`, `Consumer.fold` provides consistency around the consumer's `State`, allowing services to evaluate data with a high degree of correctness and reliability. It is similar in character to `Seq.fold` and `Seq.scan`.

```
let subscribeFrom : SubscriberFrom<'Input,'InputIndex> =
    Channel.subscriberFrom input
let compete : Competitor<Consumer<'State,'InputIndex>> =
    Channel.competitor output
let folder (current:'State option) (input:'Input) : 'State =
    // given some current state and an input, update the state

Consumer.fold subscribeFrom compete folder
|> Consumer.start
```

For example, suppose you want to maintain the total volume of all shares traded for a stock in real time. To derive this aggregate volume, you might subscribe to an input stream of trades, and add the size of each trade to the total volume accumulated so far. Moreover, it is essential that each trade contributes precisely once to the total volume, i.e. it is neither skipped nor counted multiple times, just once.

```
let sumVolume (total:Volume option) { StockTrade.Size=size } =
    match total with
    | None -> size
    | Some total -> total+size

let trades : Channel<StockTrade,SequenceNumber> = tradesFor "ibm"
let volume : Channel<Consumer<Volume,SequenceNumber>,_> = tradeVolumeFor "ibm"

Consumer.fold (Channel.subscriberFrom trades) (Channel.competitor volume) sumVolume
|> Consumer.start
```

An important design benefit, demonstrated above, is the ability to decouple our domain logic, i.e. `sumVolume`, from the I/O channels and the execution algorithm. Often, you will want to define the domain types in an external assembly referenced by your micro-services. This allows you to do extensive unit testing on the domain models themselves, and it also enables you to easily change *how* you run your micro-services, as well as *where* you store your data.

2.2.4 foldEvent

An overload of `fold`, called `foldEvent` is also provided in order to allow for domain functions that accept the `Nata.IO.Event<'Input>` data type instead of just `'Input`. All else being equal, the definition above can be altered as follows:

```
let folder (current:'State option) (input:Event<'Input>) : 'State =
    // given some current state and an input event, update the state
```

This overload can be useful if behavior should change depending on event metadata. If, for example, we wanted to sum volume only for trade events received during market hours, we could simply alter our `sumVolume` function to use `foldEvent` instead:

```
let sumVolume (total:Volume option) { Event.Data={ StockTrade.Size=size }; At=at } =
    let size =
        if at.TimeOfDay < TimeSpan.FromHours(9.5) then
            0 // pre-market
        elif at.TimeOfDay > TimeSpan.FromHours(16.) then
```

```
                0 // after market
            else size
        match total with
        | None -> size
        | Some total -> total+size

Consumer.foldEvent (Channel.subscriberFrom trades) (Channel.competitor volume) sumVolume
|> Consumer.start
```

2.2.5 map - *compete for input and apply a projection*

A projection will map each input into an output value. When applied to an input stream, it can be used to extract information, derive additional values from it, or even transform and translate that input. This can be based on established formulas or even rules that are loaded at runtime.

In general, the map function will produce one event on the output channel for each event on the input channel.

```
let subscribeFrom : SubscriberFrom<'Input,'InputIndex> =
    Channel.subscriberFrom input
let compete : Competitor<Consumer<'Output,'InputIndex>> =
    Channel.competitor output
let map (input:'Input) : 'Output =
    // given some input, project it into an output value

Consumer.map subscribeFrom compete map
|> Consumer.start
```

Suppose you're creating a trading strategy that relies on sentiment derived from real-time Twitter posts. Given that not all posts are in English, it may be useful to apply a projection to the input channel that invokes Google Translate, mapping the results onto a second channel:

```
let translate { Tweet.Characters=characters } : TranslatedTweet =
    let translation, _ = callGoogleTranslateAPI characters
    { EnglishText=translation }

let tweets : Channel<Tweet,TweetId> = input
let translated : Channel<Consumer<TranslatedTweet,TweetId>,_> = output

Consumer.map (Channel.subscriberFrom tweets) (Channel.competitor translated) translate
|> Consumer.start
```

2.2.6 mapEvent

An overload of `map`, called `mapEvent` is also provided. All else being equal, the definition above can be altered as follows:

```
let map (input:Event<'Input>) : 'Output =
    // given some input event, project it into an output value

Consumer.mapEvent subscribeFrom compete map
|> Consumer.start
```

2.2.7 bifold - *compete to fold over two inputs of differing type*

Similar to `Consumer.fold`, it can be especially useful for a consumer service to accept input from two different channels. Below, these are denoted according to *left* and *right*. As a result, the next input to be processed is represented as a `Choice<'L,'R>` with either a `left:'L` value or a `right:'R` value.

```
let l : SubscriberFrom<'L,'IndexL> =
    Channel.subscriberFrom leftInput
let r : SubscriberFrom<'R,'IndexR> =
    Channel.subscriberFrom rightInput
let compete : Competitor<Consumer<'State,'IndexL option*'IndexR option>> =
    Channel.competitor output
let folder (current:'State option) (input:Choice<'L,'R>) : 'State =
    // given some current state and an input (from the left or right), update the state

Consumer.bifold l r compete folder
|> Consumer.start
```

Note that this is an inherently non-deterministic operation. Events may arrive on either input channel at any time. When processing a large backlog of data, the bifold operation may asymmetrically favor the left or right channel. In real-time operation, generally the first to produce a value is the first to be processed with no requirement that either channel yield an event. Thus, in practice, it can be safely used to merge events as they happen on either input with no latency.

2.2.8 bifoldEvent

As before, a `bifoldEvent` overload is available for bifold functions relying on event metadata. All else being equal, the definition above can be altered as follows:

```
let folder (current:'State option) (input:Choice<Event<'L>,Event<'R>>) : 'State =
    // given some current state and an input (from the left or right), update the state

Consumer.bifold l r compete folder
|> Consumer.start
```

2.2.9 bimap - *compete to project from two inputs of differing type*

This variant of `Consumer.map` accepts input from two different channels (potentially of different types). These are denoted below as *left* and *right*. Correspondingly, each input to be processed is represented as a `Choice<'L,'R>` containing either a `left:'L` value or a `right:'R` value.

```
let l : SubscriberFrom<'L,'IndexL> =
    Channel.subscriberFrom leftInput
let r : SubscriberFrom<'R,'IndexR> =
    Channel.subscriberFrom rightInput
let compete : Competitor<Consumer<'State,'IndexL option*'IndexR option>> =
    Channel.competitor output
let map (input:Choice<'L,'R>) : 'Output =
    // given an input (from the left or right), project it into an output value

Consumer.bimap l r compete map
|> Consumer.start
```

As with `Consumer.bifold`, this is also a potentially non-deterministic function. Please refer to the notes for that function to understand their behavioral implications.

2.2.10 bimapEvent

Similarly, a `bimapEvent` overload is available for bimap functions relying on event metadata. All else being equal, the definition above can be altered as follows:

```
let map (input:Choice<Event<'L>,Event<'R>>) : 'Output =
    // given an input event (from the left or right), project it into an output value

Consumer.bimapEvent l r compete map
|> Consumer.start
```

2.2.11 multifold - *compete to fold over many inputs of the same type*

In order to fold over multiple input channels in a reliable way, the `'InputIndex` on each channel will need to be recorded. As a result, it is necessary to be able to identify each of the separate subscriptions. To accomplish this, we use an arbitrary `'SourceId`. It can be an integer, string, or any other type provided `'SourceId` can be used as a key by the `FSharp.Collections.Map` datastructure, i.e. it satisfies the `(requires comparison)` constraint.

```
let subscribersById : Map<'SourceId, SubscriberFrom<'Input,'InputIndex>> =
    [
        for unique_id, input in inputs ->
            unique_id, Channel.subscriberFrom input

    ] |> Map.ofList

let compete : Competitor<Consumer<'State,Map<'SourceId,'InputIndex>>> =
    Channel.competitor output
let folder (current:'State option) (input:'Input) : 'State =
    // given some current state and an input, update the state

Consumer.multifold subscribersById compete folder
|> Consumer.start
```

An additional requirement is that all inputs must have the same type. If we wanted to fold over three or four channels of differing types using `multifold`, a useful strategy is to map each of the inputs into a distinct case of a discriminated union. For example, suppose that we want to write an algorithmic trading strategy that accepts inputs including market data quotes, trades, orders executions, and user commands:

```
let inputs : Map<string, SubscriberFrom<Input,'InputIndex>> =
    [
        "quotes",
        Channel.subscriberFrom quotes
        |> SubscriberFrom.mapData Input.Quote

        "trades",
        Channel.subscriberFrom trades
        |> SubscriberFrom.mapData Input.Trade

        "executions",
        Channel.subscriberFrom executions
        |> SubscriberFrom.mapData Input.Execution

        "commands",
        Channel.subscriberFrom commands
        |> SubscriberFrom.mapData Input.Command

    ]
    |> Map.ofList
```

```
let algorithm (current:'State option) = function
    | Input.Quote { Bid=bid; Ask=ask } ->
      // update the state based on a new quote
    | Input.Trade { Price=price; Size=size } ->
      // update the state based on a trade
    | Input.Execution { OrderResult=result } ->
      // update the state based on an order execution
    | Input.Command Command.WorkHarder ->
      // update the state when the trader asks us to work harder
    | Input.Command Command.WorkSmarter ->
      // update the state when the trader asks us to work smarter

Consumer.multifold inputs compete algorithm
|> Consumer.start
```

In this way, positions on each input channel can be identified using a simple `'SourceId` of `string`, and the strategy can be written with an easy-to-read finite state machine.

2.2.12 multifoldEvent

An overload of `multifold` is also available for `Nata.IO.Event<'Input>`:

```
let folder (current:'State option) (input:Event<'Input>) : 'State =
    // given some current state and an input event, update the state

Consumer.multifoldEvent subscribersById compete folder
|> Consumer.start
```

Since event metadata can be different depending on the input source, the discriminated union strategy for multiple input channels described above should be used with care. Metadata will accurately reflect the information available from the input source, including any subtle differences among those sources. For instance, Apache Kafka may include partition information, whereas EventStore would not.

2.2.13 multimap - *compete to project from many inputs of the same type*

```
let subscribersById : Map<'SourceId, SubscriberFrom<'Input,'InputIndex>> =
    [
        for unique_id, input in inputs ->
            unique_id, Channel.subscriberFrom input

    ] |> Map.ofList

let compete : Competitor<Consumer<'Output,Map<'SourceId,'InputIndex>>> =
    Channel.competitor output
let map (input:'Input) : 'Output =
    // given an input (from any subscriber), project it into an output value

Consumer.multimap subscribersById compete map
|> Consumer.start
```

This can be very helpful when you need to simply merge multiple channels together. For instance, suppose that order executions are received from four different order adapters. The data is the same type, but we need to merge them together so they can be sent into our reporting database. In the following example, we use the identity function, `id`, to copy the values across without any additional transformation.

```
let subscribersById : Map<string, SubscriberFrom<'Input,'InputIndex>> =
    [   "gemini", Channel.subscriberFrom geminiExecutions
        "hitbtc", Channel.subscriberFrom hitbtcExecutions
        "kraken", Channel.subscriberFrom krakenExecutions
        "bitfinex", Channel.subscriberFrom bitfinexExecutions
    ] |> Map.ofList

let compete : Competitor<Consumer<'Input,Map<string,'InputIndex>>> =
    Channel.competitor output

Consumer.multimap subscribersById compete id // use id to simply merge 1-to-1
|> Consumer.start
```

2.2.14 multimapEvent

A `Consumer.multimap` overload is also available for `Nata.IO.Event<'Input>`:

```
let folder (input:Event<'Input>) : 'Output =
    // given an input event (from any subscriber), project it into an output value

Consumer.multifoldEvent subscribersById compete folder
|> Consumer.start
```

2.2.15 partition - *compete to partition input to an output channel*

Partition will "fan-out" events from an input channel onto a specific output channel according to the given partitioning function. All data on the published output channels must originate from this input, since output consumers are *only* written if their last input index is less than the current index of the input subscription. This helps to guarantee the correct ordering of events within a partition.

```
let subscribeFrom : SubscriberFrom<'Input,'InputIndex> =
    Channel.subscriberFrom input
let checkpoint : Competitor<Consumer<unit,'InputIndex>> =
    Channel.competitor output

let partition (input:'Input) :
    (ReaderFrom<Consumer<'Input,'InputIndex>,'OutputIndex>
     * WriterTo<Consumer<'Input,'InputIndex>,'OutputIndex>) =

    // returns the readerFrom and writerTo the output channel
    // based on any characteristic of the input
    selectOutputFor input

Consumer.partition subscribeFrom checkpoint partition
|> Consumer.start
```

Suppose we want to partition even numbers to one stream, and odd numbers to another. We might use partition as follows:

```
let odd = Channel.readerFrom odds, Channel.writerTo odds
let even = Channel.readerFrom evens, Channel.writerTo evens

let partition (input:int) =
    if input % 2 = 0 then even else odd

Consumer.partition subscribeFrom checkpoint partition
```

2.2.16 partitionEvent

Moreover, `Consumer.partitionEvent` allows for partitioning based on the underlying `Nata.IO.Event<'Input>` metadata:

```
// partition events into separate morning(AM) and afternoon(PM) channels:
let partition { Event.At=at } =
    if at.TimeOfDay < TimeSpan.FromHours 12. then AM else PM

Consumer.partitionEvent subscribeFrom checkpoint partition
|> Consumer.start
```

2.2.17 distribute - *compete to distribute input among output channels*

`Consumer.distribute` elaborates upon `partition` by permitting multiple output channels for an input. Additionally, it also allows for an empty list of output channels to be returned, in which case the input is effectively skipped, although the checkpoint will report it as processed.

As with partition, all data published on the selected outputs must originate from this input channel, since input indexes are used to guarantee that an output is written precisely once.

```
let subscribeFrom : SubscriberFrom<'Input,'InputIndex> =
    Channel.subscriberFrom input
let checkpoint : Competitor<Consumer<unit,'InputIndex>> =
    Channel.competitor output

let distribute (input:'Input) :
    (ReaderFrom<Consumer<'Output,'InputIndex>,'OutputIndex>
     * WriterTo<Consumer<'Output,'InputIndex>,'OutputIndex>
     * Merge<'Input,'Output>) list =

    // returns a list of readerFrom+writerTo+merge for output channels
    // based on any characteristic of the input
    selectOutputsFor input

Consumer.distribute subscribeFrom checkpoint distribute
|> Consumer.start
```

An additional subtlety of `distribute` is that it requires a merge function to be returned along with each output channel selected. `Merge<'Input,'Output>` defines how the operation should merge the new input into the selected output channel. For instance, returning `Merge.usingInput` will simply use the new input value. This can be adjusted during selection by providing a `Merge.using x` where x is some determined value of the `'Input` type.

2.2.18 distributeEvent

Using the `Consumer.distributeEvent` overload only requires a slight change to the distribution function:

```
let distribute (input:Event<'Input>) = // selection is exactly as before

Consumer.distributeEvent subscribeFrom checkpoint distribute
|> Consumer.start
```

2.2.19 multipartition - *compete to partition inputs among output channels*

`Consumer.multipartition`, `multipartitionEvent`, `multidistribute`, and `multidistributeEvent` differ in a very important way from their regular counterparts. In addition to performing a ”fan-out” from one input channel onto many, they can also perform ”fan-in” where multiple services publish to the same output.

The difference is most notable in the representation of the consumer index on the output channels. Instead of using `Consumer<'Output,'InputIndex>`, they now use `Consumer<'Output,Map<'SourceId,'InputIndex>>`. This allows us to provide a unique `'SourceId` for this input to differentiate it from others. One important caveat, however, is that the input index needs to be the same data type across all channels. If you need to mix input source technologies (such as EventHub which uses a `string` index versus EventStore which uses an `int64` index versus Kafka which uses a `partition, offset` pair), then it will be necessary to first apply a mapIndex operation to each source (converting their respective positions to and from `JsonValue`, for instance).

```
let subscribeFrom : SubscriberFrom<'Input,'InputIndex> =
    Channel.subscriberFrom input
let checkpoint : Competitor<Consumer<unit,'InputIndex>> =
    Channel.competitor output

let partition (input:'Input) :
    (ReaderFrom<Consumer<'Input,Map<'SourceId,'InputIndex>>,'OutputIndex>
     * WriterTo<Consumer<'Input,Map<'SourceId,'InputIndex>>,'OutputIndex>) =

    // returns the readerFrom and writerTo the output channel
    // based on any characteristic of the input
    selectOutputFor input

Consumer.multipartition subscribeFrom checkpoint sourceId partition
|> Consumer.start
```

2.2.20 multipartitionEvent

```
let partition (input:Event<'Input>) = // selection is exactly as before

Consumer.multipartitionEvent subscribeFrom checkpoint sourceId partition
|> Consumer.start
```

2.2.21 multidistribute - *compete to distribute inputs among output channels*

Although more complicated than the other functions of this module, `Consumer.multidistribute` is quite possibly the most powerful in terms of implementing sophisticated topologies. One factor to keep in mind, however, is that these functions will store a mapping from `'SourceId` to `'InputIndex` in the output channel. If we have a channel for each order in a trading system getting distributed onto an account, the I/O cost to update the account balance will grow as the number of orders increases.

```
let subscribeFrom : SubscriberFrom<'Input,'InputIndex> =
    Channel.subscriberFrom input
let checkpoint : Competitor<Consumer<unit,'InputIndex>> =
    Channel.competitor output

let distribute (input:'Input) :
    (ReaderFrom<Consumer<'Output,Map<'SourceId,'InputIndex>>,'OutputIndex>
     * WriterTo<Consumer<'Output,Map<'SourceId,'InputIndex>>,'OutputIndex>
     * Merge<'Input,'Output>) list =

    // returns a list of readerFrom+writerTo+merge for output channels
    // based on any characteristic of the input
    selectOutputsFor input

Consumer.multidistribute subscribeFrom checkpoint sourceId distribute
|> Consumer.start
```

2.2.22 multidistributeEvent

```
let distribute (input:Event<'Input>) = // selection is exactly as before

Consumer.multidistributeEvent subscribeFrom checkpoint sourceId distribute
|> Consumer.start
```

3 Service.Binding

In the previous section covering Nata.Service.Consumer, we introduced some useful ways to construct microservices with very specific semantic guarantees. When building networks of event-driven microservices on top of that library, it becomes important to streamline the way that channels and domain functions are combined in order to solve problems at real-world scale. Nata.Service.Binding makes it possible to snap channels together along with their intermediate domain logic in a way that maximizes inference and type safety while allowing a greater focus on the system topology.

3.1 Module

- Prerequisites from Nata.IO:
  - We need a mechanism to obtain a channel that supports either one of:
    - Competitor<'Data>
    - ReaderFrom<'Data,'Index> and WriterTo<'Data,'Index>
  - For these examples, consider using any of the following:
    - Memory.Stream
    - File.Stream
    - EventStore.Stream

Given such a source, let's assume the following type signature. We mark it inline to allow for different types of 'Data. In the case of Memory.Stream, the index is always int64, so we will use '_' to infer this in subsequent examples:

```
let inline channelFor (name:ChannelName) : Channel<'Data,_> =
    Nata.IO.Memory.Stream.create name
```

3.1.1 fold

Let's suppose we have an input channel of integers and we want to track the sum of all integers seen so far. We can bind two channels and the sum function together very easily:

```
let input : Channel<int,_> = channelFor "input"
let sums : Channel<Consumer<int,_>,_> = channelFor "sums"

let sum : int option -> int -> int =
    Option.defaultValue 0 >> fun total x -> total + x

let binding : seq<Consumer<int,_>> =
    input
    |> Binding.fold sum sums
```

3.1.2 start

Having created the binding from input to output, we learn something important about the evaluation of microservices in Nata.Service: that bindings themselves are sequences. We can force evaluation of the binding using Seq.iter ignore. An alternative is to pipe the binding into Binding.start : seq<'a> -> unit which will do the same thing.

```
binding
|> Binding.start
```

3.1.3 startAsync

In a script or service where you'd like to create multiple bindings and start them on the thread pool, you can use `Binding.startAsync : seq<'a> -> unit` instead, for example:

```
input
|> Binding.fold sum sums
|> Binding.startAsync
```

This comparatively simple execution mechanism opens the door for a host of powerful capabilities, such as custom supervision and monitoring.

3.1.4 asInput

Having created an output `Channel<Consumer<int,_>,_>`, we might like to use this channel as an input into a subsequent operation. This is really very simple:

```
let sumsAsInput : Channel<int,_> =
    sums
    |> Binding.asInput
```

3.1.5 map

So when we want to track the sum divided by two, we can map the `sums` channel as follows:

```
let halfSums : Channel<Consumer<int,_>, _> = channelFor "half-sums"

sums
|> Binding.asInput
|> Binding.map (fun x -> x / 2) halfSums
|> Binding.startAsync
```

3.1.6 bifold

Next, let's suppose we're watching revenue and expense events in order to determine how much money we've made. For this example, we start with an initial balance of `0m`, and a real-time stream of `revenue`, such as cash or checks received from our summer job, and `expenses`, such as cash paid for lunch or checks for the rent.

```
let updateBalance (balance:decimal) = function
    | Choice1Of2 x -> balance+x
    | Choice2Of2 x -> balance-x
let updateBalanceStartingAtZero =
    Option.defaultValue 0m >> updateBalance

(channelFor "revenue", channelFor "expenses")
|> Binding.bifold updateBalanceStartingAtZero (channelFor "balances")
|> Binding.startAsync
```

Here we've started a binding that folds over both streams, adding or removing money from our balance depending on the input channel.

3.1.7 bimap

For our next service, we're given channels for `purchases` and `leases` that contain information we can use to feed a system that estimates demand for products. As a result, we will extract product name, quantity and price into a `transactions` channel that will be used later.

```
let toTransaction = function
    | Choice1Of2 { ProductPurchased=name
                   PurchasePrice=price
                   Quantity=quantity } ->
        { Transaction.Product=name; Price=decimal price; Quantity=quantity }
    | Choice2Of2 { Name=name
                   LeaseAmount=price } ->
        { Transaction.Product=name; Price=price; Quantity=1 }

(channelFor "purchases", channelFor "leases")
|> Binding.bimap toTransaction (channelFor "transactions")
|> Binding.startAsync
```

As you can see, the new `transactions` channel will contain key elements from the underlying channels for `purchases` and `leases` merged into a new record type called `Transaction`.

3.1.8 multifold

Our smart home system manages each of the 100 watt light bulbs on our property. We want to text our teenage children when the wattage being consumed exceeds a certain threshold. Since each device publishes an event to its channel when a light is turned on or off, we should be able to track the total wattage in use.

```
let updateTotalWattageConsumed (total:int<Watts>) = function
    | TurnedOn -> total + 100<Watts>
    | TurnedOff -> total - 100<Watts>
let updateTotalWattageConsumedFromZero =
    Option.defaultValue 0<Watts> >> updateTotalWattageConsumed

let deviceChannels : Map<DeviceId, Channel<DeviceEvent,_>> =
    // from our smart home system

let totalWattageChannel = channelFor "total-wattage"

Map.toSeq deviceChannels
|> Binding.multifold updateTotalWattageConsumedFromZero totalWattageChannel
|> Binding.startAsync
```

From here, it's a simple matter to consume the `"total-wattage"` channel and text the kids when too many lights are left on:

```
let checkpoint =
    Channel.competitor(channelFor "text-alert-checkpoint")
let subscriberFrom =
    Channel.subscriberFrom(Binding.asInput totalWattageChannel)

Consumer.consume subscriberFrom checkpoint sendTextMessage
|> Consumer.startAsync
```

3.1.9 multimap

Suppose we want to create a signal for our trading system based on how much volume is being traded in the market. Since `multimap` accepts many channels of the same type, we can subscribe to trades for multiple stock symbols and compute the value of each transaction.

```
let tradeValue = function
    | { Price=price
```

```
            Quantity=quantity } -> price * decimal quantity

    let tradeValues = channelFor "trade-values"
    let tradesBySymbol : Map<Symbol, Channel<Trade,_>> =
        // from our market data

    Map.toSeq tradesBySymbol
    |> Binding.multimap tradeValue tradeValues
    |> Binding.startAsync
```

3.1.10 partition

Having created a channel of trade values, we can partition it according to the transaction size:

```
    let partitionByTradeValue =

        let large = channelFor "large-trade-values"
        let medium = channelFor "medium-trade-values"
        let small = channelFor "small-trade-values"

        fun (tradeValue:decimal) ->
            if tradeValue > 10000000.00m then large
            elif tradeValue < 1000.00m then small
            else medium

    let checkpoint =
        channelFor "partition-trade-value-checkpoint"

    tradeValues
    |> Binding.asInput
    |> Binding.partition partitionByTradeValue checkpoint
    |> Binding.startAsync
```

As a result, trades valued greater than 10 million dollars are partitioned separately from trades less than 1 thousand dollars as well as those trades of only moderate value.

3.1.11 distribute

In the next example, we accept a stream of integers, and we need to place each integer on streams according to its prime factors. For instance, 502978 will need to be placed on channels $\{2, 7, 37, 971\}$.

```
    // choose any fun F# prime factorization algorithm:
    let primeFactorsOf : int -> int Set =
        let rec loop acc x i =
            if i = x then Set.add x acc
            elif i % x = 0 then loop (Set.add x acc) x (i / x)
            else loop acc (x + 1) i
        loop Set.empty 2

    // let's also keep the prime-factor channels around
    let channelForPrime : int -> Channel<_,_> =
        let lookup = new Dictionary<_,_>()
        fun id ->
            if not(lookup.ContainsKey(id)) then
                lookup.[id] <- channelFor(sprintf "prime-factor-%d" id)
            lookup.[id]

    // based on the prime factors, merge the integer into each channel
    let distributePrimeFactors =
        primeFactorsOf
        >> Seq.map (fun x -> Merge.using x, channelForPrime x)
        >> Seq.toList

    let input = channelFor "input-numbers"
    let checkpoint = channelFor "prime-factor-checkpoint"

    input
    |> Binding.distribute distributePrimeFactors checkpoint
    |> Binding.startAsync
```

3.1.12 multipartition

We receive orders from multiple channels, such as online and in-store. Each product can be manufactured at one of our factories. Let's route orders, as a `WorkUnit`, to one of those factories using a partitioning algorithm. In this case, the offset for each order channel is tracked according to the service `id` that produced it.

```
let orderChannels =
    [ "online", channelFor "online-orders"
      "in-store", channelFor "in-store-orders" ]

let factories =
    [ channelFor "factory-0"
      channelFor "factory-1"
      channelFor "factory-2" ]

let partition { WorkUnit.Id=x } =
    factories.[ x.GetHashCode() % factories.Length ]

let checkpoint =
    sprintf "%s-order-partition-checkpoint"
    >> channelFor

for id, orderChannel in orderChannels do
    orderChannel
    |> Binding.multipartition id partition (checkpoint id)
    |> Binding.startAsync
```

As you see, we can bind multiple input channels to multiple output channels according to a partitioning algorithm - in this case, a very simple one.

3.1.13 multidistribute

When we upload photos to social media, we might want to distribute those to the news feed for each of our friends. In the following example, we accept a photo (consisting of a `Photographer` and `Link` among other fields. For each photo, we obtain list of channels for this photographer and share the link.

```
let friendsOf (name:Name) : Channel<_,_> list =
    // return channels for this person's friend list
```

```
let postPhotoFor { Photographer=name
                   Link=link } =
    [   let sharedPhoto = SharedContent.Photo link
        for friendChannel in friendsOf name ->
            Merge.using sharedPhoto, friendChannel ]

let photoUploadChannel = channelFor "photo-uploads"
let id, checkpoint =
    "shared-photos", channelFor "photo-share-checkpoint"

photoUploadChannel
|> Binding.multidistribute id postPhotoFor checkpoint
|> Binding.startAsync
```

First, each friend's feed can accept `SharedContent` from a variety of different sources in addition to our `"shared-photos"` service. Additionally, the `Merge.using` operation allows us to transform our input data for our output channels. We'll cover the `Merge` and `MergeEvent` modules next.

3.1.14 reset

Finally, it may be necessary to reset a service to an alternate position or state. For the previous example, let's obtain the index of `Position.Start` in the input channel, and reset the checkpoint. Remember, the state of a checkpoint channel is unit.

```
Channel.indexer photoUploadChannel Position.Start
|> Binding.reset () checkpoint
```

3.2 Service.Merge

When using `distribute` and `multidistribute` from either `Service.Consumer` or `Service.Binding`, merge strategies can be provided. As a result, two convenience functions are available:

usingInput places the input value directly on the target channel.

using accepts an alternative value corresponding to the channel.

You can also supply a custom implementation that accepts the current value, the incoming value, and a strategy for resolving them, of type `'State option->'Input->'State`. In both cases, the type of the merge value will need to match the `'State` of the channel receiving it.

3.3 Service.MergeEvent

For `Consumer.distributeEvent` and `Consumer.multidistributeEvent`, merge functions are also provided that use `Nata.IO.Event<'Input>`:

usingInput places the input event value directly on the target channel

using accepts an alternate event value corresponding to the channel.

As before, a custom merge strategy of `'State option->Event<'Input>->'State` can be also provided, where `'State` matches the receiving channel.

4 Service.Hub.Snapshot

4.1 Module

A snapshot hub makes a single subscription to an input channel. It's useful when subscriptions are expensive, and when consumers lagging under heavy load can be satisfied with the most recent events from the subscription.

4.1.1 create

```
// create a hub and subscribe it to a channel
let connectToHubFor (channel:Channel<'Data,'Index>) : unit->seq<Event<'Data>> =

    let subscribeFrom : SubscriberFrom<'Data,'Index> =
        Channel.subscriberFrom channel

    Hub.Snapshot.create subscribeFrom
```

For example, you might use this to provide the latest quote from a streaming ticker of stock prices. A connection to the market data feed is potentially expensive, and a client will prefer the most recent snapshot if there are several in the stream. Consider the following example:

```
let ibm : StockSymbol = "IBM"
let quotes : Channel<StockQuote,LiveFeedIndex> = feedFor ibm

let connectToHub : unit->seq<Event<StockQuote>> =
    quotes
    |> Channel.subscriberFrom
    |> Hub.Snapshot.create

[
  // Consumer #1: processes quotes after some initial lag
  async {
    do! Async.Sleep(10)
    for { Data=quote } in connectToHub() do
      consume quote
  }
  // Consumer #2: incurs heavy lag while processing each quote
  async {
    for { Data=quote } in connectToHub() do
      do! Async.Sleep(500)
      consume quote
  }
]
|> Async.Parallel
|> Async.RunSynchronously
```

Suppose the underlying quote stream contains the sequence $\{q_1..q_5\}$. In this example, the first consumer might observe the sequence $\{q_2, q_3, q_4, q_5\}$, while the second consumer might see $\{q_1, q_5\}$. Both consumers will always receive the latest snapshot q_5 in the stream. However, the first consumer was late to start and missed q_1, whereas the second consumer incurred a sizeable lag processing q_1 and next observed q_5.

A snapshot hub retains the most recent event, and provides that event (if available) to a new consumer. As a result, memory consumption during high-latency processing or heavy load is minimized. Additionally, note that event aggregation is not provided by the snapshot hub: if aggregation is required, you should apply that step to the input channel using something like `fold`.

5 Nata.Core

5.1 Overview

Nata.Core gathers small, reusable building blocks that make day-to-day functional programming more expressive and safe. They emphasize predictable semantics, ergonomic pipelining, and minimal ceremony. You will find tuple helpers for shaping data, sequence operators for time-series and merging, option/null handling, numeric and string utilities, and small bridges between Async and Task.

5.1.1 Design goals

- Ergonomic piping: helpers like swap, mapFst, and mapSnd keep pipelines readable.
- Safety by default: null- and exception-safe adapters (Null.toOption, Option.tryFunction) reduce accidental partiality.
- Time-series friendly: Seq.mergeBy, Seq.changesBy, and Seq.delta focus on ordered streams.
- Pragmatic interop: small shims for parsing and clamping (*.ofString, *.between) and for Task¡-¿Async.

5.1.2 Guided examples

Tuples and pipelining Shaping events often starts with tuples. Use mapFst and mapSnd to transform one side without touching the other.

```
// unpack (Event * Index) into (Data * Index)
let unpack : (Event<'a>*'i) -> ('a*'i) =
    mapFst Event.data

// look up a sequence of keys via a map, left-to-right
let tryFindAll (m:Map<'k,'v>) (keys:seq<'k>) : seq<'v> =
    keys
    |> Seq.choose (m |> swap Map.tryFind)
```

Time-series: changes and deltas Filter out consecutive duplicates or compute first-order differences with Seq.changesBy and Seq.delta.

```
// keep only changes in closing price
let changedCloses : seq<decimal> =
    closes |> Seq.changesBy id

// turn a stream of values into pairwise deltas
// e.g. [10m; 12m; 11m] -> [2m; -1m]
let deltas : seq<decimal> =
    closes
    |> Seq.delta
    |> Seq.choose (fun (last,next) -> last |> Option.map (fun l -> next - l))
```

Merging ordered streams When two feeds are independently ordered (e.g., by timestamp), Seq.mergeBy preserves overall order without re-sorting.

```
// left and right are (Event * timestamp) streams
let mergedData : seq<'a> =
    Seq.mergeBy snd left right
    |> Seq.mapFst Event.data
    |> Seq.map fst  // drop timestamps if desired
```

Options, nulls, and exceptions Prefer options to represent absence. Convert nulls, coalesce fallbacks, and sink exceptions into `None` when it's appropriate.

```
let bigOrder : int option =
    qty |> Option.whenTrue (fun q -> q > 1000)

let preferred : string option =
    someNullableString |> Null.toOption

let safeParsed : int option =
    "42" |> Option.tryFunction Int32.Parse

let chosen : decimal =
    primary
    |> Option.coalesce backup
    |> Option.defaultValue 0m
```

Parsing and clamping Small helpers make validation pipelines explicit and testable.

```
let cappedRatio : decimal option =
    input
    |> Decimal.ofString                  // string -> decimal option
    |> Option.map (Decimal.between (0m, 1m)) // clamp into [0,1]
```

Tasks and async Bridge `Task`-based APIs into `Async`, or wait safely for results in controlled places.

```
// get an Async<'a> from Task<'a>
let asAsync : Async<'a> = Task.waitForResultAsync someTask

// synchronously wait (e.g., at boundaries)
let result : 'a = Task.waitForResult someTask

// rethrow with original stack trace inside async workflows
let rethrow (e:exn) = Async.reraise e
```

You can combine these primitives to build clear, reliable pipelines that keep domain logic front-and-center while handling the practical edges of real-world data.

5.2 Package

1. **Nata.Core** from NuGet[1], or using Paket[2]:

   ```
   ./.paket/paket.exe add nuget Nata.Core project MyProject
   ```

2. **Reference** and open the Nata.Core library:

   ```
   #r "packages/Nata.Core/lib/netstandard2.0/Nata.Core.dll"
   open Nata.Core
   ```

5.3 Extensions

Small, sharp tools that smooth common pipelines: generating identifiers, reshaping tuples, and composing selectors so that domain logic stays readable. The emphasis is on tiny, predictable helpers that slot into larger flows.

[1]https://www.nuget.org/packages/Nata.Core/
[2]https://fsprojects.github.io/Paket/

- generate a unique id, or the bytes of one

```
guid()
// val it : string = "b2bb73b79ad04871a75a20a478fb3c8c"

guidBytes()
// val it : byte [] = [|131uy; 238uy; 246uy; 245uy; 252uy; ...

// e.g., shuffle a list
[1..10] |> List.sortBy guid
```

- swap the parameters of a function, to make pipelining easier

```
// e.g. look up a long sequence of keys
let dataSeq = keySeq |> Seq.choose (map |> swap Map.tryFind)
```

- compose functions to mapFst or mapSnd a tuple

```
let unpack : (Event<DataT>*IndexT) -> (DataT*IndexT) =
    mapFst Event.data
```

- filterFst or filterSnd

- chooseFst or chooseSnd

Mini examples

```
// transform only the index (snd) in a (data * index) tuple
let bumpIndex : ('a * int) -> ('a * int) =
    mapSnd ((+) 1)

// filter a list of (key * value) by key only
let onlyImportant : ('k * 'v) list -> ('k * 'v) list =
    List.filter (filterFst isImportant)

// choose on the first or second element to conditionally keep tuples
let keepIfParsedFst : (string * 'b) list -> (int * 'b) list =
    List.choose (chooseFst (fun s ->
        match System.Int32.TryParse s with true, v -> Some v | _ -> None))

let keepIfParsedSnd : ('a * string) list -> ('a * int) list =
    List.choose (chooseSnd (fun s ->
        match System.Int32.TryParse s with true, v -> Some v | _ -> None))
```

See also Seq.mapFst/Seq.mapSnd for sequence variants.

5.3.1 Seq

- Seq.tryHead

- use Seq.mapFst or Seq.mapSnd to transform just the data or index of a seq

```
let data = events |> Seq.mapFst Event.data
```

See also mapFst/mapSnd for tuple variants.

- similarly use Seq.filterFst/Seq.filterSnd or Seq.chooseFst/Seq.chooseSnd to select elements by data or index:

```
events |> Seq.filterFst (Event.data >> isGood)
```

- `Seq.log` `Seq.logi`
- `Seq.trySkip`
- `Seq.consume` from multiple sequences as data becomes available

  ```
  [ blockedSeq; blockedSeq; readySeq; blockedSeq ]
  |> Seq.consume
  |> Seq.iter calculate // immediately calculates a value from readySeq
  ```

  Behavior: pulls from multiple enumerators concurrently, yields values as they arrive, stops when all are exhausted, rethrows the first exception encountered, and attempts to dispose all enumerators.

- `Seq.merge` pulls from left or right based on whichever value comes first

  ```
  Seq.merge [ 1; 2; 3; 7; 1; ] [ 0; 5; 8; 8 ]
  // val it : seq<int> = seq [0; 1; 2; 3; 5; 7; 1; 8; 8]
  ```

- `Seq.mergeBy` can be used to correctly merge streams of time series data

  ```
  let dataByTime =
      Seq.mergeBy snd leftEvents rightEvents
      |> Seq.map (fst >> Event.data)
  ```

  Semantics: inputs should already be ordered by the projection for correct global ordering; when keys are equal, both elements are yielded (left then right).

- `Seq.changes`
- `Seq.changesBy`
- `Seq.delta`

Determinism and cost: `Seq.changes`/`Seq.changesBy` and `Seq.delta` are single-pass, streaming operations that retain only the last projected value.

Guided examples

```
// trace elements in a stream during diagnostics
events
|> Seq.log (fun x -> printfn "observed = %A" x)
|> Seq.iter ignore

// trace with index when order matters
events
|> Seq.logi (fun i x -> printfn "#%d = %A" i x)
|> Seq.iter ignore

// skip the first n elements if they represent a warmup period
events
|> Seq.trySkip 10
|> Seq.iter process

// merge in order by timestamp
let ordered : seq<Event<'a>> =
    Seq.mergeBy (fun (e,ts) -> ts) left right
    |> Seq.map fst
```

```
// compute point-in-time differences
let diffs : seq<decimal> =
    values
    |> Seq.delta
    |> Seq.choose (fun (last,next) -> last |> Option.map (fun l -> next - l))
```

Small utilities: heads and change detection

```
// tryHead returns None on empty sequences
let firstEvent : Event<'a> option =
    events |> Seq.tryHead

// changes keeps first element and any element whose projection differs from the previous one
let uniqueValues : seq<'a> =
    xs |> Seq.changes

// changesBy allows custom projection (e.g., by key)
let uniqueByKey : seq<'a> =
    xs |> Seq.changesBy (fun x -> x.Key)
```

5.3.2 Option

- provides equivalent functions to `Option.defaultValue` and `Option.defaultWith` from the newer F# core library
- `Option.whenTrue` will convert a value into an option type of `Some` only if the predicate is satisfied, otherwise yielding `None`

  ```
  let big = 3 |> Option.whenTrue (fun x -> x > 10)
  ```

- similar to the SQL coalesce operator, `Option.coalesce` will take the first non-`None` value in the pipeline

  ```
  let maybeThreeOtherwiseFourOtherwiseNone =
      maybeThree
      |> Option.coalesce maybeFour
      |> Option.coalesce maybeNone
  ```

- `Option.coalesceWith` will attempt to fill in `None` using a function (similar to `Option.defaultWith`)

  ```
  let maybeFourIfNone =
      maybeNone
      |> Option.coalesceWith maybeProduceAnExpensiveFour
  ```

- `Option.tryFunction` will sink any exceptions into `None`

  ```
  input
  |> Option.tryFunction (fun x -> failwith "error to sink"; x)
  //val it : SomeType option = None
  ```

- split an optional tuple into two separate options using `Option.distribute`

  ```
  let someOne, someTwo =
      Some (1,2)
      |> Option.distribute
  // val someOne : int option = Some 1
  // val someTwo : int option = Some 2
  ```

- collapse an `'a option option` field using `Option.join`

  ```
  Some (Some 3) |> Option.join
  // val it : int option = Some 3

  Some None |> Option.join
  // val it : int option = None

  None |> Option.join
  // val it : int option = None
  ```

Notes and examples

```
// prefer the first Some; otherwise fall back
let chosenUser : User option =
    primaryUser
    |> Option.coalesce backupUser

// compute a fallback only when needed
let config : Settings =
    cached
    |> Option.coalesceWith (fun () -> loadFromDisk())
    |> Option.defaultWith (fun () -> defaultSettings())

// provide a concrete default when both are None
let threads : int =
    None |> Option.defaultValue 4

// split and join of nested options
let a,b = Some (1,2) |> Option.distribute   // Some 1, Some 2
let flattened : int option = Some (Some 3) |> Option.join  // Some 3
```

When failures must preserve error information, prefer `Result`; `Option.tryFunction` is best when "absence on failure" is the intended signal.

5.3.3 Null

- safely convert `null` values to `option`

  ```
  nullString
  |> Null.toOption
  // val it : string option = None
  ```

5.3.4 Nullable

Transform `Nullable` values, or exchange them for `option`.

- apply `Nullable.map` as you would with `Option.map`

  ```
  Nullable 3
  |> Nullable.map ((+) 2)
  // val it : Nullable<int> = 5

  Nullable ()
  |> Nullable.map ((+) 2)
  // val it : Nullable<int> = null
  ```

- convert a `Nullable.toOption` and get the `Nullable.ofOption` values

```
let threeOption =
    Nullable 3
    |> Nullable.toOption
// val threeOption : Option<int> = Some 3

threeOption
|> Nullable.ofOption
// val it : Nullable<int> = 3
```

5.3.5 Boolean

- `Boolean.ofString` / `Boolean.toString`

Boolean helpers: quick examples

```
let featureEnabled : bool option = "true"  |> Boolean.ofString  // Some true
let asText         : string      = false   |> Boolean.toString  // "False"
```

5.3.6 Decimal

- `Decimal.between`
- `Decimal.ofString` / `Decimal.toString`

5.3.7 Int64

- `Int64.between`
- `Int64.ofString` / `Int64.toString`

5.3.8 Int32

- `Int32.between`
- `Int32.ofString` / `Int32.toString`

Numeric helpers: quick examples

```
// parse with validation, clamp to a safe range
let safePercent : int option =
    input
    |> Int32.ofString
    |> Option.map (Int32.between (0, 100))

// decimal ratios clamped to [0m, 1m]
let ratio : decimal option =
    s
    |> Decimal.ofString
    |> Option.map (Decimal.between (0m, 1m))

// 64-bit parsing and clamping
let maybeId64 : int64 option = "12345" |> Int64.ofString
let cappedId64 : int64 option = maybeId64 |> Option.map (Int64.between (0L, 10000L))

// toString helpers for serialization/logging
```

```
let iAsText : string = 42            |> Int32.toString
let dAsText : string = 12.34m        |> Decimal.toString
let lAsText : string = 123456789L    |> Int64.toString
```

5.3.9 String

- `String.replace`
- `String.remove`
- `String.contains`
- `String.containsIgnoreCase`
- `String.split`
- `String.trySubstring`
- `String.tryStartAt`

Null-safe string operations

```
// case-insensitive search
let hasHello : bool =
    "Hello, World!" |> String.containsIgnoreCase "hello"

// safe slicing when indices may be out of bounds
let middle : string option =
    "abcdef" |> String.trySubstring 2 3   // Some "cde"

// tolerant splitting for nullable input
let parts : string list =
    (null:string) |> String.split ','     // []

// simple edit helpers
let masked : string = "abc-123-xyz" |> String.remove "-"
let swapped : string = "color"       |> String.replace "or" "our"   // "colour"

// contains with null-tolerance
let hasToken : bool = "a,b,c" |> String.contains "b"

// safe substring from index to end
let tail : string option = "prefix-body" |> String.tryStartAt 7   // Some "body"
```

Null semantics: `String.contains null null` returns true; if either argument is null otherwise, it returns false.

5.3.10 Async

- `Async.reraise`

5.3.11 Task

- `Task.wait`, `Task.waitAsync`
- `Task.waitForResult`, `Task.waitForResultAsync`

Concurrency interop

```
// rethrow inside async with original stack trace
let runSafely (work: Async<'a>) : Async<'a> = async {
    try
        return! work
    with e ->
        Async.reraise e
        return Unchecked.defaultof<'a> // unreachable
}

// bridge Task<'T> -> Async<'T>, or wait at well-chosen boundaries
let fromTask : Async<'a> = Task.waitForResultAsync someTask
let result   : 'a        = Task.waitForResult someTask

// wait asynchronously for Task (no result)
do! Task.waitAsync someUnitTask

// or synchronously (use sparingly, at process boundaries)
Task.wait someUnitTask
```

5.4 Codec

A codec is a small, composable pair of pure functions used to translate values back and forth between two representations. It captures both directions explicitly:

```
// A codec is (encode : 'In -> 'Out, decode : 'Out -> 'In)
type Codec<'In,'Out> = ('In -> 'Out) * ('Out -> 'In)
```

This makes codecs pleasant to wire together: you can reverse them when you need the opposite direction, and concatenate them to build multi-step translations that still round-trip.

5.4.1 Helpers: encoder, decoder, reverse, concatenate

- `Codec.encoder` and `Codec.decoder` project out the first and second function of a codec. These are convenient when you prefer named accessors over tuple deconstruction.
- `Codec.reverse` swaps directions, turning a codec `'In <-> 'Out` into `'Out <-> 'In`.
- `Codec.concatenate` composes two codecs end-to-end so you can build `'A <-> 'C` from `'A <-> 'B` and `'B <-> 'C`.

Guided examples

```
open Nata.Core

// Pull out the encode/decode functions
let encodeInt, decodeInt = Codec.Int32ToString
let text : string = encodeInt 42             // "42"
let back : int    = decodeInt "not-a-number"// 0 (tolerant default)

// Reverse a codec when you need the opposite direction
let stringToDecimal : Codec<string, decimal> =
    Codec.reverse Codec.DecimalToString

let parse : string -> decimal = Codec.encoder stringToDecimal
let d1 = parse "12.50"    // 12.50m
let d0 = parse "oops"      // 0m (tolerant default)

// Concatenate to build multi-step codecs (Int32 <-> String <-> Bytes)
let Int32ToBytes : Codec<int, byte[]> =
    Codec.Int32ToString
    |> Codec.concatenate Codec.StringToBytes

let toBytes   : int   -> byte[] = Codec.encoder Int32ToBytes
let fromBytes : byte[]-> int    = Codec.decoder Int32ToBytes

let bytes   = toBytes 42
let value42 = fromBytes bytes    // 42

// You can also decompose and pipe directly if you prefer
let textAgain : string =
    42 |> Codec.encoder Codec.Int32ToString

let safeBool : bool =
    "TRUE" |> Codec.decoder Codec.BooleanToString  // true
```

5.4.2 Built-in codecs

The following codecs provide practical defaults for everyday work. They are all tolerant of common edge cases and aim to round-trip sensibly.

Identity

- `Codec.Identity : Codec<'a,'a>` — the no-op codec.

```
let sameA, sameB = Codec.Identity
let x = sameA 123   // 123
let y = sameB 123   // 123
```

UTF-8 strings and bytes

- `Codec.StringToBytes : Codec<string, byte[]>`
- `Codec.BytesToString : Codec<byte[], string>` (the reverse)

Semantics: UTF-8 encoding. Null-safe by design — null strings behave like empty text, and null byte arrays behave like empty buffers.

```
let toBytes, toText = Codec.StringToBytes

let b1 : byte[] = toBytes "hello"  // UTF-8 bytes
let s1 : string = toText b1        // "hello"

let emptyFromNullText : byte[] = toBytes null      // [||]
let emptyTextFromNull : string  = toText null      // ""
```

Numbers and booleans (string round-trips) Each numeric/boolean codec maps invalid input to a safe default when decoding from string.

- `Codec.Int32ToString   : Codec<int,    string>` — decode default: `0`
- `Codec.Int64ToString   : Codec<int64,  string>` — decode default: `0L`

- `Codec.DecimalToString : Codec<decimal,string>` — decode default: `0m`
- `Codec.BooleanToString : Codec<bool,   string>` — decode default: `false`

Reverse variants are provided for convenience:

- `Codec.StringToInt32    : Codec<string, int>`
- `Codec.StringToInt64    : Codec<string, int64>`
- `Codec.StringToDecimal  : Codec<string, decimal>`
- `Codec.StringToBoolean  : Codec<string, bool>`

```
// Int32
let toText32, fromText32 = Codec.Int32ToString
let ok  = fromText32 "123"     // 123
let bad = fromText32 "oops"    // 0 (default)

// Boolean
let toBoolText, fromBoolText = Codec.BooleanToString
let yes = fromBoolText "true"  // true
let no  = fromBoolText "nope"  // false (default)

// Decimal and Int64 similarly
let toTextDec, fromTextDec = Codec.DecimalToString
let amt = fromTextDec "9.99"   // 9.99m

let toText64, fromText64 = Codec.Int64ToString
let id  = fromText64 "18446744073709551615" // overflowy text -> 0L default
```

5.4.3 Composing to meet I/O boundaries

Codecs shine at process boundaries: serialize to bytes for storage, reverse for parsing when reading, and keep the round-trip logic centralized.

```
// Build a codec for decimal <-> bytes via string
let DecimalToBytes : Codec<decimal, byte[]> =
    Codec.DecimalToString
    |> Codec.concatenate Codec.StringToBytes

// Use it to persist and recover
let write : decimal -> byte[] = Codec.encoder DecimalToBytes
let read  : byte[]  -> decimal = Codec.decoder DecimalToBytes

let persisted = write 12.34m
let restored  = read persisted    // 12.34m
```

Notes

- Decoding is intentionally forgiving in the built-in codecs (e.g., invalid numeric text falls back to a neutral value). If you need strict parsing, consider validating prior to decode or wrapping with a stricter adapter.
- `Codec.reverse` and `Codec.concatenate` preserve the spirit of reversible transformations while making multi-step encodings ergonomic.

5.5 JsonValue

A small interop layer around `FSharp.Data.JsonValue` that focuses on:

- parsing and safe navigation (`tryParse`, `tryGet`/`get`, `properties`/`keys`/`values`)
- converting to/from compact text and bytes (`toString`/`ofString`, `toBytes`/`ofBytes`)
- bridging JSON to domain types (`toType<'T>`/`ofType<'T>`) with pragmatic converters for options, tuples, and discriminated unions
- composable codecs for common round-trips

5.5.1 Parse and navigate

```
open FSharp.Data
open Nata.Core.JsonValue

let text = """{ "name": "Ada", "age": 36, "tags": ["math","code"] }"""

// tolerant parse: None if invalid JSON
let parsed : JsonValue option = text |> JsonValue.tryParse

// required property (throws if missing)
let name : JsonValue =
    parsed
    |> Option.defaultWith (fun () -> failwith "invalid json")
    |> JsonValue.get "name"

// optional property
let nickname : JsonValue option =
    parsed
    |> Option.bind (JsonValue.tryGet "nickname")

// list structure
let keys   : string[]    = parsed.Value |> JsonValue.keys
let values : JsonValue[] = parsed.Value |> JsonValue.values
```

Notes

- `get` assumes the property exists and will throw if it doesn't; use `tryGet` when absence is acceptable.
- `toString` emits compact JSON (no pretty formatting) to keep byte-wise round-trips stable.

5.5.2 Strings and bytes

```
open Nata.Core.JsonValue

let json : JsonValue = JsonValue.ofString """{ "n": 3 }"""

// JsonValue <-> string
let asText : string    = json |> JsonValue.toString
let back   : JsonValue = asText |> JsonValue.ofString

// JsonValue <-> bytes (UTF-8)
let asBytes : byte[]    = json |> JsonValue.toBytes
let back2   : JsonValue = asBytes |> JsonValue.ofBytes
```

5.5.3 Types and converters

You can translate between JSON and domain types via `toType<'T>` and `ofType<'T>`. Converters are provided so that common F# shapes serialize naturally:

- **Options** serialize their contained value or null (e.g., `Some 3` → `3`, `None` → `null`).
- **Tuples** serialize as JSON arrays (e.g., `(1,"a")` → `[1,"a"]`).
- **Simple unions (no fields)** serialize as their case name by default. Unions with data appear in a canonical `{"Case": "...","Fields":[...]}` form.

```
open Nata.Core.JsonValue

// simple record round-trip
type User = { Id:int; Name:string }

let asUserJson : JsonValue = { Id=1; Name="Ada" } |> JsonValue.ofType
let userBack   : User      = asUserJson |> JsonValue.toType

// unions with data use the Case/Fields shape
type NumberOrText = | Number of int | Text of string

let sample : NumberOrText = Text "hello"
let jsonDU : JsonValue = sample |> JsonValue.ofType
let backDU : NumberOrText = jsonDU |> JsonValue.toType
```

5.5.4 Codecs: compose round-trips

The `JsonValue.Codec` submodule exposes codecs to assemble end-to-end pipelines:

- `JsonValueToString` / `StringToJsonValue`
- `JsonValueToBytes` / `BytesToJsonValue`
- `createTypeToJsonValue<'T>` / `createJsonValueToType<'T>`
- `createTypeToString<'T>` / `createStringToType<'T>`
- `createTypeToBytes<'T>` / `createBytesToType<'T>`

Guided examples

```
open Nata.Core
open Nata.Core.JsonValue

// 1) User preferences <-> string
type UserPreference = { Theme:string; ReceiveEmails:bool }

let prefToText, prefOfText : Codec<UserPreference, string> =
    JsonValue.Codec.createTypeToString()

let originalPref = { Theme = "dark"; ReceiveEmails = true }

let savedText    : string         = originalPref |> Codec.encoder prefToText
let restoredPref: UserPreference  = savedText    |> Codec.decoder prefToText

assert (restoredPref = originalPref)
```

```
// 2) Shopping cart line <-> bytes
type CartLine = { Sku:string; Qty:int; Price:decimal }

let lineToBytes, lineOfBytes : Codec<CartLine, byte[]> =
    JsonValue.Codec.createTypeToBytes()

let line    = { Sku="ABC-123"; Qty=2; Price=9.99m }
let payload : byte[] = line |> Codec.encoder lineToBytes
let back    : CartLine = payload |> Codec.decoder lineToBytes

assert (back = line)

// 3) Small JSON blobs in pipelines
//    (compact text and UTF-8 bytes)
let compact : string =
    """{ "id": 42, "name": "Ada" }"""
    |> JsonValue.ofString
    |> JsonValue.toString   // compact, stable

let blob : byte[] =
    """{ "ok": true }"""
    |> JsonValue.ofString
    |> JsonValue.toBytes

let asJson : JsonValue = blob |> JsonValue.ofBytes
```

Small example JSON for a discriminated union

```
// unions without fields become their case name; with fields they use Case/Fields
type Status = Ok | Error of string

let asJson : JsonValue = Error "invalid input" |> JsonValue.ofType
// {
//   "Case": "Error",
//   "Fields": [ "invalid input" ]
// }
```

5.6 DateTime

UTC-first helpers for epoch math, tolerant parsing, JSON interop, and composable codecs. The emphasis is on predictable round-trips and clear intent when moving across process and storage boundaries.

5.6.1 Overview

- Convert dates to and from Unix time:
  - `toUnixSeconds`/`ofUnixSeconds`
  - `toUnixMilliseconds`/`ofUnixMilliseconds`
- Normalize between time zones:
  - `toUtc`/`toLocal`, `ofOffset`
- Tolerant parsing:

    – `ofString : string -> DateTime option`

- JSON bridge:
    – `toJsonValue`/`ofJsonValue` using ISO 8601 round-trip format
- Resolution helpers to floor timestamps to a boundary:
    – `Resolution.year/month/day/hour/minute/second/ms`
- Codecs for composable round-trips:
    – DateTime ↔ Unix seconds/milliseconds, JSON, and compact JSON text

5.6.2 Guided examples

Unix time (seconds and milliseconds) Convert to integral time since the Unix epoch and back. Conversions run through UTC to avoid surprises.

```
open Nata.Core

let nowUtc : System.DateTime = System.DateTime.UtcNow

// seconds since 1970-01-01T00:00:00Z
let s : DateTime.Unix = DateTime.toUnixSeconds nowUtc
let backFromS : System.DateTime = DateTime.ofUnixSeconds s

// milliseconds since epoch
let ms : DateTime.Unix = DateTime.toUnixMilliseconds nowUtc
let backFromMs : System.DateTime = DateTime.ofUnixMilliseconds ms
```

Semantics: seconds/milliseconds are measured from the UTC epoch; integer casts use the usual truncation semantics for fractional values.

Offsets and local/UTC conversions Bridge from `DateTimeOffset`, and normalize explicitly when you cross boundaries where time zones matter.

```
let fromOffset (dto:System.DateTimeOffset) : System.DateTime =
    dto |> DateTime.ofOffset   // dto.UtcDateTime

let roundTripLocal (x:System.DateTime) : System.DateTime =
    x |> DateTime.toLocal |> DateTime.toUtc
```

Tolerant parsing, JSON interop Parse text when it's acceptable to fail with `None`. Encode as JSON using an ISO 8601 round-trip string.

```
open FSharp.Data
open Nata.Core.JsonValue

let parsed : System.DateTime option =
    "2023-04-05T12:34:56Z" |> DateTime.ofString

// DateTime <-> JsonValue
let asJson : JsonValue =
    System.DateTime(2023,4,5,12,34,56,System.DateTimeKind.Utc)
    |> DateTime.toJsonValue

let back : System.DateTime =
    asJson |> DateTime.ofJsonValue

// compact JSON text for transport/logging
let compact : string =
    asJson |> JsonValue.toString   // e.g., "\"2023-04-05T12:34:56.0000000Z\""
```

Notes: `toJsonValue` emits an ISO 8601 round-trip string (format specifier `"o"`) inside a JSON string; `JsonValue.toString` returns compact JSON (so quotes are included for JSON strings).

5.6.3 DateTime.Resolution

Floor timestamps to a chosen boundary while preserving the original `DateTimeKind` (UTC or Local).

```
let sample =
    System.DateTime(2023, 4, 5, 12, 34, 56, 789, System.DateTimeKind.Utc)

let atYear   = sample |> DateTime.Resolution.year   // 2023-01-01T00:00:00.000Z
let atMonth  = sample |> DateTime.Resolution.month  // 2023-04-01T00:00:00.000Z
let atDay    = sample |> DateTime.Resolution.day    // 2023-04-05T00:00:00.000Z
let atHour   = sample |> DateTime.Resolution.hour   // 2023-04-05T12:00:00.000Z
let atMinute = sample |> DateTime.Resolution.minute // 2023-04-05T12:34:00.000Z
let atSecond = sample |> DateTime.Resolution.second // 2023-04-05T12:34:56.000Z
let atMs     = sample |> DateTime.Resolution.ms     // 2023-04-05T12:34:56.789Z
```

Semantics: each function zeroes lower-order components (e.g., `minute` keeps year/month/day/hour and sets minute's lower parts to zero). These are useful for binning, windowing, and summarization.

5.6.4 DateTime.Codec

Composable codecs for common round-trips. These pair naturally with the general `Codec` and `JsonValue.Codec` helpers elsewhere in this document.

```
open Nata.Core
open Nata.Core.JsonValue

// DateTime <-> Unix seconds
let toSec, ofSec = DateTime.Codec.DateTimeToUnixSeconds
let secValue : DateTime.Unix = System.DateTime.UtcNow |> Codec.encoder toSec
let fromSec  : System.DateTime = secValue |> Codec.decoder toSec

// DateTime <-> Unix milliseconds
let toMs, ofMs = DateTime.Codec.DateTimeToUnixMilliseconds

// DateTime <-> JsonValue
let toJson, ofJson = DateTime.Codec.DateTimeToJson

// DateTime <-> compact JSON text
// (note: this is JSON text of a JSON string value, e.g. "\"...Z\"")
let toText, ofText = DateTime.Codec.DateTimeToString

let serialized : string =
    System.DateTime(2023,1,1,0,0,0,System.DateTimeKind.Utc)
    |> Codec.encoder toText

let restored : System.DateTime =
    serialized |> Codec.decoder toText
```

```
// End-to-end: DateTime <-> bytes (via JSON)
let DateTimeToBytes : Codec<System.DateTime, byte[]> =
    DateTime.Codec.DateTimeToJson
    |> Codec.concatenate JsonValue.Codec.JsonValueToBytes

let payload : byte[] =
    System.DateTime.UtcNow |> Codec.encoder DateTimeToBytes

let roundTripped : System.DateTime =
    payload |> Codec.decoder DateTimeToBytes
```

Notes: the text/bytes codecs intentionally use compact JSON for consistency with other JSON-based codecs; reverse them when you need the opposite direction, or concatenate with additional codecs to meet I/O boundaries cleanly.

5.7 Patterns

Small, purposeful active patterns make parsing-at-the-boundary feel natural. They turn tolerant conversions into readable matches so that pipelines stay declarative while handling the messy edges of real-world data.

5.7.1 Overview

The following partial patterns are available:

- `( |Nullable|_| ) : System.Nullable<'T> -> 'T option` — bridge from `Nullable<'T>` into `option`.
- `( |Boolean|_| ) : string -> bool option` — match text that parses to a boolean.
- `( |Decimal|_| ) : string -> decimal option` — match text that parses to a decimal.
- `( |Integer32|_| ) : string -> int option` — match text that parses to Int32.
- `( |Integer64|_| ) : string -> int64 option` — match text that parses to Int64.
- `( |DateTime|_| ) : string -> System.DateTime option` — match text that parses to a timestamp.
- `( |JsonValue|_| ) : string -> JsonValue option` — match text that parses to JSON.
- `( |AggregateException|_| ) : exn -> exn option` — unwrap an `AggregateException` to its inner exception.
- `( |AbsoluteUri|_| ) : string -> System.Uri option` — match absolute URIs.
- `( |RelativeUri|_| ) : string -> System.Uri option` — match relative URIs.
- `( |Uri|_| ) : string -> System.Uri option` — match URIs of either kind.

Why patterns? At ingestion points (files, queues, sockets, webhooks) you often receive untyped text. Active patterns let you route and normalize that input succinctly:

```
open Nata.Core

let handle (line:string) =
    match line with
    | JsonValue json ->
        // structured payload; proceed to domain decode
        json |> JsonValue.toString |> printfn "json=%s"
    | Integer64 id ->
        // numeric id in text form
        printfn "id=%d" id
    | AbsoluteUri uri ->
        // fetch or enqueue work for this resource
        printfn "uri=%O" uri
    | Boolean flag ->
        printfn "flag=%b" flag
    | _ ->
        printfn "unrecognized: %s" line
```

5.7.2 Guided examples

Batch lines into typed records Normalize a CSV-like feed where each field may be missing or malformed; only keep fully-parsed rows.

```
open Nata.Core

type Order = { UserId:int64; At:System.DateTime; Amount:decimal }

let tryOrder (line:string) : Order option =
    match line.Split(',') with
    | [| Integer64 userId; DateTime at; Decimal amount |] ->
        Some { UserId = userId; At = at; Amount = amount }
    | _ -> None

let parsed : seq<Order> =
    lines |> Seq.choose tryOrder
```

Nullable interop (databases, external SDKs) Turn `Nullable<'T>` into options without ceremony.

```
open Nata.Core

let discountOption : decimal option =
    match row.NullableDiscount with
    | Nullable d -> Some d
    | _ -> None
```

URIs in routing and partitioning Accept both relative and absolute targets, or insist on absolute when necessary.

```
open Nata.Core

let route (target:string) =
    match target with
    | AbsoluteUri uri -> sprintf "fetch-remote:%O" uri
    | RelativeUri uri -> sprintf "fetch-local:%O" uri
    | _               -> "drop"

[ "/inbox/a.json"; "https://example.com/x"; "???"]
|> List.map route
// ["fetch-local:/inbox/a.json"; "fetch-remote:https://example.com/x"; "drop"]
```

Surface the real error from tasks Unwrap `AggregateException` so handlers can match on the underlying exception type.

```
open Nata.Core

let handleError (e:exn) =
    match e with
    | AggregateException inner ->
        printfn "inner: %s" inner.Message
    | _ ->
        printfn "error: %s" e.Message
```

Streaming JSON with tolerant fallbacks Allow mixed payloads: compact JSON blob or a single numeric metric as text.

```
open Nata.Core
open Nata.Core.JsonValue

type Payload = Metric of decimal | Doc of JsonValue

let classify (s:string) : Payload option =
    match s with
    | JsonValue json -> Some (Doc json)
    | Decimal d      -> Some (Metric d)
    | _              -> None

let results : seq<Payload> =
    inputs |> Seq.choose classify
```

5.7.3 Notes

- All patterns are partial: a non-match simply falls through to the next case.
- Numeric, boolean, and date parsing rely on standard `TryParse` semantics.
- `JsonValue` uses a tolerant JSON parser; compact JSON text is convenient for logging and transport.
- URI patterns distinguish absolute vs. relative using `System.Uri.TryCreate` with the corresponding `UriKind`.
- Prefer patterns when routing by shape; prefer functions/codecs when you need explicit, composable transformations in a pipeline.

5.8 GZip

Lightweight helpers for working with gzip-compressed, line-oriented text. The emphasis is on streaming: you can write an unbounded sequence of lines to a compressed sink, or lazily read large archives without loading them into memory. This is a practical fit for log pipelines, newline-delimited JSON, CSV snapshots, and other append-friendly feeds.

Overview

- Text encoding: UTF-8, one logical line per element (`WriteLine`/`ReadLine`).
- Streaming semantics: `read` returns a lazy `seq<string>` that decompresses on demand.
- Back-pressure friendly: `write` flushes per line so long-running producers keep bounded buffers.
- Resource ownership: the underlying stream/file is disposed when the gzip layer is disposed (see notes).

5.8.1 GZip.Stream

Signatures

```
open System.IO

// write a sequence of lines into a target stream as gzip
val write : target:Stream -> lines:seq<string> -> unit

// lazily read lines from a gzip source stream
val read  : source:Stream -> seq<string>
```

Write: compress a live feed to a stream

```
open System
open System.IO
open Nata.Core

// imagine a long-running sequence (metrics, logs, ids, ...)
let lines : seq<string> =
    seq {
        for i in 1 .. 1_000_000 do
            yield sprintf "%d,%O" i DateTime.UtcNow
    }

use target = new MemoryStream() // any writable stream (file, network, memory)
GZip.Stream.write target lines

// bytes are gzip-compressed; rewind to inspect
target.Position <- 0L
let compressedSize = target.Length
```

Read: process large gzip content lazily

```
open System
open System.IO
open Nata.Core

// given a gzip payload in memory (or any readable stream),
// deserialize without loading all lines at once
let process (source:Stream) =
    for line in GZip.Stream.read source do
        // parse & handle each line
        match line.Split ',' with
        | [| count; timestamp |] -> ()
        | _ -> ()

// example with MemoryStream
use buffer = new MemoryStream()
GZip.Stream.write buffer (seq { for i in 1 .. 1000 -> string i })
buffer.Position <- 0L
process buffer
```

Newline-delimited JSON (NDJSON) pipelines

```
open System.IO
open Nata.Core
open Nata.Core.JsonValue

// write NDJSON to a gzip stream
let writeJson (target:Stream) (values:seq<'T>) =
    values
    |> Seq.map (JsonValue.ofType >> JsonValue.toString) // compact one-line JSON
    |> GZip.Stream.write target

// read NDJSON from a gzip stream
let readJson<'T> (source:Stream) : seq<'T> =
    source
    |> GZip.Stream.read
    |> Seq.choose (JsonValue.tryParse >> Option.map JsonValue.toType<'T>)
```

Semantics and notes:

- The returned sequence from `read` is lazy. Enumeration performs on-the-fly decompression; the gzip and underlying streams are disposed when enumeration completes or the enumerator is disposed.
- `write` flushes after each line to keep buffers small. This favors long-running producers; if you batch large amounts, consider shaping input into larger chunks if throughput is critical.
- Resource ownership: the gzip layer closes the underlying stream when done. If you need to keep the underlying stream open for further work, write to or read from a dedicated stream instance.

5.8.2 GZip.File

Signatures

```
open System.IO

// write lines to a .gz file (creates or overwrites from the beginning)
val write : target:FileInfo -> lines:seq<string> -> unit

// lazily read lines from a .gz file
val read  : source:FileInfo -> seq<string>
```

Write: archive a daily feed

```
open System
open System.IO
open Nata.Core

let today = DateTime.UtcNow.ToString("yyyyMMdd")
let file  = FileInfo(sprintf "events-%s.gz" today)

let events : seq<string> =
    seq {
        yield "started"
        for i in 1 .. 100_000 do yield sprintf "tick=%d" i
        yield "stopped"
    }

GZip.File.write file events
```

Read: stream from disk without loading it all

```
open System.IO
open Nata.Core

let scan (gz:FileInfo) =
    gz
    |> GZip.File.read
    |> Seq.filter (fun line -> line.Contains "ERROR")
    |> Seq.iter (printfn "error: %s")

scan (FileInfo "service.log.gz")
```

File behavior and notes:

- Reading is lazy: lines are decompressed as you iterate. This allows you to process multi-GB gzip files with bounded memory.
- Writing opens the file for writing from the beginning. If a file already exists and your new compressed output is shorter than the old content, trailing bytes may remain on disk. Prefer writing to a fresh path (e.g., rotate by date/time) or delete the existing file before writing when you need a clean overwrite.
- Both `write` and `read` manage their file streams internally and dispose them when complete.

Common scenarios

- Log compaction: persist operational logs as newline-delimited text into daily `.gz` files; read them back lazily for reporting.
- Data exchange: ship NDJSON snapshots to object storage; downstream jobs stream-decompress and decode line-by-line.
- Long-running streams: keep a process emitting metrics where each line is immediately flushed into a rolling `.gz` file to bound memory while retaining good compression.

5.9 BloomFilter

A Bloom filter is a compact, probabilistic set optimized for fast membership checks. It trades exactness for space and speed: it can yield false positives ("probably seen"), but never false negatives after you add an element. In practice, this is ideal for skip-lists, de-duplication, and pre-filtering high-volume streams.

Overview

- Immutability: operations return a new filter so you can keep prior snapshots when needed.
- Union: combine knowledge from multiple filters in O(n) over their bitsets (sizes must match).
- Presets: `small` (∼2M bits), `medium` (∼20M bits), `large` (∼200M bits) cover common scales.
- Common uses: de-duplicate events, suppress repeated requests, pre-filter expensive lookups, throttle idempotent sinks.

5.9.1 Quick start

```
open Nata.Core

// start with a medium filter by default
let f0 : BloomFilter = BloomFilter.empty

// record a few keys (any string works: ids, hashes, URIs, ...)
let f1 = f0 |> BloomFilter.add "user:ada"
let f2 = f1 |> BloomFilter.add "user:alan"

// membership: "probably seen"
let seenAda   : bool = BloomFilter.contains "user:ada" f2       // true
let seenGrace : bool = BloomFilter.contains "user:grace" f2     // false (very likely)
```

5.9.2 De-duplicate a live feed

When processing real-world streams (logs, webhooks, crawls), it's common to receive duplicates. A Bloom filter lets you drop repeats without holding the full set in memory.

```
open Nata.Core

// keep only the first occurrence of each id
let dedupe (ids:seq<string>) : seq<string> =
    let folder (f, kept) id =
        if BloomFilter.contains id f then f, kept
        else BloomFilter.add id f, id :: kept
    ids
    |> Seq.fold folder (BloomFilter.small, [])
    |> snd
    |> List.rev
```

You can slot this directly after a gzip reader or JSON parser to sanitize upstream feeds before heavier work.

```
open System.IO
open Nata.Core
open Nata.Core.GZip
open Nata.Core.JsonValue

// NDJSON in, unique ids out
let uniqueIdsFromGzip (source:Stream) : seq<string> =
    source
    |> GZip.Stream.read
    |> Seq.choose (JsonValue.tryParse
                   >> Option.bind (JsonValue.tryGet "id")
                   >> Option.map string)
    |> dedupe
```

5.9.3 Union across shards

If multiple workers or shards maintain their own filters, you can merge them to get the aggregate "seen" view. All inputs must use the same size preset.

```
open Nata.Core
```

```
let merged : BloomFilter =
    [ shardA; shardB; shardC ]    // each is a BloomFilter of the same size
    |> List.reduce BloomFilter.union

let likelySeen = BloomFilter.contains "object:42" merged
```

5.9.4 Composite keys and multi-checks

Model "seen today" or "seen this user and this session" by composing stable string keys. Use `containsAll` for conjunction and `containsAny` for disjunction.

```
open Nata.Core

// key by day + user for a coarse time window
let keyOf (at:System.DateTime) (user:string) =
    sprintf "%s|%s" (at.ToString "yyyyMMdd") user

let seenToday (f:BloomFilter) (user:string) (at:System.DateTime) : bool =
    BloomFilter.contains (keyOf at user) f

// multi-checks: all or any tokens present
let hasAll  (f:BloomFilter) (tokens:string list) : bool = BloomFilter.containsAll tokens f
let hasAny  (f:BloomFilter) (tokens:string list) : bool = BloomFilter.containsAny tokens f
```

5.9.5 Rolling windows

Instead of exact expiry, rotate filters on a cadence (hourly/daily) to bound false positives over time. This works well when combined with upstream time-normalization helpers.

```
open Nata.Core

// create a fresh filter at the start of each day; carry on as usual
let freshFilterFor (at:System.DateTime) : BloomFilter =
    let yyyyMMdd = at.ToString "yyyyMMdd"
    // e.g., use yyyyMMdd as a key to pick/reset the active filter in storage
    BloomFilter.medium
```

Notes

- False positives are possible by design; false negatives are not (once added).
- If false positives become frequent, choose a larger preset or rotate filters more often.
- `union` requires filters created with the same size. Mixing sizes is not supported.
- Keys are plain strings: prefer stable, canonical encodings (e.g., lowercase IDs, normalized URIs).

5.10 SHA256

A compact, fully-immutable implementation of SHA-256 for day-to-day functional work. It favors simple, predictable composition: inputs are plain bytes or UTF-8 text; outputs are either raw digest bytes or lowercase hex strings. There is no internal state, no disposable resources, no hidden I/O — which makes it pleasant to use in pipelines, tests, and reference comparisons.

Signatures

```
open Nata.Core

// pure digests (no I/O, no mutation)
val hashBytes         : byte[]  -> byte[]   // 32-byte digest
val hash              : byte[]  -> string   // 64-char hex (lowercase)
val hashUTF8Bytes     : string  -> byte[]   // UTF-8 -> digest bytes
val hashUTF8          : string  -> string   // UTF-8 -> hex

// platform-backed comparison helpers (useful for verification)
val referenceBytes      : byte[]  -> byte[]
val reference           : byte[]  -> string
val referenceUTF8Bytes : string  -> byte[]
val referenceUTF8       : string  -> string
```

Quick start

```
open System.Text
open Nata.Core

// hash raw bytes
let digestHex : string =
    "abc" |> Encoding.UTF8.GetBytes |> SHA256.hash

// convenient UTF-8 helpers
let fox : string =
    "The quick brown fox jumps over the lazy dog"
    |> SHA256.hashUTF8
// "d7a8fbb3...c9e592"

// verify against the platform reference
let same : bool =
    let bytes = Encoding.UTF8.GetBytes "hello"
    (SHA256.hash bytes) = (SHA256.reference bytes)
```

Why an immutable implementation? Immutability keeps the surface area honest: the same input yields the same output every time. With no resources to manage—no streams to dispose, no hidden buffers—the functions drop cleanly into folds, maps, and parallel evaluations. This makes them ideal for verification as well: you can place a pure digest beside a platform digest and reason about differences without incidental state. As a bonus, the implementation is a readable reference for the algorithm itself, a portable baseline you can trust when threading hashes through larger pipelines.

Notes

- Hex output is lower-case; digest bytes are exactly 32 bytes.
- `hashUTF8`/`hashUTF8Bytes` always use UTF-8 for text input.
- For maximum throughput on large single buffers, prefer the platform implementation; for composable, audit-friendly pipelines and tests, the immutable implementation shines.
- Stable input representations matter: prefer canonical JSON (compact, ordered) or explicit binary layouts when the digest must be an id.

6 Nata.Fun

6.1 JsonPath

Real-world data almost always arrives as JSON: telemetry, logs, events, API responses, and so on. In practice, we rarely need the whole document every time. Instead, we reach into it and pull out just the parts we need to compute, route, enrich, or persist. Nata provides two complementary ways to do this:

- A string-based JsonPath API for selecting tokens across many JSON documents (excellent for streams).
- A JsonValue-based API for selecting values inside a parsed FSharp.Data `JsonValue` (excellent for in-memory traversals and slicing).

In other words: if you have a sequence of JSON strings, use the string-based helpers; if you have a parsed `JsonValue`, use the JsonValue helpers.

Query syntax at a glance

Pattern	Meaning
`.name`	Exact property descent
`..name`	Recursive descent (match here and deeper)
`"*"`	A wildcard property name
`[*]`	All array elements
`[0,2,-1]`	Literal indices (negatives count from the end)
`[start:finish:step]`	Slice; start/finish optional; step defaults to 1
`..lines[*].sku`	Combine segments to traverse structure

6.1.1 Selecting from strings with JsonPath

When you have a sequence of JSON strings—say from files, HTTP responses, a message bus, or a queue—you can query each document with a JsonPath expression and collect the results.

At the core is `strings`:

```
open Nata.Fun.JsonPath

// Select all matching tokens as strings for a single json document
let names : string list =
    Query.strings "$.customers[*].name" json
```

Some very common patterns:
- First or default: return a single value (or empty string) if present.

```
let id : string = Query.first "$.customer.id" json
```

- Fields: lift multiple paths together in order, preserving positional correspondence.

```
let keys = [ "$.orderId"; "$.customer.id"; "$.total" ]
let row : string list = Query.fields keys json
```

- Streams: collect across a sequence of JSON documents.

```
// Suppose 'events' is seq<string> of incoming JSON messages
let allSkus : string seq =
    Query.find "$..lines[*].sku" events
```

- Distinct: deduplicate values (useful for keys, tags, or categories).

```
let distinctUserIds : string seq =
    events |> Query.find "$.user.id" |> Seq.distinct
// or equivalently:
let distinctUserIds2 : string seq =
    events |> Query.findDistinct "$.user.id"
```

- Tabular extraction: project many values per document in a stable order.

```
let cols = [ "$.ts"; "$.level"; "$.message" ]
let rows : string list seq =
    Query.findFields cols events // seq of [ ts; level; message ]

let csvLines : string seq =
    Query.findFieldsCsv cols events // seq of "ts,level,message"
```

Why this matters: streams of operational data regularly need light shaping before they can be folded, joined, or written to storage. These helpers are zero-ceremony ways to normalize fields, pick identifiers, and wire up quick dashboards or sinks.

Typical uses

- Pull the first available value: `Query.first "\$.payload.correlationId" json`
- Extract every occurrence across nested arrays: `Query.find "\$..metrics[*].name" stream`
- Build CSV lines on the fly for archival or quick inspection: `Query.findFieldsCsv [ "\$.t"; "\$.id"; "\$.value"`

6.1.2 Selecting from JsonValue with structural traversal

Sometimes your pipeline already holds parsed `JsonValue` values—perhaps due to upstream validation, schema checks, or prior projections. Using the syntax summarized above, the JsonValue module provides a structural, automata-driven traversal well-suited to recursive descent and array slicing.

Core functions:

```
open Nata.Fun.JsonPath
open Nata.Fun.JsonPath.JsonValue
open FSharp.Data

let sample = JsonValue.Parse """
{
  "orders": [
    { "id": "A1", "lines": [ { "sku":"X", "qty":2 }, { "sku":"Y", "qty":1 } ] },
    { "id": "B2", "lines": [ { "sku":"X", "qty":3 }, { "sku":"Z", "qty":4 } ] }
  ]
}
"""

// Collect all matches as a sequence of JsonValue
let skus : JsonValue seq =
    JsonValue.findSeq "..lines[*].sku" sample

// Get just the first match (throws if none)
let firstOrderId : JsonValue =
    JsonValue.find ".orders[0].id" sample

// Get as a list (useful for tests or small results)
```

```
let lastLinesPerOrder : JsonValue list =
    JsonValue.findList ".orders[*].lines[-1]" sample

// Try-find (safe option)
let maybeThirdOrder : JsonValue option =
    JsonValue.tryFind ".orders[2]" sample
```

This is particularly helpful when you want to use slicing semantics that are natural for arrays:

- Last element: `[-1]`
- First N elements: `[0:N]`
- Every other element: `[::2]`
- A window: `[start:finish:step]`

Typical uses

- Fetch the last line item per order: `JsonValue.findSeq ".orders[*].lines[-1]" json`
- Recursively pick all occurrences of a property: `JsonValue.findSeq "..correlationId" json`
- Slice efficiently without materializing entire arrays: `JsonValue.findSeq ".frames[100:200:5]" json`

6.1.3 Choosing between the two

- You have a stream of raw JSON strings: favor `Query.find` and friends (`strings`, `first`, `fields`, `findFields`, `findFieldsCsv`). These compose naturally with `Seq` and work well for quick projections and exports.
- You already have `JsonValue`: favor `JsonValue.findSeq`/`find`/`findList`/`tryFind`, especially when you want recursive descent or array slicing/negative indexing behavior.

Both approaches complement the broader Nata patterns: extract what you need, project or fold it deterministically, and keep your pipelines small, pure, and observable.