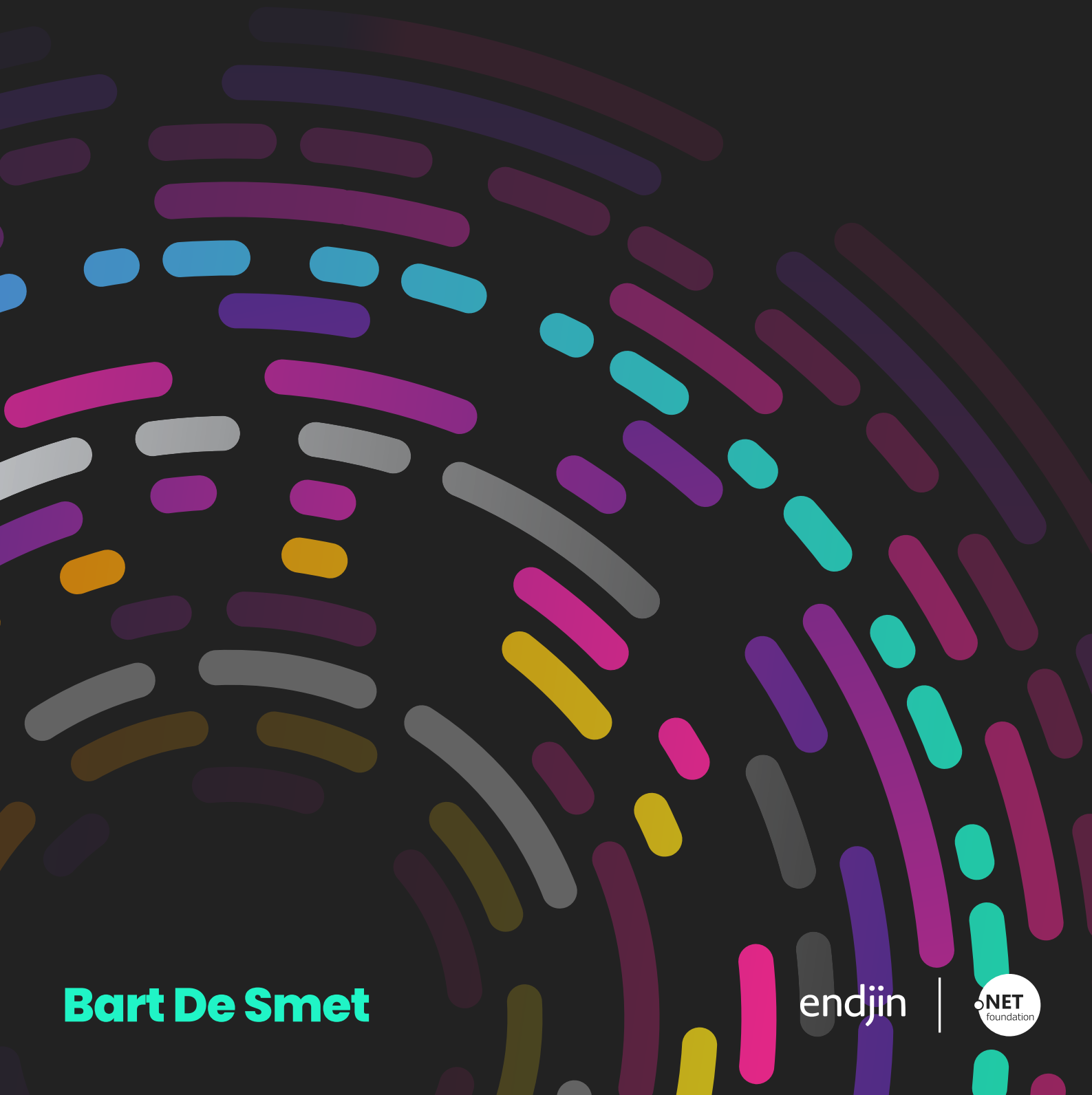
A LITTLE HISTORY OF REACTOR

Early Access Preview: May 2021



Bart De Smet

endjin | .NET foundation

Table of Contents

About the Author

Bart De Smet is a Principal Software Development Engineer at Microsoft. Bart was one of the original Reactive Extensions (Rx) team members and was the lead developer for Rx 2.0. Later he created the Reaqtor technology, serving as the chief architect of the core components, while helping various teams to operate the technology at massive scale.

He joined the WPF team at Microsoft in 2007, right after graduating in Belgium. Prior to joining Microsoft, Bart was an MVP for C# for four years. He also wrote some books, used to be addicted to blogging, authored some Pluralsight courses on C# internals, is a popular speaker at technology conferences, and frequently interviewed on Channel9. Currently, Bart is actively working on incubation projects in the space of edge computing. You can read his Reaqtor blog posts on reaqtive.net.

About the Maintainers

Ian Griffiths

Ian has worked in various aspects of computing, including computer networking, embedded real-time systems, broadcast television systems, medical imaging, and all forms of cloud computing. Ian is a Technical Fellow at endjin, and Microsoft MVP in Developer Technologies. He is the author of O'Reilly's Programming C# 8.0, and has written Pluralsight courses on WPF (and here) and the TPL. Technology brings him joy.

You can follow Ian on Twitter at @idg10 and read his Reaqtor blog posts on reaqtive.net.

Carmel Eve

Carmel is a software engineer at endjin, LinkedIn Learning instructor, and STEM ambassador who specialises in delivering cloud-first highly-performant serverless and big data architectures. She is an active blogger covering a huge range of topics, including deconstructing Rx operators and mental well-being and managing remote working. Carmel has also presented at NDC, APISpecs and SQLBits conferences covering topics from reactive big-data processing to secure Azure architectures.

Carmel is passionate about diversity and inclusivity in tech. She is a STEM ambassador in her local community and is taking part in a local mentorship scheme. Through this work she hopes to be a part of positive change in the industry. Carmel won "Apprentice Engineer of the Year" at the Computing Rising Star Awards 2019.

You can follow Carmel on Twitter at @carmel_eve and read her Reaqtor blog posts on reaqtive.net.

Howard van Rooijen

Howard is a technologist with over 20 years of experience. He's an entrepreneur, investor, the co-founder of endjin, and has been an Microsoft Azure MVP since 2016.

You can follow Howard on Twitter at @HowardvRooijen and read his Reaqtor blog posts on reaqtive.net.

About the Project Sponsors

Reaqtor was created by Microsoft, who gifted it to the .NET Foundation, where Reaqtor is available under an MIT open source license.

The project is also sponsored by endjin, a UK based technology consultancy, founded in 2010. Endjin is a Microsoft Gold Partner for Cloud Platform, Data Platform, Data Analytics, DevOps, Power BI Partner, and .NET Foundation Corporate Sponsor. Endjin is a passionate supporter of the Microsoft ecosystem, producing two free weekly newsletters; Azure Weekly and Power BI Weekly, and a very popular blog. Endjin also supports and maintains a number of open source projects, some of which it created, some of which it has adopted to ensure continued value to the .NET community.

You can follow endjin on Twitter at @endjin.

Foreword

Endjin's journey with Reaqtor started in 2016, with a hot cup of tea and a simple question:

> "What single action could you take which would deliver the biggest positive impact on customer satisfaction?"

The question popped up in a Brain Trust session with Ilario Corna, Head of Infrastructure, Content & Operations at Talk Talk (a UK Telco). The organisation had just rolled out an objective to become the UK's "most recommended provider". As part of this strategic initiative, we had designed and implemented an Azure based solution capable of ingesting and analysing over 200 million network telemetry events per day for anomalous behaviour. Feeling buoyed by success we were wondering "which hard problem could we tackle next?"

Ilario had the answer immediately. "Oh, that's easy" he said, "I would send a field engineer out to each customer's house, run diagnostics on their broadband connection and hardware, upgrade their firmware and implement any other tweaks required. But with over 1 million customers that's not commercially feasible to implement."

After taking a sip of tea and pondering for a few moments I responded "So we have a solution, but we need to find a way to implement and scale it several orders of magnitude more cheaply. If I were in your shoes and could wave a magic wand, I'd want to be able to say something like 'Cortana, monitor the Quality of Service for each of my customer's connections, and if it's below our SLA, then run a trouble-shooter to diagnose why and then apply a number of automatic remediation strategies'. Wouldn't that be a great way to manage the network?"

He responded with a chuckle, "Sure would! When can I have it?!?" We both laughed then, because it sounded like a far-off sci-fi fantasy.

On the train home, that part of the conversation kept coming back to me. Why couldn't we create some type of digital agent that could monitor not only the customer's device, but every part of the network infrastructure that led to the customers home? The nuance was that unlike traditional "big data" problems, whereby you need to reduce vast amounts of data into a format a human can comprehend, this type of problem is more akin to signals analysis: you need to process the full fidelity raw data in a stateful way.

If you have 1 million customers, you need demultiplex those 200 million messages into 1 million queries, each query will need to look at the message and ask "does this apply to me?" if so it would process the event and update its quality of service metric, evaluate if that had met a threshold and if so, trigger a new "threshold exceeded" event. Articulating the problem this way sounded a bit like the Actor Model, or perhaps an Rx query.

A few years previously we had worked on an DevOps project for a major UK online retailer. We used Rx to aggregate and disaggregate semantic logging events from all the servers in the data centre. The Rx query we implemented was simple, elegant, and incredibly powerful; a testament to the design of Rx, and a reason we're huge fans of it at endjin.

I wondered if it would be possible to run 1 million Rx queries concurrently, and remembered Bart De Smet has presented a session at NDC 2015 about Cloud Event Processing the previous year, so I found & watched the recording. In that talk he mentioned that he was working in the Bing organisation on an evolution of Rx which powered Cortana experiences, and this involved stateful, durable, high density Rx queries. This sounded very promising indeed!

While researching to see if this technology had been open sourced, I came across a Microsoft case study about "Bing Cortana" (unfortunately no longer public) that mentioned a technology called Reactor, which was used to evaluate 500 million queries per second. If this was the same technology, we only needed 1/500th of that capacity in order to solve our broadband Quality-of-Service monitoring scenario.

I reached out to David Goon, who was our Partner Manager at Microsoft, explained the situation and asked "do you think it would within the realms of possibility for us to get access to this technology?" He responded "It's a really compelling use case, and I can see there being lots more across other sectors. Let me reach out to Bart De Smet and see what he says. We can only try." Within a couple of weeks, we had an hour long call with Bart where we took him through the scenario we had elaborated, and asked if he thought Reactor would be a viable solution. He said that it seemed to be a good fit, and we agreed to start collaborating on a proof of concept.

It's hard to convey what an absolute honour and privilege it has been to work with Bart, and I will be forever grateful for the amount of his incredibly valuable time he dedicated to this endeavour. It goes without saying that without him, absolutely none of what follows would have been possible. Reaqtor is Bart's baby; we've just had the privilege of delivering it into the world.

Once Bart provided us with access to some sample Reactor code, Mike Larah and I paired on a "simple scenario"; an entirely in-memory version of Reactor hosted in a Console App, based on the NDC talk code sample Bart shared with us. We wanted to see if we could solve the broadband anomaly detection problem by processing a stream of RADIUS network telemetry and detecting the specific events that indicate there were problems in the broadband network. In this prototype we modelled the network as a graph, which allowed us to determine if an issue was localised to the customer's home or was occurring at a local (an outage for a street) or regional node level (an outage for a town). It worked, and we were fizzing with excitement.

One of the key points you need to understand is that Reactor is orders of magnitude bigger than Reactive Extensions with over 166 projects, and 90 NuGet packages, and yet it is still a framework, not a platform. While state, durability and reliability are defined in the design of the framework, the

implementation is missing, as this is tied to the hosting environment.

Reactor was born into Bing and adopted by M365 and other product teams. Bart had done a fantastic job of ensuring a clean separation of Reactor from the hosting environment, but it meant that we had to tackle the thorny problem of building our own reliable hosting platform, state store, ingress & egress adapters, management & control plane, and workbench for talking to Reactor. We had a mountain to climb.

When it comes to solving hard problems, Ian Griffiths has always been our first port of call; he's been an endjin associate since we founded the company. I knew he was very passionate about Rx as he'd included a chapter about the subject in his Programming C# series of books for O'Reilly, and we'd had many conversations about it. Over a few months we implemented an initial spike of a reliable hosting platform, and had an absolute blast while doing it. Shortly afterwards Ian joined endjin full time as our first Technical Fellow, so that he could continue to work on the endeavour.

Over the next few years, we built several technical spikes exploring different facets of the Reactor technology; we embedded ML.NET models into Reaqtor queries, we created a proof-of-concept anomaly detection temporal query (using virtual time) that could correctly identify broadband quality of service events that indicate outages; processing a days' worth of telemetry (around 40GB) in under 90 seconds. The ability to do "what-if" simulations over historic datasets, which we could then flip a switch and turn into live queries running against realtime data blew us away. I want to express my thanks to Ben Dyer and Alex KeySmith for providing us with the anonymised dataset for this proof of concept - they were early believers.

In May 2018 Matthew Adams, my co-founder at endjin, & I flew over to Seattle for the Microsoft Build conference and we managed to spend our last day of the trip with Bart. In the morning Bart told us the story of how Reactor came to be (this is covered in detail in the "A Little History of Reaqtor" ebook you can download for free on the homepage). Over a fantastic lunch we waxed lyrical about the future of data processing, and Bart talked about his vision for `IComputationProcessing` (more about this in the ebook!) and then in the afternoon we talked about whether we could open source Reactor and what that process would entail.

Carmel Eve, who had interned with us the previous year, re-joined endjin on our Apprenticeship Programme. Ian and Carmel started exploring Reactor with other proof of concepts covering different customer scenarios we had identified. Throughout this time Carmel documented her learnings about Rx and writing high performance C# in popular blog post series. The more we investigated Reactor, the more impressed we became.

In March 2019 I attended the MVP Summit in Redmond, and I bumped into Jon Galloway, who at the time was the Executive Director of the .NET Foundation. I gave him as brief a Reactor elevator pitch as I could, and said that endjin would love to become .NET Foundation Corporate Sponsors, as we wanted to demonstrate our commitment to the .NET ecosystem, while we also collaborated with Bart to create

whitepapers and a business case in order to justify the investment from Microsoft to carry out the legal processes required to open source Reactor to the .NET Foundation under The MIT License.

By the end of the year we'd filled out the paperwork, paid our sponsorship fee, and were featured on the .NET Foundation homepage alongside: AWS, DevExpress, Microsoft, Octopus Deploy, Telerik, Uno Platform and VMWare.

At this point, we had to face the problem that we had been avoiding for the past 3 years; what do we call it? "Reactor" as a name is too overused. Microsoft Reactor is the name of the Microsoft Community meet up venues around the world. Project Reactor is the name of a JVM project from VMWare, inspired by Rx, and based on Reactive Streams. React is the hugely popular reactive user interface library from Facebook. One of the key concepts that powers Reactor is `IQbservable`. We've always loved this weirdly named interface (partly due to how Bart pronounces it - `I-Cub-servable`). Carmel came up with the suggestion that we include the "Q" in the name and as soon as Reaqtor was written down, it received unanimous agreement.

This also neatly solved two other naming problems we faced. Reaqtor consists of three conceptual layers; at the bottom a set of reusable components (that can be used independently of Reaqtor) which provide compiler, JSON and LINQ, and high performance / low memory extensions and utilities, which we named Nuqleon. The middle layer contains reactive primitives that evolve many of the concepts found in Rx to enable reliable, stateful and distributed reactive processing, which we named Reaqtive. And finally the top level consists of platform level services for building reliable, stateful, distributed reactive solutions, called Reaqtor.

We started to collaborate with the .NET Foundation to onboard the project. Special thanks to Claire Novotny who helped us set up the DevOps processes on the .NET Foundation infrastructure and all the other bits of administrivia required to prepare the project to be released to the public.

We also worked with Rodney Littles and the .NET Technical Steering Group to elaborate an RFC to modernise the Expression Tree subsystem, as it has not been invested in for many years and does not have parity with the latest language features. These improvements are fundamental to the future of Reaqtor. Fortunately there seems to be some progress in this area as the Entity Framework team would also like this modernisation to occur.

Part of the problem with Expression Trees is that they are one of the most complex and least well documented parts of .NET; they are also perceived only to exist to power LINQ (and LINQ Providers). As you'll see with Reaqtor (an in particular the Nuqleon layer, which is where the Expression Tree & Bonsai subsystem lives), Expression Trees really are one of .NET super powers and enable mind-bending meta-programming scenarios.

One of the final puzzle pieces was how we could create a great documentation experience for the community. Ian has a long history in teaching programming; DevelopMentor, Pluralsight and his

O'Reilly books. We've had many conversations about what a good learning environment looks like: the ability to tell a cohesive narrative including text, images, videos and code samples that you can execute in context (to eliminate task switching). Microsoft Docs have done an excellent job in this respect, but how can you offer a similar experience when you are a small open source project?

We initially investigated Try .NET which lead us to talking to Jon Sequeira, Dr Diego Colombo, Maria Naggaga and Brett Forsgren. Initial conversations quickly turned to their current project .NET Interactive, which seemed perfect for our needs, not only for interactive documentation but also as an IDE for writing, testing and running Reaqtor queries.

We soon realised that .NET Interactive was designed around a request/response (REPL) model, but we'd really need to support long running in-process and out of process reactive queries. We collaborated with the team on a design for a client-side command pattern that would allow us to define custom commands and send them between the kernels. Ian implemented the feature and the PR was merged into the .NET Interactive codebase, allowing us to create great documentation and interesting demos, that you can use to understand Reaqtor, Reaqtive & Nuqleon.

The final significant contributor to the project is Felix Corke who designed the branding and creative assets which we have made available under a Creative Commons Attribution Share Alike 4.0 International license, so that the community can use them in blog posts, videos and presentations about Reaqtor. We have also created a Community Presentations repository containing branded, pre-canned presentations and demos using the existing interactive notebooks, so that you can easily do a presentation or lightning talk at your local .NET User Group.

Our journey with Reaqtor only covers a short period in its overall history. In this book Bart De Smet tells the full story… starting in 2005. It is an absolutely fascinating account, and I hope by the time you finish reading it, you'll understand why we believe Reaqtor is the most exciting technology in the .NET ecosystem, and that it's a game changer.

Howard van Rooijen, co-founder, endjin. May 2021.

Introduction

The history that led to the creation of Reaqtor is quite long. In this book, I'll try to present the technology's history and background, which can help to understand the context and some of the design decisions that were made.

Consider this book to be the "Old Testament of Reaqtor according to Bart", compiled using the best of his abilities to recollect a long history. We'll assume the reader is somewhat familiar with the Reaqtor technology and reactive programming in general.

Bart De Smet, Principal Software Development Engineer, Microsoft. May 2021.

What is Reaqtor?

Reaqtor is a distributed reactive event processing system built using the constructs and algebraic properties of Reactive Extensions (Rx). It supports the submission of standing queries (known as *subscriptions*) to a service where such queries are reliably executed in a distributed manner. Subscriptions consist of *observables* which act as data sources of *events*, and *observers* which act as sinks for *events*.

Commonly used observables include timers and *streams*; commonly used observers include actions that perform HTTP calls to external services, that insert events into stores, or publish events to *streams*. A rich set of *query operators* can be used to transform data sources by applying constructs such as filtering, projection, aggregation, merging, joining, windowing, etc.

The first development of Reaqtor started around 2013 in the Online Services Division at Microsoft, which also is responsible for the Bing search engine, MSN services, etc. Reaqtor has powered a variety of scenarios for first party customers such as Cortana, MSN, Office 365, and more. Some of the Reaqtor service deployment handle upwards of billions of standing event processing queries processing thousands of events per second, often using relatively small clusters with a few 100s of physical nodes.

One of the biggest strengths of Reaqtor is its support for high-density adhoc event processing, where billions of unique queries are processing events in near real-time, all based on the simple but powerful algebra of observable sequences and Rx-style query operators. Its extensibility with custom observable and observer implementations enables bridging with other "reactive" cloud services.

Language Integrated Query (LINQ) as a precursor

The grandfather of Reaqtor is LINQ, which dates back to 2005-2007. Designed by the C# and Visual Basic language teams, the goal of LINQ is to solve the "impedance mismatch" between object-oriented programming languages and a variety of queryable data sources, including in-memory collections, XML documents, relational databases, and more. While querying in-memory collections is relatively easy by using language constructs such as `foreach` loops, querying other types of data sources such as relational databases involves embedding foreign languages such as SQL into the code, often as string literals which don't benefit from IntelliSense, statement completion, type checking, etc.

```
var command = connection.CreateCommand("SELECT ProductName, UnitPrice FROM
↪  Products WHERE UnitPrice > 49.95");
var reader = command.ExecuteReader();
while (reader.Read())
{
    var name = (string)reader["ProductName"];
    var price = (decimal)reader["UnitPrice"];
```

```
    // ...
}
```

Further complexity with existing data access methods includes the need for object/relational (O/R) mapping technologies, programming models that aren't compatible with the notion of *enumerable* sequences which integrate well with language constructs such as `foreach`, and the different approaches needed to blend data from different sources together (e.g. joining between data in a relational database and data in an XML document).

These realizations led to the creation of Language Integrated Query (LINQ) which itself was based on the Comega research effort in 2004. Both C# 3.0 and Visual Basic 9 received *query expression* syntax integrated into the language, much like query comprehensions in Haskell, F#, and other functional programming languages.

```
var res = from p in ctx.GetTable<Product>("Products")
          where p.UnitPrice > 49.95m
          select new { Name = p.ProductName, Price = p.UnitPrice };

foreach (var p in res)
{
    // ...
}
```

Besides query expression syntax (`from x in xs ...`), C# 3.0 and Visual Basic 9 introduced a variety of related language features, including local variable type inference (`var`), anonymous types (`new { x = 1 }`), lambda expressions (`x => x * 2`), extension methods, etc. Query expressions translate to more primitive language constructs, including these new ones:

```
var res = ctx.GetTable<Product>("Products")
              .Where(p => p.UnitPrice > 49.95m)
              .Select(p => new {
                  Name = p.ProductName,
                  Price = p.UnitPrice
              });
```

Query expressions can be written against in-memory collections implementing `IEnumerable<T>` or external data sources implementing `IQueryable<T>`. In the former case, query operators such as `Where` and `Select` are implemented using iterators (`yield return x`) and execute in memory as an object graph of lazily evaluated enumerable sequences backed by state machines. In the latter case, query operators such as `Where` and `Select` quote their lambda expression parameters into *expression tree* data structures in order to produce an expression tree that represents the entire user intent. Upon triggering enumeration (e.g. using `foreach`), this expression tree is analyzed by the

underlying query provider library, and translated to a target language such as T-SQL in order to execute the query.

> Query expressions are the primary programming model for Reaqtor today, when used from C#, Visual Basic, or F#. Expression trees are used as the underlying representation that gets normalized and serialized over the wire between clients and the Reaqtor service, as well as across different micro-services inside Reaqtor.

Expression trees are effectively quotations of C# and Visual Basic language constructs into runtime data structures in the `System.Linq.Expressions` namespace. These data structures can be analyzed at runtime for purposes of optimization, translation, and compilation.

```
{
    // Lambda conversion to a delegate type
    Func<int, int> f = x => x * 2;

    // Can be invoked
    Assert.AreEqual(42, f(21));
}

{
    // Lambda conversion to an expression tree type
    Expression<Func<int, int>> f = x => x * 2;

    // Can be traversed
    Assert.AreEqual(2,
↪   (int)((ConstantExpression)(((BinaryExpression)x.Body).Right)).Value);

    // Can be compiled and invoked
    Func<int, int> d = f.Compile();
    Assert.AreEqual(42, d(21));
}
```

Key people involved with the creation of LINQ include Anders Hejlsberg, Don Box, and Erik Meijer. For more information, see the original LINQ: .NET Language-Integrated Query paper, and the Erik Meijer on LINQ talk.

Cloud Programmability Team

After his involvement with C# and VB language design and co-creating LINQ, Erik Meijer moved to the Live Labs team and later the SQL Server organization to form the “Cloud Programmability Team”

somewhere around 2007, with Brian Beckman.

The goal of this team was to look at programming models for cloud computing, right at the start of Microsoft's first steps into cloud computing in the Ray Ozzie days. One of the main projects was "Volta" and provided a tier-splitting approach to "lift and shift" (long before this was established terminology) typical line-of-business (LOB) applications to the cloud. The basic philosophy was to enable developers to annotate their .NET applications - regardless of the language they were written in - using tier-splitting custom attributes on classes and members, indicating where code should run. This technology was similar in spirit to Google Web Toolkit (GWT).

> One trace of the small detour through the Live Labs organization remains. The logo of Reactive Extensions (Rx) was inherited from the "Volta" project and depicts an electric eel, as a reference to Allesandro Volta.

An example of using Volta is shown below:

```
// presentation layer, e.g. user/password inputs
[RunOnClient]
class LoginForm : Form
{
}

// business logic layer, e.g. login attempt count and lockout
[RunOnServer]
class AuthenticationProvider
{
}

// data access layer, e.g. stored procs
[RunOnDatabase]
class ContosoDatabase
{
}
```

In the example shown above, the code in the `LoginForm` should run on the client, which could be a rich UI application or a web browser. To achieve this, the "Volta" project supported splitting the original monolithic application into smaller assemblies containing the tier-split IL code, which can then be retargeted to a client-compatible execution environment. When targeting rich desktop clients for the cloud-lifted application, it'd simply continue to use Windows Forms. When targeting in-browser experiences with plug-ins, it'd target Silverlight. When targeting native browser experiences, it'd transpile the IL code to JavaScript using the IL2JS technology built by the team. Code annotated with `RunOnServer` would run largely as-is in the cloud, targeting a worker role, while code annotated with [RunOnDatabase] would get split off to run in the database, targeting SQL CLR.

> Tier-splitting of applications (which express developer intent at a macroscopic scale) into little pieces for distributed execution by means of formal analysis of and transformations on the input program form the basis of a lot of work that succeeds the Volta project. In particular, the notion of delegation of expression trees in the context of Reaqtor is very akin to the concept of tier-splitting, albeit implemented in quite a different way.

To support these bold ambitions, the team built a number of technologies that were interconnected:

- `IL2JS`, an MSIL to JavaScript transpiler. By transpiling from MSIL, all languages on the CLR can be targeted. This approach differed from Script#, which was compiling C# to JavaScript. In retrospect, both approaches suffered from the semantic gap between JavaScript and languages on the CLR and JVM. This said, the idea of running managed code in a web environment on the client has carried forward. A recent technology in this space is Blazor which can benefit from newer functionality such as WebAssembly.
- A compiler for a language (whose name was never publicly released) that was a superset of JavaScript adding classes, modules, and static typing. This was meant to become an alternative compilation target for tier splitting, reducing the complexities involved with transpiling down from rich OO languages to a prototype-based type system in JavaScript. This effort is now superseded by TypeScript, though there is no direct link between the two projects. In retrospect, it was an idea ahead of its time; today, we're seeing a lot of attempt to beef up JavaScript (ClojureScript, CoffeeScript, TypeScript, etc.) or replace JavaScript (most notably Dart).
- `CloudBuild`, a distributed build system for high degrees of build parallelism using VMs in the cloud, for use by our team. The authoring language for build was based on plain vanilla .NET code, declaring object graphs and build dependencies. Reactive Extensions (Rx) was used for the eventing glue between many components of the build system. Today, the use of cloud-powered distributed builds is standard practice, but it was a giant leap forward at the time.
- `Reactive Extensions (Rx)`, a cross-platform technology to support "LINQ to Events" using the `IObservable<T>` and `IObserver<T>` interfaces describing an improved version of the *observer pattern*. This became the foundation of Reaqtor and we'll describe it in much more detail below. While the initial version only shipped for .NET (as an MSI installer back in the pre-NuGet days), the team worked on ports to JavaScript (RxJS) and C++ (RxCpp).

Folks who were instrumental to the development of Reactive Extensions (Rx) include, in alphabetic order by last name:

- Brian Beckman
- Gavin Bierman (Microsoft Research)
- Bart De Smet
- Wes Dyer

- Aaron Lahman (RxCpp)
- Daan Leijen (Microsoft Research)
- Erik Meijer
- Matthew Podwysocki (RxJS)
- Jeffrey van Gogh
- Danny van Velzen

LINQ to Everything

Bart De Smet joined the Cloud Programmability Team in 2010 after spending a number of years in the .NET Framework team, working on Windows Presentation Foundation (WPF). Prior to joining Microsoft in 2007, Bart focused heavily on building LINQ providers for a variety of data sources (most notably SharePoint through CAML queries, and Active Directory through LDAP queries). Upon joining Erik Meijer's team in 2010, his first task was to focus on building a toolkit for "democratizing LINQ providers". Various existing LINQ providers were rewritten using a more discplined compiler back-end approach, introducing general-purpose optimizers and rewrites for .NET expression trees, proper AST models for target languages, and pluggable domain-specific optimizations to produce better code.

> One paper worth pointing out is Thirteen New Players in the Team: A Ferry-based LINQ to SQL Provider which shows vastly different approaches to build LINQ providers for relational databases compared to the initial LINQ to SQL offering. Our "LINQ to Everything" vision and execution was akin to this philosophy.

One of the goals of this effort was to sell LINQ as a solid foundation to build data access and querying technologies in the larger SQL Server organization. Various projects were spawn from this, including prototyping efforts to push LINQ expressions deeper into the SQL Server product, and using LINQ to become the language underneath SQL Server Integration Services (SSIS), the Extract-Transform-Load (ETL) product that ships with SQL Server. These projects caught some interest from the technical leadership team as possible future directions for initial big data processing cloud offerings, inspired by DryadLINQ. We'll return to these efforts later.

Due to the increased focus on Reactive Extensions (Rx), the work on "LINQ to Everything" became a 20% time project with sporadic resurgences. One of the tangential results of this effort was the introduction of expression tree support in Rx, which will be discussed further on in this document.

> The "LINQ to Everything" effort led to the creation of a plethora of libraries and tools, which live on to this day in one of the essential assemblies, `Nuqleon.Linq.CompilerServices`, used by various Reaqtor components. Upon Bart's transition to the Online Services Division in 2013, these tools were also put to use in the creation of a distributed graph database, which will be

> mentioned later on.

Reactive Extensions (Rx)

Reactive Extensions (Rx) was conceived around 2009 in the Cloud Programmability Team to support the tier-splitting requirement of the "Volta" project, from the following observation. In order to support splitting of an application across the client/service boundary, asynchronous communication is required. When retargeting the client tier of the application to JavaScript, all communication between client and service has to take place asynchronously using AJAX. When the "Volta" project was started, languages such as C# and Visual Basic lacked `async` and `await` and even the fundamental abstraction of a future in the form of `Task<T>`. This required the rewrite of client-side code to continuation passing style (CPS). We had to support both single-value asychronous computation, as well as event handlers across cloud boundaries:

```
[RunOnClient]
class Dashboard : Form
{
    public void Form_Load(object sender, EventArgs e)
    {
        int res = server.Add(1, 2);
        lblSum.Text = res.ToString();

        server.StockTicks += tick =>
                 this.Invoke(() => lstTicks.Add(tick.ToString()));
    }
}

[RunOnServer]
class Service
{
    public int Add(int a, int b) => a + b;
    public event Action<StockTick> StockTicks;
}
```

If we were to implement this technology today, there wouldn't be any doubt to translate the Add call to an asynchronous one, returning a `Task<int>`. However, for the event handler we still wouldn't have a good language-integrated solution, because .NET events are merely metadata constructs rather than first-class objects. Unlike a method which can be converted to an object using a delegate type, an event does not have an object representation. This led to the introduction of a framework to act as the "glue" for cross-boundary asynchrony, regardless of whether it involves a single value or multiple values.

> An interesting tidbit is that one of Erik Meijer's former doctorate students at MSR - Daan Leijen - worked on a futures library for .NET, which got productized in the Parallel Extensions for .NET by Joe Duffy, Stephen Toub, et al. Today, these libraries are part of the .NET Framework and include types such as `Task<T>`. Reactive Extensions borrows the "Extensions" suffix from the Parallel Extensions project because they were created around the same time.

This glue ultimately became Rx, after a number of false starts with interfaces such as `IEvent<T>`. Finally, we arrived at the `IObservable<T>` and `IObserver<T>` abstractions which got added to the .NET Framework eventually:

```
namespace System
{
    public interface IObservable<out T>
    {
        IDisposable Subscribe(IObserver<T> observer);
    }

    public interface IObserver<in T>
    {
        void OnNext(T value);
        void OnError(Exception error);
        void OnCompleted();
    }
}
```

For a detailed explanation of these interfaces, see reactivex.io and the rich set of videos on Channel 9. In essence, an *observable* sequence represents a data source for events, while an *observer* represents the receiver of such events. The `Subscribe` method attaches an observer to an observable sequence in order for it to start receiving events. The `IDisposable` object returned by this method can be used to dispose the subscription when receiving events is no longer desired. This approach is much more compositional compared to adding and removing event handlers, because both observable sequences and their subscription handles can be composed using algebras on `IObservable<T>` and `IDisposable`, respectively.

The Rx library provides a set of extension methods for `IObservable<T>` that provide LINQ-style query operators. Besides standard query operators for filtering, projection, aggregation, etc. the set of Rx operators also includes time-based ones, for example to sample a sequence of events, to throttle the flow of events, to create time-based windows of events, etc. An example to compute one-hour average stock prices is shown below:

```
IObservable<decimal> GetAverageStockValuePerHour(IObservable<StockTick>
↪  stocks, string symbol)
```

```
{
    return from oneHour in (from stock in stocks
                            where stock.Symbol == symbol
                            select stock.Price)
                           .Window(TimeSpan.FromHours(1))
           from average in oneHour.Average()
           select average;
}
```

Formalization of the interfaces, their semantics, and the Rx algebra of query operators was driven by Erik Meijer in collaboration with Gavin Bierman at MSR who realized that the `IObservable<T>` and `IObserver<T>` interfaces are the mathematical dual of the `IEnumerable<T>` and `IEnumerator<T>` interfaces. The former family of interfaces deals with *pull-based* sequences (consumed using blocking `foreach` loops), while the latter family of interfaces deals with *push-based* sequences (consuming using asynchronous callbacks supplied by `Subscribe`).

Release timeline

Prior to shipping Rx as a general-purpose library, the team collaborated with the Windows Phone organization to include the technology in the Silverlight-based .NET Framework that ships on the phone. This positioned Rx as an ideal programming model to observe changes to sensor inputs such as GPS, accelerometer, etc. With the support from folks on the Windows Phone team, the initial version of Rx shipped in the `Microsoft.Phone.Reactive` namespace for Windows Phone 7. This initial drop of the technology was eventually obsoleted in favor of the full Rx framework. Subsequent efforts took place to integrate the essential `IObservable<T>` and `IObserver<T>` interfaces into the .NET Framework's Base Class Library for the .NET 4.0 release.

The first release of Rx was shipped in 2010, focusing on completeness of the operator set and rigorous correctness checking through the notion of virtual time scheduling. The core Rx team consisted of Erik Meijer, Wes Dyer, Jeffrey van Gogh, Danny van Velzen, and Bart De Smet. A second release of Rx .NET was shipped in 2012, mostly focusing on performance enhancements and supporting various modern flavors of the .NET Framework and WinRT. More information on the feature set of these releases can be found on the (now defunct) Rx team blog. For the duration of the 2.x releases, Bart De Smet led the development efforts. Rx.NET is made available as a set of NuGet packages.

Implementation efforts of Rx for other languages were bootstrapped by the original team, most notably in JavaScript (RxJS, by Jeffrey van Gogh and Matthew Podwysocki) and C++ (by Aaron Lahman). 3rd parties later implemented many other variants, including Rx for Python by Dag Brattli, RxJava by folks at Netflix, Rx for Ruby, etc. The full list of available implementations can be found at reactivex.io.

After a detour through the (now defunct) Microsoft OpenTech organization around 2012, the Rx .NET

technology transitioned to the .NET Foundation in 2016, where it's largely maintained by the OSS community nowadays. The community successfully shipped a 3.0 release supporting .NET Core and is now largely focused on ensuring the technology is well-integrated with the rest of the .NET ecosystem.

> Today, most of the innovation in the Rx technology space is happening in RxJS and RxJava. The former is quite popular due to the historical lack of standardization of event handling and asynchronous programming models in JavaScript, acting as a universal glue to unify and compose across these different programming models. RxJava has made some attempts at adding some foundational capabilities such as back pressure, although it's the author's belief that the design for these constructs is misguided. The use of `IAsyncEnumerable<T>` is a better choice for such constructs (see below).

Interactive Extensions (Ix)

Shortly after the initial release of Reactive Extensions (Rx) in 2010, upon the introduction of `Task<T>` in .NET, the team looked into the concept of *asynchronous, pull-based enumerable sequences*. This led to the introduction of the `IAsyncEnumerable<T>` interface in a technology known as Interactive Extensions (Ix):

```
// Original Ix interfaces for asynchronous streams (2011).

public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetEnumerator();
}

public interface IAsyncEnumerator<out T> : IDisposable
{
    T Current { get; }
    Task<bool> MoveNextAsync(CancellationToken token);
}
```

On top of this, Ix.NET provides LINQ query operators as extension methods on `IAsyncEnumerable<T>`. This construct is useful when dealing with asynchronous retrieval of events across networking boundaries as was embraced by the Entity Framework as a core dependency. More than 5 years later, after the introduction of `async` and `await` in C# 5.0, the language design teams started considering first-class language support through an `await foreach` construct, which ultimately shipped in C# 8.0 as part of the Asynchronous Streams work. This also includes a revisit of the interfaces using `IAsyncDisposable` and `ValueTask<bool>`.

```
// Asynchronous Streams in C# 8.0 (2019).
```

```
public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetEnumerator(CancellationToken token);
}

public interface IAsyncEnumerator<out T> : IAsyncDisposable
{
    T Current { get; }
    ValueTask<bool> MoveNextAsync();
}
```

> The `IAsyncEnumerable<T>` interface has been invented (or discovered?) independently by a few teams at Microsoft. The first was the Cloud Programmability Team where we shipped it as a NuGet package alongside Rx.NET. Later, the Service Fabric team discovered the need for asynchronous pull-based enumeration of sequences when building support for reliable collections. The construct ultimately landed in C# 8.0 several years later.

Expression tree support via `IQbservable<T>`

After finalizing the initial release of Rx.NET, we started work on expression tree support for `IObservable<T>`, thus the dual to `IQueryable<T>`. This enables the creation of query providers, similar to LINQ to SQL, that can analyze user intent at runtime, and optimize event processing expressions, translate these expressions to a target language, and/or submit queries to a remote service.

```
public interface IQbservable<out T> : IObservable<T>
{
    Type ElementType { get; }
    Expression Expression { get; }
    IQbservableProvider Provider { get; }
}

public interface IQbservableProvider
{
    IQbservable<T> CreateQuery<T>(Expression expression);
}
```

> This initial design of expression tree support for Rx omitted the ability to quote observers using expression trees. Only observable sequences had a quoted counterpart. While this is useful for connected client scenarios where an observable query expression is submitted to a service and

> connected to a local observer, it doesn't support remoting an entire event processing subscription expression (i.e. including an observer) to a service. When designing Reaqtor, this facility was added, thus allowing for offline clients that submit a standing event processing subscription into the Reaqtor service, where it's reliably maintained.

> The funky spelling of this interface with a Q instead of an O in `IQbservable<T>` is not a typo. It's a pun on homoiconicity of expressions and quotations, so we made the quoted type *look the same as* the non-quoted equivalent.

We shipped support for expression trees in Rx.NET 1.1 with a sample query provider implementation for LINQ to WMI Events, where queries get translated to WQL. Some other sample that was built over the years is LINQ to Twitter, but this functionality never got widespread adoption in part due to the lack of query langages for streaming event processing that could act as translation targets. In Reaqtor, we started using this functionality to ship expression trees between clients and services, without any transpilation from one language into another.

Collaboration with StreamInsight

While in the Cloud Programmability Team under the larger SQL Server organization, the team crossed paths with StreamInsight which is a complex event processing (CEP) technology that's bundled with the SQL Server product. Designed as a .NET library, is supports temporal processing of events, using an `IStreamable<T>` abstraction with a LINQ-style query language on top. One of the key differences to Rx is the presence of a notion of application time whereby each event has an associated application time. For more information about the temporal semantics, see Advancing Application Time. The underlying theoretical foundation of StreamInsight is known as CEDR.

Members of the Rx team worked together with the StreamInsight team in the early days to assist with the definition of the semantics. This led to the realization that the temporal semantics of both systems are quite different, with Rx treating application timestamps on events as optional (thus not making the query operator implementations dependent on them). Based on this observation, regular afternoon hacking sessions between some members on both teams took place to try to implement StreamInsight semantics using Rx primitives. This effort was successful by leveraging the higher-order nature of observable sequences in Rx, which can be used to model durations (e.g. for event lifetimes), and by taking advantage of the virtual time scheduling support in Rx, which was used to push ordering of events based on application time to the "edge" of the event processing graph.

After StreamInsight's initial few releases, the Rx and StreamInsight teams collaborated on integration efforts between the two technologies, which led to first-class support for `IObservable<T>` and `IQbservable<T>` in StreamInsight 2.1. The integration of Rx support led to better interoperability

and various event sources and sinks were using `IObservable<T>` and `IObserver<T>`. This work led to the insight that adapters on inputs and outputs could be used as a general means to shake off or tag on additional semantics (e.g. introduction or removal of application timestamps), which was later used in the design for the core Reaqtor engine.

The MSR team, led by Jonathan Goldstein that provided the CEDR algebra continued research into streaming event processing systems, which led to the creation of Trill in 2013. Also implemented as a library, Trill is used by various Microsoft products today, including Azure Stream Analytics. It's based on the same CEDR algebra and models temporal event processing using notions of application time. Trill was released as open source in 2018.

We'd also like to extend our gratitude to the following people who provided valuable input and insights during our collaboration and in the years to follow:

- Badrish Chandramouli
- Georgi Chkodrov
- Jonathan Goldstein
- Torsten Grabs
- Colin Meek
- Rafael Fernandez Moctezuma
- Beysim Sezgin
- Tiho Tarnavski

Some contributors to StreamInsight that crossed paths with Rx and/or Reaqtor include Georgi Chkodrov (Tx (LINQ to Logs and Traces)) and Tiho Tarnavski (Reaqtor engine and C++ implementation on client devices).

Cloud Programmability Team, take two

Around 2012, after having successfully shipped Reactive Extensions (Rx) technologies for .NET, JavaScript, and C++, we refreshed the Cloud Programmability Team and start to focus on a next wave of incubation projects. Some of the original team members spun off various incubation efforts in product teams and landed in Xbox, Chakra, DevOps, etc.

The refreshed team refocused to drive integration of Rx into more technologies within Microsoft, to explore new cloud programming models such as actors, and to investigate the underpinnings of NoSQL stores.

Open Management Instrumentation (OMI)

As part of the goals to spread adoption of the Rx technology across the company, various partnerships were established. Some of these partnerships are listed below:

- Office Labs, where Rx was used for UI programming.
- SQL Server Reporting Services, where Rx was used in the Silverlight client to assist with UI programming.
- System Center, where Rx is used to process system monitoring events to trigger various management actions.
- Windows Telemetry, where Rx is used to process telemetry signals.

Given the successful partnerships with various Windows teams, subsequent efforts were made to introduce Rx in various manageability and automation technologies. Out of this came a prototype for "LINQ to PowerShell" where the PowerShell pipeline was exposed as an observable sequence and remoting of queries against outputs of such pipelines can be remoted into PowerShell using expression tree transpilation. Further integration with PowerShell included bridges to the eventing and remoting systems in PowerShell. This caught attention from Jeffrey Snover for the work on Open Management Infrastructure (OMI) where a reactive component would be useful. A collaboration was set up with the Windows Server organization to integrate Rx for C++ into this stack over the course of several months.

LINQ to Logs and Traces (Tx)

Another effort was Georgi Chkodrov's work on LINQ to Logs and Traces as a successor to TraceInsight, a technology he built on top of StreamInsight for temporal processing of traces. The lightweight nature of Rx (compared to StreamInsight) and the simpler programming model for temporal event processing attracted Georgi to the team in order to build a log and trace processing framework on top of Rx.

LINQ to Logs and Traces (Tx) has various input adapters to consumes logs and traces from a variety of sources, including Event Tracing for Windows (ETW), Windows Event Logs (.evtx files), Windows Performance Counters, IIS W3C Logs, SQL Server XEvents, and many more. These adapters support receiving events in real time and/or replaying events from persisted logs, exposing them as `IObservable<T>` sequences. Tx queries are expressed using the familiar query operators in Rx. All of the observable input sequences to a Tx query are merged onto a common timeline by using the timestamps on the received events to perform a historical merge sort. Replay of historical data from persisted logs is driven by Rx's support for virtual time scheduling.

Because the same reactive query expression can be used against real-time events and the replay of historical events from persisted trace or log files, Tx enables a great system and service monitoring development lifecycle. First, one uses historical traces to prototype reactive queries that can predict

service issues from the various system events received. Effectively, a candidate reactive query is built to attempt detecting issues from the events that occur before such an issue appears. Next, this candidate query can be asserted against various occurrences of issues to assert its predictive power or to refine it. Finally, this query is deployed - without requiring any changes to its logic - to the online service where it operates on real-time events.

LINQ to Logs and Traces (Tx) was eventually shipped to GitHub where it's actively maintained. Today, Tx is used in various places across the company for system and service monitoring at data center scale.

Oslo and M

At the Professional Developers Conference (PDC) in 2008, Microsoft introduced a modeling framework code-named "Oslo", including a modeling language called "M" and a Domain Specific Language (DSL) definition language called "MGrammar". Remnants of this technology can be found on the interwebs, for example on Channel 9. Oslo introduced capabilities to model data used and exposed by services (such as Active Directory) and mapping on underlying execution platforms (such as SQL), using DSLs and the M modeling language starring a structural type system.

> Reaqtor's type system for entities - used to represent events - was indirectly inspired by the type system in M. This type system, sometimes referred to as the "entity data model", was originally used in a distributed graph database effort (see later), and ultimately ended up being the data model used by Reaqtor (see `Nuqleon.DataModel` assemblies). Today, structural typing has been popularized by type systems layered on top of dynamic programming languages such as JavaScript, most notably TypeScript.

Upon the demise of the technology around 2012, the Oslo team transitioned to the service and tools business where projects were started to define the future of data processing, considering upcoming trends around NoSQL and Big Data, such as MongoDB and Hadoop. The goal of the new organization was to sketch out the future of data at Microsoft, in the midst of the transition from on-prem to cloud-based data solutions (e.g. SQL Azure). SQL Server already had a history of adapting to a variety of other data and programming models starting in the SQL Server 2005 "Yukon" release (cf. XML support, FILESTREAM data types for blobs, ServiceBroker queue management, SQLCLR for running custom code in the database, spatial data types, Notification Services for real-time notification, etc.). In parallel, efforts on DryadLINQ provided a possible direction for the Big Data strategy at Microsoft. Some of this work ultimately led to the creation of predecessors to HDInsight, and much later Cosmos DB.

The Cloud Programmability Team chimed in to these conversations, trying to push its LINQ to Everything agenda. One working group was nick-named the "data refinery" group, where we explored new programming models to build large-scale data transformation and analysis services using formal languages based on LINQ, leveraging the semantics of various well-understood operators to build

distributed query execution plans that can glue together data stores with different "Volume, Velocity, Variety" characteristics. To provide a connection to existing assets in the SQL Server organization, our working group prototyped a query builder on top of the SQL Server Integration Services (SSIS) designer, emitting LINQ expression trees on the back-end. Imagine a split-view editor with a data flow designer on the top, a toolbox with data sources, sinks and operators on the left, and a (C#-based) code editor at the bottom, representing the logic drawn in the designer.

Nodes in the designer represented either data sources (exposed as either `IQueryable<T>` or `IQbservable<T>` collection types, for example SQL Server tables, Twitter streams, etc.), data sinks (using a first variant of `IQbserver<T>`, for example SQL Server tables, queues, etc.), general-purpose query operators (such as filtering, projections, etc.), or specialized query operators (e.g. machine learning modules). The execution planner evaluated which data sources support execution of query operators, considering the volume and velocity metrics of such transformations, in order to weigh various execution plans. A good example of this strategy is predicate pushdown, but our approach was generalized to any query operator. The resulting plan effectively consists of sending (sub)expression trees to the various data sources, leaving the remainder computation in a data processing node.

This was very different from SSIS in many ways. First, it supported pull-based (iterative) and push-based (streaming) data acquisition, dealing with the velocity mismatch of various data sources. Second, it supported mapping of data models to support the variety mismatch of various data sources (not everything is relational, e.g. document databases). Third, it supported truly distributed execution unlike SSIS where everything ran on a single compute node, thus shipping data to compute rather than the opposite in support of the volume axis.

> The concept of *delegation* in Reaqtor, whereby parts of reactive query expressions can be pushed down or delegated to other reactive services, is a child of this technology. We'll discuss this concept in more detail when talking about IReactiveProcessing (IRP) further on.

One last thing worth mentioning in this context is a first demonstration of distributed reactive programming using the `IQbservable<T>` support in Rx during an all-hands meeting with the extended SQL Server organization after the absorption of the Oslo/M teams in 2012. To illustrate our "LINQ to Everything" ambitions to ship code to data, whether it's a traditional store or an event stream producer, we built a group chat server with real-time language translation for users. The client code looked roughly as follows:

```
IQubject<Message> ladyGagaChannel;
IQubscription subscription;

void Connect(string language)
{
    ladyGagaChannel = chatServer.GetSubject<Message>("#ladygaga");
```

```
    ladyGagaChannel.Select(m => new Message { Sender = m.Sender, Text =
↪   Translate(m.Text, "EN-US", language) }).Subscribe(m =>
↪   txtChat.AppendLine(m.ToString()));
}

void Disconnect()
{
    subscription.Dispose();
}

void SendMessage(string message, string language)
{
    ladyGagaChannel.OnNext(new Message { Sender = CurrentUser, Text =
↪   Translate(message, language, "EN-US") });
}
```

This was the first internal demo of shipping LINQ query expression trees to a service for remote execution as a standing query. The system lacked various concepts that are present in today's IRP programming model used by Reaqtor, such as identifiers for remote artifacts, asynchronous programming patterns, state checkpointing to support failover of the service, reliable messaging, splitting of queries across different compute nodes, metadata querying, etc. However, the essence of Reaqtor is identical to this proof of concept which was built by Bart De Smet and Wes Dyer over a few nights for an all-hands meeting.

> Work to enable this proof of concept included building an expression tree serializer. Prior work on LINQ to Everything had focused on a general-purpose framework to support transpilation of expression trees to domain specific query languages. In this case, we wanted to send the expression tree as-is between a client and a service which were both running .NET. The initial expression tree serializer implementation worked well for purposes of the POC. Later, during the early days of Reaqtor, Bart made a more powerful expression tree serializer called "Bonsai", which we'll discuss later.

NoSQL

In 2012, Savas Parastatidis joined the team to work on a new incubation project to explore NoSQL stores. Erik Meijer had been working on a theoretical foundation for NoSQL stores modeled as a mathematical dual to traditional relational databases, with regards to the (de)normalization differences to data storage in both types of stores, thus tackling the variety axis of the 3 Vs of Big Data. These ideas were run by the remaining Rx veterans on the team who've become deeply familiar with the dual properties

in the velocity domain, leading to pull-based and push-based query processing capabilities. A remnant of this thinking dating back to 2011 can be found in Bart's ECOOP 2011 presentation on the subject. This work was accelerated by the M/Oslo merger where the larger organization was seeking for a good way to embrace the upcoming trend of document databases with MongoDB as the most relevant example at the time.

With this new project, we wanted to build a distributed document database centered around the notion of running computation (such as querying) close to the data. One way to think of this is to represent unstructured data (i.e. "documents") as a graph of nodes and edges in a graph database, enabling computations to be sent to these nodes. This led to the realization that this programming model is very similar to the actor model so collaboration was established with Phil Bernstein's team at MSR that was building "Orleans". One tangent to the thinking here was to incorporate reactive programming support to enable observation of changes to the nodes and edges in the graph.

> The ideas of this project were carried forward to a distributed graph database effort, which is discussed later in this document. Foundational ideas such as shipping code to data as well as the data model used to represent entities in such a store ultimately carried forward to the Reaqtor work. The initial north star vision for a distributed graph database with reactive capabilities was also informed by the desire to build a system that supports spatial (graph of nodes and edges) and temporal (time axis of changes to nodes and edges) traversal of data through a unified programming model.

Thinking in this space was ultimately spun off in the distributed object graph work (see further on), built on top of proven distributed systems technology in the Bing organization operating at cloud scale. Out of this came Erik's desire to research the fundamentals of actor-based programming by building an actor framework called "ActorFx" on top of the newly released Windows Fabric, which was the predecessor to Service Fabric, thus eliminating the low-level complexities of building self-healing distributed systems.

ActorFx

The last major project of the Cloud Programmability Team started around 2011-2012 and was ActorFx with the goal of researching the implementation of the essence of actor based programming in the cloud as one of the new upcoming cloud programmability models. Work on this project was led by Brian Grunkemeyer (who previously worked on .NET Framework and CLR) and Joe Hoag (who previously worked on the .NET Framework's threading and `Task<T>` APIs).

Implementation of ActorFx was on top of Service Fabric (back then called Windows Fabric) and used CLR and .NET functionality available to serialize delegates across machine boundaries in order to effectly remote execution of code. Some of these approaches were proven to be contentious with alternative

approaches using expression trees arising. Various sample applications for Facebook-like applications (with actors representing people on a social network) were built on top of this technology.

Today, actors on Service Fabric are available out of the box. For more information, see Service Fabric Reliable Actors. Orleans also pioneered innovations in this space with virtual actors which have been used to power Halo at massive scale.

Distributed Graph Databases

In late 2012, Bart joined Savas in the Online Services Division to join forces in building a new graph database with reactive capabilities. The team's charter was to bridge the gap between the world's data (e.g. entities in Bing such as artists, updates to the world's information such as weather, etc.) and various consumers of this data in different groups across the company (e.g. apps in Windows 8, Office scenarios, etc.). The goal of the graph database effort was to store and query real-world entities and relationships amongst them. For example, Xbox Music used the store to query information about artists, their songs, ratings, etc.

This new graph database was built on top of an existing distributed key/value store that was used within the Bing organization, and was implemented as a set of so-called coprocs that run colocated to the data. An HTTP-accessible front-end supported various operations, including inserting nodes and edges into the graph, and performing queries over the data in the graph. To build the latter, we decided to build a LINQ-based query language akin to LINQ to XML to support traversal of collections of nodes and collections of incoming and outgoing edges for each node. Data associated with nodes and edges was represented using entities based on the JSON type system. The execution of these LINQ queries used expression tree shipping to coprocs colocated with the collections of nodes being queried for efficient local execution by accessing the local ObjectStore database.

Bart started the effort on integrating the LINQ to Everything technology to build a LINQ front-end and query processor back-end, and to resurrect the idea of a reactive graph by integrating the work on `IQbservable<T>` and expression tree shipping into the technology. Given that these ideas provided the grass roots for the Reaqtor vision, we'll get into some more detail on these.

LINQ query language design

The design of the LINQ query language for the graph was largely based on the existing `IQueryable<T>` support in the .NET Framework. In order to model nodes and edges, data types were introduced that support the navigation from an edge to its upstream and downstream node, and from a node to its collections of incoming and outgoing edges. By modeling these as collections, enumerable style querying became readily available. Navigation across nodes and edges

then boiled down to a `SelectMany` operator application, which is akin to a cross-apply operation. Given that the `SelectMany` operator is a very fundamental operator (equivalent to `bind` in monadic functional programming), it has various nice algebraic properties that enable execution planning and optimization. For example, filters commute well with this operator, thus allowing for predicate pushdown which results in upstream filtering and less data transfer between query processors. For example:

```
from actor in graph.Nodes<Actor>()
from actedIn in actor.Edges<ActedIn>()
let actor = actedIn.Actor
let movie = actedIn.Movie
where actor.Name == "Nicole Kidman" && movie.Year < 2010
select movie
```

can be optimized into:

```
from actor in graph.Nodes<Actor>()
where actor.Name == "Nicole Kidman"
from actedIn in actor.Edges<ActedIn>()
let movie = actedIn.Movie
where movie.Year < 2010
select movie
```

Alternative graph query languages such a Neo4j's Cypher and SPARQL can be layered on top of the LINQ-based query operator algebra, recognizing a graph can be modeled as a collection of nodes and edges with traversal patterns in between. Such graph query languages merely focus on conciseness in syntax by introducing syntactic sugar for node-to-node traversal via edges:

```
MATCH (nicole:Actor {name: 'Nicole Kidman'})-[:ACTED_IN]->(movie:Movie)
WHERE movie.year < 2010
RETURN movie
```

Modeling a graph as queryable collections is very similar to LINQ to XML's approach of modeling XML as collections of elements, attributes, etc. with rich traversal patterns such as `Children`, `Descendants`, etc. In a way, this approach provides a very traversal-oriented way to accessing data in a graph database, in some way similar to using cursors to navigate a table-oriented database system. By recognizing such ideomatic traversal patterns and translating them to efficient execution plans (see below), a decent query optimizer can be built that can also benefit from very fundamental algebraic properties of LINQ operators, while also enabling expressing many more domain-specific graph query languages on top of it. This is merely a traditional compiler and execution planning pipeline with support for many front-end languages layered on top of a common intermediate representation with an optimized execution back-end underneath.

Expression shipping

In order to support rich querying across the client-service boundary, we steered away from a design where an OData endpoint is the end-all be-all querying API surface. While REST-style traversal of entities through path expressions is a valid API surface to offer, offering rich query capabilities involving complex joins, expressing non-trivial predicates and projections, etc. is not the strength of such protocols. Instead, we rely heavily on the ability to ship expression trees from client to service. Complementary API surfaces, such as REST and/or OData support, were layered on top of the underlying query language and query processor implementation.

The implementation of expression tree shipping was initially based on the LINQ to Everything work (cf. the "reactive chat service" demo and `IQbservable<T>` work) which included XML-based and JSON-based serialization functionality for the `System.Linq.Expressions` object model. While this approach worked well, it suffered from somewhat tight coupling between clients and services due to the verbatim representation of the reflection object model (i.e. types, members, and assemblies). This meant that clients and services had to share some assemblies.

> To break this coupling, Reaqtor introduced additional expression tree normalization steps, and a serialization format called "Bonsai" which abstracts away the CLR type system. We'll discuss Bonsai in more detail later, but suffice it to say that the work on expression shipping in the LINQ to Everything days provided the foundation for this work. In fact, a first step towards a normalized representation was taken by the graph database effort, namely to move to a standardized data model which is discussed further on.

Besides designing for eventual support for many graph-specific external Domain Specific Languages (DSLs) such as Cypher and SPARQL, the use of expression tree shipping facilities killed two birds with one stone. First, it enabled the use of general-purpose expression tree representations (which can be reduced to typed lambda calculus with some specialized nodes for well-known constructs such as arithmetic operators), thus avoiding the introduction of yet another intermediate language. Second, it also resulted in a fairly trivial implementation for internal DSL support using LINQ syntax in higher-level languages such as C#, Visual Basic, and F#. See Martin Fowler for a definition of internal and external DSLs.

While traditional LINQ providers have always dealt with translation between an existing query language external DSL (such as SQL) and the LINQ-based query expression internal DSL in C#, Visual Basic, and F#, the graph database's use of LINQ was fundamentally different. Rather than inventing a new language to translate LINQ expression trees into, we put this representation on the wire, after applying some normalization steps to shake off quirks that leaked into the representation from any of the host languages. This immediately side-stepped many of the complexities involved in inventing new languages, including syntactic choices (rather than a tree-based representation), type systems and

type checkers, etc. (not to mention any IDE experience to edit code with features such as IntelliSense and whatnot).

Data model type system

Nodes in the graph database were modeled using JSON data types. To support a rich querying experience from .NET languages, we built support to project these types onto "plain old CLR types", akin to what object/relational mapping technologies do as well. This led to the introduction of the so-called "Bing data model", which enables annotating .NET types with attributed to make them usable - through LINQ support - for reading and storing entities in the graph.

```
class Person
{
    [Mapping("bing://entities/person/name")]
    public string Name { get; set; }
}
```

The string passed to the `Mapping` attribute is treated as an opaque identifier in the stack, but the recommendation was to use well-known URIs from ontologies such as schema.org. Given that these types are referenced within expression trees representing query expressions that are transmitted from the client to the service, we had to come up with a strategy to break binary dependencies on user types. On top of this, we also wanted to ensure that these schemas are open-ended, i.e. additional properties can be added at any time. This led to the adoption of a structural type system underneath the .NET projection.

> The use of a structural type system projected into "plain old" (nominally typed) .NET types is in a way inspired by the work on Oslo/M. During the sunset of the Cloud Programmability Team with the "data refinery" project, a similar projection was built for the distributed LINQ to Everything demo running on top of the original SSIS designer.

The mechanism underneath the .NET type projection onto structural entities was based on a process called "anonymization" of types. The key realization here was that LINQ query expressions already needed somewhat structural type notions in order to support features such as the use of anonymous types for grouping keys or projections, or transparent identifiers used during the "desugaring" phase of translating LINQ query expressions to more primitive operations. As such, expression tree serialization already supported C# 3.0 and Visual Basic 9 anonymous types, by capturing the shape of the type (structural) rather than the name of the type (nominal). Support for data model types in query expressions was lit up by introducing an expression rewrite phase prior to serialization, causing nominal data model types to be nominally erased in favor of anonymous types retaining their structure. For the example show above, the corresponding type would look a bit like this:

```
[CompilerGenerated]
class <>__AnonymousType1
{
    public string bing://entities/person/name { get; set; }
}
```

Note that the names of types (e.g. `Person`) and members (e.g. `Name`) got erased. Type names completely vanish and get dropped in the serialization format (i.e. `<>__AnonymousType1` doesn't occur in the payload), while property names get substituted for the identifier applied to them using the `Mapping` attribute. This effectively led to zero nominal entity type sharing between client and service, which was a prerequisite to make the graph database work without requiring sharing of assemblies, versioning woes, etc.

> The data model described here is still in use by Reaqtor, albeit using a different implementation strategy. Assemblies starting with `Nuqleon.DataModel` reveal this inheritance of assets. Use of the data model in Reaqtor was motivated by the "reactive graph" vision where spatial/temporal traversal of entities was the ultimate goal. By using the same data model for persisted entities and streaming entities, unification of the object model used across these programming models was made possible. The ultimate goal of this was to support an ontology of the world's data, which was a key ingredient to the team's vision.

Note that this strategy also removes the notion of and need for a separate schema repository. By using metadata APIs to query data processing systems (such as graphs, document databases, or stream processing platforms) for available data collections (e.g. nodes, edges, streams, etc.), structural type information about the entities within these collections can be obtained. In effect, the collection and presence of entities within it acts as a witness of structural types. These metadata APIs can be used to discover the structure of data, which can be joined together to a loosely coupled ontology provider such as schema.org in order to obtain more information. A traditional developer experience that can be layered on top of this is code generation for client libraries with strong typing (i.e. similar to O/R mapping technologies).

Finally, note that the use of type projections for use in query expressions in a statically typed language is completely optional. Such types merely provide an enhanced editing experience with rich type checking and compile-time validation. However, in addition to supporting projected entity types, one can also use static types that represent a dynamic type, such as `DynamicObject` or `JObject`.

```
from person in graph.Nodes<Person>()
where person.Name == "Bill Gates"
...
```

versus

```
from person in graph.Nodes<JObject>()
where person["bing://entities/person/name"] == "Bill Gates"
...
```

In other words, this approach provides optional static typing on top of a structural and dynamic type system underneath. By erasing static nominal types in favor of a structural type representation, this also enabled eventual implementations of client libraries for these data processing services in other languages such as JavaScript or Python.

Execution planning

The initial implementation of the query planner and processor was merely an expression tree binder and compiler for the expression trees submitted by clients. This code ran in the context of an key/value store coproc, collocated with the data being queried. Very little logic went into routing the query to the appropriate location or optimizing the communication across nodes while the query was being executed. Rather, the nodes and edges touched by the query were being queried using in-memory code execution on the key/value pairs (and collections thereof) retrieved from the underlying store, through an intermediate object model.

In order to get closer to a proper implementation, steps were taken to introduce an execution planning phase, which would ultimately result in sending primitive graph query operations across different nodes to get better data locality. In addition, this would also enable optimizations that go beyond static analysis of the query expressions followed by application of algebraic rewrites, by incorporating various metrics to rank possible execution plans (cf. traditional histograms in RDBMS systems).

Translation between the incoming query expression and the execution plan was implemented using a LINQ to Everything utility called “BURS” which stands for Bottom Up Rewrite System. It’s an implementation of a variation of Todd Proebsting’s BURS for .NET expression trees and other ASTs modeled in object-oriented languages. Proebstin’s original BURS was meant to be an optimizing table-driven code generator converting an AST into a series of instructions, with every rule specifying a cost. The BURS implementation written as part of the LINQ to Everything toolkit further abstracts this by enabling table-driven cost optimizing translation across any pairs of languages modeled using generalized notions of trees, nodes, and tags. For example, a LINQ to SQL implementation starts off by converting .NET expression trees to an alternative tree representation that enables bottom-up tiling, followed by modeling the target SQL language using a similar tree object model. Finally, a set of cost-annotated rules is declared to translate between source and destination language (e.g. a `MethodCallExpression` to `Queryable.Select` can translate to a `Project` node in SQL, which is later turned into `SELECT` clause syntax). Use of BURS enabled the team to build a query processor relatively quickly.

One more stumbling block encountered while building the query processor was the impedance mis-

match between the synchronous LINQ query operators based on `IEnumerable<T>` and the asynchronous APIs provided by the key/value store to access data. These asynchronous APIs were rather cumbersome and didn't align with any of the standard .NET patterns for asynchronous programming at the time, i.e. the APM, EAP, or `Task<T>`. Instead, it used a pretty basic callback mechanism which was hard to translate to. In order to solve this mismatch, Bart extended the LINQ to Everything toolkit by adding continuation-passing style (CPS) rewriters, allowing for a straightforward mapping of synchronous language constructs to asynchronous execution plans by composing BURS with a CPS rewriting phase.

> Reaqtor doesn't use BURS today because it doesn't require a translation between a front-end query language (LINQ using Rx operators) and a back-end query language used for execution. In fact, the target language is isomorphic to the source language, given that Reaqtor is really "Rx as a service". However, BURS can still prove useful in the future to perform query optimization. By implementing BURS as a table-driven tree transformation between a source language expressed as `ITree<TFirst>` and a target language expressed as `ITree<TSecond>`, it is also possible to perform transformations between trees describing the same language. At this point, BURS becomes an optimizer rather than a translator. Ideas of BURS were later incorporated in `Nuqleon.Linq.CompilerServices.Optimizers` which was built as part of Reaqtor. This library provides a general-purpose framework to build optimizers for LINQ-style query languages based on tree transformations using declarative specification of algebraic identities. The main difference compared to BURS is the lack of a cost-driven ranking of optimization rules to apply.

IReactiveProgramming (IRP)

While the distributed graph database effort was ongoing, Bart and Savas started to entertain the idea to add a reactive component to the graph. Inspired by the "chat server" demo in the Cloud Programmability Teams days and the foundational ideas of project Detroit to a lesser extent, the initial design efforts on building a service layer for Rx were kicked off. This led to the creation of a programming model coined "IReactiveProgramming", commonly abbreviated as IRP. The goal of IRP is to describe reactive programming systems in a formal manner, enabling "Reactive as a Service" (RaaS).

> Even though "IReactiveProgramming" starts with an `I` to designate it as an interface (conform .NET naming guidelines), no such interface really exists today. Very early designs and implementations of IRP described the whole system with a single interface, from which the project name was derived. The name stuck around ever since, with some referring to it as "IReactiveProcessing"

> instead. Traces of this name can be found when spelunking through the Reaqtor codebase today.

The design of IRP focused on a number of aspects:

- Extending on the expression tree support in Rx with `IQbservable<T>`. In particular, we also needed expression representations of observers and subjects, at the very least. The "chat server" demo made this omission clear.
- Support for asynchronous loosely connected clients, resulting in the need for a `Task<T>`-based API and the ability to identify resources such as subscriptions, subjects (streams), etc.
- Decoupling of the data model used to represent entities and the design of IRP. For example, it'd be undesirable to couple IRP to either the "Bing data model", the CLR type system, or Entity Framework, to name a few.
- Further reducing the coupling between clients and services by focusing on better expression tree normalization and serialization capabilities. Work on the graph database helped to reveal shortcomings to existing art in this space.
- Reusability of the core IRP implementation across tiny devices and massive cloud services by building IRP as a set of small libraries with minimal external and internal dependencies.

Initial implementation of IRP was largely done by Bart De Smet, with Eric Rozell joining the team later.

The three dimensions of IRP

IRP generalizes Rx over three orthogonal dimensions:

- Intrinsic (Rx) versus extrinsic (IRP) identifiers, i.e. reactive entities such as observables, observers, subscriptions, and subjects are no longer described using object addresses but with explicit identifiers.
- Code (Rx) versus code-as-data (IRP) representations of reactive computations, i.e. using quotations to capture intent using expression trees, enabling shipping of code as data (cf. `IQbservable<T>`).
- Synchronous (Rx) versus asynchronous (IRP) APIs to create or delete reactive entities such as subscriptions (cf. `Subscribe` and `Dispose` methods which are synchronous in Rx).

An example of a typical type in the IRP APIs is `IAsyncReactiveQubscription`, which is a quoted representation (cf. the Q in the name) of a subscription, supporting asynchronous (cf. the `Async` prefix) operations such as `DisposeAsync`. An instance of this interface in the client API of an IRP compliant system is a local object-oriented proxy type to a standing reactive event processing computation (a "subscription") in some external IRP compliant service. In order to obtain this proxy, an identifier is specified. The source of these proxies is similar to the `DataContext` in LINQ to SQL where one can

obtain proxies to tables using `GetTable<T>` calls. In the case of IRP, an `IReactiveProxy` interface exposes various `Get` methods:

```
public interface IReactiveProxy
{
    IAsyncReactiveQbservable<T> GetObservable<T>(Uri observableId);
    IAsyncReactiveQbserver<T> GetObserver<T>(Uri observerId);
    IAsyncReactiveQubject<T> GetStream<T>(Uri streamId);
    IAsyncReactiveQubscription GetSubscription(Uri subscriptionId);
}
```

Some more advanced constructs including stream and subscription factories have been omitted in the code fragment above, but the idea of returning proxy objects to refer to a well-known servie side "reactive entity" is the same.

> The choice of `Uri` types to represent identifiers is arguably an influence of the graph database effort where we were standardizing on the use of URIs to represent properties on entities according to ontologies such as schema.org. In retrospect we could have used simple strings here instead. Actually, native implementations of IRP on some types of devices have used GUIDs to refer to entities, allowing them to be implemented as COM objects. Communication between services and client device shadows would use URIs which get translated to GUIDs. It was briefly considered to parameterize interfaces on the identifier type used (with a `TIdentifier`) but this leads to very unweildy APIs with generic parameter bloat all around.

The proxy interface with its `Get` methods looks familiar to the LINQ to SQL APIs. This is not an accident; it's a very deliberate choice to provide a similar experience to author streaming queries rather than queries over persisted data stores. As part of this analogy, IRP also introduces a strict physical/logical layering. In the logical space, the `IReactiveProxy` (and related interfaces, see later) form the base of the implementation hierarchy, which ultimately leads up to a "context" base class akin to `DataContext` in LINQ to SQL. Derived classes can further add specialization, for example by exposing properties that provide immediate access to reactive artifacts:

```
public class MyContext : ClientContext
{
    [KnownResource("reactor://platform.bing.com/weather")]
    public IAsyncReactiveQbservable<WeatherInfo> Weather { get; } =
    ↪ base.GetObservable<WeatherInfo>(new
    ↪ Uri("reactor://platform.bing.com/weather"));
}
```

Such code would typically be generated by tools from metadata that's retrieved from an IRP-compliant service at development time, similar to the LINQ to SQL designer in Visual Studio. Note the use of `KnownResource` which we'll get back to when talking about the normalization of expression trees.

The three dimensions of IRP form a cube with eight possible combinations. Classic Rx has two, more or less. The first is `IObservable<T>` which has intrinisic identifiers, a synchronous programming model, and a code-based implementation. The second is `IQbservable<T>` which provides a partial expression-based implementation (only for observables). One can envision asynchronous variants of the Rx APIs as well, which have been prototyped a number of times. This makes the use of extrinsic identifiers for reactive entities the main differentiator of IRP.

> Given that Rx comes from the pull-to-push duality of `IEnumerable<T>` to `IObservable<T>`, it makes total sense to discover the same fundamental duality for `IAsyncEnumerable<T>` resulting in an `IAsyncObservable<T>` interface. These interfaces exist in IRP and are used during various expression tree transformations, but no asynchronous implementation of query operators exists as part of IRP. Various prototypes of "async Rx" do have such implementations though. In the case of IRP, we never had a necessity for such operator implementations because we translate user intent to synchronously executing query operators that run within a query evaluation engine, which we'll describe in more detail later. This is an area of potential future innovation.

The three aspects of IRP

In addition to having the three dimensions discussed above, IRP also prescribes three aspects for reactive processing systems.

- Proxies to reactive artifacts (identified using URIs) can be obtained in order to perform composition, e.g. applying query operators to observable sources, and connecting the result to an observer sink in order to compose a subscription. This is similar to e.g. `DataContext` in LINQ to SQL.
- Definitions to create (parameterized) macros over existing reactive artifacts. This is similar to databases providing views, stored procedures, and user-defined functions.
- Metadata describing the reactive artifacts, enabling IRP-compliant systems to discover each other's reactive artifacts (to facilitate distribution of computation), and to enable tooling used to create client APIs. This is similar to the catalog in a database system fueling tools such as O/R mappers.

The six reactive artifact types of IRP

IRP defines six reactive artifact types which are logical extensions and generalizations of the Rx concepts. A top-level division of artifact types is the distinction between hot and cold.

- Cold artifacts are merely definitions akin to macros in programming languages. They're passive, aren't "running" all the time, and can be bound/inlined in the context of other definitions or hot artifacts.
- Hot artifacts are running and can be compared to instances of objects in a programming language. They're active, have a runtime lifecycle, and can be stateful and have to be maintained by a hosting infrastructure.

The six artifact types form a "Standard Model" for reactive processing with a symmetry between cold and hot artifact types. The cold artifact types are often referred to as "definitions" or "factories" (for their corresponding hot sibling). Hot artifacts are sometimes referred to as "processes" to reenforce their always-running nature.

- Observables are cold definitions of a (result) type isomorphic to `IObservable<T>`. An example of a parameterized observable definition is a query operator, e.g. `Where` and `Select`. Another example is an event source, e.g. an "EventHub observable" parameterized on some connection string or a topic.
- Observers are cold definitions of a (result) type isomorphic to `IObserver<T>`. An example of a parameterized observer is a "file observer" parameterized by the path of the file to write to, or an "EventHub observer" to act as a receiver for events.
- Subscription factories are cold definitions of a (result) type isomorphic to `IDisposable` in Rx, or `ISubscription` in IRP. An example of a parameterized subscription factory is the `Subscribe` operation itself, which is parameterized on an observable and an observer. Other examples are composite subscriptions, "parameterized user-defined queries", etc.
- Subscriptions are hot artifacts that represent a running reactive computation which composes observables and observers to achieve event processing and data flow.
- Stream factories are cold definitions of a (result) type isomorphic to `ISubject<T>` in Rx. An example of a parameterized stream factory is a "replay stream" with a parameter indicating the retention duration for events (similar to `ReplaySubject<T>` in Rx).
- Streams are hot artifacts that represent an active stream. Streams are compatible with both the observable and observer type, so they can act as sources and sinks.

The types of these artifacts are shown below, in the quoted asynchronous space with extrinsic identifiers (see above):

Artifact	Rx type	IRP type
Observable	`IObservable<T>`	`IAsyncReactiveQbservable<T>`
Observer	`IObserver<T>`	`IAsyncReactiveQbserver<T>`

Artifact	Rx type	IRP type
Subscription factory	`Func<..., IDisposable>`	`Func<..., IAsyncReactiveQubscription>`
Subscription	`IDisposable`	`IAsyncReactiveQubscription`
Stream factory	`Func<..., ISubject<T>>`	`Func<..., IAsyncReactiveQubject<T>>`
Stream	`ISubject<T>`	`IAsyncReactiveQubject<T>`

Note that the typing relationship of subjects inheriting from both observables and observers is retained between Rx and IRP:

```
interface ISubject<out T, in U> : IObservable<T>, IObserver<U> {}
interface IAsyncReactiveQubject<out T, in U> : IAsyncReactiveQbservable<T>,
↪  IAsyncReactiveQbserver<U> {}
```

Observables

Observables act as sources of events, thus enabling the creation of a subscription. The relationship between Rx and IRP is as follows:

```
// Rx
interface IObservable<out T>
{
    IDisposable Subscribe(IObserver<T> observer);
}

// IRP
interface IAsyncReactiveQbservable<out T> : IExpressible
{
    Task<IAsyncReactiveQubscription> SubscribeAsync(Uri subscriptionId,
↪  IAsyncReactiveQbserver<T> observer);
}
```

Besides the addition of an `Async` suffix, the use of `Task<T>` for the return type, and the introduction of an extrinsic identifier `subscriptionId`, the interfaces are isomorphic.

What's commonly referred to as "query operators" is merely a collection of functions that given zero or more parameters return an observable sequence. Examples include:

```
// Produces an empty event sequence.
IObservable<T> Empty<T>();

// Produces an event sequence that emits an event at the specified due
↪  time.
IObservable<long> Timer(DateTimeOffset dueTime);

// Produces an event sequence that will emit events from the source
↪  sequence that pass the predicate.
IObservable<T> Where<T>(IObservable<T> source, Func<T, bool> predicate);

// Produces an event sequence that emits transformed events from the source
↪  sequence.
IObservable<R> Select<T, R>(IObservable<T> source, Func<T, R> selector);

// Produces an event sequence that concurrently merges the events received
↪  from the selected source sequences.
IObservable<R> SelectMany<T, R>(IObservable<T> source, Func<T,
↪  IObservable<R>> selector);
```

Note that the `Subscribe` method is effectively a "subscription factory" because it's a function that given an observable sequence and an observer instance returns a subscription object. With the introduction of subscription factories in IRP, observables have actually been demoted to a special kind of subscription factory.

```
IDisposable Subscribe<T>(IObservable<T> observable, IObserver<T> observer);
```

Observers

Observers act as sinks for events. The relationship between Rx and IRP is as follows:

```
// Rx
interface IObserver<T>
{
    void OnNext(T value);
    void OnError(Exception error);
    void OnCompleted();
}

// IRP
interface IAsyncReactiveQbserver<in T> : IExpressible
{
    Task OnNextAsync(T value, CancellationToken token = default);
```

```
    Task OnErrorAsync(Exception error, CancellationToken token = default);
    Task OnCompletedAsync(CancellationToken token = default);
}
```

Again, these interfaces are isomorphic modulo the use of `Task` return types.

> The design of this interfae predates the introduction of `ValueTask` which could provide a future enhancement. In fact, it would be worth reconsidering the duality with `IAsyncEnumerable<T>`, especially around notions of cancellation. The support for cancellation on these interfaces is to support cancellation of I/O operations when publishing events across boundaries.

Note that the `On*` methods lack a `Uri` parameter to associate an identifier with an event. It was deemed easier to let IRP systems decide on how (and if) they extract event identifiers from event payloads by establishing some conventions (cf. the Data Model described earlier in this document). In retrospect, this doesn't help for the punctuation events of `OnError` and `OnCompleted`, so a future iteration of IRP may provide an observer-like veneer on top of a more primitive layer where reified notification objects (with extrinsic identifiers) are used instead:

```
// Rx
class Notification<T>
{
    public NotificationKind Kind { get; }
    public bool hasValue { get; }
    public T Value { get; }
    public Exception Error { get; }

    void Accept(IObserver<T> observer);
}

// IRP (future)
class AsyncReactiveNotification<T>
{
    public Uri NotificationId { get; }

    public NotificationKind Kind { get; }
    public bool HasValue { get; }
    public T Value { get; }
    public Exception Error { get; }

    Task AcceptAsync(IAsyncObserver<T> observer, CancellationToken token =
↪   default);
}
```

> Note that observers are "hotter" than one may think at first. It's clear that an observer is hot if it's backed by a stream, given that streams are deemed to be hot. However, even single instances of observers are hot given that they maintain state about termination (error, completion) and about the sequential relaible processing of the events presented to them. As such, a future iteration or IRP will most likely introduce the notion of "observer factories". This will enable instantiation of observers outside the context of a subscription (e.g. to use an IRP system as an abstraction layer over sinks backed by various physical implementations such as reliable queues, databases, etc. with a unified observer interface over these). When instantiated in the context of a subscription, the lifetime of a hot observer instance is tied to that subscription.

Subscription factories

Subscription factories were a later addition to IRP, enabling users to define "templates" for subscriptions that have zero or more parameters. As such, they're merely functions with a return type compatible with the susbcription type.

```
// Rx
interface IDisposable
{
    void Dispose();
}

class Program
{
    IDisposable Create(DateTimeOffset dueTime, string path) =>
        Observable.Timer(dueTime)
                  .Subscribe(new FileObserver<long>(path));
}

// IRP
interface IAsyncReactiveQubscription : IExpressible
{
    Task DisposeAsync();
}

class Program
{
    Task<IAsyncReactiveQubscription> CreateAsync(Uri subscriptionId,
↪   DateTimeOffset dueTime, string path) =>
        AsyncReactiveQbservable.Timer(dueTime)
```

```
                                .SubscribeAsync(subscriptionId,
↪   AsyncReactiveQbserver.File<long>(path));
}
```

The way these client-side methods such as `AsyncReactiveQbservable.Timer` and `AsyncReactiveQbserver.File<T>` are implemented will be discussed later. In particular, we'll discuss how these methods are used as proxies to refer to reactive artifacts in a target IRP system, which are addressed using an extrinsic identifier.

> Note that Rx has an algebra over `IDisposable`, which acts as the type representing a subscription. It seems logical to have the isomorphic equivalent in IRP, namely an algebra for the `ISubscription` family of interfaces (such as `IAsyncReactiveQubscription`). Subscription factories allow for this but no IRP system today has taken advantage of this yet.

Subscriptions

Subscriptions are the IRP analog to `IDisposable` handles to running reactive computations. Subscriptions are created through the built-in `SubscribeAsync` method on observables, or through subscription factories. The definition of an `IAsyncReactiveQubscription` simply provides a way to dispose the subscription:

```
interface IAsyncReactiveQubscription : IAsyncDisposable {}
```

Stream factories

Stream factories were added early on during the design of IRP. It was immediately apparent that stream factories had a role to play in IRP because of the many existing subject types in Rx (e.g. `Subject<T>`, `AsyncSubject<T>`, `BehaviorSubject<T>`, and `ReplaySubject<T>`). In that sense, stream factories are merely function definitions for subject constructors.

Besides these essential subject implementations, and IRP system can define stream factories for streams that are backed by external services. For example, a stream factory could be parameterized on a connection string to some persistent reliable queue. Upon invoking the stream factory with a concrete value for a connection string, a stream is created in the IRP system that can be used both as an observable source and an observer sink, being physically backed by the reliable queue underneath. This enables the creation of "adapters" to existing event streaming systems in a way similar to sources and destinations in the SQL Server Integration Services (SSIS) Extract-Transform-Load (ETL) platform.

```
// Rx
interface ISubject<T> : ISubject<T, T> {}
interface ISubject<out T, in U> : IObservable<T>, IObserver<U> {}
```

```
class Program
{
    ISubject<T> CreateReplay(int count) => new ReplaySubject<T>(count);
}

// IRP
interface IAsyncReactiveQubject<T> : IAsyncReactiveQubject<T, T> {}
interface IAsyncReactiveQubject<out T, in U> : IAsyncReactiveQbservable<T>,
↪  IAsyncReactiveQbserver<U> {}

class Program
{
    Task<ISubject<T>> CreateReplay(Uri streamId, int count) =>
        AsyncReactiveQubject.Replay(streamId, count);
}
```

Streams

Streams support both subscription creation (the observable side) as well as event publication (the observer side). The interfaces are already shown above in the context of the discussion of stream factories, but one detail was omitted. In Rx, all concrete implementations of subjects implement `IDisposable`, but the `ISubject<>` interfaces don't. In IRP, the interface does implement `IAsyncDisposable` to enable the disposal of a stream in an IRP system:

```
interface IAsyncReactiveQubject<T> : IAsyncReactiveQubject<T, T>,
↪  IAsyncDisposable {}
```

In fact, all interfaces representing hot artifacts in IRP implement `IAsyncDisposable`. As such, the interface describing subscriptions is defined as follows:

```
interface IAsyncReactiveQubscription : IAsyncDisposable {}
```

> The use of `IAsyncDisposable` is the most straightforward way to map the Rx world to an isomorphic IRP world. However, it lacks the ability to provide additional parameters to the operation. Future iterations of IRP, discussed later on, may further decompose the formulation of the disposal intent from the submission of the operation to the target IRP system.

Expression normalization

Traditional LINQ has the notion of query providers through the `IQueryable<T>` and `IQueryProvider` interfaces. The role of query providers is to take the expression trees generated by the formulation of

query expressions and transform them into an executable format which is dependent on the provider. The best example is likely LINQ to SQL, as shown below:

```
var ctx = new DataContext();

var res = from p in ctx.GetTable<Product>("Products")
          where p.UnitPrice > 49.95m
          select p.ProductName;

foreach (var p in res)
{
    // ...
}
```

In here, the formulation of the query expression merely builds an object of type `IQueryable<string>` which gets assigned to the `res` variable. This object contains an expression tree in its `Expression` property, representing the query intent, which will roughly look like:

```
Call(
    methodinfoof(Queryable.Select<Product, string>),
    Call(
        methodinfoof(Queryable.Where<Product, Product>),
        Parameter(
            typeof(IQueryable<Product>),
            "Products"
        ),
        Lambda(
            Member(p, propertyinfoof(Product.ProductName)),
            p
        )
    ),
    Lambda(
        GreaterThan(
            Member(p, propertyinfoof(Product.UnitPrice))
            Constant(49.95)
        ),
        p
    )
)
```

Upon triggering the iteration through the `foreach` loop, the query provider associated with the query will transform the expression tree into an executable form in order to obtain materialized results in the form of objects yielded to the iteraton. In the case of LINQ to SQL, the expression tree gets translated into a semantically equivalent T-SQL query, like this:

```
SELECT p.ProductName FROM Products AS p WHERE p.UnitPrice > 49.95
```

In order to achieve this, the query provider's code runs locally and has built-in awareness of `MethodInfo` objects that represent standard query operators like `Where` and `Select` defined on the `Queryable` type. That's fine for a local translation of a query expression but breaks down when trying to serialize the expression tree between different systems, because it introduces a coupling with client-side APIs.

IRP solves this issue by representing all query operators as parameterized observable definitions represented using an identifier, as discussed earlier. This allows client libraries for IRP systems to normalize an expression from whatever language projection is appropriate (e.g. LINQ in C# and VB, fluent interface patterns in Java, JavaScript, Python, etc.) to a normal form. As an example, consider the following streaming equivalent query:

```
var ctx = new ClientContext(); // IRP

var res = from p in ctx.GetObservable<Product>("Products")
          where p.UnitPrice > 49.95m
          select p.ProductName;

await res.SubscribeAsync(...);
```

Focusing purely on the query expression formulated in the `IAsyncReactiveQbservable<T>` space (rather than `IQueryable<T>` in LINQ to SQL), we end up with an expression representation that looks like this:

```
Call(
    methodinfoof(AsyncReactiveQbservable.Select<Product, string>),
    Call(
        methodinfoof(AsyncReactiveQbservable.Where<Product, Product>),
        Parameter(
            typeof(IAsyncReactiveQueryable<Product>),
            "Products"
        ),
        Lambda(
            Member(p, propertyinfoof(Product.ProductName)),
            p
        )
    ),
    Lambda(
        GreaterThan(
            Member(p, propertyinfoof(Product.UnitPrice))
            Constant(49.95)
```

```
        ),
        p
    )
)
```

In order to normalize this expression into a client language-agnostics format, we need to shake off all references to reflection that's client-specific, in particular the `MethodInfo` objects representing query operators. This is achieved by associating an identifiers with these methods:

```
static class AsyncReactiveQbservable
{
    [KnownResource("rx://operators/filter")]
    public static IAsyncReactiveQbservable<T> Where<T>(this
    ↪  IAsyncReactiveQbservable<T> source, Expression<Func<T, bool>>
    ↪  predicate) { ... }

    [KnownResource("rx://operators/map")]
    public static IAsyncReactiveQbservable<R> Select<T, R>(this
    ↪  IAsyncReactiveQbservable<T> source, Expression<Func<T, R>>
    ↪  selector) { ... }
}
```

The normalizer will pick up on these `KnownResource` attributes and turn `MethodCallExpression` nodes into `InvocationExpresson` nodes where the `MethodInfo` gets subsituted for an unbound `ParameterExpression` using the identifier specified. For example:

```
Invoke(
    Parameter(
        typeof(Func<IAsyncReactiveQbservable<Product>,
        ↪  Expression<Func<Product, string>>,
        ↪  IAsyncReactiveQbservable<string>>),
        "rx://operators/map"
    ),
    Invoke(
        Parameter(
            typeof(Func<IAsyncReactiveQbservable<Product>,
            ↪  Expression<Func<Product, Product>>,
            ↪  IAsyncReactiveQbservable<Product>>),
            "rx://operators/filter"
        ),
        Parameter(
            typeof(IAsyncReactiveQueryable<Product>),
            "Products"
        ),
```

```
        Lambda(
            Member(p, propertyinfoof(Product.ProductName)),
            p
        )
    ),
    Lambda(
        GreaterThan(
            Member(p, propertyinfoof(Product.UnitPrice))
            Constant(49.95)
        ),
        p
    )
)
```

The use of `Mapping` attributes on the properties of the Data Model compliant `Product` type are further used to remove the dependency on `PropertyInfo` objects, in a way similar to `KnownResource` for the methods shown above. The result is an expression that's independent of client-only reflection objects.

Note that the use of `KnownResource` is merely a client-side convenience mechanism in order to make expressing query intent as natural as possible for clients. IRP clients written in languages and frameworks other than C#, Visual Basic, F#, and .NET can leverage the mechanisms available in such environments to achieve similar effects. Given that this facility of using metadata attributes to drive normalization is a syntactic sugar veneer on top of the core IRP facilities, it's possible to write the same query as above using explicit calls to `GetObservable` methods to obtain proxies to the query operators:

```
var ctx = new ClientContext(); // IRP

var products = ctx.GetObservable<Product>("Products");
var where = ctx.GetObservable<IAsyncReactiveQbserver<Product>,
↪   Expression<Func<Product, bool>>, Product>(new
↪   Uri("rx://operators/filter");
var select = ctx.GetObservable<IAsyncReactiveQbserver<Product>,
↪   Expression<Func<Product, string>>, string>(new
↪   Uri("rx://operators/map");

var res = select(where(products, p => p.UnitPrice > 49.95m), p =>
↪   p.ProductName);

await res.SubscribeAsync(...);
```

This will produce exactly the same normalized expression tree and acts as an example to show how

a well-designed language projection (such as fluent interface patterns with extension methods and query expression syntax in C#) can hide a lot of the complexity in formulating user intent.

Besides the erasure of references to reflection objects in favor of invocation expressions referring to unbound global parameters (identifying artifacts such as sources, sinks, and operators), the normalization also includes a variety of other steps:

- Local partial evaluation of subexpressions that are free of side-effects.
- Various forms of (optional) expression optimization.
- Allowlist scanning of functionality used in the expression.
- Normalization of function invocations by rewriting n-ary function invocations to tuple-based invocation of unary functions.

> Future iterations of IRP (and more generally ICP for arbitrary computation expressions) will likely contain further mapping freedom in order to increase the friendliness of the language projection while retaining a stable normal form. For example, various operators can benefit from being treated as curried functions, thus enabling partial application to define specialized macros, or a more natural projection in other languages (e.g. F# or Scala where functions are curried). Rather than only normalizing operator invocations using a "tuple normal form", this introduces the addition of a "curry normal form".

Bonsai

In order to support the transport query intent between clients and services, we had a choice of either inventing a new language for event processing, or building a general-purpose mechanism to represent expression trees (similar to ASTs).

Back in the SQL organization, we had built an XML-based expression serializer which suffered from various limitations. First and foremost, it was tied to the expression tree APIs in the .NET Framework, so it wouldn't be a natural fit for other languages such as JavaScript or C++. Second, many traits of nominal static typing were inherited from its .NET roots, making it less applicable in the context of data processing with structural typing being a better default.

To address these shortcomings, the Bonsai project was started by Bart De Smet, first in the context of enabling a language-agnostic and transport-friendly format to transfer `IQueryable<T>` queries to the graph database. Given that Bonsai trees can represent arbitrary expressions, it was readily applicable to other query domains, including reactive event processing. As such, it become the obvious choice to address expression serialization needs when starting the IRP project.

> To illustrate the project's name - referring to tiny well-groomed trees - Bart went to a local garden center's nursery in Bellevue and bought a little Bonsai tree to put on his desk. During one of Bart's summer vacations, a colleague took care of the tree. Even though the original tree died, many of its children are now groomed by Bart's colleague. This almost symbolically reflects the technological trajectory of Bonsai which has been augmented and put to use in other services.

The rationale of sticking with a general-purpose representation of an expression tree rather than inventing a custom language is simple. The creation of new languages is unproductive, requiring huge investments in the design of a semantically sound language, massive amounts of solid tooling going all the way from compilers to editors, and a steep learning curve for users. Such external domain specific languages (DSLs) are often misguided, short-lived, and often end up in opaque strings in general-purpose code (think of SQL statements in string literals, or string-based expression languages in XML attributes, JSON or YAML documents, etc.) where no tooling can go. By focusing on internal DSLs instead, we can piggyback on the host language of the user's choice.

Bonsai trees are optionally statically typed representations of expression trees. The default of encoding is JSON (a binary variant was built some years later) using arrays to represent nodes of the tree. Each such array consists of a first element holding a discriminator tag, followed by elements specific to the node type, including children, optional type info, etc.

For example, the representation of a constant value 42 looks like this:

```
["::", 42]
```

In here, the `::` discriminator is used to denote a constant, and the value is stored in the second element slot. The choice to deserialize the constant's value is not specified by Bonsai and could be embedded JSON, a base64-encoded representation of a binary format, a string literal holding XML, etc. It's up to the parties participating in an exchange of Bonsai trees to agree on the serialization format of constant values.

Bonsai trees are optionally statically typed. In the example above, the value `42` may be treated as any number for use by e.g. a JavaScript or Python client. However, if we wish to add more static type info, Bonsai does support an optional third slot for this node type to do so. For example:

```
["::", 42, 0]
```

In here, `0` is an index in a type table held in the so-called (optional) "reflection context" that's sent alongside the tree. If we zoom out a bit, the full Bonsai tree looks like:

```
{
    "Context":
    {
        "Types":
```

```
        {
            ["::", "System.Int64"]
        }
    },
    "Expression": ["::", 42, 0]
}
```

The representation of a primitive type is (coincidentally) done using the `::` discriminator as well, followed by a name for the type. Again, Bonsai doesn't specify the syntax of type names, and it's up to the parties exchanging a Bonsai tree to agree upon type names. In this case, the use of `System.Int64` reveals a .NET background, though serializers and deserializers have the freedom to map this to any (semantically equivalent) type in their language or runtime.

> Various concrete uses of Bonsai with exchange of expressions across .NET and C/C++ runtimes have opted for the use of "neutral" types that are syntactically non-existent in either environment, but can be mapped to a known type in both. This is very similar to the use of `Mapping` attributes to denote properties as being well-known according to some agreed-upon schema repository or ontology. For example, a 64-bit signed integer could be represented as `type://primitive/integer/signed?bits=64` or just `signed_int64`. At the end of the day, this is coloring the bikeshed.

A slightly more complex example of a Bonsai tree (omitting the reflection context) is shown below:

```
[
    "+",
    ["$", "x"],
    ["::", 1]
]
```

This tree represents `x + 1` and shows other discriminators such as `+` for addition and `$` to refer to variables or parameters. Many more discriminators exist for e.g. member lookup, function invocation, various arithmetic operators, etc.

> Note that the use of discriminators or tags is completely analogous to the concept of e.g. S-expressions in LISP.

The origin of Bonsai is inspired by the expression tree APIs in .NET which are used by a variety of front-end languages including C#, Visual Basic, F#, IronPython, and IronRuby. These APIs are rather general-purpose and widely applicable to a wide range of languages. There's roughly a 1:1 correspondence between Bonsai discriminators and node types in the `System.Linq.Expressions` APIs in the .NET Framework. Starting with .NET 4.0, these APIs include support for statement constructs such as

blocks and loops. A later revision of Bonsai has added support for these as well, so it'd be better to refer to Bonsai trees as a general-purpose representation of statement trees.

Bonsai enables the creation of language projections whereby means to represent code as data in any host language in the most natural and user-friendly way can be mapped onto a normalized expression representation for code exchange. For example, in C# one would use transparent conversion of lambda expressions to expression trees to capture user intent:

```
// int Calculate(Expression<Func<int, int, int>> expression, ...)
calculatorService.Calculate((x, y) => x * 2 + 1 - y, ...)
```

In languages such as F#, explicit quotations could be used:

```
// or <@@ ... @@> for untyped quotations
calc(<@ (x, y) -> x * 2 + 1 - y @>, ...)
```

In languages such as JavaScript, not direct quotation support exists, but users have gotten accustomed to `eval`-like APIs that take a piece of code a string and see it getting parsed at runtime (e.g. using esprima):

```
calc("function (x, y) { return x * 2 + 1 - y; }", ...)
```

Finally, combining Bonsai with the IRP normal form representation of reactive event processing query expressions, results in "queries" that look like this:

```
[
    "()",                                   // function invocation
    ["$", "rx://operators/filter"],         // identifier of "Where"
    [
        ["$", "bing://streams/weather"],    // identifier of a weather
↪   stream
        [
            "=>",                           // lambda
            [
                ["$", "w"]                  // single parameter "w"
            ],
            [
                ">",                        // greater than
                [
                    ".",                    // member lookup
                    ["$", "w"],             // on parameter "w"
                    "schema:/weather/temp" // for temperature
                ],
                [
                    "::",                   // constant
```

```
                    25                      // with value 25
                ]
            ]
        ]
    ]
]
```

This expression represents the normalized equivalent of a query that could look as follows in C#:

```
ctx.GetObservable<Weather>("bing://streams/weather")
   .Where(w => w.Temperature > 25)
```

or, using a specialized context object with mappings for discovered streams,

```
ctx.Weather // annoted with [KnownResource("bing://streams/weather")]
   .Where(w => w.Temperature > 25)
```

The type system of Bonsai (cf. the optional reflection context) supports primitive types, arrays, generic types, and structural types. In the example above, the `Temperature` field was normalized to an identifier through the use of the Data Model library:

```
class WeatherInfo
{
    [Mapping("schema:/weather/temp")]
    public double Temperature { get; set; }
}
```

When type information is omitted from Bonsai trees, various techniques can be used to evaluate the resulting expression. One approach is to be fully dynamic and perform late binding. Another is type inference, for example by discovering the type of the events produced by the weather stream, which can subsequently flow to the `Where` operator's input type, thus allowing to discover the type of the temperature field.

In case type information is present, the parameters in the expression contain a reference to type info, for example:

```
["$", "bing://streams/weather", 3]
```

with the `Types` table in the reflection context looking roughly like this:

```
{
    "Types":
    [
        /*0*/   ["::", "System.Double"],          // primitive double
        /*1*/   [
                    "{;}",                        // record type
```

```
                    ["schema:/weather/temp", 0]  // "double temp"
                ],
        /*2*/   ["::", "IObservable`1"],         // IObservable<T>
        /*3*/   ["<>", 2, [1]]                   // IObservable<{...}>
    ]
}
```

In here, the first element represents the primitive type for a `double` floating point, the second element represents a record type with a temperature field of type `double` (using an index-based reference), the third element represents an open generic type representing an `IObservable<T>` sequence, and the fourth element represents a closed generic instantiation of the open generic at index 2 using the type parameter at index 1. This is the structurally typed equivalent of `IObservable<WeatherInfo>`.

> In the current IRP normal form used by Reaqtor, the representation of reactive entities such as observables, observers, and subscriptions are done using the `IAsyncReactiveQ*` family of types. In the example above, rather the using `IObservable<T>`, the tree would use `IAsyncReactiveQbservable<T>` instead. This is another example of the domain-agnostic approach of Bonsai where IRP is layered on top and makes decisions about the normal format of its language encoding.
>
> In later iterations of IRP, efforts have been made to further normalize references to nominal types (such as `IObservable<T>`) using contract types. This effectively introduces a mapping of type names to identifiers in a way similar to mapping members and methods to identifiers using attributes such as `KnownResource` and `Mapping`. In this representation, the normal form of the example above refers to a type identified as `CObservable<T>` instead, given producers and consumers the freedom to map this identifier on the most natural type available to them.

Finally, Bonsai trees in .NET come with an object model that's often referred to as "slim expression trees" using a type called `ExpressionSlim`. This type and its type hierarchy are roughly equivalent to the `System.Linq.Expressions.Expression` type family but with one main difference. Rather than referring to .NET reflection objects, slim expressions come with an object model that's capable of representing optional static typing using structural types. This type system is represented using APIs such as `TypeSlim`, `MemberInfoSlim`, etc. which are variants of the `System.Reflection` APIs that represent mandatory static typing using nominal types.

Conversions between the "fat" world of .NET expression trees with .NET reflection, and the "slim" space of Bonsai trees with optional typing, are provided by the library:

- Going from "fat" to "slim" involves normalization of expressions and types, e.g. mapping nominal Data Model types to their equivalent structural type representation.
- Going from "slim" to "fat" involves binding steps (optionally using type inference or late binding) and mapping of structural types to nominal types with a compatible shape.

Examples of these transformations are the back-end of a client library, and the front-end of a service library, respectively. Normalization steps carried out by a client library are effectively "unbind" steps that can leverage metadata annotations to decouple the user-friendly representation of intent from the underlying normal form. Binding steps carried out by a service library can leverage catalogs or registries of definitions to map identifiers onto implementations, also allowing for the discovery of type information. In this minimalistic world view, an IRP-compatible execution engine is merely an expression binder, compiler, and evaluator.

Note that this layering approach enables a variety of client libraries and a variety of service implementations with a single normal form in the middle. In the context of IRP and Reaqtor this has enabled the use of e.g. .NET clients that can submit reactive computations to .NET-based cloud services as well as C++-based implementations of reactive engines running on devices.

The three aspects of IRP, take two

Earlier, we mention that IRP has three aspects: proxies, definitions, and metadata. We'll discuss these briefly before diving into some specific design goals.

Proxies

The first aspect of IRP is in its ability to enable composition of intent by using proxies to artifacts. One can view these proxies are holders of identifiers (enabling service-side binding steps) and builders for expressions that represent the intent. Concretely, this gets reflected in the `Get*` methods on an IRP context, for example:

```
IAsyncReactiveQbservable<T> GetObservable<T>(Uri observableId);
IAsyncReactiveQbserver<T> GetObserver<T>(Uri observerId);
IAsyncReactiveQubscription GetSubscription(Uri subscriptionId);
```

When accessing the `Expression` property on a proxy, one will find a `ParameterExpression` whose `Name` refers to the identifier of the artifact that was passed to the `Get*` method.

Applying query operators results in the creation of bigger expression trees that compose the expressions of the operator's operands, which subsequently get encapsulated in an object that can be used for further composition:

```
var numbers = ctx.GetObservable<int>(new Uri("eg://numbers"));
// numbers.Expression
// ~ Expression.Parameter("eg://numbers")

var positive = numbers.Where(x => x >= 0);
```

```
// positive.Expression
// ~ Expression.Call(
//       Where,
//       Expression.Parameter("eg://numbers"),
//       Expression.Lambda(...)
//   )

var negative = positive.Select(x => -x);
// negative.Expression
// ~ Expression.Call(
//       Select,
//       Expression.Call(
//           Where,
//           Expression.Parameter("eg://numbers"),
//           Expression.Lambda(...)
//       ),
//       Expression.Lambda(...)
//   )
```

It's important to realize that composition of artifacts happens entirely in-memory and never communicates with a service. After composing the intent, it can be used to perform a DDL operation such as the creation of a hot artifact (e.g. a subscription or stream) or the definition of a cold artifact (e.g. a parameterized observable or observer). For example:

```
await negative.SubscribeAsync(new Uri("eg://sub/neg"), ...);
```

Definitions

The second aspect of IRP enables the definition of new artifacts, which may compose over existing artifacts and introduce parameters. For cold artifacts, definitions can be performed using `Define*` methods:

```
Task DefineObservableAsync<T, R>(Uri uri, Expression<Func<T,
↪  IAsyncReactiveQbservable<R>>> observable);
Task DefineObserverAsync<T, R>(Uri uri, Expression<Func<T,
↪  IAsyncReactiveQbserver<R>>> observer);
...
```

For example, one could define a high-temperature operator as follows:

```
await ctx.DefineObservableAsync<double, WeatherInfo>(new
↪  Uri("eg://weather/hightemp"), t => ctx.Weather.Where(w => w.Temperature
↪  > t));
```

Once a definition is submitted to a service, the defined artifact can be referred to using the specified identifier.

> IRP does not have any notion of built-in operators unlike traditional database systems where various operators such as filtering and projection are treated as primitives that are built-in. In the world of IRP, every operator ultimately gets defined. Some operators can be defined in terms of existing ones (e.g. `Min` in terms of `Aggregate`) while others are primitives that get defined using an expression that refers to a concrete implementation (backed by opaque code in an operator library hosted by the service).

Metadata

The final aspect of IRP is its self-descriptive nature using metadata APIs, similar to a catalog in a database system. Metadata APIs are exposed as queryable collections, for example:

```
IAsyncQueryable<ObservableDefinition> Observables { get; }
IAsyncQueryable<ObserverDefinition> Observers { get; }
IAsyncQueryable<SubscriptionProcess> Subscriptions { get; }
...
```

Hot artifacts are represented using entities with a `Process` suffix, while cold artifacts used a `Definition` suffix. These entities contain properies such as `Id`, `Expression`, `CreationTime`, etc. and are treated as structural types. This enables concrete implementation of IRP-compliant service to add more properties of their choice (e.g. security information, owners, etc.).

> Note that the collection types shown above use the asynchronous equivalent of `IQueryable<T>` which didn't exist at the time of the creation of IRP. Work is underway to have asynchronous enumerable support in future releases of C# and the code fragment above shows the ultimate shape of the API. When browsing current APIs, traces of synchronous interfaces may be found.
>
> The current implementation of the IRP metadata interfaces also uses queryable dictionary types. This is merely a choice of a language projection to make querying by an identifier easier using indexing syntax. The essence of metadata is in queryable collections but more structure can be imposed on top of the set of artifacts, e.g. to index by identifier and expose such organization as a dictionary.

These queryable collections can be used to formulate metadata queries, ranging from simple selections of artifacts using known identifiers to inspection of the expression trees used for their definition or creation. For example:

```
var filter = await ctx.Observables.SingleAsync(o => o.Id == new
↪  Uri("rx://operators/filter"));
```

```
Console.WriteLine(filter.Expression);

var allOperators = ctx.Observables.Where(o => o.Id.Schema == "rx");
foreach await (var op in allOperators)
{
    Console.WriteLine(op);
}

var allSubscriptionsOnWeather = from s in ctx.Subscriptions
                                let vs = s.Expression.FindFreeVariables()
                                where vs.Any(v => v.Name ==
↪   "bing://streams/weather")
                                select s.Id;

foreach await (var subId in allSubscriptionsOnWeather)
{
    Console.WriteLine(subId);
}
```

In the last example shown above, `FindFreeVariables` is defined as an extension method on `ExpressionSlim` annotated with a `KnownResource` attribute to refer to a service-side implementation of this functionality. Given that all artifacts referenced from an expression are represented as unbound parameters, `FindFreeVariables` can be used to perform dependency analysis. It's important to note that this query is entirely executed service-side.

> At the time of conception of IRP, it was envisioned that all artifacts would be stored in a graph database, making the traversal of dependencies as edges rather straightforward. This goes back to the common roots between IRP and the graph database effort, and even further to other Cloud Programmability Team projects mentioned early on in this document.
>
> In order to support IRP metadata well, the choice of the underlying metadata store used by a service is critical. When work on IRP started, some metadata stores were implemented in Azure Table or Azure Blob (to store larger Bonsai trees), which provide very limited query capability. In today's world, a natural choice to support rich query capabilities is the use of Azure CosmosDB with service-side stored procedures and a set of well-known user defined functions (such as `FindFreeVariables`) that operate on the Bonsai representation, possibly with various indexing schemes that are applied upon insertion of new documents.

The most straightforward use cases of metadata are:

- Perform (non-atomic) "check-before-create" operations.
- Query artifacts based on additional metadata, e.g. a tenant, a user, a scenario, etc.

- Export of artifacts for offline analysis.

However, in addition to these, the main motivations for a first-class metadata API exposed by IRP are twofold:

- Enabling the creation of rich developer tooling.
- Support for interoperability between IRP-compliant systems.

We'll focus on the latter separately further on in this document. With regards to developer tooling, the best way to think of metadata is the equivalent to a catalog in an RDBMS such as SQL Server, enabling the creation of tools such as O/R mappers.

In particular, IRP supports a tool called `rxmetal` for "Reactive Extensions Meta Language" which is effectively an O/E (object/event) mapper. It's the equivalent to `sqlmetal` in LINQ to SQL which is an O/R (object/relational) mapper.

Such tools can power experiences a la "Server Explorer" in Visual Studio, enabling one to connect to an IRP instance and browse the different artifact types. A typical example of this is discovery of streams (~ tables), the discovery of observables (~ views, stored procedures), etc. that are available. A drag-and-drop experience to a designer surface can then trigger code-generation of an object model.

The `rxmetal` tool emits a description of the artifacts in an XML-based format called RXML (analogous to DBML in `sqlmetal`) from which C# or VB code can be generated. The generated code defines a derives client context that provides easy accessors to the imported artifacts, e.g.:

```
class MyClientContext : ClientContext
{
    public IAsyncReactiveQubject<WeatherInfo> Weather { get; } = ...
}

class WeatherInfo
{
    [Mapping("schema:/weather/city")]
    public string City { get; set; }

    [Mapping("schema:/weather/temp")]
    public double Temperature { get; set; }
}
```

This enables users to write straightforward queries without having to use calls to `GetObservable<T>` etc.

```
var ctx = new MyClientContext(...);

var res = from weather in ctx.Weather
```

```
            group weather by weather.City into g
            from hourly in weather.Window(TimeSpan.FromHours(1))
            from avg in hourly.Average()
            select new { City = weather.City, AvgTemperature = avg };

await res.SubscribeAsync(...);
```

> Note that IRP does not have any built-in query operators. Because of this, tools like `rxmetal` can export the definition of query operators as extension methods just like any other (user-defined) artifact. This eliminates the need to ship a client library where all supported service-side query operators have an API presence, because users can generate these methods as they browse the catalog of available artifacts.
>
> For example, the `Where` and `Select` operators would show up right alongside user-defined operators such as `WhenTemperatureIsHigh`. It goes without saying that extra metadata (such as tag clouds, with `#` tags for classification and `@` tags for owner or tenant info) could be used to present operators in a more structured and hierarchical way. Alternatively, an IRP system may impose a structure on the identifiers used for artifacts, introducing the concept of namespaces (e.g. `rx://` is reserved for general purpose query operators). Strucutures similar to those leveraged by catalogs in database systems (e.g. schemas and owners in SQL Server) can be applied in IRP as well as a policy. Metadata in IRP merely provides a mechanism for discovery.

Delegation

In addition to enabling rich tooling, one of the goals of having first-class metadata and discovery support in IRP was to enable interoperability and delegation across IRP-compliant systems.

By providing the ability for an IRP system to discover the artifacts defined in another IRP system, rich query planning and distribution capabilities can be built. This can range from simple "meta-IRP" services that aggregate the catalog of various linked or registered IRP services (similar to "linked servers" in SQL Server), all the way to distributed streaming event processing by splitting and rewriting expressions for placement on IRP services.

As an example, consider the following event processing query expression:

```
var bartHome = ctx.Bart.GeoLocation.Select(g => home - g.Location < 100 *
↪  METER).DistinctUntilChanged();
var lisaHome = ctx.Lisa.GeoLocation.Select(g => home - g.Location < 100 *
↪  METER).DistinctUntilChanged();

var bothHome = bartHome.CombineLatest(lisaHome, (b, l) => b &&
↪  l).First(both => both);
```

Using the stream of geo-coordinates of Bart and Lisa, we calculate the distance between their current location and home, to determine whether they have arrived at home using a projection (cf. `Select`). We also filter out adjacent duplicates using `DistinctUntilChanged`, which gets us the "edge" events that correspond to entering home (`false` to `true`) and leaving home (`true` to `false`).

Next, we combine these two queries using `CombineLatest` to generate events that represent whether both are home, selecting the first event that indicates both are home using `First`.

In one possible implementation, we have the streams of geo-locations for both Bart and Lisa defined in a single IRP service, where we can perform all the processing centrally. However, it'd be more efficient for the distance to home calculations and the filtering of duplicate events to happen on Bart's and Lisa's devices, feeding off the GPS sensor directly. This reduces the transport of events across nodes, in this case devices and a cloud service. The concept of metadata-driven delegation in IRP enables this scenario.

In the example shown above, the `ctx` instance acts as the root for a cloud of IRP-compliant services, allowing traversal into specific services such as the IRP instance for `Bart` and `Lisa`. This hierarchical approach is just one way of representing the world; an alternative would be to present a flat world and have the binder of the target IRP service discover the location of artifacts in a DNS-like fashion for logical identifiers. The hierarchical approach with traversal patterns (such as `ctx.Bart.GeoLocation`) reveals another place in the IRP thinking where we aligned with the reactive graph database idea where entities (such as `Bart`) can expose raw data as well as traversals into data at rest (`IEnumerable<T>`) or data in motion (`IObservable<T>`).

In this world view, IRP services themselves are first-class artifact types in IRP; they are represented using an identifier just like any other artifact (such as streams, subscriptions, etc.). A context object is represented by an expression representing a traversal pattern that encodes a resource hierarchy. In the example shown above, both `Bart` and `Lisa` are proxies to (virtual) IRP services exposing the reactive artifacts made available for them.

Binding of expressions like the one shown above leverages lookups of artifacts using linked catalogs, allowing for the traversal of hierarchies of identifiers. In the example shown above, the target service represented by `ctx` has capabilities to look up the IRP instance denoted by `Bart` (which has an identifier set through a `KnownResource` attribute). By subsequently consulting the metadata service of that target IRP instance, the `GeoLocation` artifact can be looked up (similarly using an artifact identifier).

Next, the operators applied to these sources (cf. the use of `Where`, `DistinctUntilChanged`, and `CombineLatest` in the example above) could be applied in the IRP service represented by `ctx` by streaming the data to a central location. However, IRP's metadata API can also be used to discover the availability of such operators in the IRP instances associated with `Bart.GeoLocation` and `Lisa.GeoLocation`. Similarly, functions referenced by predicates, selectors, etc. passed to such

operators can also be discovered, effectively enabling rudimentary capability-based query planning and the creation of distributed execution plans.

> A true distributed implementation of IRP contains additional out-of-band mechanisms to decide on the economics of a distributed execution plan, beyond mere capability-based tiling of expressions. These can be based on conventions about availability of additional properties exposed on the metadata entities (e.g. volume and velocity estimations for streams, cost of transporting data, security requirements, etc.) or the availability of additional services that can be used to aid in (cost-based) query planning.

This example of query planning was first conceived when onboarding Cortana scenarios where queries such as "time to leave" (for an appointment) require the use of geo-coordinate data on users' devices as well as events related to traffic conditions. By modeling stream processing capabilities on the device (cf. GPS sensor data) and in the cloud (cf. public traffic data) using the same unifying IRP abstraction, we were able to split a query expression into subexpressions that can run on different IRP implementations.

> Today's implementation of Reaqtor has limited built-in support for distributed delegation (though service implementations built on top of the Reaqtor framework have built such capabilities) but used the same technique for in-memory delegation of query operators into stream proxies through a mechanism called Partitioned Multi Subject (PMS). When applying a `Where` filter to a stream that's PMS-compatible, a delegation dialogue is set up to attempt to delegate the filter conditions to the stream's in-memory proxy implementation for efficient evaluation as a decision/routing tree.
>
> For example, naive executon of a query like `weather.Where(w => w.City == "Seattle")` would result in evaluation of the predicate for each event. When many such subscription exist, this results in a linear evaluation of all predicates for each event. The local delegation employed by PMS enables pushdown of the predicate to the `weather` stream where a dictionary-based evaluation tree is built. For every event received, the `City` property is extracted and checked against a dictionary entry matching the value (e.g. `Seattle`) in order to retrieve downstream observers. This leads to sub-linear evaluation cost.

The dot-pipe equivalence

One way of thinking about delegation is the "dot-pipe equivalence". The syntactic fluent composition of query operators using `.` syntax can be seen as an opportunity to decompose and distribute execution of the computation. In order to do so in a stream processing platform, a data flow channels needs to be inserted to transport the events from the upstream computation to the downstream residual computation.

As an example, consider the following academic query:

```
xs.Where(x => x > 0).Select(x => x + 1).Subscribe(o)
```

Often, a stream processing platform has an "ingress" and "egress" system responsible for receiving events from the stream `xs` and to send events to the observer `o`. Essentially, this establishes two implicit pipes already, on the edge of the stream processing platform and the outside world.

However, the execution itself can also be distributed by rewriting the query expression into subquery expressions. In the example above, the maximum decomposition (ignoring edges to the outside world, i.e. publishers into `xs` and receivers on `o`) would look as follows:

```
var d1 = t1.Subscribe(o);
var d2 = t2.Select(x => x + 1).Subscribe(t1);
var d3 = xs.Where(x => x > 0).Subscribe(t2);
```

The representation shown above is plain Rx syntax; the IRP equivalent of asynchronous subscriptions with explicit identifiers should be apparent:

```
var d1 = await t1.SubscribeAsync("d1", o);
var d2 = await t2.Select(x => x + 1).SubscribeAsync("d2", t1);
var d3 = await xs.Where(x => x > 0).SubscribeAsync("d3", t2);
```

In this execution plan, `t1` and `t2` are intermediate streams which are the pipe equivalent to the dots in the original query expression:

```
xs.Where(x => x > 0) ()==t2==() Select (x => x + 1) ()==t1==() Subscribe(o)
```

These can be created as internal-only streams within the IRP service, specialized towards a 1:1 communication across computation nodes. Lifetime of these pipes can be managed by further rewriting the query expression using operators such as `Finally` (to add terminal side-effects to delete pipes) and `Defer` (to add initial side-effects to create pipes).

> Note this is very similar to distributed query execution planning for data at rest, e.g. in a map/reduce framework where intermediate collections are allocated to hold the results of evaluating subexpressions of the overall query intent. Various approaches can be used to allocate/deallocate such intermediate scratch pads: a central coordinator, a peer-to-peer approach (upstream allocates, downstream deallocates), or even a lifetime-based GC approach (less applicable to continuous streaming systems, though lease-based techniques could be used).

In the context of delegation, subexpressions get delegated to other IRP systems, thus requiring a data flow channel to be created between those systems, whose lifetime is controlled by the lifetime of the overall query. The use of IRP abstractions provides the necessary tools to be able to do this,

namely discovery of capabilities in IRP services, the ability to define artifacts for subexpressions being delegates, and the ability to obtain proxies to artifacts defined in IRP services in order to compose queries and perform DDL operations to create (start) and delete (stop) hot running computations.

In order to enable the creation of usable pipes to cross the boundary between IRP systems, such systems can agree on the availability of observable and observer proxies that enable 1:1 pipes to receive and send events to other IRP services, for example backed by HTTP-based pipes, web hooks, intermediate reliable and persisted queues, etc. Again, the metadata capability can be used to inquire about available pipes on both ends in order to determine a suitable means to set up the data flow, effectively achieving a handshake. Metadata associated with discovered artifacts can further reveal information about supported encryption, available QPS, reliability guarantees, etc.

Reliable Rx

IRP is part of the core framework libraries used by streaming event processing systems such as Reaqtor. The main role of IRP is to define an abstraction for streaming event processing systems, enabling service implementations to achieve “Reactive as a Service” (RaaS).

In addition to the IRP abstractions and implementation building blocks, the core framework also provides a variety of utilities to manipulate expressions (to aid in normalization, binding, compilation, and evaluation), and implementations of portable event processing engines that can be embedded in applications and services.

When building a core event processing engine based on the Rx algebra, a number of key properties came up:

- Highly portable to enable hosting in services, applications, and devices.
- Minimal dependencies and proper abstractions for I/O etc. for easy integration.
- Reliability of event processing in the case of compute node failure or swap out.
- High density to host millions of standing queries on compute nodes in the cloud.

The initial implementation efforts for the core reactive engine started in 2013 in a collaboration between Bart De Smet and Brian Beckman where we mainly focused on the reliability aspects of the core engine.

> It is worth pointing out that the landscape of event stream processing in early 2013 was vastly different from what we take for granted now, in 2021. One of the “state of the art” systems back then was Storm. There were major design goal differences between technologies like Storm versus Reaqtor, and the approach to reliability was one of them.

In order to achieve reliability, we quickly landed on the necessity for introducing sequence identifiers

for events. Early prototypes explored the option to flow sequence identifiers through the Rx algebra using a reliable counterpart to the Rx abstractions:

```
interface IReliableObserver<in T>
{
    void OnNext(long id, T value);
    void OnError(long id, Exception error);
    void OnCompleted(long id);
}

interface IReliableObservable<out T>
{
    IReliableDisposable Subscribe(IReliableObserver<T> observer);
}

interface IReliableDisposable : IDisposable
{
    void Start(long sequenceId);
    void Acknowledge(long sequenceId);
}
```

This prototype led to a couple of realizations:

- Composing operators over these abstractions has limited mileage, especially for complex operators such as `Window` and `Merge` which exhibit higher order sequences. In particular, ensuring monotonically increasing sequence identifiers isn't trivial without assumptions on logging capabilities for use by operators, introducing I/O requirements in the core event processing pipeline. Vector clocks were considered but state space isn't bounded; alternatives such as interval tree clocks can help to reduce state explosion but introduce additional complexity.
- Reliability of event delivery and processing is only one part of the puzzle; the ability to summarize operator state periodically rather than relying on unbounded replay is essential to prune the state space. To support this, operators need to provide means to persist state and running computations need a way to prune the replay buffer after successfully persisting state.
- The role of `IDisposable` in Rx is really a means to visit an operator graph for purposes of stopping the computation and disposing resources. For more complex implementations, such as reliable Rx, the lifecycle of query operator instances requires additional visitor passes, for example to restart the computation after a failover by requesting replay from a specified sequence identifier, or to request persistence or restoration of operator state.

Based on these observations, we landed on a design where we introduced a crisp boundary between the core operator library and the hosting environment. Concerns around managing the lifecycle of a query expression are left to the hosting environment, including:

- Binding and compilation of query expressions.
- Facilities to request replay of events from sources.
- State persistence implementations to store and recover operator state.

This reliefs the core query operator library from having to deal with sequence identifiers, which are maintained on the edge between the operator library and the hosting environment. In other words, the public interfaces for reactive artifacts were kept closely aligned with public Rx, while the interfaces for use by the hosting environment incorporate various crosscutting concerns.

> Note that this separation of concerns is very similar to the approach taken by Rx to deal with virtual time or application time. Unlikely other systems that flow "events with timestamps" through operators, Rx can use schedulers to establish an order for the events flowing in to a computation graph. This also forms the foundation for LINQ to Traces (Tx), as mentioned before.

Ultimately, we landed the `IReliable` interfaces (with some minor differences) shown above to support cross-engine communication, supporting a checkpoint-and-replay reliability model for the query evaluators. Incoming observable sequences support sequence IDs and replay (provided by upstream nodes), while outgoing observers introduce sequence IDs (for use by downstream nodes), allowing evaluators to fail independently. As events come in, the query evaluator shakes off sequence IDs before handing off raw events to the core operator library, keeping tracking of the latest sequence ID processed (a high water mark). As events are emitted, the query evaluator introduces sequence IDs and fills a replay buffer from which downstream nodes can replay. This approach brought composition of query operators to the physical communication layer across compute nodes, thus realizing the dot-pipe equivalence.

> IRP uses the same trick to "shake off" different concerns at the boundary of a query engine. For example, queries arrive at the engine in expression tree form, but end up being compiled and evaluated (thus shaking off the expression tree representation, i.e. the Q in `Qubscription` turns into an S in `Subscription`). Similarly, queries end up at engine boundaries in the `IAsync` space, but once the computation lands on a node for local evaluation, we move to the isomorphic synchronous space (thus shaking off the async nature, i.e. the `Async` in `AsyncSubscription` is dropped to end up with `Subscription`). A mental image is a fat wide pipe turning into a slim narrow pipe at the boundary of a box (the query engine), where aspects like quotation, asynchrony, sequence numbers, etc. smash into the wall of the box, and only the essence flows through into the heart of the box (the operator library).
>
> Philosophically, the whole IRP space is a hypercube in N dimensions with points populated by a family of interfaces for the six "Standard Model" reactive artifacts. Classic Rx is just two points in this hypercube (for `IObservable<T>` versus `IQbservable<T>`, moving along the binary axis of quotation support). Other orthogonal concerns are sync versus async, intrinsic versus

> extrinsic identifiers, reliable or non-reliable, and a few more esoteric ones that were discovered along the way.

Query evaluator design

After deciding on a reliability story, we started prototyping the query evaluator engine. An initial prototype attempt by Bart and Brian was based on Orleans, modeling standing queries and streams (really just subjects) as actors, decomposing execution all the way to individual query operators in order to enable maximum reuse of subcomputations, by representing all such operators as grains. The initial thinking here was to get to "common subexpression elimination" (CSE) by default, but it proved to be prohibitively expensive. In addition, no reliable "virtual streams" capability existed in Orleans at that time, making it hard to map our reliable Rx abstraction onto the underlying execution fabric.

> Conversations between the Reaqtor and Orleans teams at a later point led to the introduction of virtual streams in Orleans, which have interfaces that share some similarities with IRP.

Ultimately, we went back to the drawing board and Bart started an implementation of the core query engine roughly based on the demo for Rx-as-a-service which Bart and Wes built back in the Cloud Programmability Team days, using the IRP abstractions described earlier. The following design principles were put forward from the get go:

- Avoid I/O on the event processing code path for predictible performance and to ensure repeatability of the computation.
- Synchronous execution of core query operators for maximum throughput, with minimal synchronization requirements.
- Minimal dependencies of the query operator library and the hosting environment.
- Tightly controlled threading to avoid excessive context switching.
- Achieve high density of standing queries by optimizing object layouts.
- Minimal overhead for checkpointing of operator state.

Other contributors to the design and implementation of the core query engine include Tiho Tarnavsi (who previously worked on StreamInsight), various folks from an MSR team in Munich, and Eric Rozell. This virtual geodistributed team ironed out the final details of the query engine design and we finished the first version of it in late 2013.

First, we implemented two variants of the engine to evaluate different reliability models. One used an active/active strategy to achieve reliability, while another one used a checkpoint/replay strategy. The Munich team did perform various evaluations to contrast and compare both approaches for various workloads. Not surprisingly, the key trade-offs are between memory usage, replication of events, and recovery time. Ultimately, we decided to abstract out reliability to land our `IQueryEngine` interface,

and proceeded to implement the `CheckpointingQueryEngine` as the most promising route to achieve high density (with working set being the driving factor), leaving the possibility to add other implementations for reliability at a future point. To this day, most Reaqtor deployments are running with this checkpoint/replay engine, but highly available low latency workloads could be hosted in an active/active query engine quite easily.

> In fact, one hosting environment for the Reaqtor engine has been used in a quite interesting corner of the design space, with billions of standing queries with 99.9% of those queries only handling a few events per day. This deployment employs the checkpointing query engine by receiving an event, recovering the query engine(s) containing query expressions for which the event is destined, ingesting the event, and then putting the query engine(s) in a dormant state again by checkpointing and unloading. This is effectively a paging system for micro-computations with an event-based activation mechanism.

Second, we decided on a physical/logical layering of schedulers. Rx has an `IScheduler` interface to adapt to various sources of concurrency and to abstract over the notion of time. In the context of Reaqtor, parameterization of individual query operators on schedulers has little value, because we want to bind the scheduler to the hosting environment. In addition, we wanted to be able to host many query engines within the same process, to achieve density while making the unit of failure decoupled from the physical machine size. Today, production deployments of services built on Reaqtor run tens of query engine instances per process, each of them hosting tens of thousands of standing queries. Each such query engine is mapped 1-to-1 to a reliable stateful service in Service Fabric for fine-grained failover. This led to the introduction of logical scheduler which share a fixed pool of physical threads within the same process. Each query engine has its own logical scheduler which has an independent lifetime and supports pause/resume to facilitate checkpointing. The physical scheduler performs work stealing from its child logical schedulers.

The choice to support pause/resume on logical schedulers was motivated by an evaluation of different checkpointing options. One such option we used in the StreamInsight days is to flow a checkpoint marker through the data flow path. Various prototypes showed that this gets quite tricky given the dynamism provided by the Rx algebra, in particular higher order query operators, and it's hard to provide an upper bound to the time taken to perform a checkpoint. Moreover, detecting whether query operators are dirty, to facilitate differential checkpointing to reduce I/O, gets tricky as well. Based on these findings, we decided on a stop-the-world checkpoint approach, but with various design points and mechanisms in place to reduce the pause time to a minimum:

- Only pausing event processing during the snapshotting of state, thus not including the time to write to permanent storage (I/O bound), by introducing a tri-state flag on query operators.
- Clean separation of immutable data (e.g. definition of standing queries in the form of expression trees and metadata) from mutable data (e.g. operator state).

- Partitioning of state for highly dynamic query expressions, e.g. involving higher-order operators, by decomposing queries into smaller pieces.
- Support for differential checkpoints with a lightweight mechanism to detect dirty operators that need persistence.

The design work on the engine was done hand-in-handy with the hosted operator libraries to ensure a minimal interface between both. Ultimately, our goal was to converge the newly built operator library with the OSS version of Rx, merely adding enlightenments for running this popular library in a hosted service, offering reliability mechanisms and more. This led to the formalization of `ISubscribable<T>` where we added support for a visitor pattern to traverse an operator tree. In classic Rx, the `IDisposable` interface is used to tear down a standing event processing query using some traversal pattern of the nodes. With `ISubscribable<T>`, we support other traversals of the (dynamic) operator tree using an `ISubscriptionVisitor` interface, enabling a variety of uses:

- Initialize a standing query by providing it context such as the host's scheduler, logging facilities, etc.
- Start processing a standing query by sending a signal to the leaf observable nodes in the tree (effectively opening the "tap" of events).
- Traverse the operator tree to detect whether any node has state changes since the last successful checkpoint by checking dirty flags.
- Perform a checkpoint of operator state by providing a state writer to each node in a well-defined traversal order.
- Conversely, restore operator state by providing a state reader to each node in a well-defined traversal order.
- Graceful unloading of standing queries to allow releasing of resources, akin to `IDisposable`.
- Termination of a standing query, for example by decrementing reference count on sources and sinks.

Compared to classic Rx, the `ISubscribable<T>` interface adds more granular phasing to the lifecycle of standing queries, by returning an `ISubscription` which gives access to the (dynamically evolving) query operator tree to perform a variety of operations. The mere use of `Subscribe` to associate a subscriber (in the form of an `IObserver<T>`) with a source of events does not kick off the computation but rather acts as the factory of an initial operator tree, with subsequent operations taking care of providing additional context, restoring state from a previous checkpoint, starting the event flow, etc. This pattern is also useful in client applications, for example to support tombstoning of an application by persisting its state to disk, or to distribute the same scheduler to all operator nodes rather than manually parameterizing it everywhere.

Other aspects of the query engine include its implementation of IRP interface to accept expression trees representing event processing queries and to query the artifacts hosted within an engine (the

so-called “registry” or “catalog”), and the parameterization on resolvers which enable dynamic lookup of bound artifacts, allowing for cross-engine communication to scale out computation across nodes.

Halfway through the initial implementation of the query engine, Eric Rozell joined the core framework team, working on a wide range of implementation aspects including expression tree serialization, the data model to serialize event objects, expression tree binders, etc. Other contributors to the core engine include Alex Clemmer (who worked on higher-order query operator support) and Pranav Senthilnathan (who worked on transaction logging and various forms of query optimization).

At-least-once guarantees and repeatability

Reaqtor’s reliability model offers at-least-once event processing guarantees with a soft upper bound to possible repetition of processing due to replay of ingress events in case of a compute node failure (time-bound to a checkpointing interval).

The basic idea is process events without incurring any I/O on the event processing code path to persist changes to computational state, thus reducing the latency of the system. Periodic **checkpoints** are used to summarize all the state changes and reliably persist these to permanent storage, allowing for recovery in case of a node failure. These checkpoints include the (immutable) definitions of all reactive artifacts as well as the runtime state of all query operators in subscriptions (e.g. the running sum and count for an average aggregation operator).

Checkpoints are implemented using a pause-the-world scheduler mechanism, allowing to take a consistent snapshot of the state of all reactive artifacts while no events are flowing. After an in-memory snapshot is taken (which typically takes a few milliseconds), the flow of events is restored, and the state is persisted to a replicated state store. When the state is successfully persisted (which may take many seconds due to replication), artifacts get marked as successfully saved. Note that these artifacts may have been dirtied again in the meantime due to the resumed computation.

This checkpointing process repeats periodically at fixed intervals (though it could be augmented to support automatic checkpointing based on an estimation of the amount of dirty state and a configurable upper limit to checkpoint intervals). The query engine supports full and differential checkpoints, thus reducing the data volume written each checkpoint to only the state change deltas.

> Alternative implementations have been prototyped, where an engine checkpoints itself when there’s sufficient dirty state, while avoiding excessively long replay sequences for events. This is similar to a garbage collector deciding if/when to perform memory management operations. In fact, other techniques from GC (such as aging artifacts by assigning them to generations, or keeping track of nodes with high frequency edits, versus rarely changing ones to partition the state space) have been experimented with as well. This could be an area of future innovcation.

As part of these checkpoints, all artifacts that receive events from outside the containing engine (e.g. observables or subjects) persist sequence identifiers that allow them to replay events in case of a failover. After successfully persisting a checkpoint, these artifacts can prune or trim external replay history, if they wish to do so. It is this replay of events in case of failover that leads to the at-least-once processing guarantee provided by Reaqtor. In case query operators have repeatable or convergent behavior, de-duplication on the egress side can be used to filter out duplicate events that resulted from replay during a failover.

Note that the checkpointing interval dictates a retention size requirement, e.g. for checkpoints every minute on a stream producing 1,000 events/second, one needs at least retention of the last 60,000 events (not accounting for failed or missed checkpoints; in reality the retention size is a multiple of this back of the napkin calculation).

In many of the Reaqtor production deployments, we use Service Fabric with its key/value store support as the mechanism to replicate state to secondary replicas of query evaluator nodes. More recent deployments have migrated to use reliable collections instead. A typical configuration runs the query evaluators as stateful services with 5 replicas, thus each commit of a checkpoint results in a quorum of writes on at least 3 replicas. A key advantage of Service Fabric is the colocation of an engine's state store with the pimrary replica, which enables for fast recovery which involves a lot of small reads. Using a central store, recovery times would go up, but alternative approaches could be used to achieve these goals (e.g. Redis), as long as transactional consistency for checkpoints is guaranteed.

> Early on, we also investigated the use of Replicated State Library (RSL), but after consulting with key services in Azure, we decided to bet on Service Fabric as well.

To be continued...

Reaqtor will be released publicly on May 18th 2021 at Techorama. If you would like to discuss Reaqtor, drop by the Reaqtive Slack workspace.