Table of Contents

- Overview
  - What is Service Fabric?
  - Understand microservices
  - Application scenarios
  - Architecture
- Get Started
  - Set up your development environment
    - Windows
    - Linux
    - Mac OS
  - Create your first application
    - C# on Windows
    - Java on Linux
    - C# on Linux
  - Deploy apps on a local cluster
- How To
  - Build an application
    - Basics
    - Reliable Service application
    - Reliable Actor application
    - Guest executable application
    - Container application
  - Migrate from Cloud Services
    - Compare Cloud Services with Service Fabric
    - Migrate to Service Fabric
  - Create and manage clusters
    - Basics
    - Clusters on Azure
    - Standalone clusters

Manage and orchestrate cluster resources
- Cluster Resource Manager overview
- Cluster Resource Manager architecture
- Describe a cluster
- Application groups overview
- Configure Cluster Resource Manager settings
- Resource consumption metrics
- Use service affinity
- Service placement policies
- Manage a cluster
- Cluster defragmentation
- Balance a cluster
- Throttling
- Service movement

Manage application lifecycle
- Overview
- Set up continuous integration
- Deploy or remove applications
- Application upgrade overview
- Configure application upgrade
- Application upgrade parameters
- Upgrade an application
- Troubleshoot application upgrades
- Data serialization in application upgrades
- Application upgrades advanced topics
- REST-based application lifecycle sample

Inspect application and cluster health
- Monitor Service Fabric health
- Report and check service health
- Add custom health reports
- Troubleshoot with system health reports
- View health reports

- Monitor and diagnose
  - Monitor and diagnose services locally
  - Azure Diagnostics logs
  - Service Fabric application trace
  - Diagnostics in Reliable Actors
  - Diagnostics in stateful Reliable Services
  - Troubleshoot your local cluster
  - Troubleshoot common issues
- Scale applications
  - Partition Reliable Services
  - Availability of services
  - Scale applications
  - Plan capacity of applications
- Test applications and services
  - Fault Analysis overview
  - Test service-to-service communication
  - Simulate failures
  - Load test your application
Reference
- Terminology
- Reliable Actors
- Reliable Actors WCF
- Reliable Services
- Reliable Services WCF
- Data
- Data Interfaces
- System
- Sample code
- PowerShell
- Java API
- .NET
- REST

Resources

- Learning path
- Pricing
- Service Updates
- MSDN Forum
- Videos

Overview of Azure Service Fabric

11/15/2016 • 6 min to read • Edit on GitHub

Contributors

Mark Fussell • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Junshen Tao (Beyondsoft Corporation) • ShawnJackson • Joseph Molnar • MikeWedderburn-Clarke • Ryan Wike • Adriano Arce • MarkDeichmiller • Jon Langdon • Andy De George • Dene Hager • clca

Azure Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices. Service Fabric also addresses the significant challenges in developing and managing cloud applications. Developers and administrators can avoid complex infrastructure problems and focus on implementing mission-critical, demanding workloads that are scalable, reliable, and manageable. Service Fabric represents the next-generation middleware platform for building and managing these enterprise-class, tier-1, cloud-scale applications.

This short video introduces Service Fabric and microservices.

Applications composed of microservices

Service Fabric enables you to build and manage scalable and reliable applications composed of microservices that run at very high density on a shared pool of machines, which is referred to as a cluster. It provides a sophisticated runtime to build distributed, scalable, stateless and stateful microservices. It also provides comprehensive application management capabilities to provision, deploy, monitor, upgrade/patch, and delete deployed applications.

Why is a microservices approach important? The two main reasons are:

- You can scale different parts of your application depending on the needs of the application.
- Development teams can be more agile as they roll out changes and thus provide features to customers faster and more frequently.

Service Fabric powers many Microsoft services today, including Azure SQL Database, Azure DocumentDB, Cortana, Microsoft Power BI, Microsoft Intune, Azure Event Hubs, Azure IoT Hub, Skype for Business, and many core Azure services.

Service Fabric is tailored to create cloud native services that can start small, as needed, and grow to massive scale with hundreds or thousands of machines.

Today's Internet-scale services are built of microservices. Examples of microservices include protocol gateways, user profiles, shopping carts, inventory processing, queues, and caches. Service Fabric is a microservices platform that gives every microservice a unique name that can be either stateless or stateful.

Service Fabric provides comprehensive runtime and lifecycle management capabilities to applications that are composed of these microservices. It hosts microservices inside containers that are deployed and activated across the Service Fabric cluster. A move from virtual machines to containers makes possible an order-of-magnitude increase in density. Similarly, another order of magnitude in density becomes possible when you move from containers to microservices. For example, a single cluster for Azure SQL Database comprises hundreds of machines running tens of thousands of containers that host a total of hundreds of thousands of databases. Each database is a Service Fabric stateful microservice. The same is true of the other previously mentioned services, which is why the term *hyperscale* is used to describe Service Fabric capabilities. If containers give you high density, then microservices give you hyperscale.

For more on the microservices approach, read Why a microservices approach to building applications?.

Container deployment and orchestration

Service Fabric is an orchestrator of microservices across a cluster of machines. Microservices can be developed in many ways from using the Service Fabric programming models to deploying guest executables. Service Fabric can deploy services in container images. Importantly, you can mix both services in processes and services in containers in the same application. If you just want to deploy and manage container images across a cluster of machines, Service Fabric is a perfect choice.

Create clusters for Service Fabric anywhere

You can create clusters for Service Fabric in many environments, including Azure or on premises, on Windows Server, or on Linux. In addition, the development environment in the SDK is identical to the production environment, and no emulators are involved. In other words, what runs on your local development cluster deploys to the same cluster in other environments.

For more information on creating clusters on-premises, read creating a cluster on Windows Server or Linux or for Azure creating a cluster via the Azure portal.

Stateless and stateful micrososervices for Service Fabric

Service Fabric enables you to build applications that consist of microservices. Stateless microservices (such as protocol gateways and web proxies) do not maintain a mutable state outside a request and its response from the service. Azure Cloud Services worker roles are an example of a stateless service. Stateful microservices (such as user accounts, databases, devices, shopping carts, and queues) maintain a mutable, authoritative state beyond the request and its response. Today's Internet-scale applications consist of a combination of stateless and stateful microservices.

Why have stateful microservices along with stateless ones? The two main reasons are:

- You can build high-throughput, low-latency, failure-tolerant online transaction processing (OLTP) services by keeping code and data close on the same machine. Some examples are interactive storefronts, search, Internet of Things (IoT) systems, trading systems, credit card processing and fraud detection systems, and personal record management.
- You can simplify application design. Stateful microservices remove the need for additional queues and caches, which are traditionally required to address the availability and latency requirements of a purely stateless application. Stateful services are naturally high-availability and low-latency, which reduces the number of moving parts to manage in your application as a whole.

For more information about application patterns with Service Fabric, read Application scenarios and Choosing a programming model framework for your service.

Application lifecycle management

Service Fabric provides support for the full application lifecycle management of cloud applications. This lifecycle includes development through deployment, daily management, and maintenance to eventual decommissioning.

Service Fabric application lifecycle management capabilities enable application administrators and IT operators to use simple, low-touch workflows to provision, deploy, patch, and monitor applications. These built-in workflows greatly reduce the burden on IT operators to keep applications continuously available.

Most applications consist of a combination of stateless and stateful microservices and other executables/runtimes that are deployed together. By having strong types on the applications and packaged microservices, Service Fabric enables the deployment of multiple application instances. Each instance is managed and upgraded independently. Importantly, Service Fabric can deploy *any* executables or runtime and make them reliable. For example, Service Fabric deploys ASP.NET Core 1, Node.js, Java virtual machines, scripts, or anything else that makes up your application.

For more information about application lifecycle management, read Application lifecycle. For more about how to deploy any code, see Deploy a guest executable.

Key capabilities

By using Service Fabric, you can:

- Develop massively scalable applications that are self-healing.
- Develop applications that are composed of microservices by using the Service Fabric programming model. Or, you can simply host guest executables and other application frameworks of your choice, such as ASP.NET Core 1 or Node.js.
- Develop highly reliable stateless and stateful microservices.
- Deploy and orchestrate containers that include Windows containers and Docker containers across a cluster. These containers can contain guest executables or reliable stateless and stateful microservices. In either case, you get mapping from container port to host port, container discoverability, and automated failover.
- Simplify the design of your application by using stateful microservices in place of caches and queues.
- Deploy to Azure or to on-premises datacenters that run Windows or Linux with zero code changes. Write once, and then deploy anywhere to any Service Fabric cluster.
- Develop with a "datacenter on your machine" approach. The local development environment is the same code that runs in the Azure datacenters.
- Deploy applications in seconds.
- Deploy applications at higher density than virtual machines, deploying hundreds or thousands of applications per machine.
- Deploy different versions of the same application side by side, and upgrade each application independently.
- Manage the lifecycle of your stateful applications without any downtime, including breaking and nonbreaking upgrades.
- Manage applications by using .NET APIs, Java (Linux), PowerShell, Azure command-line interface (Linux), or REST interface.
- Upgrade and patch microservices within applications independently.
- Monitor and diagnose the health of your applications and set policies for performing automatic repairs.
- Scale out or scale in the number of nodes in a cluster, and scale up or scale down the size of each node. As you scale nodes, your applications automatically scale and are distributed according to the available resources.
- Watch the self-healing resource balancer orchestrate the redistribution of applications across the cluster. Service

Fabric recovers from failures and optimizes the distribution of load based on available resources.

Next steps

- For more information:
  - Why a microservices approach to building applications?
  - Terminology overview
- Setting up your Service Fabric development environment
- Choosing a programming model framework for your service

Why a microservices approach to building applications?

11/15/2016 • 14 min to read • Edit on GitHub

Contributors

Mark Fussell • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • ShawnJackson • Joseph Molnar • Dhanashri Kshirsagar • katiecumming • Alexander Zeitler • Ryan Wike • Adriano Arce • MikeBazMSFT • MarkDeichmiller • Andy De George

As software developers, there is nothing new in how we think about factoring an application into component parts. It is the central paradigm of object orientation, software abstractions, and componentization. Today, this factorization tends to take the form of classes and interfaces between shared libraries and technology layers. Typically, a tiered approach is taken with a back-end store, middle-tier business logic, and a front-end user interface (UI). What *has* changed over the last few years is that we, as developers, are building distributed applications that are for the cloud and driven by the business.

The changing business needs are:

- A service that's built and operates at scale to reach customers in new geographical regions (for example).
- Faster delivery of features and capabilities to be able to respond to customer demands in an agile way.
- Improved resource utilization to reduce costs.

These business needs are affecting *how* we build applications.

For more information about the approach of Azure to microservices, read Microservices: An application revolution powered by the cloud.

Monolithic vs. microservice design approach

All applications evolve over time. Successful applications evolve by being useful to people. Unsuccessful applications do not evolve and eventually are deprecated. The question becomes: How much do you know about your requirements today, and what will they be in the future? For example, let's say that you are building a reporting application for a department. You are sure that the application will remain within the scope of your company and that the reports will be short-lived. Your choice of approach is different from, say, building a service that delivers video content to tens of millions of customers.

Sometimes, getting something out the door as proof of concept is the driving factor, while you know that the application can be redesigned later. There is little point in over-engineering something that never gets used. It’s the usual engineering trade-off. On the other hand, when companies talk about building for the cloud, the expectation is growth and usage. The issue is that growth and scale are unpredictable. We would like to be able to prototype quickly while also knowing that we are on a path to deal with future success. This is the lean startup approach: build, measure, learn, and iterate.

During the client-server era, we tended to focus on building tiered applications by using specific technologies in each tier. The term *monolithic* application has emerged for these approaches. The interfaces tended to be between the tiers, and a more tightly coupled design was used between components within each tier. Developers designed and factored classes that were compiled into libraries and linked together into a few executables and DLLs.

There are benefits to such a monolithic design approach. It's often simpler to design, and it has faster calls between components, because these calls are often over interprocess communication (IPC). Also, everyone tests a single product, which tends to be more people-resource efficient. The downside is that there's a tight coupling

between tiered layers, and you cannot scale individual components. If you need to perform fixes or upgrades, you have to wait for others to finish their testing. It is more difficult to be agile.

Microservices address these downsides and more closely align with the preceding business requirements, but they also have both benefits and liabilities. The benefits of microservices are that each one typically encapsulates simpler business functionality, which you scale up or down, test, deploy, and manage independently. One important benefit of a microservice approach is that teams are driven more by business scenarios than by technology, which the tiered approach encourages. In practice, smaller teams develop a microservice based on a customer scenario and use any technologies they choose.

In other words, the organization doesn't need to standardize tech to maintain monolithic applications. Individual teams that own services can do what makes sense for them based on team expertise or what's most appropriate to solve the problem. In practice, a set of recommended technologies, such as a particular NoSQL store or web application framework, is preferable.

The downside of microservices comes in managing the increased number of separate entities and dealing with more complex deployments and versioning. Network traffic between the microservices increases as well as the corresponding network latencies. Lots of chatty, granular services are a recipe for a performance nightmare. Without tools to help view these dependencies, it is hard to "see" the whole system.

Standards make the microservice approach work by agreeing on how to communicate and being tolerant of only the things you need from a service, rather than rigid contracts. It is important to define these contracts up front in the design, because services update independently of each other. Another description coined for designing with a microservices approach is "fine-grained service-oriented architecture (SOA)."

At its simplest, the microservices design approach is about a decoupled federation of services, with independent changes to each, and agreed-upon standards for communication.

As more cloud apps are produced, people discover that this decomposition of the overall app into independent, scenario-focused services is a better long-term approach.

Comparison between application development approaches

- A monolithic app contains domain-specific functionality and is normally divided by functional layers, such as web, business, and data.
- You scale a monolithic app by cloning it on multiple servers/virtual machines/containers.

- A microservice application separates functionality into separate smaller services.
- The microservices approach scales out by deploying each service independently, creating instances of these services across servers/virtual machines/containers.

Designing with a microservice approach is not a panacea for all projects, but it does align more closely with the business objectives described earlier. Starting with a monolithic approach might be acceptable if you know that you will not have the opportunity to rework the code later into a microservice design if necessary. More commonly, you begin with a monolithic app and slowly break it up in stages, starting with the functional areas that need to be more scalable or agile.

To summarize, the microservice approach is to compose your application of many small services. The services run in containers that are deployed across a cluster of machines. Smaller teams develop a service that focuses on a scenario and independently test, version, deploy, and scale each service so that the entire application can evolve.

What is a microservice?

There are different definitions of microservices. If you search the Internet, you'll find many useful resources that provide their own viewpoints and definitions. However, most of the following characteristics of microservices are widely agreed upon:

- Encapsulate a customer or business scenario. What is the problem you are solving?
- Developed by a small engineering team.
- Written in any programming language and use any framework.
- Consist of code and (optionally) state, both of which are independently versioned, deployed, and scaled.
- Interact with other microservices over well-defined interfaces and protocols.
- Have unique names (URLs) used to resolve their location.
- Remain consistent and available in the presence of failures.

You can summarize these characteristics into:

Microservice applications are composed of small, independently versioned, and scalable customer-focused services that communicate with each other over standard protocols with well-defined interfaces.

We covered the first two points in the preceding section, and now we expand on and clarify the others.

Written in any programming language and use any framework

As developers, we should be free to choose a language or framework that we want, depending on our skills or the needs of the service. In some services, you might value the performance benefits of C++ above all else. In other services, the ease of managed development in C# or Java might be most important. In some cases, you may need to use a specific partner library, data storage technology, or means of exposing the service to clients.

After you have chosen a technology, you come to the operational or lifecycle management and scaling of the service.

Allows code and state to be independently versioned, deployed, and scaled

However you choose to write your microservices, the code and optionally the state should independently deploy, upgrade, and scale. This is actually one of the harder problems to solve, because it comes down to your choice of technologies. For scaling, understanding how to partition (or shard) both the code and state is challenging. When the code and state use separate technologies, which is common today, the deployment scripts for your microservice need to be able to cope with scaling them both. This is also about agility and flexibility, so you can upgrade some of the microservices without having to upgrade all of them at once.

Returning to the monolithic versus microservice approach for a moment, the following diagram shows the differences in the approach to storing state.

State storage between application styles

The monolithic approach on the left has a single database and tiers of specific technologies.

The microservices approach on the right has a graph of interconnected microservices where state is typically scoped to the microservice and various technologies are used.

In a monolithic approach, typically the application uses a single database. The advantage is that it is a single location, which makes it easy to deploy. Each component can have a single table to store its state. Teams need to strictly separate state, which is a challenge. Inevitably there are temptations to add a new column to an existing customer table, do a join between tables, and create dependencies at the storage layer. After this happens, you can't scale individual components.

In the microservices approach, each service manages and stores its own state. Each service is responsible for scaling both code and state together to meet the demands of the service. A downside is that when there is a need to create views, or queries, of your application's data, you will need to query across disparate state stores. Typically, this is solved by having a separate microservice that builds a view across a collection of microservices. If you need to perform multiple impromptu queries on the data, each microservice should consider writing its data to a data warehousing service for offline analytics.

Versioning is specific to the deployed version of a microservice so that multiple, different versions deploy and run side by side. Versioning addresses the scenarios where a newer version of a microservice fails during upgrade and needs to roll back to an earlier version. The other scenario for versioning is performing A/B-style testing, where different users experience different versions of the service. For example, it is common to upgrade a microservice for a specific set of customers to test new functionality before rolling it out more widely. After lifecycle management of microservices, this now brings us to communication between them.

Interacts with other microservices over well-defined interfaces and protocols

This topic needs little attention here, because extensive literature about service-oriented architecture that has been published over the past 10 years describes communication patterns. Generally, service communication uses a REST approach with HTTP and TCP protocols and XML or JSON as the serialization format. From an interface perspective, it is about embracing the web design approach. But nothing stops you from using binary protocols or your own data formats. Be prepared for people to have a harder time using your microservices if these are openly available.

Has a unique name (URL) used to resolve its location

Remember how we keep saying that the microservice approach is like the web? Like the web, your microservice needs to be addressable wherever it is running. If you are thinking about machines and which one is running a particular microservice, things will go bad quickly.

In the same way that DNS resolves a particular URL to a particular machine, your microservice needs to have a unique name so that its current location is discoverable. Microservices need addressable names that make them independent from the infrastructure that they are running on. This implies that there is an interaction between how your service is deployed and how it is discovered, because there needs to be a service registry. Equally, when a machine fails, the registry service must tell you where the service is now running.

This brings us to the next topic: resilience and consistency.

Remains consistent and available in the presence of failures

Dealing with unexpected failures is one of the hardest problems to solve, especially in a distributed system. Much of the code that we write as developers is handling exceptions, and this is also where the most time is spent in testing. The problem is more involved than writing code to handle failures. What happens when the machine where the microservice is running fails? Not only do you need to detect this microservice failure (a hard problem on its own), but you also need something to restart your microservice.

A microservice needs to be resilient to failures and restart often on another machine for availability reasons. This also comes down to the state that was saved on behalf of the microservice, where the microservice can recover this state from, and whether the microservice is able to restart successfully. In other words, there needs to be resilience in the compute (the process restarts) as well as resilience in the state or data (no data loss and the data remains consistent).

The problems of resiliency are compounded during other scenarios, such as when failures happen during an application upgrade. The microservice, working with the deployment system, doesn't need to recover. It also needs to then decide whether it can continue to move forward to the newer version or instead roll back to a previous version to maintain a consistent state. Questions such as whether enough machines are available to keep moving forward and how to recover previous versions of the microservice need to be considered. This requires the microservice to emit health information to be able to make these decisions.

Reports health and diagnostics

It may seem obvious, and it is often overlooked, but a microservice must report its health and diagnostics. Otherwise, there is little insight from an operations perspective. Correlating diagnostic events across a set of independent services and dealing with machine clock skews to make sense of the event order is challenging. In the same way that you interact with a microservice over agreed-upon protocols and data formats, there emerges a need for standardization in how to log health and diagnostic events that ultimately end up in an event store for querying and viewing. In a microservices approach, it is key that different teams agree on a single logging format. There needs to be a consistent approach to viewing diagnostic events in the application as a whole.

Health is different from diagnostics. Health is about the microservice reporting its current state to take appropriate actions. A good example is working with upgrade and deployment mechanisms to maintain availability. Although a service may be currently unhealthy due to a process crash or machine reboot, the service might still be operational. The last thing you need is to make this worse by performing an upgrade. The best approach is to do an investigation first or allow time for the microservice to recover. Health events from a microservice help us make informed decisions and, in effect, help create self-healing services.

Service Fabric as a microservices platform

Azure Service Fabric emerged from a transition by Microsoft from delivering box products, which were typically monolithic in style, to delivering services. The experience of building and operating large services, such as Azure SQL Database and Azure DocumentDB, shaped Service Fabric. The platform evolved over time as more and more services adopted it. Importantly, Service Fabric had to run not only in Azure but also in standalone Windows Server deployments.

The aim of Service Fabric is to solve the hard problems of building and running a service and utilize infrastructure resources efficiently, so that teams can solve business problems using a microservices approach.

Service Fabric provides two broad areas to help you build applications that use a microservices approach:

- A platform that provides system services to deploy, upgrade, detect, and restart failed services, discover service location, manage state, and monitor health. These system services in effect enable many of the characteristics of microservices previously described.
- Programming APIs, or frameworks, to help you build applications as microservices: reliable actors and reliable services. Of course, you can choose any code to build your microservice. But these APIs make the job more straightforward, and they integrate with the platform at a deeper level. This way, for example, you can get health and diagnostics information, or you can take advantage of built-in high availability.

Service Fabric is agnostic on how you build your service, and you can use any technology.However, it does provide built-in programming APIs that make it easier to build microservices.

Are microservices right for my application?

Maybe. What we experienced was that as more and more teams in Microsoft began to build for the cloud for business reasons, many of them realized the benefits of taking a microservice-like approach. Bing, for example, has been developing microservices in search for years. For other teams, the microservices approach was new. Teams found that there were hard problems to solve outside of their core areas of strength. This is why Service Fabric gained traction as the technology of choice for building services.

The objective of Service Fabric is to reduce the complexities of building applications with a microservice approach, so that you do not have to go through as many costly redesigns. Start small, scale when needed, deprecate services, add new ones, and evolve with customer usage is the approach. We also know that there are many other problems yet to be solved to make microservices more approachable for most developers. Containers and the actor programming model are examples of small steps in that direction, and we are sure that more innovations will emerge to make this easier.

Next steps

- Service Fabric terminology overview
- Microservices: An application revolution powered by the cloud

Service Fabric application scenarios

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Mark Fussell • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • alexandair • Adriano Arce • MarkDeichmiller • Andy Pasic • Cesar Mello • Jesse Benson • Matt

Azure Service Fabric offers a reliable and flexible platform that enables you to write and run many types of business applications and services. These applications and microservices can be stateless or stateful, and they are resource-balanced across virtual machines to maximize efficiency. The unique architecture of Service Fabric enables you to perform near real-time data analysis, in-memory computation, parallel transactions, and event processing in your applications. You can easily scale your applications up or down (really in or out), depending on your changing resource requirements.

The Service Fabric platform in Azure is ideal for the following categories of applications and services:

- **Highly available services**: Service Fabric services provide fast failover by creating multiple secondary service replicas. If a node, process, or individual service goes down due to hardware or other failure, one of the secondary replicas is promoted to a primary replica with minimal loss of service.
- **Scalable services**: Individual services can be partitioned, allowing for state to be scaled out across the cluster. In addition, individual services can be created and removed on the fly. Services can be quickly and easily scaled out from a few instances on a few nodes to thousands of instances on many nodes, and then scaled in again, depending on your resource needs. You can use Service Fabric to build these services and manage their complete lifecycles.
- **Computation on nonstatic data**: Service Fabric enables you to build data, input/output, and compute intensive stateful applications. Service Fabric allows the collocation of processing (computation) and data in applications. Normally, when your application requires access to data, there is network latency associated with an external data cache or storage tier. With stateful Service Fabric services, that latency is eliminated, enabling more performant reads and writes. Say for example that you have an application that performs near real-time recommendation selections for customers, with a round-trip time requirement of less than 100 milliseconds. The latency and performance characteristics of Service Fabric services (where the computation of recommendation selection is collocated with the data and rules) provides a responsive experience to the user compared with the standard implementation model of having to fetch the necessary data from remote storage.
- **Session-based interactive applications**: Service Fabric is useful if your applications, such as online gaming or instant messaging, require low latency reads and writes. Service Fabric enables you to build these interactive, stateful applications without having to create a separate store or cache, as required for stateless apps. (This increases latency and potentially introduces consistency issues.)
- **Distributed graph processing**: The growth of social networks has greatly increased the need to analyze large-scale graphs in parallel. Fast scaling and parallel load processing make Service Fabric a natural platform for processing large-scale graphs. Service Fabric enables you to build highly scalable services for groups such as social networking, business intelligence, and scientific research.
- **Data analytics and workflows**: The fast reads and writes of Service Fabric enable applications that must reliably process events or streams of data. Service Fabric also enables applications that describe processing pipelines, where results must be reliable and passed on to the next processing stage without loss. These include transactional and financial systems, where data consistency and computation guarantees are essential.

Application design case studies

A number of case studies showing how Service Fabric is used to design applications are published on the microservices solutions site

Design applications composed of stateless and stateful microservices

Building applications with Azure cloud service worker roles is an example of a stateless service. In contrast, stateful microservices maintain their authoritative state beyond the request and its response. This provides high availability and consistency of the state through simple APIs that provide transactional guarantees backed by replication. Service Fabric's stateful services democratize high availability, bringing it to all types of applications, not just databases and other data stores. This is a natural progression. Applications have already moved from using purely relational databases for high availability to NoSQL databases. Now the applications themselves can have their "hot" state and data managed within them for additional performance gains without sacrificing reliability, consistency, or availability.

When building applications consisting of microservices, you typically have a combination of stateless web apps (ASP.NET, Node.js, etc.) calling onto stateless and stateful business middle-tier services, all deployed into the same Service Fabric cluster using the Service Fabric deployment commands. Each of these services is independent with regard to scale, reliability, and resource usage, greatly improving agility in development and lifecycle management.

Stateful microservices simplify application designs because they remove the need for the additional queues and caches that have traditionally been required to address the availability and latency requirements of purely stateless applications. Since stateful services are naturally highly available and low latency, this means that there are fewer moving parts to manage in your application as a whole. The diagrams below illustrate the differences between designing an application that is stateless and one that is stateful. By taking advantage of the reliable services and reliable actors programming models, stateful services reduce application complexity while achieving high throughput and low latency.

An application built using stateless services

- Scale with partitioned storage
- Increase reliability with queues
- Reduce read latency with caches
- Write your own lock managers for state consistency
- Many moving parts each managed differently

An application built using stateful services

- Application state lives in the compute tier
- Low Latency reads and writes
- Partitions are first class for scale-out
- Built in lock managers based on primary election
- Fewer moving parts

Next steps

- Get started building stateless and stateful services with the Service Fabric reliable services and reliable actors programming models.
- Also see the following topics:
  - Tell me about microservices
  - Define and manage service state
  - Availability of Service Fabric services
  - Scale Service Fabric services
  - Partition Service Fabric services

Service Fabric architecture

11/15/2016 • 5 min to read • Edit on GitHub

Contributors

rishirsinha • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Sumit Gouthaman • PRmerger • alexandair • Adriano Arce • Mark Fussell • stevehug

---

Service Fabric is built with layered subsystems. These subsystems enable you to write applications that are:

- Highly available
- Scalable
- Manageable
- Testable

The following diagram shows the major subsystems of Service Fabric.

In a distributed system, the ability to securely communicate between nodes in a cluster is crucial. At the base of the stack is the transport subsystem, which provides secure communication between nodes. Above the transport subsystem lies the federation subsystem, which clusters the different nodes into a single entity (named clusters) so that Service Fabric can detect failures, perform leader election, and provide consistent routing. The reliability subsystem, layered on top of the federation subsystem, is responsible for the reliability of Service Fabric services through mechanisms such as replication, resource management, and failover. The federation subsystem also underlies the hosting and activation subsystem, which manages the lifecycle of an application on a single node. The management subsystem manages the lifecycle of applications and services. The testability subsystem helps application developers test their services through simulated faults before and after deploying applications and services to production environments. Service Fabric provides the ability to resolve service locations through its communication subsystem. The application programming models exposed to developers are layered on top of these subsystems along with the application model to enable tooling.

Transport subsystem

The transport subsystem implements a point-to-point datagram communication channel. This channel is used for communication within service fabric clusters and communication between the service fabric cluster and clients. It supports one-way and request-reply communication patterns, which provides the basis for implementing broadcast and multicast in the Federation layer. The transport subsystem secures communication by using X509 certificates or Windows security. This subsystem is used internally by Service Fabric and is not directly accessible to developers for application programming.

Federation subsystem

In order to reason about a set of nodes in a distributed system, you need to have a consistent view of the system. The federation subsystem uses the communication primitives provided by the transport subsystem and stitches the various nodes into a single unified cluster that it can reason about. It provides the distributed systems primitives needed by the other subsystems - failure detection, leader election, and consistent routing. The federation subsystem is built on top of distributed hash tables with a 128-bit token space. The subsystem creates a ring topology over the nodes, with each node in the ring being allocated a subset of the token space for ownership. For failure detection, the layer uses a leasing mechanism based on heart beating and arbitration. The federation subsystem also guarantees through intricate join and departure protocols that only a single owner of a token exists at any time. This provides leader election and consistent routing guarantees.

Reliability subsystem

The reliability subsystem provides the mechanism to make the state of a Service Fabric service highly available through the use of the *Replicator*, *Failover Manager*, and *Resource Balancer*.

- The Replicator ensures that state changes in the primary service replica will automatically be replicated to secondary replicas, maintaining consistency between the primary and secondary replicas in a service replica set. The replicator is responsible for quorum management among the replicas in the replica set. It interacts with the failover unit to get the list of operations to replicate, and the reconfiguration agent provides it with the configuration of the replica set. That configuration indicates which replicas the operations need to be replicated. Service Fabric provides a default replicator called Fabric Replicator, which can be used by the programming model API to make the service state highly available and reliable.
- The Failover Manager ensures that when nodes are added to or removed from the cluster, the load is automatically redistributed across the available nodes. If a node in the cluster fails, the cluster will automatically reconfigure the service replicas to maintain availability.
- The Resource Manager places service replicas across failure domain in the cluster and ensures that all failover units are operational. The Resource Manager also balances service resources across the underlying shared pool of cluster nodes to achieve optimal uniform load distribution.

Management subsystem

The management subsystem provides end-to-end service and application lifecycle management. PowerShell cmdlets and administrative APIs enable you to provision, deploy, patch, upgrade, and de-provision applications without loss of availability. The management subsystem performs this through the following services.

- **Cluster Manager**: This is the primary service that interacts with the Failover Manager from reliability to place the applications on the nodes based on the service placement constraints. The Resource Manager in failover subsystem ensures that the constraints are never broken. The cluster manager manages the lifecycle of the applications from provision to de-provision. It integrates with the health manager to ensure that application availability is not lost from a semantic health perspective during upgrades.
- **Health Manager**: This service enables health monitoring of applications, services, and cluster entities. Cluster entities (such as nodes, service partitions, and replicas) can report health information, which is then aggregated

into the centralized health store. This health information provides an overall point-in-time health snapshot of the services and nodes distributed across multiple nodes in the cluster, enabling you to take any needed corrective actions. Health query APIs enable you to query the health events reported to the health subsystem. The health query APIs return the raw health data stored in the health store or the aggregated, interpreted health data for a specific cluster entity.

- **Image Store**: This service provides storage and distribution of the application binaries. This service provides a simple distributed file store where the applications are uploaded to and downloaded from.

Hosting subsystem

The cluster manager informs the hosting subsystem (running on each node) which services it needs to manage for a particular node. The hosting subsystem then manages the lifecycle of the application on that node. It interacts with the reliability and health components to ensure that the replicas are properly placed and are healthy.

Communication subsystem

This subsystem provides reliable messaging within the cluster and service discovery through the Naming service. The Naming service resolves service names to a location in the cluster and enables users to manage service names and properties. Using the Naming service, clients can securely communicate with any node in the cluster to resolve a service name and retrieve service metadata. Using a simple Naming client API, users of Service Fabric can develop services and clients capable of resolving the current network location despite node dynamism or the re-sizing of the cluster.

Testability subsystem

Testability is a suite of tools specifically designed for testing services built on Service Fabric. The tools let a developer easily induce meaningful faults and run test scenarios to exercise and validate the numerous states and transitions that a service will experience throughout its lifetime, all in a controlled and safe manner. Testability also provides a mechanism to run longer tests that can iterate through various possible failures without losing availability. This provides you with a test-in-production environment.

Prepare your development environment

11/15/2016 • 1 min to read • Edit on GitHub

Contributors

unknown • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Sean McKenna • Mikkel Mork Hegnhoj • katiecumming • Andy De George • coreysa • Matt LaBelle

To build and run Azure Service Fabric applications on your development machine, install the runtime, SDK, and tools. You also need to enable execution of the Windows PowerShell scripts included in the SDK.

Prerequisites

Supported operating system versions

The following operating system versions are supported for development:

- Windows 7
- Windows 8/Windows 8.1
- Windows Server 2012 R2
- Windows 10

> **NOTE**
> Windows 7 only includes Windows PowerShell 2.0 by default. Service Fabric PowerShell cmdlets requires PowerShell 3.0 or higher. You can download Windows PowerShell 5.0 from the Microsoft Download Center.

Install the runtime, SDK, and tools

The Web Platform Installer offers two configurations for Service Fabric development:

- Install the Service Fabric runtime, SDK, and tools for Visual Studio 2015 (Requires Visual Studio 2015 Update 2 or later)
- Install the Service Fabric runtime and SDK only (no Visual Studio tools)

Enable PowerShell script execution

Service Fabric uses Windows PowerShell scripts for creating a local development cluster and for deploying applications from Visual Studio. By default, Windows blocks these scripts from running. To enable them, you must modify your PowerShell execution policy. Open PowerShell as an administrator and enter the following command:

```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Force -Scope CurrentUser
```

Next steps

Now that you've finished setting up your development environment, start building and running apps.

- Create your first Service Fabric application in Visual Studio
- Learn how to deploy and manage applications on your local cluster

- Learn about the programming models: Reliable Services and Reliable Actors
- Check out the Service Fabric code samples on GitHub
- Visualize your cluster by using Service Fabric Explorer
- Follow the Service Fabric learning path to get a broad introduction to the platform

Prepare your development environment on Linux

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Sean McKenna • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • James Dunn

---

To deploy and run Azure Service Fabric applications on your Linux development machine, install the runtime and common SDK. You can also install optional SDKs for Java and .NET Core.

Prerequisites

Supported operating system versions

The following operating system versions are supported for development:

- Ubuntu 16.04 (Xenial Xerus)

Update your apt sources

To install the SDK and the associated runtime package via apt-get, you must first update your apt sources.

1. Open a terminal.
2. Add the Service Fabric repo to your sources list.

```
sudo sh -c 'echo "deb [arch=amd64] http://apt-mo.trafficmanager.net/repos/servicefabric/ trusty main" >
/etc/apt/sources.list.d/servicefabric.list'
```

3. Add the new GPG key to your apt keyring.

```
sudo apt-key adv --keyserver apt-mo.trafficmanager.net --recv-keys 417A0893
```

4. Refresh your package lists based on the newly added repositories.

```
sudo apt-get update
```

Install and set up the SDK

Once your sources are updated, you can install the SDK.

1. Install the Service Fabric SDK package. You will be asked to confirm the installation and to agree to a license agreement.

```
sudo apt-get install servicefabricsdkcommon
```

2. Run the SDK setup script.

```
sudo /opt/microsoft/sdk/servicefabric/common/sdkcommonsetup.sh
```

Set up the Azure cross-platform CLI

The Azure cross-platform CLI includes commands for interacting with Service Fabric entities, including clusters and applications. It is based on Node.js so ensure that you have installed Node before proceeding with the instructions below.

1. Clone the github repo to your development machine.

```
git clone https://github.com/Azure/azure-xplat-cli.git
```

2. Switch into the cloned repo and install the CLI's dependencies using the Node Package Manager (npm).

```
cd azure-xplat-cli
npm install
```

3. Create a symlink from the bin/azure folder of the cloned repo to /usr/bin/azure so that it's added to your path and commands are available from any directory.

```
sudo ln -s $(pwd)/bin/azure /usr/bin/azure
```

4. Finally, enable auto-completion Service Fabric commands.

```
azure --completion >> ~/azure.completion.sh
echo 'source ~/azure.completion.sh' >> ~/.bash_profile
source ~/azure.completion.sh
```

Set up a local cluster

If everything has installed successfully, you should be able to start a local cluster.

1. Run the cluster setup script.

```
sudo /opt/microsoft/sdk/servicefabric/common/clustersetup/devclustersetup.sh
```

2. Open a web browser and navigate to http://localhost:19080/Explorer. If the cluster has started, you should see the Service Fabric Explorer dashboard.

At this point, you are able to deploy pre-built Service Fabric application packages or new ones based on guest

containers or guest executables. To build new services using the Java or .NET Core SDKs, follow the optional setup steps below.

Install the Java SDK and Eclipse Neon plugin (optional)

The Java SDK provides the libraries and templates required to build Service Fabric services using Java.

1. Install the Java SDK package.

```
sudo apt-get install servicefabricsdkjava
```

2. Run the SDK setup script.

```
sudo /opt/microsoft/sdk/servicefabric/java/sdkjavasetup.sh
```

You can install the Eclipse plugin for Service Fabric from within the Eclipse Neon IDE.

1. In Eclipse, ensure that you have Buildship version 1.0.17 or later installed. You can check the versions of installed components by choosing **Help > Installation Details**. You can update Buildship using the instructions here.
2. To install the Service Fabric plugin, choose **Help > Install New Software...**
3. In the "Work with" textbox, enter: http://dl.windowsazure.com/eclipse/servicefabric
4. Click Add.

5. Choose the Service Fabric plugin and click next.
6. Proceed through the installation and accept the end-user license agreement.

Install the .NET Core SDK (optional)

The .NET Core SDK provides the libraries and templates required to build Service Fabric services using cross-platform .NET Core.

1. Install the .NET Core SDK package.

```
sudo apt-get install servicefabricsdkcsharp
```

2. Run the SDK setup script.

```
sudo /opt/microsoft/sdk/servicefabric/csharp/sdkcsharpsetup.sh
```

Next steps

- Create your first Java application on Linux
- Prepare your development environment on OSX

Set up your development environment on Mac OS X

11/15/2016 • 1 min to read • Edit on GitHub

Contributors

Sean McKenna • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • James Dunn

You can build Service Fabric applications to run on Linux clusters using Mac OS X. This article covers how to set up your Mac for development.

Prerequisites

Service Fabric does not run natively on OS X. To run a local Service Fabric cluster, we provide a pre-configured Ubuntu virtual machine using Vagrant and VirtualBox. Before you get started, you need:

- Vagrant (v1.8.4 or later)
- VirtualBox

Create the local VM

To create the local VM containing a 5-node Service Fabric cluster, do the following:

1. Clone the Vagrantfile repo

   ```
   git clone https://github.com/azure/service-fabric-linux-vagrant-onebox.git
   ```

2. Navigate to the local clone of the repo

   ```
   cd service-fabric-linux-vagrant-onebox
   ```

3. (Optional) Modify the default VM settings

   By default, the local VM is configured as follows:

   - 3 GB of memory allocated
   - Private host network configured at IP 192.168.50.50 enabling passthrough of traffic from the Mac host

     You can change either of these settings or add other configuration to the VM in the Vagrantfile. See the Vagrant documentation for the full list of configuration options.

4. Create the VM

   ```
   vagrant up
   ```

   This step downloads the preconfigured VM image, boot it locally, and then set up a local Service Fabric cluster in it. You should expect it to take a few minutes. If setup completes successfully, you will see a message in the output indicating that the cluster is starting up.

```
==> default: Obtained writer lock for /home/sfuser/sfdevcluster/data/_Node_4/Fabric/work/UnreliableTransportSettings.ini
==> default: Released writer lock on /home/sfuser/sfdevcluster/data/_Node_4/Fabric/work/UnreliableTransportSettings.ini
==> default: Created the HostedService/_Node_4_Fabric section for node _Node_4.
==> default: Host Settings file generated at /home/sfuser/sfdevcluster/data/FabricHostSettings.xml
==> default: DeploymentValidator: warning: Skipping StoreConnectionString validation on consumer AzureWinFabCsv.
==> default: Current profile will be disabled by default for firewall rule
==> default: fabricNode is System.Fabric.Management.ServiceModel.InfrastructureNodeType
==> default: fabricNode is System.Fabric.Management.ServiceModel.InfrastructureNodeType
==> default: fabricNode is System.Fabric.Management.ServiceModel.InfrastructureNodeType
==> default: fabricNode is System.Fabric.Management.ServiceModel.InfrastructureNodeType
==> default: fabricNode is System.Fabric.Management.ServiceModel.InfrastructureNodeType
==> default: Released writer lock on /home/sfuser/sfdevcluster/data/lock
==> default: Setting sfuser as owner for FabricDataRoot
==> default: Setting sfuser as owner of FabricBinRoot
==> default: Enable service fabric service
==> default: Created symlink from /etc/systemd/system/multi-user.target.wants/servicefabric.service to /etc/systemd/system/servicefabric.service.
==> default: Starting service
==> default: Fabric service started. Cluster should come up in a few minutes
seanmck@seanmck-mbp sf-linux-vagrant (master)*$
```

5. Test that the cluster has been set up correctly by navigating to Service Fabric Explorer at http://192.168.50.50:19080/Explorer (assuming you kept the default private network IP).

Install the Service Fabric plugin for Eclipse Neon (optional)

Service Fabric provides a plugin for the Eclipse Neon IDE that can simplify the process of building and deploying Java services.

1. In Eclipse, ensure that you have Buildship version 1.0.17 or later installed. You can check the versions of installed components by choosing **Help > Installation Details**. You can update Buildship using the instructions here.
2. To install the Service Fabric plugin, choose **Help > Install New Software...**
3. In the "Work with" textbox, enter: http://dl.windowsazure.com/eclipse/servicefabric.
4. Click Add.

5. Choose the Service Fabric plugin and click next.
6. Proceed through the installation and accept the end-user license agreement.

Next steps

- Create your first Service Fabric application for Linux
- Create a Service Fabric cluster in the Azure portal
- Create a Service Fabric cluster using the Azure Resource Manager
- Understand the Service Fabric application model

Create your first Azure Service Fabric application

11/15/2016 • 5 min to read • Edit on GitHub

Contributors

unknown • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Todd Abel • Sean McKenna • Mark Fussell • katiecumming

The Service Fabric SDK includes an add-in for Visual Studio that provides templates and tools for creating, deploying, and debugging Service Fabric applications. This topic walks you through the process of creating your first application in Visual Studio.

Prerequisites

Before you get started, make sure that you have set up your development environment.

Video walkthrough

The following video walks through the steps in this tutorial:

Create the application

A Service Fabric application can contain one or more services, each with a specific role in delivering the application's functionality. By using the New Project wizard, you can create an application project, along with your first service project. You can add more services later.

1. Launch Visual Studio as an administrator.
2. Click **File > New Project > Cloud > Service Fabric Application**.
3. Name the application and click **OK**.

4. On the next page, choose **Stateful** as the first service type to include in your application. Name it and click **OK**.

> **NOTE**
>
> For more information about the options, see Service Fabric programming model overview.

Visual Studio creates the application project and the stateful service project and displays them in Solution

Explorer.

The application project does not contain any code directly. Instead, it references a set of service projects. In addition, it contains three other types of content:

- **Publish profiles**: Used to manage tooling preferences for different environments.
- **Scripts**: Includes a PowerShell script for deploying/upgrading your application. Visual Studio uses the script behind-the-scenes by Visual Studio. The script can also be invoked directly at the command line.
- **Application definition**: Includes the application manifest under *ApplicationPackageRoot*. Associated application parameter files are under *ApplicationParameters*, which define the application and allow you to configure it specifically for a given environment.

  For an overview of the contents of the service project, see Getting started with Reliable Services.

Deploy and debug the application

Now that you have an application, try running it.

1. Press F5 in Visual Studio to deploy the application for debugging.

   > **NOTE**
   > Deploying takes a while the first time, as Visual Studio is creating a local cluster for development. A local cluster runs the same platform code that you build on in a multi-machine cluster, just on a single machine. The cluster creation status displays in the Visual Studio output window.

   When the cluster is ready, you get a notification from the local cluster system tray manager application included with the SDK.

2. Once the application starts, Visual Studio automatically brings up the Diagnostics Event Viewer, where you can see trace output from the service.

In the case of the stateful service template, the messages simply show the counter value incrementing in the `RunAsync` method of MyStatefulService.cs.

3. Expand one of the events to see more details, including the node where the code is running. In this case, it is _Node_2, though it may differ on your machine.

The local cluster contains five nodes hosted on a single machine. It mimics a five-node cluster, where nodes are on distinct machines. Let's take down one of the nodes on the local cluster to simulate the loss of a machine while exercising the Visual Studio debugger at the same time.

> **NOTE**
>
> The application diagnostic events emitted by the project template use the included `ServiceEventSource` class. For more information, see How to monitor and diagnose services locally.

4. Find the class in your service project that derives from StatefulService (for example, MyStatefulService) and set a breakpoint on the first line of the `RunAsync` method.

```
0 references
protected override async Task RunAsync(CancellationToken cancellationToken)
{
    // TODO: Replace the following sample code with your own logic
    //       or remove this RunAsync override if it's not needed in your service.

    var myDictionary = await this.StateManager.GetOrAddAsync<IReliableDictionary<string, long>>("myDictionary");

    while (true)
    {
        cancellationToken.ThrowIfCancellationRequested();

        using (var tx = this.StateManager.CreateTransaction())
        {
            var result = await myDictionary.TryGetValueAsync(tx, "Counter");

            ServiceEventSource.Current.ServiceMessage(this, "Current Counter Value: {0}",
                result.HasValue ? result.Value.ToString() : "Value does not exist.");

            await myDictionary.AddOrUpdateAsync(tx, "Counter", 0, (key, value) => ++value);

            // If an exception is thrown before calling CommitAsync, the transaction aborts, all changes are
            // discarded, and nothing is saved to the secondary replicas.
            await tx.CommitAsync();
        }

        await Task.Delay(TimeSpan.FromSeconds(1), cancellationToken);
    }
}
```

5. Right-click the Local Cluster Manager system tray app and choose **Manage Local Cluster** to launch Service Fabric Explorer.

Service Fabric Explorer offers a visual representation of a cluster--including the set of applications deployed to it and the set of physical nodes that make it up. To learn more about Service Fabric Explorer, see Visualizing your cluster.

6. In the left pane, expand **Cluster > Nodes** and find the node where your code is running.
7. Click **Actions > Deactivate (Restart)** to simulate a machine restarting. (Note you can also deactivate from the context menu in the node list view in the left pane.)

Momentarily, you should see your breakpoint hit in Visual Studio as the computation you were doing on one node seamlessly fails over to another.

8. Return to the Diagnostic Events Viewer and observe the messages. The counter has continued incrementing, even though the events are actually coming from a different node.

16:34:23.985 ServiceMessage Current Counter Value: 184


```
{
  "Timestamp": "2016-03-27T16:34:23.9859057-07:00",
  "ProviderName": "MyCompany-MyApplication-MyStatefulService",
  "Id": 2,
  "Message": "Current Counter Value: 184",
  "ProcessId": 23220,
  "Level": "Informational",
  "Keywords": "0x0000F00000000000",
  "EventName": "ServiceMessage",
  "Payload": {
    "serviceName": "fabric:/MyApplication/MyStatefulService",
    "serviceTypeName": "MyStatefulServiceType",
    "replicaOrInstanceId": 131035950208891983,
    "partitionId": "660501c3-97e2-4297-b1cc-d1c71e2c0c7c",
    "applicationName": "fabric:/MyApplication",
    "applicationTypeName": "MyApplicationType",
    "nodeName": "_Node_0",
    "message": "Current Counter Value: 184"
  }
}
```


Switch cluster mode

By default, the local development cluster is configured to run as a 5 node cluster, which is useful for debugging services deployed across multiple nodes. Deploying an application to the 5 node development cluster can take some time, however. If you want to iterate code changes quickly, without running your app on 5 nodes, you can switch the development cluster to 1-Node mode. To run your code on a cluster with one node, right-click on the Local Cluster Manager in the system tray and select **Switch Cluster Mode -> 1 Node**.

When you change cluster mode the development cluster resets and all applications provisioned or running on the cluster are removed.

Cleaning up

Before wrapping up, it's important to remember that the local cluster is very real. Stopping the debugger removes your application instance and unregisters the application type. The cluster continues to run in the background, however. You have several options to manage the cluster:

1. To shut down the cluster but keep the application data and traces, click **Stop Local Cluster** in the system tray app.
2. To delete the cluster entirely, click **Remove Local Cluster** in the system tray app. This option will result in another slow deployment the next time you press F5 in Visual Studio. Delete the cluster only if you don't intend to use the local cluster for some time or if you need to reclaim resources.

Next steps

- Learn how to create a cluster in Azure or a standalone cluster on Windows.
- Try creating a service using the Reliable Services or Reliable Actors programming models.
- Learn how you can expose your services to the Internet with a web service front end.
- Walk through a hands-on-lab and create a stateless service, configure monitoring and health reports, and perform an application upgrade.

Create your first Azure Service Fabric application

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

Sean McKenna • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Todd Abel

Service Fabric provides SDKs for building services on Linux in both .NET Core and Java. In this tutorial, we will look at how to create an application for Linux and build a service using Java.

Prerequisites

Before you get started, make sure that you have set up your Linux development environment. If you are using Mac OS X, you can set up a Linux one-box environment in a virtual machine using Vagrant.

Create the application

A Service Fabric application can contain one or more services, each with a specific role in delivering the application's functionality. The Service Fabric SDK for Linux includes a Yeoman generator that makes it easy to create your first service and to add more later. Let's use Yeoman to create a new application with a single service.

1. In a terminal, type **yo azuresfjava**.
2. Name your application.
3. Choose the type of your first service and name it. For the purposes of this tutorial, we will choose a Reliable Actor Service.

```
1. vagrant@vagrant: /tmp (vagrant)
vagrant@vagrant:~/src$ clear
vagrant@vagrant:~/src$ yo azuresfjava

     _-----_     ╭──────────────────────────╮
    |       |    |    Welcome to Service    |
    |--(o)--|    |      Fabric java app     |
   `---------´   |         generator        |
    ( _´U`_ )    ╰──────────────────────────╯
    /___A___\   /
     |  ~  |
   __'.___.'__
 ´   `  |° ´ Y `

? Name your application myapp
? Choose a framework for your service Reliable Actor Service
? Enter the name of actor service :  myactorsvc
   create myapp/myactorsvc/src/reliableactor/myactorsvcImpl.java
   create myapp/myactorsvc/src/reliableactor/myactorsvcActorHost.java
   create myapp/myactorsvc/build.gradle
   create myapp/myapp/myactorsvcPkg/ServiceManifest.xml
   create myapp/myapp/ApplicationManifest.xml
   create myapp/myapp/myactorsvcPkg/Code/entryPoint.sh
   create myapp/myapp/myactorsvcPkg/Config/Settings.xml
   create myapp/myactorsvc/settings.gradle
   create myapp/myactorsvcInterface/src/reliableactor/myactorsvc.java
   create myapp/myactorsvcInterface/build.gradle
   create myapp/myactorsvcTestClient/src/reliableactor/test/myactorsvcTestClient.java
   create myapp/myactorsvcTestClient/build.gradle
   create myapp/myactorsvcTestClient/testclient.sh
   create myapp/myactorsvcTestClient/settings.gradle
   create myapp/build.gradle
   create myapp/settings.gradle
   create myapp/install.sh
   create myapp/uninstall.sh
   create myapp/myapp/myactorsvcPkg/Code/_readme.txt
   create myapp/myapp/myactorsvcPkg/Config/_readme.txt
   create myapp/myapp/myactorsvcPkg/Data/_readme.txt
vagrant@vagrant:~/src$
```

> **NOTE**
>
> For more information about the options, see Service Fabric programming model overview.

Build the application

The Service Fabric Yeoman templates include a build script for Gradle, which you can use to build the app from the terminal.

```
cd myapp
gradle
```

Deploy the application

Once the application is built, you can deploy it to the local cluster using the Azure CLI.

1. Connect to the local Service Fabric cluster.

   ```
   azure servicefabric cluster connect
   ```

2. Use the install script provided in the template to copy the application package to the cluster's image store, register the application type, and create an instance of the application.

```
./install.sh
```

3. Open a browser and navigate to Service Fabric Explorer at http://localhost:19080/Explorer (replace localhost with the private IP of the VM if using Vagrant on Mac OS X).
4. Expand the Applications node and note that there is now an entry for your application type and another for the first instance of that type.

Start the test client and perform a failover

Actor projects do not do anything on their own. They require another service or client to send them messages. The actor template includes a simple test script that you can use to interact with the actor service.

1. Run the script using the watch utility to see the output of the actor service.

```
cd myactorsvcTestClient
watch -n 1 ./testclient.sh
```

2. In Service Fabric Explorer, locate node hosting the primary replica for the actor service. In the screenshot below, it is node 3.

3. Click the node you found in the previous step, then select **Deactivate (restart)** from the Actions menu. This will restart one of the five nodes in your local cluster and force a failover to one of the secondary replicas running on another node. As you do this, pay attention to the output from the test client and note that the counter continues to increment despite the failover.

Build and deploy an application with the Eclipse Neon plugin

If you installed the Service Plugin for Eclipse Neon, you can use it to create, build, and deploy Service Fabric applications built with Java. When installing Eclipse, choose the **Eclipse IDE for Java developers**.

Create the application

The Service Fabric plugin is available through Eclipse extensibility.

1. In Eclipse, choose **File > Other > Service Fabric**. You see a set of options, including Actors and Containers.

2. In this case, choose Stateless Service.
3. You are asked to confirm the use of the Service Fabric perspective, which optimizes Eclipse for use with Service Fabric projects. Choose 'Yes'.

Deploy the application

The Service Fabric templates include a set of Gradle tasks for building and deploying applications, which you can trigger through Eclipse.

1. Choose **Run > Run Configurations**.
2. Expand **Gradle Project** and choose **ServiceFabricDeployer**.
3. Click **Run**.

Your app will build and deploy within a few moments. You can monitor its status from Service Fabric Explorer.

Next steps

- Learn more about Reliable Actors
- Interacting with Service Fabric clusters using the Azure CLI

Create your first Azure Service Fabric application

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Mani Ramaswamy • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

Service Fabric provides SDKs for building services on Linux in both .NET Core and Java. In this tutorial, we look at how to create an application for Linux and build a service using C# (.NET Core).

Prerequisites

Before you get started, make sure that you have set up your Linux development environment. If you are using Mac OS X, you can set up a Linux one-box environment in a virtual machine using Vagrant.

Create the application

A Service Fabric application can contain one or more services, each with a specific role in delivering the application's functionality. The Service Fabric SDK for Linux includes a Yeoman generator that makes it easy to create your first service and to add more later. Let's use Yeoman to create an application with a single service.

1. In a terminal, type the following command to start building the scaffolding: `yo azuresfcsharp`
2. Name your application.
3. Choose the type of your first service and name it. For the purposes of this tutorial, we choose a Reliable Actor Service.

```
mani@MyUbuntuVMw35hgnmsnibju:~$ yo azuresfcsharp

     _-----_     ╭──────────────────────────╮
    |       |    │      Welcome to Service  │
    |--(o)--|    │     Fabric generator for │
   `---------´   │            CSharp        │
    ( _´U`_ )    ╰──────────────────────────╯
    /___A___\   /
     |  ~  |
   __'.___.'__
 ´   `  |° ´ Y `

? Name your application myapp
? Choose a framework for your service Reliable Actor Service
? Enter the name of actor service :  myactorsvc
   create myapp/myapp/myactorsvcPkg/ServiceManifest.xml
   create myapp/myapp/ApplicationManifest.xml
   create myapp/myapp/myactorsvcPkg/Code/entryPoint.sh
   create myapp/myapp/myactorsvcPkg/Config/Settings.xml
   create myapp/src/myapp/myactorsvc.Interfaces/Imyactorsvc.cs
   create myapp/src/myapp/myactorsvc.Interfaces/project.json
   create myapp/src/myapp/myactorsvc/myactorsvc.cs
   create myapp/src/myapp/myactorsvc/project.json
   create myapp/src/myapp/myactorsvc/Program.cs
   create myapp/src/myapp/myactorsvcTestClient/project.json
   create myapp/src/myapp/myactorsvcTestClient/Program.cs
   create myapp/myactorsvcTestClient/testclient.sh
   create myapp/install.sh
   create myapp/uninstall.sh
   create myapp/build.sh
   create myapp/myapp/myactorsvcPkg/Config/_readme.txt
   create myapp/myapp/myactorsvcPkg/Data/_readme.txt
mani@MyUbuntuVMw35hgnmsnibju:~$
```

> **NOTE**
> For more information about the options, see Service Fabric programming model overview.

Build the application

The Service Fabric Yeoman templates include a build script that you can use to build the app from the terminal (after navigating to the application folder).

```
cd myapp
./build.sh
```

Deploy the application

Once the application is built, you can deploy it to the local cluster using the Azure CLI.

1. Connect to the local Service Fabric cluster.

   ```
   azure servicefabric cluster connect
   ```

2. Use the install script provided in the template to copy the application package to the cluster's image store, register the application type, and create an instance of the application.

   ```
   ./install.sh
   ```

3. Open a browser and navigate to Service Fabric Explorer at http://localhost:19080/Explorer (replace localhost with the private IP of the VM if using Vagrant on Mac OS X).
4. Expand the Applications node and note that there is now an entry for your application type and another for the first instance of that type.

Start the test client and perform a failover

Actor projects do not do anything on their own. They require another service or client to send them messages. The actor template includes a simple test script that you can use to interact with the actor service.

1. Run the script using the watch utility to see the output of the actor service.

   ```
   cd myactorsvcTestClient
   watch -n 1 ./testclient.sh
   ```

2. In Service Fabric Explorer, locate node hosting the primary replica for the actor service. In the screenshot below, it is node 3.

3. Click the node you found in the previous step, then select **Deactivate (restart)** from the Actions menu. This action restarts one of the five nodes in your local cluster forcing a failover to a secondary replica running on another node. As you perform this action, pay attention to the output from the test client and note that the counter continues to increment despite the failover.

Next steps

- Learn more about Reliable Actors
- Interacting with Service Fabric clusters using the Azure CLI

Get started with deploying and upgrading applications on your local cluster

11/15/2016 • 8 min to read • Edit on GitHub

Contributors

unknown • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Mikkel Mork Hegnhoj • Sean McKenna • katiecumming • Andy Pasic • Matt Thalman

The Azure Service Fabric SDK includes a full local development environment that you can use to quickly get started with deploying and managing applications on a local cluster. In this article, you create a local cluster, deploy an existing application to it, and then upgrade that application to a new version, all from Windows PowerShell.

> **NOTE**
> This article assumes that you already set up your development environment.

Create a local cluster

A Service Fabric cluster represents a set of hardware resources that you can deploy applications to. Typically, a cluster is made up of anywhere from five to many thousands of machines. However, the Service Fabric SDK includes a cluster configuration that can run on a single machine.

It is important to understand that the Service Fabric local cluster is not an emulator or simulator. It runs the same platform code that is found on multi-machine clusters. The only difference is that it runs the platform processes that are normally spread across five machines on one machine.

The SDK provides two ways to set up a local cluster: a Windows PowerShell script and the Local Cluster Manager system tray app. In this tutorial, we use the PowerShell script.

> **NOTE**
> If you have already created a local cluster by deploying an application from Visual Studio, you can skip this section.

1. Launch a new PowerShell window as an administrator.
2. Run the cluster setup script from the SDK folder:

   ```
   & "$ENV:ProgramFiles\Microsoft SDKs\Service Fabric\ClusterSetup\DevClusterSetup.ps1"
   ```

   Cluster setup takes a few moments. After setup is finished, you should see output similar to:

```
Administrator: Windows PowerShell
PS C:\> & "$ENV:ProgramFiles\Microsoft SDKs\Service Fabric\ClusterSetup\DevClusterSetup.ps1"
WARNING: A local Service Fabric Cluster already exists on this machine and will be removed.
Do you want to continue [Y/N]?: Y
Removing cluster configuration...
Cleaning existing certificates...
Stopping all logman sessions...
Cleaning log and data folder...

Using Cluster Data Root: C:\SfDevCluster\Data
Using Cluster Log Root: C:\SfDevCluster\Log

Create node configuration succeeded
Starting service FabricHostSvc. This may take a few minutes...

Waiting for Service Fabric Cluster to be ready. This may take a few minutes...
Local Cluster ready status: 4% completed.
Local Cluster ready status: 8% completed.
Local Cluster ready status: 12% completed.
Local Cluster ready status: 17% completed.
Local Cluster ready status: 21% completed.
Local Cluster ready status: 25% completed.
Local Cluster ready status: 29% completed.
Local Cluster ready status: 33% completed.
Local Cluster ready status: 100% completed.

Waiting for Naming Service to be ready. This may take a few minutes...
Naming Service ready status: 8% completed.
Naming Service is ready now...


Local Service Fabric Cluster created successfully.

================================================
## To connect using Powershell, open an a new powershell window and connect using 'Connect-ServiceFabricCluster' command
 (without any arguments)."

## To connect using Service Fabric Explorer, run ServiceFabricExplorer and connect using 'Local/OneBox Cluster'."

## To manage using Service Fabric Local Cluster Manager (system tray app), run ServiceFabricLocalClusterManager.exe"
================================================
PS C:\>
```

You are now ready to try deploying an application to your cluster.

Deploy an application

The Service Fabric SDK includes a rich set of frameworks and developer tooling for creating applications. If you are interested in learning how to create applications in Visual Studio, see Create your first Service Fabric application in Visual Studio.

In this tutorial, we will use an existing sample application (called WordCount) so that we can focus on the management aspects of the platform--including deployment, monitoring, and upgrade.

1. Launch a new PowerShell window as an administrator.
2. Import the Service Fabric SDK PowerShell module.

```
Import-Module "$ENV:ProgramFiles\Microsoft SDKs\Service
Fabric\Tools\PSModule\ServiceFabricSDK\ServiceFabricSDK.psm1"
```

3. Create a directory to store the application that you download and deploy, such as C:\ServiceFabric.

```
mkdir c:\ServiceFabric\
cd c:\ServiceFabric\
```

4. Download the WordCount application to the location you created. Note: the Microsoft Edge browser saves the file with a *.zip* extension. Change the file extension to *.sfpkg*.
5. Connect to the local cluster:

```
Connect-ServiceFabricCluster localhost:19000
```

6. Create a new application using the SDK's deployment command with a name and a path to the application package.

```
Publish-NewServiceFabricApplication -ApplicationPackagePath c:\ServiceFabric\WordCountV1.sfpkg -
ApplicationName "fabric:/WordCount"
```

If all goes well, you should see output like the following:

```
Copying application to image store...
Copy application package succeeded
Registering application type...
Register application type succeeded
Removing application package from image store...
Remove application package succeeded
Creating application...
Create application succeeded.
```

7. To see the application in action, launch the browser and navigate to http://localhost:8081/wordcount/index.html. You should see:

The WordCount application is very simple. It includes client-side JavaScript code to generate random five-character "words", which are then relayed to the application via ASP.NET Web API. A stateful service tracks the number of words counted. They are partitioned based on the first character of the word. You can find the source code for the WordCount app in the getting started samples.

The application that we deployed contains four partitions. So words beginning with A through G are stored in the first partition, words beginning with H through N are stored in the second partition, and so on.

View application details and status

Now that we have deployed the application, let's look at some of the app details in PowerShell.

1. Query all deployed applications on the cluster:

```
Get-ServiceFabricApplication
```

Assuming that you have only deployed the WordCount app, you see something similar to:

```
PS C:\Service Fabric> Get-ServiceFabricApplication


ApplicationName        : fabric:/WordCount
ApplicationTypeName    : WordCount
ApplicationTypeVersion : 1.0.0
ApplicationStatus      : Ready
HealthState            : Ok
ApplicationParameters  : { "WordCountWebService_InstanceCount" = "1" }
```

2. Go to the next level by querying the set of services that are included in the WordCount application.

```
Get-ServiceFabricService -ApplicationName 'fabric:/WordCount'
```

```
PS C:\Service Fabric> Get-ServiceFabricService -ApplicationName 'fabric:/WordCount'


ServiceName            : fabric:/WordCount/WordCountService
ServiceKind            : Stateful
ServiceTypeName        : WordCountServiceType
IsServiceGroup         : False
ServiceManifestVersion : 1.0.0
HasPersistedState      : True
ServiceStatus          : Active
HealthState            : Ok


ServiceKind            : Stateless
ServiceName            : fabric:/WordCount/WordCountWebService
ServiceTypeName        : WordCountWebServiceType
ServiceManifestVersion : 1.0.0
HealthState            : Ok
ServiceStatus          : Active
IsServiceGroup         : False
```

The application is made up of two services--the web front end and the stateful service that manages the words.

3. Finally, take a look at the list of partitions for WordCountService:

```
Get-ServiceFabricPartition 'fabric:/WordCount/WordCountService'
```

```
PS C:\Service Fabric> Get-ServiceFabricPartition 'fabric:/WordCount/WordCountService'


PartitionId            : 59393164-71f9-4e45-9f30-9bdce46c4349
PartitionKind          : Int64Range
PartitionLowKey        : 1
PartitionHighKey       : 7
PartitionStatus        : Ready
LastQuorumLossDuration : 00:00:00
MinReplicaSetSize      : 2
TargetReplicaSetSize   : 3
HealthState            : Ok
DataLossNumber         : 130888309725707433
ConfigurationNumber    : 8589934592

PartitionId            : d645a771-67bd-4f4f-9236-c408dcd107f8
PartitionKind          : Int64Range
PartitionLowKey        : 8
PartitionHighKey       : 14
PartitionStatus        : Ready
LastQuorumLossDuration : 00:00:00
MinReplicaSetSize      : 2
TargetReplicaSetSize   : 3
HealthState            : Ok
DataLossNumber         : 130888309725707433
ConfigurationNumber    : 8589934592

PartitionId            : 08ab1875-b3cd-4def-82f4-7c92807bfb50
PartitionKind          : Int64Range
PartitionLowKey        : 21
PartitionHighKey       : 26
PartitionStatus        : Ready
LastQuorumLossDuration : 00:00:00
MinReplicaSetSize      : 2
TargetReplicaSetSize   : 3
HealthState            : Ok
DataLossNumber         : 130888309725707433
ConfigurationNumber    : 8589934592

PartitionId            : 474326d9-3e36-4be3-aa73-bc1e3f6fca41
PartitionKind          : Int64Range
PartitionLowKey        : 15
PartitionHighKey       : 20
PartitionStatus        : Ready
LastQuorumLossDuration : 00:00:00
MinReplicaSetSize      : 2
TargetReplicaSetSize   : 3
HealthState            : Ok
DataLossNumber         : 130888309725707433
ConfigurationNumber    : 8589934592
```

The set of commands that you used, like all Service Fabric PowerShell commands, are available for any cluster that you might connect to, local or remote.

For a more visual way to interact with the cluster, you can use the web-based Service Fabric Explorer tool by navigating to http://localhost:19080/Explorer in the browser.

> **NOTE**
>
> To learn more about Service Fabric Explorer, see Visualizing your cluster with Service Fabric Explorer.

Upgrade an application

Service Fabric provides no-downtime upgrades by monitoring the health of the application as it rolls out across the cluster. Let's perform a simple upgrade of the WordCount application.

The new version of the application now counts only words that begin with a vowel. As the upgrade rolls out, we see two changes in the application's behavior. First, the rate at which the count grows should slow, since fewer words are being counted. Second, since the first partition has two vowels (A and E) and all other partitions contain only one each, its count should eventually start to outpace the others.

1. Download the WordCount v2 package to the same location where you downloaded the v1 package.
2. Return to your PowerShell window and use the SDK's upgrade command to register the new version in the cluster. Then begin upgrading the fabric:/WordCount application.

   ```
   Publish-UpgradedServiceFabricApplication -ApplicationPackagePath C:\ServiceFabric\WordCountV2.sfpkg -
   ApplicationName "fabric:/WordCount" -UpgradeParameters @{"FailureAction"="Rollback";
   "UpgradeReplicaSetCheckTimeout"=1; "Monitored"=$true; "Force"=$true}
   ```

   You should see output in PowerShell that looks similar to the following as the upgrade begins.

   ```
   Copying application package...
   Copy application package succeeded
   Registering application type...
   Register application type succeeded
   Start upgrading application...


   ApplicationName                : fabric:/WordCount
   ApplicationParameters          : { "WordCountWebService_InstanceCount" = "1" }
   TargetApplicationTypeVersion   : 2.0.0
   UpgradeKind                    : Rolling
   ForceRestart                   : False
   UpgradeMode                    : UnmonitoredAuto
   UpgradeReplicaSetCheckTimeout  : 00:00:01

   Waiting for upgrade...
   Waiting for upgrade...
   Waiting for upgrade...
   Upgrade completed
   ```

3. While the upgrade is proceeding, you may find it easier to monitor its status from Service Fabric Explorer. Launch a browser window and navigate to http://localhost:19080/Explorer. Expand **Applications** in the tree on the left, then choose **WordCount**, and finally **fabric:/WordCount**. In the essentials tab, you will see the status of the upgrade as it proceeds through the cluster's upgrade domains.

   As the upgrade proceeds through each domain, health checks are performed to ensure that the application is behaving properly.

4. If you rerun the earlier query for the set of services in the fabric:/WordCount application, notice that the version of WordCountService changed but the version of WordCountWebService did not:

   ```
   Get-ServiceFabricService -ApplicationName 'fabric:/WordCount'
   ```

```
ServiceName             : fabric:/WordCount/WordCountService
ServiceKind             : Stateful
ServiceTypeName         : WordCountServiceType
IsServiceGroup          : False
ServiceManifestVersion  : 2.0.0
HasPersistedState       : True
ServiceStatus           : Active
HealthState             : Ok


ServiceKind             : Stateless
ServiceName             : fabric:/WordCount/WordCountWebService
ServiceTypeName         : WordCountWebServiceType
ServiceManifestVersion  : 1.0.0
HealthState             : Ok
ServiceStatus           : Active
IsServiceGroup          : False
```

This highlights how Service Fabric manages application upgrades. It touches only the set of services (or code/configuration packages within those services) that have changed, which makes the process of upgrading faster and more reliable.

5. Finally, return to the browser to observe the behavior of the new application version. As expected, the count progresses more slowly, and the first partition ends up with slightly more of the volume.

Cleaning up

Before wrapping up, it's important to remember that the local cluster is real. Applications continue to run in the background until you remove them. Depending on the nature of your apps, a running app can take up significant resources on your machine. You have several options to manage applications and the cluster:

1. To remove an individual application and all its data, run the following:

```
Unpublish-ServiceFabricApplication -ApplicationName "fabric:/WordCount"
```

Or, delete the application from the **ACTIONS** menu of Service Fabric Explorer or the context menu in the

application list view of the left-hand pane.

2. After deleting the application from the cluster, you can unregister versions 1.0.0 and 2.0.0 of the WordCount application type. Deletion removes the application packages, including the code and configuration, from the cluster's image store.

```
Remove-ServiceFabricApplicationType -ApplicationTypeName WordCount -ApplicationTypeVersion 2.0.0
Remove-ServiceFabricApplicationType -ApplicationTypeName WordCount -ApplicationTypeVersion 1.0.0
```

   Or, in Service Fabric Explorer, choose **Unprovision Type** for the application.

3. To shut down the cluster but keep the application data and traces, click **Stop Local Cluster** in the system tray app.
4. To delete the cluster entirely, click **Remove Local Cluster** in the system tray app. This option will result in another slow deployment the next time you press F5 in Visual Studio. Remove the local cluster only if you don't intend to use it for some time or if you need to reclaim resources.

1 Node and 5 Node cluster mode

When working with the local cluster to develop applications, you often find yourself doing quick iterations of writing code, debugging, changing code, debugging etc. To help optimize this process, the local cluster can run in two modes: 1 Node or 5 Node. Both cluster modes have their benefits. 5 Node cluster mode enables you to work with a real cluster. You can test failover scenarios, work with more instances and replicas of your services. 1 Node cluster mode is optimized to do quick deployment and registration of services, to help you quickly validate code using the Service Fabric runtime.

Both 1 Node cluster mode and 5 Node cluster mode is not an emulator or simulator. It runs the same platform code that is found on multi-machine clusters.

> **NOTE**
> This feature is available in SDK version 5.2 and above.

To change the cluster mode to a 1 Node cluster, either use the Service Fabric Local Cluster Manager or use PowerShell the following way:

1. Launch a new PowerShell window as an administrator.
2. Run the cluster setup script from the SDK folder:

```
 & "$ENV:ProgramFiles\Microsoft SDKs\Service Fabric\ClusterSetup\DevClusterSetup.ps1" -
CreateOneNodeCluster
```

   Cluster setup takes a few moments. After setup is finished, you should see output similar to:

If you are using the Service Fabric Local Cluster Manager:

> **WARNING**
>
> When changing cluster mode, the current cluster is being removed from your system and a new cluster is being created. The data you must have stored in the cluster, will be deleted when you change cluster mode.

Next steps

- Now that you have deployed and upgraded some pre-built applications, you can try building your own in Visual Studio.
- All the actions performed on the local cluster in this article can be performed on an Azure cluster as well.
- The upgrade that we performed in this article was basic. See the upgrade documentation to learn more about the power and flexibility of Service Fabric upgrades.

Service Fabric programming model overview

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Sean McKenna • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Harry Chen • Vaclav Turecek • Ryan Wike • katiecumming • Andy De George

Service Fabric offers multiple ways to write and manage your services. Services can choose to use the Service Fabric APIs to take full advantage of the platform's features and application frameworks, or services can simply be any compiled executable program written in any language and simply hosted on a Service Fabric cluster.

Guest Executable

A guest executable is an arbitrary executable, written in any language, so you can take your existing applications and host them on a Service Fabric cluster. A guest executable can be packaged in an application and hosted alongside other services. Service Fabric handles orchestration and simple execution management of the executable, ensuring it stays up and running according to the service description. However, because guest executables do not integrate directly with Service Fabric APIs, they do not benefit from the full set of features the platform offers, such as custom health and load reporting, service endpoint registration, and stateful compute.

Get started with guest executables by deploying your first guest executable application.

Reliable Services

Reliable Services is a light-weight framework for writing services that integrate with the Service Fabric platform and benefit from the full set of platform features. Reliable Services provide a minimal set of APIs that allow the Service Fabric runtime to manage the lifecycle of your services and that allow your services to interact with the runtime. The application framework is minimal, giving you full control over design and implementation choices, and can be used to host any other application framework, such as ASP.NET MVC or Web API.

Reliable Services can be stateless, similar to most service platforms, such as web servers or Worker Roles in Azure Cloud Services, in which each instance of the service is created equal and state is persisted in an external solution, such as Azure DB or Azure Table Storage.

Reliable Services can also be stateful, exclusive to Service Fabric, where state is persisted directly in the service itself using Reliable Collections. State is made highly-available through replication and distributed through partitioning, all managed automatically by Service Fabric.

Learn more about Reliable Services or get started by writing your first Reliable Service.

Reliable Actors

Built on top of Reliable Services, the Reliable Actor framework is an application framework that implements the Virtual Actor pattern, based on the actor design pattern. The Reliable Actor framework uses independent units of compute and state with single-threaded execution called actors. The Reliable Actor framework provides built-in communication for actors and pre-set state persistence and scale-out configurations.

As Reliable Actors itself is an application framework built on Reliable Services, it is fully integrated with the Service Fabric platform and benefits from the full set of features offered by the platform.

Next steps

Learn more about Reliable Actors or get started by writing your first Reliable Actor service

Model an application in Service Fabric

11/22/2016 • 8 min to read • Edit on GitHub

Contributors

unknown • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Sean McKenna • Andy De George • ShawnJackson • alexwun • Mark Fussell • jastru • Jesse Benson • Bill Craun

This article provides an overview of the Azure Service Fabric application model. It also describes how to define an application and service via manifest files and get the application packaged and ready for deployment.

Understand the application model

An application is a collection of constituent services that perform a certain function or functions. A service performs a complete and standalone function (it can start and run independently of other services) and is composed of code, configuration, and data. For each service, code consists of the executable binaries, configuration consists of service settings that can be loaded at run time, and data consists of arbitrary static data to be consumed by the service. Each component in this hierarchical application model can be versioned and upgraded independently.

An application type is a categorization of an application and consists of a bundle of service types. A service type is a categorization of a service. The categorization can have different settings and configurations, but the core functionality remains the same. The instances of a service are the different service configuration variations of the same service type.

Classes (or "types") of applications and services are described through XML files (application manifests and service manifests) that are the templates against which applications can be instantiated from the cluster's image store. The schema definition for the ServiceManifest.xml and ApplicationManifest.xml file is installed with the Service Fabric SDK and tools to *C:\Program Files\Microsoft SDKs\Service Fabric\schemas\ServiceFabricServiceModel.xsd*.

The code for different application instances will run as separate processes even when hosted by the same Service Fabric node. Furthermore, the lifecycle of each application instance can be managed (i.e. upgraded) independently. The following diagram shows how application types are composed of service types, which in turn are composed of code, configuration, and packages. To simplify the diagram, only the code/config/data packages for `ServiceType4` are shown, though each service type would include some or all of those package types.

Two different manifest files are used to describe applications and services: the service manifest and application manifest. These are covered in detail in the ensuing sections.

There can be one or more instances of a service type active in the cluster. For example, stateful service instances, or replicas, achieve high reliability by replicating state between replicas located on different nodes in the cluster. This replication essentially provides redundancy for the service to be available even if one node in a cluster fails. A partitioned service further divides its state (and access patterns to that state) across nodes in the cluster.

The following diagram shows the relationship between applications and service instances, partitions, and replicas.

> **TIP**
> You can view the layout of applications in a cluster using the Service Fabric Explorer tool available at http://<yourclusteraddress>:19080/Explorer. For more details, see Visualizing your cluster with Service Fabric Explorer.

Describe a service

The service manifest declaratively defines the service type and version. It specifies service metadata such as service type, health properties, load-balancing metrics, service binaries, and configuration files. Put another way, it describes the code, configuration, and data packages that compose a service package to support one or more service types. Here is a simple example service manifest:

```
<?xml version="1.0" encoding="utf-8" ?>
<ServiceManifest Name="MyServiceManifest" Version="SvcManifestVersion1"
xmlns="http://schemas.microsoft.com/2011/01/fabric" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Description>An example service manifest</Description>
  <ServiceTypes>
    <StatelessServiceType ServiceTypeName="MyServiceType" />
  </ServiceTypes>
  <CodePackage Name="MyCode" Version="CodeVersion1">
    <SetupEntryPoint>
      <ExeHost>
        <Program>MySetup.bat</Program>
      </ExeHost>
    </SetupEntryPoint>
    <EntryPoint>
      <ExeHost>
        <Program>MyServiceHost.exe</Program>
      </ExeHost>
    </EntryPoint>
  </CodePackage>
  <ConfigPackage Name="MyConfig" Version="ConfigVersion1" />
  <DataPackage Name="MyData" Version="DataVersion1" />
</ServiceManifest>
```

Version attributes are unstructured strings and not parsed by the system. These are used to version each component for upgrades.

ServiceTypes declares what service types are supported by **CodePackages** in this manifest. When a service is instantiated against one of these service types, all code packages declared in this manifest are activated by running their entry points. The resulting processes are expected to register the supported service types at run time. Note that service types are declared at the manifest level and not the code package level. So when there are multiple code packages, they are all activated whenever the system looks for any one of the declared service types.

SetupEntryPoint is a privileged entry point that runs with the same credentials as Service Fabric (typically the *LocalSystem* account) before any other entry point. The executable specified by **EntryPoint** is typically the long-running service host. The presence of a separate setup entry point avoids having to run the service host with high privileges for extended periods of time. The executable specified by **EntryPoint** is run after **SetupEntryPoint** exits successfully. The resulting process is monitored and restarted (beginning again with **SetupEntryPoint**) if it ever terminates or crashes.

DataPackage declares a folder, named by the **Name** attribute, that contains arbitrary static data to be consumed by the process at run time.

ConfigPackage declares a folder, named by the **Name** attribute, that contains a *Settings.xml* file. This file contains sections of user-defined, key-value pair settings that the process can read back at run time. During an upgrade, if only the **ConfigPackage version** has changed, then the running process is not restarted. Instead, a callback notifies the process that configuration settings have changed so they can be reloaded dynamically. Here is an example *Settings.xml* file:

```
<Settings xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.microsoft.com/2011/01/fabric">
  <Section Name="MyConfigurationSecion">
    <Parameter Name="MySettingA" Value="Example1" />
    <Parameter Name="MySettingB" Value="Example2" />
  </Section>
</Settings>
```

> **NOTE**
> A service manifest can contain multiple code, configuration, and data packages. Each of those can be versioned independently.

Describe an application

The application manifest declaratively describes the application type and version. It specifies service composition metadata such as stable names, partitioning scheme, instance count/replication factor, security/isolation policy, placement constraints, configuration overrides, and constituent service types. The load-balancing domains into which the application is placed are also described.

Thus, an application manifest describes elements at the application level and references one or more service manifests to compose an application type. Here is a simple example application manifest:

```
<?xml version="1.0" encoding="utf-8" ?>
<ApplicationManifest
      ApplicationTypeName="MyApplicationType"
      ApplicationTypeVersion="AppManifestVersion1"
      xmlns="http://schemas.microsoft.com/2011/01/fabric"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Description>An example application manifest</Description>
  <ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="MyServiceManifest" ServiceManifestVersion="SvcManifestVersion1"/>
  </ServiceManifestImport>
  <DefaultServices>
     <Service Name="MyService">
         <StatelessService ServiceTypeName="MyServiceType" InstanceCount="1">
             <SingletonPartition/>
         </StatelessService>
     </Service>
  </DefaultServices>
</ApplicationManifest>
```

Like service manifests, **Version** attributes are unstructured strings and are not parsed by the system. These are also used to version each component for upgrades.

ServiceManifestImport contains references to service manifests that compose this application type. Imported service manifests determine what service types are valid within this application type.

DefaultServices declares service instances that are automatically created whenever an application is instantiated against this application type. Default services are just a convenience and behave like normal services in every respect after they have been created. They are upgraded along with any other services in the application instance and can be removed as well.

> **NOTE**
> An application manifest can contain multiple service manifest imports and default services. Each service manifest import can be versioned independently.

To learn how to maintain different application and service parameters for individual environments, see Managing application parameters for multiple environments.

Package an application

Package layout

The application manifest, service manifest(s), and other necessary package files must be organized in a specific

layout for deployment into a Service Fabric cluster. The example manifests in this article would need to be organized in the following directory structure:

```
PS D:\temp> tree /f .\MyApplicationType

D:\TEMP\MYAPPLICATIONTYPE
    ApplicationManifest.xml

    MyServiceManifest
        ServiceManifest.xml

        MyCode
            MyServiceHost.exe

        MyConfig
            Settings.xml

        MyData
            init.dat
```

The folders are named to match the **Name** attributes of each corresponding element. For example, if the service manifest contained two code packages with the names **MyCodeA** and **MyCodeB**, then two folders with the same names would contain the necessary binaries for each code package.

Use SetupEntryPoint

Typical scenarios for using **SetupEntryPoint** are when you need to run an executable before the service starts or you need to perform an operation with elevated privileges. For example:

- Setting up and initializing environment variables that the service executable needs. This is not limited to only executables written via the Service Fabric programming models. For example, npm.exe needs some environment variables configured for deploying a node.js application.
- Setting up access control by installing security certificates.

Build a package by using Visual Studio

If you use Visual Studio 2015 to create your application, you can use the Package command to automatically create a package that matches the layout described above.

To create a package, right-click the application project in Solution Explorer and choose the Package command, as shown below:

When packaging is complete, you will find the location of the package in the **Output** window. Note that the

packaging step occurs automatically when you deploy or debug your application in Visual Studio.

Test the package

You can verify the package structure locally through PowerShell by using the **Test-ServiceFabricApplicationPackage** command. This command will check for manifest parsing issues and verify all references. Note that this command only verifies the structural correctness of the directories and files in the package. It will not verify any of the code or data package contents beyond checking that all necessary files are present.

```
PS D:\temp> Test-ServiceFabricApplicationPackage .\MyApplicationType
False
Test-ServiceFabricApplicationPackage : The EntryPoint MySetup.bat is not found.
FileName:
C:\Users\servicefabric\AppData\Local\Temp\TestApplicationPackage_7195781181\nrri205a.e2h\MyApplicationType\MySe
rviceManifest\ServiceManifest.xml
```

This error shows that the *MySetup.bat* file referenced in the service manifest **SetupEntryPoint** is missing from the code package. After the missing file is added, the application verification passes:

```
PS D:\temp> tree /f .\MyApplicationType

D:\TEMP\MYAPPLICATIONTYPE
    ApplicationManifest.xml

    MyServiceManifest
        ServiceManifest.xml

        MyCode
            MyServiceHost.exe
            MySetup.bat

        MyConfig
            Settings.xml

        MyData
            init.dat

PS D:\temp> Test-ServiceFabricApplicationPackage .\MyApplicationType
True
PS D:\temp>
```

Once the application is packaged correctly and passes verification, then it's ready for deployment.

Next steps

Deploy and remove applications

Managing application parameters for multiple environments

RunAs: Running a Service Fabric application with different security permissions

Connect and communicate with services in Service Fabric

11/15/2016 • 6 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Mark Fussell • Rebecca Agiewich • Andy De George • kunaldsingh

In Service Fabric, a service runs somewhere in a Service Fabric cluster, typically distributed across multiple VMs. It can be moved from one place to another, either by the service owner, or automatically by Service Fabric. Services are not statically tied to a particular machine or address.

A Service Fabric application is generally composed of many different services, where each service performs a specialized task. These services may communicate with each other to form a complete function, such as rendering different parts of a web application. There are also client applications that connect to and communicate with services. This document discusses how to set up communication with and between your services in Service Fabric.

Bring your own protocol

Service Fabric helps manage the lifecycle of your services but it does not make decisions about what your services do. This includes communication. When your service is opened by Service Fabric, that's your service's opportunity to set up an endpoint for incoming requests, using whatever protocol or communication stack you want. Your service will listen on a normal **IP:port** address using any addressing scheme, such as a URI. Multiple service instances or replicas may share a host process, in which case they will either need to use different ports or use a port-sharing mechanism, such as the http.sys kernel driver in Windows. In either case, each service instance or replica in a host process must be uniquely addressable.

Service discovery and resolution

In a distributed system, services may move from one machine to another over time. This can happen for various reasons, including resource balancing, upgrades, failovers, or scale-out. This means service endpoint addresses change as the service moves to nodes with different IP addresses, and may open on different ports if the service uses a dynamically selected port.

Service Fabric provides a discovery and resolution service called the Naming Service. The Naming Service maintains a table that maps named service instances to the endpoint addresses they listen on. All named service instances in Service Fabric have unique names represented as URIs, for example, `"fabric:/MyApplication/MyService"`. The name of the service does not change over the lifetime of the service, it's only the endpoint addresses that can change when services move. This is analogous to websites that have constant URLs but where the IP address may change. And similar to DNS on the web, which resolves website URLs to IP addresses, Service Fabric has a registrar that maps service names to their endpoint address.

Resolving and connecting to services involves the following steps run in a loop:

- **Resolve**: Get the endpoint that a service has published from the Naming Service.
- **Connect**: Connect to the service over whatever protocol it uses on that endpoint.
- **Retry**: A connection attempt may fail for any number of reasons, for example if the service has moved since the last time the endpoint address was resolved. In that case, the preceding resolve and connect steps need to be retried, and this cycle is repeated until the connection succeeds.

Connections from external clients

Services connecting to each other inside a cluster generally can directly access the endpoints of other services because the nodes in a cluster are usually on the same local network. In some environments, however, a cluster may be behind a load balancer that routes external ingress traffic through a limited set of ports. In these cases, services can still communicate with each other and resolve addresses using the Naming Service, but extra steps must be taken to allow external clients to connect to services.

Service Fabric in Azure

A Service Fabric cluster in Azure is placed behind an Azure Load Balancer. All external traffic to the cluster must pass through the load balancer. The load balancer will automatically forward traffic inbound on a given port to a random *node* that has the same port open. The Azure Load Balancer only knows about ports open on the *nodes*, it does not know about ports open by individual *services*.

For example, in order to accept external traffic on port **80**, the following things must be configured:

1. Write a service the listens on port 80. Configure port 80 in the service's ServiceManifest.xml and open a listener in the service, for example, a self-hosted web server.

```
<Resources>
    <Endpoints>
        <Endpoint Name="WebEndpoint" Protocol="http" Port="80" />
    </Endpoints>
</Resources>
```

```
    class HttpCommunicationListener : ICommunicationListener
    {
        ...

        public Task<string> OpenAsync(CancellationToken cancellationToken)
        {
            EndpointResourceDescription endpoint =
                serviceContext.CodePackageActivationContext.GetEndpoint("WebEndpoint");

            string uriPrefix = $"{endpoint.Protocol}://+:{endpoint.Port}/myapp/";

            this.httpListener = new HttpListener();
            this.httpListener.Prefixes.Add(uriPrefix);
            this.httpListener.Start();

            string uriPublished = uriPrefix.Replace("+", FabricRuntime.GetNodeContext().IPAddressOrFQDN);

            return Task.FromResult(this.publishUri);
        }

        ...
    }

    class WebService : StatelessService
    {
        ...

        protected override IEnumerable<ServiceInstanceListener> CreateServiceInstanceListeners()
        {
            return new[] {new ServiceInstanceListener(context => new
HttpCommunicationListener(context))};
        }

        ...
    }
```

2. Create a Service Fabric Cluster in Azure and specify port **80** as a custom endpoint port for the node type that will host the service. If you have more than one node type, you can set up a *placement constraint* on the service to ensure it only runs on the node type that has the custom endpoint port opened.

3. Once the cluster has been created, configure the Azure Load Balancer in the cluster's Resource Group to forward traffic on port 80. When creating a cluster through the Azure portal, this is set up automatically for each custom endpoint port that was configured.

4. The Azure Load Balancer uses a probe to determine whether or not to send traffic to a particular node. The probe periodically checks an endpoint on each node to determine whether or not the node is responding. If the probe fails to receive a response after a configured number of times, the load balancer stops sending traffic to that node. When creating a cluster through the Azure portal, a probe is automatically set up for each custom endpoint port that was configured.

It's important to remember that the Azure Load Balancer and the probe only know about the *nodes*, not the *services* running on the nodes. The Azure Load Balancer will always send traffic to nodes that respond to the probe, so care must be taken to ensure services are available on the nodes that are able to respond to the probe.

Built-in communication API options

The Reliable Services framework ships with several pre-built communication options. The decision about which one will work best for you depends on the choice of the programming model, the communication framework, and the programming language that your services are written in.

- **No specific protocol**: If you don't have a particular choice of communication framework, but you want to get something up and running quickly, then the ideal option for you is service remoting, which allows strongly-typed remote procedure calls for Reliable Services and Reliable Actors. This is the easiest and fastest way to get started with service communication. Service remoting handles resolution of service addresses, connection, retry, and error handling. Note that service remoting is only available to C# applications.
- **HTTP**: For language-agnostic communication, HTTP provides an industry-standard choice with tools and HTTP servers available in many different langauges, all supported by Service Fabric. Services can use any HTTP stack available, including ASP.NET Web API. Clients written in C# can leverage the `ICommunicationClient` and `ServicePartitionClient` classes for service resolution, HTTP connections, and retry loops.
- **WCF**: If you have existing code that uses WCF as your communication framework, then you can use the `WcfCommunicationListener` for the server side and `WcfCommunicationClient` and `ServicePartitionClient` classes for the client. For more details, see this article about WCF-based implementation of the communication stack.

Using custom protocols and other communication frameworks

Services can use any protocol or framework for communication, whether its a custom binary protocol over TCP sockets, or streaming events through Azure Event Hubs or Azure IoT Hub. Service Fabric provides communication APIs that you can plug your communication stack into, while all the work to discover and connect is abstracted from you. See this article about the Reliable Service communication model for more details.

Next steps

Learn more about the concepts and APIs available in the Reliable Services communication model, then get started quickly with service remoting or go in-depth to learn how to write a communication listener using Web API with OWIN self-host.

Use Visual Studio to simplify writing and managing your Service Fabric applications

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Sean McKenna • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Mikkel Mork Hegnhoj • Ravikumar Palanisamy • Ryan Wike • Jesse Benson • ShawnJackson • Catherine Wang • Andy De George

You can manage your Azure Service Fabric applications and services through Visual Studio. Once you've set up your development environment, you can use Visual Studio to create Service Fabric applications, add services, or package, register, and deploy applications in your local development cluster.

Deploy your Service Fabric application

By default, deploying an application combines the following steps into one simple operation:

1. Creating the application package
2. Uploading the application package to the image store
3. Registering the application type
4. Removing any running application instances
5. Creating a new application instance

In Visual Studio, pressing **F5** will also deploy your application and attach the debugger to all application instances. You can use **Ctrl+F5** to deploy an application without debugging, or you can publish to a local or remote cluster by using the publish profile. For more information, see Publish an application to a remote cluster by using Visual Studio.

Application Debug Mode

By default, Visual Studio removes existing instances of your application type when you stop debugging or (if you deployed the app without attaching the debugger), when you redeploy the application. In that case, all the application's data is removed. While debugging locally, you may want to keep data that you've already created when testing a new version of the application, you want to keep the application running or you want subsequent debug sessions to upgrade the application. Visual Studio Service Fabric Tools provide a property called **Application Debug Mode**, which controls whether the **F5** should uninstall the application, keep the application running after a debug session ends or enable the application to be upgraded on subsequent debugging sessions, rather than removed and redeployed.

To set the Application Debug Mode property

1. On the application project's shortcut menu, choose **Properties** (or press the **F4** key).
2. In the **Properties** window, set the **Application Debug Mode** property.

These are the **Application Debug Mode** options available.

1. **Auto Upgrade**: The application continues to run when the debug session ends. The next **F5** will treat the deployment as an upgrade by using unmonitored auto mode to quickly upgrade the application to a newer version with a date string appended. The upgrade process preserves any data that you entered in a previous debug session.
2. **Keep Application**: The application keeps running in the cluster when the debug session ends. On the next **F5** the application will be removed and the newly built application will be deployed to the cluster.
3. **Remove Application** causes the application to be removed when the debug session ends.

For **Auto Upgrade** data is preserved by applying the application upgrade capabilities of Service Fabric, but it is tuned to optimize for performance rather than safety. For more information about upgrading applications and how you might perform an upgrade in a real environment, see Service Fabric application upgrade.

> **NOTE**
>
> This property doesn't exist prior to version 1.1 of the Service Fabric Tools for Visual Studio. Prior to 1.1, please use the **Preserve Data On Start** property to achieve the same behavior. The "Keep Application" option was introduced in version 1.2 of the Service Fabric Tools for Visual Studio.

Add a service to your Service Fabric application

You can add new services to your application to extend its functionality. To ensure that the service is included in your application package, add the service through the **New Fabric Service...** menu item.

Select a Service Fabric project type to add to your application, and specify a name for the service. See Choosing a framework for your service to help you decide which service type to use.

The new service will be added to your solution and existing application package. The service references and a default service instance will be added to the application manifest. The service will be created and started the next time you deploy the application.

```
ApplicationManifest.xml*
<?xml version="1.0" encoding="utf-8"?>
<ApplicationManifest ApplicationTypeName="Application56Type"
                     ApplicationTypeVersion="1.0.0.0"
                     xmlns="http://schemas.microsoft.com/2011/01/fabric"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Parameters>
    <Parameter Name="MyFrontEnd_InstanceCount" DefaultValue="1" />
    <Parameter Name="Stateless1_InstanceCount" DefaultValue="1" />
  </Parameters>
  <!-- Import the ServiceManifest from the ServicePackage. The ServiceManifestName and ServiceManifestVersion
       should match the Name and Version attributes of the ServiceManifest element defined in the
       ServiceManifest.xml file. -->
  <ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="MyFrontEndPkg" ServiceManifestVersion="1.0.0.0" />
    <ConfigOverrides />
  </ServiceManifestImport>
  <ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="Stateless1Pkg" ServiceManifestVersion="1.0.0.0" />
    <ConfigOverrides />
  </ServiceManifestImport>
  <DefaultServices>
    <!-- The section below creates instances of service types, when an instance of this
         application type is created. You can also create one or more instances of service type using the
         ServiceFabric PowerShell module.

         The attribute ServiceTypeName below must match the name defined in the imported ServiceManifest.xml file. -->
    <Service Name="MyFrontEnd">
      <StatelessService ServiceTypeName="MyFrontEndType" InstanceCount="[MyFrontEnd_InstanceCount]">
        <SingletonPartition />
      </StatelessService>
    </Service>
    <Service Name="Stateless1">
      <StatelessService ServiceTypeName="Stateless1Type" InstanceCount="[Stateless1_InstanceCount]">
        <SingletonPartition />
      </StatelessService>
    </Service>
  </DefaultServices>
</ApplicationManifest>
```

Package your Service Fabric application

To deploy the application and its services to a cluster, you need to create an application package. The package organizes the application manifest, service manifest(s), and other necessary files in a specific layout. Visual Studio sets up and manages the package in the application project's folder, in the 'pkg' directory. Clicking **Package** from the **Application** context menu creates or updates the application package. You may want to do this if you deploy the application by using custom PowerShell scripts.

Remove applications and application types using Cloud Explorer

You can perform basic cluster management operations from within Visual Studio using Cloud Explorer, which you can launch from the **View** menu. For instance, you can delete applications and unprovision application types on local or remote clusters.

> **TIP**
>
> For richer cluster management functionality, see Visualizing your cluster with Service Fabric Explorer.

Next steps

- Service Fabric application model
- Service Fabric application deployment
- Managing application parameters for multiple environments
- Debugging your Service Fabric application
- Visualizing your cluster by using Service Fabric Explorer

Debug your Service Fabric application by using Visual Studio

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Mikkel Mork Hegnhoj • Ryan Wike • Jesse Benson • ShawnJackson

Debug a local Service Fabric application

You can save time and money by deploying and debugging your Azure Service Fabric application in a local computer development cluster. Visual Studio can deploy the application to the local cluster and automatically connect the debugger to all instances of your application.

1. Start a local development cluster by following the steps in Setting up your Service Fabric development environment.
2. Press **F5** or click **Debug** > **Start Debugging**.

3. Set breakpoints in your code and step through the application by clicking commands in the **Debug** menu.

> **NOTE**
>
> Visual Studio attaches to all instances of your application. While you're stepping through code, breakpoints may get hit by multiple processes resulting in concurrent sessions. Try disabling the breakpoints after they're hit, by making each breakpoint conditional on the thread ID or by using diagnostic events.

4. The **Diagnostic Events** window will automatically open so you can view diagnostic events in real-time.

5. You can also open the **Diagnostic Events** window in Cloud Explorer. Under **Service Fabric**, right-click any node and choose **View Streaming Traces**.

If you want to filter your traces to a specific service or application, simply enable streaming traces on that specific service or application.

6. The diagnostic events can be seen in the automatically generated **ServiceEventSource.cs** file and are called from application code.

```
 ServiceEventSource.Current.ServiceMessage(this, "My ServiceMessage with a parameter {0}",
result.Value.ToString());
```

7. The **Diagnostic Events** window supports filtering, pausing, and inspecting events in real-time. The filter is a simple string search of the event message, including its contents.

8. Debugging services is like debugging any other application. You will normally set Breakpoints through Visual Studio for easy debugging. Even though Reliable Collections replicate across multiple nodes, they still implement IEnumerable. This means that you can use the Results View in Visual Studio while debugging to see what you've stored inside. Simply set a breakpoint anywhere in your code.

Debug a remote Service Fabric application

If your Service Fabric applications are running on a Service Fabric cluster in Azure, you are able to remotely debug these, directly from Visual Studio.

> **NOTE**
>
> The feature requires Service Fabric SDK 2.0 and Azure SDK for .NET 2.9.

> **WARNING**
> Remote debugging is meant for dev/test scenarios and not to be used in production environments, because of the impact on the running applications.

1. Navigate to your cluster in **Cloud Explorer**, right-click and choose **Enable Debugging**

This will kick-off the process of enabling the remote debugging extension on your cluster nodes, as well as required network configurations.

2. Right-click the cluster node in **Cloud Explorer**, and choose **Attach Debugger**

3. In the **Attach to process** dialog, choose the process you want to debug, and click **Attach**

The name of the process you want to attach to, equals the name of your service project assembly name.

The debugger will attach to all nodes running the process.

- In the case where you are debugging a stateless service, all instances of the service on all nodes are part of the debug session.
- If you are debugging a stateful service, only the primary replica of any partition will be active and therefore caught by the debugger. If the primary replica moves during the debug session, the processing of that replica will still be part of the debug session.
- In order to only catch relevant partitions or instances of a given service, you can use conditional breakpoints to only break a specific partition or instance.

> **NOTE**
>
> Currently we do not support debugging a Service Fabric cluster with multiple instances of the same service executable name.

4. Once you finish debugging your application, you can disable the remote debugging extension by right-clicking the cluster in **Cloud Explorer** and choose **Disable Debugging**

Streaming traces from a remote cluster node

You are also able to stream traces directly from a remote cluster node to Visual Studio. This feature allows you to stream ETW trace events, produced on a Service Fabric cluster node.

> **NOTE**
>
> This feature requires Service Fabric SDK 2.0 and Azure SDK for .NET 2.9. This feature only supports clusters running in Azure.

> **WARNING**
>
> Streaming traces is meant for dev/test scenarios and not to be used in production environments, because of the impact on the running applications. In a production scenario, you should rely on forwarding events using Azure Diagnostics.

1. Navigate to your cluster in **Cloud Explorer**, right-click and choose **Enable Streaming Traces**

This will kick-off the process of enabling the streaming traces extension on your cluster nodes, as well as required network configurations.

2. Expand the **Nodes** element in **Cloud Explorer**, right-click the node you want to stream traces from and choose **View Streaming Traces**

Repeat step 2 for as many nodes as you want to see traces from. Each nodes stream will show up in a dedicated window.

You are now able to see the traces emitted by Service Fabric, and your services. If you want to filter the events to only show a specific application, simply type in the name of the application in the filter.

3. Once you are done streaming traces from your cluster, you can disable remote streaming traces, by right-clicking the cluster in **Cloud Explorer** and choose **Disable Streaming Traces**

Next steps

- Test a Service Fabric service.
- Manage your Service Fabric applications in Visual Studio.

Monitor and diagnose services in a local machine development setup

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

Todd Abel • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Mark Fussell • Ryan Wike • Rebecca Agiewich • Matt Thalman • kunaldsingh • Jamie Strusz • Andy De George • Sean McKenna

Monitoring, detecting, diagnosing, and troubleshooting allow for services to continue with minimal disruption to the user experience. While monitoring and diagnostics are critical in an actual deployed production environment, the efficacy will depend on adopting a similar model during development of services to ensure they work when you move to a real-world setup. Service Fabric makes it easy for service developers to implement diagnostics that can seamlessly work across both single-machine local development setups and real-world production cluster setups.

The benefits of Event Tracing for Windows

Event Tracing for Windows (ETW) is the recommended technology for tracing messages in Service Fabric. Reasons for this are:

- **ETW is fast.** It was built as a tracing technology that has minimal impact on code execution times.
- **ETW tracing works seamlessly across local development environments and also real-world cluster setups.** This means you don't have to rewrite your tracing code when you are ready to deploy your code to a real cluster.
- **Service Fabric system code also uses ETW for internal tracing.** This allows you to view your application traces interleaved with Service Fabric system traces. It also helps you to more easily understand the sequences and interrelationships between your application code and events in the underlying system.
- **There is built-in support in Service Fabric Visual Studio tools to view ETW events.**

View Service Fabric system events in Visual Studio

Service Fabric emits ETW events to help application developers understand what's happening in the platform. If you haven't already done so, go ahead and follow the steps in Creating your first application in Visual Studio. This information will help you get an application up and running with the Diagnostics Events Viewer showing the trace messages.

1. If the diagnostics events window does not automatically show, Go to the **View** tab in Visual Studio, choose **Other Windows** and then **Diagnostic Events Viewer**.
2. Each event has standard metadata information that tells you the node, application and service the event is coming from. You can also filter the list of events by using the **Filter events** box at the top of the events window. For example, you can filter on **Node Name** or **Service Name.** And when you're looking at event details, you can also pause by using the **Pause** button at the top of the events window and resume later without any loss of events.

Add your own custom traces to the application code

The Service Fabric Visual Studio project templates contain sample code. The code shows how to add custom application code ETW traces that show up in the Visual Studio ETW viewer alongside system traces from Service Fabric. The advantage of this method is that metadata is automatically added to traces, and the Visual Studio Diagnostic Events Viewer is already configured to display them.

For projects created from the **service templates** (stateless or stateful) just search for the `RunAsync` implementation:

1. The call to `ServiceEventSource.Current.ServiceMessage` in the `RunAsync` method shows an example of a custom ETW trace from the application code.
2. In the **ServiceEventSource.cs** file, you will find an overload for the `ServiceEventSource.ServiceMessage` method that should be used for high-frequency events due to performance reasons.

For projects created from the **actor templates** (stateless or stateful):

1. Open the **"ProjectName".cs** file where *ProjectName* is the name you chose for your Visual Studio project.
2. Find the code `ActorEventSource.Current.ActorMessage(this, "Doing Work");` in the *DoWorkAsync* method. This is an example of a custom ETW trace written from application code.
3. In file **ActorEventSource.cs**, you will find an overload for the `ActorEventSource.ActorMessage` method that should be used for high-frequency events due to performance reasons.

After adding custom ETW tracing to your service code, you can build, deploy, and run the application again to see your event(s) in the Diagnostic Events Viewer. If you debug the application with **F5**, the Diagnostic Events Viewer will open automatically.

Next steps

The same tracing code that you added to your application above for local diagnostics will work with tools that you can use to view these events when running your application on an Azure cluster. Check out these articles that discuss the different options for the tools and describe how you can set them up.

- How to collect logs with Azure Diagnostics
- Using ElasticSearch as a Service Fabric application trace store

Monitor and diagnose services in a local machine development setup

11/22/2016 • 3 min to read • Edit on GitHub

Contributors

Mani Ramaswamy • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • PRmerger • carolinacmoravia

Monitoring, detecting, diagnosing, and troubleshooting allow for services to continue with minimal disruption to the user experience. Monitoring and diagnostics are critical in an actual deployed production environment. Adopting a similar model during development of services ensures that the diagnostic pipeline works when you move to a production environment. Service Fabric makes it easy for service developers to implement diagnostics that can seamlessly work across both single-machine local development setups and real-world production cluster setups.

Debugging Service Fabric Java applications

For Java applications, multiple logging frameworks are available. Since `java.util.logging` is the default option with the JRE, it is also used for the code examples in github. The following discussion explains how to configure the `java.util.logging` framework.

Using java.util.logging you can redirect your application logs to memory, output streams, console files, or sockets. For each of these options, there are default handlers already provided in the framework. You can create a `app.properties` file to configure the file handler for your application to redirect all logs to a local file.

The following code snippet contains an example configuration:

```
handlers = java.util.logging.FileHandler

java.util.logging.FileHandler.level = ALL
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.limit = 1024000
java.util.logging.FileHandler.count = 10
java.util.logging.FileHandler.pattern = /tmp/servicefabric/logs/mysfapp%u.%g.log
```

The folder pointed to by the `app.properties` file must exist. After the `app.properties` file is created, you need to also modify your entry point script, `entrypoint.sh` in the `<applicationfolder>/<servicePkg>/Code/` folder to set the property `java.util.logging.config.file` to `app.propertes` file. The entry should look like the following snippet:

```
java -Djava.library.path=$LD_LIBRARY_PATH -Djava.util.logging.config.file=<path to app.properties> -jar <service
name>.jar
```

This configuration results in logs being collected in a rotating fashion at `/tmp/servicefabric/logs/`. The **%u** and **%g** allow for creating more files, with filenames mysfapp0.log, mysfapp1.log, and so on. By default if no handler is explicitly configured, the console handler is registered. One can view the logs in syslog under /var/log/syslog.

For more information, see the code examples in github.

Debugging Service Fabric C# applications

Multiple frameworks are available for tracing CoreCLR applications on Linux. For more information, see GitHub:

logging. Since EventSource is familiar to C# developers,`this article uses EventSource for tracing in CoreCLR samples on Linux.

The first step is to include System.Diagnostics.Tracing so that you can write your logs to memory, output streams, or console files. For logging using EventSource, add the following project to your project.json:

```
    "System.Diagnostics.StackTrace": "4.0.1"
```

You can use a custom EventListener to listen for the service event and then appropriately redirect them to trace files. The following code snippet shows a sample implementation of logging using EventSource and a custom EventListener:

```
 public class ServiceEventSource : EventSource
 {
        public static ServiceEventSource Current = new ServiceEventSource();

        [NonEvent]
        public void Message(string message, params object[] args)
        {
            if (this.IsEnabled())
            {
                var finalMessage = string.Format(message, args);
                this.Message(finalMessage);
            }
        }

        // TBD: Need to add method for sample event.

}
```

```
   internal class ServiceEventListener : EventListener
   {

        protected override void OnEventSourceCreated(EventSource eventSource)
        {
            EnableEvents(eventSource, EventLevel.LogAlways, EventKeywords.All);
        }
        protected override void OnEventWritten(EventWrittenEventArgs eventData)
        {
            using (StreamWriter Out = new StreamWriter( new FileStream("/tmp/MyServiceLog.txt",
FileMode.Append)))
        {
                 // report all event information
          Out.Write(" {0} ",  Write(eventData.Task.ToString(), eventData.EventName, eventData.EventId.ToString(),
eventData.Level,""));
                if (eventData.Message != null)
            Out.WriteLine(eventData.Message, eventData.Payload.ToArray());
            else
        {
                    string[] sargs = eventData.Payload != null ? eventData.Payload.Select(o =>
o.ToString()).ToArray() : null;
                    Out.WriteLine("({0}).", sargs != null ? string.Join(", ", sargs) : "");
        }
           }
        }
    }
```

The preceding snippet outputs the logs to a file in `/tmp/MyServiceLog.txt`. This file name needs to be appropriately updated. In case you want to redirect the logs to console, use the following snippet in your customized EventListener class:

```
public static TextWriter Out = Console.Out;
```

The samples at C# Samples use EventSource and a custom EventListener to log events to a file.

Next steps

The same tracing code added to your application also works with the diagnostics of your application on an Azure cluster. Check out these articles that discuss the different options for the tools and describe how to set them up.

- How to collect logs with Azure Diagnostics

Configure security policies for your application

11/15/2016 • 11 min to read • Edit on GitHub

Contributors

Mark Fussell • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • ShawnJackson • katiecumming • Dhanashri Kshirsagar • Ryan Wike • MarkDeichmiller • pcw3187 • chrisbardon • Andy De George

By using Azure Service Fabric, you can help secure applications that are running in the cluster under different user accounts. Service Fabric also helps secure the resources that are used by applications at the time of deployment under the user accounts--for example, files, directories, and certificates. This makes running applications, even in a shared hosted environment, more secure from one another.

By default, Service Fabric applications run under the account that the Fabric.exe process runs under. Service Fabric also provides the capability to run applications under a local user account or local system account, which is specified within the application manifest. Supported local system account types are **LocalUser**, **NetworkService**, **LocalService**, and **LocalSystem**.

When you're running Service Fabric on Windows Server in your datacenter by using the standalone installer, you can use Active Directory domain accounts.

You can define and create user groups so that one or more users can be added to each group to be managed together. This is useful when there are multiple users for different service entry points and they need to have certain common privileges that are available at the group level.

Configure the policy for a service setup entry point

As described in the application model, the setup entry point, **SetupEntryPoint**, is a privileged entry point that runs with the same credentials as Service Fabric (typically the *NetworkService* account) before any other entry point. The executable that is specified by **EntryPoint** is typically the long-running service host. So having a separate setup entry point avoids having to run the service host executable with high privileges for extended periods of time. The executable that **EntryPoint** specifies is run after **SetupEntryPoint** exits successfully. The resulting process is monitored and restarted, and begins again with **SetupEntryPoint** if it ever terminates or crashes.

The following is a simple service manifest example that shows the SetupEntryPoint and the main EntryPoint for the service.

```
<?xml version="1.0" encoding="utf-8" ?>
<ServiceManifest Name="MyServiceManifest" Version="SvcManifestVersion1"
xmlns="http://schemas.microsoft.com/2011/01/fabric" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Description>An example service manifest</Description>
  <ServiceTypes>
    <StatelessServiceType ServiceTypeName="MyServiceType" />
  </ServiceTypes>
  <CodePackage Name="Code" Version="1.0.0">
    <SetupEntryPoint>
      <ExeHost>
        <Program>MySetup.bat</Program>
        <WorkingFolder>CodePackage</WorkingFolder>
      </ExeHost>
    </SetupEntryPoint>
    <EntryPoint>
      <ExeHost>
        <Program>MyServiceHost.exe</Program>
      </ExeHost>
    </EntryPoint>
  </CodePackage>
  <ConfigPackage Name="Config" Version="1.0.0" />
</ServiceManifest>
```

Configure the policy by using a local account

After you configure the service to have a setup entry point, you can change the security permissions that it runs under in the application manifest. The following example shows how to configure the service to run under user administrator account privileges.

```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationManifest xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" ApplicationTypeName="MyApplicationType" ApplicationTypeVersion="1.0.0"
xmlns="http://schemas.microsoft.com/2011/01/fabric">
   <ServiceManifestImport>
      <ServiceManifestRef ServiceManifestName="MyServiceTypePkg" ServiceManifestVersion="1.0.0" />
      <ConfigOverrides />
      <Policies>
         <RunAsPolicy CodePackageRef="Code" UserRef="SetupAdminUser" EntryPointType="Setup" />
      </Policies>
   </ServiceManifestImport>
   <Principals>
      <Users>
         <User Name="SetupAdminUser">
            <MemberOf>
               <SystemGroup Name="Administrators" />
            </MemberOf>
         </User>
      </Users>
   </Principals>
</ApplicationManifest>
```

First, create a **Principals** section with a user name, such as SetupAdminUser. This indicates that the user is a member of the Administrators system group.

Next, under the **ServiceManifestImport** section, configure a policy to apply this principal to **SetupEntryPoint**. This tells Service Fabric that when the **MySetup.bat** file is run, it should be `RunAs` with administrator privileges. Given that you have *not* applied a policy to the main entry point, the code in **MyServiceHost.exe** runs under the system **NetworkService** account. This is the default account that all service entry points are run as.

Let's now add the file MySetup.bat to the Visual Studio project to test the administrator privileges. In Visual Studio, right-click the service project and add a new file called MySetup.bat.

Next, ensure that the MySetup.bat file is included in the service package. By default, it is not. Select the file, right-

click to get the context menu, and choose **Properties**. In the Properties dialog box, ensure that **Copy to Output Directory** is set to **Copy if newer**. See the following screenshot.

Now open the MySetup.bat file and add the following commands:

```
REM Set a system environment variable. This requires administrator privilege
setx -m TestVariable "MyValue"
echo System TestVariable set to > out.txt
echo %TestVariable% >> out.txt

REM To delete this system variable us
REM REG delete "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Environment" /v TestVariable
/f
```

Next, build and deploy the solution to a local development cluster. After the service has started, as shown in Service Fabric Explorer, you can see that the MySetup.bat file was successful in a two ways. Open a PowerShell command prompt and type:

```
PS C:\ [Environment]::GetEnvironmentVariable("TestVariable","Machine")
MyValue
```

Then, note the name of the node where the service was deployed and started in Service Fabric Explorer--for example, Node 2. Next, navigate to the application instance work folder to find the out.txt file that shows the value of **TestVariable**. For example, if this service was deployed to Node 2, then you can go to this path for the **MyApplicationType**:

```
C:\SfDevCluster\Data\_App\Node.2\MyApplicationType_App\work\out.txt
```

Configure the policy by using local system accounts

Often, it's preferable to run the startup script by using a local system account rather than an administrator account. Running the RunAs policy as a member of the Administrators group typically doesn't work well because machines have User Access Control (UAC) enabled by default. In such cases, **the recommendation is to run the SetupEntryPoint as LocalSystem, instead of as a local user added to Administrators group**. The following example shows setting the SetupEntryPoint to run as LocalSystem:

```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationManifest xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" ApplicationTypeName="MyApplicationType" ApplicationTypeVersion="1.0.0"
xmlns="http://schemas.microsoft.com/2011/01/fabric">
   <ServiceManifestImport>
      <ServiceManifestRef ServiceManifestName="MyServiceTypePkg" ServiceManifestVersion="1.0.0" />
      <ConfigOverrides />
      <Policies>
         <RunAsPolicy CodePackageRef="Code" UserRef="SetupLocalSystem" EntryPointType="Setup" />
      </Policies>
   </ServiceManifestImport>
   <Principals>
      <Users>
         <User Name="SetupLocalSystem" AccountType="LocalSystem" />
      </Users>
   </Principals>
</ApplicationManifest>
```

Start PowerShell commands from a setup entry point

To run PowerShell from the **SetupEntryPoint** point, you can run **PowerShell.exe** in a batch file that points to a PowerShell file. First, add a PowerShell file to the service project--for example, **MySetup.ps1**. Remember to set the *Copy if newer* property so that the file is also included in the service package. The following example shows a sample batch file that starts a PowerShell file called MySetup.ps1, which sets a system environment variable called **TestVariable**.

MySetup.bat to start a PowerShell file:

```
powershell.exe -ExecutionPolicy Bypass -Command ".\MySetup.ps1"
```

In the PowerShell file, add the following to set a system environment variable:

```
[Environment]::SetEnvironmentVariable("TestVariable", "MyValue", "Machine")
[Environment]::GetEnvironmentVariable("TestVariable","Machine") > out.txt
```

> **NOTE**
> By default, when the batch file runs, it looks at the application folder called **work** for files. In this case, when MySetup.bat runs, we want this to find the MySetup.ps1 file in the same folder, which is the application **code package** folder. To change this folder, set the working folder:

```
<SetupEntryPoint>
    <ExeHost>
    <Program>MySetup.bat</Program>
    <WorkingFolder>CodePackage</WorkingFolder>
    </ExeHost>
</SetupEntryPoint>
```

Use console redirection for local debugging

Occasionally, it's useful to see the console output from running a script for debugging purposes. To do this, you can set a console redirection policy, which writes the output to a file. The file output is written to the application folder called **log** on the node where the application is deployed and run. (See where to find this in the preceding example.)

> **NOTE**
> Never use the console redirection policy in an application that is deployed in production because this can affect the application failover. *Only* use this for local development and debugging purposes.

The following example shows setting the console redirection with a FileRetentionCount value:

```
<SetupEntryPoint>
    <ExeHost>
    <Program>MySetup.bat</Program>
    <WorkingFolder>CodePackage</WorkingFolder>
    <ConsoleRedirection FileRetentionCount="10"/>
    </ExeHost>
</SetupEntryPoint>
```

If you now change the MySetup.ps1 file to write an **Echo** command, this will write to the output file for debugging purposes:

```
Echo "Test console redirection which writes to the application log folder on the node that the application is deployed to"
```

After you debug your script, immediately remove this console redirection policy.

Configure a policy for service code packages

In the preceding steps, you saw how to apply a RunAs policy to SetupEntryPoint. Let's look a little deeper into how to create different principals that can be applied as service policies.

Create local user groups

You can define and create user groups that allow one or more users to be added to a group. This is particularly useful if there are multiple users for different service entry points and they need to have certain common privileges that are available at the group level. The following example shows a local group called **LocalAdminGroup** that has administrator privileges. Two users, Customer1 and Customer2, are made members of this local group.

```
<Principals>
 <Groups>
   <Group Name="LocalAdminGroup">
     <Membership>
       <SystemGroup Name="Administrators"/>
     </Membership>
   </Group>
 </Groups>
  <Users>
     <User Name="Customer1">
        <MemberOf>
           <Group NameRef="LocalAdminGroup" />
        </MemberOf>
     </User>
    <User Name="Customer2">
      <MemberOf>
        <Group NameRef="LocalAdminGroup" />
      </MemberOf>
    </User>
  </Users>
</Principals>
```

Create local users

You can create a local user that can be used to help secure a service within the application. When a **LocalUser**

account type is specified in the principals section of the application manifest, Service Fabric creates local user accounts on machines where the application is deployed. By default, these accounts do not have the same names as those specified in the application manifest (for example, Customer3 in the following sample). Instead, they are dynamically generated and have random passwords.

```
<Principals>
  <Users>
     <User Name="Customer3" AccountType="LocalUser" />
  </Users>
</Principals>
```

Assign policies to the service code packages

The **RunAsPolicy** section for a **ServiceManifestImport** specifies the account from the principals section that should be used to run a code package. It also associates code packages from the service manifest with user accounts in the principals section. You can specify this for the setup or main entry points, or you can specify `All` to apply it to both. The following example shows different policies being applied:

```
<Policies>
<RunAsPolicy CodePackageRef="Code" UserRef="LocalAdmin" EntryPointType="Setup"/>
<RunAsPolicy CodePackageRef="Code" UserRef="Customer3" EntryPointType="Main"/>
</Policies>
```

If **EntryPointType** is not specified, the default is set to EntryPointType="Main". Specifying **SetupEntryPoint** is especially useful when you want to run certain high-privilege setup operations under a system account. The actual service code can run under a lower-privilege account.

Apply a default policy to all service code packages

You use the **DefaultRunAsPolicy** section to specify a default user account for all code packages that don't have a specific **RunAsPolicy** defined. If most of the code packages that are specified in the service manifest used by an application need to run under the same user, the application can just define a default RunAs policy with that user account. The following example specifies that if a code package does not have a **RunAsPolicy** specified, the code package should run under the **MyDefaultAccount** specified in the principals section.

```
<Policies>
  <DefaultRunAsPolicy UserRef="MyDefaultAccount"/>
</Policies>
```

Use an Active Directory domain group or user

For an instance of Service Fabric that was installed on Windows Server by using the standalone installer, you can run the service under the credentials for an Active Directory user or group account. Note that this is Active Directory on-premises within your domain and is not with Azure Active Directory (Azure AD). By using a domain user or group, you can then access other resources in the domain (for example, file shares) that have been granted permissions.

The following example shows an Active Directory user called *TestUser* with their domain password encrypted by using a certificate called *MyCert*. You can use the `Invoke-ServiceFabricEncryptText` PowerShell command to create the secret cipher text. See Managing secrets in Service Fabric applications for details.

You must deploy the private key of the certificate to decrypt the password to the local machine by using an out-of-band method (in Azure, this is via Azure Resource Manager). Then, when Service Fabric deploys the service package to the machine, it is able to decrypt the secret and (along with the user name) authenticate with Active Directory to run under those credentials.

```
<Principals>
  <Users>
    <User Name="TestUser" AccountType="DomainUser" AccountName="Domain\User" Password="[Put encrypted password
here using MyCert certificate]" PasswordEncrypted="true" />
  </Users>
</Principals>
<Policies>
  <DefaultRunAsPolicy UserRef="TestUser" />
  <SecurityAccessPolicies>
    <SecurityAccessPolicy ResourceRef="MyCert" PrincipalRef="TestUser" GrantRights="Full"
ResourceType="Certificate" />
  </SecurityAccessPolicies>
</Policies>
<Certificates>
```

Assign a security access policy for HTTP and HTTPS endpoints

If you apply a RunAs policy to a service and the service manifest declares endpoint resources with the HTTP protocol, you must specify a **SecurityAccessPolicy** to ensure that ports allocated to these endpoints are correctly access-control listed for the RunAs user account that the service runs under. Otherwise, **http.sys** does not have access to the service, and you get failures with calls from the client. The following example applies the Customer3 account to an endpoint called **ServiceEndpointName**, which gives it full access rights.

```
<Policies>
   <RunAsPolicy CodePackageRef="Code" UserRef="Customer1" />
   <!--SecurityAccessPolicy is needed if RunAsPolicy is defined and the Endpoint is http -->
   <SecurityAccessPolicy ResourceRef="EndpointName" PrincipalRef="Customer1" />
</Policies>
```

For the HTTPS endpoint, you also have to indicate the name of the certificate to return to the client. You can do this by using **EndpointBindingPolicy**, with the certificate defined in a certificates section in the application manifest.

```
<Policies>
   <RunAsPolicy CodePackageRef="Code" UserRef="Customer1" />
  <!--SecurityAccessPolicy is needed if RunAsPolicy is defined and the Endpoint is http -->
   <SecurityAccessPolicy ResourceRef="EndpointName" PrincipalRef="Customer1" />
  <!--EndpointBindingPolicy is needed if the EndpointName is secured with https -->
  <EndpointBindingPolicy EndpointRef="EndpointName" CertificateRef="Cert1" />
</Policies
```

A complete application manifest example

The following application manifest shows many of the different settings:

```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationManifest xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" ApplicationTypeName="Application3Type" ApplicationTypeVersion="1.0.0"
xmlns="http://schemas.microsoft.com/2011/01/fabric">
   <Parameters>
      <Parameter Name="Stateless1_InstanceCount" DefaultValue="-1" />
   </Parameters>
   <ServiceManifestImport>
      <ServiceManifestRef ServiceManifestName="Stateless1Pkg" ServiceManifestVersion="1.0.0" />
      <ConfigOverrides />
      <Policies>
         <RunAsPolicy CodePackageRef="Code" UserRef="Customer1" />
         <RunAsPolicy CodePackageRef="Code" UserRef="LocalAdmin" EntryPointType="Setup" />
        <!--SecurityAccessPolicy is needed if RunAsPolicy is defined and the Endpoint is http -->
         <SecurityAccessPolicy ResourceRef="EndpointName" PrincipalRef="Customer1" />
        <!--EndpointBindingPolicy is needed the EndpointName is secured with https -->
        <EndpointBindingPolicy EndpointRef="EndpointName" CertificateRef="Cert1" />
     </Policies>
   </ServiceManifestImport>
   <DefaultServices>
      <Service Name="Stateless1">
         <StatelessService ServiceTypeName="Stateless1Type" InstanceCount="[Stateless1_InstanceCount]">
            <SingletonPartition />
         </StatelessService>
      </Service>
   </DefaultServices>
   <Principals>
      <Groups>
         <Group Name="LocalAdminGroup">
            <Membership>
               <SystemGroup Name="Administrators" />
            </Membership>
         </Group>
      </Groups>
      <Users>
         <User Name="LocalAdmin">
            <MemberOf>
               <Group NameRef="LocalAdminGroup" />
            </MemberOf>
         </User>
         <!--Customer1 below create a local account that this service runs under -->
         <User Name="Customer1" />
         <User Name="MyDefaultAccount" AccountType="NetworkService" />
      </Users>
   </Principals>
   <Policies>
      <DefaultRunAsPolicy UserRef="LocalAdmin" />
   </Policies>
   <Certificates>
     <EndpointCertificate Name="Cert1" X509FindValue="FF EE E0 TT JJ DD JJ EE EE XX 23 4T 66 "/>
  </Certificates>
</ApplicationManifest>
```

Next steps

- Understand the application model
- Specify resources in a service manifest
- Deploy an application

Manage application parameters for multiple environments

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Sean McKenna • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Wike • katiecumming • Andy Pasic • Andy De George

You can create Azure Service Fabric clusters by using anywhere from one to many thousands of machines. While application binaries can run without modification across this wide spectrum of environments, you will often want to configure the application differently, depending on the number of machines you're deploying to.

As a simple example, consider `InstanceCount` for a stateless service. When you are running applications in Azure, you will generally want to set this parameter to the special value of "-1". This ensures that your service is running on every node in the cluster (or every node in the node type if you have set a placement constraint). However, this configuration is not suitable for a single-machine cluster since you can't have multiple processes listening on the same endpoint on a single machine. Instead, you will typically set `InstanceCount` to "1".

Specifying environment-specific parameters

The solution to this configuration issue is a set of parameterized default services and application parameter files that fill in those parameter values for a given environment. Default services and application parameters are configured in the application and service manifests. The schema definition for the ServiceManifest.xml and ApplicationManifest.xml files is installed with the Service Fabric SDK and tools to *C:\Program Files\Microsoft SDKs\Service Fabric\schemas\ServiceFabricServiceModel.xsd*.

Default services

Service Fabric applications are made up of a collection of service instances. While it is possible for you to create an empty application and then create all service instances dynamically, most applications have a set of core services that should always be created when the application is instantiated. These are referred to as "default services". They are specified in the application manifest, with placeholders for per-environment configuration included in square brackets:

```
<DefaultServices>
    <Service Name="Stateful1">
        <StatefulService
            ServiceTypeName="Stateful1Type"
            TargetReplicaSetSize="[Stateful1_TargetReplicaSetSize]"
            MinReplicaSetSize="[Stateful1_MinReplicaSetSize]">

            <UniformInt64Partition
                PartitionCount="[Stateful1_PartitionCount]"
                LowKey="-9223372036854775808"
                HighKey="9223372036854775807"
            />
    </StatefulService>
</Service>
```

Each of the named parameters must be defined within the Parameters element of the application manifest:

```
<Parameters>
    <Parameter Name="Stateful1_MinReplicaSetSize" DefaultValue="2" />
    <Parameter Name="Stateful1_PartitionCount" DefaultValue="1" />
    <Parameter Name="Stateful1_TargetReplicaSetSize" DefaultValue="3" />
</Parameters>
```

The DefaultValue attribute specifies the value to be used in the absence of a more-specific parameter for a given environment.

> **NOTE**
>
> Not all service instance parameters are suitable for per-environment configuration. In the example above, the LowKey and HighKey values for the service's partitioning scheme are explicitly defined for all instances of the service since the partition range is a function of the data domain, not the environment.

Per-environment service configuration settings

The Service Fabric application model enables services to include configuration packages that contain custom key-value pairs that are readable at run time. The values of these settings can also be differentiated by environment by specifying a `ConfigOverride` in the application manifest.

Suppose that you have the following setting in the Config\Settings.xml file for the `Stateful1` service:

```
<Section Name="MyConfigSection">
  <Parameter Name="MaxQueueSize" Value="25" />
</Section>
```

To override this value for a specific application/environment pair, create a `ConfigOverride` when you import the service manifest in the application manifest.

```
<ConfigOverrides>
 <ConfigOverride Name="Config">
    <Settings>
       <Section Name="MyConfigSection">
          <Parameter Name="MaxQueueSize" Value="[Stateful1_MaxQueueSize]" />
       </Section>
    </Settings>
 </ConfigOverride>
```

This parameter can then be configured by environment as shown above. You can do this by declaring it in the parameters section of the application manifest and specifying environment-specific values in the application parameter files.

> **NOTE**
>
> In the case of service configuration settings, there are three places where the value of a key can be set: the service configuration package, the application manifest, and the application parameter file. Service Fabric will always choose from the application parameter file first (if specified), then the application manifest, and finally the configuration package.

Application parameter files

The Service Fabric application project can include one or more application parameter files. Each of them defines the specific values for the parameters that are defined in the application manifest:

```
<!-- ApplicationParameters\Local.xml -->

<Application Name="fabric:/Application1" xmlns="http://schemas.microsoft.com/2011/01/fabric">
    <Parameters>
        <Parameter Name ="Stateful1_MinReplicaSetSize" Value="2" />
        <Parameter Name="Stateful1_PartitionCount" Value="1" />
        <Parameter Name="Stateful1_TargetReplicaSetSize" Value="3" />
    </Parameters>
</Application>
```

By default, a new application includes three application parameter files, named Local.1Node.xml, Local.5Node.xml, and Cloud.xml:

To create a new parameter file, simply copy and paste an existing one and give it a new name.

Identifying environment-specific parameters during deployment

At deployment time, you need to choose the appropriate parameter file to apply with your application. You can do this through the Publish dialog in Visual Studio or through PowerShell.

Deploy from Visual Studio

You can choose from the list of available parameter files when you publish your application in Visual Studio.

Deploy from PowerShell

The `Deploy-FabricApplication.ps1` PowerShell script included in the application project template accepts a publish

profile as a parameter and the PublishProfile contains a reference to the application parameters file.

```
    ./Deploy-FabricApplication -ApplicationPackagePath <app_package_path> -PublishProfileFile
<publishprofile_path>
```

Next steps

To learn more about some of the core concepts that are discussed in this topic, see the Service Fabric technical overview. For information about other app management capabilities that are available in Visual Studio, see Manage your Service Fabric applications in Visual Studio.

Reliable Services overview

11/15/2016 • 9 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Vaclav Turecek • Sean McKenna • MarkDeichmiller • Jesse Benson • Mani Ramaswamy

Azure Service Fabric simplifies writing and managing stateless and stateful Reliable Services. This document will talk about:

- The Reliable Services programming model for stateless and stateful services.
- The choices you have to make when writing a Reliable Service.
- Some scenarios and examples of when to use Reliable Services and how they are written.

Reliable Services is one of the programming models available on Service Fabric. For more information on the Reliable Actors programming model, see Introduction to Service Fabric Reliable Actors.

In Service Fabric, a service is composed of configuration, application code, and (optionally) state.

Service Fabric manages the lifetime of services, from provisioning and deployment through upgrade and deletion, via Service Fabric application management.

What are Reliable Services?

Reliable Services gives you a simple, powerful, top-level programming model to help you express what is important to your application. With the Reliable Services programming model, you get:

- For stateful services, the Reliable Services programming model allows you to consistently and reliably store your state right inside your service by using Reliable Collections. This is a simple set of highly available collection classes that will be familiar to anyone who has used C# collections. Traditionally, services needed external systems for Reliable state management. With Reliable Collections, you can store your state next to your compute with the same high availability and reliability you've come to expect from highly available external stores, and with the additional latency improvements that co-locating the compute and state provide.
- A simple model for running your own code that looks like programming models you are used to. Your code has a well-defined entry point and easily managed lifecycle.
- A pluggable communication model. Use the transport of your choice, such as HTTP with Web API, WebSockets, custom TCP protocols, etc. Reliable Services provide some great out-of-the-box options you can use, or you can provide your own.

What makes Reliable Services different?

Reliable Services in Service Fabric is different from services you may have written before. Service Fabric provides reliability, availability, consistency, and scalability.

- **Reliability**--Your service will stay up even in unreliable environments where your machines may fail or hit network issues.
- **Availability**--Your service will be reachable and responsive. (This doesn't mean that you can't have services that can't be found or reached from outside.)
- **Scalability**--Services are decoupled from specific hardware, and they can grow or shrink as necessary through the addition or removal of hardware or virtual resources. Services are easily partitioned (especially in

the stateful case) to ensure that independent portions of the service can scale and respond to failures independently. Finally, Service Fabric encourages services to be lightweight by allowing thousands of services to be provisioned within a single process, rather than requiring or dedicating entire OS instances to a single instance of a particular workload.

- **Consistency**--Any information stored in this service can be guaranteed to be consistent (this applies only to stateful services - more on this later)

Service lifecycle

Whether your service is stateful or stateless, Reliable Services provide a simple lifecycle that lets you quickly plug in your code and get started. There are just one or two methods that you need to implement to get your service up and running.

- **CreateServiceReplicaListeners/CreateServiceInstanceListeners** - This is where the service defines the communication stack that it wants to use. The communication stack, such as Web API, is what defines the listening endpoint or endpoints for the service (how clients will reach it). It also defines how the messages that appear end up interacting with the rest of the service code.
- **RunAsync** - This is where your service runs its business logic. The cancellation token that is provided is a signal for when that work should stop. For example, if you have a service that needs to constantly pull messages out of a Reliable Queue and process them, this is where that work would happen.

Service startup

The major events in the lifecycle of a Reliable Service are:

1. The service object (the thing that derives from the stateless service or stateful service) is constructed.
2. The `CreateServiceReplicaListeners` / `CreateServiceInstanceListeners` method is called, giving the service a chance to return one or more communication listeners of its choice.
   - Note that this is optional, although most services will expose some endpoint directly.
3. Once the communication listeners are created, it is opened.
   - Communication listeners have a method called `OpenAsync()`, which is called at this point and which returns the listening address for the service. If your Reliable Service uses one of the built-in ICommunicationListeners, this is handled for you.
4. Once the communication listener is open, the `RunAsync()` method on the main service is called.
   - Note that `RunAsync()` is optional. If the service does all its work directly in response to user calls only, there is no need for it to implement `RunAsync()`.

Service shutdown

When the service is being shut down (to be deleted, upgraded, or moved) the call order is mirrored: First, the cancellation token held by `RunAsync()` is canceled; then `CloseAsync()` is called on the communication listeners.

There are a few important things to note about shutdown for stateful services:

- Service Fabric will not promote another replica of your service to Primary status until `CloseAsync` and `RunAsync` have returned. If you are using a built-in communication listener, the `CloseAsync` method is handled for you.
- While there is no time limit on returning from these methods, you immediately lose the ability to write to Reliable Collections and therefore cannot complete any real work. It is recommended that you return as quickly as possible upon receiving the cancellation request.

Example services

Knowing this programming model, let's take a quick look at two different services to see how these pieces fit

together.

Stateless Reliable Services

A stateless service is one where there is literally no state maintained within the service, or the state that is present is entirely disposable and doesn't require synchronization, replication, persistence, or high availability.

For example, consider a calculator that has no memory and receives all terms and operations to perform at once.

In this case, the RunAsync() of the service can be empty, since there is no background task-processing that the service needs to do. When the calculator service is created, it will return an ICommunicationListener (for example Web API) that opens up a listening endpoint on some port. This listening endpoint will hook up to the different methods (example: "Add(n1, n2)") that define the calculator's public API.

When a call is made from a client, the appropriate method is invoked, and the calculator service performs the operations on the data provided and returns the result. It doesn't store any state.

Not storing any internal state makes this example calculator very simple. But most services aren't truly stateless. Instead, they externalize their state to some other store. (For example, any web app that relies on keeping session state in a backing store or cache is not completely stateless.)

A common example of how stateless services are used in Service Fabric is as a front-end that exposes the public-facing API for a web application. The front-end service then talks to stateful services to complete a user request. In this case, calls from clients are directed to a known port, such as 80, where the stateless service is listening. This stateless service receives the call and determines whether the call is from a trusted party, as well as which service it's destined for. Then, the stateless service forwards the call to the correct partition of the stateful service and waits for a response. When the stateless service receives a response, it replies back to the original client.

Stateful Reliable Services

A stateful service is one that must have some portion of state kept consistent and present in order for the service to function. Consider a service that constantly computes a rolling average of some value based on updates it receives. To do this, it must have the current set of incoming requests it needs to process, as well as the current average. Any service that retrieves, processes, and stores information in an external store (such as an Azure blob or table store today) is stateful. It just keeps its state in the external state store.

Most services today store their state externally, since the external store is what provides reliability, availability, scalability, and consistency for that state. In Service Fabric, stateful services aren't required to store their state externally; Service Fabric takes care of these requirements for both the service code and the service state.

Let's say we want to write a service that takes requests for a series of conversions that need to be performed on an image, and the image that needs to be converted. For this service, it would return a communication listener (let's suppose Web API) that opens up a communication port and allows submissions via an API like `ConvertImage(Image i, IList<Conversion> conversions)`. In this API, the service could take the information and store the request in a Reliable Queue, and then return some token to the client so it could keep track of the request (since the requests could take some time).

In this service, RunAsync could be more complex. The service could have a loop inside its RunAsync that pulls requests out of IReliableQueue, performs the conversions listed, and stores the results in IReliableDictionary, so that when the client comes back, they can get their converted images. To ensure that even if something fails the image isn't lost, this Reliable Service would pull out of the queue, perform the conversions, and store the result in a transaction. In this case, the message is actually removed only from the queue and the results are stored in the result dictionary when the conversions are complete. If something fails in the middle (such as the machine this instance of the code is running on), the request remains in the queue waiting to be processed again.

One thing to note about this service is that it sounds like a normal .NET service. The only difference is that the data structures being used (IReliableQueue and IReliableDictionary) are provided by Service Fabric, and hence are made highly reliable, available, and consistent.

When to use Reliable Services APIs

If any of the following characterize your application service needs, then you should consider Reliable Services APIs:

- You need to provide application behavior across multiple units of state (e.g., orders and order line items).
- Your application's state can be naturally modeled as Reliable Dictionaries and Queues.
- Your state needs to be highly available with low latency access.
- Your application needs to control the concurrency or granularity of transacted operations across one or more Reliable Collections.
- You want to manage the communications or control the partitioning scheme for your service.
- Your code needs a free-threaded runtime environment.
- Your application needs to dynamically create or destroy Reliable Dictionaries or Queues at runtime.
- You need to programmatically control Service Fabric-provided backup and restore features for your service's state*.
- Your application needs to maintain change history for its units of state*.
- You want to develop or consume third-party-developed, custom state providers*.

> **NOTE**
>
> *Features available at SDK general availability.

Next steps

- Reliable Services quick start
- Reliable Services advanced usage
- The Reliable Actors programming model

Get started with Reliable Services

11/15/2016 • 9 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Wike • MarkDeichmiller • v-aljenk • Cesar Mello • Jesse Benson • Sean McKenna • Andy De George

An Azure Service Fabric application contains one or more services that run your code. This guide shows you how to create both stateless and stateful Service Fabric applications with Reliable Services.

Basic concepts

To get started with Reliable Services, you only need to understand a few basic concepts:

- **Service type**: This is your service implementation. It is defined by the class you write that extends `StatelessService` and any other code or dependencies used therein, along with a name and a version number.
- **Named service instance**: To run your service, you create named instances of your service type, much like you create object instances of a class type. Service instances are in fact object instantiations of your service class that you write.
- **Service host**: The named service instances you create need to run inside a host. The service host is just a process where instances of your service can run.
- **Service registration**: Registration brings everything together. The service type must be registered with the Service Fabric runtime in a service host to allow Service Fabric to create instances of it to run.

Create a stateless service

A stateless service is a type of service that is currently the norm in cloud applications. It is considered stateless because the service itself does not contain data that needs to be stored reliably or made highly available. If an instance of a stateless service shuts down, all of its internal state is lost. In this type of service, state must be persisted to an external store, such as Azure Tables or a SQL database, for it to be made highly available and reliable.

Launch Visual Studio 2015 as an administrator, and create a new Service Fabric application project named *HelloWorld*:

Then create a stateless service project named *HelloWorldStateless*:

Your solution now contains two projects:

- *HelloWorld*. This is the *application* project that contains your *services*. It also contains the application manifest that describes the application, as well as a number of PowerShell scripts that help you to deploy your application.

- *HelloWorldStateless*. This is the service project. It contains the stateless service implementation.

Implement the service

Open the **HelloWorldStateless.cs** file in the service project. In Service Fabric, a service can run any business logic. The service API provides two entry points for your code:

- An open-ended entry point method, called *RunAsync*, where you can begin executing any workloads, including long-running compute workloads.

```
protected override async Task RunAsync(CancellationToken cancellationToken)
{
    ...
}
```

- A communication entry point where you can plug in your communication stack of choice, such as ASP.NET Web API. This is where you can start receiving requests from users and other services.

```
protected override IEnumerable<ServiceInstanceListener> CreateServiceInstanceListeners()
{
    ...
}
```

In this tutorial, we will focus on the `RunAsync()` entry point method. This is where you can immediately start running your code. The project template includes a sample implementation of `RunAsync()` that increments a rolling count.

> **NOTE**
> For details about how to work with a communication stack, see Service Fabric Web API services with OWIN self-hosting

RunAsync

```
protected override async Task RunAsync(CancellationToken cancellationToken)
{
    // TODO: Replace the following sample code with your own logic
    //       or remove this RunAsync override if it's not needed in your service.

    long iterations = 0;

    while (true)
    {
        cancellationToken.ThrowIfCancellationRequested();

        ServiceEventSource.Current.ServiceMessage(this, "Working-{0}", ++iterations);

        await Task.Delay(TimeSpan.FromSeconds(1), cancellationToken);
    }
}
```

The platform calls this method when an instance of a service is placed and ready to execute. For a stateless service, that simply means when the service instance is opened. A cancellation token is provided to coordinate when your service instance needs to be closed. In Service Fabric, this open/close cycle of a service instance can occur many times over the lifetime of the service as a whole. This can happen for various reasons, including:

- The system moves your service instances for resource balancing.
- Faults occur in your code.
- The application or system is upgraded.

- The underlying hardware experiences an outage.

This orchestration is managed by the system to keep your service highly available and properly balanced.

`RunAsync()` should not block synchronously. Your implementation of RunAsync should return a Task or await on any long-running or blocking operations to allow the runtime to continue - note in the `while(true)` loop in the previous example, a Task-returning `await Task.Delay()` is used. If your workload must block synchronously, you should schedule a new Task with `Task.Run()` in your `RunAsync` implementation.

Cancellation of your workload is a cooperative effort orchestrated by the provided cancellation token. The system will wait for your task to end (by successful completion, cancellation, or fault) before it moves on. It is important to honor the cancellation token, finish any work, and exit `RunAsync()` as quickly as possible when the system requests cancellation.

In this stateless service example, the count is stored in a local variable. But because this is a stateless service, the value that's stored exists only for the current lifecycle of its service instance. When the service moves or restarts, the value is lost.

Create a stateful service

Service Fabric introduces a new kind of service that is stateful. A stateful service can maintain state reliably within the service itself, co-located with the code that's using it. State is made highly available by Service Fabric without the need to persist state to an external store.

To convert a counter value from stateless to highly available and persistent, even when the service moves or restarts, you need a stateful service.

In the same *HelloWorld* application, you can add a new service by right-clicking on the Services references in the application project and selecting **Add -> New Service Fabric Service**.

Select **Stateful Service** and name it *HelloWorldStateful*. Click **OK**.

Your application should now have two services: the stateless service *HelloWorldStateless* and the stateful service *HelloWorldStateful*.

A stateful service has the same entry points as a stateless service. The main difference is the availability of a *state provider* that can store state reliably. Service Fabric comes with a state provider implementation called Reliable Collections, which lets you create replicated data structures through the Reliable State Manager. A stateful Reliable Service uses this state provider by default.

Open **HelloWorldStateful.cs** in *HelloWorldStateful*, which contains the following RunAsync method:

```
protected override async Task RunAsync(CancellationToken cancellationToken)
{
    // TODO: Replace the following sample code with your own logic
    //       or remove this RunAsync override if it's not needed in your service.

    var myDictionary = await this.StateManager.GetOrAddAsync<IReliableDictionary<string, long>>
("myDictionary");

    while (true)
    {
        cancellationToken.ThrowIfCancellationRequested();

        using (var tx = this.StateManager.CreateTransaction())
        {
            var result = await myDictionary.TryGetValueAsync(tx, "Counter");

            ServiceEventSource.Current.ServiceMessage(this, "Current Counter Value: {0}",
                result.HasValue ? result.Value.ToString() : "Value does not exist.");

            await myDictionary.AddOrUpdateAsync(tx, "Counter", 0, (key, value) => ++value);

            // If an exception is thrown before calling CommitAsync, the transaction aborts, all changes are
            // discarded, and nothing is saved to the secondary replicas.
            await tx.CommitAsync();
        }

        await Task.Delay(TimeSpan.FromSeconds(1), cancellationToken);
    }
```

RunAsync

`RunAsync()` operates similarly in stateful and stateless services. However, in a stateful service, the platform performs additional work on your behalf before it executes `RunAsync()`. This work can include ensuring that the Reliable State Manager and Reliable Collections are ready to use.

Reliable Collections and the Reliable State Manager

```
var myDictionary = await this.StateManager.GetOrAddAsync<IReliableDictionary<string, long>>("myDictionary");
```

IReliableDictionary is a dictionary implementation that you can use to reliably store state in the service. With Service Fabric and Reliable Collections, you can store data directly in your service without the need for an external persistent store. Reliable Collections make your data highly available. Service Fabric accomplishes this by creating and managing multiple *replicas* of your service for you. It also provides an API that abstracts away the complexities of managing those replicas and their state transitions.

Reliable Collections can store any .NET type, including your custom types, with a couple of caveats:

- Service Fabric makes your state highly available by *replicating* state across nodes, and Reliable Collections store your data to local disk on each replica. This means that everything that is stored in Reliable Collections must be *serializable*. By default, Reliable Collections use DataContract for serialization, so it's important to make sure that your types are supported by the Data Contract Serializer when you use the default serializer.
- Objects are replicated for high availability when you commit transactions on Reliable Collections. Objects stored in Reliable Collections are kept in local memory in your service. This means that you have a local reference to the object.

  It is important that you do not mutate local instances of those objects without performing an update operation on the reliable collection in a transaction. This is because changes to local instances of objects will not be replicated automatically. You must re-insert the object back into the dictionary or use one of the *update* methods on the dictionary.

The Reliable State Manager manages Reliable Collections for you. You can simply ask the Reliable State Manager

for a reliable collection by name at any time and at any place in your service. The Reliable State Manager ensures that you get a reference back. We don't recommended that you save references to reliable collection instances in class member variables or properties. Special care must be taken to ensure that the reference is set to an instance at all times in the service lifecycle. The Reliable State Manager handles this work for you, and it's optimized for repeat visits.

Transactional and asynchronous operations

```
using (ITransaction tx = this.StateManager.CreateTransaction())
{
    var result = await myDictionary.TryGetValueAsync(tx, "Counter-1");

    await myDictionary.AddOrUpdateAsync(tx, "Counter-1", 0, (k, v) => ++v);

    await tx.CommitAsync();
}
```

Reliable Collections have many of the same operations that their `System.Collections.Generic` and `System.Collections.Concurrent` counterparts do, except LINQ. Operations on Reliable Collections are asynchronous. This is because write operations with Reliable Collections perform I/O operations to replicate and persist data to disk.

Reliable Collection operations are *transactional*, so that you can keep state consistent across multiple Reliable Collections and operations. For example, you may dequeue a work item from a Reliable Queue, perform an operation on it, and save the result in a Reliable Dictionary, all within a single transaction. This is treated as an atomic operation, and it guarantees that either the entire operation will succeed or the entire operation will roll back. If an error occurs after you dequeue the item but before you save the result, the entire transaction is rolled back and the item remains in the queue for processing.

Run the application

We now return to the *HelloWorld* application. You can now build and deploy your services. When you press **F5**, your application will be built and deployed to your local cluster.

After the services start running, you can view the generated Event Tracing for Windows (ETW) events in a **Diagnostic Events** window. Note that the events displayed are from both the stateless service and the stateful service in the application. You can pause the stream by clicking the **Pause** button. You can then examine the details of a message by expanding that message.

> **NOTE**
> Before you run the application, make sure that you have a local development cluster running. Check out the getting started guide for information on setting up your local environment.

Next steps

Debug your Service Fabric application in Visual Studio

Get started: Service Fabric Web API services with OWIN self-hosting

Learn more about Reliable Collections

Deploy an application

Application upgrade

Developer reference for Reliable Services

Get started with Reliable Services

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

This article explains the basics of Azure Service Fabric Reliable Services and walks you through creating and deploying a simple Reliable Service application written in Java.

Installation and setup

Before you start, make sure you have the Service Fabric development environment set up on your machine. If you need to set it up, go to getting started on Mac or getting started on Linux.

Basic concepts

To get started with Reliable Services, you only need to understand a few basic concepts:

- **Service type**: This is your service implementation. It is defined by the class you write that extends `StatelessService` and any other code or dependencies used therein, along with a name and a version number.
- **Named service instance**: To run your service, you create named instances of your service type, much like you create object instances of a class type. Service instances are in fact object instantiations of your service class that you write.
- **Service host**: The named service instances you create need to run inside a host. The service host is just a process where instances of your service can run.
- **Service registration**: Registration brings everything together. The service type must be registered with the Service Fabric runtime in a service host to allow Service Fabric to create instances of it to run.

Create a stateless service

Start by creating a new Service Fabric application. The Service Fabric SDK for Linux includes a Yeoman generator to provide the scaffolding for a Service Fabric application with a stateless service. Start by running the following Yeoman command:

```
$ yo azuresfjava
```

Follow the instructions to create a **Reliable Stateless Service**. For this tutorial, name the application "HelloWorldApplication" and the service "HelloWorld". The result will include directories for the `HelloWorldApplication` and `HelloWorld`.

```
HelloWorldApplication/
□□□ build.gradle
□□□ HelloWorld
□   □□□ build.gradle
□   □□□ src
□       □□□ statelessservice
□           □□□ HelloWorldServiceHost.java
□           □□□ HelloWorldService.java
□□□ HelloWorldApplication
□   □□□ ApplicationManifest.xml
□   □□□ HelloWorldPkg
□       □□□ Code
□       □   □□□ entryPoint.sh
□       □   □□□ _readme.txt
□       □□□ Config
□       □   □□□ _readme.txt
□       □□□ Data
□       □   □□□ _readme.txt
□       □□□ ServiceManifest.xml
□□□ install.sh
□□□ settings.gradle
□□□ uninstall.sh
```

Implement the service

Open **HelloWorldApplication/HelloWorld/src/statelessservice/HelloWorldService.java**. This class defines the service type, and can run any code. The service API provides two entry points for your code:

- An open-ended entry point method, called `runAsync()`, where you can begin executing any workloads, including long-running compute workloads.

```
@Override
protected CompletableFuture<?> runAsync() {
    ...
}
```

- A communication entry point where you can plug in your communication stack of choice. This is where you can start receiving requests from users and other services.

```
@Override
protected List<ServiceInstanceListener> createServiceInstanceListeners() {
    ...
}
```

In this tutorial, we will focus on the `runAsync()` entry point method. This is where you can immediately start running your code.

RunAsync

The platform calls this method when an instance of a service is placed and ready to execute. The open/close cycle of a service instance can occur many times over the lifetime of the service as a whole. This can happen for various reasons, including:

- The system moves your service instances for resource balancing.
- Faults occur in your code.
- The application or system is upgraded.
- The underlying hardware experiences an outage.

This orchestration is managed by Service Fabric to keep your service highly available and properly balanced.

Cancellation

It is vital that your code in `runAsync()` can stop execution when notified by Service Fabric. The `CompletableFuture` returned from `runAsync()` is canceled when Service Fabric requires your service to stop execution. The following example demonstrates how to handle a cancellation event:

```
    @Override
    protected CompletableFuture<?> runAsync() {

        CompletableFuture<?> completableFuture = new CompletableFuture<>();
        ExecutorService service = Executors.newFixedThreadPool(1);

        Future<?> userTask = service.submit(() -> {
            while (!Thread.currentThread().isInterrupted()) {
                try
                {
                   logger.log(Level.INFO, this.context().serviceName().toString());
                   Thread.sleep(1000);
                }
                catch (InterruptedException ex)
                {
                    logger.log(Level.INFO, this.context().serviceName().toString() + " interrupted. Exiting");
                    return;
                }
            }
         });

        completableFuture.handle((r, ex) -> {
            if (ex instanceof CancellationException) {
                userTask.cancel(true);
                service.shutdown();
            }
            return null;
        });

        return completableFuture;
   }
```

Service registration

Service types must be registered with the Service Fabric runtime. The service type is defined in the `ServiceManifest.xml` and your service class that implements `StatelessService`. Service registration is performed in the process main entry point. In this example, the process main entry point is `HelloWorldServiceHost.java`:

```
public static void main(String[] args) throws Exception {
    try {
        ServiceRuntime.registerStatelessServiceAsync("HelloWorldType", (context) -> new HelloWorldService(),
Duration.ofSeconds(10));
        logger.log(Level.INFO, "Registered stateless service type HelloWorldType.");
        Thread.sleep(Long.MAX_VALUE);
    }
    catch (Exception ex) {
        logger.log(Level.SEVERE, "Exception in registration: {0}", ex.toString());
        throw ex;
    }
}
```

Run the application

The Yeoman scaffolding includes a gradle script to build the application and bash scripts to deploy and un-deploy the application. To run the application, first build the application with gradle:

```
$ gradle
```

This will produce a Service Fabric application package that can be deployed using Service Fabric Azure CLI. The install.sh script contains the necessary Azure CLI commands to deploy the application package. Simply run the install.sh script to deploy:

```
$ ./install.sh
```

Architecture for stateful and stateless Reliable Services

11/15/2016 • 5 min to read • Edit on GitHub

Contributors

AlanWarwick • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Vaclav Turecek • katiecumming • Jesse Benson

An Azure Service Fabric Reliable Service may be stateful or stateless. Each type of service runs within a specific architecture. These architectures are described in this article. See the Reliable Service overview for more information about the differences between stateful and stateless services.

Stateful Reliable Services

Architecture of a stateful service

Stateful Reliable Service

A stateful Reliable Service can derive from either the StatefulService or StatefulServiceBase class. Both of these base classes are provided by Service Fabric. They offer various levels of support and abstraction for the stateful service to

interface with Service Fabric--and to participate as a service within the Service Fabric cluster.

StatefulService derives from StatefulServiceBase. StatefulServiceBase offers services more flexibility, but requires more understanding of the internals of Service Fabric. See the Reliable Service overview and Reliable Service advanced usage for more information on the specifics of writing services by using the StatefulService and StatefulServiceBase classes.

Both base classes manage the lifetime and role of the service implementation. The service implementation may override virtual methods of either base class if the service implementation has work to do at those points in the service implementation lifecycle--or if it wants to create a communication listener object. Note that although a service implementation may implement its own communication listener object exposing ICommunicationListener, in the diagram above, the communication listener is implemented by Service Fabric--as the service implementation uses a communication listener that is implemented by Service Fabric.

A stateful Reliable Service uses the reliable state manager to take advantage of reliable collections. Reliable collections are local data structures that are highly available to the service--that is, they are always available, regardless of service failovers. Each type of reliable collection is implemented by a reliable state provider. For more information on reliable collections, see the reliable collections overview.

Reliable state manager and state providers

The reliable state manager is the object that manages reliable state providers. It has the functionality to create, delete, enumerate, and ensure that the reliable state providers are persisted and highly available. A reliable state provider instance represents an instance of a persisted and highly available data structure, such as a dictionary or a queue.

Each reliable state provider exposes an interface that is used by a stateful service to interact with the reliable state provider. For example, IReliableDictionary is used to interface with the reliable dictionary, while IReliableQueue is used to interface with the reliable queue. All reliable state providers implement the IReliableState interface.

The reliable state manager has an interface named IReliableStateManager, which allows access to it from a stateful service. Interfaces to reliable state providers are returned through IReliableStateManager.

The reliable state manager uses a plug-in architecture so that new types of reliable collections can be plugged in dynamically.

The reliable dictionary and reliable queue are built upon the implementation of a high-performance, versioned differential store.

Transactional replicator

The transactional replicator component is responsible for ensuring that the state of a service (that is, the state within the reliable state manager and the reliable collections) is consistent across all replicas running the service. It also ensures that the state is persisted in the log. The reliable state manager interfaces with the transactional replicator via a private mechanism.

The transactional replicator uses a network protocol to communicate state with other replicas of the service instance so that all replicas have up-to-date state information.

The transactional replicator uses a log to persist state information so that the state information survives process or node crashes. The interface to the log is via a private mechanism.

Log

The log component provides a high-performance persistent store that can be optimized for writing to spinning or solid-state disks. The design of the log is for the persistent storage (i.e., hard disks) to be local to the nodes that are running the stateful service. This allows for low latencies and high throughput, as compared to remote persistent storage, which is not local to the node.

The log component uses multiple log files. There is a node-wide shared log file that all replicas use as it can provide

the lowest latency and highest throughput for storing state data. By default the shared log is placed in the Service Fabric node work directory but it may also be configured to be placed at another location, ideally on a disk reserved for only the shared log. Each replica for the service also has a dedicated log file and the dedicated log is placed within the service's work directory. There is no mechanism to configure the dedicated log to be placed at a different location.

The shared log is a transitional area for the replica's state information, while the dedicated log file is the final destination where it is persisted. In this design, the state information is first written to the shared log file and then lazily moved to the dedicated log file in the background. In this way, the write to the shared log would have the lowest latency and highest throughput which allows the service to make progress faster.

Reads and writes to the shared log are done via direct IO to preallocated space on the disk for the shared log file. To allow optimal use of disk space on the drive with dedicated logs, the dedicated log file is created as a NTFS sparse file. Note that this will allow overprovisioning of disk space and the OS will show the dedicated log files using much more disk space than is actually used.

Aside from a minimal user-mode interface to the log, the log is written as a kernel-mode driver. By running as a kernel-mode driver, the log can provide the highest performance to all services that use it.

For more information about configuring the log, see Configuring stateful Reliable Services.

Stateless Reliable Service

Architecture of a stateless service

Stateless Reliable Service

Stateless service implementations derive from the StatelessService or StatelessServiceBase class. The StatelessServiceBase class allows more flexibility than the StatelessService class. Both base classes manage the lifetime and role of a service.

The service implementation may override virtual methods of either base class if the service has work to do at those points in the service lifecycle--or if it wants to create a communication listener object. Note that although the service may implement its own communication listener object exposing ICommunicationListener, in the diagram above, the communication listener is implemented by Service Fabric, as that service implementation uses a communication listener that is implemented by Service Fabric.

See the Reliable Service overview and Reliable Service advanced usage for more information on the specifics of writing services using the StatelessService and StatelessServiceBase classes.

Next steps

For more information about Service Fabric, see:

Reliable service overview

Quick start

Reliable collections overview

Reliable service advanced usage

Reliable service configuration

Introduction to Reliable Collections in Azure Service Fabric stateful services

11/15/2016 • 8 min to read • Edit on GitHub

Contributors

Mert Coskun • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Preetha Subbarayalu • Vaclav Turecek • Matt Barry • Jesse Benson

Reliable Collections enable you to write highly available, scalable, and low-latency cloud applications as though you were writing single computer applications. The classes in the **Microsoft.ServiceFabric.Data.Collections** namespace provide a set of out-of-the-box collections that automatically make your state highly available. Developers need to program only to the Reliable Collection APIs and let Reliable Collections manage the replicated and local state.

The key difference between Reliable Collections and other high-availability technologies (such as Redis, Azure Table service, and Azure Queue service) is that the state is kept locally in the service instance while also being made highly available. This means that:

- All reads are local, which results in low latency and high-throughput reads.
- All writes incur the minimum number of network IOs, which results in low latency and high-throughput writes.

Reliable Collections can be thought of as the natural evolution of the **System.Collections** classes: a new set of collections that are designed for the cloud and multi-computer applications without increasing complexity for the developer. As such, Reliable Collections are:

- Replicated: State changes are replicated for high availability.
- Persisted: Data is persisted to disk for durability against large-scale outages (for example, a datacenter power outage).
- Asynchronous: APIs are asynchronous to ensure that threads are not blocked when incurring IO.
- Transactional: APIs utilize the abstraction of transactions so you can manage multiple Reliable Collections within a service easily.

Reliable Collections provide strong consistency guarantees out of the box in order to make reasoning about application state easier. Strong consistency is achieved by ensuring transaction commits finish only after the entire transaction has been logged on a majority quorum of replicas, including the primary. To achieve weaker

consistency, applications can acknowledge back to the client/requester before the asynchronous commit returns.

The Reliable Collections APIs are an evolution of concurrent collections APIs (found in the **System.Collections.Concurrent** namespace):

- Asynchronous: Returns a task since, unlike concurrent collections, the operations are replicated and persisted.
- No out parameters: Uses `ConditionalValue<T>` to return a bool and a value instead of out parameters. `ConditionalValue<T>` is like `Nullable<T>` but does not require T to be a struct.
- Transactions: Uses a transaction object to enable the user to group actions on multiple Reliable Collections in a transaction.

Today, **Microsoft.ServiceFabric.Data.Collections** contains two collections:

- Reliable Dictionary: Represents a replicated, transactional, and asynchronous collection of key/value pairs. Similar to **ConcurrentDictionary**, both the key and the value can be of any type.
- Reliable Queue: Represents a replicated, transactional, and asynchronous strict first-in, first-out (FIFO) queue. Similar to **ConcurrentQueue**, the value can be of any type.

Isolation levels

Isolation level defines the degree to which the transaction must be isolated from modifications made by other transactions. There are two isolation levels that are supported in Reliable Collections:

- **Repeatable Read**: Specifies that statements cannot read data that has been modified but not yet committed by other transactions and that no other transactions can modify data that has been read by the current transaction until the current transaction finishes. For more details, see https://msdn.microsoft.com/library/ms173763.aspx.
- **Snapshot**: Specifies that data read by any statement in a transaction will be the transactionally consistent version of the data that existed at the start of the transaction. The transaction can recognize only data modifications that were committed before the start of the transaction. Data modifications made by other transactions after the start of the current transaction are not visible to statements executing in the current transaction. The effect is as if the statements in a transaction get a snapshot of the committed data as it existed at the start of the transaction. Snapshots are consistent across Reliable Collections. For more details, see https://msdn.microsoft.com/library/ms173763.aspx.

Reliable Collections automatically choose the isolation level to use for a given read operation depending on the operation and the role of the replica at the time of transaction's creation. Following is the table that depicts isolation level defaults for Reliable Dictionary and Queue operations.

OPERATION \ ROLE	PRIMARY	SECONDARY
Single Entity Read	Repeatable Read	Snapshot
Enumeration \ Count	Snapshot	Snapshot

> **NOTE**
> Common examples for Single Entity Operations are `IReliableDictionary.TryGetValueAsync`, `IReliableQueue.TryPeekAsync`.

Both the Reliable Dictionary and the Reliable Queue support Read Your Writes. In other words, any write within a transaction will be visible to a following read that belongs to the same transaction.

Locking

In Reliable Collections, all transactions are two-phased: a transaction does not release the locks it has acquired until the transaction terminates with either an abort or a commit.

Reliable Dictionary uses row level locking for all single entity operations. Reliable Queue trades off concurrency for strict transactional FIFO property. Reliable Queue uses operation level locks allowing one transaction with `TryPeekAsync` and/or `TryDequeueAsync` and one transaction with `EnqueueAsync` at a time. Note that to preserve FIFO, if a `TryPeekAsync` or `TryDequeueAsync` ever observes that the Reliable Queue is empty, they will also lock `EnqueueAsync`.

Write operations always take Exclusive locks. For read operations, the locking depends on a couple of factors. Any read operation done using Snapshot isolation is lock free. Any Repeatable Read operation by default takes Shared locks. However, for any read operation that supports Repeatable Read, the user can ask for an Update lock instead of the Shared lock. An Update lock is an asymmetric lock used to prevent a common form of deadlock that occurs when multiple transactions lock resources for potential updates at a later time.

The lock compatibility matrix can be found below:

REQUEST \ GRANTED	NONE	SHARED	UPDATE	EXCLUSIVE
Shared	No conflict	No conflict	Conflict	Conflict
Update	No conflict	No conflict	Conflict	Conflict
Exclusive	No conflict	Conflict	Conflict	Conflict

Note that a time-out argument in the Reliable Collections APIs is used for deadlock detection. For example, two transactions (T1 and T2) are trying to read and update K1. It is possible for them to deadlock, because they both end up having the Shared lock. In this case, one or both of the operations will time out.

Note that the above deadlock scenario is a great example of how an Update lock can prevent deadlocks.

Persistence model

The Reliable State Manager and Reliable Collections follow a persistence model that is called Log and Checkpoint. This is a model where each state change is logged on disk and applied only in memory. The complete state itself is persisted only occasionally (a.k.a. Checkpoint). The benefit is that deltas are turned into sequential append-only writes on disk for improved performance.

To better understand the Log and Checkpoint model, let’s first look at the infinite disk scenario. The Reliable State Manager logs every operation before it is replicated. This allows the Reliable Collection to apply only the operation in memory. Since logs are persisted, even when the replica fails and needs to be restarted, the Reliable State Manager has enough information in its logs to replay all the operations the replica has lost. As the disk is infinite, log records never need to be removed and the Reliable Collection needs to manage only the in-memory state.

Now let’s look at the finite disk scenario. As log records accumulate, the Reliable State Manager will run out of disk space. Before that happens, the Reliable State Manager needs to truncate its log to make room for the newer records. It will request the Reliable Collections to checkpoint their in-memory state to disk. It is the Reliable Collections' responsibility to persist its state up to that point. Once the Reliable Collections complete their checkpoints, the Reliable State Manager can truncate the log to free up disk space. This way, when the replica needs to be restarted, Reliable Collections will recover their checkpointed state, and the Reliable State Manager will recover and play back all the state changes that occurred since the checkpoint.

> **NOTE**
> Another value add of checkpointing is that it improves recovery performance in common cases. This is because checkpoints contain only the latest versions.

Recommendations

- Do not modify an object of custom type returned by read operations (e.g., `TryPeekAsync` or `TryGetValueAsync`). Reliable Collections, just like Concurrent Collections, return a reference to the objects and not a copy.
- Do deep copy the returned object of a custom type before modifying it. Since structs and built-in types are pass-by-value, you do not need to do a deep copy on them.
- Do not use `TimeSpan.MaxValue` for time-outs. Time-outs should be used to detect deadlocks.
- Do not use a transaction after it has been committed, aborted, or disposed.
- Do not use an enumeration outside of the transaction scope it was created in.
- Do not create a transaction within another transaction's `using` statement because it can cause deadlocks.
- Do ensure that your `IComparable<TKey>` implementation is correct. The system takes dependency on this for merging checkpoints.
- Do use Update lock when reading an item with an intention to update it to prevent a certain class of deadlocks.
- Consider using backup and restore functionality to have disaster recovery.
- Avoid mixing single entity operations and multi-entity operations (e.g `GetCountAsync`, `CreateEnumerableAsync`) in the same transaction due to the different isolation levels.

Here are some things to keep in mind:

- The default time-out is 4 seconds for all the Reliable Collection APIs. Most users should not override this.
- The default cancellation token is `CancellationToken.None` in all Reliable Collections APIs.
- The key type parameter (*TKey*) for a Reliable Dictionary must correctly implement `GetHashCode()` and `Equals()`. Keys must be immutable.
- To achieve high availability for the Reliable Collections, each service should have at least a target and minimum replica set size of 3.
- Read operations on the secondary may read versions that are not quorum committed. This means that a version of data that is read from a single secondary might be false progressed. Of course, reads from Primary are always stable: can never be false progressed.

Next steps

- Reliable Services quick start
- Working with Reliable Collections
- Reliable Services notifications
- Reliable Services backup and restore (disaster recovery)
- Reliable State Manager configuration
- Getting started with Service Fabric Web API services
- Advanced usage of the Reliable Services programming model
- Developer reference for Reliable Collections

Working with Reliable Collections

11/15/2016 • 10 min to read • Edit on GitHub

Contributors

Jeffrey Richter • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Todd Abel

Service Fabric offers a stateful programming model available to .NET developers via Reliable Collections. Specifically, Service Fabric provides reliable dictionary and reliable queue classes. When you use these classes, your state is partitioned (for scalability), replicated (for availability), and transacted within a partition (for ACID semantics). Let's look at a typical usage of a reliable dictionary object and see what its actually doing.

```
retry:
try {
   // Create a new Transaction object for this partition
   using (ITransaction tx = base.StateManager.CreateTransaction()) {
      // AddAsync takes key's write lock; if >4 secs, TimeoutException
      // Key & value put in temp dictionary (read your own writes),
      // serialized, redo/undo record is logged & sent to
      // secondary replicas
      await m_dic.AddAsync(tx, key, value, cancellationToken);

      // CommitAsync sends Commit record to log & secondary replicas
      // After quorum responds, all locks released
      await tx.CommitAsync();
   }
   // If CommitAsync not called, Dispose sends Abort
   // record to log & all locks released
}
catch (TimeoutException) {
   await Task.Delay(100, cancellationToken); goto retry;
}
```

All operations on reliable dictionary objects (except for ClearAsync which is not undoable), require an ITransaction object. This object has associated with it any and all changes you're attempting to make to any reliable dictionary and/or reliable queue objects within a single partition. You acquire an ITransaction object by calling the partition's StateManager's CreateTransaction method.

In the code above, the ITransaction object is passed to a reliable dictionary's AddAsync method. Internally, dictionary methods that accepts a key take a reader/writer lock associated with the key. If the method modifies the key's value, the method takes a write lock on the key and if the method only reads from the key's value, then a read lock is taken on the key. Since AddAsync modifies the key's value to the new, passed-in value, the key's write lock is taken. So, if 2 (or more) threads attempt to add values with the same key at the same time, one thread will acquire the write lock and the other threads will block. By default, methods block for up to 4 seconds to acquire the lock; after 4 seconds, the methods throw a TimeoutException. Method overloads exist allowing you to pass an explicit timeout value if you'd prefer.

Usually, you write your code to react to a TimeoutException by catching it and retrying the entire operation (as shown in the code above). In my simple code, I'm just calling Task.Delay passing 100 milliseconds each time. But, in reality, you might be better off using some kind of exponential back-off delay instead.

Once the lock is acquired, AddAsync adds the key and value object references to an internal temporary dictionary associated with the ITransaction object. This is done to provide you with read-your-own-writes semantics. That is, after you call AddAsync, a later call to TryGetValueAsync (using the same ITransaction object) will return the value even if you have not yet committed the transaction. Next, AddAsync serializes your key and value objects to byte

arrays and appends these byte arrays to a log file on the local node. Finally, AddAsync sends the byte arrays to all the secondary replicas so they have the same key/value information. Even though the key/value information has been written to a log file, the information is not considered part of the dictionary until the transaction that they are associated with has been committed.

In the code above, the call to CommitAsync commits all of the transaction's operations. Specifically, it appends commit information to the log file on the local node and also sends the commit record to all the secondary replicas. Once a quorum (majority) of the replicas has replied, all data changes are considered permanent and any locks associated with keys that were manipulated via the ITransaction object are released so other threads/transactions can manipulate the same keys and their values.

If CommitAsync is not called (usually due to an exception being thrown), then the ITransaction object gets disposed. When disposing an uncommitted ITransaction object, Service Fabric appends abort information to the local node's log file and nothing needs to be sent to any of the secondary replicas. And then, any locks associated with keys that were manipulated via the transaction are released.

Common pitfalls and how to avoid them

Now that you understand how the reliable collections work internally, let's take a look at some common misuses of them. See the code below:

```
using (ITransaction tx = StateManager.CreateTransaction()) {
   // AddAsync serializes the name/user, logs the bytes,
   // & sends the bytes to the secondary replicas.
   await m_dic.AddAsync(tx, name, user);

   // The line below updates the property's value in memory only; the
   // new value is NOT serialized, logged, & sent to secondary replicas.
   user.LastLogin = DateTime.UtcNow;  // Corruption!

   await tx.CommitAsync();
}
```

When working with a regular .NET dictionary, you can add a key/value to the dictionary and then change the value of a property (such as LastLogin). However, this code will not work correctly with a reliable dictionary. Remember from the earlier discussion, the call to AddAsync serializes the key/value objects to byte arrays and then saves the arrays to a local file and also sends them to the secondary replicas. If you later change a property, this changes the property's value in memory only; it does not impact the local file or the data sent to the replicas. If the process crashes, what's in memory is thrown away. When a new process starts or if another replica becomes primary, then the old property value is what is available.

I cannot stress enough how easy it is to make the kind of mistake shown above. And, you will only learn about the mistake if/when the process goes down. The correct way to write the code is simply to reverse the two lines:

```
using (ITransaction tx = StateManager.CreateTransaction()) {
   user.LastLogin = DateTime.UtcNow;  // Do this BEFORE calling AddAsync
   await m_dic.AddAsync(tx, name, user);
   await tx.CommitAsync();
}
```

Here is another example showing a common mistake:

```
using (ITransaction tx = StateManager.CreateTransaction()) {
   // Use the user's name to look up their data
   ConditionalValue<User> user =
      await m_dic.TryGetValueAsync(tx, name);

   // The user exists in the dictionary, update one of their properties.
   if (user.HasValue) {
      // The line below updates the property's value in memory only; the
      // new value is NOT serialized, logged, & sent to secondary replicas.
      user.Value.LastLogin = DateTime.UtcNow; // Corruption!
      await tx.CommitAsync();
   }
}
```

Again, with regular .NET dictionaries, the code above works fine and is a common pattern: the developer uses a key to look up a value. If the value exists, the developer changes a property's value. However, with reliable collections, this code exhibits the same problem as already discussed: **you MUST not modify an object once you have given it to a reliable collection.**

The correct way to update a value in a reliable collection, is to get a reference to the existing value and consider the object referred to by this reference immutable. Then, create a new object which is an exact copy of the original object. Now, you can modify the state of this new object and write the new object into the collection so that it gets serialized to byte arrays, appended to the local file and sent to the replicas. After committing the change(s), the in-memory objects, the local file, and all the replicas have the same exact state. All is good!

The code below shows the correct way to update a value in a reliable collection:

```
using (ITransaction tx = StateManager.CreateTransaction()) {
   // Use the user's name to look up their data
   ConditionalValue<User> currentUser =
      await m_dic.TryGetValueAsync(tx, name);

   // The user exists in the dictionary, update one of their properties.
   if (currentUser.HasValue) {
      // Create new user object with the same state as the current user object.
      // NOTE: This must be a deep copy; not a shallow copy. Specifically, only
      // immutable state can be shared by currentUser & updatedUser object graphs.
      User updatedUser = new User(currentUser);

      // In the new object, modify any properties you desire
      updatedUser.LastLogin = DateTime.UtcNow;

      // Update the key's value to the updateUser info
      await m_dic.SetValue(tx, name, updatedUser);

      await tx.CommitAsync();
   }
}
```

Define immutable data types to prevent programmer error

Ideally, we'd like the compiler to report errors when you accidentally produce code that mutates state of an object that you are supposed to consider immutable. But, the C# compiler does not have the ability to do this. So, to avoid potential programmer bugs, we highly recommend that you define the types you use with reliable collections to be immutable types. Specifically, this means that you stick to core value types (such as numbers [Int32, UInt64, etc.], DateTime, Guid, TimeSpan, and the like). And, of course, you can also use String. It is best to avoid collection properties as serializing and deserializing them can frequently can hurt performance. However, if you want to use collection properties, we highly recommend the use of .NET's immutable collections library (System.Collections.Immutable). This library is available for download from http://nuget.org. We also recommend sealing your classes and making fields read-only whenever possible.

The UserInfo type below demonstrates how to define an immutable type taking advantage of aforementioned recommendations.

```
[DataContract]
// If you don't seal, you must ensure that any derived classes are also immutable
public sealed class UserInfo {
   private static readonly IEnumerable<ItemId> NoBids = ImmutableList<ItemId>.Empty;

   public UserInfo(String email, IEnumerable<ItemId> itemsBidding = null) {
      Email = email;
      ItemsBidding = (itemsBidding == null) ? NoBids : itemsBidding.ToImmutableList();
   }

   [OnDeserialized]
   private void OnDeserialized(StreamingContext context) {
      // Convert the deserialized collection to an immutable collection
      ItemsBidding = ItemsBidding.ToImmutableList();
   }

   [DataMember]
   public readonly String Email;

   // Ideally, this would be a readonly field but it can't be because OnDeserialized
   // has to set it. So instead, the getter is public and the setter is private.
   [DataMember]
   public IEnumerable<ItemId> ItemsBidding { get; private set; }

   // Since each UserInfo object is immutable, we add a new ItemId to the ItemsBidding
   // collection by creating a new immutable UserInfo object with the added ItemId.
   public UserInfo AddItemBidding(ItemId itemId) {
      return new UserInfo(Email, ((ImmutableList<ItemId>)ItemsBidding).Add(itemId));
   }
}
```

The ItemId type is also an immutable type as shown here:

```
[DataContract]
public struct ItemId {

   [DataMember] public readonly String Seller;
   [DataMember] public readonly String ItemName;
   public ItemId(String seller, String itemName) {
      Seller = seller;
      ItemName = itemName;
   }
}
```

Schema versioning (upgrades)

Internally, Reliable Collections serialize your objects using .NET's DataContractSerializer. The serialized objects are persisted to the primary replica's local disk and are also transmitted to the secondary replicas. As your service matures, it's likely you'll want to change the kind of data (schema) your service requires. You must approach versioning of your data with great care. First and foremost, you must always be able to deserialize old data. Specifically, this means your deserialization code must be infinitely backward compatible: Version 333 of your service code must be able to operate on data placed in a reliable collection by version 1 of your service code 5 years ago.

Furthermore, service code is upgraded one upgrade domain at a time. So, during an upgrade, you have two different versions of your service code running simultaneously. You must avoid having the new version of your service code use the new schema as old versions of your service code might not be able to handle the new schema. When possible, you should design each version of your service to be forward compatible by 1 version. Specifically,

this means that V1 of your service code should be able to simply ignore any schema elements it does not explicitly handle. However, it must be able to save any data it doesn't explicitly know about and simply write it back out when updating a dictionary key or value.

> **WARNING**
>
> While you can modify the schema of a key, you must ensure that your key's hash code and equals algorithms are stable. If you change how either of these algorithms operate, you will not be able to look up the key within the reliable dictionary ever again.

Alternatively, you can perform what is typically referred to as a 2-phase upgrade. With a 2-phase upgrade, you upgrade your service from V1 to V2: V2 contains the code that knows how to deal with the new schema change but this code doesn't execute. When the V2 code reads V1 data, it operates on it and writes V1 data. Then, after the upgrade is complete across all upgrade domains, you can somehow signal to the running V2 instances that the upgrade is complete. (One way to signal this is to roll out a configuration upgrade; this is what makes this a 2-phase upgrade.) Now, the V2 instances can read V1 data, convert it to V2 data, operate on it, and write it out as V2 data. When other instances read V2 data, they do not need to convert it, they just operate on it, and write out V2 data.

Next Steps

To learn about creating forward compatible data contracts, see Forward-Compatible Data Contracts.

To learn best practices on versioning data contracts, see Data Contract Versioning.

To learn how to implement version tolerant data contracts, see Version-Tolerant Serialization Callbacks.

To learn how to provide a data structure that can interoperate across multiple versions, see IExtensibleDataObject.

Configure Stateful reliable services

11/15/2016 • 9 min to read • Edit on GitHub

Contributors

Sumukh Shivaprakash • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Mert Coskun • Hareesh Nagaraj • Vaclav Turecek • AlanWarwick • stevehug • Andy Pasic • Tyler Adams • Andy De George • Ryan Wike

There are two sets of configuration settings for reliable services. One set is global for all reliable services in the cluster while the other set is specific to a particular reliable service.

Global Configuration

The global reliable service configuration is specified in the cluster manifest for the cluster under the KtlLogger section. It allows configuration of the shared log location and size plus the global memory limits used by the logger. The cluster manifest is a single XML file that holds settings and configurations that apply to all nodes and services in the cluster. The file is typically called ClusterManifest.xml. You can see the cluster manifest for your cluster using the Get-ServiceFabricClusterManifest powershell command.

Configuration names

NAME	UNIT	DEFAULT VALUE	REMARKS
WriteBufferMemoryPoolMinimumInKB	Kilobytes	8388608	Minimum number of KB to allocate in kernel mode for the logger write buffer memory pool. This memory pool is used for caching state information before writing to disk.
WriteBufferMemoryPoolMaximumInKB	Kilobytes	No Limit	Maximum size to which the logger write buffer memory pool can grow.
SharedLogId	GUID	""	Specifies a unique GUID to use for identifying the default shared log file used by all reliable services on all nodes in the cluster that do not specify the SharedLogId in their service specific configuration. If SharedLogId is specified, then SharedLogPath must also be specified.

NAME	UNIT	DEFAULT VALUE	REMARKS
SharedLogPath	Fully qualified path name	""	Specifies the fully qualified path where the shared log file used by all reliable services on all nodes in the cluster that do not specify the SharedLogPath in their service specific configuration. However, if SharedLogPath is specified, then SharedLogId must also be specified.
SharedLogSizeInMB	Megabytes	8192	Specifies the number of MB of disk space to statically allocate for the shared log. The value must be 2048 or larger.

Sample cluster manifest section

```
<Section Name="KtlLogger">
  <Parameter Name="WriteBufferMemoryPoolMinimumInKB" Value="8192" />
  <Parameter Name="WriteBufferMemoryPoolMaximumInKB" Value="8192" />
  <Parameter Name="SharedLogId" Value="{7668BB54-FE9C-48ed-81AC-FF89E60ED2EF}"/>
  <Parameter Name="SharedLogPath" Value="f:\SharedLog.Log"/>
  <Parameter Name="SharedLogSizeInMB" Value="16383"/>
</Section>
```

Remarks

The logger has a global pool of memory allocated from non paged kernel memory that is available to all reliable services on a node for caching state data before being written to the dedicated log associated with the reliable service replica. The pool size is controlled by the WriteBufferMemoryPoolMinimumInKB and WriteBufferMemoryPoolMaximumInKB settings. WriteBufferMemoryPoolMinimumInKB specifies both the initial size of this memory pool and the lowest size to which the memory pool may shrink. WriteBufferMemoryPoolMaximumInKB is the highest size to which the memory pool may grow. Each reliable service replica that is opened may increase the size of the memory pool by a system determined amount up to WriteBufferMemoryPoolMaximumInKB. If there is more demand for memory from the memory pool than is available, requests for memory will be delayed until memory is available. Therefore if the write buffer memory pool is too small for a particular configuration then performance may suffer.

The SharedLogId and SharedLogPath settings are always used together to define the GUID and location for the default shared log for all nodes in the cluster. The default shared log is used for all reliable services that do not specify the settings in the settings.xml for the specific service. For best performance, shared log files should be placed on disks that are used solely for the shared log file to reduce contention.

SharedLogSizeInMB specifies the amount of disk space to preallocate for the default shared log on all nodes. SharedLogId and SharedLogPath do not need to be specified in order for SharedLogSizeInMB to be specified.

Service Specific Configuration

You can modify stateful Reliable Services' default configurations by using the configuration package (Config) or the service implementation (code).

- **Config** - Configuration via the config package is accomplished by changing the Settings.xml file that is generated in the Microsoft Visual Studio package root under the Config folder for each service in the

application.

- **Code** - Configuration via code is accomplished by creating a ReliableStateManager using a ReliableStateManagerConfiguration object with the appropriate options set.

By default, the Azure Service Fabric runtime looks for predefined section names in the Settings.xml file and consumes the configuration values while creating the underlying runtime components.

> **NOTE**
> Do **not** delete the section names of the following configurations in the Settings.xml file that is generated in the Visual Studio solution unless you plan to configure your service via code. Renaming the config package or section names will require a code change when configuring the ReliableStateManager.

Replicator security configuration

Replicator security configurations are used to secure the communication channel that is used during replication. This means that services will not be able to see each other's replication traffic, ensuring that the data that is made highly available is also secure. By default, an empty security configuration section prevents replication security.

Default section name

ReplicatorSecurityConfig

> **NOTE**
> To change this section name, override the replicatorSecuritySectionName parameter to the ReliableStateManagerConfiguration constructor when creating the ReliableStateManager for this service.

Replicator configuration

Replicator configurations configure the replicator that is responsible for making the stateful Reliable Service's state highly reliable by replicating and persisting the state locally. The default configuration is generated by the Visual Studio template and should suffice. This section talks about additional configurations that are available to tune the replicator.

Default section name

ReplicatorConfig

> **NOTE**
> To change this section name, override the replicatorSettingsSectionName parameter to the ReliableStateManagerConfiguration constructor when creating the ReliableStateManager for this service.

Configuration names

NAME	UNIT	DEFAULT VALUE	REMARKS

NAME	UNIT	DEFAULT VALUE	REMARKS
BatchAcknowledgementInterval	Seconds	0.015	Time period for which the replicator at the secondary waits after receiving an operation before sending back an acknowledgement to the primary. Any other acknowledgements to be sent for operations processed within this interval are sent as one response.
ReplicatorEndpoint	N/A	No default--required parameter	IP address and port that the primary/secondary replicator will use to communicate with other replicators in the replica set. This should reference a TCP resource endpoint in the service manifest. Refer to Service manifest resources to read more about defining endpoint resources in a service manifest.
MaxPrimaryReplicationQueueSize	Number of operations	8192	Maximum number of operations in the primary queue. An operation is freed up after the primary replicator receives an acknowledgement from all the secondary replicators. This value must be greater than 64 and a power of 2.
MaxSecondaryReplicationQueueSize	Number of operations	16384	Maximum number of operations in the secondary queue. An operation is freed up after making its state highly available through persistence. This value must be greater than 64 and a power of 2.
CheckpointThresholdInMB	MB	50	Amount of log file space after which the state is checkpointed.
MaxRecordSizeInKB	KB	1024	Largest record size that the replicator may write in the log. This value must be a multiple of 4 and greater than 16.

NAME	UNIT	DEFAULT VALUE	REMARKS
MinLogSizeInMB	MB	0 (system determined)	Minimum size of the transactional log. The log will not be allowed to truncate to a size below this setting. 0 indicates that the replicator will determine the minimum log size. Increasing this value increases the possibility of doing partial copies and incremental backups since chances of relevant log records being truncated is lowered.
TruncationThresholdFactor	Factor	2	Determines at what size of the log, truncation will be triggered. Truncation threshold is determined by MinLogSizeInMB multiplied by TruncationThresholdFactor. TruncationThresholdFactor must be greater than 1. MinLogSizeInMB * TruncationThresholdFactor must be less than MaxStreamSizeInMB.
ThrottlingThresholdFactor	Factor	4	Determines at what size of the log, the replica will start being throttled. Throttling threshold (in MB) is determined by Max((MinLogSizeInMB * ThrottlingThresholdFactor), (CheckpointThresholdInMB * ThrottlingThresholdFactor)). Throttling threshold (in MB) must be greater than truncation threshold (in MB). Truncation threshold (in MB) must be less than MaxStreamSizeInMB.
MaxAccumulatedBackupLogSizeInMB	MB	800	Max accumulated size (in MB) of backup logs in a given backup log chain. An incremental backup requests will fail if the incremental backup would generate a backup log that would cause the accumulated backup logs since the relevant full backup to be larger than this size. In such cases, user is required to take a full backup.

NAME	UNIT	DEFAULT VALUE	REMARKS
SharedLogId	GUID	""	Specifies a unique GUID to use for identifying the shared log file used with this replica. Typically, services should not use this setting. However, if SharedLogId is specified, then SharedLogPath must also be specified.
SharedLogPath	Fully qualified path name	""	Specifies the fully qualified path where the shared log file for this replica will be created. Typically, services should not use this setting. However, if SharedLogPath is specified, then SharedLogId must also be specified.
SlowApiMonitoringDuration	Seconds	300	Sets the monitoring interval for managed API calls. Example: user provided backup callback function. After the interval has passed, a warning health report will be sent to the Health Manager.

Sample configuration via code

```
class Program
{
    /// <summary>
    /// This is the entry point of the service host process.
    /// </summary>
    static void Main()
    {
        ServiceRuntime.RegisterServiceAsync("HelloWorldStatefulType",
            context => new HelloWorldStateful(context,
                new ReliableStateManager(context,
        new ReliableStateManagerConfiguration(
                        new ReliableStateManagerReplicatorSettings()
            {
                RetryInterval = TimeSpan.FromSeconds(3)
                        }
            )))).GetAwaiter().GetResult();
    }
}
```

```
class MyStatefulService : StatefulService
{
    public MyStatefulService(StatefulServiceContext context, IReliableStateManagerReplica stateManager)
        : base(context, stateManager)
    { }
    ...
}
```

Sample configuration file

```
<?xml version="1.0" encoding="utf-8"?>
<Settings xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.microsoft.com/2011/01/fabric">
   <Section Name="ReplicatorConfig">
      <Parameter Name="ReplicatorEndpoint" Value="ReplicatorEndpoint" />
      <Parameter Name="BatchAcknowledgementInterval" Value="0.05"/>
      <Parameter Name="CheckpointThresholdInMB" Value="512" />
   </Section>
   <Section Name="ReplicatorSecurityConfig">
      <Parameter Name="CredentialType" Value="X509" />
      <Parameter Name="FindType" Value="FindByThumbprint" />
      <Parameter Name="FindValue" Value="9d c9 06 b1 69 dc 4f af fd 16 97 ac 78 1e 80 67 90 74 9d 2f" />
      <Parameter Name="StoreLocation" Value="LocalMachine" />
      <Parameter Name="StoreName" Value="My" />
      <Parameter Name="ProtectionLevel" Value="EncryptAndSign" />
      <Parameter Name="AllowedCommonNames" Value="My-Test-SAN1-Alice,My-Test-SAN1-Bob" />
   </Section>
</Settings>
```

Remarks

BatchAcknowledgementInterval controls replication latency. A value of '0' results in the lowest possible latency, at the cost of throughput (as more acknowledgement messages must be sent and processed, each containing fewer acknowledgements). The larger the value for BatchAcknowledgementInterval, the higher the overall replication throughput, at the cost of higher operation latency. This directly translates to the latency of transaction commits.

The value for CheckpointThresholdInMB controls the amount of disk space that the replicator can use to store state information in the replica's dedicated log file. Increasing this to a higher value than the default could result in faster reconfiguration times when a new replica is added to the set. This is due to the partial state transfer that takes place due to the availability of more history of operations in the log. This can potentially increase the recovery time of a replica after a crash.

The MaxRecordSizeInKB setting defines the maximum size of a record that can be written by the replicator into the log file. In most cases, the default 1024-KB record size is optimal. However, if the service is causing larger data items to be part of the state information, then this value might need to be increased. There is little benefit in making MaxRecordSizeInKB smaller than 1024, as smaller records use only the space needed for the smaller record. We expect that this value would need to be changed in only rare cases.

The SharedLogId and SharedLogPath settings are always used together to make a service use a separate shared log from the default shared log for the node. For best efficiency, as many services as possible should specify the same shared log. Shared log files should be placed on disks that are used solely for the shared log file to reduce head movement contention. We expect that this value would need to be changed in only rare cases.

Next steps

- Debug your Service Fabric application in Visual Studio
- Developer reference for Reliable Services

Reliable Services notifications

11/15/2016 • 6 min to read • Edit on GitHub

Contributors

Mert Coskun • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • ShawnJackson

Notifications allow clients to track the changes that are being made to an object that they're interested in. Two types of objects support notifications: *Reliable State Manager* and *Reliable Dictionary*.

Common reasons for using notifications are:

- Building materialized views, such as secondary indexes or aggregated filtered views of the replica's state. An example is a sorted index of all keys in Reliable Dictionary.
- Sending monitoring data, such as the number of users added in the last hour.

Notifications are fired as part of applying operations. Because of that, notifications should be handled as fast as possible, and synchronous events shouldn't include any expensive operations.

Reliable State Manager notifications

Reliable State Manager provides notifications for the following events:

- Transaction
  - Commit
- State manager
  - Rebuild
  - Addition of a reliable state
  - Removal of a reliable state

Reliable State Manager tracks the current inflight transactions. The only change in transaction state that causes a notification to be fired is a transaction being committed.

Reliable State Manager maintains a collection of reliable states like Reliable Dictionary and Reliable Queue. Reliable State Manager fires notifications when this collection changes: a reliable state is added or removed, or the entire collection is rebuilt. The Reliable State Manager collection is rebuilt in three cases:

- Recovery: When a replica starts, it recovers its previous state from the disk. At the end of recovery, it uses **NotifyStateManagerChangedEventArgs** to fire an event that contains the set of recovered reliable states.
- Full copy: Before a replica can join the configuration set, it has to be built. Sometimes, this requires a full copy of Reliable State Manager's state from the primary replica to be applied to the idle secondary replica. Reliable State Manager on the secondary replica uses **NotifyStateManagerChangedEventArgs** to fire an event that contains the set of reliable states that it acquired from the primary replica.
- Restore: In disaster recovery scenarios, the replica's state can be restored from a backup via **RestoreAsync**. In such cases, Reliable State Manager on the primary replica uses **NotifyStateManagerChangedEventArgs** to fire an event that contains the set of reliable states that it restored from the backup.

To register for transaction notifications and/or state manager notifications, you need to register with the **TransactionChanged** or **StateManagerChanged** events on Reliable State Manager. A common place to register with these event handlers is the constructor of your stateful service. When you register on the constructor, you won't miss any notification that's caused by a change during the lifetime of **IReliableStateManager**.

```
public MyService(StatefulServiceContext context)
    : base(MyService.EndpointName, context, CreateReliableStateManager(context))
{
    this.StateManager.TransactionChanged += this.OnTransactionChangedHandler;
    this.StateManager.StateManagerChanged += this.OnStateManagerChangedHandler;
}
```

The **TransactionChanged** event handler uses **NotifyTransactionChangedEventArgs** to provide details about the event. It contains the action property (for example, **NotifyTransactionChangedAction.Commit**) that specifies the type of change. It also contains the transaction property that provides a reference to the transaction that changed.

> **NOTE**
> Today, **TransactionChanged** events are raised only if the transaction is committed. The action is then equal to **NotifyTransactionChangedAction.Commit**. But in the future, events might be raised for other types of transaction state changes. We recommend checking the action and processing the event only if it's one that you expect.

Following is an example **TransactionChanged** event handler.

```
private void OnTransactionChangedHandler(object sender, NotifyTransactionChangedEventArgs e)
{
    if (e.Action == NotifyTransactionChangedAction.Commit)
    {
        this.lastCommitLsn = e.Transaction.CommitSequenceNumber;
        this.lastTransactionId = e.Transaction.TransactionId;

        this.lastCommittedTransactionList.Add(e.Transaction.TransactionId);
    }
}
```

The **StateManagerChanged** event handler uses **NotifyStateManagerChangedEventArgs** to provide details about the event. **NotifyStateManagerChangedEventArgs** has two subclasses: **NotifyStateManagerRebuildEventArgs** and **NotifyStateManagerSingleEntityChangedEventArgs**. You use the action property in **NotifyStateManagerChangedEventArgs** to cast **NotifyStateManagerChangedEventArgs** to the correct subclass:

- **NotifyStateManagerChangedAction.Rebuild**: **NotifyStateManagerRebuildEventArgs**
- **NotifyStateManagerChangedAction.Add** and **NotifyStateManagerChangedAction.Remove**: **NotifyStateManagerSingleEntityChangedEventArgs**

Following is an example **StateManagerChanged** notification handler.

```
public void OnStateManagerChangedHandler(object sender, NotifyStateManagerChangedEventArgs e)
{
    if (e.Action == NotifyStateManagerChangedAction.Rebuild)
    {
        this.ProcessStataManagerRebuildNotification(e);

        return;
    }

    this.ProcessStateManagerSingleEntityNotification(e);
}
```

Reliable Dictionary notifications

Reliable Dictionary provides notifications for the following events:

- Rebuild: Called when **ReliableDictionary** has recovered its state from a recovered or copied local state or backup.
- Clear: Called when the state of **ReliableDictionary** has been cleared through the **ClearAsync** method.
- Add: Called when an item has been added to **ReliableDictionary**.
- Update: Called when an item in **IReliableDictionary** has been updated.
- Remove: Called when an item in **IReliableDictionary** has been deleted.

To get Reliable Dictionary notifications, you need to register with the **DictionaryChanged** event handler on **IReliableDictionary**. A common place to register with these event handlers is in the **ReliableStateManager.StateManagerChanged** add notification. Registering when **IReliableDictionary** is added to **IReliableStateManager** ensures that you won't miss any notifications.

```
private void ProcessStateManagerSingleEntityNotification(NotifyStateManagerChangedEventArgs e)
{
    var operation = e as NotifyStateManagerSingleEntityChangedEventArgs;

    if (operation.Action == NotifyStateManagerChangedAction.Add)
    {
        if (operation.ReliableState is IReliableDictionary<TKey, TValue>)
        {
            var dictionary = (IReliableDictionary<TKey, TValue>)operation.ReliableState;
            dictionary.RebuildNotificationAsyncCallback = this.OnDictionaryRebuildNotificationHandlerAsync;
            dictionary.DictionaryChanged += this.OnDictionaryChangedHandler;
            }
        }
    }
}
```

> **NOTE**
>
> **ProcessStateManagerSingleEntityNotification** is the sample method that the preceding **OnStateManagerChangedHandler** example calls.

The preceding code sets the **IReliableNotificationAsyncCallback** interface, along with **DictionaryChanged**. Because **NotifyDictionaryRebuildEventArgs** contains an **IAsyncEnumerable** interface--which needs to be enumerated asynchronously--rebuild notifications are fired through **RebuildNotificationAsyncCallback** instead of **OnDictionaryChangedHandler**.

```
public async Task OnDictionaryRebuildNotificationHandlerAsync(
    IReliableDictionary<TKey, TValue> origin,
    NotifyDictionaryRebuildEventArgs<TKey, TValue> rebuildNotification)
{
    this.secondaryIndex.Clear();

    var enumerator = e.State.GetAsyncEnumerator();
    while (await enumerator.MoveNextAsync(CancellationToken.None))
    {
        this.secondaryIndex.Add(enumerator.Current.Key, enumerator.Current.Value);
    }
}
```

> **NOTE**
>
> In the preceding code, as part of processing the rebuild notification, first the maintained aggregated state is cleared. Because the reliable collection is being rebuilt with a new state, all previous notifications are irrelevant.

The **DictionaryChanged** event handler uses **NotifyDictionaryChangedEventArgs** to provide details about the

event. **NotifyDictionaryChangedEventArgs** has five subclasses. Use the action property in **NotifyDictionaryChangedEventArgs** to cast **NotifyDictionaryChangedEventArgs** to the correct subclass:

- **NotifyDictionaryChangedAction.Rebuild**: **NotifyDictionaryRebuildEventArgs**
- **NotifyDictionaryChangedAction.Clear**: **NotifyDictionaryClearEventArgs**
- **NotifyDictionaryChangedAction.Add** and **NotifyDictionaryChangedAction.Remove**: **NotifyDictionaryItemAddedEventArgs**
- **NotifyDictionaryChangedAction.Update**: **NotifyDictionaryItemUpdatedEventArgs**
- **NotifyDictionaryChangedAction.Remove**: **NotifyDictionaryItemRemovedEventArgs**

```
public void OnDictionaryChangedHandler(object sender, NotifyDictionaryChangedEventArgs<TKey, TValue> e)
{
    switch (e.Action)
    {
        case NotifyDictionaryChangedAction.Clear:
            var clearEvent = e as NotifyDictionaryClearEventArgs<TKey, TValue>;
            this.ProcessClearNotification(clearEvent);
            return;

        case NotifyDictionaryChangedAction.Add:
            var addEvent = e as NotifyDictionaryItemAddedEventArgs<TKey, TValue>;
            this.ProcessAddNotification(addEvent);
            return;

        case NotifyDictionaryChangedAction.Update:
            var updateEvent = e as NotifyDictionaryItemUpdatedEventArgs<TKey, TValue>;
            this.ProcessUpdateNotification(updateEvent);
            return;

        case NotifyDictionaryChangedAction.Remove:
            var deleteEvent = e as NotifyDictionaryItemRemovedEventArgs<TKey, TValue>;
            this.ProcessRemoveNotification(deleteEvent);
            return;

        default:
            break;
    }
}
```

Recommendations

- *Do* complete notification events as fast as possible.
- *Do not* execute any expensive operations (for example, I/O operations) as part of synchronous events.
- *Do* check the action type before you process the event. New action types might be added in the future.

Here are some things to keep in mind:

- Notifications are fired as part of the execution of an operation. For example, a restore notification is fired as the last step of a restore operation. A restore will not finish until the notification event is processed.
- Because notifications are fired as part of the applying operations, clients see only notifications for locally committed operations. And because operations are guaranteed only to be locally committed (in other words, logged), they might or might not be undone in the future.
- On the redo path, a single notification is fired for each applied operation. This means that if transaction T1 includes Create(X), Delete(X), and Create(X), you'll get one notification for the creation of X, one for the deletion, and one for the creation again, in that order.
- For transactions that contain multiple operations, operations are applied in the order in which they were received on the primary replica from the user.
- As part of processing false progress, some operations might be undone. Notifications are raised for such undo

operations, rolling the state of the replica back to a stable point. One important difference of undo notifications is that events that have duplicate keys are aggregated. For example, if transaction T1 is being undone, you'll see a single notification to Delete(X).

Next steps

- Reliable Collections
- Reliable Services quick start
- Reliable Services backup and restore (disaster recovery)
- Developer reference for Reliable Collections

Back up and restore Reliable Services and Reliable Actors

11/15/2016 • 13 min to read • Edit on GitHub

Contributors

Mert Coskun • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Wike • Sean McKenna • Jesse Benson • Matt Barry • Mani Ramaswamy

Azure Service Fabric is a high-availability platform that replicates the state across multiple nodes to maintain this high availability. Thus, even if one node in the cluster fails, the services continue to be available. While this in-built redundancy provided by the platform may be sufficient for some, in certain cases it is desirable for the service to back up data (to an external store).

> **NOTE**
> It is critical to backup and restore your data (and test that it works as expected) so you can recover from data loss scenarios.

For example, a service may want to back up data in the following scenarios:

- In the event of the permanent loss of an entire Service Fabric cluster or all nodes that are running a given partition.
- Administrative errors whereby the state accidentally gets deleted or corrupted. For example, this may happen if an administrator with sufficient privilege erroneously deletes the service.
- Bugs in the service that cause data corruption. For example, this may happen when a service code upgrade starts writing faulty data to a Reliable Collection. In such a case, both the code and the data may have to be reverted to an earlier state.
- Offline data processing. It might be convenient to have offline processing of data for business intelligence that happens separately from the service that generates the data.

The Backup/Restore feature allows services built on the Reliable Services API to create and restore backups. The backup APIs provided by the platform allow to take backup(s) of a service partition's state, without blocking read or write operations. The restore APIs allow a service partition's state to be restored from a chosen backup.

Types of Backup

There are two backup options: Full and Incremental. A full backup is a backup that contains all the data required to recreate the state of the replica: checkpoints and all log records. Since it has the checkpoints and the log, a full backup can be restored by itself.

The problem with full backups arises when the checkpoints are large. For example, a replica that has 16 GB of state will have checkpoints that add up approximately to 16 GB. If we have an Recovery Point Objective of 5 minutes, the replica needs to be backed up every 5 minutes. Each time it backs up it will need to copy 16 GB of checkpoints in addition to 50 MB (configurable using **CheckpointThresholdInMB**) worth of logs.

The solution to this problem is incremental backups, where only the log records since the last backup are backed up.

Since incremental backups are only changes since the last backup (does not include the checkpoints), they tend to be faster but they cannot be restored on their own. To restore an incremental backup, the entire backup chain is required. A backup chain is a chain of backups starting with a full backup and followed by a number of contiguous incremental backups.

Backup Reliable Services

The service author has full control of when to make backups and where backups will be stored.

To start a backup, the service needs to invoke the inherited member function **BackupAsync**.
Backups can be made only from primary replicas, and they require write status to be granted.

As shown below, **BackupAsync** takes in a **BackupDescription** object, where one can specify a full or incremental backup, as well as a callback function, **Func<< BackupInfo, CancellationToken, Task>>** which is invoked when the backup folder has been created locally and is ready to be moved out to some external storage.

```
BackupDescription myBackupDescription = new
BackupDescription(backupOption.Incremental,this.BackupCallbackAsync);

await this.BackupAsync(myBackupDescription);
```

Request to take an incremental backup can fail with **FabricMissingFullBackupException** which indicates that either

- the replica has never taken a full backup since it has become primary,
- some of the log records since the last backup has been truncated or
- replica passed the **MaxAccumulatedBackupLogSizeInMB** limit.

Users can increase the likelihood of being able to do incremental backups by configuring **MinLogSizeInMB** or **TruncationThresholdFactor**. Note that increasing these values will increase the per replica disk usage. For more information, see Reliable Services Configuration

BackupInfo provides information regarding the backup, including the location of the folder where the runtime saved the backup (**BackupInfo.Directory**). The callback function can move the **BackupInfo.Directory** to an external store or another location. This function also returns a bool that indicates whether it was able to successfully move the backup folder to its target location.

The following code demonstrates how the **BackupCallbackAsync** method can be used to upload the backup to

Azure Storage:

```
private async Task<bool> BackupCallbackAsync(BackupInfo backupInfo, CancellationToken cancellationToken)
{
    var backupId = Guid.NewGuid();

    await externalBackupStore.UploadBackupFolderAsync(backupInfo.Directory, backupId, cancellationToken);

    return true;
}
```

In the above example, **ExternalBackupStore** is the sample class that is used to interface with Azure Blob storage, and **UploadBackupFolderAsync** is the method that compresses the folder and places it in the Azure Blob store.

Note that:

- There can be only one backup operation in-flight per replica at any given time. More than one **BackupAsync** call at a time will throw **FabricBackupInProgressException** to limit inflight backups to one.
- If a replica fails over while a backup is in progress, the backup may not have been completed. Thus, once the failover finishes, it is the service's responsibility to restart the backup by invoking **BackupAsync** as necessary.

Restore Reliable Services

In general, the cases when you might need to perform a restore operation fall into one of these categories:

- The service partition lost data. For example, the disk for two out of three replicas for a partition (including the primary replica) gets corrupted or wiped. The new primary may need to restore data from a backup.
- The entire service is lost. For example, an administrator removes the entire service and thus the service and the data need to be restored.
- The service replicated corrupt application data (e.g., because of an application bug). In this case, the service has to be upgraded or reverted to remove the cause of the corruption, and non-corrupt data has to be restored.

While many approaches are possible, we offer some examples on using **RestoreAsync** to recover from the above scenarios.

Partition data loss in Reliable Services

In this case, the runtime would automatically detect the data loss and invoke the **OnDataLossAsync** API.

The service author needs to perform the following to recover:

- Override the virtual base class method **OnDataLossAsync**.
- Find the latest backup in the external location that contains the service's backups.
- Download the latest backup (and uncompress the backup into the backup folder if it was compressed).
- The **OnDataLossAsync** method provides a **RestoreContext**. Call the **RestoreAsync** API on the provided **RestoreContext**.
- Return true if the restoration was a success.

Following is an example implementation of the **OnDataLossAsync** method:

```
protected override async Task<bool> OnDataLossAsync(RestoreContext restoreCtx, CancellationToken
cancellationToken)
{
    var backupFolder = await this.externalBackupStore.DownloadLastBackupAsync(cancellationToken);

    var restoreDescription = new RestoreDescription(backupFolder);

    await restoreCtx.RestoreAsync(restoreDescription);

    return true;
}
```

RestoreDescription passed in to the **RestoreContext.RestoreAsync** call contains a member called **BackupFolderPath**. When restoring a single full backup, this **BackupFolderPath** should be set to the local path of the folder that contains your full backup. When restoring a full backup and a number of incremental backups, **BackupFolderPath** should be set to the local path of the folder that not only contains the full backup, but also all the incremental backups. **RestoreAsync** call can throw **FabricMissingFullBackupException** if the **BackupFolderPath** provided does not contain a full backup. It can also throw **ArgumentException** if **BackupFolderPath** has a broken chain of incremental backups. For example, if it contains the full backup, the first incremental and the third incremental backup but no the second incremental backup.

> **NOTE**
> The RestorePolicy is set to Safe by default. This means that the **RestoreAsync** API will fail with ArgumentException if it detects that the backup folder contains a state that is older than or equal to the state contained in this replica. **RestorePolicy.Force** can be used to skip this safety check. This is specified as part of **RestoreDescription**.

Deleted or lost service

If a service is removed, you must first re-create the service before the data can be restored. It is important to create the service with the same configuration, e.g., partitioning scheme, so that the data can be restored seamlessly. Once the service is up, the API to restore data (**OnDataLossAsync** above) has to be invoked on every partition of this service. One way of achieving this is by using **FabricClient.TestManagementClient.StartPartitionDataLossAsync** on every partition.

From this point, implementation is the same as the above scenario. Each partition needs to restore the latest relevant backup from the external store. One caveat is that the partition ID may have now changed, since the runtime creates partition IDs dynamically. Thus, the service needs to store the appropriate partition information and service name to identify the correct latest backup to restore from for each partition.

> **NOTE**
> It is not recommended to use **FabricClient.ServiceManager.InvokeDataLossAsync** on each partition to restore the entire service, since that may corrupt your cluster state.

Replication of corrupt application data

If the newly deployed application upgrade has a bug, that may cause corruption of data. For example, an application upgrade may start to update every phone number record in a Reliable Dictionary with an invalid area code. In this case, the invalid phone numbers will be replicated since Service Fabric is not aware of the nature of the data that is being stored.

The first thing to do after you detect such an egregious bug that causes data corruption is to freeze the service at the application level and, if possible, upgrade to the version of the application code that does not have the bug.

However, even after the service code is fixed, the data may still be corrupt and thus data may need to be restored. In such cases, it may not be sufficient to restore the latest backup, since the latest backups may also be corrupt. Thus, you have to find the last backup that was made before the data got corrupted.

If you are not sure which backups are corrupt, you could deploy a new Service Fabric cluster and restore the backups of affected partitions just like the above "Deleted or lost service" scenario. For each partition, start restoring the backups from the most recent to the least. Once you find a backup that does not have the corruption, move/delete all backups of this partition that were more recent (than that backup). Repeat this process for each partition. Now, when **OnDataLossAsync** is called on the partition in the production cluster, the last backup found in the external store will be the one picked by the above process.

Now, steps in the "Deleted or lost service" section can be used to restore the state of the service to the state before the buggy code corrupted the state.

Note that:

- When you restore, there is a chance that the backup being restored is older than the state of the partition before the data was lost. Because of this, you should restore only as a last resort to recover as much data as possible.
- The string that represents the backup folder path and the paths of files inside the backup folder can be greater than 255 characters, depending on the FabricDataRoot path and Application Type name's length. This can cause some .NET methods, like **Directory.Move**, to throw the **PathTooLongException** exception. One workaround is to directly call kernel32 APIs, like **CopyFile**.

Backup and restore Reliable Actors

Backup and restore for Reliable Actors builds on the backup and restore functionality provided by reliable services. The service owner should create a custom Actor service that derives from **ActorService** (which is a Service Fabric reliable service hosting actors) and then do backup/restore similar to reliable services as described above in previous sections. Since backups will be taken on a per-partition basis, states for all actors in that particular partition will be backed up (and restoration is similar and will happen on a per-partition basis).

- When you create a custom actor service, you will need to register the custom actor service as well while registering the actor. See **ActorRuntime.RegistorActorAsync**.
- The **KvsActorStateProvider** currently only supports full backup. Also the option **RestorePolicy.Safe** is ignored by the **KvsActorStateProvider**.

> **NOTE**
> The default ActorStateProvider (i.e., **KvsActorStateProvider**) **does not** cleanup the backup folders by itself (under the application work folder obtained through ICodePackageActivationContext.WorkDirectory). This may cause your work folder to get filled up. You should explicitly cleanup the backup folder in the backup callback after you have moved the backup to an external storage.

Testing Backup and Restore

It is important to ensure that critical data is being backed up, and can be restored from. This can be done by invoking the **Invoke-ServiceFabricPartitionDataLoss** cmdlet in PowerShell that can induce data loss in a particular partition to test whether the data backup and restore functionality for your service is working as expected. It is also possible to programmatically invoke data loss and restore from that event as well.

> **NOTE**
> You can find a sample implementation of backup and restore functionality in the Web Reference App on Github. Please look at the Inventory.Service service for more details.

Under the hood: more details on backup and restore

Here's some more details on backup and restore.

Backup

The Reliable State Manager provides the ability to create consistent backups without blocking any read or write operations. To do so, it utilizes a checkpoint and log persistence mechanism. The Reliable State Manager takes fuzzy (lightweight) checkpoints at certain points to relieve pressure from the transactional log and improve recovery times. When **BackupAsync** is called, the Reliable State Manager instructs all Reliable objects to copy their latest checkpoint files to a local backup folder. Then, the Reliable State Manager copies all log records, starting from the "start pointer" to the latest log record into the backup folder. Since all the log records up to the latest log record are included in the backup and the Reliable State Manager preserves write-ahead logging, the Reliable State Manager guarantees that all transactions that are committed (**CommitAsync** has returned successfully) are included in the backup.

Any transaction that commits after **BackupAsync** has been called may or may not be in the backup. Once the local backup folder has been populated by the platform (i.e., local backup is completed by the runtime), the service's backup callback is invoked. This callback is responsible for moving the backup folder to an external location such as Azure Storage.

Restore

The Reliable State Manager provides the ability to restore from a backup by using the **RestoreAsync** API.
The **RestoreAsync** method on **RestoreContext** can be called only inside the **OnDataLossAsync** method. The bool returned by **OnDataLossAsync** indicates whether the service restored its state from an external source. If the **OnDataLossAsync** returns true, Service Fabric will rebuild all other replicas from this primary. Service Fabric ensures that replicas that will receive **OnDataLossAsync** call first transition to the primary role but are not granted read status or write status. This implies that for StatefulService implementers, **RunAsync** will not be called until **OnDataLossAsync** finishes successfully. Then, **OnDataLossAsync** will be invoked on the new primary. Until a service completes this API successfully (by returning true or false) and finishes the relevant reconfiguration, the API will keep being called one at a time.

RestoreAsync first drops all existing state in the primary replica that it was called on.
Then the Reliable State Manager creates all the Reliable objects that exist in the backup folder.
Next, the Reliable objects are instructed to restore from their checkpoints in the backup folder.
Finally, the Reliable State Manager recovers its own state from the log records in the backup folder and performs recovery.
As part of the recovery process, operations starting from the "starting point" that have commit log records in the backup folder are replayed to the Reliable objects.
This step ensures that the recovered state is consistent.

Next steps

- Reliable Collections
- Reliable Services quick start
- Reliable Services notifications
- Reliable Services configuration
- Developer reference for Reliable Collections

How to use the Reliable Services communication APIs

11/15/2016 • 9 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • henrybeen • MarkDeichmiller • Andy De George • Jesse Benson • BharatNarasimman

Azure Service Fabric as a platform is completely agnostic about communication between services. All protocols and stacks are acceptable, from UDP to HTTP. It's up to the service developer to choose how services should communicate. The Reliable Services application framework provides built-in communication stacks as well as APIs that you can use to build your custom communication components.

Set up service communication

The Reliable Services API uses a simple interface for service communication. To open an endpoint for your service, simply implement this interface:

```
public interface ICommunicationListener
{
    Task<string> OpenAsync(CancellationToken cancellationToken);

    Task CloseAsync(CancellationToken cancellationToken);

    void Abort();
}
```

You can then add your communication listener implementation by returning it in a service-based class method override.

For stateless services:

```
class MyStatelessService : StatelessService
{
    protected override IEnumerable<ServiceInstanceListener> CreateServiceInstanceListeners()
    {
        ...
    }
    ...
}
```

For stateful services:

```
class MyStatefulService : StatefulService
{
    protected override IEnumerable<ServiceReplicaListener> CreateServiceReplicaListeners()
    {
        ...
    }
    ...
}
```

In both cases, you return a collection of listeners. This allows your service to listen on multiple endpoints, potentially using different protocols, by using multiple listeners. For example, you may have an HTTP listener and a

separate WebSocket listener. Each listener gets a name, and the resulting collection of *name : address* pairs is represented as a JSON object when a client requests the listening addresses for a service instance or a partition.

In a stateless service, the override returns a collection of ServiceInstanceListeners. A ServiceInstanceListener contains a function to create an ICommunicationListener and gives it a name. For stateful services, the override returns a collection of ServiceReplicaListeners. This is slightly different from its stateless counterpart, because a ServiceReplicaListener has an option to open an ICommunicationListener on secondary replicas. Not only can you use multiple communication listeners in a service, but you can also specify which listeners accept requests on secondary replicas and which ones listen only on primary replicas.

For example, you can have a ServiceRemotingListener that takes RPC calls only on primary replicas, and a second, custom listener that takes read requests on secondary replicas over HTTP:

```
protected override IEnumerable<ServiceReplicaListener> CreateServiceReplicaListeners()
{
    return new[]
    {
        new ServiceReplicaListener(context =>
            new MyCustomHttpListener(context),
            "HTTPReadonlyEndpoint",
            true),

        new ServiceReplicaListener(context =>
            this.CreateServiceRemotingListener(context),
            "rpcPrimaryEndpoint",
            false)
    };
}
```

> **NOTE**
>
> When creating multiple listeners for a service, each listener **must** be given a unique name.

Finally, describe the endpoints that are required for the service in the service manifest under the section on endpoints.

```
<Resources>
    <Endpoints>
      <Endpoint Name="WebServiceEndpoint" Protocol="http" Port="80" />
      <Endpoint Name="OtherServiceEndpoint" Protocol="tcp" Port="8505" />
    <Endpoints>
</Resources>
```

The communication listener can access the endpoint resources allocated to it from the `CodePackageActivationContext` in the `ServiceContext`. The listener can then start listening for requests when it is opened.

```
var codePackageActivationContext = serviceContext.CodePackageActivationContext;
var port = codePackageActivationContext.GetEndpoint("ServiceEndpoint").Port;
```

> **NOTE**
>
> Endpoint resources are common to the entire service package, and they are allocated by Service Fabric when the service package is activated. Multiple service replicas hosted in the same ServiceHost may share the same port. This means that the communication listener should support port sharing. The recommended way of doing this is for the communication listener to use the partition ID and replica/instance ID when it generates the listen address.

Service address registration

A system service called the *Naming Service* runs on Service Fabric clusters. The Naming Service is a registrar for services and their addresses that each instance or replica of the service is listening on. When the `OpenAsync` method of an `ICommunicationListener` completes, its return value gets registered in the Naming Service. This return value that gets published in the Naming Service is a string whose value can be anything at all. This string value is what clients will see when they ask for an address for the service from the Naming Service.

```
public Task<string> OpenAsync(CancellationToken cancellationToken)
{
    EndpointResourceDescription serviceEndpoint =
serviceContext.CodePackageActivationContext.GetEndpoint("ServiceEndpoint");
    int port = serviceEndpoint.Port;

    this.listeningAddress = string.Format(
                CultureInfo.InvariantCulture,
                "http://+:{0}/",
                port);

    this.publishAddress = this.listeningAddress.Replace("+", FabricRuntime.GetNodeContext().IPAddressOrFQDN);

    this.webApp = WebApp.Start(this.listeningAddress, appBuilder => this.startup.Invoke(appBuilder));

    // the string returned here will be published in the Naming Service.
    return Task.FromResult(this.publishAddress);
}
```

Service Fabric provides APIs that allows clients and other services to then ask for this address by service name. This is important because the service address is not static. Services are moved around in the cluster for resource balancing and availability purposes. This is the mechanism that allows clients to resolve the listening address for a service.

> **NOTE**
>
> For a complete walk-through of how to write an `ICommunicationListener`, see Service Fabric Web API services with OWIN self-hosting

Communicating with a service

The Reliable Services API provides the following libraries to write clients that communicate with services.

Service endpoint resolution

The first step to communication with a service is to resolve an endpoint address of the partition or instance of the service you want to talk to. The `ServicePartitionResolver` utility class is a basic primitive that helps clients determine the endpoint of a service at runtime. In Service Fabric terminology, the process of determining the endpoint of a service is referred to as the *service endpoint resolution*.

To connect to services within a cluster, `ServicePartitionResolver` can be created using default settings. This is the recommended usage for most situations:

```
ServicePartitionResolver resolver = ServicePartitionResolver.GetDefault();
```

To connect to services in a different cluster, a `ServicePartitionResolver` can be created with a set of cluster gateway endpoints. Note that gateway endpoints are just different endpoints for connecting to the same cluster. For example:

```
ServicePartitionResolver resolver = new  ServicePartitionResolver("mycluster.cloudapp.azure.com:19000",
"mycluster.cloudapp.azure.com:19001");
```

Alternatively, `ServicePartitionResolver` can be given a function for creating a `FabricClient` to use internally:

```
public delegate FabricClient CreateFabricClientDelegate();
```

`FabricClient` is the object that is used to communicate with the Service Fabric cluster for various management operations on the cluster. This is useful when you want more control over how `ServicePartitionResolver` interacts with your cluster. `FabricClient` performs caching internally and is generally expensive to create, so it is important to reuse `FabricClient` instances as much as possible.

```
ServicePartitionResolver resolver = new  ServicePartitionResolver(() => CreateMyFabricClient());
```

A resolve method is then used to retrieve the address of a service or a service partition for partitioned services.

```
ServicePartitionResolver resolver = ServicePartitionResolver.GetDefault();

ResolvedServicePartition partition =
    await resolver.ResolveAsync(new Uri("fabric:/MyApp/MyService"), new ServicePartitionKey(),
cancellationToken);
```

A service address can be resolved easily using a `ServicePartitionResolver`, but more work is required to ensure the resolved address can be used correctly. Your client will need to detect whether the connection attempt failed because of a transient error and can be retried (e.g., service moved or is temporarily unavailable), or a permanent error (e.g., service was deleted or the requested resource no longer exists). Service instances or replicas can move around from node to node at any time for multiple reasons. The service address resolved through `ServicePartitionResolver` may be stale by the time your client code attempts to connect. In that case again the client will need to re-resolve the address. Providing the previous `ResolvedServicePartition` indicates that the resolver needs to try again rather than simply retrieve a cached address.

Typically, the client code need not work with the `ServicePartitionResolver` directly. It is created and passed on to communication client factories in the Reliable Services API. The factories use the resolver internally to generate a client object that can be used to communicate with services.

Communication clients and factories

The communication factory library implements a typical fault-handling retry pattern that makes retrying connections to resolved service endpoints easier. The factory library provides the retry mechanism while you provide the error handlers.

`ICommunicationClientFactory` defines the base interface implemented by a communication client factory that produces clients that can talk to a Service Fabric service. The implementation of the CommunicationClientFactory depends on the communication stack used by the Service Fabric service where the client wants to communicate. The Reliable Services API provides a `CommunicationClientFactoryBase<TCommunicationClient>`. This provides a base implementation of the `ICommunicationClientFactory` interface and performs tasks that are common to all the communication stacks. (These tasks include using a `ServicePartitionResolver` to determine the service endpoint). Clients usually implement the abstract CommunicationClientFactoryBase class to handle logic that is specific to the communication stack.

The communication client just receives an address and uses it to connect to a service. The client can use whatever protocol it wants.

```
class MyCommunicationClient : ICommunicationClient
{
    public ResolvedServiceEndpoint Endpoint { get; set; }

    public string ListenerName { get; set; }

    public ResolvedServicePartition ResolvedServicePartition { get; set; }
}
```

The client factory is primarily responsible for creating communication clients. For clients that don't maintain a persistent connection, such as an HTTP client, the factory only needs to create and return the client. Other protocols that maintain a persistent connection, such as some binary protocols, should also be validated by the factory to determine whether or not the connection needs to be re-created.

```
public class MyCommunicationClientFactory : CommunicationClientFactoryBase<MyCommunicationClient>
{
    protected override void AbortClient(MyCommunicationClient client)
    {
    }

    protected override Task<MyCommunicationClient> CreateClientAsync(string endpoint, CancellationToken
cancellationToken)
    {
    }

    protected override bool ValidateClient(MyCommunicationClient clientChannel)
    {
    }

    protected override bool ValidateClient(string endpoint, MyCommunicationClient client)
    {
    }
}
```

Finally, an exception handler is reponsible for determining what action to take when an exception occurs. Exceptions are categorized into **retriable** and **non retriable**.

- **Non retriable** exceptions simply get re-thrown back to the caller.
- **Retriable** exceptions are further categorized into **transient** and **non-transient**.
  - **Transient** exceptions are those that can simply be retried without re-resolving the service endpoint address. These will include transient network problems or service error responses other than those that indicate the service endpoint address does not exist.
  - **Non-transient** exceptions are those that require the service endpoint address to be re-resolved. These include exceptions that indicate the service endpoint could not be reached, indicating the service has moved to a different node.

The `TryHandleException` makes a decision about a given exception. If it **does not know** what decisions to make about an exception, it should return **false**. If it **does know** what decision to make, it should set the result accordingly and return **true**.

```
class MyExceptionHandler : IExceptionHandler
{
    public bool TryHandleException(ExceptionInformation exceptionInformation, OperationRetrySettings
retrySettings, out ExceptionHandlingResult result)
    {
        // if exceptionInformation.Exception is known and is transient (can be retried without re-resolving)
        result = new ExceptionHandlingRetryResult(exceptionInformation.Exception, true, retrySettings,
retrySettings.DefaultMaxRetryCount);
        return true;


        // if exceptionInformation.Exception is known and is not transient (indicates a new service endpoint
address must be resolved)
        result = new ExceptionHandlingRetryResult(exceptionInformation.Exception, false, retrySettings,
retrySettings.DefaultMaxRetryCount);
        return true;

        // if exceptionInformation.Exception is unknown (let the next IExceptionHandler attempt to handle it)
        result = null;
        return false;
    }
}
```

Putting it all together

With an `ICommunicationClient`, `ICommunicationClientFactory`, and `IExceptionHandler` built around a communication protocol, a `ServicePartitionClient` is wraps it all together and provides the fault-handling and service partition address resolution loop around these components.

```
private MyCommunicationClientFactory myCommunicationClientFactory;
private Uri myServiceUri;

var myServicePartitionClient = new ServicePartitionClient<MyCommunicationClient>(
    this.myCommunicationClientFactory,
    this.myServiceUri,
    myPartitionKey);

var result = await myServicePartitionClient.InvokeWithRetryAsync(async (client) =>
   {
      // Communicate with the service using the client.
   },
   CancellationToken.None);
```

Next steps

- See an example of HTTP communication between services in a sample project on GitHUb.
- Remote procedure calls with Reliable Services remoting
- Web API that uses OWIN in Reliable Services
- WCF communication by using Reliable Services

Get started: Service Fabric Web API services with OWIN self-hosting

11/15/2016 • 15 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • katiecumming • CJM81 • Sjoerd Nijhof • Matt Kotsenas • MarkDeichmiller • Ryan Wike • Sean McKenna

Azure Service Fabric puts the power in your hands when you're deciding how you want your services to communicate with users and with each other. This tutorial focuses on implementing service communication using ASP.NET Web API with Open Web Interface for .NET (OWIN) self-hosting in Service Fabric's Reliable Services API. We'll delve deeply into the Reliable Services pluggable communication API. We'll also use Web API in a step-by-step example to show you how to set up a custom communication listener.

Introduction to Web API in Service Fabric

ASP.NET Web API is a popular and powerful framework for building HTTP APIs on top of the .NET Framework. If you're not already familiar with the framework, see Getting started with ASP.NET Web API 2 to learn more.

Web API in Service Fabric is the same ASP.NET Web API you know and love. The difference is in how you *host* a Web API application. You won't be using Microsoft Internet Information Services (IIS). To better understand the difference, let's break it into two parts:

1. The Web API application (including controllers and models)
2. The host (the web server, usually IIS)

A Web API application itself doesn't change. It's no different from Web API applications you may have written in the past, and you should be able to simply move over most of your application code. But if you've been hosting on IIS, where you host the application may be a little different from what you're used to. Before we get to the hosting part, let's start with something more familiar: the Web API application.

Create the application

Start by creating a new Service Fabric application with a single stateless service in Visual Studio 2015:

A Visual Studio template for a stateless service using Web API is available to you. In this tutorial, we'll build a Web API project from scratch that results in what you'd get if you selected this template.

Select a blank Stateless Service project to learn how to build a Web API project from scratch, or you can start with the stateless service Web API template and simply follow along.

The first step is to pull in some NuGet packages for Web API. The package we want to use is

Microsoft.AspNet.WebApi.OwinSelfHost. This package includes all the necessary Web API packages and the *host* packages. This will be important later.

After the packages have been installed, you can begin building out the basic Web API project structure. If you've used Web API, the project structure should look very familiar. Start by adding a `Controllers` directory and a simple values controller:

ValuesController.cs

```
using System.Collections.Generic;
using System.Web.Http;

namespace WebService.Controllers
{
    public class ValuesController : ApiController
    {
        // GET api/values
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET api/values/5
        public string Get(int id)
        {
            return "value";
        }

        // POST api/values
        public void Post([FromBody]string value)
        {
        }

        // PUT api/values/5
        public void Put(int id, [FromBody]string value)
        {
        }

        // DELETE api/values/5
        public void Delete(int id)
        {
        }
    }
}
```

Next, add a Startup class at the project root to register the routing, formatters, and any other configuration setup. This is also where Web API plugs in to the *host*, which will be revisited again later.

Startup.cs

```
using System.Web.Http;
using Owin;

namespace WebService
{
    public static class Startup
    {
        public static void ConfigureApp(IAppBuilder appBuilder)
        {
            // Configure Web API for self-host.
            HttpConfiguration config = new HttpConfiguration();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );

            appBuilder.UseWebApi(config);
        }
    }
}
```

That's it for the application part. At this point, we've set up just the basic Web API project layout. So far, it

shouldn't look much different from Web API projects you may have written in the past or from the basic Web API template. Your business logic goes in the controllers and models as usual.

Now what do we do about hosting so that we can actually run it?

Service hosting

In Service Fabric, your service runs in a *service host process*, an executable file that runs your service code. When you write a service by using the Reliable Services API, your service project just compiles to an executable file that registers your service type and runs your code. This is true in most cases when you write a service on Service Fabric in .NET. When you open Program.cs in the stateless service project, you should see:

```
using System;
using System.Diagnostics;
using System.Threading;
using Microsoft.ServiceFabric.Services.Runtime;

internal static class Program
{
    private static void Main()
    {
        try
        {
            ServiceRuntime.RegisterServiceAsync("WebServiceType",
                context => new WebService(context)).GetAwaiter().GetResult();

            ServiceEventSource.Current.ServiceTypeRegistered(Process.GetCurrentProcess().Id,
typeof(WebService).Name);

            // Prevents this host process from terminating so services keeps running.
            Thread.Sleep(Timeout.Infinite);
        }
        catch (Exception e)
        {
            ServiceEventSource.Current.ServiceHostInitializationFailed(e.ToString());
            throw;
        }
    }
}
```

If that looks suspiciously like the entry point to a console application, that's because it is.

Further details about the service host process and service registration are beyond the scope of this article. But it's important to know for now that *your service code is running in its own process*.

Self-host Web API with an OWIN host

Given that your Web API application code is hosted in its own process, how do you hook it up to a web server? Enter OWIN. OWIN is simply a contract between .NET web applications and web servers. Traditionally when ASP.NET (up to MVC 5) is used, the web application is tightly coupled to IIS through System.Web. However, Web API implements OWIN, so you can write a web application that is decoupled from the web server that hosts it. Because of this, you can use a *self-hosted* OWIN web server that you can start in your own process. This fits perfectly with the Service Fabric hosting model we just described.

In this article, we'll use Katana as the OWIN host for the Web API application. Katana is an open-source OWIN host implementation built on System.Net.HttpListener and the Windows HTTP Server API.

> **NOTE**
> To learn more about Katana, go to the Katana site. For a quick overview of how to use Katana to self-host Web API, see Use OWIN to Self-Host ASP.NET Web API 2.

Set up the web server

The Reliable Services API provides a communication entry point where you can plug in communication stacks that allow users and clients to connect to the service:

```
protected override IEnumerable<ServiceInstanceListener> CreateServiceInstanceListeners()
{
    ...
}
```

The web server (and any other communication stack you use in the future, such as WebSockets) should use the ICommunicationListener interface to integrate correctly with the system. The reasons for this will become more apparent in the following steps.

First, create a class called OwinCommunicationListener that implements ICommunicationListener:

OwinCommunicationListener.cs

```
using Microsoft.Owin.Hosting;
using Microsoft.ServiceFabric.Services.Communication.Runtime;
using Owin;
using System;
using System.Fabric;
using System.Globalization;
using System.Threading;
using System.Threading.Tasks;

namespace WebService
{
    internal class OwinCommunicationListener : ICommunicationListener
    {
        public void Abort()
        {
        }

        public Task CloseAsync(CancellationToken cancellationToken)
        {
        }

        public Task<string> OpenAsync(CancellationToken cancellationToken)
        {
        }
    }
}
```

The ICommunicationListener interface provides three methods to manage a communication listener for your service:

- *OpenAsync*. Start listening for requests.
- *CloseAsync*. Stop listening for requests, finish any in-flight requests, and shut down gracefully.
- *Abort*. Cancel everything and stop immediately.

To get started, add private class members for things the listener will need to function. These will be initialized through the constructor and used later when you set up the listening URL.

```
internal class OwinCommunicationListener : ICommunicationListener
{
    private readonly ServiceEventSource eventSource;
    private readonly Action<IAppBuilder> startup;
    private readonly ServiceContext serviceContext;
    private readonly string endpointName;
    private readonly string appRoot;

    private IDisposable webApp;
    private string publishAddress;
    private string listeningAddress;

    public OwinCommunicationListener(Action<IAppBuilder> startup, ServiceContext serviceContext,
ServiceEventSource eventSource, string endpointName)
        : this(startup, serviceContext, eventSource, endpointName, null)
    {
    }

    public OwinCommunicationListener(Action<IAppBuilder> startup, ServiceContext serviceContext,
ServiceEventSource eventSource, string endpointName, string appRoot)
    {
        if (startup == null)
        {
            throw new ArgumentNullException(nameof(startup));
        }

        if (serviceContext == null)
        {
            throw new ArgumentNullException(nameof(serviceContext));
        }

        if (endpointName == null)
        {
            throw new ArgumentNullException(nameof(endpointName));
        }

        if (eventSource == null)
        {
            throw new ArgumentNullException(nameof(eventSource));
        }

        this.startup = startup;
        this.serviceContext = serviceContext;
        this.endpointName = endpointName;
        this.eventSource = eventSource;
        this.appRoot = appRoot;
    }


    ...
```

Implement OpenAsync

To set up the web server, you need two pieces of information:

- *A URL path prefix*. Although this is optional, it's good for you to set this up now so that you can safely host multiple web services in your application.
- *A port*.

Before you get a port for the web server, it's important that you understand that Service Fabric provides an application layer that acts as a buffer between your application and the underlying operating system that it runs on. As such, Service Fabric provides a way to configure *endpoints* for your services. Service Fabric ensures that endpoints are available for your service to use. This way, you don't have to configure them yourself in the underlying OS environment. You can easily host your Service Fabric application in different environments

without having to make any changes to your application. (For example, you can host the same application in Azure or in your own datacenter.)

Configure an HTTP endpoint in PackageRoot\ServiceManifest.xml:

```
<Resources>
    <Endpoints>
        <Endpoint Name="ServiceEndpoint" Type="Input" Protocol="http" Port="8281" />
    </Endpoints>
</Resources>
```

This step is important because the service host process runs under restricted credentials (Network Service on Windows). This means that your service won't have access to set up an HTTP endpoint on its own. By using the endpoint configuration, Service Fabric knows to set up the proper access control list (ACL) for the URL that the service will listen on. Service Fabric also provides a standard place to configure endpoints.

Back in OwinCommunicationListener.cs, you can start implementing OpenAsync. This is where you start the web server. First, get the endpoint information and create the URL that the service will listen on. The URL will be different depending on whether the listener is used in a stateless service or a stateful service. For a stateful service, the listener needs to create a unique address for every stateful service replica it listens on. For stateless services, the address can be much simpler.

```
public Task<string> OpenAsync(CancellationToken cancellationToken)
{
    var serviceEndpoint = this.serviceContext.CodePackageActivationContext.GetEndpoint(this.endpointName);
    var protocol = serviceEndpoint.Protocol;
    int port = serviceEndpoint.Port;

    if (this.serviceContext is StatefulServiceContext)
    {
        StatefulServiceContext statefulServiceContext = this.serviceContext as StatefulServiceContext;

        this.listeningAddress = string.Format(
            CultureInfo.InvariantCulture,
            "{0}://+:{1}/{2}{3}/{4}/{5}",
            protocol,
            port,
            string.IsNullOrWhiteSpace(this.appRoot)
                ? string.Empty
                : this.appRoot.TrimEnd('/') + '/',
            statefulServiceContext.PartitionId,
            statefulServiceContext.ReplicaId,
            Guid.NewGuid());
    }
    else if (this.serviceContext is StatelessServiceContext)
    {
        this.listeningAddress = string.Format(
            CultureInfo.InvariantCulture,
            "{0}://+:{1}/{2}",
            protocol,
            port,
            string.IsNullOrWhiteSpace(this.appRoot)
                ? string.Empty
                : this.appRoot.TrimEnd('/') + '/');
    }
    else
    {
        throw new InvalidOperationException();
    }

    ...
```

Note that "http://+" is used here. This is to make sure that the web server is listening on all available addresses, including localhost, FQDN, and the machine IP.

The OpenAsync implementation is one of the most important reasons why the web server (or any communication stack) is implemented as an ICommunicationListener, rather than just having it open directly from `RunAsync()` in the service. The return value from OpenAsync is the address that the web server is listening on. When this address is returned to the system, it registers the address with the service. Service Fabric provides an API that allows clients and other services to then ask for this address by service name. This is important because the service address is not static. Services are moved around in the cluster for resource balancing and availability purposes. This is the mechanism that allows clients to resolve the listening address for a service.

With that in mind, OpenAsync starts the web server and returns the address that it's listening on. Note that it listens on "http://+", but before OpenAsync returns the address, the "+" is replaced with the IP or FQDN of the node it is currently on. The address that is returned by this method is what's registered with the system. It's also what clients and other services see when they ask for a service's address. For clients to correctly connect to it, they need an actual IP or FQDN in the address.

```
    ...

    this.publishAddress = this.listeningAddress.Replace("+", FabricRuntime.GetNodeContext().IPAddressOrFQDN);

    try
    {
        this.eventSource.Message("Starting web server on " + this.listeningAddress);

        this.webApp = WebApp.Start(this.listeningAddress, appBuilder => this.startup.Invoke(appBuilder));

        this.eventSource.Message("Listening on " + this.publishAddress);

        return Task.FromResult(this.publishAddress);
    }
    catch (Exception ex)
    {
        this.eventSource.Message("Web server failed to open endpoint {0}. {1}", this.endpointName,
ex.ToString());

        this.StopWebServer();

        throw;
    }
}
```

Note that this references the Startup class that was passed in to the OwinCommunicationListener in the constructor. This startup instance is used by the web server to bootstrap the Web API application.

The `ServiceEventSource.Current.Message()` line will appear in the Diagnostic Events window later, when you run the application to confirm that the web server has started successfully.

Implement CloseAsync and Abort

Finally, implement both CloseAsync and Abort to stop the web server. The web server can be stopped by disposing the server handle that was created during OpenAsync.

```
public Task CloseAsync(CancellationToken cancellationToken)
{
    this.eventSource.Message("Closing web server on endpoint {0}", this.endpointName);

    this.StopWebServer();

    return Task.FromResult(true);
}

public void Abort()
{
    this.eventSource.Message("Aborting web server on endpoint {0}", this.endpointName);

    this.StopWebServer();
}

private void StopWebServer()
{
    if (this.webApp != null)
    {
        try
        {
            this.webApp.Dispose();
        }
        catch (ObjectDisposedException)
        {
            // no-op
        }
    }
}
```

In this implementation example, both CloseAsync and Abort simply stop the web server. You may opt to perform a more gracefully coordinated shutdown of the web server in CloseAsync. For example, the shutdown could wait for in-flight requests to be completed before the return.

Start the web server

You're now ready to create and return an instance of OwinCommunicationListener to start the web server. Back in the Service class (WebService.cs), override the `CreateServiceInstanceListeners()` method:

```
protected override IEnumerable<ServiceInstanceListener> CreateServiceInstanceListeners()
{
    var endpoints = Context.CodePackageActivationContext.GetEndpoints()
                            .Where(endpoint => endpoint.Protocol == EndpointProtocol.Http || endpoint.Protocol
== EndpointProtocol.Https)
                            .Select(endpoint => endpoint.Name);

    return endpoints.Select(endpoint => new ServiceInstanceListener(
        serviceContext => new OwinCommunicationListener(Startup.ConfigureApp, serviceContext,
ServiceEventSource.Current, endpoint), endpoint));
}
```

This is where the Web API *application* and the OWIN *host* finally meet. The host (OwinCommunicationListener) is given an instance of the *application* (Web API) via the Startup class. Service Fabric then manages its lifecycle. This same pattern can typically be followed with any communication stack.

Put it all together

In this example, you don't need to do anything in the `RunAsync()` method, so that override can simply be removed.

The final service implementation should be very simple. It only needs to create the communication listener:

```
using System;
using System.Collections.Generic;
using System.Fabric;
using System.Fabric.Description;
using System.Linq;
using Microsoft.ServiceFabric.Services.Communication.Runtime;
using Microsoft.ServiceFabric.Services.Runtime;

namespace WebService
{
    internal sealed class WebService : StatelessService
    {
        public WebService(StatelessServiceContext context)
            : base(context)
        { }

        protected override IEnumerable<ServiceInstanceListener> CreateServiceInstanceListeners()
        {
            var endpoints = Context.CodePackageActivationContext.GetEndpoints()
                                    .Where(endpoint => endpoint.Protocol == EndpointProtocol.Http ||
endpoint.Protocol == EndpointProtocol.Https)
                                    .Select(endpoint => endpoint.Name);

            return endpoints.Select(endpoint => new ServiceInstanceListener(
                serviceContext => new OwinCommunicationListener(Startup.ConfigureApp, serviceContext,
ServiceEventSource.Current, endpoint), endpoint));
        }
    }
}
```

The complete `OwinCommunicationListener` class:

```
using System;
using System.Diagnostics;
using System.Fabric;
using System.Globalization;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Owin.Hosting;
using Microsoft.ServiceFabric.Services.Communication.Runtime;
using Owin;

namespace WebService
{
    internal class OwinCommunicationListener : ICommunicationListener
    {
        private readonly ServiceEventSource eventSource;
        private readonly Action<IAppBuilder> startup;
        private readonly ServiceContext serviceContext;
        private readonly string endpointName;
        private readonly string appRoot;

        private IDisposable webApp;
        private string publishAddress;
        private string listeningAddress;

        public OwinCommunicationListener(Action<IAppBuilder> startup, ServiceContext serviceContext,
ServiceEventSource eventSource, string endpointName)
            : this(startup, serviceContext, eventSource, endpointName, null)
        {
        }

        public OwinCommunicationListener(Action<IAppBuilder> startup, ServiceContext serviceContext,
ServiceEventSource eventSource, string endpointName, string appRoot)
        {
            if (startup == null)
            {
```

```
                throw new ArgumentNullException(nameof(startup));
            }

            if (serviceContext == null)
            {
                throw new ArgumentNullException(nameof(serviceContext));
            }

            if (endpointName == null)
            {
                throw new ArgumentNullException(nameof(endpointName));
            }

            if (eventSource == null)
            {
                throw new ArgumentNullException(nameof(eventSource));
            }

            this.startup = startup;
            this.serviceContext = serviceContext;
            this.endpointName = endpointName;
            this.eventSource = eventSource;
            this.appRoot = appRoot;
        }

        public Task<string> OpenAsync(CancellationToken cancellationToken)
        {
            var serviceEndpoint =
this.serviceContext.CodePackageActivationContext.GetEndpoint(this.endpointName);
            var protocol = serviceEndpoint.Protocol;
            int port = serviceEndpoint.Port;

            if (this.serviceContext is StatefulServiceContext)
            {
                StatefulServiceContext statefulServiceContext = this.serviceContext as StatefulServiceContext;

                this.listeningAddress = string.Format(
                    CultureInfo.InvariantCulture,
                    "{0}://+:{1}/{2}{3}/{4}/{5}",
                    protocol,
                    port,
                    string.IsNullOrWhiteSpace(this.appRoot)
                        ? string.Empty
                        : this.appRoot.TrimEnd('/') + '/',
                    statefulServiceContext.PartitionId,
                    statefulServiceContext.ReplicaId,
                    Guid.NewGuid());
            }
            else if (this.serviceContext is StatelessServiceContext)
            {
                this.listeningAddress = string.Format(
                    CultureInfo.InvariantCulture,
                    "{0}://+:{1}/{2}",
                    protocol,
                    port,
                    string.IsNullOrWhiteSpace(this.appRoot)
                        ? string.Empty
                        : this.appRoot.TrimEnd('/') + '/');
            }
            else
            {
                throw new InvalidOperationException();
            }

            this.publishAddress = this.listeningAddress.Replace("+",
FabricRuntime.GetNodeContext().IPAddressOrFQDN);

            try
            {
```

```
            {
                this.eventSource.Message("Starting web server on " + this.listeningAddress);

                this.webApp = WebApp.Start(this.listeningAddress, appBuilder =>
this.startup.Invoke(appBuilder));

                this.eventSource.Message("Listening on " + this.publishAddress);

                return Task.FromResult(this.publishAddress);
            }
            catch (Exception ex)
            {
                this.eventSource.Message("Web server failed to open endpoint {0}. {1}", this.endpointName,
ex.ToString());

                this.StopWebServer();

                throw;
            }
        }

        public Task CloseAsync(CancellationToken cancellationToken)
        {
            this.eventSource.Message("Closing web server on endpoint {0}", this.endpointName);

            this.StopWebServer();

            return Task.FromResult(true);
        }

        public void Abort()
        {
            this.eventSource.Message("Aborting web server on endpoint {0}", this.endpointName);

            this.StopWebServer();
        }

        private void StopWebServer()
        {
            if (this.webApp != null)
            {
                try
                {
                    this.webApp.Dispose();
                }
                catch (ObjectDisposedException)
                {
                    // no-op
                }
            }
        }
    }
}
```

Now that you have put all the pieces in place, your project should look like a typical Web API application with Reliable Services API entry points and an OWIN host:

Run and connect through a web browser

If you haven't done so, set up your development environment.

You can now build and deploy your service. Press **F5** in Visual Studio to build and deploy the application. In the Diagnostic Events window, you should see a message that indicates that the web server opened on http://localhost:8281/.

> **NOTE**
>
> If the port has already been opened by another process on your machine, you may see an error here. This indicates that the listener couldn't be opened. If that's the case, try using a different port for the endpoint configuration in ServiceManifest.xml.

Once the service is running, open a browser and navigate to http://localhost:8281/api/values to test it.

Scale it out

Scaling out stateless web apps typically means adding more machines and spinning up the web apps on them. Service Fabric's orchestration engine can do this for you whenever new nodes are added to a cluster. When you create instances of a stateless service, you can specify the number of instances you want to create. Service Fabric places that number of instances on nodes in the cluster. And it makes sure not to create more than one instance on any one node. You can also instruct Service Fabric to always create an instance on every node by specifying **-1** for the instance count. This guarantees that whenever you add nodes to scale out your cluster, an instance of your stateless service will be created on the new nodes. This value is a property of the service instance, so it is set when you create a service instance. You can do this through PowerShell:

```
New-ServiceFabricService -ApplicationName "fabric:/WebServiceApplication" -ServiceName
"fabric:/WebServiceApplication/WebService" -ServiceTypeName "WebServiceType" -Stateless -
PartitionSchemeSingleton -InstanceCount -1
```

You can also do this when you define a default service in a Visual Studio stateless service project:

```
<DefaultServices>
  <Service Name="WebService">
    <StatelessService ServiceTypeName="WebServiceType" InstanceCount="-1">
      <SingletonPartition />
    </StatelessService>
  </Service>
</DefaultServices>
```

For more information on how to create application and service instances, see Deploy an application.

Next steps

Debug your Service Fabric application by using Visual Studio

Service remoting with Reliable Services

11/15/2016 • 1 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Michael Vlaar • Pushkar Newaskar • Matt Barry

For services that are not tied to a particular communication protocol or stack, such as WebAPI, Windows Communication Foundation (WCF), or others, the Reliable Services framework provides a remoting mechanism to quickly and easily set up remote procedure call for services.

Set up remoting on a service

Setting up remoting for a service is done in two simple steps:

1. Create an interface for your service to implement. This interface defines the methods that will be available for a remote procedure call on your service. The methods must be task-returning asynchronous methods. The interface must implement `Microsoft.ServiceFabric.Services.Remoting.IService` to signal that the service has a remoting interface.
2. Use a remoting listener in your service. This is an `ICommunicationListener` implementation that provides remoting capabilities. The `Microsoft.ServiceFabric.Services.Remoting.Runtime` namespace contains an extension method, `CreateServiceRemotingListener` for both stateless and stateful services that can be used to create a remoting listener using the default remoting transport protocol.

For example, the following stateless service exposes a single method to get "Hello World" over a remote procedure call.

```
using Microsoft.ServiceFabric.Services.Communication.Runtime;
using Microsoft.ServiceFabric.Services.Remoting;
using Microsoft.ServiceFabric.Services.Remoting.Runtime;
using Microsoft.ServiceFabric.Services.Runtime;

public interface IMyService : IService
{
    Task<string> HelloWorldAsync();
}

class MyService : StatelessService, IMyService
{
    public MyService(StatelessServiceContext context)
        : base (context)
    {
    }

    public Task HelloWorldAsync()
    {
        return Task.FromResult("Hello!");
    }

    protected override IEnumerable<ServiceInstanceListener> CreateServiceInstanceListeners()
    {
        return new[] { new ServiceInstanceListener(context =>
            this.CreateServiceRemotingListener(context)) };
    }
}
```

> **NOTE**
> The arguments and the return types in the service interface can be any simple, complex, or custom types, but they must be serializable by the .NET DataContractSerializer.

Call remote service methods

Calling methods on a service by using the remoting stack is done by using a local proxy to the service through the `Microsoft.ServiceFabric.Services.Remoting.Client.ServiceProxy` class. The `ServiceProxy` method creates a local proxy by using the same interface that the service implements. With that proxy, you can simply call methods on the interface remotely.

```
IMyService helloWorldClient = ServiceProxy.Create<IMyService>(new
Uri("fabric:/MyApplication/MyHelloWorldService"));

string message = await helloWorldClient.HelloWorldAsync();
```

The remoting framework propagates exceptions thrown at the service to the client. So exception-handling logic at the client by using `ServiceProxy` can directly handle exceptions that the service throws.

Next steps

- Web API with OWIN in Reliable Services
- WCF communication with Reliable Services
- Securing communication for Reliable Services

WCF-based communication stack for Reliable Services

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

BharatNarasimman • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Pushkar Newaskar • unknown • Nikhil13x • Matt Barry • Jesse Benson • Andy De George • Vaclav Turecek

The Reliable Services framework allows service authors to choose the communication stack that they want to use for their service. They can plug in the communication stack of their choice via the **ICommunicationListener** returned from the CreateServiceReplicaListeners or CreateServiceInstanceListeners methods. The framework provides an implementation of the communication stack based on the Windows Communication Foundation (WCF) for service authors who want to use WCF-based communication.

WCF Communication Listener

The WCF-specific implementation of **ICommunicationListener** is provided by the **Microsoft.ServiceFabric.Services.Communication.Wcf.Runtime.WcfCommunicationListener** class.

Lest say we have a service contract of type `ICalculator`

```
[ServiceContract]
public interface ICalculator
{
    [OperationContract]
    Task<int> Add(int value1, int value2);
}
```

We can create a WCF communication listener in the service the following manner.

```
protected override IEnumerable<ServiceReplicaListener> CreateServiceReplicaListeners()
{
    return new[] { new ServiceReplicaListener((context) =>
        new WcfCommunicationListener<ICalculator>(
            wcfServiceObject:this,
            serviceContext:context,
            //
            // The name of the endpoint configured in the ServiceManifest under the Endpoints section
            // that identifies the endpoint that the WCF ServiceHost should listen on.
            //
            endpointResourceName: "WcfServiceEndpoint",

            //
            // Populate the binding information that you want the service to use.
            //
            listenerBinding: WcfUtility.CreateTcpListenerBinding()
        )
    )};
}
```

Writing clients for the WCF communication stack

For writing clients to communicate with services by using WCF, the framework provides **WcfClientCommunicationFactory**, which is the WCF-specific implementation of ClientCommunicationFactoryBase.

```
public WcfCommunicationClientFactory(
    Binding clientBinding = null,
    IEnumerable<IExceptionHandler> exceptionHandlers = null,
    IServicePartitionResolver servicePartitionResolver = null,
    string traceId = null,
    object callback = null);
```

The WCF communication channel can be accessed from the **WcfCommunicationClient** created by the **WcfCommunicationClientFactory**.

```
public class WcfCommunicationClient : ServicePartitionClient<WcfCommunicationClient<ICalculator>>
   {
       public WcfCommunicationClient(ICommunicationClientFactory<WcfCommunicationClient<ICalculator>>
communicationClientFactory, Uri serviceUri, ServicePartitionKey partitionKey = null, TargetReplicaSelector
targetReplicaSelector = TargetReplicaSelector.Default, string listenerName = null, OperationRetrySettings
retrySettings = null)
           : base(communicationClientFactory, serviceUri, partitionKey, targetReplicaSelector, listenerName,
retrySettings)
       {
       }
   }
```

Client code can use the **WcfCommunicationClientFactory** along with the **WcfCommunicationClient** which implements **ServicePartitionClient** to determine the service endpoint and communicate with the service.

```
// Create binding
Binding binding = WcfUtility.CreateTcpClientBinding();
// Create a partition resolver
IServicePartitionResolver partitionResolver = ServicePartitionResolver.GetDefault();
// create a  WcfCommunicationClientFactory object.
var wcfClientFactory = new WcfCommunicationClientFactory<ICalculator>
    (clientBinding: binding, servicePartitionResolver: partitionResolver);

//
// Create a client for communicating with the ICalculator service that has been created with the
// Singleton partition scheme.
//
var calculatorServiceCommunicationClient =  new WcfCommunicationClient(
                wcfClientFactory,
                ServiceUri,
                ServicePartitionKey.Singleton);

//
// Call the service to perform the operation.
//
var result = calculatorServiceCommunicationClient.InvokeWithRetryAsync(
                client => client.Channel.Add(2, 3)).Result;
```

> **NOTE**
>
> The default ServicePartitionResolver assumes that the client is running in same cluster as the service. If that is not the case, create a ServicePartitionResolver object and pass in the cluster connection endpoints.

Next steps

- Remote procedure call with Reliable Services remoting
- Web API with OWIN in Reliable Services
- Securing communication for Reliable Services

Service Fabric Reverse Proxy

11/15/2016 • 7 min to read • Edit on GitHub

Contributors

BharatNarasimman • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • James Dunn • Tim Hess • carolinacmoravia

The Service Fabric Reverse proxy is a reverse proxy built-into service fabric that allows addressing microservices in the service fabric cluster that expose HTTP endpoints.

Microservices communication model

Microservices in service fabric typically run on a subset of VM's in the cluster and can move from one VM to another for various reasons. So the endpoints for microservices can change dynamically. The typical pattern to communicate to the microservice is the resolve loop below,

1. Resolve the service location initially through the Naming Service.
2. Connect to the service.
3. Determine the cause of connection failures and re-resolve the service location when necessary.

This process generally involves wrapping client-side communication libraries in a retry loop that implements the service resolution and retry policies. For more information on this topic, see communicating with services.

Communicating via SF reverse proxy

Service Fabric reverse proxy runs on all the nodes in the cluster. It performs the entire service resolution process on a client's behalf and then forwards the client request. So clients running on the cluster can just use any client-side HTTP communication libraries to talk to the target service via the SF reverse proxy running locally on the same node.

Reaching Microservices from outside the cluster

The default external communication model for microservices is **opt-in** where each service by default cannot be accessed directly from external clients. The Azure Load Balancer is a network boundary between microservices and external clients, that performs network address translation and forwards external requests to internal **IP:port** endpoints. In order to make a microservice's endpoint directly accessible to external clients, the Azure Load Balancer must first be configured to forward traffic to each port used by the service in the cluster. Furthermore, most microservices(esp. stateful microservices) dont live on all the nodes of the cluster and they can move between nodes on failover, so in such cases, the Azure Load Balancer cannot effectively determine the target node of the replicas are located to forward the traffic to.

Reaching Microservices via the SF reverse proxy from outside the cluster

Instead of configuring individual service's ports in the azure load balancer, just the SF Reverse proxy port can be configured in the Azure Load Balancer. This allows clients outside the cluster to reach services inside the cluster via the reverse proxy without an additional configurations.

> **WARNING**
> Configuring the reverse proxy's port on the load balancer, makes all the micro services in the cluster that expose a http endpoint, to be addressible from outside the cluster.

URI format for addressing services via the reverse proxy

The Reverse proxy uses a specific URI format to identify which service partition the incoming request should be forwarded to :

```
http(s)://<Cluster FQDN | internal IP>:Port/<ServiceInstanceName>/<Suffix path>?PartitionKey=<key>&PartitionKind=
<partitionkind>&Timeout=<timeout_in_seconds>
```

- **http(s):** The reverse proxy can be configured to accept HTTP or HTTPS traffic. In case of HTTPS traffic, SSL termination occurs at the reverse proxy. Requests that are forwarded by the reverse proxy to services in the

cluster are over http.

- **Cluster FQDN | internal IP:** For external clients, the reverse proxy can be configured so that it is reachable through the cluster domain (e.g., mycluster.eastus.cloudapp.azure.com). By default the reverse proxy runs on every node, so for internal traffic it can be reached on localhost or on any internal node IP (e.g., 10.0.0.1).
- **Port:** The port that has been specified for the reverse proxy. Eg: 19008.
- **ServiceInstanceName:** This is the fully-qualified deployed service instance name of the service you are trying to reach sans the "fabric:/" scheme. For example, to reach service *fabric:/myapp/myservice/*, you would use *myapp/myservice*.
- **Suffix path:** This is the actual URL path for the service that you want to connect to. For example, *myapi/values/add/3*
- **PartitionKey:** For a partitioned service, this is the computed partition key of the partition you want to reach. Note that this is *not* the partition ID GUID. This parameter is not required for services using the singleton partition scheme.
- **PartitionKind:** The service partition scheme. This can be 'Int64Range' or 'Named'. This parameter is not required for services using the singleton partition scheme.
- **Timeout:** This specifies the timeout for the http request created by the reverse proxy to the service on behalf of the client request. The default value for this is 60 seconds. This is an optional parameter.

Example usage

As an example, let's take service **fabric:/MyApp/MyService** that opens an HTTP listener on the following URL:

```
http://10.0.0.5:10592/3f0d39ad-924b-4233-b4a7-02617c6308a6-130834621071472715/
```

With the following resources:

- `/index.html`
- `/api/users/<userId>`

If the service uses the singleton partitioning scheme, the *PartitionKey* and *PartitionKind* query string parameters are not required, and the service can be reached via the gateway as:

- Externally: `http://mycluster.eastus.cloudapp.azure.com:19008/MyApp/MyService`
- Internally: `http://localhost:19008/MyApp/MyService`

If the service uses the Uniform Int64 partitioning scheme, the *PartitionKey* and *PartitionKind* query string parameters must be used to reach a partition of the service:

- Externally: `http://mycluster.eastus.cloudapp.azure.com:19008/MyApp/MyService?PartitionKey=3&PartitionKind=Int64Range`
- Internally: `http://localhost:19008/MyApp/MyService?PartitionKey=3&PartitionKind=Int64Range`

To reach the resources exposed by the service, simply place the resource path after the service name in the URL:

- Externally: `http://mycluster.eastus.cloudapp.azure.com:19008/MyApp/MyService/index.html?PartitionKey=3&PartitionKind=Int64Range`
- Internally: `http://localhost:19008/MyApp/MyService/api/users/6?PartitionKey=3&PartitionKind=Int64Range`

The gateway will then forward these requests to the service's URL:

- `http://10.0.0.5:10592/3f0d39ad-924b-4233-b4a7-02617c6308a6-130834621071472715/index.html`
- `http://10.0.0.5:10592/3f0d39ad-924b-4233-b4a7-02617c6308a6-130834621071472715/api/users/6`

Special handling for port-sharing services

The Application Gateway attempts to re-resolve a service address and retry the request when a service cannot be reached. This is one of the major benefits of the gateway, as client code does not need to implement its own service resolution and resolve loop.

Generally when a service cannot be reached it means the service instance or replica has moved to a different node as part of its normal lifecycle. When this happens, the gateway may receive a network connection error indicating an endpoint is no longer open on the originally resolved address.

However, replicas or service instances can share a host process and may also share a port when hosted by an http.sys-based web server, including:

- System.Net.HttpListener
- ASP.NET Core WebListener
- Katana

In this situation it is likely that the web server is available in the host process and responding to requests but the resolved service instance or replica is no longer available on the host. In this case, the gateway will receive an HTTP 404 response from the web server. As a result, an HTTP 404 has two distinct meanings:

1. The service address is correct, but the resource requested by the user does not exist.
2. The service address is incorrect, and the resource requested by the user may actually exist on a different node.

In the first case, this is a normal HTTP 404, which is considered a user error. However, in the second case, the user has requested a resource that does exist, but the gateway was unable to locate it because the service itself has moved, in which case the gateway needs to re-resolve the address and try again.

The gateway thus needs a way to distinguish between these two cases. In order to make that distinction, a hint from the server is required.

- By default, the Application Gateway assumes case #2 and attempts to re-resolve and re-issue the request.
- To indicate case #1 to the Application Gateway, the service should return the following HTTP response header:

`X-ServiceFabric : ResourceNotFound`

This HTTP response header indicates a normal HTTP 404 situation in which the requested resource does not exist, and the gateway will not attempt to re-resolve the service address.

Setup and configuration

The service fabric Reverse proxy can be enabled for the cluster via the Azure Resource Manager template.

Once you have the template for the cluster that you want to deploy(either from the sample templates or by creating a custom resource manager template) the Reverse proxy can be enabled in the template by the following steps.

1. Define a port for the reverse proxy in the Parameters section of the template.

```
"SFReverseProxyPort": {
    "type": "int",
    "defaultValue": 19008,
    "metadata": {
        "description": "Endpoint for Service Fabric Reverse proxy"
    }
},
```

2. Specify the port for each of the nodetype objects in the **Cluster** Resource type section

   For apiVersion's prior to '2016-09-01' the port is identified by the parameter name ***httpApplicationGatewayEndpointPort***

```
{
    "apiVersion": "2016-03-01",
    "type": "Microsoft.ServiceFabric/clusters",
    "name": "[parameters('clusterName')]",
    "location": "[parameters('clusterLocation')]",
    ...
   "nodeTypes": [
      {
       ...
       "httpApplicationGatewayEndpointPort": "[parameters('SFReverseProxyPort')]",
       ...
      },
    ...
    ],
    ...
}
```

For apiVersion's on or after '2016-09-01' the port is identified by the parameter name *reverseProxyEndpointPort*

```
{
    "apiVersion": "2016-09-01",
    "type": "Microsoft.ServiceFabric/clusters",
    "name": "[parameters('clusterName')]",
    "location": "[parameters('clusterLocation')]",
    ...
   "nodeTypes": [
      {
       ...
       "reverseProxyEndpointPort": "[parameters('SFReverseProxyPort')]",
       ...
      },
    ...
    ],
    ...
}
```

3. To address the reverse proxy from outside the azure cluster, setup the **azure load balancer rules** for the port specified in step 1.

```
{
    "apiVersion": "[variables('lbApiVersion')]",
    "type": "Microsoft.Network/loadBalancers",
    ...
    ...
    "loadBalancingRules": [
        ...
        {
            "name": "LBSFReverseProxyRule",
            "properties": {
                "backendAddressPool": {
                    "id": "[variables('lbPoolID0')]"
                },
                "backendPort": "[parameters('SFReverseProxyPort')]",
                "enableFloatingIP": "false",
                "frontendIPConfiguration": {
                    "id": "[variables('lbIPConfig0')]"
                },
                "frontendPort": "[parameters('SFReverseProxyPort')]",
                "idleTimeoutInMinutes": "5",
                "probe": {
                    "id": "[concat(variables('lbID0'),'/probes/SFReverseProxyProbe')]"
                },
                "protocol": "tcp"
            }
        }
    ],
    "probes": [
        ...
        {
            "name": "SFReverseProxyProbe",
            "properties": {
                "intervalInSeconds": 5,
                "numberOfProbes": 2,
                "port":     "[parameters('SFReverseProxyPort')]",
                "protocol": "tcp"
            }
        }
    ]
}
```

4. To configure SSL certificates on the port for the Reverse proxy, add the certificate to the httpApplicationGatewayCertificate property in the **Cluster** Resource type section

   For apiVersion's prior to '2016-09-01' the certificate is identified by the parameter name ***httpApplicationGatewayCertificate***

```
{
    "apiVersion": "2016-03-01",
    "type": "Microsoft.ServiceFabric/clusters",
    "name": "[parameters('clusterName')]",
    "location": "[parameters('clusterLocation')]",
    "dependsOn": [
        "[concat('Microsoft.Storage/storageAccounts/', parameters('supportLogStorageAccountName'))]"
    ],
    "properties": {
        ...
        "httpApplicationGatewayCertificate": {
            "thumbprint": "[parameters('sfReverseProxyCertificateThumbprint')]",
            "x509StoreName": "[parameters('sfReverseProxyCertificateStoreName')]"
        },
        ...
        "clusterState": "Default",
    }
}
```

For apiVersion's on or after '2016-09-01' the certificate is identified by the parameter name *reverseProxyCertificate*

```
{
    "apiVersion": "2016-09-01",
    "type": "Microsoft.ServiceFabric/clusters",
    "name": "[parameters('clusterName')]",
    "location": "[parameters('clusterLocation')]",
    "dependsOn": [
        "[concat('Microsoft.Storage/storageAccounts/', parameters('supportLogStorageAccountName'))]"
    ],
    "properties": {
        ...
        "reverseProxyCertificate": {
            "thumbprint": "[parameters('sfReverseProxyCertificateThumbprint')]",
            "x509StoreName": "[parameters('sfReverseProxyCertificateStoreName')]"
        },
        ...
        "clusterState": "Default",
    }
}
```

Next steps

- See an example of HTTP communication between services in a sample project on GitHUb.
- Remote procedure calls with Reliable Services remoting
- Web API that uses OWIN in Reliable Services
- WCF communication by using Reliable Services

Advanced usage of the Reliable Services programming model

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • unknown • Jesse Benson • ShawnJackson • Ryan Wike

Azure Service Fabric simplifies writing and managing reliable stateless and stateful services. This guide will talk about advanced usages of Reliable Services to gain more control and flexibility over your services. Prior to reading this guide, familiarize yourself with the Reliable Services programming model.

Both stateful and stateless services have two primary entry points for user code:

- `RunAsync` is a general-purpose entry point for your service code.
- `CreateServiceReplicaListeners` and `CreateServiceInstanceListeners` is for opening communication listeners for client requests.

For most services, these two entry points are sufficient. In rare cases when more control over a service's lifecycle is required, additional lifecycle events are available.

Stateless service instance lifecycle

A stateless service's lifecycle is very simple. A stateless service can only be opened, closed, or aborted. `RunAsync` in a stateless service is executed when a service instance is opened, and cancelled when a service instance is closed or aborted.

Although `RunAsync` should be sufficient in almost all cases, the open, close, and abort events in a stateless service are also available:

- `Task OnOpenAsync(IStatelessServicePartition, CancellationToken)` OnOpenAsync is called when the stateless service instance is about to be used. Extended service initialization tasks can be started at this time.
- `Task OnCloseAsync(CancellationToken)` OnCloseAsync is called when the stateless service instance is going to be gracefully shut down. This can occur when the service's code is being upgraded, the service instance is being moved due to load balancing, or a transient fault is detected. OnCloseAsync can be used to safely close any resources, stop any background processing, finish saving external state, or close down existing connections.
- `void OnAbort()` OnAbort is called when the stateless service instance is being forcefully shut down. This is generally called when a permanent fault is detected on the node, or when Service Fabric cannot reliably manage the service instance's lifecycle due to internal failures.

Stateful service replica lifecycle

A stateful service replica's lifecycle is much more complex than a stateless service instance. In addition to open, close, and abort events, a stateful service replica undergoes role changes during its lifetime. When a stateful service replica changes role, the `OnChangeRoleAsync` event is triggered:

- `Task OnChangeRoleAsync(ReplicaRole, CancellationToken)` OnChangeRoleAsync is called when the stateful service replica is changing role, for example to primary or secondary. Primary replicas are given write status (are allowed to create and write to Reliable Collections). Secondary replicas are given read status (can only read from existing Reliable Collections). Most work in a stateful service is performed at the primary replica. Secondary replicas can perform read-only validation, report generation, data mining, or other read-only jobs.

In a stateful service, only the primary replica has write access to state and thus is generally when the service is performing actual work. The `RunAsync` method in a stateful service is executed only when the stateful service replica is primary. The `RunAsync` method is cancelled when a primary replica's role changes away from primary, as well as during the close and abort events.

Using the `OnChangeRoleAsync` event allows you to perform work depending on replica role as well as in response to role change.

A stateful service also provides the same four lifecycle events as a stateless service, with the same semantics and use cases:

- `Task OnOpenAsync(IStatefulServicePartition, CancellationToken)`
- `Task OnCloseAsync(CancellationToken)`
- `void OnAbort()`

Next steps

For more advanced topics related to Service Fabric, see the following articles:

- Configuring stateful Reliable Services
- Service Fabric health introduction
- Using system health reports for troubleshooting
- Configuring Services with the Service Fabric Cluster Resource Manager

Introduction to Service Fabric Reliable Actors

11/15/2016 • 10 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Theano Petersen • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Aaron Dandy • David Nissimoff • MarkDeichmiller • Ryan Wike • jessebenson • Sean McKenna

Reliable Actors is a Service Fabric application framework based on the Virtual Actor pattern. The Reliable Actors API provides a single-threaded programming model built on the scalability and reliability guarantees provided by Service Fabric.

What are Actors?

An actor is an isolated, independent unit of compute and state with single-threaded execution. The actor pattern is a computational model for concurrent or distributed systems in which a large number of these actors can execute simultaneously and independently of each other. Actors can communicate with each other and they can create more actors.

When to use Reliable Actors

Service Fabric Reliable Actors is an implementation of the actor design pattern. As with any software design pattern, the decision whether to use a specific pattern is made based on whether or not a software design problem fits the pattern.

Although the actor design pattern can be a good fit to a number of distributed systems problems and scenarios, careful consideration of the constraints of the pattern and the framework implementing it must be made. As general guidance, consider the actor pattern to model your problem or scenario if:

- Your problem space involves a large number (thousands or more) of small, independent, and isolated units of state and logic.
- You want to work with single-threaded objects that do not require significant interaction from external components, including querying state across a set of actors.
- Your actor instances won't block callers with unpredictable delays by issuing I/O operations.

Actors in Service Fabric

In Service Fabric, actors are implemented in the Reliable Actors framework: An actor-pattern-based application framework built on top of Service Fabric Reliable Services. Each Reliable Actor service you write is actually a partitioned, stateful Reliable Service.

Every actor is defined as an instance of an actor type, identical to the way a .NET object is an instance of a .NET type. For example, there may be an actor type that implements the functionality of a calculator and there could be many actors of that type that are distributed on various nodes across a cluster. Each such actor is uniquely identified by an actor ID.

Actor Lifetime

Service Fabric actors are virtual, meaning that their lifetime is not tied to their in-memory representation. As a result, they do not need to be explicitly created or destroyed. The Reliable Actors runtime automatically activates an actor the first time it receives a request for that actor ID. If an actor is not used for a period of time, the Reliable Actors runtime garbage-collects the in-memory object. It will also maintain knowledge of the actor's existence should it need to be reactivated later. For more details, see Actor lifecycle and garbage collection.

This virtual actor lifetime abstraction carries some caveats as a result of the virtual actor model, and in fact the Reliable Actors implementation deviates at times from this model.

- An actor is automatically activated (causing an actor object to be constructed) the first time a message is sent to its actor ID. After some period of time, the actor object is garbage collected. In the future, using the actor ID again, causes a new actor object to be constructed. An actor's state outlives the object's lifetime when stored in the state manager.
- Calling any actor method for an actor ID activates that actor. For this reason, actor types have their constructor called implicitly by the runtime. Therefore, client code cannot pass parameters to the actor type's constructor, although parameters may be passed to the actor's constructor by the service itself. The result is that actors may be constructed in a partially-initialized state by the time other methods are called on it, if the actor requires initialization parameters from the client. There is no single entry point for the activation of an actor from the client.
- Although Reliable Actors implicitly create actor objects; you do have the ability to explicitly delete an actor and its state.

Distribution and failover

To provide scalability and reliability, Service Fabric distributes actors throughout the cluster and automatically migrates them from failed nodes to healthy ones as required. This is an abstraction over a partitioned, stateful Reliable Service. Distribution, scalability, reliability, and automatic failover are all provided by virtue of the fact that actors are running inside a stateful Reliable Service called the *Actor Service*.

Actors are distributed across the partitions of the Actor Service, and those partitions are distributed across the nodes in a Service Fabric cluster. Each service partition contains a set of actors. Service Fabric manages distribution and failover of the service partitions.

For example, an actor service with nine partitions deployed to three nodes using the default actor partition placement would be distributed thusly:

The Actor Framework manages partition scheme and key range settings for you. This simplifies some choices but also carries some consideration:

- Reliable Services allows you to choose a partitioning scheme, key range (when using a range partitioning scheme), and partition count. Reliable Actors is restricted to the range partitioning scheme (the uniform Int64 scheme) and requires you use the full Int64 key range.
- By default, actors are randomly placed into partitions resulting in uniform distribution.
- Because actors are randomly placed, it should be expected that actor operations will always require network communication, including serialization and deserialization of method call data, incurring latency and overhead.
- In advanced scenarios, it is possible to control actor partition placement by using Int64 actor IDs that map to specific partitions. However, doing so can result in an unbalanced distribution of actors across partitions.

For more information on how actor services are partitioned, refer to partitioning concepts for actors.

Actor communication

Actor interactions are defined in an interface that is shared by the actor that implements the interface, and the client that gets a proxy to an actor via the same interface. Because this interface is used to invoke actor methods asynchronously, every method on the interface must be Task-returning.

Method invocations and their responses ultimately result in network requests across the cluster, so the arguments and the result types of the tasks that they return must be serializable by the platform. In particular, they must be data contract serializable.

The actor proxy

The Reliable Actors client API provides communication between an actor instance and an actor client. To communicate with an actor, a client creates an actor proxy object that implements the actor interface. The client interacts with the actor by invoking methods on the proxy object. The actor proxy can be used for client-to-actor and actor-to-actor communication.

```
// Create a randomly distributed actor ID
ActorId actorId = ActorId.CreateRandom();

// This only creates a proxy object, it does not activate an actor or invoke any methods yet.
IMyActor myActor = ActorProxy.Create<IMyActor>(actorId, new Uri("fabric:/MyApp/MyActorService"));

// This will invoke a method on the actor. If an actor with the given ID does not exist, it will be activated
by this method call.
await myActor.DoWorkAsync();
```

Note that the two pieces of information used to create the actor proxy object are the actor ID and the application name. The actor ID uniquely identifies the actor, while the application name identifies the Service Fabric application where the actor is deployed.

The `ActorProxy` class on the client side performs the necessary resolution to locate the actor by ID and open a communication channel with it. The `ActorProxy` also retries to locate the actor in the cases of communication failures and failovers. As a result, message delivery has the following characteristics:

- Message delivery is best effort.
- Actors may receive duplicate messages from the same client.

Concurrency

The Reliable Actors runtime provides a simple turn-based access model for accessing actor methods. This means that no more than one thread can be active inside an actor object's code at any time. Turn-based access greatly simplifies concurrent systems as there is no need for synchronization mechanisms for data access. It also means systems must be designed with special considerations for the single-threaded access nature of each actor instance.

- A single actor instance cannot process more than one request at a time. An actor instance can cause a throughput bottleneck if it is expected to handle concurrent requests.
- Actors can deadlock on each other if there is a circular request between two actors while an external request is made to one of the actors simultaneously. The actor runtime will automatically time out on actor calls and throw an exception to the caller to interrupt possible deadlock situations.

Turn-based access

A turn consists of the complete execution of an actor method in response to a request from other actors or clients, or the complete execution of a timer/reminder callback. Even though these methods and callbacks are asynchronous, the Actors runtime does not interleave them. A turn must be fully finished before a new turn is allowed. In other words, an actor method or timer/reminder callback that is currently executing must be fully finished before a new call to a method or callback is allowed. A method or callback is considered to have finished if the execution has returned from the method or callback and the task returned by the method or callback has finished. It is worth emphasizing that turn-based concurrency is respected even across different methods, timers, and callbacks.

The Actors runtime enforces turn-based concurrency by acquiring a per-actor lock at the beginning of a turn and releasing the lock at the end of the turn. Thus, turn-based concurrency is enforced on a per-actor basis and not across actors. Actor methods and timer/reminder callbacks can execute simultaneously on behalf of different actors.

The following example illustrates the above concepts. Consider an actor type that implements two asynchronous methods (say, *Method1* and *Method2*), a timer, and a reminder. The diagram below shows an example of a timeline for the execution of these methods and callbacks on behalf of two actors (*ActorId1* and *ActorId2*) that belong to this actor type.

This diagram follows these conventions:

- Each vertical line shows the logical flow of execution of a method or a callback on behalf of a particular actor.
- The events marked on each vertical line occur in chronological order, with newer events occurring below older ones.
- Different colors are used for timelines corresponding to different actors.
- Highlighting is used to indicate the duration for which the per-actor lock is held on behalf of a method or callback.

Some important points to consider:

- While *Method1* is executing on behalf of *ActorId2* in response to client request *xyz789*, another client request (*abc123*) arrives that also requires *Method1* to be executed by *ActorId2*. However, the second execution of *Method1* does not begin until the prior execution has finished. Similarly, a reminder registered by *ActorId2* fires while *Method1* is being executed in response to client request *xyz789*. The reminder callback is executed only after both executions of *Method1* are complete. All of this is due to turn-based concurrency being enforced for *ActorId2*.
- Similarly, turn-based concurrency is also enforced for *ActorId1*, as demonstrated by the execution of *Method1*, *Method2*, and the timer callback on behalf of *ActorId1* happening in a serial fashion.
- Execution of *Method1* on behalf of *ActorId1* overlaps with its execution on behalf of *ActorId2*. This is because turn-based concurrency is enforced only within an actor and not across actors.
- In some of the method/callback executions, the `Task` returned by the method/callback finishes after the method returns. In some others, the `Task` has already finished by the time the method/callback returns. In both cases, the per-actor lock is released only after both the method/callback returns and the `Task` finishes.

Reentrancy

The Actors runtime allows reentrancy by default. This means that if an actor method of *Actor A* calls a method on *Actor B*, which in turn calls another method on *Actor A*, that method is allowed to run. This is because it is part of the same logical call-chain context. All timer and reminder calls start with the new logical call context. See the

Reliable Actors reentrancy for more details.

Scope of concurrency guarantees

The Actors runtime provides these concurrency guarantees in situations where it controls the invocation of these methods. For example, it provides these guarantees for the method invocations that are done in response to a client request, as well as for timer and reminder callbacks. However, if the actor code directly invokes these methods outside of the mechanisms provided by the Actors runtime, then the runtime cannot provide any concurrency guarantees. For example, if the method is invoked in the context of some task that is not associated with the task returned by the actor methods, then the runtime cannot provide concurrency guarantees. If the method is invoked from a thread that the actor creates on its own, then the runtime also cannot provide concurrency guarantees. Therefore, to perform background operations, actors should use actor timers and actor reminders that respect turn-based concurrency.

Next steps

- Getting started with Reliable Actors
- How Reliable Actors use the Service Fabric platform
- Actor state management
- Actor lifecycle and garbage collection
- Actor timers and reminders
- Actor events
- Actor reentrancy
- Actor polymorphism and object-oriented design patterns
- Actor diagnostics and performance monitoring

Getting started with Reliable Actors

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Wike • MarkDeichmiller • Matt Thalman • Andy De George • jessebenson • Sean McKenna • clca

This article explains the basics of Azure Service Fabric Reliable Actors and walks you through creating, debugging, and deploying a simple Reliable Actor application in Visual Studio.

Installation and setup

Before you start, ensure that you have the Service Fabric development environment set up on your machine. If you need to set it up, see detailed instructions on how to set up the development environment.

Basic concepts

To get started with Reliable Actors, you only need to understand a few basic concepts:

- **Actor service**. Reliable Actors are packaged in Reliable Services that can be deployed in the Service Fabric infrastructure. Actor instances are activated in a named service instance.
- **Actor registration**. As with Reliable Services, a Reliable Actor service needs to be registered with the Service Fabric runtime. In addition, the actor type needs to be registered with the Actor runtime.
- **Actor interface**. The actor interface is used to define a strongly typed public interface of an actor. In the Reliable Actor model terminology, the actor interface defines the types of messages that the actor can understand and process. The actor interface is used by other actors and client applications to "send" (asynchronously) messages to the actor. Reliable Actors can implement multiple interfaces.
- **ActorProxy class**. The ActorProxy class is used by client applications to invoke the methods exposed through the actor interface. The ActorProxy class provides two important functionalities:
  - Name resolution: It is able to locate the actor in the cluster (find the node of the cluster where it is hosted).
  - Failure handling: It can retry method invocations and re-resolve the actor location after, for example, a failure that requires the actor to be relocated to another node in the cluster.

The following rules that pertain to actor interfaces are worth mentioning:

- Actor interface methods cannot be overloaded.
- Actor interface methods must not have out, ref, or optional parameters.
- Generic interfaces are not supported.

Create a new project in Visual Studio

After you have installed the Service Fabric tools for Visual Studio, you can create new project types. The new project types are under the **Cloud** category of the **New Project** dialog box.

In the next dialog box, you can choose the type of project that you want to create.

For the HelloWorld project, let's use the Service Fabric Reliable Actors service.

After you have created the solution, you should see the following structure:

Reliable Actors basic building blocks

A typical Reliable Actors solution is composed of three projects:

- **The application project (MyActorApplication)**. This is the project that packages all of the services together for deployment. It contains the *ApplicationManifest.xml* and PowerShell scripts for managing the application.
- **The interface project (MyActor.Interfaces)**. This is the project that contains the interface definition for the actor. In the MyActor.Interfaces project, you can define the interfaces that will be used by the actors in the solution. Your actor interfaces can be defined in any project with any name, however the interface defines the actor contract that is shared by the actor implementation and the clients calling the actor, so it typically makes sense to define it in an assembly that is separate from the actor implementation and can be shared by multiple other projects.

```
public interface IMyActor : IActor
{
    Task<string> HelloWorld();
}
```

- **The actor service project (MyActor)**. This is the project used to define the Service Fabric service that is going to host the actor. It contains the implementation of the actor. An actor implementation is a class that derives from the base type `Actor` and implements the interface(s) that are defined in the MyActor.Interfaces project. An actor class must also implement a constructor that accepts an `ActorService` instance and an `ActorId` and passes them to the base `Actor` class. This allows for constructor dependency injection of platform dependencies.

```
[StatePersistence(StatePersistence.Persisted)]
class MyActor : Actor, IMyActor
{
    public MyActor(ActorService actorService, ActorId actorId)
        : base(actorService, actorId)
    {
    }

    public Task<string> HelloWorld()
    {
        return Task.FromResult("Hello world!");
    }
}
```

The actor service must be registered with a service type in the Service Fabric runtime. In order for the Actor Service to run your actor instances, your actor type must also be registered with the Actor Service. The `ActorRuntime` registration method performs this work for actors.

```
internal static class Program
{
    private static void Main()
    {
        try
        {
            ActorRuntime.RegisterActorAsync<MyActor>(
                (context, actorType) => new ActorService(context, actorType, () => new
MyActor())).GetAwaiter().GetResult();

            Thread.Sleep(Timeout.Infinite);
        }
        catch (Exception e)
        {
            ActorEventSource.Current.ActorHostInitializationFailed(e.ToString());
            throw;
        }
    }
}
```

If you start from a new project in Visual Studio and you have only one actor definition, the registration is included by default in the code that Visual Studio generates. If you define other actors in the service, you need to add the actor registration by using:

```
 ActorRuntime.RegisterActorAsync<MyOtherActor>();
```

> **TIP**
> The Service Fabric Actors runtime emits some events and performance counters related to actor methods. They are useful in diagnostics and performance monitoring.

Debugging

The Service Fabric tools for Visual Studio support debugging on your local machine. You can start a debugging session by hitting the F5 key. Visual Studio builds (if necessary) packages. It also deploys the application on the local Service Fabric cluster and attaches the debugger.

During the deployment process, you can see the progress in the **Output** window.

```
Output
Show output from: Debug

ApplicationName        : fabric:/Application1
ApplicationTypeName    : Application1Type
ApplicationTypeVersion : 1.0.0.0
ApplicationParameters  : { "_WFDebugParams_" = "[{"ServiceManifestName":"Actor1Pkg","CodePackageName":"Code","EntryP
                         tType":"Main","DebugExePath":"C:\\Program Files (x86)\\Microsoft Visual Studio
                         14.0\\Common7\\Packages\\Debugger\\VsDebugLaunchNotify.exe","DebugArguments":"
                         {e0e3e79d-88aa-41ba-97c9-1a70b0c17e50} -p [ProcessId] -tid
                         [ThreadId]","EnvironmentBlock":"_NO_DEBUG_HEAP=1\u0000"}]" }


Finished executing script 'Create-FabricApplication.ps1'.
Time elapsed: 00:00:02.5983385
Started executing script 'Get-FabricApplicationStatus.ps1'.
&'E:\ServiceFabricProjects\Application1\Application1\Scripts\Get-FabricApplicationStatus.ps1' -ApplicationManifestPa
The application has started.
Service Status:
fabric:/Application1/Actor1ActorService is ready.

Error List   Output
```

Next steps

- How Reliable Actors use the Service Fabric platform
- Actor state management
- Actor lifecycle and garbage collection
- Actor API reference documentation
- Sample code

Getting started with Reliable Actors

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

This article explains the basics of Azure Service Fabric Reliable Actors and walks you through creating and deploying a simple Reliable Actor application in Java.

Installation and setup

Before you start, make sure you have the Service Fabric development environment set up on your machine. If you need to set it up, go to getting started on Mac or getting started on Linux.

Basic concepts

To get started with Reliable Actors, you only need to understand a few basic concepts:

- **Actor service**. Reliable Actors are packaged in Reliable Services that can be deployed in the Service Fabric infrastructure. Actor instances are activated in a named service instance.
- **Actor registration**. As with Reliable Services, a Reliable Actor service needs to be registered with the Service Fabric runtime. In addition, the actor type needs to be registered with the Actor runtime.
- **Actor interface**. The actor interface is used to define a strongly typed public interface of an actor. In the Reliable Actor model terminology, the actor interface defines the types of messages that the actor can understand and process. The actor interface is used by other actors and client applications to "send" (asynchronously) messages to the actor. Reliable Actors can implement multiple interfaces.
- **ActorProxy class**. The ActorProxy class is used by client applications to invoke the methods exposed through the actor interface. The ActorProxy class provides two important functionalities:
  - Name resolution: It is able to locate the actor in the cluster (find the node of the cluster where it is hosted).
  - Failure handling: It can retry method invocations and re-resolve the actor location after, for example, a failure that requires the actor to be relocated to another node in the cluster.

The following rules that pertain to actor interfaces are worth mentioning:

- Actor interface methods cannot be overloaded.
- Actor interface methods must not have out, ref, or optional parameters.
- Generic interfaces are not supported.

Create an actor service

Start by creating a new Service Fabric application. The Service Fabric SDK for Linux includes a Yeoman generator to provide the scaffolding for a Service Fabric application with a stateless service. Start by running the following Yeoman command:

```
$ yo azuresfjava
```

Follow the instructions to create a **Reliable Actor Service**. For this tutorial, name the application

"HelloWorldActorApplication" and the actor "HelloWorldActor." The following scaffolding will be created:

```
HelloWorldActorApplication/
□□□ build.gradle
□□□ HelloWorldActor
□   □□□ build.gradle
□   □□□ settings.gradle
□   □□□ src
□       □□□ reliableactor
□           □□□ HelloWorldActorHost.java
□           □□□ HelloWorldActorImpl.java
□□□ HelloWorldActorApplication
□   □□□ ApplicationManifest.xml
□   □□□ HelloWorldActorPkg
□       □□□ Code
□       □   □□□ entryPoint.sh
□       □   □□□ _readme.txt
□       □□□ Config
□       □   □□□ _readme.txt
□       □   □□□ Settings.xml
□       □□□ Data
□       □   □□□ _readme.txt
□       □□□ ServiceManifest.xml
□□□ HelloWorldActorInterface
□   □□□ build.gradle
□   □□□ src
□       □□□ reliableactor
□           □□□ HelloWorldActor.java
□□□ HelloWorldActorTestClient
□   □□□ build.gradle
□   □□□ settings.gradle
□   □□□ src
□   □   □□□ reliableactor
□   □       □□□ test
□   □           □□□ HelloWorldActorTestClient.java
□   □□□ testclient.sh
□□□ install.sh
□□□ settings.gradle
□□□ uninstall.sh
```

Reliable Actors basic building blocks

The basic concepts described earlier translate into the basic building blocks of a Reliable Actor service.

Actor interface

This contains the interface definition for the actor. This interface defines the actor contract that is shared by the actor implementation and the clients calling the actor, so it typically makes sense to define it in a place that is separate from the actor implementation and can be shared by multiple other services or client applications.

`HelloWorldActorInterface/src/reliableactor/HelloWorldActor.java` :

```
public interface HelloWorldActor extends Actor {
    @Readonly
    CompletableFuture<Integer> getCountAsync();

    CompletableFuture<?> setCountAsync(int count);
}
```

Actor service

This contains your actor implementation and actor registration code. The actor class implements the actor interface. This is where your actor does its work.

`HelloWorldActor/src/reliableactor/HelloWorldActorImpl`:

```
@ActorServiceAttribute(name = "HelloWorldActor.HelloWorldActorService")
@StatePersistenceAttribute(statePersistence = StatePersistence.Persisted)
public class HelloWorldActorImpl extends ReliableActor implements HelloWorldActor {
    Logger logger = Logger.getLogger(this.getClass().getName());

    protected CompletableFuture<?> onActivateAsync() {
        logger.log(Level.INFO, "onActivateAsync");

        return this.stateManager().tryAddStateAsync("count", 0);
    }

    @Override
    public CompletableFuture<Integer> getCountAsync() {
        logger.log(Level.INFO, "Getting current count value");
        return this.stateManager().getStateAsync("count");
    }

    @Override
    public CompletableFuture<?> setCountAsync(int count) {
        logger.log(Level.INFO, "Setting current count value {0}", count);
        return this.stateManager().addOrUpdateStateAsync("count", count, (key, value) -> count > value ? count :
value);
    }
}
```

Actor registration

The actor service must be registered with a service type in the Service Fabric runtime. In order for the Actor Service to run your actor instances, your actor type must also be registered with the Actor Service. The `ActorRuntime` registration method performs this work for actors.

`HelloWorldActor/src/reliableactor/HelloWorldActorHost`:

```
public class HelloWorldActorHost {

    public static void main(String[] args) throws Exception {

        try {
            ActorRuntime.registerActorAsync(HelloWorldActorImpl.class, (context, actorType) -> new
ActorServiceImpl(context, actorType, ()-> new HelloWorldActorImpl()), Duration.ofSeconds(10));

            Thread.sleep(Long.MAX_VALUE);

        } catch (Exception e) {
            e.printStackTrace();
            throw e;
        }
    }
}
```

Test client

This is a simple test client application you can run separately from the Service Fabric application to test your actor service. This is an example of where the ActorProxy can be used to activate and communicate with actor instances. It does not get deployed with your service.

The application

Finally, the application packages the actor service and any other services you might add in the future together for deployment. It contains the *ApplicationManifest.xml* and place holders for the actor service package.

Run the application

The Yeoman scaffolding includes a gradle script to build the application and bash scripts to deploy and un-deploy the application. To run the application, first build the application with gradle:

```
$ gradle
```

This will produce a Service Fabric application package that can be deployed using Service Fabric Azure CLI. The install.sh script contains the necessary Azure CLI commands to deploy the application package. Simply run the install.sh script to deploy:

```
$ ./install.sh
```

How Reliable Actors use the Service Fabric platform

11/15/2016 • 6 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Can Bilgin • Eli Arbel • Aman Bhardwaj • ShawnJackson • Andy De George • Sean McKenna • Jesse Benson • Abhishek Ram

This article explains how Reliable Actors work on the Service Fabric platform. Reliable Actors run in a framework that is hosted in an implementation of a stateful Reliable Service called the *Actor Service*. The Actor Service contains all the components necessary to manage the lifecycle and message dispatching for your actors:

- The Actor Runtime manages lifecycle, garbage collection, and enforces single-threaded access.
- An actor service remoting listener accepts remote access calls to actors and sends them to a dispatcher to route to the appropriate actor instance.
- The Actor State Provider wraps state providers (such as the Reliable Collections state provider) and provides an adapter for actor state management.

These components together form the Reliable Actor framework.

Service Layering

Because the Actor Service itself is a Reliable Service, all of the application model, lifecycle, packaging, [deployment]((service-fabric-deploy-remove-applications.md#deploy-an-application), upgrade, and scaling concepts of Reliable Services apply the same way to Actor services.

The diagram above shows the relationship between the Service Fabric application frameworks and user code. Blue elements represent the Reliable Services application framework, orange represents the Reliable Actor framework, and green represents user code.

In Reliable Services, your service inherits the `StatefulService` class, which itself is derived from `StatefulServiceBase`. (or `StatelessService` for stateless services). In Reliable Actors, you use the Actor Service which is a different implementation of the `StatefulServiceBase` class that implements the actor pattern where your actors execute. Since the Actor Service itself is just an implementation of `StatefulServiceBase`, you can write your own service that derives from `ActorService` and implement service-level features the same way you would when inheriting `StatefulService`, such as:

- Service back-up and restore.
- Shared functionality for all Actors, for example, a circuit-breaker.
- Remoting procedure calls on the actor service itself, as well as on each individual actor.

Using the Actor Service

Actor instances have access to the Actor Service in which they are executing. Through the Actor Service, actor instances can programmatically obtain the Service Context which has the partition ID, service name, application name, and other Service Fabric platform-specific information:

```
Task MyActorMethod()
{
    Guid partitionId = this.ActorService.Context.PartitionId;
    string serviceTypeName = this.ActorService.Context.ServiceTypeName;
    Uri serviceInstanceName = this.ActorService.Context.ServiceName;
    string applicationInstanceName = this.ActorService.Context.CodePackageActivationContext.ApplicationName;
}
```

Like all Reliable Services, the Actor Service must be registered with a service type in the Service Fabric runtime. In order for the Actor Service to run your actor instances, your actor type must also be registered with the Actor Service. The `ActorRuntime` registration method performs this work for actors. In the simplest case, you can just register your actor type, and the Actor Service with default settings will implicitly be used:

```
static class Program
{
    private static void Main()
    {
        ActorRuntime.RegisterActorAsync<MyActor>().GetAwaiter().GetResult();

        Thread.Sleep(Timeout.Infinite);
    }
}
```

Alternatively, you can use a lambda provided by the registration method to construct the Actor Service yourself. This allows you to configure the Actor Service as well as explicitly construct your actor instances, where you can inject dependencies to your actor through its constructor:

```
static class Program
{
    private static void Main()
    {
        ActorRuntime.RegisterActorAsync<MyActor>(
            (context, actorType) => new ActorService(context, actorType, () => new MyActor()))
            .GetAwaiter().GetResult();

        Thread.Sleep(Timeout.Infinite);
    }
}
```

Actor Service methods

The Actor Service implements `IActorService` which in turn implements `IService`. This is the interface used by Reliable Services remoting, which allows remote procedure calls on service methods. It contains service-level methods that can be called remotely using service remoting.

Enumerating actors

The Actor Service allows a client to enumerate metadata about the actors being hosted by the service. Since the Actor Service is a partitioned stateful service, enumeration is performed per partition. Because each partition may contain a large number of actors, the enumeration is return as a set of paged results. The pages are looped over until all pages are read. The following example shows how to create a list of all active actors in one partition of an actor service:

```
IActorService actorServiceProxy = ActorServiceProxy.Create(
    new Uri("fabric:/MyApp/MyService"), partitionKey);

ContinuationToken continuationToken = null;
List<ActorInformation> activeActors = new List<ActorInformation>();

do
{
    PagedResult<ActorInformation> page = await actorServiceProxy.GetActorsAsync(continuationToken,
cancellationToken);

    activeActors.AddRange(page.Items.Where(x => x.IsActive));

    continuationToken = page.ContinuationToken;
}
while (continuationToken != null);
```

Deleting actors

The Actor Service also provides a function for deleting actors:

```
ActorId actorToDelete = new ActorId(id);

IActorService myActorServiceProxy = ActorServiceProxy.Create(
    new Uri("fabric:/MyApp/MyService"), actorToDelete);

await myActorServiceProxy.DeleteActorAsync(actorToDelete, cancellationToken)
```

For more information on deleting actors and their state, refer to the actor lifecycle documentation.

Custom Actor Service

Using the actor registration lambda, you can also register your own custom actor service that derives from `ActorService` where you can implement your own service-level functionality. This is done by writing a service class that inherits `ActorService`. A custom actor service inherits all of the actor runtime functionality from `ActorService` and can be used to implement your own service methods.

```
class MyActorService : ActorService
{
    public MyActorService(StatefulServiceContext context, ActorTypeInformation typeInfo, Func<ActorBase>
newActor)
        : base(context, typeInfo, newActor)
    { }
}
```

```
static class Program
{
    private static void Main()
    {
        ActorRuntime.RegisterActorAsync<MyActor>(
            (context, actorType) => new MyActorService(context, actorType, () => new MyActor()))
            .GetAwaiter().GetResult();

        Thread.Sleep(Timeout.Infinite);
    }
}
```

Implementing actor back-up and restore

In the following example, the custom actor service exposes a method to back-up actor data by taking advantage of the remoting listener already present in `ActorService`:

```
public interface IMyActorService : IService
{
    Task BackupActorsAsync();
}

class MyActorService : ActorService, IMyActorService
{
    public MyActorService(StatefulServiceContext context, ActorTypeInformation typeInfo, Func<ActorBase>
newActor)
        : base(context, typeInfo, newActor)
    { }

    public Task BackupActorsAsync()
    {
        return this.BackupAsync(new BackupDescription(PerformBackupAsync));
    }

    private async Task<bool> PerformBackupAsync(BackupInfo backupInfo, CancellationToken cancellationToken)
    {
        try
        {
           // store the contents of backupInfo.Directory
           return true;
        }
        finally
        {
           Directory.Delete(backupInfo.Directory, recursive: true);
        }
    }
}
```

In this example, `IMyActorService` is a remoting contract that implements `IService` and is then implemented by `MyActorService`. By adding this remoting contract, methods on `IMyActorService` are now also available to a client by creating a remoting proxy using `ActorServiceProxy`:

```
IMyActorService myActorServiceProxy = ActorServiceProxy.Create<IMyActorService>(
    new Uri("fabric:/MyApp/MyService"), ActorId.CreateRandom());

await myActorServiceProxy.BackupActorsAsync();
```

Application model

Actor services are Reliable Services, so the application model is the same. However, the actor framework build tools generate much of the application model files for you.

Service Manifest

The contents of your actor service's ServiceManifest.xml are generated automatically by the actor framework build tools. This includes:

- The actor service type. The type name is generated based on your actor project name. Based on the persistence attribute on your actor, the HasPersistedState flag is also set accordingly.
- Code package.
- Config package.
- Resources and endpoints

Application Manifest

The actor framework build tools automatically create a default service definition for your actor service. The default service properties are populated by the build tools:

- Replica set count is determined by the persistence attribute on your actor. Each time the persistence attribute

on your actor is changed, the replica set count in the default service definition will be reset accordingly.

- Partition scheme and range is set to Uniform Int64 with the full Int64 key range.

Service Fabric partition concepts for actors

Actor services are partitioned stateful services. Each partition of an actor service contains a set of actors. Service partitions are automatically distributed over multiple nodes in Service Fabric. Thus, actor instances are distributed as a result.

Reliable Services can be created with different partition schemes and partition key ranges. The Actor Service uses the Int64 partitioning scheme with the full Int64 key range to map actors to partitions.

Actor ID

Each actor that's created in the service has a unique ID associated with it, represented by the `ActorId` class. The `ActorId` is an opaque id value that can be used for uniform distribution of actors across the service partitions by generating random IDs:

```
ActorProxy.Create<IMyActor>(ActorId.CreateRandom());
```

Every `ActorId` is hashed to an Int64, which is why the actor service must use an Int64 partitioning scheme with the full Int64 key range. However, custom ID values can be used for an `ActorID`, including GUIDs, strings, and Int64s.

```
ActorProxy.Create<IMyActor>(new ActorId(Guid.NewGuid()));
ActorProxy.Create<IMyActor>(new ActorId("myActorId"));
ActorProxy.Create<IMyActor>(new ActorId(1234));
```

When using GUIDs and strings, the values are hashed to an Int64. However, when explicitly providing an Int64 to an `ActorId`, the Int64 will map directly to a partition without further hashing. This can be used to control which partition actors are placed in.

Next steps

- Actor state management
- Actor lifecycle and garbage collection
- Actors API reference documentation
- Sample code

Actor lifecycle, automatic garbage collection, and manual delete

11/15/2016 • 5 min to read • Edit on GitHub

Contributors

Aman Bhardwaj • Theano Petersen • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Vaclav Turecek • ShawnJackson • jessebenson • Bill Craun

An actor is activated the first time a call is made to any of its methods. An actor is deactivated (garbage collected by the Actors runtime) if it is not used for a configurable period of time. An actor and its state can also be deleted manually at any time.

Actor activation

When an actor is activated, the following occurs:

- When a call comes for an actor and one is not already active, a new actor is created.
- The actor's state is loaded if it's maintaining state.
- The `OnActivateAsync` method (which can be overridden in the actor implementation) is called.
- The actor is now considered active.

Actor deactivation

When an actor is deactivated, the following occurs:

- When an actor is not used for some period of time, it is removed from the Active Actors table.
- The `OnDeactivateAsync` method (which can be overridden in the actor implementation) is called. This clears all the timers for the actor. Actor operations like state changes should not be called from this method.

> **TIP**
> The Fabric Actors runtime emits some events related to actor activation and deactivation. They are useful in diagnostics and performance monitoring.

Actor garbage collection

When an actor is deactivated, references to the actor object are released and it can be garbage collected normally by the common language runtime (CLR) garbage collector. Garbage collection only cleans up the actor object; it does **not** remove state stored in the actor's State Manager. The next time the actor is activated, a new actor object is created and its state is restored.

What counts as “being used” for the purpose of deactivation and garbage collection?

- Receiving a call
- `IRemindable.ReceiveReminderAsync` method being invoked (applicable only if the actor uses reminders)

> **NOTE**
> if the actor uses timers and its timer callback is invoked, it does **not** count as "being used".

Before we go into the details of deactivation, it is important to define the following terms:

- *Scan interval*. This is the interval at which the Actors runtime scans its Active Actors table for actors that can be deactivated and garbage collected. The default value for this is 1 minute.
- *Idle timeout*. This is the amount of time that an actor needs to remain unused (idle) before it can be deactivated and garbage collected. The default value for this is 60 minutes.

Typically, you do not need to change these defaults. However, if necessary, these intervals can be changed through `ActorServiceSettings` when registering your Actor Service:

```
public class Program
{
    public static void Main(string[] args)
    {
        ActorRuntime.RegisterActorAsync<MyActor>((context, actorType) =>
                new ActorService(context, actorType,
                    settings:
                        new ActorServiceSettings()
                        {
                            ActorGarbageCollectionSettings =
                                new ActorGarbageCollectionSettings(10, 2)
                        }))
            .GetAwaiter()
            .GetResult();
    }
}
```

For each active actor, the actor runtime keeps track of the amount of time that it has been idle (i.e. not used). The actor runtime checks each of the actors every `ScanIntervalInSeconds` to see if it can be garbage collected and collects it if it has been idle for `IdleTimeoutInSeconds`.

Anytime an actor is used, its idle time is reset to 0. After this, the actor can be garbage collected only if it again remains idle for `IdleTimeoutInSeconds`. Recall that an actor is considered to have been used if either an actor interface method an actor reminder callback is executed. An actor is **not** considered to have been used if its timer callback is executed.

The following diagram shows the lifecycle of a single actor to illustrate these concepts.

The example shows the impact of actor method calls, reminders, and timers on the lifetime of this actor. The following points about the example are worth mentioning:

- ScanInterval and IdleTimeout are set to 5 and 10 respectively. (Units do not matter here, since our purpose is only to illustrate the concept.)
- The scan for actors to be garbage collected happens at T=0,5,10,15,20,25, as defined by the scan interval of 5.
- A periodic timer fires at T=4,8,12,16,20,24, and its callback executes. It does not impact the idle time of the

actor.

- An actor method call at T=7 resets the idle time to 0 and delays the garbage collection of the actor.
- An actor reminder callback executes at T=14 and further delays the garbage collection of the actor.
- During the garbage collection scan at T=25, the actor's idle time finally exceeds the idle timeout of 10, and the actor is garbage collected.

An actor will never be garbage collected while it is executing one of its methods, no matter how much time is spent in executing that method. As mentioned earlier, the execution of actor interface methods and reminder callbacks prevents garbage collection by resetting the actor's idle time to 0. The execution of timer callbacks does not reset the idle time to 0. However, the garbage collection of the actor is deferred until the timer callback has completed execution.

Deleting actors and their state

Gabrage collection of deactivated actors only cleans up the actor object, but it does not remove data that is stored in an actor's State Manager. When an actor is re-activated, its data is again made available to it through the State Manager. In cases where actors store data in State Manager and are deactivated but never re-activated, it may be necessary to clean up their data.

The Actor Service provides a function for deleting actors from a remote caller:

```
ActorId actorToDelete = new ActorId(id);

IActorService myActorServiceProxy = ActorServiceProxy.Create(
    new Uri("fabric:/MyApp/MyService"), actorToDelete);

await myActorServiceProxy.DeleteActorAsync(actorToDelete, cancellationToken)
```

Deleting an actor has the following effects depending on whether or not the actor is currently active:

- **Active Actor**
  - Actor is removed from active actors list and is deactivated.
  - Its state is deleted permanently.
- **Inactive Actor**
  - Its state is deleted permanently.

Note that an actor cannot call delete on itself from one of its actor methods because the actor cannot be deleted while executing within an actor call context, in which the runtime has obtained a lock around the actor call to enforce single-threaded access.

Next steps

- Actor timers and reminders
- Actor events
- Actor reentrancy
- Actor diagnostics and performance monitoring
- Actor API reference documentation
- Sample code

Polymorphism in the Reliable Actors framework

11/15/2016 • 1 min to read • Edit on GitHub

Contributors

Sean McKenna • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Vaclav Turecek • katiecumming • Andy Pasic

The Reliable Actors framework allows you to build actors using many of the same techniques that you would use in object-oriented design. One of those techniques is polymorphism, which allows types and interfaces to inherit from more generalized parents. Inheritance in the Reliable Actors framework generally follows the .NET model with a few additional constraints.

Interfaces

The Reliable Actors framework requires you to define at least one interface to be implemented by your actor type. This interface is used to generate a proxy class that can be used by clients to communicate with your actors. Interfaces can inherit from other interfaces as long as every interface that is implemented by an actor type and all of its parents ultimately derive from IActor. IActor is the platform-defined base interface for actors. Thus, the classic polymorphism example using shapes might look something like this:

Types

You can also create a hierarchy of actor types, which are derived from the base Actor class that is provided by the platform. In the case of shapes, you might have a base `Shape` type:

```
public abstract class Shape : Actor, IShape
{
    public abstract Task<int> GetVerticeCount();

    public abstract Task<double> GetAreaAsync();
}
```

Subtypes of `Shape` can override methods from the base.

```
[ActorService(Name = "Circle")]
[StatePersistence(StatePersistence.Persisted)]
public class Circle : Shape, ICircle
{
    public override Task<int> GetVerticeCount()
    {
        return Task.FromResult(0);
    }

    public override async Task<double> GetAreaAsync()
    {
        CircleState state = await this.StateManager.GetStateAsync<CircleState>("circle");

        return Math.PI *
            state.Radius *
            state.Radius;
    }
}
```

Note the `ActorService` attribute on the actor type. This attribute tells the Reliable Actor framework that it should automatically create a service for hosting actors of this type. In some cases, you may wish to create a base type that is solely intended for sharing functionality with subtypes and will never be used to instantiate concrete actors. In those cases, you should use the `abstract` keyword to indicate that you will never create an actor based on that type.

Next steps

- See how the Reliable Actors framework leverages the Service Fabric platform to provide reliability, scalability, and consistent state.
- Learn about the actor lifecycle.

Reliable Actors reentrancy

11/15/2016 • 1 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Aman Bhardwaj • ShawnJackson • jessebenson • Andy Pasic • Bill Craun

---

The Reliable Actors runtime, by default, allows logical call context-based reentrancy. This allows for actors to be reentrant if they are in the same call context chain. For example, Actor A sends a message to Actor B, who sends a message to Actor C. As part of the message processing, if Actor C calls Actor A, the message is reentrant, so it will be allowed. Any other messages that are part of a different call context will be blocked on Actor A until it finishes processing.

There are two options available for actor reentrancy defined in the `ActorReentrancyMode` enum:

- `LogicalCallContext` (default behavior)
- `Disallowed` - disables reentrancy

```
public enum ActorReentrancyMode
{
    LogicalCallContext = 1,
    Disallowed = 2
}
```

Reentrancy can be configured in an `ActorService`'s settings during registration. The setting applies to all actor instances created in the actor service.

The following example shows an actor service that sets the reentrancy mode to `ActorReentrancyMode.Disallowed`. In this case, if an actor sends a reentrant message to another actor, an exception of type `FabricException` will be thrown.

```
static class Program
{
    static void Main()
    {
        try
        {
            ActorRuntime.RegisterActorAsync<Actor1>(
                (context, actorType) => new ActorService(
                    context,
                    actorType, () => new Actor1(),
                    settings: new ActorServiceSettings()
                    {
                        ActorConcurrencySettings = new ActorConcurrencySettings()
                        {
                            ReentrancyMode = ActorReentrancyMode.Disallowed
                        }
                    }))
                .GetAwaiter().GetResult();

            Thread.Sleep(Timeout.Infinite);
        }
        catch (Exception e)
        {
            ActorEventSource.Current.ActorHostInitializationFailed(e.ToString());
            throw;
        }
    }
}
```

Next steps

- Actor diagnostics and performance monitoring
- Actor API reference documentation
- Sample code

Actor timers and reminders

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Kevin • Chris Missal • Aman Bhardwaj • ShawnJackson • jessebenson • Andy Pasic • Bill Craun • Philip Taffet

Actors can schedule periodic work on themselves by registering either timers or reminders. This article shows how to use timers and reminders and explains the differences between them.

Actor timers

Actor timers provide a simple wrapper around .NET timer to ensure that the callback methods respect the turn-based concurrency guarantees that the Actors runtime provides.

Actors can use the `RegisterTimer` and `UnregisterTimer` methods on their base class to register and unregister their timers. The example below shows the use of timer APIs. The APIs are very similar to the .NET timer. In this example, when the timer is due, the Actors runtime will call the `MoveObject` method. The method is guaranteed to respect the turn-based concurrency. This means that no other actor methods or timer/reminder callbacks will be in progress until this callback completes execution.

```
class VisualObjectActor : Actor, IVisualObject
{
    private IActorTimer _updateTimer;

    public VisualObjectActor(ActorService actorService, ActorId actorId)
        : base(actorService, actorId)
    {
    }

    protected override Task OnActivateAsync()
    {
        ...

        _updateTimer = RegisterTimer(
            MoveObject,                     // Callback method
            null,                           // Parameter to pass to the callback method
            TimeSpan.FromMilliseconds(15),  // Amount of time to delay before the callback is invoked
            TimeSpan.FromMilliseconds(15)); // Time interval between invocations of the callback method

        return base.OnActivateAsync();
    }

    protected override Task OnDeactivateAsync()
    {
        if (_updateTimer != null)
        {
            UnregisterTimer(_updateTimer);
        }

        return base.OnDeactivateAsync();
    }

    private Task MoveObject(object state)
    {
        ...
        return Task.FromResult(true);
    }
}
```

The next period of the timer starts after the callback completes execution. This implies that the timer is stopped while the callback is executing and is started when the callback finishes.

The Actors runtime saves changes made to the actor's State Manager when the callback finishes. If an error occurs in saving the state, that actor object will be deactivated and a new instance will be activated.

All timers are stopped when the actor is deactivated as part of garbage collection. No timer callbacks are invoked after that. Also, the Actors runtime does not retain any information about the timers that were running before deactivation. It is up to the actor to register any timers that it needs when it is reactivated in the future. For more information, see the section on actor garbage collection.

Actor reminders

Reminders are a mechanism to trigger persistent callbacks on an actor at specified times. Their functionality is similar to timers. But unlike timers, reminders are triggered under all circumstances until the actor explicitly unregisters them or the actor is explicitly deleted. Specifically, reminders are triggered across actor deactivations and failovers because the Actors runtime persists information about the actor's reminders.

To register a reminder, an actor calls the `RegisterReminderAsync` method provided on the base class, as shown in the following example:

```
protected override async Task OnActivateAsync()
{
    string reminderName = "Pay cell phone bill";
    int amountInDollars = 100;

    IActorReminder reminderRegistration = await this.RegisterReminderAsync(
        reminderName,
        BitConverter.GetBytes(amountInDollars),
        TimeSpan.FromDays(3),
        TimeSpan.FromDays(1));
}
```

In this example, `"Pay cell phone bill"` is the reminder name. This is a string that the actor uses to uniquely identify a reminder. `BitConverter.GetBytes(amountInDollars)` is the context that is associated with the reminder. It will be passed back to the actor as an argument to the reminder callback, i.e. `IRemindable.ReceiveReminderAsync`.

Actors that use reminders must implement the `IRemindable` interface, as shown in the example below.

```
public class ToDoListActor : Actor, IToDoListActor, IRemindable
{
    public ToDoListActor(ActorService actorService, ActorId actorId)
        : base(actorService, actorId)
    {
    }

    public Task ReceiveReminderAsync(string reminderName, byte[] context, TimeSpan dueTime, TimeSpan period)
    {
        if (reminderName.Equals("Pay cell phone bill"))
        {
            int amountToPay = BitConverter.ToInt32(context, 0);
            System.Console.WriteLine("Please pay your cell phone bill of ${0}!", amountToPay);
        }
        return Task.FromResult(true);
    }
}
```

When a reminder is triggered, the Reliable Actors runtime will invoke the `ReceiveReminderAsync` method on the Actor. An actor can register multiple reminders, and the `ReceiveReminderAsync` method is invoked when any of those reminders is triggered. The actor can use the reminder name that is passed in to the `ReceiveReminderAsync` method to figure out which reminder was triggered.

The Actors runtime saves the actor's state when the `ReceiveReminderAsync` call finishes. If an error occurs in saving the state, that actor object will be deactivated and a new instance will be activated.

To unregister a reminder, an actor calls the `UnregisterReminderAsync` method, as shown in the example below.

```
IActorReminder reminder = GetReminder("Pay cell phone bill");
Task reminderUnregistration = UnregisterReminderAsync(reminder);
```

As shown above, the `UnregisterReminderAsync` method accepts an `IActorReminder` interface. The actor base class supports a `GetReminder` method that can be used to retrieve the `IActorReminder` interface by passing in the reminder name. This is convenient because the actor does not need to persist the `IActorReminder` interface that was returned from the `RegisterReminder` method call.

Next Steps

- Actor events
- Actor reentrancy
- Actor diagnostics and performance monitoring

- Actor API reference documentation
- Sample code

Actor events

11/15/2016 • 1 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Aman Bhardwaj • troyanov • ShawnJackson • jessebenson • Andy Pasic • Bill Craun

---

Actor events provide a way to send best-effort notifications from the actor to the clients. Actor events are designed for actor-to-client communication and should not be used for actor-to-actor communication.

The following code snippets show how to use actor events in your application.

Define an interface that describes the events published by the actor. This interface must be derived from the `IActorEvents` interface. The arguments of the methods must be data contract serializable. The methods must return void, as event notifications are one way and best effort.

```
public interface IGameEvents : IActorEvents
{
    void GameScoreUpdated(Guid gameId, string currentScore);
}
```

Declare the events published by the actor in the actor interface.

```
public interface IGameActor : IActor, IActorEventPublisher<IGameEvents>
{
    Task UpdateGameStatus(GameStatus status);

    Task<string> GetGameScore();
}
```

On the client side, implement the event handler.

```
class GameEventsHandler : IGameEvents
{
    public void GameScoreUpdated(Guid gameId, string currentScore)
    {
        Console.WriteLine(@"Updates: Game: {0}, Score: {1}", gameId, currentScore);
    }
}
```

On the client, create a proxy to the actor that publishes the event and subscribe to its events.

```
var proxy = ActorProxy.Create<IGameActor>(
                    new ActorId(Guid.Parse(arg)), ApplicationName);

await proxy.SubscribeAsync<IGameEvents>(new GameEventsHandler());
```

In the event of failovers, the actor may fail over to a different process or node. The actor proxy manages the active subscriptions and automatically re-subscribes them. You can control the re-subscription interval through the `ActorProxyEventExtensions.SubscribeAsync<TEvent>` API. To unsubscribe, use the `ActorProxyEventExtensions.UnsubscribeAsync<TEvent>` API.

On the actor, simply publish the events as they happen. If there are subscribers to the event, the Actors runtime

will send them the notification.

```
var ev = GetEvent<IGameEvents>();
ev.GameScoreUpdated(Id.GetGuidId(), score);
```

Next steps

- Actor reentrancy
- Actor diagnostics and performance monitoring
- Actor API reference documentation
- Sample code

Reliable Actors state management

11/15/2016 • 7 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Luis Delgado

Reliable Actors are single-threaded objects that can encapsulate both logic and state. Since actors run on Reliable Services, they can maintain state reliably using the same persistence and replication mechanisms used by Reliable Services. This way, actors don't lose their state after failures, re-activation after garbage collection, or when they are moved around between nodes in a cluster due to resource balancing or upgrades.

State persistence and replication

All Reliable Actors are considered *stateful* because each actor instance maps to a unique ID. This means that repeated calls to the same actor ID will be routed to the same actor instance. This is in contrast to a stateless system in which client calls are not guaranteed to be routed to the same server every time. For this reason, actor services are always stateful services.

However, even though actors are considered stateful, it does not mean they must store state reliably. Actors can choose the level of state persistence and replication based on their data storage requirements:

- **Persisted state:** State is persisted to disk and is replicated to 3 or more replicas. This is the most durable state storage option, where state can persist through complete cluster outage.
- **Volatile state:** State is replicated to 3 or more replicas and only kept in memory. This provides resilience against node failure, actor failure, and during upgrades and resource balancing. However, state is not persisted to disk, so if all replicas are lost at once, the state is lost as well.
- **No persisted state:** State is not replicated nor is it written to disk. For actors that simply don't need to maintain state reliably.

Each level of persistence is simply a different *state provider* and *replication* configuration of your service. Whether or not state is written to disk depends on the *state provider* - the component in a Reliable Service that stores state - and replication depends on how many replicas a service is deployed with. Just as with Reliable Services, both the state provider and replica count can easily be set manually. The actor framework provides an attribute, that, when used on an actor will automatically select a default state provider and auto-generate settings for replica count to achieve one of these three persistence settings.

Persisted state

```
[StatePersistence(StatePersistence.Persisted)]
class MyActor : Actor, IMyActor
{
}
```

This setting uses a state provider that stores data on disk and automatically sets the service replica count to 3.

Volatile state

```
[StatePersistence(StatePersistence.Volatile)]
class MyActor : Actor, IMyActor
{
}
```

This setting uses an in-memory-only state provider and sets the replica count to 3.

No persisted state

```
[StatePersistence(StatePersistence.None)]
class MyActor : Actor, IMyActor
{
}
```

This setting uses an in-memory-only state provider and sets the replica count to 1.

Defaults and generated settings

When using the `StatePersistence` attribute, a state provider is automatically selected for you at runtime when the actor service starts. The replica count, however, is set at compile time by the Visual Studio actor build tools. The build tools automatically generate a *default service* for the actor service in ApplicationManifest.xml. Parameters are created for **min replica set size** and **target replica set size**. You can of course change these parameters manually, however each time the `StatePersistence` attribute is changed, the parameters will be set to the default replica set size values for the selected `StatePersistence` attribute, overriding any previous values. In other words, the values you set in ServiceManifest.xml will **only** be overridden at build time when you change the `StatePersistence` attribute value.

```
<ApplicationManifest xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" ApplicationTypeName="Application12Type" ApplicationTypeVersion="1.0.0"
xmlns="http://schemas.microsoft.com/2011/01/fabric">
   <Parameters>
      <Parameter Name="MyActorService_PartitionCount" DefaultValue="10" />
      <Parameter Name="MyActorService_MinReplicaSetSize" DefaultValue="3" />
      <Parameter Name="MyActorService_TargetReplicaSetSize" DefaultValue="3" />
   </Parameters>
   <ServiceManifestImport>
      <ServiceManifestRef ServiceManifestName="MyActorPkg" ServiceManifestVersion="1.0.0" />
   </ServiceManifestImport>
   <DefaultServices>
      <Service Name="MyActorService" GeneratedIdRef="77d965dc-85fb-488c-bd06-c6c1fe29d593|Persisted">
         <StatefulService ServiceTypeName="MyActorServiceType" TargetReplicaSetSize="
[MyActorService_TargetReplicaSetSize]" MinReplicaSetSize="[MyActorService_MinReplicaSetSize]">
            <UniformInt64Partition PartitionCount="[MyActorService_PartitionCount]" LowKey="-
9223372036854775808" HighKey="9223372036854775807" />
         </StatefulService>
      </Service>
   </DefaultServices>
</ApplicationManifest>
```

State Manager

Every actor instance has its own State Manager: A dictionary-like data structure that reliably stores key-value pairs. The State Manager is a wrapper around a state provider. It can be used to store data regardless of which persistence setting is used, but it does not provide any guarantees that a running actor service can be changed from a volatile (in-memory-only) state setting to a persisted state setting through a rolling upgrade while preserving data. However, it is possible to change replica count for a running service.

State Manager keys must be strings, while values are generic and can be any type, including custom types. Values stored in the State Manager must be Data Contract serializable because they may be transmitted over the network to other nodes during replication and may be written to disk, depending on an actor's state persistence setting.

The State Manager exposes common dictionary methods for managing state, similar to those found in Reliable Dictionary.

Accessing state

State can be accessed through the State Manager by key. State Manager methods are all asynchronous as they may require disk I/O when actors have persisted state. Upon first access, state objects are cached in memory. Repeat access operations access objects directly from memory and return synchronously without incurring disk I/O or asynchronous context switching overhead. A state object is removed from the cache in the following cases:

- An actor method throws an unhandled exception after retrieving an object from the State Manager.
- An actor is re-activated, either after being deactivated or due to failure.
- If the state provider pages state to disk. This behavior depends on the state provider implementation. The default state provider for the `Persisted` setting has this behavior.

State can be retrieved using a standard *Get* operation that throws `KeyNotFoundException` if an entry does not exist for the given key:

```
[StatePersistence(StatePersistence.Persisted)]
class MyActor : Actor, IMyActor
{
    public MyActor(ActorService actorService, ActorId actorId)
        : base(actorService, actorId)
    {
    }

    public Task<int> GetCountAsync()
    {
        return this.StateManager.GetStateAsync<int>("MyState");
    }
}
```

State can also be retrieved using a *TryGet* method that does not throw if an entry does not exist for a given key:

```
class MyActor : Actor, IMyActor
{
    public MyActor(ActorService actorService, ActorId actorId)
        : base(actorService, actorId)
    {
    }

    public async Task<int> GetCountAsync()
    {
        ConditionalValue<int> result = await this.StateManager.TryGetStateAsync<int>("MyState");
        if (result.HasValue)
        {
            return result.Value;
        }

        return 0;
    }
}
```

Saving state

The State Manager retrieval methods return a reference to an object in local memory. Modifying this object in local memory alone does not cause it to be saved durably. When an object is retrieved from the State Manager and modified, it must be re-inserted into the State Manager to be saved durably.

State can be inserted using an unconditional *Set*, which is the equivalent of the `dictionary["key"] = value` syntax:

```
[StatePersistence(StatePersistence.Persisted)]
class MyActor : Actor, IMyActor
{
    public MyActor(ActorService actorService, ActorId actorId)
        : base(actorService, actorId)
    {
    }

    public Task SetCountAsync(int value)
    {
        return this.StateManager.SetStateAsync<int>("MyState", value);
    }
}
```

State can be added using an *Add* method, which will throw `InvalidOperationException` when trying to add a key that already exists:

```
[StatePersistence(StatePersistence.Persisted)]
class MyActor : Actor, IMyActor
{
    public MyActor(ActorService actorService, ActorId actorId)
        : base(actorService, actorId)
    {
    }

    public Task AddCountAsync(int value)
    {
        return this.StateManager.AddStateAsync<int>("MyState", value);
    }
}
```

State can also be added using a *TryAdd* method, which will not throw when trying to add a key that already exists:

```
[StatePersistence(StatePersistence.Persisted)]
class MyActor : Actor, IMyActor
{
    public MyActor(ActorService actorService, ActorId actorId)
        : base(actorService, actorId)
    {
    }

    public async Task AddCountAsync(int value)
    {
        bool result = await this.StateManager.TryAddStateAsync<int>("MyState", value);

        if (result)
        {
            // Added successfully!
        }
    }
}
```

At the end of an actor method, the State Manager automatically saves any values that have been added or modified by an insert or update operation. A "save" can include persisting to disk and replication, depending on the settings used. Values that have not been modified are not persisted or replicated. If no values have been modified, the save operation does nothing. In the event that saving fails, the modified state is discarded and the original state is reloaded.

State can also be saved manually be calling the `SaveStateAsync` method on the actor base:

```
async Task IMyActor.SetCountAsync(int count)
{
    await this.StateManager.AddOrUpdateStateAsync("count", count, (key, value) => count > value ? count :
value);

    await this.SaveStateAsync();
}
```

Removing state

State can be removed permanently from an actor's State Manager by calling the *Remove* method. This method will throw `KeyNotFoundException` when trying to remove a key that doesn't exist:

```
[StatePersistence(StatePersistence.Persisted)]
class MyActor : Actor, IMyActor
{
    public MyActor(ActorService actorService, ActorId actorId)
        : base(actorService, actorId)
    {
    }

    public Task RemoveCountAsync()
    {
        return this.StateManager.RemoveStateAsync("MyState");
    }
}
```

State can also be removed permanently by using the *TryRemove* method, which will not throw when trying to remove a key that doesn't exist:

```
[StatePersistence(StatePersistence.Persisted)]
class MyActor : Actor, IMyActor
{
    public MyActor(ActorService actorService, ActorId actorId)
        : base(actorService, actorId)
    {
    }

    public async Task RemoveCountAsync()
    {
        bool result = await this.StateManager.TryRemoveStateAsync("MyState");

        if (result)
        {
            // State removed!
        }
    }
}
```

Next steps

- Actor type serialization
- Actor polymorphism and object-oriented design patterns
- Actor diagnostics and performance monitoring
- Actor API reference documentation
- Sample code

Configuring Reliable Actors--KVSActorStateProvider

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

Sumukh Shivaprakash • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • stevehug • Andy De George • Ryan Wike

---

You can modify the default configuration of KVSActorStateProvider by changing the settings.xml file that is generated in the Microsoft Visual Studio package root under the Config folder for the specified actor.

The Azure Service Fabric runtime looks for predefined section names in the settings.xml file and consumes the configuration values while creating the underlying runtime components.

> **NOTE**
> Do **not** delete or modify the section names of the following configurations in the settings.xml file that is generated in the Visual Studio solution.

Replicator security configuration

Replicator security configurations are used to secure the communication channel that is used during replication. This means that services cannot see each other's replication traffic, ensuring that the data that is made highly available is also secure. By default, an empty security configuration section prevents replication security.

Section name

<ActorName>ServiceReplicatorSecurityConfig

Replicator configuration

Replicator configurations configure the replicator that is responsible for making the Actor State Provider state highly reliable. The default configuration is generated by the Visual Studio template and should suffice. This section talks about additional configurations that are available to tune the replicator.

Section name

<ActorName>ServiceReplicatorConfig

Configuration names

NAME	UNIT	DEFAULT VALUE	REMARKS
BatchAcknowledgementInterval	Seconds	0.015	Time period for which the replicator at the secondary waits after receiving an operation before sending back an acknowledgement to the primary. Any other acknowledgements to be sent for operations processed within this interval are sent as one response.

NAME	UNIT	DEFAULT VALUE	REMARKS
ReplicatorEndpoint	N/A	No default--required parameter	IP address and port that the primary/secondary replicator will use to communicate with other replicators in the replica set. This should reference a TCP resource endpoint in the service manifest. Refer to Service manifest resources to read more about defining endpoint resources in the service manifest.
RetryInterval	Seconds	5	Time period after which the replicator re-transmits a message if it does not receive an acknowledgement for an operation.
MaxReplicationMessageSize	Bytes	50 MB	Maximum size of replication data that can be transmitted in a single message.
MaxPrimaryReplicationQueueSize	Number of operations	1024	Maximum number of operations in the primary queue. An operation is freed up after the primary replicator receives an acknowledgement from all the secondary replicators. This value must be greater than 64 and a power of 2.
MaxSecondaryReplicationQueueSize	Number of operations	2048	Maximum number of operations in the secondary queue. An operation is freed up after making its state highly available through persistence. This value must be greater than 64 and a power of 2.

Store configuration

Store configurations are used to configure the local store that is used to persist the state that is being replicated. The default configuration is generated by the Visual Studio template and should suffice. This section talks about additional configurations that are available to tune the local store.

Section name

<ActorName>ServiceLocalStoreConfig

Configuration names

NAME	UNIT	DEFAULT VALUE	REMARKS

NAME	UNIT	DEFAULT VALUE	REMARKS
MaxAsyncCommitDelayInMilliseconds	Milliseconds	200	Sets the maximum batching interval for durable local store commits.
MaxVerPages	Number of pages	16384	The maximum number of version pages in the local store database. It determines the maximum number of outstanding transactions.

Sample configuration file

```
<?xml version="1.0" encoding="utf-8"?>
<Settings xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.microsoft.com/2011/01/fabric">
   <Section Name="MyActorServiceReplicatorConfig">
      <Parameter Name="ReplicatorEndpoint" Value="MyActorServiceReplicatorEndpoint" />
      <Parameter Name="BatchAcknowledgementInterval" Value="0.05"/>
   </Section>
   <Section Name="MyActorServiceLocalStoreConfig">
      <Parameter Name="MaxVerPages" Value="8192" />
   </Section>
   <Section Name="MyActorServiceReplicatorSecurityConfig">
      <Parameter Name="CredentialType" Value="X509" />
      <Parameter Name="FindType" Value="FindByThumbprint" />
      <Parameter Name="FindValue" Value="9d c9 06 b1 69 dc 4f af fd 16 97 ac 78 1e 80 67 90 74 9d 2f" />
      <Parameter Name="StoreLocation" Value="LocalMachine" />
      <Parameter Name="StoreName" Value="My" />
      <Parameter Name="ProtectionLevel" Value="EncryptAndSign" />
      <Parameter Name="AllowedCommonNames" Value="My-Test-SAN1-Alice,My-Test-SAN1-Bob" />
   </Section>
</Settings>
```

Remarks

The BatchAcknowledgementInterval parameter controls replication latency. A value of '0' results in the lowest possible latency, at the cost of throughput (as more acknowledgement messages must be sent and processed, each containing fewer acknowledgements). The larger the value for BatchAcknowledgementInterval, the higher the overall replication throughput, at the cost of higher operation latency. This directly translates to the latency of transaction commits.

Notes on Service Fabric Reliable Actors type serialization

11/15/2016 • 1 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Theano Petersen • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • MarkDeichmiller • Ryan Wike • clca • jessebenson

The arguments of all methods, result types of the tasks returned by each method in an actor interface, and objects stored in an actor's State Manager must be Data Contract serializable.. This also applies to the arguments of the methods defined in actor event interfaces. (Actor event interface methods always return void.)

Custom data types

In this example, the following actor interface defines a method that returns a custom data type called `VoicemailBox` .

```
public interface IVoiceMailBoxActor : IActor
{
    Task<VoicemailBox> GetMailBoxAsync();
}
```

The interface is impelemented by an actor, which uses the State Manager to store a `VoicemailBox` object:

```
[StatePersistence(StatePersistence.Persisted)]
public class VoiceMailBoxActor : Actor, IVoicemailBoxActor
{
    public VoiceMailBoxActor(ActorService actorService, ActorId actorId)
        : base(actorService, actorId)
    {
    }

    public Task<VoicemailBox> GetMailboxAsync()
    {
        return this.StateManager.GetStateAsync<VoicemailBox>("Mailbox");
    }
}
```

In this example, the `VoicemailBox` object is serialized when:

- The object is transmitted between an actor instance and a caller.
- The object is saved in the State Manager where it is persisted to disk and replicated to other nodes.

The Reliable Actor framework uses DataContract serialization. Therefore, the custom data objects and their members must be annotated with the **DataContract** and **DataMember** attributes, respectively

```
[DataContract]
public class Voicemail
{
    [DataMember]
    public Guid Id { get; set; }

    [DataMember]
    public string Message { get; set; }

    [DataMember]
    public DateTime ReceivedAt { get; set; }
}
```

```
[DataContract]
public class VoicemailBox
{
    public VoicemailBox()
    {
        this.MessageList = new List<Voicemail>();
    }

    [DataMember]
    public List<Voicemail> MessageList { get; set; }

    [DataMember]
    public string Greeting { get; set; }
}
```

Next steps

- Actor lifecycle and garbage collection
- Actor timers and reminders
- Actor events
- Actor reentrancy
- Actor polymorphism and object-oriented design patterns
- Actor diagnostics and performance monitoring

Deploy a guest executable to Service Fabric

11/15/2016 • 13 min to read • Edit on GitHub

Contributors

Mark Fussell • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • ShawnJackson • James Dunn • carolinacmoravia • Dhanashri Kshirsagar • PRmerger • v-anacga • Ming Han Teh • Mikkel Mork Hegnhoj • Ryan Wike • Stephane Lapointe • katiecumming • Sean McKenna • clca

---

You can run any type of application, such as Node.js, Java, or native applications in Azure Service Fabric. Service Fabric refers to these types of applications as guest executables. Guest executables are treated by Service Fabric like stateless services. As a result, they are placed on nodes in a cluster, based on availability and other metrics. This article describes how to package and deploy a guest executable to a Service Fabric cluster, by using Visual Studio or a command-line utility.

In this article, we cover the steps to package a guest executable and deploy it to Service Fabric.

Benefits of running a guest executable in Service Fabric

There are several advantages to running a guest executable in a Service Fabric cluster:

- High availability. Applications that run in Service Fabric are made highly available. Service Fabric ensures that instances of an application are running.
- Health monitoring. Service Fabric health monitoring detects if an application is running, and provides diagnostic information if there is a failure.
- Application lifecycle management. Besides providing upgrades with no downtime, Service Fabric provides automatic rollback to the previous version if there is a bad health event reported during an upgrade.
- Density. You can run multiple applications in a cluster, which eliminates the need for each application to run on its own hardware.

Overview of application and service manifest files

As part of deploying a guest executable, it is useful to understand the Service Fabric packaging and deployment model as described in application model. The Service Fabric packaging model relies on two XML files: the application and service manifests. The schema definition for the ApplicationManifest.xml and ServiceManifest.xml files is installed with the Service Fabric SDK into *C:\Program Files\Microsoft SDKs\Service Fabric\schemas\ServiceFabricServiceModel.xsd*.

- **Application manifest** The application manifest is used to describe the application. It lists the services that compose it, and other parameters that are used to define how one or more services should be deployed, such as the number of instances.

  In Service Fabric, an application is a unit of deployment and upgrade. An application can be upgraded as a single unit where potential failures and potential rollbacks are managed. Service Fabric guarantees that the upgrade process is either successful, or, if the upgrade fails, does not leave the application in an unknown or unstable state.

- **Service manifest** The service manifest describes the components of a service. It includes data, such as the name and type of service, and its code and configuration. The service manifest also includes some additional parameters that can be used to configure the service once it is deployed.

Application package file structure

To deploy an application to Service Fabric, the application should follow a predefined directory structure. The following is an example of that structure.

```
|-- ApplicationPackageRoot
    |-- GuestService1Pkg
        |-- Code
            |-- existingapp.exe
        |-- Config
            |-- Settings.xml
        |-- Data
        |-- ServiceManifest.xml
    |-- ApplicationManifest.xml
```

The ApplicationPackageRoot contains the ApplicationManifest.xml file that defines the application. A subdirectory for each service included in the application is used to contain all the artifacts that the service requires. These subdirectories are the ServiceManifest.xml and, typically, the following:

- *Code*. This directory contains the service code.
- *Config*. This directory contains a Settings.xml file (and other files if necessary) that the service can access at runtime to retrieve specific configuration settings.
- *Data*. This is an additional directory to store additional local data that the service may need. Data should be used to store only ephemeral data. Service Fabric does not copy or replicate changes to the data directory if the service needs to be relocated (for example, during failover).

> **NOTE**
> You don't have to create the `config` and `data` directories if you don't need them.

Package an existing executable

When packaging a guest executable, you can choose either to use a Visual Studio project template or to create the application package manually. Using Visual Studio, the application package structure and manifest files are created by the new project template for you.

> **TIP**
> The easiest way to package an existing Windows executable into a service is to use Visual Studio.

Use Visual Studio to package an existing executable

Visual Studio provides a Service Fabric service template to help you deploy a guest executable to a Service Fabric cluster.

1. Choose **File** > **New Project**, and create a Service Fabric application.
2. Choose **Guest Executable** as the service template.
3. Click **Browse** to select the folder with your executable and fill in the rest of the parameters to create the service.
   - *Code Package Behavior*. Can be set to copy all the content of your folder to the Visual Studio Project, which is useful if the executable does not change. If you expect the executable to change and want the ability to pick up new builds dynamically, you can choose to link to the folder instead. Note that you can use linked folders when creating the application project in Visual Studio. This links to the source location from within the project, making it possible for you to update the guest executable in its source

destination. Those updates become part of the application package on build.

- *Program* specifies the executable that should be run to start the service.
- *Arguments* specifies the arguments that should be passed to the executable. It can be a list of parameters with arguments.
- *WorkingFolder* specifies the working directory for the process that is going to be started. You can specify three values:
  - `CodeBase` specifies that the working directory is going to be set to the code directory in the application package (`Code` directory shown in the preceding file structure).
  - `CodePackage` specifies that the working directory is going to be set to the root of the application package (`GuestService1Pkg` shown in the preceding file structure).
  - `Work` specifies that the files are placed in a subdirectory called work.

4. Give your service a name, and click **OK**.
5. If your service needs an endpoint for communication, you can now add the protocol, port, and type to the ServiceManifest.xml file. For example:

```
<Endpoint Name="NodeAppTypeEndpoint" Protocol="http" Port="3000" UriScheme="http" PathSuffix="myapp/"
Type="Input" />
```

.

6. You can now use the package and publish action against your local cluster by debugging the solution in Visual Studio. When ready, you can publish the application to a remote cluster or check in the solution to source control.
7. Go to the end of this article to see how to view your guest executable service running in Service Fabric Explorer.

Manually package and deploy an existing executable

The process of manually packaging a guest executable is based on the following general steps:

1. Create the package directory structure.
2. Add the application's code and configuration files.
3. Edit the service manifest file.
4. Edit the application manifest file.

Create the package directory structure

You can start by creating the directory structure, as described in the preceding section, "Application package file structure."

Add the application's code and configuration files

After you have created the directory structure, you can add the application's code and configuration files under the code and config directories. You can also create additional directories or subdirectories under the code or config directories.

Service Fabric does an xcopy of the content of the application root directory, so there is no predefined structure to use other than creating two top directories, code and settings. (You can pick different names if you want. More details are in the next section.)

> **NOTE**
>
> Make sure that you include all the files and dependencies that the application needs. Service Fabric copies the content of the application package on all nodes in the cluster where the application's services are going to be deployed. The package should contain all the code that the application needs to run. Do not assume that the dependencies are already installed.

Edit the service manifest file

The next step is to edit the service manifest file to include the following information:

- The name of the service type. This is an ID that Service Fabric uses to identify a service.
- The command to use to launch the application (ExeHost).
- Any script that needs to be run to set up the application (SetupEntrypoint).

The following is an example of a `ServiceManifest.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceManifest xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" Name="NodeApp" Version="1.0.0.0" xmlns="http://schemas.microsoft.com/2011/01/fabric">
   <ServiceTypes>
      <StatelessServiceType ServiceTypeName="NodeApp" UseImplicitHost="true"/>
   </ServiceTypes>
   <CodePackage Name="code" Version="1.0.0.0">
      <SetupEntryPoint>
         <ExeHost>
             <Program>scripts\launchConfig.cmd</Program>
         </ExeHost>
      </SetupEntryPoint>
      <EntryPoint>
         <ExeHost>
            <Program>node.exe</Program>
            <Arguments>bin/www</Arguments>
            <WorkingFolder>CodePackage</WorkingFolder>
         </ExeHost>
      </EntryPoint>
   </CodePackage>
   <Resources>
      <Endpoints>
         <Endpoint Name="NodeAppTypeEndpoint" Protocol="http" Port="3000" Type="Input" />
      </Endpoints>
   </Resources>
</ServiceManifest>
```

The following sections go over the different parts of the file that you need to update.

Update ServiceTypes

```
<ServiceTypes>
  <StatelessServiceType ServiceTypeName="NodeApp" UseImplicitHost="true" />
</ServiceTypes>
```

- You can pick any name that you want for `ServiceTypeName`. The value is used in the `ApplicationManifest.xml` file to identify the service.
- Specify `UseImplicitHost="true"`. This attribute tells Service Fabric that the service is based on a self-contained app, so all Service Fabric needs to do is to launch it as a process and monitor its health.

Update CodePackage

The CodePackage element specifies the location (and version) of the service's code.

```
<CodePackage Name="Code" Version="1.0.0.0">
```

The `Name` element is used to specify the name of the directory in the application package that contains the service's code. `CodePackage` also has the `version` attribute. This can be used to specify the version of the code, and can also potentially be used to upgrade the service's code by using the application lifecycle management infrastructure in Service Fabric.

Optional: Update SetupEntrypoint

```
<SetupEntryPoint>
   <ExeHost>
       <Program>scripts\launchConfig.cmd</Program>
   </ExeHost>
</SetupEntryPoint>
```

The SetupEntryPoint element is used to specify any executable or batch file that should be executed before the service's code is launched. It is an optional step, so it does not need to be included if there is no initialization required. The SetupEntryPoint is executed every time the service is restarted.

There is only one SetupEntryPoint, so setup scripts need to be grouped in a single batch file if the application's setup requires multiple scripts. The SetupEntryPoint can execute any type of file: executable files, batch files, and PowerShell cmdlets. For more details, see Configure SetupEntryPoint.

In the preceding example, the SetupEntryPoint runs a batch file called `LaunchConfig.cmd` that is located in the `scripts` subdirectory of the code directory (assuming the WorkingFolder element is set to CodeBase).

Update EntryPoint

```
<EntryPoint>
  <ExeHost>
    <Program>node.exe</Program>
    <Arguments>bin/www</Arguments>
    <WorkingFolder>CodeBase</WorkingFolder>
  </ExeHost>
</EntryPoint>
```

The `EntryPoint` element in the service manifest file is used to specify how to launch the service. The `ExeHost` element specifies the executable (and arguments) that should be used to launch the service.

- `Program` specifies the name of the executable that should start the service.
- `Arguments` specifies the arguments that should be passed to the executable. It can be a list of parameters with arguments.
- `WorkingFolder` specifies the working directory for the process that is going to be started. You can specify three values:
  - `CodeBase` specifies that the working directory is going to be set to the code directory in the application package (`Code` directory in the preceding file structure).
  - `CodePackage` specifies that the working directory is going to be set to the root of the application package (`GuestService1Pkg` in the preceding file structure).
    - `Work` specifies that the files are placed in a subdirectory called work.

The WorkingFolder is useful to set the correct working directory so that relative paths can be used by either the application or initialization scripts.

Update Endpoints and register with Naming Service for communication

```
<Endpoints>
   <Endpoint Name="NodeAppTypeEndpoint" Protocol="http" Port="3000" Type="Input" />
</Endpoints>
```

In the preceding example, the `Endpoint` element specifies the endpoints that the application can listen on. In this example, the Node.js application listens on http on port 3000.

Furthermore you can ask Service Fabric to publish this endpoint to the Naming Service so other services can discover the endpoint address to this service. This enables you to be able to communicate between services that are guest executables. The published endpoint address is of the form `UriScheme://IPAddressOrFQDN:Port/PathSuffix` . `UriScheme` and `PathSuffix` are optional attributes. `IPAddressOrFQDN` is the IP address or fully qualified domain name of the node this executable gets placed on, and it

is calculated for you.

In the following example, once the service is deployed, in Service Fabric Explorer you see an endpoint similar to `http://10.1.4.92:3000/myapp/` published for the service instance. Or if this is a local machine, you see `http://localhost:3000/myapp/`.

```
<Endpoints>
   <Endpoint Name="NodeAppTypeEndpoint" Protocol="http" Port="3000"  UriScheme="http" PathSuffix="myapp/"
Type="Input" />
</Endpoints>
```

You can use these addresses with reverse proxy to communicate between services.

Edit the application manifest file

Once you have configured the `Servicemanifest.xml` file, you need to make some changes to the `ApplicationManifest.xml` file to ensure that the correct service type and name are used.

```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationManifest xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" ApplicationTypeName="NodeAppType" ApplicationTypeVersion="1.0"
xmlns="http://schemas.microsoft.com/2011/01/fabric">
   <ServiceManifestImport>
      <ServiceManifestRef ServiceManifestName="NodeApp" ServiceManifestVersion="1.0.0.0" />
   </ServiceManifestImport>
</ApplicationManifest>
```

ServiceManifestImport

In the `ServiceManifestImport` element, you can specify one or more services that you want to include in the app. Services are referenced with `ServiceManifestName`, which specifies the name of the directory where the `ServiceManifest.xml` file is located.

```
<ServiceManifestImport>
  <ServiceManifestRef ServiceManifestName="NodeApp" ServiceManifestVersion="1.0.0.0" />
</ServiceManifestImport>
```

Set up logging

For guest executables, it is useful to be able to see console logs to find out if the application and configuration scripts show any errors. Console redirection can be configured in the `ServiceManifest.xml` file using the `ConsoleRedirection` element.

```
<EntryPoint>
  <ExeHost>
    <Program>node.exe</Program>
    <Arguments>bin/www</Arguments>
    <WorkingFolder>CodeBase</WorkingFolder>
    <ConsoleRedirection FileRetentionCount="5" FileMaxSizeInKb="2048"/>
  </ExeHost>
</EntryPoint>
```

`ConsoleRedirection` can be used to redirect console output (both stdout and stderr) to a working directory. This provides the ability to verify that there are no errors during the setup or execution of the application in the Service Fabric cluster.

`FileRetentionCount` determines how many files are saved in the working directory. A value of 5, for example, means that the log files for the previous five executions are stored in the working directory.

`FileMaxSizeInKb` specifies the maximum size of the log files.

Log files are saved in one of the service's working directories. To determine where the files are located, use Service Fabric Explorer to determine which node the service is running on, and which working directory is being used. This process is covered later in this article.

Deployment

The last step is to deploy your application. The following PowerShell script shows how to deploy your application to the local development cluster, and start a new Service Fabric service.

```
Connect-ServiceFabricCluster localhost:19000

Write-Host 'Copying application package...'
Copy-ServiceFabricApplicationPackage -ApplicationPackagePath 'C:\Dev\MultipleApplications' -
ImageStoreConnectionString 'file:C:\SfDevCluster\Data\ImageStoreShare' -ApplicationPackagePathInImageStore
'nodeapp'

Write-Host 'Registering application type...'
Register-ServiceFabricApplicationType -ApplicationPathInImageStore 'nodeapp'

New-ServiceFabricApplication -ApplicationName 'fabric:/nodeapp' -ApplicationTypeName 'NodeAppType' -
ApplicationTypeVersion 1.0

New-ServiceFabricService -ApplicationName 'fabric:/nodeapp' -ServiceName 'fabric:/nodeapp/nodeappservice' -
ServiceTypeName 'NodeApp' -Stateless -PartitionSchemeSingleton -InstanceCount 1
```

A Service Fabric service can be deployed in various "configurations." For example, it can be deployed as single or multiple instances, or it can be deployed in such a way that there is one instance of the service on each node of the Service Fabric cluster.

The `InstanceCount` parameter of the `New-ServiceFabricService` cmdlet is used to specify how many instances of the service should be launched in the Service Fabric cluster. You can set the `InstanceCount` value, depending on the type of application that you are deploying. The two most common scenarios are:

- `InstanceCount = "1"`. In this case, only one instance of the service is deployed in the cluster. Service Fabric's scheduler determines which node the service is going to be deployed on.
- `InstanceCount ="-1"`. In this case, one instance of the service is deployed on every node in the Service Fabric cluster. The result is having one (and only one) instance of the service for each node in the cluster.

This is a useful configuration for front-end applications (for example, a REST endpoint), because client applications need to "connect" to any of the nodes in the cluster to use the endpoint. This configuration can also be used when, for example, all nodes of the Service Fabric cluster are connected to a load balancer. Client traffic can then be distributed across the service that is running on all nodes in the cluster.

Check your running application

In Service Fabric Explorer, identify the node where the service is running. In this example, it runs on Node1:

If you navigate to the node and browse to the application, you see the essential node information, including its location on disk.

If you browse to the directory by using Server Explorer, you can find the working directory and the service's log folder, as shown in the following screenshot.

Next steps

In this article, you have learned how to package a guest executable and deploy it to Service Fabric. See the following articles for related information and tasks.

- Sample for packaging and deploying a guest executable on GitHub, including a link to the prerelease of the packaging tool
- Deploy multiple guest executables
- Create your first Service Fabric application using Visual Studio

Deploy multiple guest executables

11/15/2016 • 5 min to read • Edit on GitHub

Contributors

Mark Fussell • Theano Petersen • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Mikkel Mork Hegnhoj • Anthony Chu • katiecumming • coreysa

This article shows how to package and deploy multiple guest executables to Azure Service Fabric. For building and deploying a single Service Fabric package read how to deploy a guest executable to Service Fabric.

While this walkthrough shows how to deploy an application with a Node.js front end that uses MongoDB as the data store, you can apply the steps to any application that has dependencies on another application.

You can use Visual Studio to produce the application package that contains multiple guest executables. See Using Visual Studio to package an existing application. After you have added the first guest executable, right click on the application project and select the **Add->New Service Fabric service** to add the second guest executable project to the solution. Note: If you choose to link the source in the Visual Studio project, building the Visual Studio solution, will make sure that your application package is up to date with changes in the source.

Manually package the multiple guest executable application

Alternatively you can manually package the guest executable. For the manual packaging, this article uses the Service Fabric packaging tool, which is available at http://aka.ms/servicefabricpacktool.

Packaging the Node.js application

This article assumes that Node.js is not installed on the nodes in the Service Fabric cluster. As a consequence, you need to add Node.exe to the root directory of your node application before packaging. The directory structure of the Node.js application (using Express web framework and Jade template engine) should look similar to the one below:

```
|-- NodeApplication
    |-- bin
        |-- www
    |-- node_modules
        |-- .bin
        |-- express
        |-- jade
        |-- etc.
    |-- public
        |-- images
        |-- etc.
    |-- routes
        |-- index.js
        |-- users.js
    |-- views
        |-- index.jade
        |-- etc.
    |-- app.js
    |-- package.json
    |-- node.exe
```

As a next step, you create an application package for the Node.js application. The code below creates a Service Fabric application package that contains the Node.js application.

```
.\ServiceFabricAppPackageUtil.exe /source:'[yourdirectory]\MyNodeApplication' /target:'[yourtargetdirectory]
/appname:NodeService /exe:'node.exe' /ma:'bin/www' /AppType:NodeAppType
```

Below is a description of the parameters that are being used:

- **/source** points to the directory of the application that should be packaged.
- **/target** defines the directory in which the package should be created. This directory has to be different from the source directory.
- **/appname** defines the application name of the existing application. It's important to understand that this translates to the service name in the manifest, and not to the Service Fabric application name.
- **/exe** defines the executable that Service Fabric is supposed to launch, in this case `node.exe`.
- **/ma** defines the argument that is being used to launch the executable. As Node.js is not installed, Service Fabric needs to launch the Node.js web server by executing `node.exe bin/www`. `/ma:'bin/www'` tells the packaging tool to use `bin/ma` as the argument for node.exe.
- **/AppType** defines the Service Fabric application type name.

If you browse to the directory that was specified in the /target parameter, you can see that the tool has created a fully functioning Service Fabric package as shown below:

```
|--[yourtargetdirectory]
    |-- NodeApplication
        |-- C
              |-- bin
              |-- data
              |-- node_modules
              |-- public
              |-- routes
              |-- views
              |-- app.js
              |-- package.json
              |-- node.exe
        |-- config
              |--Settings.xml
        |-- ServiceManifest.xml
    |-- ApplicationManifest.xml
```

The generated ServiceManifest.xml now has a section that describes how the Node.js web server should be launched, as shown in the code snippet below:

```
<CodePackage Name="C" Version="1.0">
    <EntryPoint>
        <ExeHost>
            <Program>node.exe</Program>
            <Arguments>'bin/www'</Arguments>
            <WorkingFolder>CodePackage</WorkingFolder>
        </ExeHost>
    </EntryPoint>
</CodePackage>
```

In this sample, the Node.js web server listens to port 3000, so you need to update the endpoint information in the ServiceManifest.xml file as shown below.

```
<Resources>
      <Endpoints>
         <Endpoint Name="NodeServiceEndpoint" Protocol="http" Port="3000" Type="Input" />
      </Endpoints>
</Resources>
```

Packaging the MongoDB application

Now that you have packaged the Node.js application, you can go ahead and package MongoDB. As mentioned before, the steps that you go through now are not specific to Node.js and MongoDB. In fact, they apply to all applications that are meant to be packaged together as one Service Fabric application.

To package MongoDB, you want to make sure you package Mongod.exe and Mongo.exe. Both binaries are located in the `bin` directory of your MongoDB installation directory. The directory structure looks similar to the one below.

```
|-- MongoDB
    |-- bin
        |-- mongod.exe
        |-- mongo.exe
        |-- anybinary.exe
```

Service Fabric needs to start MongoDB with a command similar to the one below, so you need to use the `/ma` parameter when packaging MongoDB.

```
mongod.exe --dbpath [path to data]
```

> **NOTE**
> The data is not being preserved in the case of a node failure if you put the MongoDB data directory on the local directory of the node. You should either use durable storage or implement a MongoDB replica set in order to prevent data loss.

In PowerShell or the command shell, we run the packaging tool with the following parameters:

```
.\ServiceFabricAppPackageUtil.exe /source: [yourdirectory]\MongoDB' /target:'[yourtargetdirectory]'
/appname:MongoDB /exe:'bin\mongod.exe' /ma:'--dbpath [path to data]' /AppType:NodeAppType
```

In order to add MongoDB to your Service Fabric application package, you need to make sure that the /target parameter points to the same directory that already contains the application manifest along with the Node.js application. You also need to make sure that you are using the same ApplicationType name.

Let's browse to the directory and examine what the tool has created.

```
|--[yourtargetdirectory]
    |-- MyNodeApplication
    |-- MongoDB
        |-- C
            |--bin
                |-- mongod.exe
                |-- mongo.exe
                |-- etc.
        |-- config
            |--Settings.xml
        |-- ServiceManifest.xml
    |-- ApplicationManifest.xml
```

As you can see, the tool added a new folder, MongoDB, to the directory that contains the MongoDB binaries. If you open the `ApplicationManifest.xml` file, you can see that the package now contains both the Node.js application and MongoDB. The code below shows the content of the application manifest.

```
<ApplicationManifest xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" ApplicationTypeName="MyNodeApp" ApplicationTypeVersion="1.0"
xmlns="http://schemas.microsoft.com/2011/01/fabric">
   <ServiceManifestImport>
      <ServiceManifestRef ServiceManifestName="MongoDB" ServiceManifestVersion="1.0" />
   </ServiceManifestImport>
   <ServiceManifestImport>
      <ServiceManifestRef ServiceManifestName="NodeService" ServiceManifestVersion="1.0" />
   </ServiceManifestImport>
   <DefaultServices>
      <Service Name="MongoDBService">
         <StatelessService ServiceTypeName="MongoDB">
            <SingletonPartition />
         </StatelessService>
      </Service>
      <Service Name="NodeServiceService">
         <StatelessService ServiceTypeName="NodeService">
            <SingletonPartition />
         </StatelessService>
      </Service>
   </DefaultServices>
</ApplicationManifest>
```

Publishing the application

The last step is to publish the application to the local Service Fabric cluster by using the PowerShell scripts below:

```
Connect-ServiceFabricCluster localhost:19000

Write-Host 'Copying application package...'
Copy-ServiceFabricApplicationPackage -ApplicationPackagePath '[yourtargetdirectory]' -ImageStoreConnectionString
'file:C:\SfDevCluster\Data\ImageStoreShare' -ApplicationPackagePathInImageStore 'NodeAppType'

Write-Host 'Registering application type...'
Register-ServiceFabricApplicationType -ApplicationPathInImageStore 'NodeAppType'

New-ServiceFabricApplication -ApplicationName 'fabric:/NodeApp' -ApplicationTypeName 'NodeAppType' -
ApplicationTypeVersion 1.0
```

Once the application is successfully published to the local cluster, you can access the Node.js application on the port that we have entered in the service manifest of the Node.js application--for example http://localhost:3000.

In this tutorial, you have seen how to easily package two existing applications as one Service Fabric application. You have also learned how to deploy it to Service Fabric so that it can benefit from some of the Service Fabric features, such as high availability and health system integration.

Next steps

- Learn about deploying containers with Service Fabric and containers overview

Preview: Service Fabric and containers

11/15/2016 • 5 min to read • Edit on GitHub

Contributors

Mark Fussell • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • ShawnJackson • Joseph Molnar

> **NOTE**
> This feature is in preview for Linux and is not currently available on Windows Server 2016. This will be in preview for Windows Server on the next release of Azure Service Fabric after Windows Server 2016 GA and supported in the subsequent release after that.

Introduction

Azure Service Fabric is an orchestrator of services across a cluster of machines. Services can be developed in many ways, from using the Service Fabric programming models to deploying guest executables. By default, Service Fabric deploys and activates these services as processes. Processes provide the fastest activation and highest density usage of the resources in a cluster. Service Fabric can also deploy services in container images. Importantly, you can mix services in processes and services in containers in the same application. You get the best of both worlds, depending on your scenario.

What are containers?

Containers are encapsulated, individually deployable components that run as isolated instances on the same kernel to take advantage of virtualization that an operating system provides. This means that each application and its runtime, dependencies, and system libraries run inside a container with full, private access to the container's own isolated view of operating system constructs. Along with portability, this degree of security and resource isolation is the main benefit for using containers with Service Fabric, which otherwise runs services in processes.

Containers are a virtualization technology that virtualizes the underlying operating system from applications. Containers provide an immutable environment for applications to run with varying degrees of isolation. Containers run directly on top of the kernel and have an isolated view of the file system and other resources. Compared to virtual machines, containers have the following advantages:

- **Small in size**: Containers use a single storage space and smaller deltas for each layer to increase efficiency.
- **Fast boot-up time**: Containers don't have to boot an operating system, so they can be up and available much faster, typically in seconds, than virtual machines.
- **Portability**: A containerized application image can be ported to run in the cloud, on premises, inside virtual machines, or directly on physical machines.
- **Resource governance**: A container can limit the physical resources that it can consume on its host.

Container types

Service Fabric supports the following types of containers.

Docker containers on Linux

Docker provides high-level APIs to create and manage containers on top of Linux kernel containers. Docker Hub is a central repository to store and retrieve container images.

For a walkthrough about how to do this, read Deploy a Docker container to Service Fabric.

Windows Server containers

Windows Server 2016 provides two different types of containers that differ in the level of provided isolation. Windows Server containers and Docker containers are similar because both have namespace and file system isolation but share the kernel with the host they are running on. On Linux, this isolation has traditionally been provided by cgroups and namespaces, and Windows Server containers behave similarly.

Windows Hyper-V containers provide more isolation and security because each container does not share the operating system kernel with other containers or with the host. With this higher level of security isolation, Hyper-V containers are particularly targeted at hostile, multitenant scenarios.

For a walkthrough about how to do this, read Deploy a Windows container to Service Fabric.

The following figure shows the different types of virtualization and isolation levels available in the operating system.

Scenarios for using containers

Here are typical examples where a container is a good choice:

- **IIS lift and shift**: If you have existing ASP.NET MVC apps that you want to continue to use, put them in a container instead of migrating them to ASP.NET Core. These ASP.NET MVC apps are dependent on Internet Information Services (IIS). You can package these into container images from the precreated IIS image and deploy them with Service Fabric. See Container Images on Windows Server for information about how to create IIS images.
- **Mix containers and Service Fabric microservices**: Use an existing container image for part of your application. For example, you might use the NGINX container for the web front end of your application and stateful services built with Reliable Services for the more intensive back-end computation. An example of this scenario includes gaming applications.
- **Reduce impact of "noisy neighbors" services**: You can use the resource governance ability of containers to restrict the resources that a service uses on a host. If services might consume a lot of resources and affect the performance of others (such as a long-running, query-like operation), consider putting these services into

containers that have resource governance.

Service Fabric support for containers

Service Fabric currently supports deployment of Docker containers on Linux and Windows Server containers on Windows Server 2016. Support for Hyper-V containers will be added in a future release.

In the Service Fabric application model, a container represents an application host in which multiple service replicas are placed. The following scenarios are supported for containers:

- **Guest containers**: This scenario is identical to the guest executable scenario where you can deploy existing applications in a container. Examples include Node.js, JavaScript, or any code (executables).
- **Stateless services inside containers**: These are stateless services that use the Reliable Services and the Reliable Actors programming models. This is currently only supported on Linux. Support for stateless services in Windows containers will be added in future release.
- **Stateful services inside containers**: These are stateful services that use the Reliable Actors programming model on Linux. Reliable Services are not currently supported on Linux. Support for stateful services in Windows containers will be added in future release.

Service Fabric has several container capabilities that help you build applications that are composed of microservices that are containerized. These are called containerized services. The capabilities include:

- Container image deployment and activation.
- Resource governance.
- Repository authentication.
- Container port to host port mapping.
- Container-to-container discovery and communication.
- Ability to configure and set environment variables.

Next steps

In this article, you learned about containers, that Service Fabric is a container orchestrator, and that Service Fabric has features that support containers. As a next step, we will go over examples of each of the features to show you how to use them.

Deploy a Windows container to Service Fabric on Windows Server 2016

Deploy a Docker container to Service Fabric on Linux

Preview: Deploy a Windows container to Service Fabric

11/22/2016 • 6 min to read • Edit on GitHub

Contributors

Mark Fussell • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Junshen Tao (Beyondsoft Corporation) • ShawnJackson • alexandair • Rebecca Agiewich • Dhanashri Kshirsagar

This article walks you through the process of building containerized services in Windows containers.

> **NOTE**
> This feature is in preview for Linux and is not available for Windows Server 2016 (yet). The feature will be in preview for Windows Server 2016 in the upcoming release of Azure Service Fabric.

Service Fabric has several container capabilities that help you with building applications that are composed of microservices that are containerized.

The capabilities include:

- Container image deployment and activation
- Resource governance
- Repository authentication
- Container port-to-host port mapping
- Container-to-container discovery and communication
- Ability to configure and set environment variables

Let's look at how each of capabilities works when you're packaging a containerized service to be included in your application.

Package a Windows container

When you package a container, you can choose to use either a Visual Studio project template or create the application package manually. When you use Visual Studio, the application package structure and manifest files are created by the New Project Template for you. The VS template will be released in a future release.

Use Visual Studio to package an existing container image

> **NOTE**
> In an upcoming release of Visual Studio for Service Fabric, you can add a container to an application in the same way that you can add a guest executable today. For more information, see Deploy a guest executable to Service Fabric topic. Currently you have to manually package a container as described in the next section.

Manually package and deploy a container

The process of manually packaging a containerized service is based on the following steps:

1. Publish the containers to your repository.
2. Create the package directory structure.
3. Edit the service manifest file.
4. Edit the application manifest file.

Deploy and activate a container image

In the Service Fabric application model, a container represents an application host in which multiple service replicas are placed. To deploy and activate a container, put the name of the container image into a `ContainerHost` element in the service manifest.

In the service manifest, add a `ContainerHost` for the entry point. Then set the `ImageName` to be the name of the container repository and image. The following partial manifest shows an example of how to deploy the container called `myimage:v1` from a repository called `myrepo`:

```
<CodePackage Name="Code" Version="1.0">
    <EntryPoint>
      <ContainerHost>
        <ImageName>myrepo/myimage:v1</ImageName>
        <Commands></Commands>
      </ContainerHost>
    </EntryPoint>
</CodePackage>
```

You can provide input commands by specifying the optional `Commands` element with a comma-delimited set of commands to run inside the container.

Understand resource governance

Resource governance is a capability of the container that restricts the resources that the container can use on the host. The `ResourceGovernancePolicy`, which is specified in the application manifest is used to declare resource limits for a service code package. Resource limits can be set for the following resources:

- Memory
- MemorySwap
- CpuShares (CPU relative weight)
- MemoryReservationInMB
- BlkioWeight (BlockIO relative weight).

> **NOTE**
> In a future release, support for specifying specific block IO limits such as IOPs, read/write BPS, and others will be included.

```
<ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="FrontendServicePackage" ServiceManifestVersion="1.0"/>
    <Policies>
        <ResourceGovernancePolicy CodePackageRef="FrontendService.Code" CpuShares="500"
        MemoryInMB="1024" MemorySwapInMB="4084" MemoryReservationInMB="1024" />
    </Policies>
</ServiceManifestImport>
```

Authenticate a repository

To download a container, you might have to provide sign-in credentials to the container repository. The sign-in credentials, specified in the application manifest, are used to specify the sign-in information, or SSH key, for

downloading the container image from the image repository. The following example shows an account called *TestUser* along with the password in clear text (*not* recommended):

```
<ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="FrontendServicePackage" ServiceManifestVersion="1.0"/>
    <Policies>
        <ContainerHostPolicies CodePackageRef="FrontendService.Code">
            <RepositoryCredentials AccountName="TestUser" Password="12345" PasswordEncrypted="false"/>
        </ContainerHostPolicies>
    </Policies>
</ServiceManifestImport>
```

We recommend that you encrypt the password by using a certificate that's deployed to the machine.

The following example shows an account called *TestUser*, where the password was encrypted by using a certificate called *MyCert*. You can use the `Invoke-ServiceFabricEncryptText` PowerShell command to create the secret cipher text for the password. For more information, see the article Managing secrets in Service Fabric applications.

The private key of the certificate that's used to decrypt the password must be deployed to the local machine in an out-of-band method. (In Azure, this method is Azure Resource Manager.) Then, when Service Fabric deploys the service package to the machine, it can decrypt the secret. By using the secret along with the account name, it can then authenticate with the container repository.

```
<ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="FrontendServicePackage" ServiceManifestVersion="1.0"/>
    <Policies>
        <ContainerHostPolicies CodePackageRef="FrontendService.Code">
            <RepositoryCredentials AccountName="TestUser" Password="[Put encrypted password here using MyCert
certificate ]" PasswordEncrypted="true"/>
        </ContainerHostPolicies>
    </Policies>
</ServiceManifestImport>
```

Configure container port-to-host port mapping

You can configure a host port used to communicate with the container by specifying a `PortBinding` in the application manifest. The port binding maps the port to which the service is listening inside the container to a port on the host.

```
<ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="FrontendServicePackage" ServiceManifestVersion="1.0"/>
    <Policies>
        <ContainerHostPolicies CodePackageRef="FrontendService.Code">
            <PortBinding ContainerPort="8905"/>
        </ContainerHostPolicies>
    </Policies>
</ServiceManifestImport>
```

Configure container-to-container discovery and communication

By using the `PortBinding` policy, you can map a container port to an `Endpoint` in the service manifest as shown in the following example. The endpoint `Endpoint1` can specify a fixed port (for example, port 80). It can also specify no port at all, in which case a random port from the cluster's application port range is chosen for you.

If you specify an endpoint, using the `Endpoint` tag in the service manifest of a guest container, Service Fabric can automatically publish this endpoint to the Naming service. Other services that are running in the cluster can thus discover this container using the REST queries for resolving.

```
<ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="FrontendServicePackage" ServiceManifestVersion="1.0"/>
    <Policies>
        <ContainerHostPolicies CodePackageRef="FrontendService.Code">
            <PortBinding ContainerPort="8905" EndpointRef="Endpoint1"/>
        </ContainerHostPolicies>
    </Policies>
</ServiceManifestImport>
```

By registering with the Naming service, you can easily do container-to-container communication in the code within your container by using the reverse proxy. Communication is performed by providing the reverse proxy http listening port and the name of the services that you want to communicate with as environment variables. For more information, see the next section.

Configure and set environment variables

Environment variables can be specified for each code package in the service manifest, both for services that are deployed in containers or for services that are deployed as processes/guest executables. These environment variable values can be overridden specifically in the application manifest or specified during deployment as application parameters.

The following service manifest XML snippet shows an example of how to specify environment variables for a code package:

```
<ServiceManifest Name="FrontendServicePackage" Version="1.0" xmlns="http://schemas.microsoft.com/2011/01/fabric"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Description>a guest executable service in a container</Description>
    <ServiceTypes>
        <StatelessServiceType ServiceTypeName="StatelessFrontendService"  UseImplicitHost="true"/>
    </ServiceTypes>
    <CodePackage Name="FrontendService.Code" Version="1.0">
        <EntryPoint>
        <ContainerHost>
            <ImageName>myrepo/myimage:v1</ImageName>
            <Commands></Commands>
        </ContainerHost>
        </EntryPoint>
        <EnvironmentVariables>
            <EnvironmentVariable Name="HttpGatewayPort" Value=""/>
            <EnvironmentVariable Name="BackendServiceName" Value=""/>
        </EnvironmentVariables>
    </CodePackage>
</ServiceManifest>
```

These environment variables can be overridden at the application manifest level:

```
<ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="FrontendServicePackage" ServiceManifestVersion="1.0"/>
    <EnvironmentOverrides CodePackageRef="FrontendService.Code">
        <EnvironmentVariable Name="BackendServiceName" Value="[BackendSvc]"/>
        <EnvironmentVariable Name="HttpGatewayPort" Value="19080"/>
    </EnvironmentOverrides>
</ServiceManifestImport>
```

In the previous example, we specified an explicit value for the `HttpGateway` environment variable (19000), while we set the value for `BackendServiceName` parameter via the `[BackendSvc]` application parameter. These settings enable you to specify the value for `BackendServiceName` value when you deploy the application and not have a fixed value in the manifest.

Complete examples for application and service manifest

An example application manifest follows:

```
<ApplicationManifest ApplicationTypeName="SimpleContainerApp" ApplicationTypeVersion="1.0"
xmlns="http://schemas.microsoft.com/2011/01/fabric" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Description>A simple service container application</Description>
    <Parameters>
        <Parameter Name="ServiceInstanceCount" DefaultValue="3"></Parameter>
        <Parameter Name="BackEndSvcName" DefaultValue="bkend"></Parameter>
    </Parameters>
    <ServiceManifestImport>
        <ServiceManifestRef ServiceManifestName="FrontendServicePackage" ServiceManifestVersion="1.0"/>
        <EnvironmentOverrides CodePackageRef="FrontendService.Code">
            <EnvironmentVariable Name="BackendServiceName" Value="[BackendSvcName]"/>
            <EnvironmentVariable Name="HttpGatewayPort" Value="19080"/>
        </EnvironmentOverrides>
        <Policies>
            <ResourceGovernancePolicy CodePackageRef="Code" CpuShares="500" MemoryInMB="1024"
MemorySwapInMB="4084" MemoryReservationInMB="1024" />
            <ContainerHostPolicies CodePackageRef="FrontendService.Code">
                <RepositoryCredentials AccountName="username" Password="****" PasswordEncrypted="true"/>
                <PortBinding ContainerPort="8905" EndpointRef="Endpoint1"/>
            </ContainerHostPolicies>
        </Policies>
    </ServiceManifestImport>
</ApplicationManifest>
```

An example service manifest (specified in the preceding application manifest) follows:

```
<ServiceManifest Name="FrontendServicePackage" Version="1.0" xmlns="http://schemas.microsoft.com/2011/01/fabric"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Description> A service that implements a stateless front end in a container</Description>
    <ServiceTypes>
        <StatelessServiceType ServiceTypeName="StatelessFrontendService"  UseImplicitHost="true"/>
    </ServiceTypes>
    <CodePackage Name="FrontendService.Code" Version="1.0">
        <EntryPoint>
        <ContainerHost>
            <ImageName>myrepo/myimage:v1</ImageName>
            <Commands></Commands>
        </ContainerHost>
        </EntryPoint>
        <EnvironmentVariables>
            <EnvironmentVariable Name="HttpGatewayPort" Value=""/>
            <EnvironmentVariable Name="BackendServiceName" Value=""/>
        </EnvironmentVariables>
    </CodePackage>
    <ConfigPackage Name="FrontendService.Config" Version="1.0" />
    <DataPackage Name="FrontendService.Data" Version="1.0" />
    <Resources>
        <Endpoints>
            <Endpoint Name="Endpoint1" Port="80"  UriScheme="http" />
        </Endpoints>
    </Resources>
</ServiceManifest>
```

Next steps

Now that you have deployed a containerized service, learn how to manage its lifecycle by reading Service Fabric application lifecycle.

Deploy a Docker container to Service Fabric

11/15/2016 • 1 min to read • Edit on GitHub

Contributors

Mark Fussell • Theano Petersen • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

---

This article walks you through building containerized services in Docker containers on Linux.

Service Fabric has several container capabilities that help you with building applications that are composed of microservices that are containerized. These are called containerized services.

The capabilities include;

- Container image deployment and activation
- Resource governance
- Repository authentication
- Container port to host port mapping
- Container-to-container discovery and communication
- Ability to configure and set environment variables

Packaging a docker container with yeoman

When packaging a container on Linux, you can choose either to use a yeoman template or create the application package manually.

A Service Fabric application can contain one or more containers, each with a specific role in delivering the application's functionality. The Service Fabric SDK for Linux includes a Yeoman generator that makes it easy to create your application and add a container image. Let's use Yeoman to create a new application with a single Docker container called *SimpleContainerApp*. You can add more services later by editing the generated manifest files.

Create the application

1. In a terminal, type **yo azuresfguest**.
2. For the framework choose **Container** .
3. Name your application e.g. SimpleContainerApp
4. Provide the URL for the container image from a DockerHub repo. This takes the form [repo]/[image name]

```
Make sure you are in the directory you want to scaffold into.
This generator can also be run with: yo azuresfguest

     _-----_     ╭──────────────────────╮
    |       |    │    Welcome to Service│
    |--(o)--|    │Fabric guest application│
   `---------´   │        generator     │
    ( _´U`_ )    ╰──────────────────────╯
    /___A___\   /
     |  ~  |
   __'.___.'__
 ´   `  |° ´ Y `

? Choose a framework for your service Container
? Name your application SimpleContainerApp
? Name of the application service: FrontEnd
? Input the Image Name: mfussell/sffrontend:v1
? Commands:
? Number of instances of guest container application: 1
   create SimpleContainerApp/SimpleContainerApp/ApplicationManifest.xml
   create SimpleContainerApp/SimpleContainerApp/FrontEndPkg/ServiceManifest.xml
   create SimpleContainerApp/SimpleContainerApp/FrontEndPkg/config/Settings.xml
   create SimpleContainerApp/install.sh
   create SimpleContainerApp/uninstall.sh
   create SimpleContainerApp/SimpleContainerApp/FrontEndPkg/code/Dummy.txt

     _-----_     ╭──────────────────────╮
    |       |    │      Bye from us!    │
    |--(o)--|    │       Chat soon.     │
   `---------´   │      Yeoman team     │
    ( _´U`_ )    │   http://yeoman.io   │
    /___A___\   /╰──────────────────────╯
```

Deploy the application

Once the application is built, you can deploy it to the local cluster using the Azure CLI.

1. Connect to the local Service Fabric cluster.

   ```
   azure servicefabric cluster connect
   ```

2. Use the install script provided in the template to copy the application package to the cluster's image store, register the application type, and create an instance of the application.

   ```
   ./install.sh
   ```

3. Open a browser and navigate to Service Fabric Explorer at http://localhost:19080/Explorer (replace localhost with the private IP of the VM if using Vagrant on Mac OS X).
4. Expand the Applications node and note that there is now an entry for your application type and another for the first instance of that type.
5. Use the uninstall script provided in the template to delete the application instance and unregister the application type.

   ```
   ./uninstall.sh
   ```

Next steps

- Overview of Service Fabric and containers
- Interacting with Service Fabric clusters using the Azure CLI

Learn about the differences between Cloud Services and Service Fabric before migrating applications.

11/15/2016 • 5 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • alexandair

Microsoft Azure Service Fabric is the next-generation cloud application platform for highly scalable, highly reliable distributed applications. It introduces many new features for packaging, deploying, upgrading, and managing distributed cloud applications.

This is an introductory guide to migrating applications from Cloud Services to Service Fabric. It focuses primarily on architectural and design differences between Cloud Services and Service Fabric.

Applications and infrastructure

A fundamental difference between Cloud Services and Service Fabric is the relationship between VMs, workloads, and applications. A workload here is defined as the code you write to perform a specific task or provide a service.

- **Cloud Services is about deploying applications as VMs.** The code you write is tightly coupled to a VM instance, such as a Web or Worker Role. To deploy a workload in Cloud Services is to deploy one or more VM instances that run the workload. There is no separation of applications and VMs, and so there is no formal definition of an application. An application can be thought of as a set of Web or Worker Role instances within a Cloud Services deployment or as an entire Cloud Services deployment. In this example, an application is shown as a set of role instances.

- **Service Fabric is about deploying applications to existing VMs or machines running Service Fabric on Windows or Linux.** The services you write are completely decoupled from the underlying infrastructure, which is abstracted away by the Service Fabric application platform, so an application can be deployed to multiple environments. A workload in Service Fabric is called a "service," and one or more services are grouped in a formally-defined application that runs on the Service Fabric application platform. Multiple applications can be deployed to a single Service Fabric cluster.

Service Fabric itself is an application platform layer that runs on Windows or Linux, whereas Cloud Services is a system for deploying Azure-managed VMs with workloads attached. The Service Fabric application model has a number of advantages:

- Fast deployment times. Creating VM instances can be time consuming. In Service Fabric, VMs are only deployed once to form a cluster that hosts the Service Fabric application platform. From that point on, application packages can be deployed to the cluster very quickly.
- High-density hosting. In Cloud Services, a Worker Role VM hosts one workload. In Service Fabric, applications are separate from the VMs that run them, meaning you can deploy a large number of applications to a small number of VMs, which can lower overall cost for larger deployments.
- The Service Fabric platform can run anywhere that has Windows Server or Linux machines, whether it's Azure or on-premises. The platform provides an abstraction layer over the underlying infrastructure so your application can run on different environments.
- Distributed application management. Service Fabric is a platform that not only hosts distributed applications, but also helps manage their lifecycle independently of the hosting VM or machine lifecycle.

Application architecture

The architecture of a Cloud Services application usually includes numerous external service dependencies, such as Service Bus, Azure Table and Blob Storage, SQL, Redis, and others to manage the state and data of an application and communication between Web and Worker Roles in a Cloud Services deployment. An example of a complete Cloud Services application might look like this:

Service Fabric applications can also choose to use the same external services in a complete application. Using this example Cloud Services architecture, the simplest migration path from Cloud Services to Service Fabric is to replace only the Cloud Services deployment with a Service Fabric application, keeping the overall architecture the same. The Web and Worker Roles can be ported to Service Fabric stateless services with minimal code changes.

At this stage, the system should continue to work the same as before. Taking advantage of Service Fabric's stateful features, external state stores can be internalized as stateful services where applicable. This is more involved than a simple migration of Web and Worker Roles to Service Fabric stateless services, as it requires writing custom services that provide equivalent functionality to your application as the external services did before. The benefits of doing so include:

- Removing external dependencies
- Unifying the deployment, management, and upgrade models.

An example resulting architecture of internalizing these services could look like this:

Communication and workflow

Most Cloud Service applications consist of more than one tier. Similarly, a Service Fabric application consists of more than one service (typically many services). Two common communication models are direct communication and via an external durable storage.

Direct communication

With direct communication, tiers can communicate directly through endpoint exposed by each tier. In stateless environments such as Cloud Services, this means selecting an instance of a VM role, either randomly or round-robin to balance load, and connecting to its endpoint directly.

Direct communication is a common communication model in Service Fabric. The key difference between Service Fabric and Cloud Services is that in Cloud Services you connect to a VM, whereas in Service Fabric you connect to a service. This is an important distinction for a couple reasons:

- Services in Service Fabric are not bound to the VMs that host them; services may move around in the cluster, and in fact, are expected to move around for various reasons: Resource balancing, failover, application and infrastructure upgrades, and placement or load constraints. This means a service instance's address can change at any time.
- A VM in Service Fabric can host multiple services, each with unique endpoints.

Service Fabric provides a service discovery mechanism, called the Naming Service, which can be used to resolve endpoint addresses of services.

Queues

A common communication mechanism between tiers in stateless environments such as Cloud Services is to use an external storage queue to durably store work tasks from one tier to another. A common scenario is a web tier that sends jobs to an Azure Queue or Service Bus where Worker Role instances can dequeue and process the jobs.

The same communication model can be used in Service Fabric. This can be useful when migrating an existing Cloud Services application to Service Fabric.

Next Steps

The simplest migration path from Cloud Services to Service Fabric is to replace only the Cloud Services deployment with a Service Fabric application, keeping the overall architecture of your application roughly the same. The following article provides a guide to help convert a Web or Worker Role to a Service Fabric stateless service.

- Simple migration: convert a Web or Worker Role to a Service Fabric stateless service

Guide to converting Web and Worker Roles to Service Fabric stateless services

11/15/2016 • 8 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • alexandair

This article describes how to migrate your Cloud Services Web and Worker Roles to Service Fabric stateless services. This is the simplest migration path from Cloud Services to Service Fabric for applications whose overall architecture is going to stay roughly the same.

Cloud Service project to Service Fabric application project

A Cloud Service project and a Service Fabric Application project have a similar structure and both represent the deployment unit for your application - that is, they each define the complete package that is deployed to run your application. A Cloud Service project contains one or more Web or Worker Roles. Similarly, a Service Fabric Application project contains one or more services.

The difference is that the Cloud Service project couples the application deployment with a VM deployment and thus contains VM configuration settings in it, whereas the Service Fabric Application project only defines an application that will be deployed to a set of existing VMs in a Service Fabric cluster. The Service Fabric cluster itself is only deployed once, either through an ARM template or through the Azure Portal, and multiple Service Fabric applications can be deployed to it.

Cloud Services

Service Fabric

Worker Role to stateless service

Conceptually, a Worker Role represents a stateless workload, meaning every instance of the workload is identical and requests can be routed to any instance at any time. Each instance is not expected to remember the previous request. State that the workload operates on is managed by an external state store, such as Azure Table Storage or Azure Document DB. In Service Fabric, this type of workload is represented by a Stateless Service. The simplest

approach to migrating a Worker Role to Service Fabric can be done by converting Worker Role code to a Stateless Service.

Web Role to stateless service

Similar to Worker Role, a Web Role also represents a stateless workload, and so conceptually it too can be mapped to a Service Fabric stateless service. However, unlike Web Roles, Service Fabric does not support IIS. To migrate a web application from a Web Role to a stateless service requires first moving to a web framework that can be self-hosted and does not depend on IIS or System.Web, such as ASP.NET Core 1.

**APPLICATION **	SUPPORTED	MIGRATION PATH
ASP.NET Web Forms	No	Convert to ASP.NET Core 1 MVC
ASP.NET MVC	With Migration	Upgrade to ASP.NET Core 1 MVC
ASP.NET Web API	With Migration	Use self-hosted server or ASP.NET Core 1
ASP.NET Core 1	Yes	N/A

Entry point API and lifecycle

Worker Role and Service Fabric service APIs offer similar entry points:

ENTRY POINT	WORKER ROLE	SERVICE FABRIC SERVICE
Processing	`Run()`	`RunAsync()`
VM start	`OnStart()`	N/A
VM stop	`OnStop()`	N/A
Open listener for client requests	N/A	• `CreateServiceInstanceListener()` for stateless • `CreateServiceReplicaListener()` for stateful

Worker Role

```
using Microsoft.WindowsAzure.ServiceRuntime;

namespace WorkerRole1
{
    public class WorkerRole : RoleEntryPoint
    {
        public override void Run()
        {
        }

        public override bool OnStart()
        {
        }

        public override void OnStop()
        {
        }
    }
}
```

Service Fabric Stateless Service

```
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.ServiceFabric.Services.Communication.Runtime;
using Microsoft.ServiceFabric.Services.Runtime;

namespace Stateless1
{
    public class Stateless1 : StatelessService
    {
        protected override IEnumerable<ServiceInstanceListener> CreateServiceInstanceListeners()
        {
        }

        protected override Task RunAsync(CancellationToken cancelServiceInstance)
        {
        }
    }
}
```

Both have a primary "Run" override in which to begin processing. Service Fabric services combine `Run`, `Start`, and `Stop` into a single entry point, `RunAsync`. Your service should begin working when `RunAsync` starts, and should stop working when the `RunAsync` method's CancellationToken is signaled.

There are several key differences between the lifecycle and lifetime of Worker Roles and Service Fabric services:

- **Lifecycle:** The biggest difference is that a Worker Role is a VM and so its lifecycle is tied to the VM, which includes events for when the VM starts and stops. A Service Fabric service has a lifecycle that is separate from the VM lifecycle, so it does not include events for when the host VM or machine starts and stop, as they are not related.
- **Lifetime:** A Worker Role instance will recycle if the `Run` method exits. The `RunAsync` method in a Service Fabric service however can run to completion and the service instance will stay up.

Service Fabric provides an optional communication setup entry point for services that listen for client requests. Both the RunAsync and communication entry point are optional overrides in Service Fabric services - your service may choose to only listen to client requests, or only run a processing loop, or both - which is why the RunAsync method is allowed to exit without restarting the service instance, because it may continue to listen for client requests.

Application API and environment

The Cloud Services environment API provides information and functionality for the current VM instance as well as information about other VM role instances. Service Fabric provides information related to its runtime and some information about the node a service is currently running on.

ENVIRONMENT TASK	CLOUD SERVICES	SERVICE FABRIC
Configuration Settings and change notification	`RoleEnvironment`	`CodePackageActivationContext`
Local Storage	`RoleEnvironment`	`CodePackageActivationContext`
Endpoint Information	`RoleInstance` • Current instance: `RoleEnvironment.CurrentRoleInstance` • Other roles and instance: `RoleEnvironment.Roles`	• `NodeContext` for current Node address • `FabricClient` and `ServicePartitionResolver` for service endpoint discovery
Environment Emulation	`RoleEnvironment.IsEmulated`	N/A
Simultaneous change event	`RoleEnvironment`	N/A

Configuration settings

Configuration settings in Cloud Services are set for a VM role and apply to all instances of that VM role. These settings are key-value pairs set in ServiceConfiguration.*.cscfg files and can be accessed directly through RoleEnvironment. In Service Fabric, settings apply individually to each service and to each application, rather than to a VM, because a VM can host multiple services and applications. A service is composed of three packages:

- **Code:** contains the service executables, binaries, DLLs, and any other files a service needs to run.
- **Config:** all configuration files and settings for a service.
- **Data:** static data files associated with the service.

Each of these packages can be independently versioned and upgraded. Similar to Cloud Services, a config package can be accessed programmatically through an API and events are available to notify the service of a config package change. A Settings.xml file can be used for key-value configuration and programmatic access similar to the app settings section of an App.config file. However, unlike Cloud Services, a Service Fabric config package can contain any configuration files in any format, whether it's XML, JSON, YAML, or a custom binary format.

Accessing configuration

Cloud Services

Configuration settings from ServiceConfiguration.*.cscfg can be accessed through `RoleEnvironment`. These settings are globally available to all role instances in the same Cloud Service deployment.

```
string value = RoleEnvironment.GetConfigurationSettingValue("Key");
```

Service Fabric

Each service has its own individual configuration package. There is no built-in mechanism for global configuration settings accessible by all applications in a cluster. When using Service Fabric's special Settings.xml configuration file within a configuration package, values in Settings.xml can be overwritten at the application level, making application-level configuration settings possible.

Configuration settings are accesses within each service instance through the service's `CodePackageActivationContext`.

```
ConfigurationPackage configPackage =
this.Context.CodePackageActivationContext.GetConfigurationPackageObject("Config");

// Access Settings.xml
KeyedCollection<string, ConfigurationProperty> parameters =
configPackage.Settings.Sections["MyConfigSection"].Parameters;

string value = parameters["Key"]?.Value;

// Access custom configuration file:
using (StreamReader reader = new StreamReader(Path.Combine(configPackage.Path, "CustomConfig.json")))
{
    MySettings settings = JsonConvert.DeserializeObject<MySettings>(reader.ReadToEnd());
}
```

Configuration update events

Cloud Services

The `RoleEnvironment.Changed` event is used to notify all role instances when a change occurs in the environment, such as a configuration change. This is used to consume configuration updates without recycling role instances or restarting a worker process.

```
RoleEnvironment.Changed += RoleEnvironmentChanged;

private void RoleEnvironmentChanged(object sender, RoleEnvironmentChangedEventArgs e)
{
   // Get the list of configuration changes
   var settingChanges = e.Changes.OfType<RoleEnvironmentConfigurationSettingChange>();
foreach (var settingChange in settingChanges)
   {
      Trace.WriteLine("Setting: " + settingChange.ConfigurationSettingName, "Information");
   }
}
```

Service Fabric

Each of the three package types in a service - Code, Config, and Data - have events that notify a service instance when a package is updated, added, or removed. A service can contain multiple packages of each type. For example, a service may have multiple config packages, each individually versioned and upgradeable.

These events are available to consume changes in service packages without restarting the service instance.

```
this.Context.CodePackageActivationContext.ConfigurationPackageModifiedEvent +=
                    this.CodePackageActivationContext_ConfigurationPackageModifiedEvent;

private void CodePackageActivationContext_ConfigurationPackageModifiedEvent(object sender,
PackageModifiedEventArgs<ConfigurationPackage> e)
{
    this.UpdateCustomConfig(e.NewPackage.Path);
    this.UpdateSettings(e.NewPackage.Settings);
}
```

Startup tasks

Startup tasks are actions that are taken before an application starts. A startup task is typically used to run setup scripts using elevated privileges. Both Cloud Services and Service Fabric support start-up tasks. The main difference

is that in Cloud Services, a startup task is tied to a VM because it is part of a role instance, whereas in Service Fabric a startup task is tied to a service, which is not tied to any particular VM.

CLOUD SERVICES	SERVICE FABRIC
Configuration location	ServiceDefinition.csdef
Privileges	"limited" or "elevated"
Sequencing	"simple", "background", "foreground"

Cloud Services

In Cloud Services a startup entry point is configured per role in ServiceDefinition.csdef.

```
<ServiceDefinition>
    <Startup>
        <Task commandLine="Startup.cmd" executionContext="limited" taskType="simple" >
            <Environment>
                <Variable name="MyVersionNumber" value="1.0.0.0" />
            </Environment>
        </Task>
    </Startup>
    ...
</ServiceDefinition>
```

Service Fabric

In Service Fabric a startup entry point is configured per service in ServiceManifest.xml:

```
<ServiceManifest>
  <CodePackage Name="Code" Version="1.0.0">
    <SetupEntryPoint>
      <ExeHost>
        <Program>Startup.bat</Program>
      </ExeHost>
    </SetupEntryPoint>
    ...
</ServiceManifest>
```

A note about development environment

Both Cloud Services and Service Fabric are integrated with Visual Studio with project templates and support for debugging, configuring, and deploying both locally and to Azure. Both Cloud Services and Service Fabric also provide a local development runtime environment. The difference is that while the Cloud Service development runtime emulates the Azure environment on which it runs, Service Fabric does not use an emulator - it uses the complete Service Fabric runtime. The Service Fabric environment you run on your local development machine is the same environment that runs in production.

Next steps

Read more about Service Fabric Reliable Services and the fundamental differences between Cloud Services and Service Fabric application architecture to understand how to take advantage of the full set of Service Fabric features.

- Getting started with Service Fabric Reliable Services
- Conceptual guide to the differences between Cloud Services and Service Fabric

Create Service Fabric clusters on Windows Server or Linux

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

Chacko Daniel • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • alexandair • Mark Fussell • Ryan Wike • Rebecca Agiewich • pcw3187 • kunaldsingh • Andy De George

Azure Service Fabric allows for the creation of Service Fabric clusters on any VMs or computers running Windows Server or Linux. This means you are able to deploy and run Service Fabric applications in any environment where you have a set of Windows Server or Linux computers that are interconnected, be it on-premises, Microsoft Azure or with any cloud provider.

Create Service Fabric clusters on Azure

Creating a cluster on Azure is done either via a Resource Model template or the Azure portal. Read Create a Service Fabric cluster by using a Resource Manager template or Create a Service Fabric cluster from the Azure portal for more information.

Supported operating systems for clusters on Azure

You are able to create clusters on VMs running these operating systems:

- Windows Server 2012 R2
- Windows Server 2016 (After it is announced as generally available)
- Linux Ubuntu 16.04 (in public preview)

Create Service Fabric standalone clusters on-premise or with any cloud provider

Service Fabric provides an install package for you to create standalone Service Fabric clusters on-premises or on any cloud provider

For more information on setting up standalone service fabric clusters on Windows Server, read Service Fabric cluster creation for Windows Server

Any cloud deployments vs. on-premises deployments

The process for creating a Service Fabric cluster on-premises is similar to the process of creating a cluster on any cloud of your choice with a set of VMs. The initial steps to provision the VMs are governed by the cloud provider or on-premises environment that you are using. Once you have a set of VMs with network connectivity enabled between them, then the steps to set up the Service Fabric package, edit the cluster settings, and run the cluster creation and management scripts are identical. This ensures that your knowledge and experience of operating and managing Service Fabric clusters is transferable when you choose to target new hosting environments.

Benefits of creating standalone Service Fabric clusters

- You are free to choose any cloud provider to host your cluster.
- Service Fabric applications, once written, can be run in multiple hosting environments with minimal to no changes.

- Knowledge of building Service Fabric applications carries over from one hosting environment to another.
- Operational experience of running and managing Service Fabric clusters carries over from one environment to another.
- Broad customer reach is unbounded by hosting environment constraints.
- An extra layer of reliability and protection against widespread outages exists because you can move the services over to another deployment environment if a data center or cloud provider has a blackout.

Supported operating systems for standalone clusters

You are able to create clusters on VMs or computers running these operating systems:

- Windows Server 2012 R2
- Windows Server 2016 (After it is announced as generally available)
- Linux (coming soon)

Advantages of Service Fabric clusters on Azure over standalone Service Fabric clusters created on-premises

Running Service Fabric clusters on Azure provides advantages over the on-premises option, so if you don't have specific needs for where you run your clusters, then we suggest that you run them on Azure. On Azure, we provide integration with other Azure features and services, which makes operations and management of the cluster easier and more reliable.

- **Azure portal:** Azure portal makes it easy to create and manage clusters.
- **Azure Resource Manager:** Use of Azure Resource Manager allows easy management of all resources used by the cluster as a unit and simplifies cost tracking and billing.
- **Service Fabric Cluster as an Azure Resource** A Service Fabric cluster is an ARM resource, so you can model it like you do other ARM resources in Azure.
- **Integration with Azure Infrastructure** Service Fabric coordinates with the underlying Azure infrastructure for OS, network, and other upgrades to improve availability and reliability of your applications.
- **Diagnostics:** On Azure, we provide integration with Azure diagnostics and Log Analytics.
- **Auto-scaling:** For clusters on Azure, we provide built-in auto-scaling functionality due to Virtual Machine scale-sets. In on-premises and other cloud environments, you have to build your own auto-scaling feature or scale manually using the APIs that Service Fabric exposes for scaling clusters.

Next steps

Create a cluster on VMs or computers running Windows Server: Service Fabric cluster creation for Windows Server

Create a cluster on VMs or computers running Linux: Service Fabric on Linux

Describing a service fabric cluster

11/15/2016 • 20 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • PepijnGramberg • Eran Ben-Shahar • Natalia

The Service Fabric Cluster Resource Manager provides several mechanisms for describing a cluster. During runtime, the Cluster Resource Manager uses this information to ensure high availability of the services running in the cluster while also ensuring that the resources in the cluster are being used appropriately.

Key concepts

The Cluster Resource Manager supports several features that describe a cluster:

- Fault Domains
- Upgrade Domains
- Node Properties
- Node Capacities

Fault domains

A fault domain is any area of coordinated failure. A single machine is a fault domain (since it alone can fail for a variety of reasons, from power supply failures to drive failures to bad NIC firmware). A bunch of machines connected to the same Ethernet switch are in the same fault domain, as would be those connected to a single source of power. Since it's natural for these to overlap, Fault Domains are inherently hierarchal and are represented as URIs in Service Fabric.

If you were setting up your own cluster you'd need to think about all of these different areas of failure and make sure that your fault domains were set up correctly so that Service Fabric would know where it was safe to place services. By "safe" we really mean smart – we don't want to place services such that a loss of a fault domain (the failure of any of the components listed above, for example) causes the service to go down. In the Azure environment we leverage the fault domain information provided by the environment in order to correctly configure the nodes in the cluster on your behalf. In the graphic below (Fig. 7) we color all of the entities that reasonably result in a fault domain as a simple example and list out all of the different fault domains that result. In this example, we have datacenters (DC), racks (R), and blades (B). Conceivably, if each blade holds more than one virtual machine, there could be another layer in the fault domain hierarchy.

During run time, the Service Fabric Cluster Resource Manager considers the fault domains in the cluster and attempts to spread out the replicas for a given service so that they are all in separate fault domains. This process helps ensure that in case of failure of any one fault domain (at any level in the hierarchy), that the availability of that service is not compromised.

Service Fabric's Cluster Resource Manager doesn't really care about how many layers there are in the hierarchy, however since it does try to ensure that the loss of any one portion of the hierarchy doesn't impact the cluster or the services running on top of it, it is generally best if at each level of depth in the fault domain there are the same number of machines. This prevents one portion of the hierarchy from having to contain more services at the end of the day than others.

Configuring your cluster in such a way that the "tree" of fault domains is unbalanced makes it rather hard for the Cluster Resource Manager to figure out what the best allocation of replicas is, particularly since it means that the loss of a particular domain can overly impact the availability of the cluster – the Cluster Resource Manager is torn between using the machines in that "heavy" domain efficiently by placing services on them and placing services so that the loss of the domain doesn't cause problems.

In the diagram below we show two different example cluster layouts, one where the nodes are well distributed across the fault domains, and another where one fault domain ends up with many more nodes. Note that in Azure the choices about which nodes end up in which fault and upgrade domains is handled for you, so you should never see these sorts of imbalances. However, if you ever stand up your own cluster on-premise or in another environment, it's something you have to think about.

Upgrade domains

Upgrade Domains are another feature that helps the Service Fabric Resource Manager to understand the layout of

the cluster so that it can plan ahead for failures. Upgrade Domains define areas (sets of nodes, really) that will go down at the same time during an upgrade.

Upgrade Domains are a lot like Fault Domains, but with a couple key differences. First, Upgrade Domains are usually defined by policy; whereas Fault Domains are rigorously defined by the areas of coordinated failures (and hence usually the hardware layout of the environment). In the case of Upgrade Domains however you get to decide how many you want. Another difference is that (today at least) Upgrade Domains are not hierarchical – they are more like a simple tag than a hierarchy.

The picture below shows a fictional setup where we have three upgrade domains striped across three fault domains. It also shows one possible placement for three different replicas of a stateful service. Note that they are all in different fault and upgrade domains. This means that we could lose a fault domain while in the middle of a service upgrade and there would still be one running copy of the code and data in the cluster. Depending on your needs this could be good enough, however you may notice though that this copy could be old (as Service Fabric uses quorum based replication). In order to truly survive two failures you'd need more replicas (five at a minimum).

There are pros and cons to having large numbers of upgrade domains – the pro is that each step of the upgrade is more granular and therefore affects a smaller number of nodes or services. This results in fewer services having to move at a time, introducing less churn into the system and overall improving reliability (since less of the service will be impacted by any issue). The downside of having many upgrade domains is that Service Fabric verifies the health of each Upgrade Domain as it is upgraded and ensures that the Upgrade Domain is healthy before moving on to the next Upgrade Domain. The goal of this check is to ensure that services have a chance to stabilize and that their health is validated before the upgrade proceeds, so that any issues are detected. The tradeoff is acceptable because it prevents bad changes from affecting too much of the service at a time.

Too few upgrade domains has its own side effects – while each individual upgrade domain is down and being upgraded a large portion of your overall capacity is unavailable. For example, if you only have three upgrade domains you are taking down about 1/3 of your overall service or cluster capacity at a time. This isn't desirable as you have to have enough capacity in the rest of your cluster to cover the workload, meaning that in the normal case those nodes are less-loaded than they would otherwise be, increasing COGS.

There's no real limit to the total number of fault or upgrade domains in an environment, or constraints on how they overlap. Common structures that we've seen are 1:1 (where each unique fault domain maps to its own upgrade domain as well), an Upgrade Domain per Node (physical or virtual OS instance), and a "striped" or "matrix" model where the Fault Domains and Upgrade Domains form a matrix with machines usually running down the diagonal.

UD Per Node

	FD1	FD2	FD3
UD1	Node1		
UD2		Node2	
UD3			Node3
UD4	Node4		
UD5		Node5	
UD6			Node6
UD7	Node7		
UD8		Node8	
UD9			Node9

FD/UD Matrix (Sparse/Diagonal)

	FD1	FD2	FD3	FD4	FD5
UD1	Node1				
UD2		Node2			
UD3			Node3		
UD4				Node4	
UD5					Node5

1:1 FD & UD

FD1/UD1	FD2/UD2	FD3/UD3
Node1	Node2	Node3
Node4	Node5	Node6
Node7	Node8	Node9

FD/UD Matrix

	FD1	FD2	FD3
UD1	Node1	Node2	Node3
UD2	Node4	Node5	Node6
UD3	Node7	Node8	Node9

There's no best answer which layout to choose, each has some pros and cons. For example, the 1FD:1UD model is fairly simple to set up, whereas the 1 UD per Node model is most like what people are used to from managing small sets of machines in the past where each would be taken down independently.

The most common model (and the one that we use for the hosted Azure Service Fabric clusters) is the FD/UD matrix, where the FDs and UDs form a table and nodes are placed starting along the diagonal. Whether this ends up sparse or packed depends on the total number of nodes compared to the number of FDs and UDs (put differently, for sufficiently large clusters, almost everything ends up looking like the dense matrix pattern, shown in the bottom right option of Figure 10).

Fault and upgrade domain constraints and resulting behavior

The Cluster Resource manager treats the desire to keep a service balanced across fault and upgrade domains as a constraint. You can find out more about constraints in this article. The fault and upgrade domain constraints are defined as following: "For a given service partition there should never be a difference *greater than one* in the number of replicas between two domains." Practically what this means is that for a given service certain movements or certain arrangements might not be valid in the cluster, because doing so would violate the fault or upgrade domain constraint.

Let's take a look at one example. Let's say that we have a cluster with 6 nodes, configured with 5 fault domains and 5 upgrade domains.

	FD0	FD1	FD2	FD3	FD4
UD0	N1				
UD1	N6	N2			
UD2			N3		
UD3				N4	
UD4					N5

Now let's say that we create a service with a TargetReplicaSetSize of 5. The replicas land on N1-N5. In fact, N6 will never get used. But why? Well let's take a look at the difference between the current layout and what would happen if we had chosen N6 instead, and think about how that relates to our definition of the FD and UD constraint.

Here's the layout we got and the total number of replicas per fault and upgrade domain.

	FD0	FD1	FD2	FD3	FD4	UDTOTAL
UD0	R1					1
UD1		R2				1
UD2			R3			1
UD3				R4		1
UD4					R5	1
FDTotal	1	1	1	1	1	-

Note that this layout is balanced in terms of nodes per fault domain and upgrade domain, and it is also balanced in terms of the number of replicas per fault and upgrade domain. Each domain has the same number of nodes and the same number of replicas.

Now, let's take a look at what would happen if instead of N2, we'd used N6. How would the replicas be distributed then? Well, they'd look something like this:

	FD0	FD1	FD2	FD3	FD4	UDTOTAL
UD0	R1					1
UD1	R5					1
UD2			R2			1
UD3				R3		1
UD4					R4	1
FDTotal	2	0	1	1	1	-

This violates our definition for the fault domain constraint, since FD0 has 2 replicas, while FD1 has 0, making the total difference 2 and thus the Cluster Resource Manager will not allow this arrangement. Similarly if we had picked N2-6 we'd get:

	FD0	FD1	FD2	FD3	FD4	UDTOTAL
UD0						0
UD1	R5	R1				2
UD2			R2			1
UD3				R3		1
UD4					R4	1

	FD0	FD1	FD2	FD3	FD4	UDTOTAL
FDTotal	1	1	1	1	1	-

Which while balanced in terms of fault domains is violating the upgrade domain constraint (since UD0 has 0 replicas while UD1 has 2), and hence is invalid as well.

Configuring fault and upgrade domains

Defining Fault Domains and Upgrade Domains is done automatically in Azure hosted Service Fabric deployments; Service Fabric just picks up the environment information from Azure. In Azure both the fault and upgrade domain information looks "single level" but it really is encapsulating information from lower layers of the Azure stack and just presenting the logical fault and upgrade domains from the user's perspective.

If you're standing up your own cluster (or just want to try running a particular topology on your development machine) you'll need to provide the fault domain and upgrade domain information yourself. In this example we define a 9 node local development cluster that spans three "datacenters" (each with three racks), and three upgrade domains striped across those three datacenters. In the cluster manifest template, it looks something like this:

ClusterManifest.xml

```
  <Infrastructure>
    <!-- IsScaleMin indicates that this cluster runs on one-box /one single server -->
    <WindowsServer IsScaleMin="true">
      <NodeList>
        <Node NodeName="Node01" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType01"
FaultDomain="fd:/DC01/Rack01" UpgradeDomain="UpgradeDomain1" IsSeedNode="true" />
        <Node NodeName="Node02" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType02"
FaultDomain="fd:/DC01/Rack02" UpgradeDomain="UpgradeDomain2" IsSeedNode="true" />
        <Node NodeName="Node03" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType03"
FaultDomain="fd:/DC01/Rack03" UpgradeDomain="UpgradeDomain3" IsSeedNode="true" />
        <Node NodeName="Node04" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType04"
FaultDomain="fd:/DC02/Rack01" UpgradeDomain="UpgradeDomain1" IsSeedNode="true" />
        <Node NodeName="Node05" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType05"
FaultDomain="fd:/DC02/Rack02" UpgradeDomain="UpgradeDomain2" IsSeedNode="true" />
        <Node NodeName="Node06" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType06"
FaultDomain="fd:/DC02/Rack03" UpgradeDomain="UpgradeDomain3" IsSeedNode="true" />
        <Node NodeName="Node07" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType07"
FaultDomain="fd:/DC03/Rack01" UpgradeDomain="UpgradeDomain1" IsSeedNode="true" />
        <Node NodeName="Node08" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType08"
FaultDomain="fd:/DC03/Rack02" UpgradeDomain="UpgradeDomain2" IsSeedNode="true" />
        <Node NodeName="Node09" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType09"
FaultDomain="fd:/DC03/Rack03" UpgradeDomain="UpgradeDomain3" IsSeedNode="true" />
      </NodeList>
    </WindowsServer>
  </Infrastructure>
```

> **NOTE**
> In Azure deployments, fault domains and upgrade domains are assigned by Azure. Therefore, the definition of your nodes and roles within the infrastructure option for Azure does not include fault domain or upgrade domain information.

Placement constraints and node properties

Sometimes (in fact, most of the time) you're going to want to ensure that certain workloads run only on certain nodes or certain sets of nodes in the cluster. For example, some workload may require GPUs or SSDs while others may not. A great example of this is pretty much every n-tier architecture out there, where certain machines serve as the front end/interface serving side of the application (and hence are probably exposed to the internet) while a

different set (often with different hardware resources) handle the work of the compute or storage layers (and usually are not exposed to the internet). Service Fabric expects that even in a microservices world there are cases where particular workloads will need to run on particular hardware configurations, for example:

- an existing n-tier application has been “lifted and shifted” into a Service Fabric environment
- a workload wants to run on specific hardware for performance, scale, or security isolation reasons
- A workload needs to be isolated from other workloads for policy or resource consumption reasons

In order to support these sorts of configurations Service Fabric has a first class notion of what we call placement constraints. Placement constraints can be used to indicate where certain services should run. The set of constraints is extensible by users, meaning that people can tag nodes with custom properties and then select for those as well.

The different key/value tags on nodes are known as node placement *properties* (or just node properties), whereas the statement at the service is called a placement *constraint*. The value specified in the node property can be a string, bool, or signed long. The constraint can be any Boolean statement that operates on the different node properties in the cluster. The valid selectors in these boolean statements (which are strings) are:

- conditional checks for creating particular statements
  - "equal to" ==
  - "greater than" >
  - "less than" <
  - "not equal to" !=
  - "greater than or equal to" >=
  - "less than or equal to" <=
- boolean statements for grouping and negation
  - "and" &&
  - "or" ||
  - "not" !
- parenthesis for grouping operations
  - ()

  Here are some examples of basic constraint statements that use some of the symbols above. Note that node properties can be strings, bools, or numerical values.

  - "Foo >= 5"
  - "NodeColor != green"
  - "((OneProperty < 100) || ((AnotherProperty == false) && (OneProperty >= 100)))"

Only nodes where the overall statement evaluates to “True” can have the service placed on it. Nodes without a

property defined do not match any placement constraint that contains that property.

Service Fabric also defines some default properties which can be used automatically without the user having to define them. As of this writing the default properties defined at each node are the NodeType and the NodeName. So for example you could write a placement constraint as "(NodeType == NodeType03)". Generally we have found NodeType to be one of the most commonly used properties, as it usually corresponds 1:1 with a type of a machine, which in turn correspond to a type of workload in a traditional n-tier application architecture.

Let's say that the following node properties were defined for a given node type: ClusterManifest.xml

```
<NodeType Name="NodeType01">
  <PlacementProperties>
    <Property Name="HasSSD" Value="true"/>
    <Property Name="NodeColor" Value="green"/>
    <Property Name="SomeProperty" Value="5"/>
  </PlacementProperties>
</NodeType>
```

You can create service placement constraints for a service like this:

C#

```
FabricClient fabricClient = new FabricClient();
StatefulServiceDescription serviceDescription = new StatefulServiceDescription();
serviceDescription.PlacementConstraints = "(HasSSD == true && SomeProperty >= 4)";
// add other required servicedescription fields
//...
await fabricClient.ServiceManager.CreateServiceAsync(serviceDescription);
```

Powershell:

```
New-ServiceFabricService -ApplicationName $applicationName -ServiceName $serviceName -ServiceTypeName
$serviceType -Stateful -MinReplicaSetSize 2 -TargetReplicaSetSize 3 -PartitionSchemeSingleton -
PlacementConstraint "HasSSD == true && SomeProperty >= 4"
```

If you are sure that all nodes of NodeType01 are valid, you could also just select that node type, using placement constraints like those show in the pictures above.

One of the cool things about a service's placement constraints is that they can be updated dynamically during runtime. So if you need to, you can move a service around in the cluster, add and remove requirements, etc. Service Fabric takes care of ensuring that the service stays up and available even when these types of changes are ongoing.

C#:

```
StatefulServiceUpdateDescription updateDescription = new StatefulServiceUpdateDescription();
updateDescription.PlacementConstraints = "NodeType == NodeType01";
await fabricClient.ServiceManager.UpdateServiceAsync(new Uri("fabric:/app/service"), updateDescription);
```

Powershell:

```
Update-ServiceFabricService -Stateful -ServiceName $serviceName -PlacementConstraints "NodeType == NodeType01"
```

Placement constraints (along with many other orchestrator controls that we're going to talk about) are specified for every different named service instance. Updates always take the place (overwrite) what was previously specified.

It is also worth noting that at this point the properties on a node are defined via the cluster definition and hence cannot be updated without an upgrade to the cluster and will require each node to go down and then come back up in order to refresh its properties.

Capacity

One of the most important jobs of any orchestrator is to help manage resource consumption in the cluster. The last thing you want if you're trying to run services efficiently is a bunch of nodes which are hot (leading to resource contention and poor performance) while others are cold (wasted resources). But let's think even more basic than balancing (which we'll get to in a minute) – what about just ensuring that nodes don't run out of resources in the first place?

Service Fabric represents resources as "Metrics". Metrics are any logical or physical resource that you want to describe to Service Fabric. Examples of metrics are things like "WorkQueueDepth" or "MemoryInMb". Metrics are different from placements constraints and node properties in that node properties are generally static descriptors of the nodes themselves, whereas metrics are about resources that nodes have and that services consume when they are running on a node. So a property would be something like HasSSD and could be set to true or false, but the amount of space available on that SSD (and consumed by services) would be a metric like "DriveSpaceInMb". Capacity on the node would set the "DriveSpaceInMb" to the amount of total non-reserved space on the drive, and services would report how much of the metric they used during runtime.

If you turned off all resource *balancing*, Service Fabric's Cluster Resource Manager would still be able to ensure that no node ended up over its capacity (unless the cluster as a whole was too full). Capacity is another *constraint* which the Cluster Resource Manager uses to understand how much of a resource a node has. Both the capacity and the consumption at the service level are expressed in terms of metrics. So for example, the metric might be "MemoryInMb" - a given Node may have a capacity for MemoryInMb of 2048, while a given service can say it is currently consuming 64 of MemoryInMb.

During runtime, the Cluster Resource Manager tracks how much of each resource is present on each node (defined by its capacity) and how much is remaining (by subtracting any declared usage from each service). With this information, the Service Fabric Resource Manager can figure out where to place or move replicas so that nodes don't go over capacity.

C#:

```
StatefulServiceDescription serviceDescription = new StatefulServiceDescription();
ServiceLoadMetricDescription metric = new ServiceLoadMetricDescription();
metric.Name = "MemoryInMb";
metric.PrimaryDefaultLoad = 64;
metric.SecondaryDefaultLoad = 64;
metric.Weight = ServiceLoadMetricWeight.High;
serviceDescription.Metrics.Add(metric);
await fabricClient.ServiceManager.CreateServiceAsync(serviceDescription);
```

Powershell:

```
New-ServiceFabricService -ApplicationName $applicationName -ServiceName $serviceName -ServiceTypeName
$serviceTypeName –Stateful -MinReplicaSetSize 2 -TargetReplicaSetSize 3 -PartitionSchemeSingleton –Metric
@("Memory,High,64,64)
```

Memory Capacity	4096	768	2048
Memory Consumption	3584	0	0
Remaining	512	768	2048

You can see these in the cluster manifest:

ClusterManifest.xml

```
    <NodeType Name="NodeType02">
      <Capacities>
        <Capacity Name="MemoryInMb" Value="2048"/>
        <Capacity Name="Disk" Value="10000"/>
      </Capacities>
    </NodeType>
```

It is also possible that a service's load changes dynamically. Say that a replica's load changed from 64 to 1024, but the node it was running on at that time only had 512 (of the "MemoryInMb" metric) remaining. Because of this, where a replica or instance is currently placed becomes invalid since the combined usage of all of the replicas and instances on that node exceeds that node's capacity. We'll talk more about this scenario where load can change dynamically later, but as far as capacity goes it is handled the same way – the Cluster Resource Manager automatically kicks in and gets the node back below capacity by moving one or more of the replicas or instances on that node to different nodes. When doing this the Cluster Resource Manager tries to minimize the cost of all of the movements (we'll come back to the notion of Cost later).

Cluster capacity

So how do we keep the overall cluster from being too full? Well, with dynamic load there's actually not a lot we can do (since services can have their load spike independent of actions taken by the Cluster Resource Manager – your cluster with a lot of headroom today may be rather underpowered when you become famous tomorrow), but there are some controls that are baked in to prevent basic problems. The first thing we can do is prevent the creation of new workloads that would cause the cluster to become full.

Say that you go to create a simple stateless service and it has some load associated with it (more on default and dynamic load reporting later). For this service, let's say that it cares about some resource (let's say DiskSpace) and that by default it is going to consume 5 units of DiskSpace for every instance of the service. You want to create 3 instances of the service. Great! So that means that we need 15 units of DiskSpace to be present in the cluster in order for us to even be able to create these service instances. Service Fabric is continually calculating the overall capacity and consumption of each metric, so we can easily make the determination and reject the create service call if there's insufficient space.

Note that since the requirement is only that there be 15 units available, this space could be allocated many different

ways; it could be one remaining unit of capacity on 15 different nodes, for example, or three remaining units of capacity on 5 different nodes, etc. If there isn't sufficient capacity on three different nodes Service Fabric will reorganize the services already in the cluster in order to make room on the three necessary nodes. Such rearrangement is almost always possible unless the cluster as a whole is almost entirely full.

Buffered Capacity

Another thing that helps people manage overall cluster capacity is the notion of some reserved buffer to the capacity specified at each node. This setting is optional, but allows people to reserve some portion of the overall node capacity so that it is only used to place services during upgrades and failures – cases where the capacity of the cluster is otherwise reduced. Today buffer is specified globally per metric for all nodes via the ClusterManifest. The value you pick for the reserved capacity will be a function of which resources your services are more constrained on, as well as the number of fault and upgrade domains you have in the cluster. Generally more fault and upgrade domains means that you can pick a lower number for your buffered capacity, as you will expect smaller amounts of your cluster to be unavailable during upgrades and failures. Note that specifying the buffer percentage only makes sense if you have also specified the node capacity for a metric.

Here's an example of how to specify buffered capacity:

ClusterManifest.xml

```
        <Section Name="NodeBufferPercentage">
            <Parameter Name="DiskSpace" Value="0.10" />
            <Parameter Name="Memory" Value="0.15" />
            <Parameter Name="SomeOtherMetric" Value="0.20" />
        </Section>
```

The creation of new services will fail when the cluster is out of buffered capacity, ensuring that the cluster retains enough spare overhead such that upgrades and failures don't result in nodes being actually over capacity. The Cluster Resource Manager exposes a lot of this information via PowerShell and the Query APIs, letting you see the buffered capacity settings, the total capacity, and the current consumption for every metric in use in the cluster. Here we see an example of that output:

```
PS C:\Users\user> Get-ServiceFabricClusterLoadInformation
LastBalancingStartTimeUtc : 9/1/2015 12:54:59 AM
LastBalancingEndTimeUtc   : 9/1/2015 12:54:59 AM
LoadMetricInformation     :
                            LoadMetricName        : Metric1
                            IsBalancedBefore      : False
                            IsBalancedAfter       : False
                            DeviationBefore       : 0.192450089729875
                            DeviationAfter        : 0.192450089729875
                            BalancingThreshold    : 1
                            Action                : NoActionNeeded
                            ActivityThreshold     : 0
                            ClusterCapacity       : 189
                            ClusterLoad           : 45
                            ClusterRemainingCapacity : 144
                            NodeBufferPercentage  : 10
                            ClusterBufferedCapacity : 170
                            ClusterRemainingBufferedCapacity : 125
                            ClusterCapacityViolation : False
                            MinNodeLoadValue      : 0
                            MinNodeLoadNodeId     : 3ea71e8e01f4b0999b121abcbf27d74d
                            MaxNodeLoadValue      : 15
                            MaxNodeLoadNodeId     : 2cc648b6770be1bc9824fa995d5b68b1
```

Next steps

- For information on the architecture and information flow within the Cluster Resource manager, check out this article
- Defining Defragmentation Metrics is one way to consolidate load on nodes instead of spreading it out. To learn how to configure defragmentation, refer to this article
- Start from the beginning and get an Introduction to the Service Fabric Cluster Resource Manager
- To find out about how the Cluster Resource Manager manages and balances load in the cluster, check out the article on balancing load

Service Fabric cluster capacity planning considerations

11/15/2016 • 6 min to read • Edit on GitHub

Contributors

Chacko Daniel • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Simmen • Ryan Wike

For any production deployment, capacity planning is an important step. Here are some of the items that you have to consider as a part of that process.

- The number of node types your cluster needs to start out with
- The properties of each of node type (size, primary, internet facing, number of VMs, etc.)
- The reliability and durability characteristics of the cluster

Let us briefly review each of these items.

The number of node types your cluster needs to start out with

First, you need to figure out what the cluster you are creating is going to be used for and what kinds of applications you are planning to deploy into this cluster. If you are not clear on the purpose of the cluster, you are most likely not yet ready to enter the capacity planning process.

Establish the number of node types your cluster needs to start out with. Each node type is mapped to a Virtual Machine Scale Set. Each node type can then be scaled up or down independently, have different sets of ports open, and can have different capacity metrics. So the decision of the number of node types essentially comes down to the following considerations:

- Does your application have multiple services, and do any of them need to be public or internet facing? Typical applications contain a front-end gateway service that receives input from a client, and one or more back-end services that communicate with the front-end services. So in this case, you end up having at least two node types.
- Do your services (that make up your application) have different infrastructure needs such as greater RAM or higher CPU cycles? For example, let us assume that the application that you want to deploy contains a front-end service and a back-end service. The front-end service can run on smaller VMs (VM sizes like D2) that have ports open to the internet. The back-end service, however, is computation intensive and needs to run on larger VMs (with VM sizes like D4, D6, D15) that are not internet facing.

  In this example, although you can decide to put all the services on one node type, we recommended that you place them in a cluster with two node types. This allows for each node type to have distinct properties such as internet connectivity or VM size. The number of VMs can be scaled independently, as well.

- Since you cannot predict the future, go with facts you know of and decide on the number of node types that your applications need to start with. You can always add or remove node types later. A Service Fabric cluster must have at least one node type.

The properties of each node type

The **node type** can be seen as equivalent to roles in Cloud Services. Node types define the VM sizes, the number of VMs, and their properties. Every node type that is defined in a Service Fabric cluster is set up as a separate Virtual Machine Scale Set. VM Scale Sets are an Azure compute resource you can use to deploy and manage a

collection of virtual machines as a set. Being defined as distinct VM Scale Sets, each node type can then be scaled up or down independently, have different sets of ports open, and can have different capacity metrics.

Your cluster can have more than one node type, but the primary node type (the first one that you define on the portal) must have at least five VMs for clusters used for production workloads (or at least three VMs for test clusters). If you are creating the cluster using an Resource Manager template, then you will find a **is Primary** attribute under the node type definition. The primary node type is the node type where Service Fabric system services are placed.

Primary node type

For a cluster with multiple node types, you will need to choose one of them to be primary. Here are the characteristics of a primary node type:

- The minimum size of VMs for the primary node type is determined by the durability tier you choose. The default for the durability tier is Bronze. Scroll down for details on what the durability tier is and the values it can take.
- The minimum number of VMs for the primary node type is determined by the reliability tier you choose. The default for the reliability tier is Silver. Scroll down for details on what the reliability tier is and the values it can take.
- The Service Fabric system services (for example, the Cluster Manager service or Image Store service) are placed on the primary node type and so the reliability and durability of the cluster is determined by the reliability tier value and durability tier value you select for the primary node type.

Non-primary node type

For a cluster with multiple node types, there is one primary node type and the rest of them are non-primary. Here are the characteristics of a non-primary node type:

- The minimum size of VMs for this node type is determined by the durability tier you choose. The default for the durability tier is Bronze. Scroll down for details on what the durability tier is and the values it can take.
- The minimum number of VMs for this node type can be one. However you should choose this number based on the number of replicas of the application/services that you would like to run in this node type. The number of VMs in a node type can be increased after you have deployed the cluster.

The durability characteristics of the cluster

The durability tier is used to indicate to the system the privileges that your VMs have with the underlying Azure infrastructure. In the primary node type, this privilege allows Service Fabric to pause any VM level infrastructure request (such as a VM reboot, VM reimage, or VM migration) that impact the quorum requirements for the system services and your stateful services. In the non-primary node types, this privilege allows Service Fabric to pause any VM level infrastructure request like VM reboot, VM reimage, VM migration etc., that impact the quorum requirements for your stateful services running in it.

This privilege is expressed in the following values:

- Gold - The infrastructure Jobs can be paused for a duration of 2 hours per UD
- Silver - The infrastructure Jobs can be paused for a duration of 30 minutes per UD (This is currently not enabled for use. Once enabled this will be available on all standard VMs of single core and above).
- Bronze - No privileges. This is the default.

The reliability characteristics of the cluster

The reliability tier is used to set the number of replicas of the system services that you want to run in this cluster on the primary node type. The more the number of replicas, the more reliable the system services are in your cluster.

The reliability tier can take the following values.

- Platinum - Run the System services with a target replica set count of 9
- Gold - Run the System services with a target replica set count of 7
- Silver - Run the System services with a target replica set count of 5
- Bronze - Run the System services with a target replica set count of 3

> **NOTE**
> The reliability tier you choose determines the minimum number of nodes your primary node type must have. The reliability tier has no bearing on the max size of the cluster. So you can have a 20 node cluster, that is running at Bronze reliability.

You can choose to update the reliability of your cluster from one tier to another. Doing this will trigger the cluster upgrades needed to change the system services replica set count. Wait for the upgrade in progress to complete before making any other changes to the cluster, like adding nodes etc. You can monitor the progress of the upgrade on Service Fabric Explorer or by running Get-ServiceFabricClusterUpgrade

Next steps

Once you finish your capacity planning and set up a cluster, please read the following:

- Service Fabric cluster security
- Service Fabric health model introduction

Visualize your cluster with Service Fabric Explorer

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Sean McKenna • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Vaclav Turecek • ShawnJackson • Catherine Wang • Ross McAllister • jessebenson

Service Fabric Explorer is a web-based tool for inspecting and managing applications and nodes in an Azure Service Fabric cluster. Service Fabric Explorer is hosted directly within the cluster, so it is always available, regardless of where your cluster is running.

Connect to Service Fabric Explorer

If you have followed the instructions to prepare your development environment, you can launch Service Fabric Explorer on your local cluster by navigating to http://localhost:19080/Explorer.

> **NOTE**
>
> If you are using Internet Explorer with Service Fabric Explorer to manage a remote cluster, you need to configure some Internet Explorer settings. To ensure that all information loads correctly, go to **Tools** > **Compatibility View Settings** and uncheck **Display intranet sites in Compatibility View**.

Understand the Service Fabric Explorer layout

You can navigate through Service Fabric Explorer by using the tree on the left. At the root of the tree, the cluster dashboard provides an overview of your cluster, including a summary of application and node health.

View the cluster's layout

Nodes in a Service Fabric cluster are placed across a two-dimensional grid of fault domains and upgrade domains. This placement ensures that your applications remain available in the presence of hardware failures and application upgrades. You can view how the current cluster is laid out by using the cluster map.

View applications and services

The cluster contains two subtrees: one for applications and another for nodes.

You can use the application view to navigate through Service Fabric's logical hierarchy: applications, services, partitions, and replicas.

In the example below, the application **MyApp** consists of two services, **MyStatefulService** and **WebService**. Since **MyStatefulService** is stateful, it includes a partition with one primary and two secondary replicas. By contrast, WebSvcService is stateless and contains a single instance.

At each level of the tree, the main pane shows pertinent information about the item. For example, you can see

the health status and version for a particular service.

View the cluster's nodes

The node view shows the physical layout of the cluster. For a given node, you can inspect which applications have code deployed on that node. More specifically, you can see which replicas are currently running there.

Actions

Service Fabric Explorer offers a quick way to invoke actions on nodes, applications, and services within your cluster.

For example, to delete an application instance, simply choose the application from the tree on the left, and then choose **Actions** > **Delete Application**.

> **TIP**
> You can perform the same actions by clicking the ellipsis next to each element.

The following table lists the actions available for each entity:

ENTITY	ACTION	DESCRIPTION
Application type	Unprovision type	Removes the application package from the cluster's image store. Requires all applications of that type to be removed first.

ENTITY	ACTION	DESCRIPTION
Application	Delete Application	Delete the application, including all its services and their state (if any).
Service	Delete Service	Delete the service and its state (if any).
Node	Activate	Activate the node.
Deactivate (pause)	Pause the node in its current state. Services continue to run but Service Fabric does not proactively move anything onto or off it unless it is required to prevent an outage or data inconsistency. This action is typically used to enable debugging services on a specific node to ensure that they do not move during inspection.	
Deactivate (restart)	Safely move all in-memory services off a node and close persistent services. Typically used when the host processes or machine need to be restarted.	
Deactivate (remove data)	Safely close all services running on the node after building sufficient spare replicas. Typically used when a node (or at least its storage) is being permanently taken out of commission.	
Remove node state	Remove knowledge of a node's replicas from the cluster. Typically used when an already failed node is deemed unrecoverable.	

Since many actions are destructive, you may be asked to confirm your intent before the action is completed.

> **TIP**
> Every action that can be performed through Service Fabric Explorer can also be performed through PowerShell or a REST API, to enable automation.

You can also use Service Fabric Explorer to create new application instances for a given application type and version. Choose the application type in the tree view, then click the **Create app instance** link next to the version you'd like in the right pane.

> **NOTE**
>
> Application instances created through Service Fabric Explorer cannot currently be parameterized. They are created using default parameter values.

Connect to a remote Service Fabric cluster

Since Service Fabric Explorer is web-based and runs within the cluster, it is accessible from any browser, as long as you know the cluster's endpoint and have sufficient permissions to access it.

Discover the Service Fabric Explorer endpoint for a remote cluster

In order to reach Service Fabric Explorer for a given cluster, simply point your browser to:

http://<your-cluster-endpoint>:19080/Explorer

The full URL is also available in the cluster essentials pane of the Azure portal.

Connect to a secure cluster

You can control client access to your Service Fabric cluster either with certificates or using Azure Active Directory (AAD).

If you attempt to connect to Service Fabric Explorer on a secure cluster, you will either be required to present a client certificate or login using AAD, depending on the cluster's configuration.

Next steps

- Testability overview
- Managing your Service Fabric applications in Visual Studio
- Service Fabric application deployment using PowerShell

Connect to a secure cluster

11/22/2016 • 6 min to read • Edit on GitHub

Contributors

unknown • Ling Geng • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Dhanashri Kshirsagar

When a client connects to a Service Fabric cluster node, the client can be authenticated and secure communication established using certificate security or Azure Active Directory (AAD). This authentication ensures that only authorized users can access the cluster and deployed applications and perform management tasks. Certificate or AAD security must have been previously enabled on the cluster when the cluster was created. For more information on cluster security scenarios, see Cluster security. If you are connecting to a cluster secured with certificates, set up the client certificate on the computer that will connect to the cluster.

Connect to a secure cluster using Azure CLI

The following Azure CLI commands describe how to connect to a secure cluster.

Connect to a secure cluster using a client certificate

The certificate details must match a certificate on the cluster nodes.

If your certificate has Certificate Authorities (CAs), you need to add the parameter `--ca-cert-path` as shown in the following example:

```
 azure servicefabric cluster connect --connection-endpoint https://ip:19080 --client-key-path /tmp/key --client-cert-path /tmp/cert --ca-cert-path /tmp/ca1,/tmp/ca2
```

If you have multiple CAs, use commas as the delimiter.

If your Common Name in the certificate does not match the connection endpoint, you could use the parameter `--strict-ssl-false` to bypass the verification.

```
azure servicefabric cluster connect --connection-endpoint https://ip:19080 --client-key-path /tmp/key --client-cert-path /tmp/cert --ca-cert-path /tmp/ca1,/tmp/ca2 --strict-ssl-false
```

If you would like to skip the CA verification, you could add the `--reject-unauthorized-false` parameter, like the following command:

```
azure servicefabric cluster connect --connection-endpoint https://ip:19080 --client-key-path /tmp/key --client-cert-path /tmp/cert --reject-unauthorized-false
```

For connecting to a cluster secured with a self-signed certificate, use the following command removing both the CA verification and Common Name verification:

```
azure servicefabric cluster connect --connection-endpoint https://ip:19080 --client-key-path /tmp/key --client-cert-path /tmp/cert --strict-ssl-false --reject-unauthorized-false
```

After you connect, you should be able to run other CLI commands to interact with the cluster.

Connect to a secure cluster using PowerShell

Before you perform operations on a cluster through PowerShell, first establish a connection to the cluster. The cluster connection is used for all subsequent commands in the given PowerShell session.

Connect to an unsecure cluster

To connect to an unsecure cluster, provide the cluster endpoint address to the **Connect-ServiceFabricCluster** command:

```
Connect-ServiceFabricCluster -ConnectionEndpoint <Cluster FQDN>:19000
```

Connect to a secure cluster using Azure Active Directory

To connect to a secure cluster that uses Azure Active Directory to authorize cluster administrator access, provide the cluster certificate thumbprint and use the *AzureActiveDirectory* flag.

```
Connect-ServiceFabricCluster -ConnectionEndpoint <Cluster FQDN>:19000 `
-ServerCertThumbprint <Server Certificate Thumbprint> `
-AzureActiveDirectory
```

Connect to a secure cluster using a client certificate

Run the following PowerShell command to connect to a secure cluster that uses client certificates to authorize administrator access. Provide the cluster certificate thumbprint as well as the thumbprint of the client certificate that has been granted permissions for cluster management. The certificate details must match a certificate on the cluster nodes.

```
Connect-ServiceFabricCluster -ConnectionEndpoint <Cluster FQDN>:19000 `
          -KeepAliveIntervalInSec 10 `
          -X509Credential -ServerCertThumbprint <Certificate Thumbprint> `
          -FindType FindByThumbprint -FindValue <Certificate Thumbprint> `
          -StoreLocation CurrentUser -StoreName My
```

ServerCertThumbprint is the thumbprint of the server certificate installed on the cluster nodes. *FindValue* is the thumbprint of the admin client certificate. When the parameters are filled in, the command looks like the following example:

```
Connect-ServiceFabricCluster -ConnectionEndpoint clustername.westus.cloudapp.azure.com:19000 `
          -KeepAliveIntervalInSec 10 `
          -X509Credential -ServerCertThumbprint A8136758F4AB8962AF2BF3F27921BE1DF67F4326 `
          -FindType FindByThumbprint -FindValue 71DE04467C9ED0544D021098BCD44C71E183414E `
          -StoreLocation CurrentUser -StoreName My
```

Connect to a secure cluster using the FabricClient APIs

The Service Fabric SDK provides the FabricClient class for cluster management.

Connect to an unsecure cluster

To connect to a remote unsecured cluster, simply create a FabricClient instance and provide the cluster address:

```
FabricClient fabricClient = new FabricClient("clustername.westus.cloudapp.azure.com:19000");
```

For code that is running from within a cluster, for example, in a Reliable Service, create a FabricClient *without* specifying the cluster address. FabricClient connects to the local management gateway on the node the code is currently running on, avoiding an extra network hop.

```
FabricClient fabricClient = new FabricClient();
```

Connect to a secure cluster using a client certificate

The nodes in the cluster must have valid certificates whose common name or DNS name in SAN appears in the RemoteCommonNames property set on FabricClient. Following this process enables mutual authentication between the client and the cluster nodes.

```
string clientCertThumb = "71DE04467C9ED0544D021098BCD44C71E183414E";
string serverCertThumb = "A8136758F4AB8962AF2BF3F27921BE1DF67F4326";
string CommonName = "www.clustername.westus.azure.com";
string connection = "clustername.westus.cloudapp.azure.com:19000";

X509Credentials xc = GetCredentials(clientCertThumb, serverCertThumb, CommonName);
FabricClient fc = new FabricClient(xc, connection);
Task<bool> t = fc.PropertyManager.NameExistsAsync(new Uri("fabric:/any"));
try
{
    bool result = t.Result;
    Console.WriteLine("Cluster is connected");
}
catch (AggregateException ae)
{
    Console.WriteLine("Connect failed: {0}", ae.InnerException.Message);
}
catch (Exception e)
{
    Console.WriteLine("Connect failed: {0}", e.Message);
}

...

static X509Credentials GetCredentials(string clientCertThumb, string serverCertThumb, string name)
{
    X509Credentials xc = new X509Credentials();

    // Client certificate
    xc.StoreLocation = StoreLocation.CurrentUser;
    xc.StoreName = "MY";
    xc.FindType = X509FindType.FindByThumbprint;
    xc.FindValue = thumb;

    // Server certificate
    xc.RemoteCertThumbprints.Add(thumb);
    xc.RemoteCommonNames.Add(name);

    xc.ProtectionLevel = ProtectionLevel.EncryptAndSign;
    return xc;
}
```

Connect to a secure cluster using Azure Active Directory

Following this process enables Azure Active Directory for client identity and server certificate for server identity.

To use interactive mode which pops up an AAD interactive signin dialog:

```
string serverCertThumb = "A8136758F4AB8962AF2BF3F27921BE1DF67F4326";
string connection = "clustername.westus.cloudapp.azure.com:19000";

ClaimsCredentials claimsCredentials = new ClaimsCredentials();
claimsCredentials.ServerThumbprints.Add(serverCertThumb);

FabricClient fc = new FabricClient(
    claimsCredentials,
    connection);

try
{
    var ret = fc.ClusterManager.GetClusterManifestAsync().Result;
    Console.WriteLine(ret.ToString());
}
catch (AggregateException ae)
{
    Console.WriteLine("Connect failed: {0}", ae.InnerException.Message);
}
catch (Exception e)
{
    Console.WriteLine("Connect failed: {0}", e.Message);
}
```

To use silent mode without any human interaction:

(This example relies on Microsoft.IdentityModel.Clients.ActiveDirectory, Version: 2.19.208020213

Refer to Microsoft.IdentityModel.Clients.ActiveDirectory Namespace about how to acquire token and more information)

```
string tenantId = "c15cfcea-02c1-40dc-8466-fbd0ee0b05d2";
string clientApplicationId = "118473c2-7619-46e3-a8e4-6da8d5f56e12";
string webApplicationId = "53E6948C-0897-4DA6-B26A-EE2A38A690B4";

string token = GetAccessToken(
    tenantId,
    webApplicationId,
    clientApplicationId,
    "urn:ietf:wg:oauth:2.0:oob"
    );

string serverCertThumb = "A8136758F4AB8962AF2BF3F27921BE1DF67F4326";
string connection = "clustername.westus.cloudapp.azure.com:19000";
ClaimsCredentials claimsCredentials = new ClaimsCredentials();
claimsCredentials.ServerThumbprints.Add(serverCertThumb);
claimsCredentials.LocalClaims = token;

FabricClient fc = new FabricClient(
   claimsCredentials,
   connection);

try
{
    var ret = fc.ClusterManager.GetClusterManifestAsync().Result;
    Console.WriteLine(ret.ToString());
}
catch (AggregateException ae)
{
    Console.WriteLine("Connect failed: {0}", ae.InnerException.Message);
}
catch (Exception e)
{
    Console.WriteLine("Connect failed: {0}", e.Message);
}

...

static string GetAccessToken(
    string tenantId,
    string resource,
    string clientId,
    string redirectUri)
{
    string authorityFormat = @"https://login.microsoftonline.com/{0}";
    string authority = string.Format(CultureInfo.InvariantCulture, authorityFormat, tenantId);
    AuthenticationContext authContext = new AuthenticationContext(authority);

    string token = "";
    try
    {
        var authResult = authContext.AcquireToken(
            resource,
            clientId,
            new UserCredential("TestAdmin@clustenametenant.onmicrosoft.com", "TestPassword"));
        token = authResult.AccessToken;
    }
    catch (AdalException ex)
    {
        Console.WriteLine("Get AccessToken failed: {0}", ex.Message);
    }

    return token;
}
```

Connect to a secure cluster using Service Fabric Explorer

In order to reach Service Fabric Explorer for a given cluster, point your browser to:

```
http://<your-cluster-endpoint>:19080/Explorer
```

The full URL is also available in the cluster essentials pane of the Azure portal.

Connect to a secure cluster using Azure Active Directory

To connect to a cluster that is secured with AAD, point your browser to:

```
https://<your-cluster-endpoint>:19080/Explorer
```

You will automatically be prompted to log in with AAD.

Connect to a secure cluster using a client certificate

To connect to a cluster that is secured with certifcates, point your browser to:

```
https://<your-cluster-endpoint>:19080/Explorer
```

You will automatically be prompted to select a client certificate.

Set up a client certificate on the remote computer

At least two certificates should be used for securing the cluster, one for the cluster and server certificate and another for client access. We recommend that you also use additional secondary certificates and client access certificates. To secure the communication between a client and a cluster node using certificate security, you first need to obtain and install the client certificate. The certificate can be installed into the Personal (My) store of the local computer or the current user. You also need the thumbprint of the server certificate so that the client can authenticate the cluster.

Run the following PowerShell cmdlet to set up the client certificate on the computer from which you access the cluster.

```
Import-PfxCertificate -Exportable -CertStoreLocation Cert:\CurrentUser\My `
        -FilePath C:\docDemo\certs\DocDemoClusterCert.pfx `
        -Password (ConvertTo-SecureString -String test -AsPlainText -Force)
```

If it is a self-signed certificate, you need to import it to your machine's "trusted people" store before you can use this certificate to connect to a secure cluster.

```
Import-PfxCertificate -Exportable -CertStoreLocation Cert:\CurrentUser\TrustedPeople `
-FilePath C:\docDemo\certs\DocDemoClusterCert.pfx `
-Password (ConvertTo-SecureString -String test -AsPlainText -Force)
```

Next steps

- Service Fabric Cluster upgrade process and expectations from you
- Managing your Service Fabric applications in Visual Studio.
- Service Fabric Health model introduction
- Application Security and RunAs

Service Fabric cluster security scenarios

11/15/2016 • 6 min to read • Edit on GitHub

Contributors

Chacko Daniel • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Vaclav Turecek • Ryan Wike • Mark Fussell • Matt Kotsenas • Brian Moore • loekd • stevehug • Andy De George

A Service Fabric cluster is a resource that you own. Clusters should always be secured to prevent unauthorized users from connecting to your cluster, especially when it has production workloads running on it. Although it is possible to create an unsecured cluster, doing so will allow any anonymous user to connect to it if it exposes management endpoints to the public Internet.

This article provides an overview of the security scenarios for clusters running on Azure or standalone and the various technologies used to implement those scenarios. The cluster security scenarios are:

- Node-to-node security
- Client-to-node security
- Role-based access control (RBAC)

Node-to-node security

Secures communication between the VMs or machines in the cluster. This ensures that only computers that are authorized to join the cluster can participate in hosting applications and services in the cluster.

Clusters running on Azure or standalone clusters running on Windows can use either Certificate Security or Windows Security for Windows Server machines.

Node-to-node certificate security

Service Fabric uses X.509 server certificates that you specify as a part of the node-type configurations when you create a cluster. A quick overview of what these certificates are and how you can acquire or create them is provided at the end of this article.

Certificate security is configured while creating the cluster either through the Azure portal, Azure Resource Manager templates, or a standalone JSON template. You can specify a primary certificate and an optional secondary certificate that is used for certificate rollovers. The primary and secondary certificates you specify should be different than the admin client and read-only client certificates you specify for Client-to-node security.

For Azure read Set up a cluster by using an Azure Resource Manager template to learn how to configure certificate security in a cluster.

For standalone Windows Server read Secure a standalone cluster on Windows using X.509 certificates

Node-to-node windows security

For standalone Windows Server read Secure a standalone cluster on Windows using Windows security

Client-to-node security

Authenticates clients and secures communication between a client and individual nodes in the cluster. This type of security authenticates and secures client communications, which ensures that only authorized users can access the cluster and the applications deployed on the cluster. Clients are uniquely identified through either their Windows Security credentials or their certificate security credentials.

Clusters running on Azure or standalone clusters running on Windows can use either Certificate Security or Windows Security.

Client-to-node certificate security

Client-to-node certificate security is configured while creating the cluster either through the Azure portal, Resource Manager templates or a standalone JSON template by specifying an admin client certificate and/or a user client certificate. The admin client and user client certificates you specify should be different than the primary and secondary certificates you specify for Node-to-node security.

Clients connecting to the cluster using the admin certificate have full access to management capabilities. Clients connecting to the cluster using the read-only user client certificate have only read access to management capabilities. In other words these certificates are used for the role bases access control (RBAC) described later in this article.

For Azure read Set up a cluster by using an Azure Resource Manager template to learn how to configure certificate security in a cluster.

For standalone Windows Server read Secure a standalone cluster on Windows using X.509 certificates

Client-to-node Azure Active Directory (AAD) security on Azure

Clusters running on Azure can also secure access to the management endpoints using Azure Active Directory (AAD). See Set up a cluster by using an Azure Resource Manager template for information on how to create the necessary AAD artifacts, how to populate them during cluster creation, and how to connect to those clusters afterwards.

Security Recommendations

For Azure clusters, it is recommended that you use AAD security to authenticate clients and certificates for node-to-node security.

For standalone Windows Server clusters it is recommended that you use Windows security with group managed accounts (GMA) if you have Windows Server 2012 R2 and Active Directory. Otherwise still use Windows security with Windows accounts.

Role based access control (RBAC)

Access control allows the cluster administrator to limit access to certain cluster operations for different groups of users, making the cluster more secure. Two different access control types are supported for clients connecting to a cluster: Administrator role and User role.

Administrators have full access to management capabilities (including read/write capabilities). Users, by default, have only read access to management capabilities (for example, query capabilities), and the ability to resolve applications and services.

You specify the administrator and user client roles at the time of cluster creation by providing separate identities (certificates, AAD etc.) for each. For more information on the default access control settings and how to change the default settings, see Role based access control for Service Fabric clients.

X.509 certificates and Service Fabric

X.509 digital certificates are commonly used to authenticate clients and servers and to encrypt and digitally sign messages. For more details on these certificates, go to Working with certificates.

Some important things to consider:

- Certificates used in clusters running production workloads should be created by using a correctly configured Windows Server certificate service or obtained from an approved Certificate Authority (CA).
- Never use any temporary or test certificates in production that are created with tools such as MakeCert.exe.
- You can use a self-signed certificate, but should only do so for test clusters and not in production.

Server X.509 certificates

Server certificates have the primary task of authenticating a server (node) to clients, or authenticating a server (node) to a server (node). One of the initial checks when a client or node authenticates a node is to check the value of the common name in the Subject field. Either this common name or one of the certificates' subject alternative names must be present in the list of allowed common names.

The following article describes how to generate certificates with subject alternative names (SAN): How to add a subject alternative name to a secure LDAP certificate.

The Subject field can contain several values, each prefixed with an initialization to indicate the value type. Most commonly, the initialization is "CN" for common name; for example, "CN = www.contoso.com". It is also possible for the Subject field to be blank. If the optional Subject Alternative Name field is populated, it must contain both the common name of the certificate and one entry per subject alternative name. These are entered as DNS Name values.

The value of the Intended Purposes field of the certificate should include an appropriate value, such as "Server Authentication" or "Client Authentication".

Client X.509 certificates

Client certificates are not typically issued by a third-party certificate authority. Instead, the Personal store of the current user location typically contains client certificates placed there by a root authority, with an intended purpose of "Client Authentication". The client can use such a certificate when mutual authentication is required.

> **NOTE**
>
> All management operations on a Service Fabric cluster require server certificates. Client certificates cannot be used for management.

Next steps

This article provides conceptual information about cluster security. Next, create a cluster in Azure using a

Resource Manager template or through the Azure portal.

Disaster recovery in Azure Service Fabric

11/15/2016 • 6 min to read • Edit on GitHub

Contributors

Sean McKenna • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Andy Pasic

A critical part of delivering a high-availability cloud application is ensuring that it can survive all different types of failures, including those that are outside of your control. This article describes the physical layout of an Azure Service Fabric cluster in the context of potential disasters and provides guidance on how to deal with such disasters to limit or eliminate the risk of downtime or data loss.

Physical layout of Service Fabric clusters in Azure

To understand the risk posed by different types of failures, it is useful to know how clusters are physically laid out in Azure.

When you create a Service Fabric cluster in Azure, you are required to choose a region where it will be hosted. The Azure infrastructure then provisions the resources for that cluster within the region, most notably the number of virtual machines (VMs) requested. Let's look more closely at how and where those VMs are provisioned.

Fault domains

By default, the VMs in the cluster are evenly spread across logical groups known as fault domains (FDs), which segment the machines based on potential failures in the host hardware. Specifically, if two VMs reside in two distinct FDs, you can be sure that they do not share the same power source or network switch. As a result, a local network or power failure affecting one VM will not affect the other, allowing Service Fabric to rebalance the work load of the unresponsive machine within the cluster.

You can visualize the layout of your cluster across fault domains using the cluster map provided in Service Fabric Explorer:

> **NOTE**
> The other axis in the cluster map shows upgrade domains, which logically group nodes based on planned maintenance activities. Service Fabric clusters in Azure are always laid out across five upgrade domains.

Geographic distribution

There are currently 26 Azure regions throughout the world, with several more announced. An individual region can contain one or more physical data centers depending on demand and the availability of suitable locations, among other factors. Note, however, that even in regions that contain multiple physical data centers, there is no guarantee that your cluster's VMs will be evenly spread across those physical locations. Indeed, currently, all VMs for a given cluster are provisioned within a single physical site.

Dealing with failures

There are several types of failures that can impact your cluster, each with its own mitigation. We will look at them in order of likelihood to occur.

Individual machine failures

As mentioned, individual machine failures, either within the VM or in the hardware or software hosting it within a fault domain, pose no risk on their own. Service Fabric will typically detect the failure within seconds and respond accordingly based on the state of the cluster. For instance, if the node was hosting the primary replicas for a partition, a new primary is elected from the partition's secondary replicas. When Azure brings the failed machine back up, it will rejoin the cluster automatically and once again take on its share of the workload.

Multiple concurrent machine failures

While fault domains significantly reduce the risk of concurrent machine failures, there is always the potential for multiple random failures to bring down several machines in a cluster simultaneously.

In general, as long as a majority of the nodes remain available, the cluster will continue to operate, albeit at lower capacity as stateful replicas get packed into a smaller set of machines and fewer stateless instances are available to spread load.

Quorum loss

If a majority of the replicas for a stateful service's partition go down, that partition enters a state known as "quorum loss." At this point, Service Fabric stops allowing writes to that partition to ensure that its state remains consistent and reliable. In effect, we are choosing to accept a period of unavailability to ensure that clients are not told that their data was saved when in fact it was not. Note that if you have opted in to allowing reads from secondary replicas for that stateful service, you can continue to perform those read operations while in this state. A partition remains in quorum loss until a sufficient number of replicas come back or until the cluster administrator forces the system to move on using the Repair-ServiceFabricPartition API.

> **WARNING**
> Performing a repair action while the primary replica is down will result in data loss.

System services can also suffer quorum loss, with the impact being specific to the service in question. For instance, quorum loss in the naming service will impact name resolution, whereas quorum loss in the failover manager service will block new service creation and failovers. Note that unlike for your own services, attempting to repair system services is *not* recommended. Instead, it is preferable to simply wait until the down replicas return.

Minimizing the risk of quorum loss

You can minimize your risk of quorum loss by increasing the target replica set size for your service. It is helpful to think of the number of replicas you need in terms of the number of unavailable nodes you can tolerate at once

while remaining available for writes, keeping in mind that application or cluster upgrades can make nodes temporarily unavailable, in addition to hardware failures.

Consider the following examples assuming that you've configured your services to have a MinReplicaSetSize of three, the smallest number recommended for production services. With a TargetReplicaSetSize of three (one primary and two secondaries), a hardware failure during an upgrade (two replicas down) will result in quorum loss and your service will become read-only. Alternatively, if you have five replicas, you would be able to withstand two failures during upgrade (three replicas down) as the remaining two replicas can still form a quorum within the minimum replica set.

Data center outages or destruction

In rare cases, physical data centers can become temporarily unavailable due to loss of power or network connectivity. In these cases, your Service Fabric clusters and applications will likewise be unavailable but your data will be preserved. For clusters running in Azure, you can view updates on outages on the Azure status page.

In the highly unlikely event that an entire physical data center is destroyed, any Service Fabric clusters hosted there will be lost, along with their state.

To protect against this possibility, it is critically important to periodically backup your state to a geo-redundant store and ensure that you have validated the ability to restore it. How often you perform a backup will be dependent on your recovery point objective (RPO). Even if you have not fully implemented backup and restore yet, you should implement a handler for the `OnDataLoss` event so that you can log when it occurs as follows:

```
protected virtual Task<bool> OnDataLoss(CancellationToken cancellationToken)
{
  ServiceEventSource.Current.ServiceMessage(this, "OnDataLoss event received.");
  return Task.FromResult(false);
}
```

Software failures and other sources of data loss

As a cause of data loss, code defects in services, human operational errors, and security breaches are more common than widespread data center failures. However, in all cases, the recovery strategy is the same: take regular backups of all stateful services and exercise your ability to restore that state.

Next Steps

- Learn how to simulate various failures using the testability framework
- Read other disaster-recovery and high-availability resources. Microsoft has published a large amount of guidance on these topics. While some of these documents refer to specific techniques for use in other products, they contain many general best practices you can apply in the Service Fabric context as well:
  - Availability checklist
  - Performing a disaster recovery drill
  - Disaster recovery and high availability for Azure applications

Create a Service Fabric cluster in Azure using the Azure portal

11/15/2016 • 12 min to read • Edit on GitHub

Contributors

Chacko Daniel • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Vaclav Turecek • jurajsucik • Ryan Wike • Aaron Dandy • Sean McKenna • stevehug • Andy Pasic • coreysa • Andy De George

This is a step-by-step guide that walks you through the steps of setting up a secure Service Fabric cluster in Azure using the Azure portal. This guide walks you through the following steps:

- Set up Key Vault to manage keys for cluster security.
- Create a secured cluster in Azure through the Azure portal.
- Authenticate administrators using certificates.

> **NOTE**
>
> For more advanced security options, such as user authentication with Azure Active Directory and setting up certificates for application security, create your cluster using Azure Resource Manager.

A secure cluster is a cluster that prevents unauthorized access to management operations, which includes deploying, upgrading, and deleting applications, services, and the data they contain. An unsecure cluster is a cluster that anyone can connect to at any time and perform management operations. Although it is possible to create an unsecure cluster, it is **highly recommended to create a secure cluster**. An unsecure cluster **cannot be secured later** - a new cluster must be created.

The concepts are the same for creating secure clusters, whether the clusters are Linux clusters or Windows clusters. For more information and helper scripts for creating secure Linux clusters, please see Creating secure clusters on Linux. The parameters obtained by the helper script provided can be input directly into the portal as described in the section Create a cluster in the Azure portal.

Log in to Azure

This guide uses Azure PowerShell. When starting a new PowerShell session, log in to your Azure account and select your subscription before executing Azure commands.

Log in to your azure account:

```
Login-AzureRmAccount
```

Select your subscription:

```
Get-AzureRmSubscription
Set-AzureRmContext -SubscriptionId <guid>
```

Set up Key Vault

This part of the guide walks you through creating a Key Vault for a Service Fabric cluster in Azure and for Service

Fabric applications. For a complete guide on Key Vault, see the Key Vault getting started guide.

Service Fabric uses X.509 certificates to secure a cluster. Azure Key Vault is used to manage certificates for Service Fabric clusters in Azure. When a cluster is deployed in Azure, the Azure resource provider responsible for creating Service Fabric clusters pulls certificates from Key Vault and installs them on the cluster VMs.

The following diagram illustrates the relationship between Key Vault, a Service Fabric cluster, and the Azure resource provider that uses certificates stored in Key Vault when it creates a cluster:

Create a Resource Group

The first step is to create a new resource group specifically for Key Vault. Putting Key Vault into its own resource group is recommended so that you can remove compute and storage resource groups - such as the resource group that has your Service Fabric cluster - without losing your keys and secrets. The resource group that has your Key Vault must be in the same region as the cluster that is using it.

```
PS C:\Users\vturecek> New-AzureRmResourceGroup -Name mycluster-keyvault -Location 'West US'
WARNING: The output object type of this cmdlet will be modified in a future release.

ResourceGroupName : mycluster-keyvault
Location          : westus
ProvisioningState : Succeeded
Tags              :
ResourceId        : /subscriptions/<guid>/resourceGroups/mycluster-keyvault
```

Create Key Vault

Create a Key Vault in the new resource group. The Key Vault **must be enabled for deployment** to allow the Service Fabric resource provider to get certificates from it and install on cluster nodes:

```
    PS C:\Users\vturecek> New-AzureRmKeyVault -VaultName 'myvault' -ResourceGroupName 'mycluster-keyvault' -
Location 'West US' -EnabledForDeployment


    Vault Name                      : myvault
    Resource Group Name             : mycluster-keyvault
    Location                        : West US
    Resource ID                     : /subscriptions/<guid>/resourceGroups/mycluster-
keyvault/providers/Microsoft.KeyVault/vaults/myvault
    Vault URI                       : https://myvault.vault.azure.net
    Tenant ID                       : <guid>
    SKU                             : Standard
    Enabled For Deployment?         : False
    Enabled For Template Deployment? : False
    Enabled For Disk Encryption?    : False
    Access Policies                 :
                                      Tenant ID               :    <guid>
                                      Object ID               :    <guid>
                                      Application ID          :
                                      Display Name            :
                                      Permissions to Keys     :    get, create, delete, list, update,
import, backup, restore
                                      Permissions to Secrets  :    all


    Tags                            :
```

If you have an existing Key Vault, you can enable it for deployment using Azure CLI:

```
> azure login
> azure account set "your account"
> azure config mode arm
> azure keyvault list
> azure keyvault set-policy --vault-name "your vault name" --enabled-for-deployment true
```

Add certificates to Key Vault

Certificates are used in Service Fabric to provide authentication and encryption to secure various aspects of a cluster and its applications. For more information on how certificates are used in Service Fabric, see Service Fabric cluster security scenarios.

Cluster and server certificate (required)

This certificate is required to secure a cluster and prevent unauthorized access to it. It provides cluster security in a couple ways:

- **Cluster authentication:** Authenticates node-to-node communication for cluster federation. Only nodes that can prove their identity with this certificate can join the cluster.
- **Server authentication:** Authenticates the cluster management endpoints to a management client, so that the management client knows it is talking to the real cluster. This certificate also provides SSL for the HTTPS management API and for Service Fabric Explorer over HTTPS.

To serve these purposes, the certificate must meet the following requirements:

- The certificate must contain a private key.
- The certificate must be created for key exchange, exportable to a Personal Information Exchange (.pfx) file.
- The certificate's subject name must match the domain used to access the Service Fabric cluster. This is required to provide SSL for the cluster's HTTPS management endpoints and Service Fabric Explorer. You cannot obtain an SSL certificate from a certificate authority (CA) for the `.cloudapp.azure.com` domain. Acquire a custom domain name for your cluster. When you request a certificate from a CA the certificate's subject

name must match the custom domain name used for your cluster.

Client authentication certificates

Additional client certificates authenticate administrators for cluster management tasks. Service Fabric has two access levels: **admin** and **read-only user**. At minimum, a single certificate for administrative access should be used. For additional user-level access, a separate certificate must be provided. For more information on access roles, see role-based access control for Service Fabric clients.

You do not need to upload Client authentication certificates to Key Vault to work with Service Fabric. These certificates only need to be provided to users who are authorized for cluster management.

> **NOTE**
> Azure Active Directory is the recommended way to authenticate clients for cluster management operations. To use Azure Active Directory, you must create a cluster using Azure Resource Manager.

Application certificates (optional)

Any number of additional certificates can be installed on a cluster for application security purposes. Before creating your cluster, consider the application security scenarios that require a certificate to be installed on the nodes, such as:

- Encryption and decryption of application configuration values
- Encryption of data across nodes during replication

Application certificates cannot be configured when creating a cluster through the Azure portal. To configure application certificates at cluster setup time, you must create a cluster using Azure Resource Manager. You can also add application certificates to the cluster after it has been created.

Formatting certificates for Azure resource provider use

Private key files (.pfx) can be added and used directly through Key Vault. However, the Azure resource provider requires keys to be stored in a special JSON format that includes the .pfx as a base-64 encoded string and the private key password. To accommodate these requirements, keys must be placed in a JSON string and then stored as *secrets* in Key Vault.

To make this process easier, a PowerShell module is available on GitHub. Follow these steps to use the module:

1. Download the entire contents of the repo into a local directory.
2. Import the module in your PowerShell window:

```
  PS C:\Users\vturecek> Import-Module
"C:\users\vturecek\Documents\ServiceFabricRPHelpers\ServiceFabricRPHelpers.psm1"
```

The `Invoke-AddCertToKeyVault` command in this PowerShell module automatically formats a certificate private key into a JSON string and uploads it to Key Vault. Use it to add the cluster certificate and any additional application certificates to Key Vault. Repeat this step for any additional certificates you want to install in your cluster.

```
PS C:\Users\vturecek> Invoke-AddCertToKeyVault -SubscriptionId <guid> -ResourceGroupName mycluster-keyvault -
Location "West US" -VaultName myvault -CertificateName mycert -Password "<password>" -UseExistingCertificate -
ExistingPfxFilePath "C:\path\to\mycertkey.pfx"

    Switching context to SubscriptionId <guid>
    Ensuring ResourceGroup mycluster-keyvault in West US
    WARNING: The output object type of this cmdlet will be modified in a future release.
    Using existing valut myvault in West US
    Reading pfx file from C:\path\to\key.pfx
    Writing secret to myvault in vault myvault


Name  : CertificateThumbprint
Value : <value>

Name  : SourceVault
Value : /subscriptions/<guid>/resourceGroups/mycluster-keyvault/providers/Microsoft.KeyVault/vaults/myvault

Name  : CertificateURL
Value : https://myvault.vault.azure.net:443/secrets/mycert/4d087088df974e869f1c0978cb100e47
```

These are all the Key Vault prerequisites for configuring a Service Fabric cluster Resource Manager template that installs certificates for node authentication, management endpoint security and authentication, and any additional application security features that use X.509 certificates. At this point, you should now have the following setup in Azure:

- Key Vault resource group
  - Key Vault
    - Cluster server authentication certificate

Create cluster in the Azure portal

Search for the Service Fabric cluster resource

1. Sign in to the Azure portal.
2. Click **New** to add a new resource template. Search for the Service Fabric Cluster template in the **Marketplace** under **Everything**.
3. Select **Service Fabric Cluster** from the list.
4. Navigate to the **Service Fabric Cluster** blade, click **Create**,

5. The **Create Service Fabric cluster** blade has the following four steps.

1. Basics

In the Basics blade you need to provide the basic details for your cluster.

1. Enter the name of your cluster.
2. Enter a **user name** and **password** for Remote Desktop for the VMs.
3. Make sure to select the **Subscription** that you want your cluster to be deployed to, especially if you have multiple subscriptions.
4. Create a **new resource group**. It is best to give it the same name as the cluster, since it helps in finding them later, especially when you are trying to make changes to your deployment or delete your cluster.

> **NOTE**
> Although you can decide to use an existing resource group, it is a good practice to create a new resource group. This makes it easy to delete clusters that you do not need.

5. Select the **region** in which you want to create the cluster. You must use the same region that your Key Vault is in.

2. Cluster configuration

Configure your cluster nodes. Node types define the VM sizes, the number of VMs, and their properties. Your cluster can have more than one node type, but the primary node type (the first one that you define on the portal) must have at least five VMs, as this is the node type where Service Fabric system services are placed. Do not configure **Placement Properties** because a default placement property of "NodeTypeName" is added automatically.

> **NOTE**
>
> A common scenario for multiple node types is an application that contains a front-end service and a back-end service. You want to put the front-end service on smaller VMs (VM sizes like D2) with ports open to the Internet, but you want to put the back-end service on larger VMs (with VM sizes like D4, D6, D15, and so on) with no Internet-facing ports open.

1. Choose a name for your node type (1 to 12 characters containing only letters and numbers).
2. The minimum **size** of VMs for the primary node type is driven by the **durability** tier you choose for the cluster. The default for the durability tier is bronze. For more information on durability, see how to choose the Service Fabric cluster reliability and durability.
3. Select the VM size and pricing tier. D-series VMs have SSD drives and are highly recommended for stateful applications. Do not use any VM SKU that has partial cores or have less than 7 GB of available disk capacity.
4. The minimum **number** of VMs for the primary node type is driven by the **reliability** tier you choose. The default for the reliability tier is Silver. For more information on reliability, see how to choose the Service Fabric cluster reliability and durability.
5. Choose the number of VMs for the node type. You can scale up or down the number of VMs in a node type later on, but on the primary node type, the minimum is driven by the reliability level that you have chosen.

Other node types can have a minimum of 1 VM.

6. Configure custom endpoints. This field allows you to enter a comma-separated list of ports that you want to expose through the Azure Load Balancer to the public Internet for your applications. For example, if you plan to deploy a web application to your cluster, enter "80" here to allow traffic on port 80 into your cluster. For more information on endpoints, see communicating with applications
7. Configure cluster **diagnostics**. By default, diagnostics are enabled on your cluster to assist with troubleshooting issues. If you want to disable diagnostics change the **Status** toggle to **Off**. Turning off diagnostics is **not** recommended.
8. Select the Fabric upgrade mode you want set your cluster to. Select **Automatic**, if you want the system to automatically pick up the latest available version and try to upgrade your cluster to it. Set the mode to **Manual**, if you want to choose a supported version.

> **NOTE**
>
> We support only clusters that are running supported versions of service Fabric. By selecting the **Manual** mode, you are taking on the responsibility to upgrade your cluster to a supported version. For more details on the Fabric upgrade mode see the service-fabric-cluster-upgrade document.

3. Security

The final step is to provide certificate information to secure the cluster using the Key Vault and certificate information created earlier.

- Populate the primary certificate fields with the output obtained from uploading the **cluster certificate** to Key

Vault using the `Invoke-AddCertToKeyVault` PowerShell command.

```
Name  : CertificateThumbprint
Value : <value>

Name  : SourceVault
Value : /subscriptions/<guid>/resourceGroups/mycluster-keyvault/providers/Microsoft.KeyVault/vaults/myvault

Name  : CertificateURL
Value : https://myvault.vault.azure.net:443/secrets/mycert/4d087088df974e869f1c0978cb100e47
```

- Check the **Configure advanced settings** box to enter client certificates for **admin client** and **read-only client**. In these fields, enter the thumbprint of your admin client certificate and the thumbprint of your read-only user client certificate, if applicable. When administrators attempt to connect to the cluster, they are granted access only if they have a certificate with a thumbprint that matches the thumbprint values entered here.

4. Summary

To complete the cluster creation, click **Summary** to see the configurations that you have provided, or download the Azure Resource Manager template that that used to deploy your cluster. After you have provided the mandatory settings, the **OK** button becomes green and you can start the cluster creation process by clicking it.

You can see the creation progress in the notifications. (Click the "Bell" icon near the status bar at the upper right of your screen.) If you clicked **Pin to Startboard** while creating the cluster, you will see **Deploying Service Fabric Cluster** pinned to the **Start** board.

View your cluster status

Once your cluster is created, you can inspect your cluster in the portal:

1. Go to **Browse** and click **Service Fabric Clusters**.
2. Locate your cluster and click it.
3. You can now see the details of your cluster in the dashboard, including the cluster's public endpoint and a link to Service Fabric Explorer.

The **Node Monitor** section on the cluster's dashboard blade indicates the number of VMs that are healthy and not healthy. You can find more details about the cluster's health at Service Fabric health model introduction.

> **NOTE**
> Service Fabric clusters require a certain number of nodes to be up always to maintain availability and preserve state - referred to as "maintaining quorum". Therfore, it is typically not safe to shut down all machines in the cluster unless you have first performed a full backup of your state.

Remote connect to a Virtual Machine Scale Set instance or a cluster node

Each of the NodeTypes you specify in your cluster results in a VM Scale Set getting set-up. See Remote connect to a VM Scale Set instance for details.

Next steps

At this point, you have a secure cluster using certificates for management authentication. Next, connect to your cluster and learn how to manage application secrets.

Create a Service Fabric cluster in Azure using Azure Resource Manager

11/15/2016 • 17 min to read • Edit on GitHub

Contributors

Chacko Daniel • Ling Geng • Theano Petersen • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Joseph Molnar • loekd • Dhanashri Kshirsagar • Vaclav Turecek • Sean McKenna • stevehug • coreysa

This is a step-by-step guide that walks you through the steps of setting up a secure Azure Service Fabric cluster in Azure using Azure Resource Manager. This guide walks you through the following steps:

- Set up Key Vault to manage keys for cluster and application security.
- Create a secured cluster in Azure with Azure Resource Manager.
- Authenticate users with Azure Active Directory (AAD) for cluster management.

A secure cluster is a cluster that prevents unauthorized access to management operations, which includes deploying, upgrading, and deleting applications, services, and the data they contain. An unsecure cluster is a cluster that anyone can connect to at any time and perform management operations. Although it is possible to create an unsecure cluster, it is **highly recommended to create a secure cluster**. An unsecure cluster **cannot be secured later** - a new cluster must be created.

The concepts are the same for creating secure clusters, whether the clusters are Linux clusters or Windows clusters. For more information and helper scripts for creating secure Linux clusters, see Creating secure clusters on Linux

Log in to Azure

This guide uses Azure PowerShell. When starting a new PowerShell session, log in to your Azure account and select your subscription before executing Azure commands.

Log in to your azure account:

```
Login-AzureRmAccount
```

Select your subscription:

```
Get-AzureRmSubscription
Set-AzureRmContext -SubscriptionId <guid>
```

Set up Key Vault

This section walks through creating a Key Vault for a Service Fabric cluster in Azure and for Service Fabric applications. For a complete guide on Key Vault, refer to the Key Vault getting started guide.

Service Fabric uses X.509 certificates to secure a cluster and provide application security features. Azure Key Vault is used to manage certificates for Service Fabric clusters in Azure. When a cluster is deployed in Azure, the Azure resource provider responsible for creating Service Fabric clusters pulls certificates from Key Vault and installs them on the cluster VMs.

The following diagram illustrates the relationship between Key Vault, a Service Fabric cluster, and the Azure

resource provider that uses certificates stored in Key Vault when it creates a cluster:

Create a Resource Group

The first step is to create a resource group specifically for Key Vault. Putting Key Vault into its own resource group is recommended. This allows you to remove the compute and storage resource groups, including the the resource group that has your Service Fabric cluster without losing your keys and secrets. The resource group that has your Key Vault must be in the same region as the cluster that is using it.

```
New-AzureRmResourceGroup -Name mycluster-keyvault -Location 'West US'
WARNING: The output object type of this cmdlet is going to be modified in a future release.

ResourceGroupName : mycluster-keyvault
Location          : westus
ProvisioningState : Succeeded
Tags              :
ResourceId        : /subscriptions/<guid>/resourceGroups/mycluster-keyvault
```

Create Key Vault

Create a Key Vault in the new resource group. The Key Vault **must be enabled for deployment** to allow the Service Fabric resource provider to get certificates from it and install on cluster nodes:

```
    New-AzureRmKeyVault -VaultName 'myvault' -ResourceGroupName 'mycluster-keyvault' -Location 'West US' -
EnabledForDeployment


    Vault Name                      : myvault
    Resource Group Name             : mycluster-keyvault
    Location                        : West US
    Resource ID                     : /subscriptions/<guid>/resourceGroups/mycluster-
keyvault/providers/Microsoft.KeyVault/vaults/myvault
    Vault URI                       : https://myvault.vault.azure.net
    Tenant ID                       : <guid>
    SKU                             : Standard
    Enabled For Deployment?         : False
    Enabled For Template Deployment? : False
    Enabled For Disk Encryption?    : False
    Access Policies                 :
                                      Tenant ID               :    <guid>
                                      Object ID               :    <guid>
                                      Application ID          :
                                      Display Name            :
                                      Permissions to Keys     :    get, create, delete, list, update,
import, backup, restore
                                      Permissions to Secrets  :    all


    Tags                            :
```

If you have an existing Key Vault, you can enable it for deployment using Azure CLI:

```
> azure login
> azure account set "your account"
> azure config mode arm
> azure keyvault list
> azure keyvault set-policy --vault-name "your vault name" --enabled-for-deployment true
```

Add certificates to Key Vault

Certificates are used in Service Fabric to provide authentication and encryption to secure various aspects of a cluster and its applications. For more information on how certificates are used in Service Fabric, see Service Fabric cluster security scenarios.

Cluster and server certificate (required)

This certificate is required to secure a cluster and prevent unauthorized access to it. It provides cluster security in a couple ways:

- **Cluster authentication:** Authenticates node-to-node communication for cluster federation. Only nodes that can prove their identity with this certificate can join the cluster.
- **Server authentication:** Authenticates the cluster management endpoints to a management client, so that the management client knows it is talking to the real cluster. This certificate also provides SSL for the HTTPS management API and for Service Fabric Explorer over HTTPS.

To serve these purposes, the certificate must meet the following requirements:

- The certificate must contain a private key.
- The certificate must be created for key exchange, exportable to a Personal Information Exchange (.pfx) file.
- The certificate's subject name must match the domain used to access the Service Fabric cluster. This matchng is required to provide SSL for the cluster's HTTPS management endpoints and Service Fabric Explorer. You cannot obtain an SSL certificate from a certificate authority (CA) for the `.cloudapp.azure.com` domain. You must acquire a custom domain name for your cluster. When you request a certificate from a CA, the

certificate's subject name must match the custom domain name used for your cluster.

Application certificates (optional)

Any number of additional certificates can be installed on a cluster for application security purposes. Before creating your cluster, consider the application security scenarios that require a certificate to be installed on the nodes, such as:

- Encryption and decryption of application configuration values
- Encryption of data across nodes during replication

Formatting certificates for Azure resource provider use

Private key files (.pfx) can be added and used directly through Key Vault. However, the Azure resource provider requires keys to be stored in a special JSON format that includes the .pfx as a base-64 encoded string and the private key password. To accommodate these requirements, keys must be placed in a JSON string and then stored as *secrets* in Key Vault.

To make this process easier, a PowerShell module is available on GitHub. Follow these steps to use the module:

1. Download the entire contents of the repo into a local directory.
2. Import the module in your PowerShell window:

```
PS C:\Users\vturecek> Import-Module
"C:\users\vturecek\Documents\ServiceFabricRPHelpers\ServiceFabricRPHelpers.psm1"
```

The `Invoke-AddCertToKeyVault` command in this PowerShell module automatically formats a certificate private key into a JSON string and uploads it to Key Vault. Use it to add the cluster certificate and any additional application certificates to Key Vault. Repeat this step for any additional certificates you want to install in your cluster.

```
 Invoke-AddCertToKeyVault -SubscriptionId <guid> -ResourceGroupName mycluster-keyvault -Location "West US" -
VaultName myvault -CertificateName mycert -Password "<password>" -UseExistingCertificate -ExistingPfxFilePath
"C:\path\to\mycertkey.pfx"

    Switching context to SubscriptionId <guid>
    Ensuring ResourceGroup mycluster-keyvault in West US
    WARNING: The output object type of this cmdlet is going to be modified in a future release.
    Using existing valut myvault in West US
    Reading pfx file from C:\path\to\key.pfx
    Writing secret to myvault in vault myvault


Name  : CertificateThumbprint
Value : <value>

Name  : SourceVault
Value : /subscriptions/<guid>/resourceGroups/mycluster-keyvault/providers/Microsoft.KeyVault/vaults/myvault

Name  : CertificateURL
Value : https://myvault.vault.azure.net:443/secrets/mycert/4d087088df974e869f1c0978cb100e47
```

The preceding strings are all the Key Vault prerequisites for configuring a Service Fabric cluster Resource Manager template that installs certificates for node authentication, management endpoint security and authentication, and any additional application security features that use X.509 certificates. At this point, you should now have the following setup in Azure:

- Key Vault resource group
  - Key Vault
    - Cluster server authentication certificate

- Application certificates

Set up Azure Active Directory for client authentication

AAD enables organizations (known as tenants) to manage user access to applications, which are divided into applications with a web-based login UI and applications with a native client experience. In this document, we assume that you have already created a tenant. If not, start by reading How to get an Azure Active Directory tenant.

A Service Fabric cluster offers several entry points to its management functionality, including the web-based Service Fabric Explorer and Visual Studio. As a result, you create two AAD applications to control access to the cluster, one web application and one native application.

To simplify some of the steps involved in configuring AAD with a Service Fabric cluster, we have created a set of Windows PowerShell scripts.

> **NOTE**
> You must perform these steps *before* creating the cluster so in cases where the scripts expect cluster names and endpoints, these should be the planned values, not ones that you have already created.

1. Download the scripts to your computer.
2. Right-click the zip file, choose **Properties**, then check the **Unblock** checkbox and apply.
3. Extract the zip file.
4. Run `SetupApplications.ps1`, providing the TenantId, ClusterName, and WebApplicationReplyUrl as parameters. For example:

   ```
   .\SetupApplications.ps1 -TenantId '690ec069-8200-4068-9d01-5aaf188e557a' -ClusterName 'mycluster' -
   WebApplicationReplyUrl 'https://mycluster.westus.cloudapp.azure.com:19080/Explorer/index.html'
   ```

   You can find your **TenantId** by executing the PowerShell command ``Get-AzureSubscription```. This will display the **TenantId** for every subscription.

   The **ClusterName** is used to prefix the AAD applications created by the script. It does not need to match the actual cluster name exactly as it is only intended to make it easier for you to map AAD artifacts to the Service Fabric cluster that they're being used with.

   The **WebApplicationReplyUrl** is the default endpoint that AAD returns to your users after completing the sign-in process. You should set this to the Service Fabric Explorer endpoint for your cluster, which by default is:

   https://<cluster_domain>:19080/Explorer

   You are prompted to sign into an account that has administrative privileges for the AAD tenant. Once you do, the script proceeds to create the web and native applications to represent your Service Fabric cluster. If you look at the tenant's applications in the Azure classic portal, you should see two new entries:

   - *ClusterName*_Cluster
   - *ClusterName*_Client

   The script prints the Json required by the Azure Resource Manager template when you create the cluster in the next section so keep the PowerShell window open.

```
"azureActiveDirectory": {
  "tenantId":"<guid>",
  "clusterApplication":"<guid>",
  "clientApplication":"<guid>"
},
```

Create a Service Fabric cluster Resource Manager template

In this section, the output of the preceding PowerShell commands are used in a Service Fabric cluster Resource Manager template.

Sample Resource Manager templates are available in the Azure quick-start template gallery on GitHub. These templates can be used as a starting point for your cluster template.

Create the Resource Manager template

This guide uses the 5-node secure cluster example template and template parameters. Download `azuredeploy.json` and `azuredeploy.parameters.json` to your computer and open both files in your favorite text editor.

Add certificates

Certificates are added to a cluster Resource Manager template by referencing the Key Vault that contains the certificate keys. It is recommended that these Key Vault values are placed in a Resource Manager template parameters file to keep the Resource Manager template file reusable and free of values specific to a deployment.

Add all certificates to the VMSS osProfile

Every certificate that needs to be installed in the cluster must be configured in the osProfile section of the VMSS resource (Microsoft.Compute/virtualMachineScaleSets). This instructs the resource provider to install the certificate on the VMs. This includes the cluster certificate as well as any application security certificates you plan to use for your applications:

```
{
  "apiVersion": "2016-03-30",
  "type": "Microsoft.Compute/virtualMachineScaleSets",
  ...
  "properties": {
    ...
    "osProfile": {
      ...
      "secrets": [
        {
          "sourceVault": {
            "id": "[parameters('sourceVaultValue')]"
          },
          "vaultCertificates": [
            {
              "certificateStore": "[parameters('clusterCertificateStorevalue')]",
              "certificateUrl": "[parameters('clusterCertificateUrlValue')]"
            },
            {
              "certificateStore": "[parameters('applicationCertificateStorevalue')",
              "certificateUrl": "[parameters('applicationCertificateUrlValue')]"
            },
            ...
          ]
        }
      ]
    }
  }
}
```

Configure Service Fabric cluster certificate

The cluster authentication certificate must also be configured in the Service Fabric cluster resource (Microsoft.ServiceFabric/clusters) and in the Service Fabric extension for VMSS in the VMSS resource. This allows the Service Fabric resource provider to configure it for use for cluster authentication and server authentication for management endpoints.

VMSS resource:

```
{
  "apiVersion": "2016-03-30",
  "type": "Microsoft.Compute/virtualMachineScaleSets",
  ...
  "properties": {
    ...
    "virtualMachineProfile": {
      "extensionProfile": {
        "extensions": [
          {
            "name": "[concat('ServiceFabricNodeVmExt','_vmNodeType0Name')]",
            "properties": {
              ...
              "settings": {
                ...
                "certificate": {
                  "thumbprint": "[parameters('clusterCertificateThumbprint')]",
                  "x509StoreName": "[parameters('clusterCertificateStoreValue')]"
                },
                ...
              }
            }
          }
        ]
      }
    }
  }
}
```

Service Fabric resource:

```
{
  "apiVersion": "2016-03-01",
  "type": "Microsoft.ServiceFabric/clusters",
  "name": "[parameters('clusterName')]",
  "location": "[parameters('clusterLocation')]",
  "dependsOn": [
    "[concat('Microsoft.Storage/storageAccounts/', variables('supportLogStorageAccountName'))]"
  ],
  "properties": {
    "certificate": {
      "thumbprint": "[parameters('clusterCertificateThumbprint')]",
      "x509StoreName": "[parameters('clusterCertificateStoreValue')]"
    },
    ...
  }
}
```

Insert AAD config

The AAD configuration created earlier can be inserted directly into your Resource Manager template, however it is recommended to extract the values into parameters first into a parameters file to keep the Resource Manager template reusable and free of values specific to a deployment.

```
{
  "apiVersion": "2016-03-01",
  "type": "Microsoft.ServiceFabric/clusters",
  "name": "[parameters('clusterName')]",
  ...
  "properties": {
    "certificate": {
      "thumbprint": "[parameters('clusterCertificateThumbprint')]",
      "x509StoreName": "[parameters('clusterCertificateStorevalue')]"
    },
    ...
    "azureActiveDirectory": {
      "tenantId": "[parameters('aadTenantId')]",
      "clusterApplication": "[parameters('aadClusterApplicationId')]",
      "clientApplication": "[parameters('aadClientApplicationId')]"
    },
    ...
  }
}
```

Configure Resource Manager template parameters

Finally, use the output values from the Key Vault and AAD PowerShell commands to populate the parameters file:

```
{
    "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        ...
        "clusterCertificateStoreValue": {
            "value": "My"
        },
        "clusterCertificateThumbprint": {
            "value": "<thumbprint>"
        },
        "clusterCertificateUrlValue": {
            "value":
"https://myvault.vault.azure.net:443/secrets/myclustercert/4d087088df974e869f1c0978cb100e47"
        },
        "applicationCertificateStorevalue": {
            "value": "My"
        },
        "applicationCertificateUrlValue": {
            "value":
"https://myvault.vault.azure.net:443/secrets/myapplicationcert/2e035058ae274f869c4d0348ca100f08"
        },
        "sourceVaultvalue": {
            "value": "/subscriptions/<guid>/resourceGroups/mycluster-
keyvault/providers/Microsoft.KeyVault/vaults/myvault"
        },
        "aadTenantId": {
            "value": "<guid>"
        },
        "aadClusterApplicationId": {
            "value": "<guid>"
        },
        "aadClientApplicationId": {
            "value": "<guid>"
        },
        ...
    }
}
```

At this point, you should now have the following:

- Key Vault resource group

    - Key Vault
    - Cluster server authentication certificate
    - Data encipherment certificate
- Azure Active Directory tenant
    - AAD Application for web-based management and Service Fabric Explorer
    - AAD Application for native client management
    - Users with roles assigned
- Service Fabric cluster Resource Manager template
    - Certificates configured through Key Vault
    - Azure Active Directory configured

The following diagram illustrates where Key Vault and AAD configuration fit into your Resource Manager template.

Create the cluster

You are now ready to create the cluster using ARM deployment.

Test it

Use the following PowerShell command to test your Resource Manager template with a parameters file:

```
Test-AzureRmResourceGroupDeployment -ResourceGroupName "myresourcegroup" -TemplateFile .\azuredeploy.json -
TemplateParameterFile .\azuredeploy.parameters.json
```

Deploy it

If the Resource Manager template test passes, use the following PowerShell command to deploy your Resource Manager template with a parameters file:

```
New-AzureRmResourceGroupDeployment -ResourceGroupName "myresourcegroup" -TemplateFile .\azuredeploy.json -
TemplateParameterFile .\azuredeploy.parameters.json
```

Assign users to roles

Once you have created the applications to represent your cluster, you need to assign your users to the roles supported by Service Fabric: read-only and admin. You can do this using the Azure classic portal.

1. Navigate to your tenant and choose Applications.
2. Choose the web application, which has a name like `myTestCluster_Cluster`.
3. Click the Users tab.
4. Choose a user to assign and click the **Assign** button at the bottom of the screen.

5. Select the role to assign to the user.

> **NOTE**
>
> For more information about roles in Service Fabric, see Role-based access control for Service Fabric clients.

Create secure clusters on Linux

To make the process easier, a helper script has been provided here. For using this helper script, it is assumed that you have already Azure CLI installed, and it is in your path. Make sure that the script has permissions to execute by running `chmod +x cert_helper.py` after downloading it. The first step is to log into your Azure account using the CLI with the `azure login` command. After logging into your Azure account, use the helper with your CA

signed certificate, as the following command shows:

```
./cert_helper.py [-h] CERT_TYPE [-ifile INPUT_CERT_FILE] [-sub SUBSCRIPTION_ID] [-rgname RESOURCE_GROUP_NAME]
[-kv KEY_VAULT_NAME] [-sname CERTIFICATE_NAME] [-l LOCATION] [-p PASSWORD]

The -ifile parameter can take a .pfx or a .pem file as input, with the certificate type (pfx or pem, or ss if
it is a self-signed cert).
The parameter -h prints out the help text.
```

This command returns the following three strings as the output:

1. A SourceVaultID, which is the ID for the new KeyVault ResourceGroup it created for you.
2. A CertificateUrl for accessing the certificate.
3. A CertificateThumbprint, which is used for authentication.

The following example shows how to use the command:

```
./cert_helper.py pfx -sub "fffffff-ffff-ffff-ffff-ffffffffffff"  -rgname "mykvrg" -kv "mykevname" -ifile
"/home/test/cert.pfx" -sname "mycert" -l "East US" -p "pfxtest"
```

Executing the preceding command provides you with the three strings as follows:

```
SourceVault: /subscriptions/fffffff-ffff-ffff-ffff-
ffffffffffff/resourceGroups/mykvrg/providers/Microsoft.KeyVault/vaults/mykvname
CertificateUrl: https://myvault.vault.azure.net/secrets/mycert/00000000000000000000000000000000
CertificateThumbprint: 0xffffffffffffffffffffffffffffffffffffffff
```

The certificate's subject name must match the domain used to access the Service Fabric cluster. This is required to provide SSL for the cluster's HTTPS management endpoints and Service Fabric Explorer. You cannot obtain an SSL certificate from a certificate authority (CA) for the `.cloudapp.azure.com` domain. You must acquire a custom domain name for your cluster. When you request a certificate from a CA the certificate's subject name must match the custom domain name used for your cluster.

These are the entries needed for creating a secure service fabric cluster (without AAD) as described at Configure Resource Manager template parameters. You can connect to the secure cluster via instructions at authenticating client access to a cluster. Linux preview clusters do not support AAD authentication. You can assign admin and client roles as described in the section Assign roles to users. When specifying admin and client roles for a Linux preview cluster, you have to provide certificate thumbprints for authentication (as opposed to subject name, since no chain validation or revocation is being performed in this preview release).

If you wish to use a self-signed certificate for testing, you could use the same script to generate a self-signed certificate and upload it to KeyVault, by providing the flag `ss` instead of providing the certificate path and certificate name. For example, see the following command for creating and uploading a self-signed certificate:

```
./cert_helper.py ss -rgname "mykvrg" -sub "fffffff-ffff-ffff-ffff-ffffffffffff" -kv "mykevname"   -sname
"mycert" -l "East US" -p "selftest" -subj "mytest.eastus.cloudapp.net"
```

This command returns the same three strings, SourceVault, CertificateUrl and CertificateThumbprint, which is used to create a secure Linux cluster, along with the location where the self-signed certificate was placed. You will need the self-signed certificate to connect to the cluster. You can connect to the secure cluster via instructions at authenticating client access to a cluster. The certificate's subject name must match the domain used to access the Service Fabric cluster. This is required to provide SSL for the cluster's HTTPS management endpoints and Service Fabric Explorer. You cannot obtain an SSL certificate from a certificate authority (CA) for the `.cloudapp.azure.com` domain. You must acquire a custom domain name for your cluster. When you request a certificate from a CA the certificate's subject name must match the custom domain name used for your cluster.

The parameters provided by the helper script can be filled in the portal as described in the section Create a cluster in the Azure portal.

Next steps

At this point, you have a secure cluster with Azure Active Directory providing management authentication. Next, connect to your cluster and learn how to manage application secrets.

Troubleshoot setting up Azure Active Directory for client authentication

If you run into an issue while setting up Azure Active Directory for client authentication, review the following suggestings for potential solutions.

Service Fabric Explorer prompts for selecting certificate

Problem

After login successfully on AAD login page in Service Fabric Explorer, the browser returns to home page but prompts a dialog for selecting a certificate.

Reason

The user wasn't assigned a role in AAD cluster application. Thus AAD authentication fails on Service Fabric cluster. Service Fabric Explorer falls back to certificate authentication.

Solution

Follow the instructions of setting up AAD and assign user roles. Also, "USER ASSIGNMENT REQUIRED TO

ACCESS APP" is recommended to be turned on as `SetupApplications.ps1` does.

Connect with PowerShell fails with error: The specified credentials are invalid

Problem

When use PowerShell to connect to cluster using "AzureActiveDirectory" security mode, after login successfully on AAD login page, connection fails with error: the specified credentials are invalid shown.

Solution

Same as above.

Service Fabric Explorer signing in return failure: AADSTS50011

Problem

After login on AAD sign in page in Service Fabric Explorer, page returns sign in failure - AADSTS50011: The reply address <url> does not match the reply addresses configured for the application: <guid>.

Reason

The cluster(web) application representing Service Fabric Explorer attempts to authenticate against AAD, as part of the request it provides the redirect return URL. But it is not listed in the AAD application 'REPLY URL' list.

Solution

Add url of Service Fabric Explorer to 'REPLY URL' in the configure tab of cluster(web) application or replace one of the items in the list. Then save.

How to connect cluster with AAD authentication via PowerShell

Use the following PowerShell command example to connect Service Fabric cluster:

```
Connect-ServiceFabricCluster -ConnectionEndpoint <endpoint> -KeepAliveIntervalInSec 10 -AzureActiveDirectory -
ServerCertThumbprint <thumbprint>
```

To learn about Connect-ServiceFabricCluster cmdlet, see Connect-ServiceFabricCluster.

Can I reuse the same AAD tenant for multiple clusters?

Yes. But remember to add the URL of Service Fabric Explorer to your cluster(web) application otherwise Service Fabric Explorer won't work.

Why do I still need server certificate while AAD enabled?

FabricClient and FabricGateway perform mutual authentication. In case of AAD authentication, AAD integration provides client identity to server and server certificate is used to verify server identity. For more information about how certificate works on Service Fabric, check X.509 certificates and Service Fabric

The relationship between Service Fabric node types and Virtual Machine Scale Sets

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Chacko Daniel • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Wike

Virtual Machine Scale Sets are an Azure Compute resource you can use to deploy and manage a collection of virtual machines as a set. Every node type that is defined in a Service Fabric cluster is set up as a separate VM Scale Set. Each node type can then be scaled up or down independently, have different sets of ports open, and can have different capacity metrics.

The following screen shot shows a cluster that has two node types: FrontEnd and BackEnd. Each node type has five nodes each.

Mapping VM Scale Set instances to nodes

As you can see above, the VM Scale Set instances start from instance 0 and then goes up. The numbering is reflected in the names. For example, BackEnd_0 is instance 0 of the BackEnd VM Scale Set. This particular VM Scale Set has five instances, named BackEnd_0, BackEnd_1, BackEnd_2, BackEnd_3 and BackEnd_4.

When you scale up a VM Scale Set a new instance is created. The new VM Scale Set instance name will typically be the VM Scale Set name + the next instance number. In our example, it is BackEnd_5.

Mapping VM scale set load balancers to each node type/VM Scale Set

If you have deployed your cluster from the portal or have used the sample Resource Manager template that we provided, then when you get a list of all resources under a Resource Group then you will see the load balancers for each VM Scale Set or node type.

The name would something like: **LB-<NodeType name>**. For example, LB-sfcluster4doc-0, as shown in this screenshot:

Remote connect to a VM Scale Set instance or a cluster node

Every Node type that is defined in a cluster is set up as a separate VM Scale Set. That means the node types can be scaled up or down independently and can be made of different VM SKUs. Unlike single instance VMs, the VM Scale Set instances do not get a virtual IP address of their own. So it can be a bit challenging when you are looking for an IP address and port that you can use to remote connect to a specific instance.

Here are the steps you can follow to discover them.

Step 1: Find out the virtual IP address for the node type and then Inbound NAT rules for RDP

In order to get that, you need to get the inbound NAT rules values that were defined as a part of the resource definition for **Microsoft.Network/loadBalancers**.

In the portal, navigate to the Load balancer blade and then **Settings**.

In **Settings**, click on **Inbound NAT rules**. This now gives you the IP address and port that you can use to remote connect to the first VM Scale Set instance. In the screenshot below, it is **104.42.106.156** and **3389**

Step 2: Find out the port that you can use to remote connect to the specific VM Scale Set instance/node

Earlier in this document, I talked about how the VM Scale Set instances map to the nodes. We will use that to figure out the exact port.

The ports are allocated in ascending order of the VM Scale Set instance. so in my example for the FrontEnd node type, the ports for each of the five instances are the following. you now need to do the same mapping for your VM Scale Set instance.

VM SCALE SET INSTANCE	PORT
FrontEnd_0	3389
FrontEnd_1	3390
FrontEnd_2	3391

VM SCALE SET INSTANCE	PORT
FrontEnd_3	3392
FrontEnd_4	3393
FrontEnd_5	3394

Step 3: Remote connect to the specific VM Scale Set instance

In the screenshot below I use Remote Desktop Connection to connect to the FrontEnd_1:

How to change the RDP port range values

Before cluster deployment

When you are setting up the cluster using an Resource Manager template, you can specify the range in the **inboundNatPools**.

Go to the resource definition for **Microsoft.Network/loadBalancers**. Under that you find the description for **inboundNatPools**. Replace the *frontendPortRangeStart* and *frontendPortRangeEnd* values.

```
        }
      ],
      "inboundNatPools": [
        {
          "name": "LoadBalancerBEAddressNatPool",
          "properties": {
            "backendPort": "3389",
            "frontendIPConfiguration": {
              "id": "[variables('lbIPConfig0')]"
            },
            "frontendPortRangeEnd": "4500",
            "frontendPortRangeStart": "3389",
            "protocol": "tcp"
          }
        }
      ]
    },
```

After cluster deployment

This is a bit more involved and may result in the VMs getting recycled. You will now have to set new values using Azure PowerShell. Make sure that Azure PowerShell 1.0 or later is installed on your machine. If you have not done this before, I strongly suggest that you follow the steps outlined in How to install and configure Azure PowerShell.

Sign in to your Azure account. If this PowerShell command fails for some reason, you should check whether you have Azure PowerShell installed correctly.

```
Login-AzureRmAccount
```

Run the following to get details on your load balancer and you see the values for the description for **inboundNatPools**:

```
Get-AzureRmResource -ResourceGroupName <RGname> -ResourceType Microsoft.Network/loadBalancers -ResourceName <load
balancer name>
```

Now set *frontendPortRangeEnd* and *frontendPortRangeStart* to the values you want.

```
$PropertiesObject = @{
    #Property = value;
}
Set-AzureRmResource -PropertyObject $PropertiesObject -ResourceGroupName <RG name> -ResourceType
Microsoft.Network/loadBalancers -ResourceName <load Balancer name> -ApiVersion <use the API version that get
returned> -Force
```

Next steps

- Overview of the "Deploy anywhere" feature and a comparison with Azure-managed clusters
- Cluster security
- Service Fabric SDK and getting started

Scale a Service Fabric cluster in or out using auto-scale rules

11/15/2016 • 6 min to read • Edit on GitHub

Contributors

Chacko Daniel • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ben Katz • Ryan Wike • Sean McKenna • stevehug • Andy De George

Virtual machine scale sets are an Azure compute resource that you can use to deploy and manage a collection of virtual machines as a set. Every node type that is defined in a Service Fabric cluster is set up as a separate VM scale set. Each node type can then be scaled in or out independently, have different sets of ports open, and can have different capacity metrics. Read more about it in the Service Fabric nodetypes document. Since the Service Fabric node types in your cluster are made of VM scale sets at the backend, you need to set up auto-scale rules for each node type/VM scale set.

> **NOTE**
> Your subscription must have enough cores to add the new VMs that make up this cluster. There is no model validation currently, so you get a deployment time failure, if any of the quota limits are hit.

Choose the node type/VM scale set to scale

Currently, you are not able to specify the auto-scale rules for VM scale sets using the portal, so let us use Azure PowerShell (1.0+) to list the node types and then add auto-scale rules to them.

To get the list of VM scale set that make up your cluster, run the following cmdlets:

```
Get-AzureRmResource -ResourceGroupName <RGname> -ResourceType Microsoft.Compute/VirtualMachineScaleSets

Get-AzureRmVmss -ResourceGroupName <RGname> -VMScaleSetName <VM Scale Set name>
```

Set auto-scale rules for the node type/VM scale set

If your cluster has multiple node types, then repeat this for each node types/VM scale sets that you want to scale (in or out). Take into account the number of nodes that you must have before you set up auto-scaling. The minimum number of nodes that you must have for the primary node type is driven by the reliability level you have chosen. Read more about reliability levels.

> **NOTE**
> Scaling down the primary node type to less than the minimum number make the cluster unstable or bring it down. This could result in data loss for your applications and for the system services.

Currently the auto-scale feature is not driven by the loads that your applications may be reporting to Service Fabric. So at this time the auto-scale you get is purely driven by the performance counters that are emitted by each of the VM scale set instances.

Follow these instructions to set up auto-scale for each VM scale set.

> **NOTE**
> In a scale down scenario, unless your node type has a durability level of Gold or Silver you need to call the Remove-ServiceFabricNodeState cmdlet with the appropriate node name.

Manually add VMs to a node type/VM scale set

Follow the sample/instructions in the quick start template gallery to change the number of VMs in each Nodetype.

> **NOTE**
> Adding of VMs takes time, so do not expect the additions to be instantaneous. So plan to add capacity well in time, to allow for over 10 minutes before the VM capacity is available for the replicas/ service instances to get placed.

Manually remove VMs from the primary node type/VM scale set

> **NOTE**
> The service fabric system services run in the Primary node type in your cluster. So should never shut down or scale down the number of instances in that node types less than what the reliability tier warrants. Refer to the details on reliability tiers here.

You need to execute the following steps one VM instance at a time. This allows for the system services (and your stateful services) to be shut down gracefully on the VM instance you are removing and new replicas created on other nodes.

1. Run Disable-ServiceFabricNode with intent 'RemoveNode' to disable the node you're going to remove (the highest instance in that node type).
2. Run Get-ServiceFabricNode to make sure that the node has indeed transitioned to disabled. If not, wait until the node is disabled. You cannot hurry this step.
3. Follow the sample/instructions in the quick start template gallery to change the number of VMs by one in that Nodetype. The instance removed is the highest VM instance.
4. Repeat steps 1 through 3 as needed, but never scale down the number of instances in the primary node types less than what the reliability tier warrants. Refer to the details on reliability tiers here.

Manually remove VMs from the non-primary node type/VM scale set

> **NOTE**
> For a stateful service, you need a certain number of nodes to be always up to maintain availability and preserve state of your service. At the very minimum, you need the number of nodes equal to the target replica set count of the partition/service.

You need the execute the following steps one VM instance at a time. This allows for the system services (and your stateful services) to be shut down gracefully on the VM instance you are removing and new replicas created else where.

1. Run Disable-ServiceFabricNode with intent 'RemoveNode' to disable the node you're going to remove (the highest instance in that node type).
2. Run Get-ServiceFabricNode to make sure that the node has indeed transitioned to disabled. If not wait till the node is disabled. You cannot hurry this step.
3. Follow the sample/instructions in the quick start template gallery to change the number of VMs by one in that Nodetype. This will now remove the highest VM instance.

4. Repeat steps 1 through 3 as needed, but never scale down the number of instances in the primary node types less than what the reliability tier warrants. Refer to the details on reliability tiers here.

Behaviors you may observe in Service Fabric Explorer

When you scale up a cluster the Service Fabric Explorer will reflect the number of nodes (VM scale set instances) that are part of the cluster. However, when you scale a cluster down you will see the removed node/VM instance displayed in an unhealthy state unless you call Remove-ServiceFabricNodeState cmd with the appropriate node name.

Here is the explanation for this behavior.

The nodes listed in Service Fabric Explorer are a reflection of what the Service Fabric system services (FM specifically) knows about the number of nodes the cluster had/has. When you scale the VM scale set down, the VM was deleted but FM system service still thinks that the node (that was mapped to the VM that was deleted) will come back. So Service Fabric Explorer continues to display that node (though the health state may be error or unknown).

In order to make sure that a node is removed when a VM is removed, you have two options:

1) Choose a durability level of Gold or Silver (available soon) for the node types in your cluster, which gives you the infrastructure integration. Which will then automatically remove the nodes from our system services (FM) state when you scale down. Refer to the details on durability levels here

2) Once the VM instance has been scaled down, you need to call the Remove-ServiceFabricNodeState cmdlet.

> **NOTE**
> Service Fabric clusters require a certain number of nodes to be up at all the time in order to maintain availability and preserve state - referred to as "maintaining quorum." So, it is typically unsafe to shut down all the machines in the cluster unless you have first performed a full backup of your state.

Next steps

Read the following to also learn about planning cluster capacity, upgrading a cluster, and partitioning services:

- Plan your cluster capacity
- Cluster upgrades
- Partition stateful services for maximum scale

Upgrade an Azure Service Fabric cluster

11/15/2016 • 10 min to read • Edit on GitHub

Contributors

Chacko Daniel • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Dhanashri Kshirsagar • unknown • katiecumming • stevehug • Andy Pasic • Sean McKenna • Andy De George

For any modern system, designing for upgradability is key to achieving long-term success of your product. An Azure Service Fabric cluster is a resource that you own, but is partly managed by Microsoft. This article describes what is managed automatically and what you can configure yourself.

Controlling the fabric version that runs on your Cluster

You can set your cluster to receive automatic fabric upgrades, when Microsoft releases a new version or choose to select a supported fabric version you want your cluster to be on.

You do this by setting the "upgradeMode" cluster configuration on the portal or using Resource Manager at the time of creation or later on a live cluster

> **NOTE**
>
> Make sure to keep your cluster running a supported fabric version always. As and when we announce the release of a new version of service fabric, the previous version is marked for end of support after a minimum of 60 days from that date. the The new releases are announced on the service fabric team blog. The new release is available to choose then.

14 days prior to the expiry of the release your cluster is running, a health event is generated that puts your cluster into a warning health state. The cluster remains in a warning state until you upgrade to a supported fabric version.

Setting the upgrade mode via portal

You can set the cluster to automatic or manual when you are creating the cluster.

You can set the cluster to automatic or manual when on a live cluster, using the manage experience.

Upgrading to a new version on a cluster that is set to Manual mode via portal.

To upgrade to a new version, all you need to do is select the available version from the dropdown and save. The Fabric upgrade gets kicked off automatically. The cluster health policies (a combination of node health and the health all the applications running in the cluster) are adhered to during the upgrade.

If the cluster health policies are not met, the upgrade is rolled back. Scroll down this document to read more on how to set those custom health policies.

Once you have fixed the issues that resulted in the rollback, you need to initiate the upgrade again, by following the same steps as before.

Setting the upgrade mode via a Resource Manager template

Add the "upgradeMode" configuration to the Microsoft.ServiceFabric/clusters resource definition and set the "clusterCodeVersion" to one of the supported fabric versions as shown below and then deploy the template. The valid values for "upgradeMode" are "Manual" or "Automatic"

```
{
    "apiVersion": "2016-09-01",
    "type": "Microsoft.ServiceFabric/clusters",
    "name": "[parameters('clusterName')]",
    "location": "[parameters('clusterLocation')]",
    "dependsOn": [
        "[concat('Microsoft.Storage/storageAccounts/', parameters('supportLogStorageA
    ],
    "properties": {
        "clientCertificateCommonNames": [],
        "clientCertificateThumbprints": [],
        "clusterCodeVersion": "5.3.121.9494",
        "clusterState": "Default",
        "diagnosticsStorageAccountConfig": {
            "blobEndpoint": "[reference(concat('Microsoft.Storage/storageAccounts/',
            "protectedAccountKeyName": "StorageAccountKey1",
            "queueEndpoint": "[reference(concat('Microsoft.Storage/storageAccounts/',
            "storageAccountName": "[parameters('supportLogStorageAccountName')]",
            "tableEndpoint": "[reference(concat('Microsoft.Storage/storageAccounts/',
        },
        "fabricSettings": [],
        "managementEndpoint": "[concat('http://',reference(concat(parameters('lbIPNam
        "nodeTypes": [
            {
                "name": "[parameters('vmNodeType0Name')]",
                "applicationPorts": {
                    "endPort": "[parameters('nt0applicationEndPort')]",
                    "startPort": "[parameters('nt0applicationStartPort')]"
                },
                "clientConnectionEndpointPort": "[parameters('nt0fabricTcpGatewayPort
                "durabilityLevel": "Bronze",
                "ephemeralPorts": {
                    "endPort": "[parameters('nt0ephemeralEndPort')]",
                    "startPort": "[parameters('nt0ephemeralStartPort')]"
                },
                "httpGatewayEndpointPort": "[parameters('nt0fabricHttpGatewayPort')]",
                "isPrimary": true,
                "vmInstanceCount": "[parameters('nt0InstanceCount')]"
            }
        ],
        "provisioningState": "Default",
        "reliabilityLevel": "Silver",
        "upgradeMode": "Manual",
        "vmImage": "Windows"
    }
```

Upgrading to a new version on a cluster that is set to Manual mode via a Resource Manager template.

When the cluster is in Manual mode, to upgrade to a new version, change the "clusterCodeVersion" to a supported version and deploy it. The deployment of the template, kicks of the Fabric upgrade gets kicked off automatically. The cluster health policies (a combination of node health and the health all the applications running in the cluster) are adhered to during the upgrade.

If the cluster health policies are not met, the upgrade is rolled back. Scroll down this document to read more on how to set those custom health policies.

Once you have fixed the issues that resulted in the rollback, you need to initiate the upgrade again, by following the same steps as before.

Get list of all available version for all environments for a given subscription

Run the following command, and you should get an output similar to this.

"supportExpiryUtc" tells your when a given release is expiring or has expired. The latest release does not have a valid date - it has a value of "9999-12-31T23:59:59.9999999", which just means that the expiry date is not yet set.

```
GET https://<endpoint>/subscriptions/{{subscriptionId}}/providers/Microsoft.ServiceFabric/clusterVersions?api-version= 2016-09-01

Output:
{
                   "value": [
                     {
                       "id": "subscriptions/35349203-a0b3-405e-8a23-9f1450984307/providers/Microsoft.ServiceFabric/environments/Windows/clusterVersions/5.0.1427.9490",
                       "name": "5.0.1427.9490",
                       "type": "Microsoft.ServiceFabric/environments/clusterVersions",
                       "properties": {
                         "codeVersion": "5.0.1427.9490",
                         "supportExpiryUtc": "2016-11-26T23:59:59.9999999",
                         "environment": "Windows"
                       }
                     },
                     {
                       "id": "subscriptions/35349203-a0b3-405e-8a23-9f1450984307/providers/Microsoft.ServiceFabric/environments/Windows/clusterVersions/4.0.1427.9490",
                       "name": "5.1.1427.9490",
                       "type": " Microsoft.ServiceFabric/environments/clusterVersions",
                       "properties": {
                         "codeVersion": "5.1.1427.9490",
                         "supportExpiryUtc": "9999-12-31T23:59:59.9999999",
                         "environment": "Windows"
                       }
                     },
                     {
                       "id": "subscriptions/35349203-a0b3-405e-8a23-9f1450984307/providers/Microsoft.ServiceFabric/environments/Windows/clusterVersions/4.4.1427.9490",
                       "name": "4.4.1427.9490",
                       "type": " Microsoft.ServiceFabric/environments/clusterVersions",
                       "properties": {
                         "codeVersion": "4.4.1427.9490",
                         "supportExpiryUtc": "9999-12-31T23:59:59.9999999",
                         "environment": "Linux"
                       }
                     }
                   ]
                 }
```

Fabric upgrade behavior when the cluster Upgrade Mode is Automatic

Microsoft maintains the fabric code and configuration that runs in an Azure cluster. We perform automatic monitored upgrades to the software on an as-needed basis. These upgrades could be code, configuration, or both. To make sure that your application suffers no impact or minimal impact due to these upgrades, we perform the upgrades in the following phases:

Phase 1: An upgrade is performed by using all cluster health policies

During this phase, the upgrades proceed one upgrade domain at a time, and the applications that were running in the cluster continue to run without any downtime. The cluster health policies (a combination of node health and the health all the applications running in the cluster) are adhered to during the upgrade.

If the cluster health policies are not met, the upgrade is rolled back. Then an email is sent to the owner of the subscription. The email contains the following information:

- Notification that we had to roll back a cluster upgrade.
- Suggested remedial actions, if any.
- The number of days (n) until we execute Phase 2.

We try to execute the same upgrade a few more times in case any upgrades failed for infrastructure reasons. After

the n days from the date the email was sent, we proceed to Phase 2.

If the cluster health policies are met, the upgrade is considered successful and marked complete. This can happen during the initial upgrade or any of the upgrade reruns in this phase. There is no email confirmation of a successful run. This is to avoid sending you too many emails--receiving an email should be seen as an exception to normal. We expect most of the cluster upgrades to succeed without impacting your application availability.

Phase 2: An upgrade is performed by using default health policies only

The health policies in this phase are set in such a way that the number of applications that were healthy at the beginning of the upgrade remains the same for the duration of the upgrade process. As in Phase 1, the Phase 2 upgrades proceed one upgrade domain at a time, and the applications that were running in the cluster continue to run without any downtime. The cluster health policies (a combination of node health and the health all the applications running in the cluster) are adhered to for the duration of the upgrade.

If the cluster health policies in effect are not met, the upgrade is rolled back. Then an email is sent to the owner of the subscription. The email contains the following information:

- Notification that we had to roll back a cluster upgrade.
- Suggested remedial actions, if any.
- The number of days (n) until we execute Phase 3.

We try to execute the same upgrade a few more times in case any upgrades failed for infrastructure reasons. A reminder email is sent a couple of days before n days are up. After the n days from the date the email was sent, we proceed to Phase 3. The emails we send you in Phase 2 must be taken seriously and remedial actions must be taken.

If the cluster health policies are met, the upgrade is considered successful and marked complete. This can happen during the initial upgrade or any of the upgrade reruns in this phase. There is no email confirmation of a successful run.

Phase 3: An upgrade is performed by using aggressive health policies

These health policies in this phase are geared towards completion of the upgrade rather than the health of the applications. Very few cluster upgrades end up in this phase. If your cluster gets to this phase, there is a good chance that your application becomes unhealthy and/or lose availability.

Similar to the other two phases, Phase 3 upgrades proceed one upgrade domain at a time.

If the cluster health policies are not met, the upgrade is rolled back. We try to execute the same upgrade a few more times in case any upgrades failed for infrastructure reasons. After that, the cluster is pinned, so that it will no longer receive support and/or upgrades.

An email with this information is sent to the subscription owner, along with the remedial actions. We do not expect any clusters to get into a state where Phase 3 has failed.

If the cluster health policies are met, the upgrade is considered successful and marked complete. This can happen during the initial upgrade or any of the upgrade reruns in this phase. There is no email confirmation of a successful run.

Cluster configurations that you control

In addition to the ability to set the cluster upgrade mode, Here are the configurations that you can change on a live cluster.

Certificates

You can add new or delete certificates for the cluster and client via the portal easily. Refer to this document for detailed instructions

Application ports

You can change application ports by changing the Load Balancer resource properties that are associated with the node type. You can use the portal, or you can use Resource Manager PowerShell directly.

To open a new port on all VMs in a node type, do the following:

1. Add a new probe to the appropriate load balancer.

   If you deployed your cluster by using the portal, the load balancers are named "LB-name of the Resource group-NodeTypename", one for each node type. Since the load balancer names are unique only within a resource group, it is best if you search for them under a specific resource group.

2. Add a new rule to the load balancer.

   Add a new rule to the same load balancer by using the probe that you created in the previous step.

Placement properties

For each of the node types, you can add custom placement properties that you want to use in your applications. NodeType is a default property that you can use without adding it explicitly.

> **NOTE**
>
> For details on the use of placement constraints, node properties, and how to define them, refer to the section "Placement Constraints and Node Properties" in the Service Fabric Cluster Resource Manager Document on Describing Your Cluster.

Capacity metrics

For each of the node types, you can add custom capacity metrics that you want to use in your applications to report load. For details on the use of capacity metrics to report load, refer to the Service Fabric Cluster Resource Manager Documents on Describing Your Cluster and Metrics and Load.

Fabric upgrade settings - Health polices

You can specify custom health polices for fabric upgrade. If you have set your cluster to Automatic fabric upgrades, then these policies get applied to the Phase-1 of the automatic fabric upgrades. If you have set your cluster for Manual fabric upgrades, then these policies get applied each time you select a new version triggering the system to kick off the fabric upgrade in your cluster. If you do not override the policies, the defaults are used.

You can specify the custom health policies or review the current settings under the "fabric upgrade" blade, by selecting the advanced upgrade settings. Review the following picture on how to.

Customize Fabric settings for your cluster

Refer to service fabric cluster fabric settings on what and how you can customize them.

OS patches on the VMs that make up the cluster

This capability is planned for the future as an automated feature. But currently, you are responsible to patch your VMs. You must do this one VM at a time, so that you do not take down more than one at a time.

OS upgrades on the VMs that make up the cluster

If you must upgrade the OS image on the virtual machines of the cluster, you must do it one VM at a time. You are responsible for this upgrade--there is currently no automation for this.

Next steps

- Learn how to customize some of the service fabric cluster fabric settings
- Learn how to scale your cluster in and out
- Learn about application upgrades

Delete a Service Fabric cluster on Azure and the resources it uses

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

Chacko Daniel • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Wike

A Service Fabric cluster is made up of many other Azure resources in addition to the cluster resource itself. So to completely delete a Service Fabric cluster you also need to delete all the resources it is made of. You have two options: Either delete the resource group that the cluster is in (which deletes the cluster resource and any other resources in the resource group) or specifically delete the cluster resource and it's associated resources (but not other resources in the resource group).

> **NOTE**
> Deleting the cluster resource **does not** delete all the other resources that your Service Fabric cluster is composed of.

Delete the entire resource group (RG) that the Service Fabric cluster is in

This is the easiest way to ensure that you delete all the resources associated with your cluster, including the resource group. You can delete the resource group using PowerShell or through the Azure portal. If your resource group has resources that are not related to Service fabric cluster, then you can delete specific resources.

Delete the resource group using Azure PowerShell

You can also delete the resource group by running the following Azure PowerShell cmdlets. Make sure Azure PowerShell 1.0 or greater is installed on your computer. If you have not done this before, follow the steps outlined in How to install and Configure Azure PowerShell.

Open a PowerShell window and run the following PS cmdlets:

```
Login-AzureRmAccount

Remove-AzureRmResourceGroup -Name <name of ResouceGroup> -Force
```

You will get a prompt to confirm the deletion if you did not use the *-Force* option. On confirmation the RG and all the resources it contains are deleted.

Delete a resource group in the Azure portal

1. Login to the Azure portal.
2. Navigate to the Service Fabric cluster you want to delete.
3. Click on the Resource Group name on the cluster essentials page.
4. This brings up the **Resource Group Essentials** page.
5. Click **Delete**.
6. Follow the instructions on that page to complete the deletion of the resource group.

Delete the cluster resource and the resources it uses, but not other resources in the resource group

If your resource group has only resources that are related to the Service Fabric cluster you want to delete, then it is easier to delete the entire resource group. If you want to selectively delete the resources one-by-one in your resource group, then follow these steps.

If you deployed your cluster using the portal or using one of the Service Fabric Resource Manager templates from the template gallery, then all the resources that the cluster uses are tagged with the following two tags. You can use them to decide which resources you want to delete.

Tag#1: Key = clusterName, Value = 'name of the cluster'

Tag#2: Key = resourceName, Value = ServiceFabric

Delete specific resources in the Azure portal

1. Login to the Azure portal.
2. Navigate to the Service Fabric cluster you want to delete.
3. Go to **All settings** on the essentials blade.
4. Click on **Tags** under **Resource Management** in the settings blade.
5. Click on one of the **Tags** in the tags blade to get a list of all the resources with that tag.

6. Once you have the list of tagged resources, click on each of the resources and delete them.

Delete the resources using Azure PowerShell

You can delete the resources one-by-one by running the following Azure PowerShell cmdlets. Make sure Azure PowerShell 1.0 or greater is installed on your computer. If you have not done this before, follow the steps outlined in How to install and Configure Azure PowerShell.

Open a PowerShell window and run the following PS cmdlets:

```
Login-AzureRmAccount
```

For each of the resources you want to delete, run the following:

```
Remove-AzureRmResource -ResourceName "<name of the Resource>" -ResourceType "<Resource Type>" -ResourceGroupName
"<name of the resource group>" -Force
```

To delete the cluster resource, run the following:

```
Remove-AzureRmResource -ResourceName "<name of the Resource>" -ResourceType "Microsoft.ServiceFabric/clusters" -
ResourceGroupName "<name of the resource group>" -Force
```

Next steps

Read the following to also learn about upgrading a cluster and partitioning services:

- Learn about cluster upgrades
- Learn about partitioning stateful services for maximum scale

Role-based access control for Service Fabric clients

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Mani Ramaswamy • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Rebecca Agiewich

Azure Service Fabric supports two different access control types for clients that are connected to a Service Fabric cluster: administrator and user. Access control allows the cluster administrator to limit access to certain cluster operations for different groups of users, making the cluster more secure.

Administrators have full access to management capabilities (including read/write capabilities). By default, **users** only have read access to management capabilities (for example, query capabilities), and the ability to resolve applications and services.

You specify the two client roles (administrator and client) at the time of cluster creation by providing separate certificates for each. See Service Fabric cluster security for details on setting up a secure Service Fabric cluster.

Default access control settings

The administrator access control type has full access to all the FabricClient APIs. It can perform any reads and writes on the Service Fabric cluster, including the following operations:

Application and service operations

- **CreateService**: service creation
- **CreateServiceFromTemplate**: service creation from template
- **UpdateService**: service updates
- **DeleteService**: service deletion
- **ProvisionApplicationType**: application type provisioning
- **CreateApplication**: application creation
- **DeleteApplication**: application deletion
- **UpgradeApplication**: starting or interrupting application upgrades
- **UnprovisionApplicationType**: application type unprovisioning
- **MoveNextUpgradeDomain**: resuming application upgrades with an explicit update domain
- **ReportUpgradeHealth**: resuming application upgrades with the current upgrade progress
- **ReportHealth**: reporting health
- **PredeployPackageToNode**: predeployment API
- **CodePackageControl**: restarting code packages
- **RecoverPartition**: recovering a partition
- **RecoverPartitions**: recovering partitions
- **RecoverServicePartitions**: recovering service partitions
- **RecoverSystemPartitions**: recovering system service partitions

Cluster operations

- **ProvisionFabric**: MSI and/or cluster manifest provisioning
- **UpgradeFabric**: starting cluster upgrades
- **UnprovisionFabric**: MSI and/or cluster manifest unprovisioning
- **MoveNextFabricUpgradeDomain**: resuming cluster upgrades with an explicit update domain

- **ReportFabricUpgradeHealth**: resuming cluster upgrades with the current upgrade progress
- **StartInfrastructureTask**: starting infrastructure tasks
- **FinishInfrastructureTask**: finishing infrastructure tasks
- **InvokeInfrastructureCommand**: infrastructure task management commands
- **ActivateNode**: activating a node
- **DeactivateNode**: deactivating a node
- **DeactivateNodesBatch**: deactivating multiple nodes
- **RemoveNodeDeactivations**: reverting deactivation on multiple nodes
- **GetNodeDeactivationStatus**: checking deactivation status
- **NodeStateRemoved**: reporting node state removed
- **ReportFault**: reporting fault
- **FileContent**: image store client file transfer (external to cluster)
- **FileDownload**: image store client file download initiation (external to cluster)
- **InternalList**: image store client file list operation (internal)
- **Delete**: image store client delete operation
- **Upload**: image store client upload operation
- **NodeControl**: starting, stopping, and restarting nodes
- **MoveReplicaControl**: moving replicas from one node to another

Miscellaneous operations

- **Ping**: client pings
- **Query**: all queries allowed
- **NameExists**: naming URI existence checks

The user access control type is, by default, limited to the following operations:

- **EnumerateSubnames**: naming URI enumeration
- **EnumerateProperties**: naming property enumeration
- **PropertyReadBatch**: naming property read operations
- **GetServiceDescription**: long-poll service notifications and reading service descriptions
- **ResolveService**: complaint-based service resolution
- **ResolveNameOwner**: resolving naming URI owner
- **ResolvePartition**: resolving system services
- **ServiceNotifications**: event-based service notifications
- **GetUpgradeStatus**: polling application upgrade status
- **GetFabricUpgradeStatus**: polling cluster upgrade status
- **InvokeInfrastructureQuery**: querying infrastructure tasks
- **List**: image store client file list operation
- **ResetPartitionLoad**: resetting load for a failover unit
- **ToggleVerboseServicePlacementHealthReporting**: toggling verbose service placement health reporting

The admin access control also has access to the preceding operations.

Changing default settings for client roles

In the cluster manifest file, you can provide admin capabilities to the client if needed. You can change the defaults by going to the **Fabric Settings** option during cluster creation, and providing the preceding settings in the **name**, **admin**, **user**, and **value** fields.

Next steps

Service Fabric cluster security

Service Fabric cluster creation

Customize Service Fabric cluster settings and Fabric Upgrade policy

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Chacko Daniel • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

This document tells you how to customize the various fabric settings and the fabric upgrade policy for your service fabric cluster. You can customize them on the portal or using a Resource Manager template.

Fabric settings that you can customize

Here are the Fabric settings that you can customize:

Section Name: Security

PARAMETER	ALLOWED VALUES	GUIDANCE OR SHORT DESCRIPTION
ClusterProtectionLevel	None or EncryptAndSign	None (default) for unsecured clusters, EncryptAndSign for secure clusters.

Section Name: Hosting

PARAMETER	ALLOWED VALUES	GUIDANCE OR SHORT DESCRIPTION
ServiceTypeRegistrationTimeout	Time in Seconds, default is 300	Maximum time allowed for the ServiceType to be registered with fabric
ServiceTypeDisableFailureThreshold	Whole number, default is 1	This is the threshold for the failure count after which FailoverManager (FM) is notified to disable the service type on that node and try a different node for placement.
ActivationRetryBackoffInterval	Time in Seconds, default is 5	Backoff interval on every activation failure; On every continuous activation failure, the system retries the activation for up to the MaxActivationFailureCount. The retry interval on every try is a product of continuous activation failure and the activation back-off interval.
ActivationMaxRetryInterval	Time in seconds, default is 300	On every continuous activation failure, the system retries the activation for up to ActivationMaxFailureCount. ActivationMaxRetryInterval specifies Wait time interval before retry after every activation failure
ActivationMaxFailureCount	Whole number, default is 10	Number of times system retries failed activation before giving up

Section Name: FailoverManager

PARAMETER	ALLOWED VALUES	GUIDANCE OR SHORT DESCRIPTION
PeriodicLoadPersistInterval	Time in seconds, default is 10	This determines how often the FM check for new load reports

Section Name: Federation

PARAMETER	ALLOWED VALUES	GUIDANCE OR SHORT DESCRIPTION
LeaseDuration	Time in seconds, default is 30	Duration that a lease lasts between a node and its neighbors.
LeaseDurationAcrossFaultDomain	Time in seconds, default is 30	Duration that a lease lasts between a node and its neighbors across fault domains.

Section Name: ClusterManager

PARAMETER	ALLOWED VALUES	GUIDANCE OR SHORT DESCRIPTION
UpgradeStatusPollInterval	Time in seconds, default is 60	The frequency of polling for application upgrade status. This value determines the rate of update for any GetApplicationUpgradeProgress call
UpgradeHealthCheckInterval	Time in seconds, default is 60	The frequency of health status checks during a monitored application upgrades
FabricUpgradeStatusPollInterval	Time in seconds, default is 60	The frequency of polling for Fabric upgrade status. This value determines the rate of update for any GetFabricUpgradeProgress call
FabricUpgradeHealthCheckInterval	Time in seconds, default is 60	The frequency of health status check during a monitored Fabric upgrade

Next steps

Read these articles for more information on cluster management:

Add, Roll over, remove certificates from your Azure cluster

Create and manage a cluster running on Windows Server

11/15/2016 • 12 min to read • Edit on GitHub

Contributors

Chacko Daniel • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Dhanashri Kshirsagar • carolinacmoravia • ShawnJackson • Ryan Wike • Mark Fussell • Max Burlik

You can use Azure Service Fabric to create Service Fabric clusters on any virtual machines or computers running Windows Server. This means you can deploy and run Service Fabric applications in any environment that contains a set of interconnected Windows Server computers, be it on premises or with any cloud provider. Service Fabric provides a setup package to create Service Fabric clusters called the standalone Windows Server package.

This article walks you through the steps for creating a cluster by using the standalone package for Service Fabric on premises, though it can be easily adapted for any other environment such as other cloud providers.

> **NOTE**
> This standalone Windows Server package may contain features that are currently in preview and are not supported for commercial use. To see the list of features that are in preview, see "Preview features included in this package." You can also download a copy of the EULA now.

Get support for the Service Fabric standalone package

- Ask the community about the Service Fabric standalone package for Windows Server in the Azure Service Fabric forum.
- Open a ticket for Professional Support for Service Fabric. Learn more about Professional Support from Microsoft.

Download the Service Fabric standalone package

Download the standalone package for Service Fabric for Windows Server 2012 R2 and later, which is named Microsoft.Azure.ServiceFabric.WindowsServer.<version>.zip.

In the download package, you will find the following files:

FILE NAME	SHORT DESCRIPTION
MicrosoftAzureServiceFabric.cab	The .cab file that contains the binaries that are deployed to each machine in the cluster.
ClusterConfig.Unsecure.DevCluster.json	A cluster configuration sample file that contains the settings for an unsecured, three-node, single-machine (or virtual machine) development cluster, including the information for each node in the cluster.
ClusterConfig.Unsecure.MultiMachine.json	A cluster configuration sample file that contains the settings for an unsecured, multi-machine (or virtual machine) cluster, including the information for each machine in the cluster.

FILE NAME	SHORT DESCRIPTION
ClusterConfig.Windows.DevCluster.json	A cluster configuration sample file that contains all the settings for a secure, three-node, single-machine (or virtual machine) development cluster, including the information for each node that is in the cluster. The cluster is secured by using Windows identities.
ClusterConfig.Windows.MultiMachine.json	A cluster configuration sample file that contains all the settings for a secure, multi-machine (or virtual machine) cluster using Windows security, including the information for each machine that is in the secure cluster. The cluster is secured by using Windows identities.
ClusterConfig.x509.DevCluster.json	A cluster configuration sample file that contains all the settings for a secure, three-node, single-machine (or virtual machine) development cluster, including the information for each node in the cluster. The cluster is secured using x509 certificates.
ClusterConfig.x509.MultiMachine.json	A cluster configuration sample file that contains all the settings for the secure, multi-machine (or virtual machine) cluster, including the information for each node in the secure cluster. The cluster is secured using x509 certificates.
EULA_ENU.txt	The license terms for the use of Microsoft Azure Service Fabric standalone Windows Server package. You can download a copy of the EULA now.
Readme.txt	A link to the release notes and basic installation instructions. It is a subset of the instructions in this document.
CreateServiceFabricCluster.ps1	A PowerShell script that creates the cluster using the settings in ClusterConfig.json.
RemoveServiceFabricCluster.ps1	A PowerShell script that removes a cluster using the settings in ClusterConfig.json.
ThirdPartyNotice.rtf	Notice of third-party software that is in the package.
AddNode.ps1	A PowerShell script for adding a node to an existing deployed cluster.
RemoveNode.ps1	A PowerShell script for removing a node from an existing deployed cluster.
CleanFabric.ps1	A PowerShell script for cleaning a standalone Service Fabric installation off the current machine. Previous MSI installations should be removed using their own associated uninstallers.
TestConfiguration.ps1	A PowerShell script for analyzing the infrastructure as specified in the Cluster.json.

Plan and prepare your cluster deployment

Perform the following steps before you create your cluster.

Step 1: Plan your cluster infrastructure

You are about to create a Service Fabric cluster on machines you own, so you can decide what kinds of failures you want the cluster to survive. For example, do you need separate power lines or Internet connections supplied to these machines? In addition, consider the physical security of these machines. Where are the machines located and who needs access to them? After you make these decisions, you can logically map the machines to the various fault domains (see Step 4). The infrastructure planning for production clusters is more involved than for test clusters.

Step 2: Prepare the machines to meet the prerequisites

Prerequisites for each machine that you want to add to the cluster:

- A minimum of 16 GB of RAM is recommended.
- A minimum of 40 of GB available disk space is recommended.
- A 4 core or greater CPU is recommended.
- Connectivity to a secure network or networks for all machines.
- Windows Server 2012 R2 or Windows Server 2012 (you need to have KB2858668 installed).
- .NET Framework 4.5.1 or higher, full install.
- Windows PowerShell 3.0.
- The RemoteRegistry service should be running on all the machines.

The cluster administrator deploying and configuring the cluster must have administrator privileges on each of the machines. You cannot install Service Fabric on a domain controller.

Step 3: Determine the initial cluster size

Each node in a standalone Service Fabric cluster has the Service Fabric runtime deployed and is a member of the cluster. In a typical production deployment, there is one node per OS instance (physical or virtual). The cluster size is determined by your business needs. However, you must have a minimum cluster size of three nodes (machines or virtual machines). For development purposes, you can have more than one node on a given machine. In a production environment, Service Fabric supports only one node per physical or virtual machine.

Step 4: Determine the number of fault domains and upgrade domains

A *fault domain* (FD) is a physical unit of failure and is directly related to the physical infrastructure in the data centers. A fault domain consists of hardware components (computers, switches, networks, and more) that share a single point of failure. Although there is no 1:1 mapping between fault domains and racks, loosely speaking, each rack can be considered a fault domain. When considering the nodes in your cluster, we strongly recommend that the nodes be distributed among at least three fault domains.

When you specify FDs in ClusterConfig.json, you can choose the name for each FD. Service Fabric supports hierarchical FDs, so you can reflect your infrastructure topology in them. For example, the following FDs are valid:

- "faultDomain": "fd:/Room1/Rack1/Machine1"
- "faultDomain": "fd:/FD1"
- "faultDomain": "fd:/Room1/Rack1/PDU1/M1"

An *upgrade domain* (UD) is a logical unit of nodes. During Service Fabric orchestrated upgrades (either an application upgrade or a cluster upgrade), all nodes in a UD are taken down to perform the upgrade while nodes in other UDs remain available to serve requests. The firmware upgrades you perform on your machines do not honor UDs, so you must do them one machine at a time.

The simplest way to think about these concepts is to consider FDs as the unit of unplanned failure and UDs as the unit of planned maintenance.

When you specify UDs in ClusterConfig.json, you can choose the name for each UD. For example, the following names are valid:

- "upgradeDomain": "UD0"
- "upgradeDomain": "UD1A"
- "upgradeDomain": "DomainRed"
- "upgradeDomain": "Blue"

For more detailed information on upgrade domains and fault domains, see Describing a Service Fabric cluster.

Step 5: Download the Service Fabric standalone package for Windows Server

Download the Service Fabric standalone package for Windows Server and unzip the package, either to a deployment machine that is not part of the cluster, or to one of the machines that will be a part of your cluster. You may rename the unzipped folder `Microsoft.Azure.ServiceFabric.WindowsServer`.

Create your cluster

After you have gone through the planning and preparation steps, you are ready to create your cluster.

Step 1: Modify cluster configuration

The cluster is described in ClusterConfig.json. For details on the sections in this file, see Configuration settings for standalone Windows cluster. Open one of the ClusterConfig.json files from the package you downloaded and modify the following settings:

CONFIGURATION SETTING	DESCRIPTION
NodeTypes	Node types allow you to separate your cluster nodes into various groups. A cluster must have at least one NodeType. All nodes in a group have the following common characteristics: **Name** - This is the node type name. **Endpoint Ports** - These are various named end points (ports) that are associated with this node type. You can use any port number that you wish, as long as they do not conflict with anything else in this manifest and are not already in use by any other application running on the machine/VM. **Placement Properties** - These describe properties for this node type that you use as placement constraints for the system services or your services. These properties are user-defined key/value pairs that provide extra meta data for a given node. Examples of node properties would be whether the node has a hard drive or graphics card, the number of spindles in its hard drive, cores, and other physical properties. **Capacities** - Node capacities define the name and amount of a particular resource that a particular node has available for consumption. For example, a node may define that it has capacity for a metric called "MemoryInMb" and that it has 2048 MB available by default. These capacities are used at runtime to ensure that services that require particular amounts of resources are placed on the nodes that have those resources available in the required amounts. **IsPrimary** - If you have more than one NodeType defined ensure that only one is set to primary with the value *true*, which is where the system services run. All other node types should be set to the value *false*

CONFIGURATION SETTING	DESCRIPTION
Nodes	These are the details for each of the nodes that are part of the cluster (node type, node name, IP address, fault domain, and upgrade domain of the node). The machines you want the cluster to be created on need to be listed here with their IP addresses. If you use the same IP address for all the nodes, then a one-box cluster is created, which you can use for testing purposes. Do not use One-box clusters for deploying production workloads.

Step 2: Run the TestConfiguration script

The TestConfiguration script tests your infrastructure as defined in cluster.json to make sure that the needed permissions are assigned, the machines are connected to each other, and other attributes are defined so that the deployment can succeed. It is basically a mini Best Practices Analyzer. We will continue to add more validations to this tool over time to make it more robust.

This script can be run on any machine that has administrator access to all the machines that are listed as nodes in the cluster configuration file. The machine that this script is run on does not have to be part of the cluster.

```
PS C:\temp\Microsoft.Azure.ServiceFabric.WindowsServer> .\TestConfiguration.ps1 -ClusterConfigFilePath
.\ClusterConfig.Unsecure.DevCluster.json
Trace folder already exists. Traces will be written to existing trace folder:
C:\temp\Microsoft.Azure.ServiceFabric.WindowsServer\DeploymentTraces
Running Best Practices Analyzer...
Best Practices Analyzer completed successfully.


LocalAdminPrivilege        : True
IsJsonValid                : True
IsCabValid                 : True
RequiredPortsOpen          : True
RemoteRegistryAvailable    : True
FirewallAvailable          : True
RpcCheckPassed             : True
NoConflictingInstallations : True
FabricInstallable          : True
Passed                     : True
```

Step 3: Run the create cluster script

After you have modified the cluster configuration in the JSON doc and added all the node information to it, run the cluster creation *CreateServiceFabricCluster.ps1* PowerShell script from the package folder and pass in the path to the JSON configuration file. When this is complete, accept the EULA.

This script can be run on any machine that has administrator access to all the machines that are listed as nodes in the cluster configuration file. The machine that this script is run on does not have to be part of the cluster.

```
#Create an unsecured local development cluster

.\CreateServiceFabricCluster.ps1 -ClusterConfigFilePath .\ClusterConfig.Unsecure.DevCluster.json -AcceptEULA
```

```
#Create an unsecured multi-machine cluster

.\CreateServiceFabricCluster.ps1 -ClusterConfigFilePath .\ClusterConfig.Unsecure.MultiMachine.json -AcceptEULA
```

> **NOTE**
> The deployment logs are available locally on the VM/machine that you ran the CreateServiceFabricCluster PowerShell on. They are in a subfolder called DeploymentTraces under the folder where you ran the PowerShell command. To see if Service Fabric was deployed correctly to a machine, you can find the installed files in the C:\ProgramData directory, and the FabricHost.exe and Fabric.exe processes can be seen running in Task Manager.

Step 4: Connect to the cluster

To connect to a secure cluster, see Service fabric connect to secure cluster.

To connect to an unsecure cluster, run the following PowerShell command:

```
Connect-ServiceFabricCluster -ConnectionEndpoint <*IPAddressofaMachine*>:<Client connection end point port>

Connect-ServiceFabricCluster -ConnectionEndpoint 192.13.123.2345:19000
```

Step 5: Bring up Service Fabric Explorer

Now you can connect to the cluster with Service Fabric Explorer either directly from one of the machines with http://localhost:19080/Explorer/index.html or remotely with http://<*IPAddressofaMachine*>:19080/Explorer/index.html.

Add and remove nodes

You can add or remove nodes to your standalone Service Fabric cluster as your business needs change. See Add or Remove nodes to a Service Fabric standalone cluster for detailed steps.

Remove a cluster

To remove a cluster, run the *RemoveServiceFabricCluster.ps1* PowerShell script from the package folder and pass in the path to the JSON configuration file. You can optionally specify a location for the log of the deletion.

This script can be run on any machine that has administrator access to all the machines that are listed as nodes in the cluster configuration file. The machine that this script is run on does not have to be part of the cluster.

```
.\RemoveServiceFabricCluster.ps1 -ClusterConfigFilePath .\ClusterConfig.Unsecure.MultiMachine.json
```

Telemetry data collected and how to opt out of it

As a default, the product collects telemetry on the Service Fabric usage to improve the product. The Best Practice Analyzer that runs as a part of the setup checks for connectivity to https://vortex.data.microsoft.com/collect/v1. If it is not reachable, the setup fails unless you opt out of telemetry.

1. The telemetry pipeline tries to upload the following data to https://vortex.data.microsoft.com/collect/v1 once every day. It is a best-effort upload and has no impact on the cluster functionality. The telemetry is only sent from the node that runs the failover manager primary. No other nodes send out telemetry.
2. The telemetry consists of the following:

- Number of services
- Number of ServiceTypes
- Number of Applications
- Number of ApplicationUpgrades
- Number of FailoverUnits

- Number of InBuildFailoverUnits
- Number of UnhealthyFailoverUnits
- Number of Replicas
- Number of InBuildReplicas
- Number of StandByReplicas
- Number of OfflineReplicas
- CommonQueueLength
- QueryQueueLength
- FailoverUnitQueueLength
- CommitQueueLength
- Number of Nodes
- IsContextComplete: True/False
- ClusterId: This is a GUID randomly generated for each cluster
- ServiceFabricVersion
- IP address of the virtual machine or machine from which the telemetry is uploaded

To disable telemetry, add the following to *properties* in your cluster config: *enableTelemetry: false*.

Preview features included in this package

None.

> **NOTE**
>
> With the new GA version of the standalone cluster for Windows Server (version 5.3.204.x), you can upgrade your cluster to future releases, manually or automatically. Because this feature is not available on the preview versions, you will need to create a cluster by using the GA version and migrate your data and applications from the preview cluster. Stay tuned for more details on this feature.

Next steps

- Configuration settings for standalone Windows cluster
- Add or remove nodes to a standalone Service Fabric cluster
- Create a standalone Service Fabric cluster with Azure VMs running Windows
- Secure a standalone cluster on Windows using Windows security
- Secure a standalone cluster on Windows using X509 certificates

Add or remove nodes to a standalone Service Fabric cluster running on Windows Server

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Dhanashri Kshirsagar • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Chacko Daniel

After you have created your standalone Service Fabric cluster on Windows Server machines, your business needs may change so that you might need to add or remove multiple nodes to your cluster. This article provides detailed steps to achieve this goal.

Add nodes to your cluster

1. Prepare the VM/machine you want to add to your cluster by following the steps mentioned in the Prepare the machines to meet the prerequisites for cluster deployment section.
2. Plan which fault domain and upgrade domain you are going to add this VM/machine to.
3. Remote desktop (RDP) into the VM/machine that you want to add to the cluster.
4. Copy or download the standalone package for Service Fabric for Windows Server to this VM/machine and unzip the package.
5. Run Powershell as an administrator, and navigate to the location of the unzipped package.
6. Run *AddNode.ps1* Powershell with the parameters describing the new node to add. The example below adds a new node called VM5, with type NodeType0, IP address 182.17.34.52 into UD1 and FD1. The *ExistingClusterConnectionEndPoint* is a connection endpoint for a node already in the existing cluster. For this endpoint, you can choose the IP address of *any* node in the cluster.

```
.\AddNode.ps1 -NodeName VM5 -NodeType NodeType0 -NodeIPAddressorFQDN 182.17.34.52 -
ExistingClientConnectionEndpoint 182.17.34.50:19000 -UpgradeDomain UD1 -FaultDomain FD1 -AcceptEULA
```

Remove nodes from your cluster

1. Depending on the Reliablity level chosen for the cluster, you cannot remove the first n (3/5/7/9) nodes of the primary node type
2. Running RemoveNode command on a dev cluster is not supported.
3. Remote desktop (RDP) into the VM/machine that you want to remove from the cluster.
4. Copy or download the standalone package for Service Fabric for Windows Server and unzip the package to this VM/machine.
5. Run Powershell as an administrator, and navigate to the location of the unzipped package.
6. Run *RemoveNode.ps1* in PowerShell. The example below removes the current node from the cluster. The *ExistingClientConnectionEndpoint* is a client connection endpoint for any node that will remain in the cluster. Choose the IP address and the endpoint port of *any* **other node** in the cluster. This **other node** will in turn update the cluster configuration for the removed node.

```
.\RemoveNode.ps1 -ExistingClientConnectionEndpoint 182.17.34.50:19000
```

> **NOTE**
> Some nodes may not be removed due to system services dependencies. These nodes are primary nodes and can be identified by querying the cluster manifest using `Get-ServiceFabricClusterManifest` and finding node entries marked with `IsSeedNode="true"`.

Even after removing a node, if it shows up as being down in queries and SFX, please note that this is a known defect. It will be fixed in an upcoming release.

Next steps

- Configuration settings for standalone Windows cluster
- Secure a standalone cluster on Windows using Windows security
- Secure a standalone cluster on Windows using X509 certificates
- Create a standalone Service Fabric cluster with Azure VMs running Windows

Upgrade your standalone Service Fabric cluster on Windows Server

11/15/2016 • 5 min to read • Edit on GitHub

Contributors

Chacko Daniel • Rakshita Tandon • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

For any modern system, designing for upgradability is key to achieving long-term success of your product. A Service Fabric cluster is a resource that you own. This article describes how you can make sure that the cluster always runs supported versions of the service fabric code and configurations.

Controlling the fabric version that runs on your Cluster

You can set your cluster to download the service fabric updates, when Microsoft releases a new version or choose to select a supported fabric version you want your cluster to be on.

You do this by setting the "fabricClusterAutoupgradeEnabled" cluster configuration to true or false.

> **NOTE**
>
> Make sure to keep your cluster always running a supported Service Fabric version. As and when we announce the release of a new version of service fabric, the previous version is marked for end of support after a minimum of 60 days from that date. The new releases are announced on the service fabric team blog. The new release is available to choose then.

You can upgrade your cluster to the new version only if you are using a production-style node configuration, where each Service Fabric node is allocated on a separate physical or virtual machine. If you have a development cluster, where there are more than one service fabric nodes on a single physical or virtual machine, you must tear down your cluster and recreate it with the new version.

There are two distinct workflows for upgrading your cluster to the latest or a supported service fabric version. One for clusters that have connectivity to download the latest version automatically and the second one for clusters that are no connectivity to download the latest Service Fabric version.

Upgrade the clusters with connectivity to download the latest code and configuration

Use these steps to upgrade your cluster to a supported version, if your cluster nodes have internet connectivity to http://download.microsoft.com

For clusters that have connectivity to http://download.microsoft.com, we periodically check for the availability of new service fabric versions.

When a new service fabric version is available, the package is downloaded locally to the cluster and provisioned for upgrade. Additionally to inform the customer of this new version, the system places an explicit cluster health warning similar to the following:

“The current cluster version [version#] support ends [Date].", Once the cluster is running the latest version, the warning goes away.

Cluster Upgrade workflow.

Once you see the cluster health warning, you need to do the following:

1. Connect to the cluster from any machine that has administrator access to all the machines that are listed as

nodes in the cluster. The machine that this script is run on does not have to be part of the cluster

```
###### connect to the secure cluster using certs
$ClusterName= "mysecurecluster.something.com:19000"
$CertThumbprint= "70EF5E22ADB649799DA3C8B6A6BF7FG2D630F8F3"
Connect-serviceFabricCluster -ConnectionEndpoint $ClusterName -KeepAliveIntervalInSec 10 `
    -X509Credential `
    -ServerCertThumbprint $CertThumbprint  `
    -FindType FindByThumbprint `
    -FindValue $CertThumbprint `
    -StoreLocation CurrentUser `
    -StoreName My
```

2. Get the list of service fabric versions that you can upgrade to

```
###### Get the list of available service fabric versions
Get-ServiceFabricRegisteredClusterCodeVersion
```

you should get an output similar to this:

```
PS C:\> Get-ServiceFabricRegisteredClusterCodeVersion

CodeVersion
-----------
5.3.204.9494
5.3.301.9590
```

3. Kick off a cluster upgrade to one of the versions that is available using the Start-ServiceFabricClusterUpgrade PowerShell cmd

```
 Start-ServiceFabricClusterUpgrade -Code -CodePackageVersion <codeversion#> -Monitored -FailureAction
Rollback

 ###### Here is a filled out example

 Start-ServiceFabricClusterUpgrade -Code -CodePackageVersion 5.3.301.9590 -Monitored -FailureAction
Rollback
```

You can monitor the progress of the upgrade on Service fabric explorer or by running the following power shell command

```
 Get-ServiceFabricClusterUpgrade
```

If the cluster health policies are not met, the upgrade is rolled back. You can specify custom health policies at the time for the Start-ServiceFabricClusterUpgrade command refer to this document for details.

Once you have fixed the issues that resulted in the rollback, you need to initiate the upgrade again, by following the same steps as before.

Upgrade the clusters with no connectivity to download the latest code and configuration

Use these steps to upgrade your cluster to a supported version, if your cluster nodes **do not have** internet connectivity to http://download.microsoft.com

> **NOTE**
>
> If you are running a cluster that is not internet connected, you will have to monitor the service fabric team blog to get notified of a new release. The system **does not** place any cluster health warning to alert you of it.

1. Modify your cluster configuration to set the following property to false.

```
"fabricClusterAutoupgradeEnabled": false,
```

and kick off a configuration upgrade. Refer to Start-ServiceFabricClusterConfigurationUpgrade PS cmd for usage details. Make sure to update the 'clusterConfigurationVersion' in your JSON before you kick off the configuration upgrade.

```
Start-ServiceFabricClusterConfigurationUpgrade -ClusterConfigPath <Path to Configuration File>
```

Cluster Upgrade workflow.

1. Download the latest version of the package from Create service fabric cluster for windows server document
2. Connect to the cluster from any machine that has administrator access to all the machines that are listed as nodes in the cluster. The machine that this script is run on does not have to be part of the cluster

```
###### connect to the cluster
$ClusterName= "mysecurecluster.something.com:19000"
$CertThumbprint= "70EF5E22ADB649799DA3C8B6A6BF7FG2D630F8F3"
Connect-serviceFabricCluster -ConnectionEndpoint $ClusterName -KeepAliveIntervalInSec 10 `
    -X509Credential `
    -ServerCertThumbprint $CertThumbprint  `
    -FindType FindByThumbprint `
    -FindValue $CertThumbprint `
    -StoreLocation CurrentUser `
    -StoreName My
```

3. Copy the downloaded package into the cluster image store.

```
###### Get the list of available service fabric versions
 Copy-ServiceFabricClusterPackage -Code -CodePackagePath <name of the .cab file including the path to it>
-ImageStoreConnectionString "fabric:ImageStore"

###### Here is a filled out example
 Copy-ServiceFabricClusterPackage -Code -CodePackagePath .\MicrosoftAzureServiceFabric.5.3.301.9590.cab -
ImageStoreConnectionString "fabric:ImageStore"
```

4. Register the copied package

```
 ###### Get the list of available service fabric versions
 Register-ServiceFabricClusterPackage -Code -CodePackagePath <name of the .cab file>

 ###### Here is a filled out example
 Register-ServiceFabricClusterPackage -Code -CodePackagePath MicrosoftAzureServiceFabric.5.3.301.9590.cab
```

5. Kick off a cluster upgrade to one of the versions that is available.

```
 Start-ServiceFabricClusterUpgrade -Code -CodePackageVersion <codeversion#> -Monitored -FailureAction
Rollback

 ###### Here is a filled out example
 Start-ServiceFabricClusterUpgrade -Code -CodePackageVersion 5.3.301.9590 -Monitored -FailureAction
Rollback
```

You can monitor the progress of the upgrade on Service fabric explorer or by running the following power

shell command

```
Get-ServiceFabricClusterUpgrade
```

If the cluster health policies are not met, the upgrade is rolled back. You can specify custom health policies at the time for the start-serviceFabricClusterUpgrade command refer to this document for details.

Once you have fixed the issues that resulted in the rollback, you need to initiate the upgrade again, by following the same steps as before.

Next steps

- Learn how to customize some of the service fabric cluster fabric settings
- Learn how to scale your cluster in and out
- Learn about application upgrades

Secure a standalone cluster on Windows using X.509 certificates

11/15/2016 • 8 min to read • Edit on GitHub

Contributors

Dhanashri Kshirsagar • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Rakshita Tandon • Ryan Wike • Mark Fussell

This article describes how to secure the communication between the various nodes of your standalone Windows cluster, as well as how to authenticate clients connecting to this cluster, using X.509 certificates. This ensures that only authorized users can access the cluster, the deployed applications and perform management tasks. Certificate security should be enabled on the cluster when the cluster is created.

For more information on cluster security such as node-to-node security, client-to-node security, and role-based access control, see Cluster security scenarios.

Which certificates will you need?

To start with, download the standalone cluster package to one of the nodes in your cluster. In the downloaded package, you will find a **ClusterConfig.X509.MultiMachine.json** file. Open the file and review the section for **security** under the **properties** section:

```
"security": {
    "metadata": "The Credential type X509 indicates this is cluster is secured using X509 Certificates. The
thumbprint format is - d5 ec 42 3b 79 cb e5 07 fd 83 59 3c 56 b9 d5 31 24 25 42 64.",
    "ClusterCredentialType": "X509",
    "ServerCredentialType": "X509",
    "CertificateInformation": {
        "ClusterCertificate": {
            "Thumbprint": "[Thumbprint]",
            "ThumbprintSecondary": "[Thumbprint]",
            "X509StoreName": "My"
        },
        "ServerCertificate": {
            "Thumbprint": "[Thumbprint]",
            "ThumbprintSecondary": "[Thumbprint]",
            "X509StoreName": "My"
        },
        "ClientCertificateThumbprints": [{
            "CertificateThumbprint": "[Thumbprint]",
            "IsAdmin": false
        }, {
            "CertificateThumbprint": "[Thumbprint]",
            "IsAdmin": true
        }],
        "ClientCertificateCommonNames": [{
            "CertificateCommonName": "[CertificateCommonName]",
            "CertificateIssuerThumbprint" : "[Thumbprint]",
            "IsAdmin": true
        }]
        "HttpApplicationGatewayCertificate":{
            "Thumbprint": "[Thumbprint]",
            "X509StoreName": "My"
        }
    }
}
```

This section describes the certificates that you need for securing your standalone Windows cluster. To enable certificate-based security set the values of **ClusterCredentialType** and **ServerCredentialType** to *X509*.

> **NOTE**
> A thumbprint is the primary identity of a certificate. Read How to retrieve thumbprint of a certificate to find out the thumbprint of the certificates that you create.

The following table lists the certificates that you will need on your cluster setup:

CERTIFICATEINFORMATION SETTING	DESCRIPTION
ClusterCertificate	This certificate is required to secure the communication between the nodes on a cluster. You can use two different certificates, a primary and a secondary for upgrade. Set the thumbprint of the primary certificate in the **Thumbprint** section and that of the secondary in the **ThumbprintSecondary** variables.
ServerCertificate	This certificate is presented to the client when it tries to connect to this cluster. For convenience, you can choose to use the same certificate for *ClusterCertificate* and *ServerCertificate*. You can use two different server certificates, a primary and a secondary for upgrade. Set the thumbprint of the primary certificate in the **Thumbprint** section and that of the secondary in the **ThumbprintSecondary** variables.
ClientCertificateThumbprints	This is a set of certificates that you want to install on the authenticated clients. You can have a number of different client certificates installed on the machines that you want to allow access to the cluster. Set the thumbprint of each certificate in the **CertificateThumbprint** variable. If you set the **IsAdmin** to *true*, then the client with this certificate installed on it can do administrator management activities on the cluster. If the **IsAdmin** is *false*, the client with this certificate can only perform the actions allowed for user access rights, typically read-only. For more information on roles read Role based access control (RBAC)
ClientCertificateCommonNames	Set the common name of the first client certificate for the **CertificateCommonName**. The **CertificateIssuerThumbprint** is the thumbprint for the issuer of this certificate. Read Working with certificates to know more about common names and the issuer.
HttpApplicationGatewayCertificate	This is an optional certificate that can be specified if you want to secure your Http Application Gateway. Make sure reverseProxyEndpointPort is set in nodeTypes if you are using this certificate.

Here is example cluster configuration where the Cluster, Server, and Client certificates have been provided.

```
{
   "name": "SampleCluster",
   "clusterConfigurationVersion": "1.0.0",
   "apiVersion": "2016-09-26",
   "nodes": [{
       "nodeName": "vm0",
       "metadata": "Replace the localhost below with valid IP address or FQDN",
       "iPAddress": "10.7.0.5",
```

```
        "nodeTypeRef": "NodeType0",
        "faultDomain": "fd:/dc1/r0",
        "upgradeDomain": "UD0"
    }, {
      "nodeName": "vm1",
              "metadata": "Replace the localhost with valid IP address or FQDN",
        "iPAddress": "10.7.0.4",
        "nodeTypeRef": "NodeType0",
        "faultDomain": "fd:/dc1/r1",
        "upgradeDomain": "UD1"
    }, {
        "nodeName": "vm2",
      "iPAddress": "10.7.0.6",
              "metadata": "Replace the localhost with valid IP address or FQDN",
        "nodeTypeRef": "NodeType0",
        "faultDomain": "fd:/dc1/r2",
        "upgradeDomain": "UD2"
    }],
    "properties": {
        "diagnosticsStore": {
        "metadata":  "Please replace the diagnostics store with an actual file share accessible from all cluster
machines.",
        "dataDeletionAgeInDays": "7",
        "storeType": "FileShare",
        "IsEncrypted": "false",
        "connectionstring": "c:\\ProgramData\\SF\\DiagnosticsStore"
        }
        "security": {
            "metadata": "The Credential type X509 indicates this is cluster is secured using X509 Certificates.
The thumbprint format is - d5 ec 42 3b 79 cb e5 07 fd 83 59 3c 56 b9 d5 31 24 25 42 64.",
            "ClusterCredentialType": "X509",
            "ServerCredentialType": "X509",
            "CertificateInformation": {
                "ClusterCertificate": {
                    "Thumbprint": "a8 13 67 58 f4 ab 89 62 af 2b f3 f2 79 21 be 1d f6 7f 43 26",
                    "X509StoreName": "My"
                },
                "ServerCertificate": {
                    "Thumbprint": "a8 13 67 58 f4 ab 89 62 af 2b f3 f2 79 21 be 1d f6 7f 43 26",
                    "X509StoreName": "My"
                },
                "ClientCertificateThumbprints": [{
                    "CertificateThumbprint": "c4 c18 8e aa a8 58 77 98 65 f8 61 4a 0d da 4c 13 c5 a1 37 6e",
                    "IsAdmin": false
                }, {
                    "CertificateThumbprint": "71 de 04 46 7c 9e d0 54 4d 02 10 98 bc d4 4c 71 e1 83 41 4e",
                    "IsAdmin": true
                }]
            }
        },
        "reliabilityLevel": "Bronze",
        "nodeTypes": [{
            "name": "NodeType0",
            "clientConnectionEndpointPort": "19000",
            "clusterConnectionEndpoint": "19001",
            "httpGatewayEndpointPort": "19080",
            "applicationPorts": {
                "startPort": "20001",
                "endPort": "20031"
            },
            "ephemeralPorts": {
                "startPort": "20032",
                "endPort": "20062"
            },
            "isPrimary": true
        }
         ],
        "fabricSettings": [{
            "name": "Setup",
```

```
            "parameters": [{
                "name": "FabricDataRoot",
                "value": "C:\\ProgramData\\SF"
            }, {
                "name": "FabricLogRoot",
                "value": "C:\\ProgramData\\SF\\Log"
            }]
        }]
    }
}
```

Acquire the X.509 certificates

To secure communication within the cluster, you will first need to obtain X.509 certificates for your cluster nodes. Additionally, to limit connection to this cluster to authorized machines/users, you will need to obtain and install certificates for the client machines.

For clusters that are running production workloads, you should use a Certificate Authority (CA) signed X.509 certificate to secure the cluster. For details on obtaining these certificates, go to How to: Obtain a Certificate.

For clusters that you use for test purposes, you can choose to use a self-signed certificate.

Optional: Create a self-signed certificate

One way to create a self-signed cert that can be secured correctly is to use the *CertSetup.ps1* script in the Service Fabric SDK folder in the directory *C:\Program Files\Microsoft SDKs\Service Fabric\ClusterSetup\Secure*. Edit this file and use this to create a certificate with a suitable name.

Now export the certificate to a PFX file with a protected password. First you need to get the thumbprint of the certificate. Run the certmgr.exe application. Navigate to the **Local Computer\Personal** folder and find the certificate you just created. Double-click the certificate to open it, select the *Details* tab and scroll down to the *Thumbprint* field. Copy the thumbprint value into the PowerShell command below, removing the spaces. Change the *$pswd* value to a suitability secure password to protect it and run the PowerShell:

```
$pswd = ConvertTo-SecureString -String "1234" -Force –AsPlainText
Get-ChildItem -Path cert:\localMachine\my\<Thumbprint> | Export-PfxCertificate -FilePath C:\mypfx.pfx -Password
$pswd
```

To see the details of a certificate installed on the machine you can run the following PowerShell command:

```
$cert = Get-Item Cert:\LocalMachine\My\<Thumbprint>
Write-Host $cert.ToString($true)
```

Alternatively, if you have an Azure subscription, follow the section Add certificates to Key Vault.

Install the certificates

Once you have certificate(s), you can install them on the cluster nodes. Your nodes need to have the latest Windows PowerShell 3.x installed on them. You will need to repeat these steps on each node, for both Cluster and Server certificates and any secondary certificates.

1. Copy the .pfx file(s) to the node.
2. Open a PowerShell window as an administrator and enter the following commands. Replace the *$pswd* with the password that you used to create this certificate. Replace the *$PfxFilePath* with the full path of the .pfx copied to this node.

```
 $pswd = "1234"
 $PfcFilePath ="C:\mypfx.pfx"
 Import-PfxCertificate -Exportable -CertStoreLocation Cert:\LocalMachine\My -FilePath $PfxFilePath -
Password (ConvertTo-SecureString -String $pswd -AsPlainText -Force)
```

3. Next you need to set the access control on this certificate so that the Service Fabric process, which runs under the Network Service account, can use it by running the following script. Provide the thumbprint of the certificate and "NETWORK SERVICE" for the service account. You can check that the ACLs on the certificate are correct using the certmgr.exe tool and looking at the Manage Private Keys on the cert.

```
 param
 (
 [Parameter(Position=1, Mandatory=$true)]
 [ValidateNotNullOrEmpty()]
 [string]$pfxThumbPrint,

 [Parameter(Position=2, Mandatory=$true)]
 [ValidateNotNullOrEmpty()]
 [string]$serviceAccount
 )

 $cert = Get-ChildItem -Path cert:\LocalMachine\My | Where-Object -FilterScript { $PSItem.ThumbPrint -eq
$pfxThumbPrint; }

 # Specify the user, the permissions and the permission type
 $permission = "$($serviceAccount)","FullControl","Allow"
 $accessRule = New-Object -TypeName System.Security.AccessControl.FileSystemAccessRule -ArgumentList
$permission

 # Location of the machine related keys
 $keyPath = $Join-Path -Path $env:ProgramData -ChildPath "\Microsoft\Crypto\RSA\MachineKeys"
 $keyName = $cert.PrivateKey.CspKeyContainerInfo.UniqueKeyContainerName
 $keyFullPath = Join-Path -Path $keyPath -ChildPath $keyName

 # Get the current acl of the private key
 $acl = (Get-Item $keyFullPath).GetAccessControl('Access')

 # Add the new ace to the acl of the private key
 $acl.SetAccessRule($accessRule)

 # Write back the new acl
 Set-Acl -Path $keyFullPath -AclObject $acl -ErrorAction Stop

 # Observe the access rights currently assigned to this certificate.
 get-acl $keyFullPath| fl
```

4. Repeat the steps above for each server certificate. You can also use these steps to install the client certificates on the machines that you want to allow access to the cluster.

Create the secure cluster

After configuring the **security** section of the **ClusterConfig.X509.MultiMachine.json** file, you can proceed to Create your cluster section to configure the nodes and create the standalone cluster. Remember to use the **ClusterConfig.X509.MultiMachine.json** file while creating the cluster. For example, your command might look like the following:

```
.\CreateServiceFabricCluster.ps1 -ClusterConfigFilePath .\ClusterConfig.X509.MultiMachine.json -
MicrosoftServiceFabricCabFilePath .\MicrosoftAzureServiceFabric.cab -AcceptEULA $true
```

Once you have the secure standalone Windows cluster successfully running, and have setup the authenticated clients to connect to it, follow the section Connect to a secure cluster using PowerShell to connect to it. For

example:

```
Connect-ServiceFabricCluster -ConnectionEndpoint 10.7.0.4:19000 -KeepAliveIntervalInSec 10 -X509Credential -
ServerCertThumbprint 057b9544a6f2733e0c8d3a60013a58948213f551 -FindType FindByThumbprint -FindValue
057b9544a6f2733e0c8d3a60013a58948213f551 -StoreLocation CurrentUser -StoreName My
```

If you are logged on to one of the machines in the cluster, since this already has the certificate installed locally you can simply run the Powershell command to connect to the cluster and show a list of nodes:

```
Connect-ServiceFabricCluster
Get-ServiceFabricNode
```

To remove the cluster call the following command:

```
.\RemoveServiceFabricCluster.ps1 -ClusterConfigFilePath .\ClusterConfig.X509.MultiMachine.json   -
MicrosoftServiceFabricCabFilePath .\MicrosoftAzureServiceFabric.cab
```

> **NOTE**
>
> Incorrect certificate configuration may prevent the cluster from coming up during deployment. To self-diagnose security issues, please look in event viewer group *Applications and Services Logs* > *Microsoft-Service Fabric*.

Role-based access control for Service Fabric clients

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Mani Ramaswamy • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Rebecca Agiewich

Azure Service Fabric supports two different access control types for clients that are connected to a Service Fabric cluster: administrator and user. Access control allows the cluster administrator to limit access to certain cluster operations for different groups of users, making the cluster more secure.

Administrators have full access to management capabilities (including read/write capabilities). By default, **users** only have read access to management capabilities (for example, query capabilities), and the ability to resolve applications and services.

You specify the two client roles (administrator and client) at the time of cluster creation by providing separate certificates for each. See Service Fabric cluster security for details on setting up a secure Service Fabric cluster.

Default access control settings

The administrator access control type has full access to all the FabricClient APIs. It can perform any reads and writes on the Service Fabric cluster, including the following operations:

Application and service operations

- **CreateService**: service creation
- **CreateServiceFromTemplate**: service creation from template
- **UpdateService**: service updates
- **DeleteService**: service deletion
- **ProvisionApplicationType**: application type provisioning
- **CreateApplication**: application creation
- **DeleteApplication**: application deletion
- **UpgradeApplication**: starting or interrupting application upgrades
- **UnprovisionApplicationType**: application type unprovisioning
- **MoveNextUpgradeDomain**: resuming application upgrades with an explicit update domain
- **ReportUpgradeHealth**: resuming application upgrades with the current upgrade progress
- **ReportHealth**: reporting health
- **PredeployPackageToNode**: predeployment API
- **CodePackageControl**: restarting code packages
- **RecoverPartition**: recovering a partition
- **RecoverPartitions**: recovering partitions
- **RecoverServicePartitions**: recovering service partitions
- **RecoverSystemPartitions**: recovering system service partitions

Cluster operations

- **ProvisionFabric**: MSI and/or cluster manifest provisioning
- **UpgradeFabric**: starting cluster upgrades
- **UnprovisionFabric**: MSI and/or cluster manifest unprovisioning
- **MoveNextFabricUpgradeDomain**: resuming cluster upgrades with an explicit update domain

- **ReportFabricUpgradeHealth**: resuming cluster upgrades with the current upgrade progress
- **StartInfrastructureTask**: starting infrastructure tasks
- **FinishInfrastructureTask**: finishing infrastructure tasks
- **InvokeInfrastructureCommand**: infrastructure task management commands
- **ActivateNode**: activating a node
- **DeactivateNode**: deactivating a node
- **DeactivateNodesBatch**: deactivating multiple nodes
- **RemoveNodeDeactivations**: reverting deactivation on multiple nodes
- **GetNodeDeactivationStatus**: checking deactivation status
- **NodeStateRemoved**: reporting node state removed
- **ReportFault**: reporting fault
- **FileContent**: image store client file transfer (external to cluster)
- **FileDownload**: image store client file download initiation (external to cluster)
- **InternalList**: image store client file list operation (internal)
- **Delete**: image store client delete operation
- **Upload**: image store client upload operation
- **NodeControl**: starting, stopping, and restarting nodes
- **MoveReplicaControl**: moving replicas from one node to another

Miscellaneous operations

- **Ping**: client pings
- **Query**: all queries allowed
- **NameExists**: naming URI existence checks

The user access control type is, by default, limited to the following operations:

- **EnumerateSubnames**: naming URI enumeration
- **EnumerateProperties**: naming property enumeration
- **PropertyReadBatch**: naming property read operations
- **GetServiceDescription**: long-poll service notifications and reading service descriptions
- **ResolveService**: complaint-based service resolution
- **ResolveNameOwner**: resolving naming URI owner
- **ResolvePartition**: resolving system services
- **ServiceNotifications**: event-based service notifications
- **GetUpgradeStatus**: polling application upgrade status
- **GetFabricUpgradeStatus**: polling cluster upgrade status
- **InvokeInfrastructureQuery**: querying infrastructure tasks
- **List**: image store client file list operation
- **ResetPartitionLoad**: resetting load for a failover unit
- **ToggleVerboseServicePlacementHealthReporting**: toggling verbose service placement health reporting

The admin access control also has access to the preceding operations.

Changing default settings for client roles

In the cluster manifest file, you can provide admin capabilities to the client if needed. You can change the defaults by going to the **Fabric Settings** option during cluster creation, and providing the preceding settings in the **name**, **admin**, **user**, and **value** fields.

Next steps

Service Fabric cluster security

Service Fabric cluster creation

Configuration settings for standalone Windows cluster

11/15/2016 • 7 min to read • Edit on GitHub

Contributors

Dhanashri Kshirsagar • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Rakshita Tandon • alexandair

This article describes how to configure a standalone Service Fabric cluster using the ***ClusterConfig.JSON*** file. You can use this file to specify information such as the Service Fabric nodes and their IP addresses, different types of nodes on the cluster, the security configurations as well as the network topology in terms of fault/upgrade domains, for your standalone cluster.

When you download the standalone Service Fabric package, a few samples of the ClusterConfig.JSON file are downloaded to your work machine. The samples having *DevCluster* in their names will help you create a cluster with all three nodes on the same machine, like logical nodes. Out of these, at least one node must be marked as a primary node. This cluster is useful for a development or test environment and is not supported as a production cluster. The samples having *MultiMachine* in their names, will help you create a production quality cluster, with each node on a separate machine. The number of primary nodes for these cluster will be based on the reliability level.

1. *ClusterConfig.Unsecure.DevCluster.JSON* and *ClusterConfig.Unsecure.MultiMachine.JSON* show how to create an unsecured test or production cluster respectively.
2. *ClusterConfig.Windows.DevCluster.JSON* and *ClusterConfig.Windows.MultiMachine.JSON* show how to create test or production cluster, secured using Windows security.
3. *ClusterConfig.X509.DevCluster.JSON* and *ClusterConfig.X509.MultiMachine.JSON* show how to create test or production cluster, secured using X509 certificate-based security.

Now we will examine the various sections of a ***ClusterConfig.JSON*** file as below.

General cluster configurations

This covers the broad cluster specific configurations, as shown in the JSON snippet below.

```
"name": "SampleCluster",
"clusterConfigurationVersion": "1.0.0",
"apiVersion": "2015-01-01-alpha",
```

You can give any friendly name to your Service Fabric cluster by assigning it to the **name** variable. The **clusterConfigurationVersion** is the version number of your cluster; you should increase it every time you upgrade your Service Fabric cluster. You should however leave the **apiVersion** to the default value.

Nodes on the cluster

You can configure the nodes on your Service Fabric cluster by using the **nodes** section, as the following snippet shows.

```
"nodes": [{
    "nodeName": "vm0",
    "iPAddress": "localhost",
    "nodeTypeRef": "NodeType0",
    "faultDomain": "fd:/dc1/r0",
    "upgradeDomain": "UD0"
}, {
    "nodeName": "vm1",
    "iPAddress": "localhost",
    "nodeTypeRef": "NodeType1",
    "faultDomain": "fd:/dc1/r1",
    "upgradeDomain": "UD1"
}, {
    "nodeName": "vm2",
    "iPAddress": "localhost",
    "nodeTypeRef": "NodeType2",
    "faultDomain": "fd:/dc1/r2",
    "upgradeDomain": "UD2"
}],
```

A Service Fabric cluster must contain at least 3 nodes. You can add more nodes to this section as per your setup. The following table explains the configuration settings for each node.

NODE CONFIGURATION	DESCRIPTION
nodeName	You can give any friendly name to the node.
iPAddress	Find out the IP address of your node by opening a command window and typing `ipconfig`. Note the IPV4 address and assign it to the **iPAddress** variable.
nodeTypeRef	Each node can be assigned a different node type. The node types are defined in the section below.
faultDomain	Fault domains enable cluster administrators to define the physical nodes that might fail at the same time due to shared physical dependencies.
upgradeDomain	Upgrade domains describe sets of nodes that are shut down for Service Fabric upgrades at about the same time. You can choose which nodes to assign to which Upgrade domains, as they are not limited by any physical requirements.

Cluster properties

The **properties** section in the ClusterConfig.JSON is used to configure the cluster as follows.

Reliability

The **reliabilityLevel** section defines the number of copies of the system services that can run on the primary nodes of the cluster. This increases the reliability of these services and hence the cluster. You can set this variable to either *Bronze*, *Silver*, *Gold* or *Platinum* for 3, 5, 7 or 9 copies of these services respectively. See an example below.

```
"reliabilityLevel": "Bronze",
```

Note that since a primary node runs a single copy of the system services, you would need a minimum of 3 primary nodes for *Bronze*, 5 for *Silver*, 7 for *Gold* and 9 for *Platinum* reliability levels.

Diagnostics

The **diagnosticsStore** section allows you to configure parameters to enable diagnostics and troubleshooting node or cluster failures, as shown in the following snippet.

```
"diagnosticsStore": {
    "metadata":  "Please replace the diagnostics store with an actual file share accessible from all cluster
machines.",
    "dataDeletionAgeInDays": "7",
    "storeType": "FileShare",
    "IsEncrypted": "false",
    "connectionstring": "c:\\ProgramData\\SF\\DiagnosticsStore"
}
```

The **metadata** is a description of your cluster diagnostics and can be set as per your setup. These variables help in collecting ETW trace logs, crash dumps as well as performance counters. Read Tracelog and ETW Tracing for more information on ETW trace logs. All logs including Crash dumps and performance counters can be directed to the **connectionString** folder on your machine. You can also use *AzureStorage* for storing diagnostics. See below for a sample snippet.

```
"diagnosticsStore": {
    "metadata":  "Please replace the diagnostics store with an actual file share accessible from all cluster
machines.",
    "dataDeletionAgeInDays": "7",
    "storeType": "AzureStorage",
    "IsEncrypted": "false",
    "connectionstring": "xstore:DefaultEndpointsProtocol=https;AccountName=[AzureAccountName];AccountKey=
[AzureAccountKey]"
}
```

Security

The **security** section is necessary for a secure standalone Service Fabric cluster. The following snippet shows a part of this section.

```
"security": {
    "metadata": "This cluster is secured using X509 certificates.",
    "ClusterCredentialType": "X509",
    "ServerCredentialType": "X509",
    . . .
}
```

The **metadata** is a description of your secure cluster and can be set as per your setup. The **ClusterCredentialType** and **ServerCredentialType** determine the type of security that the cluster and the nodes will implement. They can be set to either *X509* for a certificate-based security, or *Windows* for an Azure Active Directory-based security. The rest of the **security** section will be based on the type of the security. Read Certificates-based security in a standalone cluster or Windows security in a standalone cluster for information on how to fill out the rest of the **security** section.

Node Types

The **nodeTypes** section describes the type of the nodes that your cluster has. At least one node type must be specified for a cluster, as shown in the snippet below.

```
"nodeTypes": [{
    "name": "NodeType0",
    "clientConnectionEndpointPort": "19000",
    "clusterConnectionEndpointPort": "19001",
    "leaseDriverEndpointPort": "19002"
    "serviceConnectionEndpointPort": "19003",
    "httpGatewayEndpointPort": "19080",
    "applicationPorts": {
        "startPort": "20575",
        "endPort": "20605"
    },
    "ephemeralPorts": {
        "startPort": "20606",
        "endPort": "20861"
    },
    "isPrimary": true
}]
```

The **name** is the friendly name for this particular node type. To create a node of this node type, assign its friendly name to the **nodeTypeRef** variable for that node, as mentioned above. For each node type, define the connection endpoints that will be used. You can choose any port number for these connection endpoints, as long as they do not conflict with any other endpoints in this cluster. In a multi-node cluster, there will be one or more primary nodes (i.e. **isPrimary** set to *true*), depending on the **reliabilityLevel**. Read Service Fabric cluster capacity planning considerations for information on **nodeTypes** and **reliabilityLevel** values, and to know what are primary and the non-primary node types.

Endpoints used to configure the node types

- *clientConnectionEndpointPort* is the port used by the client to connect to the cluster, when using the client APIs.
- *clusterConnectionEndpointPort* is the port at which the nodes communicate with each other.
- *leaseDriverEndpointPort* is the port used by the cluster lease driver to find out if the nodes are still active.
- *serviceConnectionEndpointPort* is the port used by the applications and services deployed on a node, to communicate with the Service Fabric client on that particular node.
- *httpGatewayEndpointPort* is the port used by the Service Fabric Explorer to connect to the cluster.
- *ephemeralPorts* override the dynamic ports used by the OS. Service Fabric will use a part of these as application ports and the remaining will be available for the OS. It will also map this range to the existing range present in the OS, so for all purposes, you can use the ranges given in the sample JSON files. You need to make sure that the difference between the start and the end ports is at least 255. You may run into conflicts if this difference is too low, since this range is shared with the operating system. See the configured dynamic port range by running `netsh int ipv4 show dynamicport tcp`.
- *applicationPorts* are the ports that will be used by the Service Fabric applications. These should be a subset of the *ephemeralPorts*, enough to cover the endpoint requirement of your applications. Service Fabric will use these whenever new ports are required, as well as take care of opening the firewall for these ports.
- *reverseProxyEndpointPort* is an optional reverse proxy endpoint. See Service Fabric Reverse Proxy for more details.

Other Settings

The **fabricSettings** section allows you to set the root directories for the Service Fabric data and logs. You can customize these only during the initial cluster creation. See below for a sample snippet of this section.

```
"fabricSettings": [{
    "name": "Setup",
    "parameters": [{
        "name": "FabricDataRoot",
        "value": "C:\\ProgramData\\SF"
    }, {
        "name": "FabricLogRoot",
        "value": "C:\\ProgramData\\SF\\Log"
}]
```

We recommended using a non-OS drive as the FabricDataRoot and FabricLogRoot as it provides more reliability against OS crashes. Note that if you customize only the data root, then the log root will be placed one level below the data root.

Next steps

Once you have a complete ClusterConfig.JSON file configured as per your standalone cluster setup, you can deploy your cluster by following the article Create an Azure Service Fabric cluster on-premises or in the cloud and then proceed to visualizing your cluster with Service Fabric Explorer.

Introducing the Service Fabric cluster resource manager

11/15/2016 • 6 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

Traditionally managing IT systems or a set of services meant getting a few physical or virtual machines dedicated to those specific services or systems. Many major services were broken down into a "web" tier and a "data" or "storage" tier, maybe with a few other specialized components like a cache. Other types of applications would have a messaging tier where requests flowed in and out, connected to a work tier for any analysis or transformation necessary as a part of the messaging. Each type of workload got a specific machines dedicated to it: the database got a couple machines dedicated to it, the web servers a few. If a particular type of workload caused the machines it was on to run too hot, then you added more machines with that type of workload configured to run on it, or replaced a few of the machines with larger machines. Easy. If a machine failed, that part of the overall application ran at lower capacity until the machine could be restored. Still fairly easy (if not necessarily fun).

Now however, let's say you've found a need to scale out and have taken the containers and/or microservice plunge. Suddenly you find yourself with tens, hundreds, or even thousands of machines, dozens of different types of services, perhaps hundreds of different instances of those services, each with one or more instances or replicas for High Availability (HA).

Suddenly managing your environment is not so simple as managing a few machines dedicated to single types of workloads. Your servers are virtual and no longer have names (you *have* switched mindsets from pets to cattle after all). Configuration is less about the machines and more about the services themselves. Dedicated hardware is a thing of the past, and services have become small distributed systems, spanning multiple smaller pieces of commodity hardware.

As a consequence of breaking your formerly monolithic, tiered app into separate services running on commodity hardware, you now have many more combinations to deal with. Who decides what types of workloads can run on which hardware, or how many? Which workloads work well on the same hardware, and which conflict? When a machine goes down... what was even running there? Who is in charge of making sure that workload starts running again? Do you wait for the (virtual?) machine to come back or do your workloads automatically fail over to other machines and keep running? Is human intervention required? What about upgrades in this sort of environment?

As developers and operators living in this sort of world, we're going to need some help managing this complexity, and you get the sense that a hiring binge and trying to paper over the complexity with people is not the right answer.

What to do?

Introducing orchestrators

An "Orchestrator" is the general term for a piece of software that helps administrators manage these types of environments. Orchestrators are the components that take in requests like "I would like 5 copies of this service running in my environment", make it true, and then (try) to keep it that way.

Orchestrators (not humans) are what swing into action when a machine fails or a workload terminates for some

unexpected reason. Most Orchestrators do more than just deal with failure, such as helping with new deployments, handling upgrades, and dealing with resource consumption, but all are fundamentally about maintaining some desired state of configuration in the environment. You want to be able to tell an Orchestrator what you want and have it do the heavy lifting. Chronos or Marathon on top of Mesos, Fleet, Swarm, Kubernetes, and Service Fabric are all examples of Orchestrators (or have them built in). More are being created all the time as the complexities of managing real world deployments in different types of environments and conditions grow and change.

Orchestration as a service

The job of the Orchestrator within a Service Fabric cluster is handled primarily by the Cluster Resource Manager. The Service Fabric Cluster Resource Manager is one of the System Services within Service Fabric and is automatically started up within each cluster. Generally, the Cluster Resource Manager's job is broken down into three parts:

1. Enforcing Rules
2. Optimizing Your Environment
3. Assisting in Other Processes

What it isn't

In traditional N tier web-apps there was always some notion of a "Load Balancer", usually referred to as a Network Load Balancer (NLB) or an Application Load Balancer (ALB) depending on where it sat in the networking stack. Some load balancers are Hardware based like F5's BigIP offering, others are software based such as Microsoft's NLB. In other environments you might see something like HAProxy in this role. In these architectures the job of load balancing is to make sure that all of the different stateless front end machines or the different machines in the cluster receive (roughly) the same amount of work. Strategies for this varied, from sending each different call to a different server, to session pinning/stickiness, to actual estimation and call allocation based on its expected cost and current machine load.

Note that this was at best the mechanism for ensuring that the web tier remained roughly balanced. Strategies for balancing the data tier were completely different and depended on the data storage mechanism, usually centering around data sharding, caching, database managed views and stored procedures, etc.

While some of these strategies are interesting, the Service Fabric Cluster Resource Manager is not anything like a network load balancer or a cache. While a Network Load Balancer ensures that the front ends are balanced by moving traffic to where the services are running, the Service Fabric Cluster Resource Manager takes a completely different strategy – fundamentally, Service Fabric moves *services* to where they make the most sense (and expects traffic or load to follow). This can be, for example, nodes which are currently cold because the services which are there are not doing a lot of work right now, or which were deleted or moved elsewhere. As another example the Cluster Resource Manager could also move a service away from a machine which is about to be upgraded or which is overloaded due to a spike in consumption by the services which were running on it. Because the Cluster Resource Manager is responsible for moving services around (not delivering network traffic to where services already are), it contains a significantly different feature set compared to what you would find in a network load balancer, and employs fundamentally different strategies for ensuring that the hardware resources in the cluster are well utilized.

Next steps

- For information on the architecture and information flow within the Cluster Resource manager, check out this article
- The Cluster Resource Manager has a lot of options for describing the cluster. To find out more about them check out this article on describing a Service Fabric cluster
- For more information about the other options available for configuring services check out the topic on the other Cluster Resource Manager configurations available Learn about configuring Services

- Metrics are how the Service Fabric Cluster Resource Manger manages consumption and capacity in the cluster. To learn more about them and how to configure them check out this article
- The Cluster Resource Manager works with Service Fabric's management capabilities. To find out more about that integration, read this article
- To find out about how the Cluster Resource Manager manages and balances load in the cluster, check out the article on balancing load

Cluster resource manager architecture overview

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

In order to manage the resources in your cluster, the Service Fabric Cluster Resource Manager must have several pieces of information. It has to know which services currently exist and the current (or default) amount of resources that those services are consuming. It has to know the actual capacity of the nodes in the cluster, and thus the amount of resources available both in the cluster as a whole and remaining on a particular node. The resource consumption of a given service can change over time, as well as the fact that services, in reality, usually care about more than one resource. Across many different services there may be both real physical resources being measured and reported as metrics like memory and disk consumption, as well as (and actually more commonly) logical metrics - things like "WorkQueueDepth" or "TotalRequests". Both logical and physical metrics may be used across many different types of services or maybe specific to only a couple services.

Other considerations

The owners and operators of the cluster are occasionally different from the service authors, or at a minimum are the same people wearing different hats; for example when developing your service you know a few things about what it requires in terms of resources and how the different components should ideally be deployed, but as the person handling a live-site incident for that service in production you have a different job to do, and require different tools. In addition, neither the cluster nor the services themselves are a statically configured: the number of nodes in the cluster can grow and shrink, nodes of different sizes can come and go, and services can be created, removed, and change their desired resource allocations on the fly. Upgrades or other management operations can roll through the cluster, and of course things can fail at any time.

Cluster resource manager components and data flow

The Cluster Resource Manager will have to know many things about the overall cluster, as well as the requirements of individual services and the stateless instances or stateful replicas that make it up. To accomplish this, we have agents of the Cluster Resource Manager that run on individual nodes in order to aggregate local resource consumption information, and a centralized, fault-tolerant Cluster Resource Manager service that aggregates all of the information about the services and the cluster and reacts based on its current configuration. The fault tolerance for the Cluster Resource Manager service (and all other system services) is achieved via exactly the same mechanisms that we use for your services, namely replication of the service's state to quorums of some number of replicas in the cluster (usually 7).

Let's take a look at this diagram above as an example. During runtime there are a whole bunch of changes which could happen: For example, let's say there are some changes in the amount of resources services consume, some service failures, some nodes join and leave the cluster, etc. All the changes on a specific node are aggregated and periodically sent to the Cluster Resource Manager service (1,2) where they are aggregated again, analyzed, and stored. Every few seconds that service looks at all of the changes, and determines if there are any actions necessary (3). For example, it could notice that nodes have been added to the cluster and are empty, and decide to move some services to those nodes. It could also notice that a particular node is overloaded, or that certain services have failed (or been deleted), freeing up resources on other nodes.

Let's take a look at the following diagram and see what happens next. Let's say that the Cluster Resource Manager determines that changes are necessary. It coordinates with other system services (in particular the Failover Manager) to make the necessary changes. Then the change requests are sent to the appropriate nodes (4). In this case, we presume that the Resource Manager noticed that Node 5 was overloaded, and so decided to move service B from N5 to N4. At the end of the reconfiguration (5), the cluster looks like this:

Next steps

- The Cluster Resource Manager has a lot of options for describing the cluster. To find out more about them

check out this article on describing a Service Fabric cluster

Describing a service fabric cluster

11/15/2016 • 20 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • PepijnGramberg • Eran Ben-Shahar • Natalia

The Service Fabric Cluster Resource Manager provides several mechanisms for describing a cluster. During runtime, the Cluster Resource Manager uses this information to ensure high availability of the services running in the cluster while also ensuring that the resources in the cluster are being used appropriately.

Key concepts

The Cluster Resource Manager supports several features that describe a cluster:

- Fault Domains
- Upgrade Domains
- Node Properties
- Node Capacities

Fault domains

A fault domain is any area of coordinated failure. A single machine is a fault domain (since it alone can fail for a variety of reasons, from power supply failures to drive failures to bad NIC firmware). A bunch of machines connected to the same Ethernet switch are in the same fault domain, as would be those connected to a single source of power. Since it's natural for these to overlap, Fault Domains are inherently hierarchal and are represented as URIs in Service Fabric.

If you were setting up your own cluster you'd need to think about all of these different areas of failure and make sure that your fault domains were set up correctly so that Service Fabric would know where it was safe to place services. By "safe" we really mean smart – we don't want to place services such that a loss of a fault domain (the failure of any of the components listed above, for example) causes the service to go down. In the Azure environment we leverage the fault domain information provided by the environment in order to correctly configure the nodes in the cluster on your behalf. In the graphic below (Fig. 7) we color all of the entities that reasonably result in a fault domain as a simple example and list out all of the different fault domains that result. In this example, we have datacenters (DC), racks (R), and blades (B). Conceivably, if each blade holds more than one virtual machine, there could be another layer in the fault domain hierarchy.

During run time, the Service Fabric Cluster Resource Manager considers the fault domains in the cluster and attempts to spread out the replicas for a given service so that they are all in separate fault domains. This process helps ensure that in case of failure of any one fault domain (at any level in the hierarchy), that the availability of that service is not compromised.

Service Fabric's Cluster Resource Manager doesn't really care about how many layers there are in the hierarchy, however since it does try to ensure that the loss of any one portion of the hierarchy doesn't impact the cluster or the services running on top of it, it is generally best if at each level of depth in the fault domain there are the same number of machines. This prevents one portion of the hierarchy from having to contain more services at the end of the day than others.

Configuring your cluster in such a way that the "tree" of fault domains is unbalanced makes it rather hard for the Cluster Resource Manager to figure out what the best allocation of replicas is, particularly since it means that the loss of a particular domain can overly impact the availability of the cluster – the Cluster Resource Manager is torn between using the machines in that "heavy" domain efficiently by placing services on them and placing services so that the loss of the domain doesn't cause problems.

In the diagram below we show two different example cluster layouts, one where the nodes are well distributed across the fault domains, and another where one fault domain ends up with many more nodes. Note that in Azure the choices about which nodes end up in which fault and upgrade domains is handled for you, so you should never see these sorts of imbalances. However, if you ever stand up your own cluster on-premise or in another environment, it's something you have to think about.

Upgrade domains

Upgrade Domains are another feature that helps the Service Fabric Resource Manager to understand the layout

of the cluster so that it can plan ahead for failures. Upgrade Domains define areas (sets of nodes, really) that will go down at the same time during an upgrade.

Upgrade Domains are a lot like Fault Domains, but with a couple key differences. First, Upgrade Domains are usually defined by policy; whereas Fault Domains are rigorously defined by the areas of coordinated failures (and hence usually the hardware layout of the environment). In the case of Upgrade Domains however you get to decide how many you want. Another difference is that (today at least) Upgrade Domains are not hierarchical – they are more like a simple tag than a hierarchy.

The picture below shows a fictional setup where we have three upgrade domains striped across three fault domains. It also shows one possible placement for three different replicas of a stateful service. Note that they are all in different fault and upgrade domains. This means that we could lose a fault domain while in the middle of a service upgrade and there would still be one running copy of the code and data in the cluster. Depending on your needs this could be good enough, however you may notice though that this copy could be old (as Service Fabric uses quorum based replication). In order to truly survive two failures you'd need more replicas (five at a minimum).

There are pros and cons to having large numbers of upgrade domains – the pro is that each step of the upgrade is more granular and therefore affects a smaller number of nodes or services. This results in fewer services having to move at a time, introducing less churn into the system and overall improving reliability (since less of the service will be impacted by any issue). The downside of having many upgrade domains is that Service Fabric verifies the health of each Upgrade Domain as it is upgraded and ensures that the Upgrade Domain is healthy before moving on to the next Upgrade Domain. The goal of this check is to ensure that services have a chance to stabilize and that their health is validated before the upgrade proceeds, so that any issues are detected. The tradeoff is acceptable because it prevents bad changes from affecting too much of the service at a time.

Too few upgrade domains has its own side effects – while each individual upgrade domain is down and being upgraded a large portion of your overall capacity is unavailable. For example, if you only have three upgrade domains you are taking down about 1/3 of your overall service or cluster capacity at a time. This isn't desirable as you have to have enough capacity in the rest of your cluster to cover the workload, meaning that in the normal case those nodes are less-loaded than they would otherwise be, increasing COGS.

There's no real limit to the total number of fault or upgrade domains in an environment, or constraints on how they overlap. Common structures that we've seen are 1:1 (where each unique fault domain maps to its own upgrade domain as well), an Upgrade Domain per Node (physical or virtual OS instance), and a "striped" or "matrix" model where the Fault Domains and Upgrade Domains form a matrix with machines usually running down the diagonal.

UD Per Node

	FD1	FD2	FD3
UD1	Node1		
UD2		Node2	
UD3			Node3
UD4	Node4		
UD5		Node5	
UD6			Node6
UD7	Node7		
UD8		Node8	
UD9			Node9

FD/UD Matrix (Sparse/Diagonal)

	FD1	FD2	FD3	FD4	FD5
UD1	Node1				
UD2		Node2			
UD3			Node3		
UD4				Node4	
UD5					Node5

1:1 FD & UD

FD1/UD1	FD2/UD2	FD3/UD3
Node1	Node2	Node3
Node4	Node5	Node6
Node7	Node8	Node9

FD/UD Matrix

	FD1	FD2	FD3
UD1	Node1	Node2	Node3
UD2	Node4	Node5	Node6
UD3	Node7	Node8	Node9

There's no best answer which layout to choose, each has some pros and cons. For example, the 1FD:1UD model is fairly simple to set up, whereas the 1 UD per Node model is most like what people are used to from managing small sets of machines in the past where each would be taken down independently.

The most common model (and the one that we use for the hosted Azure Service Fabric clusters) is the FD/UD matrix, where the FDs and UDs form a table and nodes are placed starting along the diagonal. Whether this ends up sparse or packed depends on the total number of nodes compared to the number of FDs and UDs (put differently, for sufficiently large clusters, almost everything ends up looking like the dense matrix pattern, shown in the bottom right option of Figure 10).

Fault and upgrade domain constraints and resulting behavior

The Cluster Resource manager treats the desire to keep a service balanced across fault and upgrade domains as a constraint. You can find out more about constraints in this article. The fault and upgrade domain constraints are defined as following: "For a given service partition there should never be a difference *greater than one* in the number of replicas between two domains." Practically what this means is that for a given service certain movements or certain arrangements might not be valid in the cluster, because doing so would violate the fault or upgrade domain constraint.

Let's take a look at one example. Let's say that we have a cluster with 6 nodes, configured with 5 fault domains and 5 upgrade domains.

	FD0	FD1	FD2	FD3	FD4
UD0	N1				
UD1	N6	N2			
UD2			N3		
UD3				N4	
UD4					N5

Now let's say that we create a service with a TargetReplicaSetSize of 5. The replicas land on N1-N5. In fact, N6 will never get used. But why? Well let's take a look at the difference between the current layout and what would happen if we had chosen N6 instead, and think about how that relates to our definition of the FD and UD

constraint.

Here's the layout we got and the total number of replicas per fault and upgrade domain.

	FD0	FD1	FD2	FD3	FD4	UDTOTAL
UD0	R1					1
UD1		R2				1
UD2			R3			1
UD3				R4		1
UD4					R5	1
FDTotal	1	1	1	1	1	-

Note that this layout is balanced in terms of nodes per fault domain and upgrade domain, and it is also balanced in terms of the number of replicas per fault and upgrade domain. Each domain has the same number of nodes and the same number of replicas.

Now, let's take a look at what would happen if instead of N2, we'd used N6. How would the replicas be distributed then? Well, they'd look something like this:

	FD0	FD1	FD2	FD3	FD4	UDTOTAL
UD0	R1					1
UD1	R5					1
UD2			R2			1
UD3				R3		1
UD4					R4	1
FDTotal	2	0	1	1	1	-

This violates our definition for the fault domain constraint, since FD0 has 2 replicas, while FD1 has 0, making the total difference 2 and thus the Cluster Resource Manager will not allow this arrangement. Similarly if we had picked N2-6 we'd get:

	FD0	FD1	FD2	FD3	FD4	UDTOTAL
UD0						0
UD1	R5	R1				2
UD2			R2			1
UD3				R3		1
UD4					R4	1

	FD0	FD1	FD2	FD3	FD4	UDTOTAL
FDTotal	1	1	1	1	1	-

Which while balanced in terms of fault domains is violating the upgrade domain constraint (since UD0 has 0 replicas while UD1 has 2), and hence is invalid as well.

Configuring fault and upgrade domains

Defining Fault Domains and Upgrade Domains is done automatically in Azure hosted Service Fabric deployments; Service Fabric just picks up the environment information from Azure. In Azure both the fault and upgrade domain information looks "single level" but it really is encapsulating information from lower layers of the Azure stack and just presenting the logical fault and upgrade domains from the user's perspective.

If you're standing up your own cluster (or just want to try running a particular topology on your development machine) you'll need to provide the fault domain and upgrade domain information yourself. In this example we define a 9 node local development cluster that spans three "datacenters" (each with three racks), and three upgrade domains striped across those three datacenters. In the cluster manifest template, it looks something like this:

ClusterManifest.xml

```
  <Infrastructure>
    <!-- IsScaleMin indicates that this cluster runs on one-box /one single server -->
    <WindowsServer IsScaleMin="true">
      <NodeList>
        <Node NodeName="Node01" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType01"
FaultDomain="fd:/DC01/Rack01" UpgradeDomain="UpgradeDomain1" IsSeedNode="true" />
        <Node NodeName="Node02" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType02"
FaultDomain="fd:/DC01/Rack02" UpgradeDomain="UpgradeDomain2" IsSeedNode="true" />
        <Node NodeName="Node03" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType03"
FaultDomain="fd:/DC01/Rack03" UpgradeDomain="UpgradeDomain3" IsSeedNode="true" />
        <Node NodeName="Node04" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType04"
FaultDomain="fd:/DC02/Rack01" UpgradeDomain="UpgradeDomain1" IsSeedNode="true" />
        <Node NodeName="Node05" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType05"
FaultDomain="fd:/DC02/Rack02" UpgradeDomain="UpgradeDomain2" IsSeedNode="true" />
        <Node NodeName="Node06" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType06"
FaultDomain="fd:/DC02/Rack03" UpgradeDomain="UpgradeDomain3" IsSeedNode="true" />
        <Node NodeName="Node07" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType07"
FaultDomain="fd:/DC03/Rack01" UpgradeDomain="UpgradeDomain1" IsSeedNode="true" />
        <Node NodeName="Node08" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType08"
FaultDomain="fd:/DC03/Rack02" UpgradeDomain="UpgradeDomain2" IsSeedNode="true" />
        <Node NodeName="Node09" IPAddressOrFQDN="localhost" NodeTypeRef="NodeType09"
FaultDomain="fd:/DC03/Rack03" UpgradeDomain="UpgradeDomain3" IsSeedNode="true" />
      </NodeList>
    </WindowsServer>
  </Infrastructure>
```

> **NOTE**
> In Azure deployments, fault domains and upgrade domains are assigned by Azure. Therefore, the definition of your nodes and roles within the infrastructure option for Azure does not include fault domain or upgrade domain information.

Placement constraints and node properties

Sometimes (in fact, most of the time) you're going to want to ensure that certain workloads run only on certain

nodes or certain sets of nodes in the cluster. For example, some workload may require GPUs or SSDs while others may not. A great example of this is pretty much every n-tier architecture out there, where certain machines serve as the front end/interface serving side of the application (and hence are probably exposed to the internet) while a different set (often with different hardware resources) handle the work of the compute or storage layers (and usually are not exposed to the internet). Service Fabric expects that even in a microservices world there are cases where particular workloads will need to run on particular hardware configurations, for example:

- an existing n-tier application has been “lifted and shifted” into a Service Fabric environment
- a workload wants to run on specific hardware for performance, scale, or security isolation reasons
- A workload needs to be isolated from other workloads for policy or resource consumption reasons

In order to support these sorts of configurations Service Fabric has a first class notion of what we call placement constraints. Placement constraints can be used to indicate where certain services should run. The set of constraints is extensible by users, meaning that people can tag nodes with custom properties and then select for those as well.

The different key/value tags on nodes are known as node placement *properties* (or just node properties), whereas the statement at the service is called a placement *constraint*. The value specified in the node property can be a string, bool, or signed long. The constraint can be any Boolean statement that operates on the different node properties in the cluster. The valid selectors in these boolean statements (which are strings) are:

- conditional checks for creating particular statements
  - "equal to" ==
  - "greater than" >
  - "less than" <
  - "not equal to" !=
  - "greater than or equal to" >=
  - "less than or equal to" <=
- boolean statements for grouping and negation
  - "and" &&
  - "or" ||
  - "not" !
- parenthesis for grouping operations
  - ()

  Here are some examples of basic constraint statements that use some of the symbols above. Note that node properties can be strings, bools, or numerical values.
  - "Foo >= 5"

- "NodeColor != green"
- "((OneProperty < 100) || ((AnotherProperty == false) && (OneProperty >= 100)))"

Only nodes where the overall statement evaluates to “True” can have the service placed on it. Nodes without a property defined do not match any placement constraint that contains that property.

Service Fabric also defines some default properties which can be used automatically without the user having to define them. As of this writing the default properties defined at each node are the NodeType and the NodeName. So for example you could write a placement constraint as "(NodeType == NodeType03)". Generally we have found NodeType to be one of the most commonly used properties, as it usually corresponds 1:1 with a type of a machine, which in turn correspond to a type of workload in a traditional n-tier application architecture.

Let’s say that the following node properties were defined for a given node type: ClusterManifest.xml

```
<NodeType Name="NodeType01">
  <PlacementProperties>
    <Property Name="HasSSD" Value="true"/>
    <Property Name="NodeColor" Value="green"/>
    <Property Name="SomeProperty" Value="5"/>
  </PlacementProperties>
</NodeType>
```

You can create service placement constraints for a service like this:

C#

```
FabricClient fabricClient = new FabricClient();
StatefulServiceDescription serviceDescription = new StatefulServiceDescription();
serviceDescription.PlacementConstraints = "(HasSSD == true && SomeProperty >= 4)";
// add other required servicedescription fields
//...
await fabricClient.ServiceManager.CreateServiceAsync(serviceDescription);
```

Powershell:

```
New-ServiceFabricService -ApplicationName $applicationName -ServiceName $serviceName -ServiceTypeName
$serviceType -Stateful -MinReplicaSetSize 2 -TargetReplicaSetSize 3 -PartitionSchemeSingleton -
PlacementConstraint "HasSSD == true && SomeProperty >= 4"
```

If you are sure that all nodes of NodeType01 are valid, you could also just select that node type, using placement constraints like those show in the pictures above.

One of the cool things about a service's placement constraints is that they can be updated dynamically during runtime. So if you need to, you can move a service around in the cluster, add and remove requirements, etc. Service Fabric takes care of ensuring that the service stays up and available even when these types of changes are ongoing.

C#:

```
StatefulServiceUpdateDescription updateDescription = new StatefulServiceUpdateDescription();
updateDescription.PlacementConstraints = "NodeType == NodeType01";
await fabricClient.ServiceManager.UpdateServiceAsync(new Uri("fabric:/app/service"), updateDescription);
```

Powershell:

```
Update-ServiceFabricService -Stateful -ServiceName $serviceName -PlacementConstraints "NodeType == NodeType01"
```

Placement constraints (along with many other orchestrator controls that we're going to talk about) are specified for every different named service instance. Updates always take the place (overwrite) what was previously specified.

It is also worth noting that at this point the properties on a node are defined via the cluster definition and hence cannot be updated without an upgrade to the cluster and will require each node to go down and then come back up in order to refresh its properties.

Capacity

One of the most important jobs of any orchestrator is to help manage resource consumption in the cluster. The last thing you want if you're trying to run services efficiently is a bunch of nodes which are hot (leading to resource contention and poor performance) while others are cold (wasted resources). But let's think even more basic than balancing (which we'll get to in a minute) – what about just ensuring that nodes don't run out of resources in the first place?

Service Fabric represents resources as "Metrics". Metrics are any logical or physical resource that you want to describe to Service Fabric. Examples of metrics are things like "WorkQueueDepth" or "MemoryInMb". Metrics are different from placements constraints and node properties in that node properties are generally static descriptors of the nodes themselves, whereas metrics are about resources that nodes have and that services consume when they are running on a node. So a property would be something like HasSSD and could be set to true or false, but the amount of space available on that SSD (and consumed by services) would be a metric like "DriveSpaceInMb". Capacity on the node would set the "DriveSpaceInMb" to the amount of total non-reserved space on the drive, and services would report how much of the metric they used during runtime.

If you turned off all resource *balancing*, Service Fabric's Cluster Resource Manager would still be able to ensure that no node ended up over its capacity (unless the cluster as a whole was too full). Capacity is another *constraint* which the Cluster Resource Manager uses to understand how much of a resource a node has. Both the capacity and the consumption at the service level are expressed in terms of metrics. So for example, the metric might be "MemoryInMb" - a given Node may have a capacity for MemoryInMb of 2048, while a given service can say it is currently consuming 64 of MemoryInMb.

During runtime, the Cluster Resource Manager tracks how much of each resource is present on each node (defined by its capacity) and how much is remaining (by subtracting any declared usage from each service). With this information, the Service Fabric Resource Manager can figure out where to place or move replicas so that nodes don't go over capacity.

C#:

```
StatefulServiceDescription serviceDescription = new StatefulServiceDescription();
ServiceLoadMetricDescription metric = new ServiceLoadMetricDescription();
metric.Name = "MemoryInMb";
metric.PrimaryDefaultLoad = 64;
metric.SecondaryDefaultLoad = 64;
metric.Weight = ServiceLoadMetricWeight.High;
serviceDescription.Metrics.Add(metric);
await fabricClient.ServiceManager.CreateServiceAsync(serviceDescription);
```

Powershell:

```
New-ServiceFabricService -ApplicationName $applicationName -ServiceName $serviceName -ServiceTypeName
$serviceTypeName –Stateful -MinReplicaSetSize 2 -TargetReplicaSetSize 3 -PartitionSchemeSingleton –Metric
@("Memory,High,64,64)
```

Memory Capacity	4096	768	2048
Memory Consumption	3584	0	0
Remaining	512	768	2048

You can see these in the cluster manifest:

ClusterManifest.xml

```
    <NodeType Name="NodeType02">
      <Capacities>
        <Capacity Name="MemoryInMb" Value="2048"/>
        <Capacity Name="Disk" Value="10000"/>
      </Capacities>
    </NodeType>
```

It is also possible that a service's load changes dynamically. Say that a replica's load changed from 64 to 1024, but the node it was running on at that time only had 512 (of the "MemoryInMb" metric) remaining. Because of this, where a replica or instance is currently placed becomes invalid since the combined usage of all of the replicas and instances on that node exceeds that node's capacity. We'll talk more about this scenario where load can change dynamically later, but as far as capacity goes it is handled the same way – the Cluster Resource Manager automatically kicks in and gets the node back below capacity by moving one or more of the replicas or instances on that node to different nodes. When doing this the Cluster Resource Manager tries to minimize the cost of all of the movements (we'll come back to the notion of Cost later).

Cluster capacity

So how do we keep the overall cluster from being too full? Well, with dynamic load there's actually not a lot we can do (since services can have their load spike independent of actions taken by the Cluster Resource Manager – your cluster with a lot of headroom today may be rather underpowered when you become famous tomorrow), but there are some controls that are baked in to prevent basic problems. The first thing we can do is prevent the creation of new workloads that would cause the cluster to become full.

Say that you go to create a simple stateless service and it has some load associated with it (more on default and

dynamic load reporting later). For this service, let's say that it cares about some resource (let's say DiskSpace) and that by default it is going to consume 5 units of DiskSpace for every instance of the service. You want to create 3 instances of the service. Great! So that means that we need 15 units of DiskSpace to be present in the cluster in order for us to even be able to create these service instances. Service Fabric is continually calculating the overall capacity and consumption of each metric, so we can easily make the determination and reject the create service call if there's insufficient space.

Note that since the requirement is only that there be 15 units available, this space could be allocated many different ways; it could be one remaining unit of capacity on 15 different nodes, for example, or three remaining units of capacity on 5 different nodes, etc. If there isn't sufficient capacity on three different nodes Service Fabric will reorganize the services already in the cluster in order to make room on the three necessary nodes. Such rearrangement is almost always possible unless the cluster as a whole is almost entirely full.

Buffered Capacity

Another thing that helps people manage overall cluster capacity is the notion of some reserved buffer to the capacity specified at each node. This setting is optional, but allows people to reserve some portion of the overall node capacity so that it is only used to place services during upgrades and failures – cases where the capacity of the cluster is otherwise reduced. Today buffer is specified globally per metric for all nodes via the ClusterManifest. The value you pick for the reserved capacity will be a function of which resources your services are more constrained on, as well as the number of fault and upgrade domains you have in the cluster. Generally more fault and upgrade domains means that you can pick a lower number for your buffered capacity, as you will expect smaller amounts of your cluster to be unavailable during upgrades and failures. Note that specifying the buffer percentage only makes sense if you have also specified the node capacity for a metric.

Here's an example of how to specify buffered capacity:

ClusterManifest.xml

```
<Section Name="NodeBufferPercentage">
    <Parameter Name="DiskSpace" Value="0.10" />
    <Parameter Name="Memory" Value="0.15" />
    <Parameter Name="SomeOtherMetric" Value="0.20" />
</Section>
```

The creation of new services will fail when the cluster is out of buffered capacity, ensuring that the cluster retains enough spare overhead such that upgrades and failures don't result in nodes being actually over capacity. The Cluster Resource Manager exposes a lot of this information via PowerShell and the Query APIs, letting you see the buffered capacity settings, the total capacity, and the current consumption for every metric in use in the cluster. Here we see an example of that output:

```
PS C:\Users\user> Get-ServiceFabricClusterLoadInformation
LastBalancingStartTimeUtc : 9/1/2015 12:54:59 AM
LastBalancingEndTimeUtc   : 9/1/2015 12:54:59 AM
LoadMetricInformation     :
                            LoadMetricName        : Metric1
                            IsBalancedBefore      : False
                            IsBalancedAfter       : False
                            DeviationBefore       : 0.192450089729875
                            DeviationAfter        : 0.192450089729875
                            BalancingThreshold    : 1
                            Action                : NoActionNeeded
                            ActivityThreshold     : 0
                            ClusterCapacity       : 189
                            ClusterLoad           : 45
                            ClusterRemainingCapacity : 144
                            NodeBufferPercentage  : 10
                            ClusterBufferedCapacity : 170
                            ClusterRemainingBufferedCapacity : 125
                            ClusterCapacityViolation : False
                            MinNodeLoadValue      : 0
                            MinNodeLoadNodeId     : 3ea71e8e01f4b0999b121abcbf27d74d
                            MaxNodeLoadValue      : 15
                            MaxNodeLoadNodeId     : 2cc648b6770be1bc9824fa995d5b68b1
```

Next steps

- For information on the architecture and information flow within the Cluster Resource manager, check out this article
- Defining Defragmentation Metrics is one way to consolidate load on nodes instead of spreading it out. To learn how to configure defragmentation, refer to this article
- Start from the beginning and get an Introduction to the Service Fabric Cluster Resource Manager
- To find out about how the Cluster Resource Manager manages and balances load in the cluster, check out the article on balancing load

Introduction to Application Groups

11/15/2016 • 9 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

Service Fabric's Cluster Resource Manager typically manages cluster resources by spreading the load (represented via Metrics) evenly throughout the cluster. Service Fabric also manages the capacity of the nodes in the cluster and the cluster as a whole through the notion of capacity. This works great for a lot of different types of workloads, but patterns that make heavy use of different Service Fabric Application Instances sometimes bring in additional requirements. Some additional requirements are typically:

- Ability to reserve capacity for an Application Instance's services on some number of nodes
- Ability to limit the total number of nodes that a given set of services within an application is allowed to run on
- Defining capacities on the application instance itself in order to limit the number or total resource consumption of the services inside it

In order to meet these requirements, we developed support for what we call Application Groups.

Managing Application capacity

Application capacity can be used to limit the number of nodes spanned by an application, as well as the total metric load of that the applications' instances on individual nodes. It can also be used to reserve resources in the cluster for the application.

Application capacity can be set for new applications when they are created; it can also be updated for existing applications that were created without specifying Application capacity.

Limiting the maximum number of nodes

The simplest use case for Application capacity is when an application instantiation needs to be limited to a certain maximum number of nodes. If no Application Capacity is specified, the Service Fabric Cluster Resource Manager will instantiate replicas according to normal rules (balancing or defragmentation), which usually means that its services will be spread across all of the available nodes in the cluster, or if defragmentation is turned on some arbitrary but smaller number of nodes.

The following image shows the potential placement of an application instance without the maximum number of nodes defined and then same application with a maximum number of nodes set. Note that there is no guarantees made about which replicas or instances of which services will get placed together.

In the left example, the application doesn't have Application Capacity set, and it has three services. CRM has made a logical decision to spread out all replicas across six available nodes in order to achieve the best balance in the cluster. In the right example, we see the same application that is constrained on three nodes, and where Service Fabric CRM has achieved the best balance for the replicas of application's services.

The parameter that controls this behavior is called MaximumNodes. This parameter can be set during application creation, or updated for an application instance which was already running, in which case Service Fabric CRM will constrain the replicas of application's services to the defined maximum number of nodes.

Powershell

```
New-ServiceFabricApplication -ApplicationName fabric:/AppName -ApplicationTypeName AppType1 -
ApplicationTypeVersion 1.0.0.0 -MaximumNodes 3
Update-ServiceFabricApplication –Name fabric:/AppName –MaximumNodes 5
```

C#

```
ApplicationDescription ad = new ApplicationDescription();
ad.ApplicationName = new Uri("fabric:/AppName");
ad.ApplicationTypeName = "AppType1";
ad.ApplicationTypeVersion = "1.0.0.0";
ad.MaximumNodes = 3;
fc.ApplicationManager.CreateApplicationAsync(ad);

ApplicationUpdateDescription adUpdate = new ApplicationUpdateDescription(new Uri("fabric:/AppName"));
adUpdate.MaximumNodes = 5;
fc.ApplicationManager.UpdateApplicationAsync(adUpdate);

var appMetric = new ApplicationMetricDescription();
appMetric.Name = "Metric1";
appMetric.TotalApplicationCapacity = 1000;

adUpdate.Metrics.Add(appMetric);
```

Application Metrics, Load, and Capacity

Application Groups also allow you to define metrics associated with a given application instance, as well as the capacity of the application with regard to those metrics. So for example you could define that as many services as you want could be created in

For each metric, there are 2 values that can be set to describe the capacity for that application instance:

- Total Application Capacity – Represents the total capacity of the application for a particular metric. Service Fabric CRM will try to limit the sum of metric loads of this application's services to the specified value; furthermore, if the application's services are already consuming load up to this limit, Service Fabric Cluster Resource Manager will disallow the creation of any new services or partitions which would cause total load to go over this limit.
- Maximum Node Capacity – Specifies the maximum total load for replicas of the applications' services on a single node. If total load on the node goes over this capacity, Service Fabric CRM will attempt to move replicas to other nodes so that the capacity constraint is respected.

Reserving Capacity

Another common use for application groups is to ensure that resources within the cluster are reserved for a given application instance, even if the application instance doesn't have the services within it yet, or even if they aren't consuming the resources yet. Let's take a look at how that would work.

Specifying a minimum number of nodes and resource reservation

Reserving resources for an application instance requires specifying a couple additional parameters: *MinimumNodes* and *NodeReservationCapacity*

- MinimumNodes - Just like specifying a target maximum number of nodes that the services within an application can run on, you can also specify the minimum number of nodes that an application should run on. This setting effectively defines the number of nodes that the resources should be reserved on at a minimum, guaranteeing capacity within the cluster as a part of creating the application instance.
- NodeReservationCapacity - The NodeReservationCapacity can be defined for each metric within the application. This defines the amount of metric load reserved for the application on any node where any of the replicas or instances of the services within it are placed.

Let's take a look at an example of capacity reservation:

In the left example, applications do not have any Application Capacity defined. Service Fabric Cluster Resource Manager will balance the application's child services replicas and instances along with those from other services (outside of the application) to ensure balance in the cluster.

In the example on the right, let's say that the application was created with a MinimumNodes set to 2, MaximumNodes set to 3 and an application Metric defined with a reservation of 20, max capacity on a node of 50, and a total application capacity of 100, Service Fabric will reserve capacity on two nodes for the blue application, and will not allow other replicas in the cluster to consume that capacity. This reserved application capacity will be considered consumed and counted against the remaining cluster capacity.

When an application is created with reservation, the Cluster Resource Manager will reserve capacity equal to

MinimumNodes * NodeReservationCapacity in the cluster, but it will not reserve capacity on specific nodes until the replicas of the application's services are created and placed. This allows for flexibility, since nodes are chosen for new replicas only when they are created. Capacity is reserved on a specific node when at least one replica is placed on it.

Obtaining the application load information

For each application that has Application Capacity defined you can obtain the information about the aggregate load reported by replicas of its services. Service Fabric provides PowerShell and Managed API queries for this purpose.

For example, load can be retrieved using the following PowerShell cmdlet:

```
Get-ServiceFabricApplicationLoad –ApplicationName fabric:/MyApplication1
```

The output of this query will contain the basic information about Application Capacity that was specified for the application, such as Minimum Nodes and Maximum Nodes. There will also be information about the number of nodes that the application is currently using. Thus, for each load metric there will be information about:

- Metric Name: Name of the metric.
- Reservation Capacity: Cluster Capacity that is reserved in the cluster for this Application.
- Application Load: Total Load of this Application's child replicas.
- Application Capacity: Maximum permitted value of Application Load.

Removing Application Capacity

Once the Application Capacity parameters are set for an application, they can be removed using Update Application APIs or PowerShell cmdlets. For example:

```
Update-ServiceFabricApplication –Name fabric:/MyApplication1 –RemoveApplicationCapacity
```

This command will remove all Application Capacity parameters from the application, and Service Fabric Cluster Resource Manager will start treating this application as any other application in the cluster that does not have these parameters defined. The effect of the command is immediate, and Cluster Resource Manager will delete all Application Capacity parameters for this application; specifying them again would require Update Application APIs to be called with the appropriate parameters.

Restrictions on Application Capacity

There are several restrictions on Application Capacity parameters that must be respected. In case of validation errors, the creation or update of the application will be rejected with an error. All integer parameters must be non-negative numbers. Moreover, for individual parameters restrictions are as follows:

- MinimumNodes must never be greater than MaximumNodes.
- If capacities for a load metric are defined, then they must follow these rules:
  - Node Reservation Capacity must not be greater than Maximum Node Capacity. For example, you cannot limit the capacity for metric "CPU" on the node to 2 units, and try to reserve 3 units on each node.
  - If MaximumNodes is specified, then the product of MaximumNodes and Maximum Node Capacity must not be greater than Total Application Capacity. For example, if you set the Maximum Node Capacity for load metric "CPU" to 8 and you set the Maximum Nodes to 10, then Total Application Capacity must be greater than 80 for this load metric.

The restrictions are enforced both during application creation (on the client side), and during application update (on the server side). During creation, this is an example of a clear violation of the requirements since MaximumNodes < MinimumNodes, and the command will fail in the client before the request is even sent to Service Fabric cluster:

```
New-ServiceFabricApplication -Name fabric:/MyApplication1 -MinimumNodes 6 -MaximumNodes 2
```

An example of invalid update is as follows. If we take an existing application and update maximum nodes to some value, the update will pass:

```
Update-ServiceFabricApplication -Name fabric:/MyApplication1 6 -MaximumNodes 2
```

After that, we can attempt to update minimum nodes:

```
Update-ServiceFabricApplication -Name fabric:/MyApplication1 6 -MinimumNodes 6
```

The client does not have enough context about the application so it will allow the update to pass to the Service Fabric cluster. However, in the cluster, Service Fabric will validate the new parameter together with the existing parameters and will fail the update operation because the value foe minimum nodes is greater than the value for maximum nodes. In this case, Application Capacity parameters will remain unchanged.

These restrictions are put in place in order for Cluster Resource Manager to be able to provide the best placement for replicas of applications' services.

How not to use Application Capacity

- Do not use the Application Capacity to constrain the application to a specific subset of nodes: Although Service Fabric will ensure that Maximum Nodes is respected for each application that has Application Capacity specified, users cannot decide which nodes it will be instantiated on. This can be achieved using placement constraints for services.
- Do not use the Application Capacity to ensure that two services from the same application will always be placed alongside each other. This can be achieved by using affinity relationship between services, and affinity can be limited only to the services that should actually be placed together.

Next steps

- For more information about the other options available for configuring services check out the topic on the other Cluster Resource Manager configurations available Learn about configuring Services
- To find out about how the Cluster Resource Manager manages and balances load in the cluster, check out the article on balancing load
- Start from the beginning and get an Introduction to the Service Fabric Cluster Resource Manager
- For more information on how metrics work generally, read up on Service Fabric Load Metrics
- The Cluster Resource Manager has a lot of options for describing the cluster. To find out more about them check out this article on describing a Service Fabric cluster

Configuring cluster resource manager settings for service fabric services

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Matt • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

The Service Fabric Cluster Resource manager allows very fine grained control over the rules which govern every individual named service. Each named service instance can specify rules for how it should be allocated in the cluster, and can define the set of metrics that it wants to report, including how important they are to that service. Generally configuring services breaks down into three different tasks:

1. Configuring placement constraints
2. Configuring metrics
3. Configuring advanced placement policies (less common)

Let's talk about each of these in turn:

Placement constraints

Placement constraints are used to control which nodes in the cluster a service can actually run on. Typically you'll see a particular named service instance or all services of a given type constrained to run on a particular type of node. That said, placement constraints are extensible - you can define any set of properties on a node type basis, and then select for them with constraints when the service is created. Placement constraints are also dynamically updatable over the lifetime of the service, allowing you to respond to changes in the cluster. The properties of a given node can also be updated dynamically in the cluster. More information on placement constraints and how to configure them can be found in this article

Metrics

Metrics are the set of resources that a given named service instance needs, including information about how much of that resource each stateful replica or stateless instance of that service consumes by default. Metrics also include a weight which indicates how important balancing that metric is to that service, in case tradeoffs are necessary.

Other placement rules

There are other types of placement rules that are mainly useful in clusters which are geographically distributed, or in other less common scenarios. These are configured via either Correlations or Policies. While they're not used in a lot of scenarios, we'll describe them for completeness.

Next steps

- Metrics are how the Service Fabric Cluster Resource Manger manages consumption and capacity in the cluster. To learn more about them and how to configure them check out this article
- Affinity is one mode you can configure for your services. It is not common, but if you need it you can learn about it here
- There are many different placement rules that can be configured on your service to handle additional scenarios. You can find out about those different placement policies here

- Start from the beginning and get an Introduction to the Service Fabric Cluster Resource Manager
- To find out about how the Cluster Resource Manager manages and balances load in the cluster, check out the article on balancing load
- The Cluster Resource Manager has a lot of options for describing the cluster. To find out more about them check out this article on describing a Service Fabric cluster

Managing resource consumption and load in Service Fabric with metrics

11/15/2016 • 15 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Wike

Metrics are the generic term within Service Fabric for the resources that your services care about and which are provided by the nodes in the cluster. Generally, a metric is anything that you want to manage in order to deal with the performance of your services.

Things like Memory, Disk, CPU usage – all of these are examples of metrics. These are physical metrics, resources that correspond to physical resources on the node that need to be managed. Metrics can also be (and commonly are) logical metrics, things like "MyWorkQueueDepth" that are application-defined and which correspond to some level of resource consumption (but where the application don't really know it or know how to measure it). Most metrics that we see people use are logical metrics. There's a variety of reasons for this, but the most common is that today many of our customers write their services in managed code, and from within a given stateless service instance or stateful service replica object it is actually quite hard to measure and report your consumption of actual physical resources. The complexity of reporting your own metrics is also why we provide some default metrics out of the box.

Default metrics

Let's say that you just want to get started and don't know what resources you are going to consume or even which ones would be important to you. So you go implement and then create your services without specifying any metrics. That's fine! We'll pick some metrics for you. The default metrics that we use for you today if you don't specify any of your own are called PrimaryCount, ReplicaCount, and (somewhat vaguely, we realize) Count. The table below shows how much load for each of these metrics is associated with each service object:

METRIC	STATELESS INSTANCE LOAD	STATEFUL SECONDARY LOAD	STATEFUL PRIMARY LOAD
PrimaryCount	0	0	1
ReplicaCount	0	1	1
Count	1	1	1

Ok, so with these default metrics, what do you get? Well it turns out that for basic workloads you get a pretty good distribution of work. In this example below let's see what happens when we create one stateful service with three partitions and a target replica set size of three, and also a single stateless service with an instance count of three - you'll get something like this!

In this example we see

- Primary replicas for the stateful service are not stacked up on a single node
- Replicas for the same partition are not on the same node
- The total number of primaries and secondaries is well distributed in the cluster
- The total number of service objects (stateless and stateful) are evenly allocated on each node

Pretty good!

This works great until you start to think about it: What's the likelihood that the partitioning scheme you picked will result in perfectly even utilization by all partitions over time? Coupled with that, what's the chance that the load for a given service is constant over time, or even just the same right now? Turns, out for any serious workload the odds of all replicas being equivalent is actually rather low, so if you're interested in getting the most out of your cluster you'll probably want to start looking into custom metrics.

Realistically, you could absolutely run with just the default metrics but doing so usually means that your cluster utilization is lower than you'd like (since reporting isn't adaptive and presumes everything is equivalent); in the worst case it can also result in overscheduled nodes resulting in performance issues. We can do better with custom metrics and dynamic load reports, which we'll cover next.

Custom metrics

We've already discussed that there can be both physical and logical metrics, and that people can define their own metrics. Great! But how? Well, it's actually pretty easy! Just configure the metric and the default initial load when creating the service and you're done! Any set of metrics and default values representing how much the service is expected to consume can be configured on a per-named-service-instance basis when you're creating the service.

Note that when you start defining custom metrics you need to explicitly add back in the default metrics if you want us to use them to balance your service as well. This is because we want you to be clear about the relationship between the default metrics and your custom metrics – maybe you care about Memory consumption or WorkQueueDepth way more than you care about Primary distribution.

Let's say you wanted to configure a service which would report a metric called "Memory" (in addition to the default metrics). For memory, let's say that you've done some basic measurements and know that normally a primary replica of that service takes up 20Mb of Memory, while secondaries of that same service will take up 5Mb. You know that Memory is the most important metric in terms of managing the performance of this particular service, but you still want primary replicas to be balanced so that the loss of some node or fault domain doesn't take an inordinate number of primary replicas along with it. Other than that you'll take the defaults.

Here's what you'd do:

Code:

```
StatefulServiceDescription serviceDescription = new StatefulServiceDescription();
StatefulServiceLoadMetricDescription memoryMetric = new StatefulServiceLoadMetricDescription();
memoryMetric.Name = "MemoryInMb";
memoryMetric.PrimaryDefaultLoad = 20;
memoryMetric.SecondaryDefaultLoad = 5;
memoryMetric.Weight = ServiceLoadMetricWeight.High;

StatefulServiceLoadMetricDescription primaryCountMetric = new StatefulServiceLoadMetricDescription();
primaryCountMetric.Name = "PrimaryCount";
primaryCountMetric.PrimaryDefaultLoad = 1;
primaryCountMetric.SecondaryDefaultLoad = 0;
primaryCountMetric.Weight = ServiceLoadMetricWeight.Medium;

StatefulServiceLoadMetricDescription replicaCountMetric = new StatefulServiceLoadMetricDescription();
replicaCountMetric.Name = "ReplicaCount";
replicaCountMetric.PrimaryDefaultLoad = 1;
replicaCountMetric.SecondaryDefaultLoad = 1;
replicaCountMetric.Weight = ServiceLoadMetricWeight.Low;

StatefulServiceLoadMetricDescription totalCountMetric = new StatefulServiceLoadMetricDescription();
totalCountMetric.Name = "Count";
totalCountMetric.PrimaryDefaultLoad = 1;
totalCountMetric.SecondaryDefaultLoad = 1;
totalCountMetric.Weight = ServiceLoadMetricWeight.Low;

serviceDescription.Metrics.Add(memoryMetric);
serviceDescription.Metrics.Add(primaryCountMetric);
serviceDescription.Metrics.Add(replicaCountMetric);
serviceDescription.Metrics.Add(totalCountMetric);

await fabricClient.ServiceManager.CreateServiceAsync(serviceDescription);
```

Powershell:

```
New-ServiceFabricService -ApplicationName $applicationName -ServiceName $serviceName -ServiceTypeName
$serviceTypeName –Stateful -MinReplicaSetSize 2 -TargetReplicaSetSize 3 -PartitionSchemeSingleton –Metric
@("Memory,High,20,5”,"PrimaryCount,Medium,1,0”,"ReplicaCount,Low,1,1”,"Count,Low,1,1”)
```

(Reminder: if you just want to use the default metrics, you don't need to touch the metrics collection at all or do anything special when creating your service.)

Now that we've shown you how to define your own metrics, let's talk about the different properties that metrics can have. We've already shown them to you, but it's time to talk about what they actually mean! There are four different properties a metric can have today:

- Metric Name: This is the name of the metric. This is a unique identifier for the metric within the cluster from the Resource Manager's perspective.
- Default Load: The default load is represented differently depending on whether the service is stateless or stateful.
  - For stateless services each metric just has a single property named Default Load

- For stateful services you define
  - PrimaryDefaultLoad: The default amount of load that this service will exert for this metric as a Primary
  - SecondaryDefaultLoad: The default amount of load that this service will exert for this metric as a Secondary replica
- Weight: This is how important the metric is relative to the other configured metrics for this service.

Load

Load is the general notion of how much of a given metric is consumed by some service instance or replica on a given node.

Default load

Default load is how much load the Cluster Resource Manager should assume each service instance or replica of this service will consume until it receives any updates from the actual service instances or replicas. For simpler services, this ends up being a static definition that is never updated dynamically and hence will be used for the lifetime of the service. This works great for simple capacity planning because it's exactly what we are used to doing – dedicating certain resources to certain workloads, but the benefit is that at least now we're operating in the microservices mindset where resources aren't actually statically assigned to particular workloads and where people aren't in the decision-making loop.

We allow stateful services to specify default load for both their Primaries and Secondaries – realistically for a lot of services these numbers are different due to the different workloads executed by primary replicas and secondary replicas, and since primaries usually serve both reads and writes (as well as most of the computational burden) the default load for a primary replica is higher than for secondary replicas.

But now let's say that you've been running your service for a while and you've noticed that some instances or replicas of your service consume way more resources than others or that their consumption varies over time – maybe they're associated with a particular customer, maybe they just correspond to workloads that vary over the course of the day like messaging traffic, phone calls, or stock trades. At any rate, you notice that there's no "single number" that you can use for the load without being off by a significant amount for at least some portion of the time. You also notice that "being off" in your initial estimate results in the Cluster Resource Manager either over or under allocating resources to your service, and consequently you have nodes which are over or under utilized.

What to do? Well, your service could be reporting load on the fly!

Dynamic load

Dynamic load reports allow replicas or instances to adjust their allocation/reported use of metrics in the cluster over their lifetime. A service replica or instance that was cold and not doing any work would usually report that it was using low amounts of a given metric, while busy replicas or instances report that they are using more. This general level of churn in the cluster allows us to reorganize the service replicas and instances in the cluster on the fly in order to ensure that the get the resources they require – in effect that busy services are able to reclaim resources from other replicas or instances which are currently cold or doing less work. Reporting load on the fly can be done via the ReportLoad method, available on the ServicePartition, available as a property on the base StatefulService or StatelessService class via the Reliable Services programming model. Within your service the code would look like this:

Code:

```
this.ServicePartition.ReportLoad(new List<LoadMetric> { new LoadMetric("Memory", 1234), new
LoadMetric("metric1", 42) });
```

Services replicas or instances may only report load for the metrics that they have been configured to use. The metric list is set when each service is created and may be updated later. If a service replica or instance tries to report load for a metric that it is not currently configured to use, Service Fabric logs the report but ignores it, meaning that we won't use it when calculating or reporting on the state of the cluster. This is neat because it allows for greater experimentation – the code can measure and report on everything it knows how to, and the operator can configure, tweak, and update the resource balancing rules for that service on the fly without ever having to change the code. This can include for example, disabling a metric with a buggy report, reconfiguring the weights of metrics based on behavior, or enabling a new metric only after the code has already been deployed and validated via other mechanisms.

Mixing default load values and dynamic load reports

Does it make sense to have a default load specified for a service which is going to be reporting load dynamically? Absolutely! In this case the default load serves as an estimate until the real reports start to show up from the actual service replica or instance. This is great because it gives the Cluster Resource Manager something to work with - the default load estimate allows it to place the service instances or replicas in good places right from the start. When no default load information is provided the placement of services is effectively random at creation time, and if loads change later the Cluster Resource Manager would almost certainly have to move things around.

So let's take our previous example and see what happens when we add some custom load and then when after the service is created it gets updated dynamically. In this example, we'll use "Memory" as an example, and let's presume that we initially created the stateful service with the following command:

Powershell:

```
New-ServiceFabricService -ApplicationName $applicationName -ServiceName $serviceName -ServiceTypeName
$serviceTypeName –Stateful -MinReplicaSetSize 2 -TargetReplicaSetSize 3 -PartitionSchemeSingleton –Metric
@("Memory,High,21,11”,"PrimaryCount,Medium,1,0”,"ReplicaCount,Low,1,1”,"Count,Low,1,1”)
```

We talked about this syntax earlier (MetricName, MetricWeight, PrimaryDefaultLoad, SecondaryDefaultLoad), but we'll talk more about what the specific value for Weight means later.

Let's see what one possible cluster layout could look like:

Object	PrimaryCount	ReplicaCount	Count	Memory
Primary	1	1	1	21
Secondary	0	1	1	11
Secondary	0	1	1	11
Primary	1	1	1	21
Secondary	0	1	1	8
Secondary	0	1	1	10
Primary	1	1	1	17
Secondary	0	1	1	10
Secondary	0	1	1	9
Instance1	0	0	1	1
Instance2	0	0	1	6
Instance2	0	0	1	3

Node	PrimaryCount	ReplicaCount	Count	Memory
N1	1	1	2	22
N2	0	1	2	16
N3	1	2	2	28
N4	0	2	2	20
N5	1	1	2	24
N6	0	2	2	21

Some things that are worth noting:

- Since replicas or instances use the service's default load until they report their own load, we know that the replicas inside of partition 1 of the stateful service haven't reported load on their own
- Secondary replicas within a partition can have their own load
- Overall the metrics look pretty good, with the difference between the maximum and minimum load on a node (for memory – the custom metric we said we cared the most about) of only a factor of 1.75 (the node with the most load for the memory is N3, the least is N2, and 28/16 = 1.75) – pretty balanced!

There are some things that we still need to explain

- What determined whether a ratio of 1.75 was reasonable or not? How do we know that's good enough or if there is more work to do?
- When does balancing happen?
- What does it mean that Memory was weighted "High"?

Metric weights

Metric Weights are what allows two different services to report the same metrics but to view the importance of balancing that metric differently. For example, consider an in-memory analytics engine and a persistent database; both probably care about the "Memory" metric, but the in-memory service probably doesn't care much about the "Disk" metric – it might consume a little of it, but it is not critical to the service's performance, so it probably doesn't even report it. Being able to track the same metrics across different services is great since that's what allows the Cluster Resource Manager to track real consumption in the cluster, ensure that nodes don't go over capacity, etc.

Metric weights also allow the Cluster Resource Manager to make decisions about how to balance the cluster when there's no perfect answer (which is a lot of the time). Metrics can have four different weight levels: Zero, Low, Medium, and High. A metric with a weight of Zero contributes nothing when considering whether things are balanced or not, but its load does still contribute to things like capacity measurement.

The real impact of different metric weights in the cluster is that we arrive at different arrangements of the services since the Cluster Resource Manager has been told, in aggregate, that certain metrics are more important than others. Because it knows this, when metrics which have different weights conflict with other the Cluster Resource Manager can prefer solutions which balance the higher weighted metrics better.

Let's take a look at a simple example of some load reports and how different metric weights can result in different allocations in the cluster. In this example we see that switching the relative weight of the metrics results in the Resource Manager preferring certain solutions by creating different arrangements of services.

In this example there are four different services, all reporting different values for two different metrics A and B. In one case all the services define Metric A is the most important one (Weight = High) and MetricB as relatively unimportant (Weight = Low), and indeed we see that the Cluster Resource Manager places the services so that MetricA is better balanced (has a lower standard deviation) than MetricB. In the second case, we reverse the metric weights, and we see that the Cluster Resource Manager would probably swap services A and B in order to come up with an allocation where MetricB is better balanced than MetricA.

Global metric weights

So if ServiceA defines MetricA as most important, and ServiceB doesn't care about it at all, what's the actual weight that ends up getting used?

Well there are actually two weights we keep track of for every metric – the weight the service itself defined and the global average weight across all of the services that care about that metric. We use both these weights when calculating the scores of the solutions we generate, since it is important to ensure that both a service is balanced with regard to its own priorities, but also that the cluster as a whole is allocated correctly.

What would happen if we didn't care about both global and local balance? Well, it's trivial to construct solutions that are globally balanced but which result in very poor balance and resource allocation for individual services. In the example below let's consider the default metrics that a stateful service is configured with, PrimaryCount, ReplicaCount, and Count, and see what happens when we only consider global balance:

In the top example where we only looked at global balance, the cluster as a whole is indeed balanced – all nodes have the same count of primaries, and total replicas. However if you look at the actual impact of this allocation it's not so good: the loss of any node impacts a particular workload disproportionally, as it takes out all of its primaries. Take for example if we were to lose the first node. If this happened the three primaries for the three different partitions of the Circle service would all be lost simultaneously. Conversely, the other two services (Triangle and Hexagon) have their partitions lose a replica, which causes no disruption (other than having to recover the down replica).

In the bottom example we have distributed the replicas based on both the global and per-service balance. When calculating the score of the solution we give a majority of the weight to the global solution, but a (configurable) portion does go to ensuring that the services are balanced within themselves as much as possible as well. As a result, if we were to lose the same first node, we see that the loss of primaries (and secondaries) is distributed across all partitions of all services and the impact to each is the same.

Taking metric weights into account, the global balance is calculated based on the average of the metric weights configured for each of the services. We balance a service with regard to its own defined metric weights.

Next steps

- For more information about the other options available for configuring services check out the topic on the other Cluster Resource Manager configurations available Learn about configuring Services
- Defining Defragmentation Metrics is one way to consolidate load on nodes instead of spreading it out. To learn how to configure defragmentation, refer to this article
- To find out about how the Cluster Resource Manager manages and balances load in the cluster, check out the article on balancing load
- Start from the beginning and get an Introduction to the Service Fabric Cluster Resource Manager
- Movement Cost is one way of signaling to the Cluster Resource Manager that certain services are more expensive to move than others. To learn more about movement cost, refer to this article

Configuring and using service affinity in Service Fabric

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

Affinity is a control that is provided mainly to help ease the transition of larger monolithic applications into the cloud and microservices world. That said it can also be used in certain cases as a legitimate optimization for improving the performance of services, though this can have side effects.

Let's say you're bringing a larger app, or one that just wasn't designed with microservices in mind, to Service Fabric. This transition is actually common, and we've had several customers (both internal and external) in this situation. You start by lifting up the entire app into the environment, getting it packaged and running. Then you start breaking it down into different smaller services that all talk to each other.

Then there's an "Oops...". The "Oops" usually falls into one of these categories:

1. Some component X in the monolithic app had an undocumented dependency on component Y, and we just turned those into separate services. Since these are now running on different nodes in the cluster, they're broken.
2. These things communicate via (local named pipes | shared memory | files on disk) but I really need to be able to update it independently to speed things up a bit. I'll remove the hard dependency later.
3. Everything is fine, but it turns out that these two components are actually very chatty/performance sensitive. When they moved them into separate services overall application performance tanked or latency increased. As a result, the overall application is not meeting expectations.

In these cases we don't want to lose our refactoring work, and don't want to go back to the monolith, but we do need some sense of locality. This will persist either until we can redesign the components to work naturally as services, or until we can solve the performance expectations some other way, if possible.

What to do? Well you could try turning on affinity.

How to configure affinity

To set up affinity, you define an affinity relationship between two different services. You can think of affinity as "pointing" one service at another and saying "This service can only run where that service is running." Sometimes we refer to affinity as a parent/child relationship (where you point the child at the parent). Affinity ensures that the replicas or instances of one service are placed on the same nodes as the replicas or instances of another.

```
ServiceCorrelationDescription affinityDescription = new ServiceCorrelationDescription();
affinityDescription.Scheme = ServiceCorrelationScheme.Affinity;
affinityDescription.ServiceName = new Uri("fabric:/otherApplication/parentService");
serviceDescription.Correlations.Add(affinityDescription);
await fabricClient.ServiceManager.CreateServiceAsync(serviceDescription);
```

Different affinity options

Affinity is represented via one of several correlation schemes, and has two different modes. The most common mode of affinity is what we call NonAlignedAffinity. In NonAlignedAffinity the replicas or instances of the different services are placed on the same nodes. The other mode is AlignedAffinity. Aligned Affinity is useful only with

stateful services. Configuring two stateful services to have aligned affinity ensures that the primaries of those services are placed on the same nodes as each other. It also causes each pair of secondaries for those services to be placed on the same nodes. It is also possible (though less common) to configure NonAlignedAffinity for stateful services. For NonAlignedAffinity the different replicas of the two stateful services would be collocated on the same nodes, but no attempt would be made to align their primaries or secondaries.

Best effort desired state

There are a few differences between affinity and monolithic architectures. Many of them are because an affinity relationship is best effort. The services in an affinity relationship are fundamentally different entities that can fail and be moved independently. There are also causes for why an affinity relationship could break. For example, capacity limitations where only some of the service objects in the affinity relationship can fit on a given node. In these cases even though there's an affinity relationship in place, it can't be enforced due to the other constraints. If it is possible to enforce all the other constraints and affinity at a later time the violation of the affinity constraint will be automatically corrected.

Chains vs. stars

Today we aren’t able to model chains of affinity relationships. What this means is that a service that is a child in one affinity relationship can’t be a parent in another affinity relationship. If you want to model this type of relationship, you effectively have to model it as a star, rather than a chain. In order to do this, the bottommost child would be parented to the “middle” child’s parent instead. Depending on the arrangement of your services, this may require creating a "placeholder" service to serve as the parent for multiple children.

Another thing to note about affinity relationships today is that they are directional. This means that the “affinity” rule only enforces that the child is where the parent is. If for example the parent suddenly fails over to another

node then the Cluster Resource Manager doesn't actually think there's anything wrong until it notices that the child is not located with a parent; the relationship is not immediately enforced.

Partitioning support

The final thing to notice about affinity is that affinity relationships aren't supported where the parent is partitioned. This is something that we may support eventually, but today it is not allowed.

Next steps

- For more information about the other options available for configuring services check out the topic on the other Cluster Resource Manager configurations available Learn about configuring Services
- Many reasons where people use affinity, such as limiting services to a small set of machines and trying to aggregate the load of a collection of services, are better supported through Application Groups. Check out Application Groups

Placement policies for service fabric services

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

There are many different additional rules that you may end up caring about if your Service Fabric cluster is spanned across a geographic distances, say multiple datacenters or Azure regions, or if your environment spans multiple areas of geopolitical control (or some other case where you have legal or policy boundaries you care about, or the distances involved have actual performance/latency impact). Most of these could be configured via node properties and placement constraints, but some are more complicated. To make things simpler we provide these additional commmands. Just like with other placement constraints, placement policies can be configured on a per-named service instance basis.

Specifying invalid domains

The InvalidDomain placement policy allows you to specify that a particular Fault Domain is invalid for this workload. This policy ensures that a particular service never runs in a particular area, for example for geopolitical or corporate policy reasons. Multiple invalid domains may be specified via separate policies.

Code:

```
ServicePlacementInvalidDomainPolicyDescription invalidDomain = new
ServicePlacementInvalidDomainPolicyDescription();
invalidDomain.DomainName = "fd:/DCEast"; //regulations prohibit this workload here
serviceDescription.PlacementPolicies.Add(invalidDomain);
```

Powershell:

```
New-ServiceFabricService -ApplicationName $applicationName -ServiceName $serviceName -ServiceTypeName
$serviceTypeName –Stateful -MinReplicaSetSize 2 -TargetReplicaSetSize 3 -PartitionSchemeSingleton -
PlacementPolicy @("InvalidDomain,fd:/DCEast”)
```

Specifying required domains

The required domain placement policy requires that all of the stateful replicas or stateless service instances for the service be present in the specified domain. Multiple required domains can be specified via separate policies.

Code:

```
ServicePlacementRequiredDomainPolicyDescription requiredDomain = new
ServicePlacementRequiredDomainPolicyDescription();
requiredDomain.DomainName = "fd:/DC01/RK03/BL2";
serviceDescription.PlacementPolicies.Add(requiredDomain);
```

Powershell:

```
New-ServiceFabricService -ApplicationName $applicationName -ServiceName $serviceName -ServiceTypeName
$serviceTypeName –Stateful -MinReplicaSetSize 2 -TargetReplicaSetSize 3 -PartitionSchemeSingleton -
PlacementPolicy @("RequiredDomain,fd:/DC01/RK03/BL2")
```

Specifying a preferred domain for the primary replicas

The Preferred Primary Domain is an interesting control, since it allows selection of the fault domain in which the primary should be placed if it is possible to do so. When everything is healthy the primary will end up in this domain. Should the domain or the primary replica fail or be shut down for some reason the Primary will be migrated to some other location. If this location isn't in the preferred domain, then when possible the Cluster Resource Manager will move it back to the preferred domain. Naturally this setting only makes sense for stateful services. This policy is most useful in clusters which are spanned across Azure regions or multiple datacenters. In these situations you're using all the locations for redundancy, but would prefer that the primary replicas be placed in a certain location in order to provide lower latency for operations which go to the primary (writes and also by default all reads are served by the primary).

```
ServicePlacementPreferPrimaryDomainPolicyDescription primaryDomain = new
ServicePlacementPreferPrimaryDomainPolicyDescription();
primaryDomain.DomainName = "fd:/EastUS/";
serviceDescription.PlacementPolicies.Add(invalidDomain);
```

Powershell:

```
New-ServiceFabricService -ApplicationName $applicationName -ServiceName $serviceName -ServiceTypeName
$serviceTypeName –Stateful -MinReplicaSetSize 2 -TargetReplicaSetSize 3 -PartitionSchemeSingleton -
PlacementPolicy @("PreferredPrimaryDomain,fd:/EastUS")
```

Requiring replicas to be distributed among all domains and disallowing packing

Another policy you can specify is to require replicas to always be distributed among the available fault domains. This will happen by default in most cases where the cluster is healthy, however there are degenerate cases where replicas for a given partition may end up temporarily packed into a single fault or upgrade domain. For example, let's say that although the cluster has 9 nodes in 3 fault domains (0, 1, and 2), and your service has 3 replicas, the nodes that were being used for those replicas in Fault Domains 1 and 2 went down, and due to capacity issues none of the other nodes in those domains were valid. If Service Fabric were to build replacements for those replicas, the Cluster Resource Manager would have to put them in Fault Domain 0, but that creates a situation where the Fault Domain constraint is being violated. It also increases the chance that the whole replica set could be lost (if FD 0 were to be permananently lost). (For more information on constraints and constraint priorities generally, check out this topic)

If you've ever seen a health warning like "The Load Balancer has detected a Constraint Violation for this Replica:fabric:/ Secondary Partition is violating the Constraint: FaultDomain" you've hit this condition or something like it. Usually these situations are transient (the nodes don't stay down long, or if they do and we need to build replacements there are other nodes in the correct fault domains which are valid), but there are some workloads

that would rather trade availability for the risk of losing all their replicas. We can do this by specifying the "RequireDomainDistribution" policy, which will guarantee that no two replicas from the same partition are ever allowed in the same fault or upgrade domain.

Some workloads would rather have the target number of replicas (copies of state) at all times (betting against total domain failures and knowing that they can usually recover local state), whereas others would rather take the downtime earlier than risk the correctness and dataloss concerns. Since most production workloads run with more than 3 replicas, the default is to not require domain distribution and let balancing and failover handle cases normally even if that means that temporarily a domain has multiple replicas packed into it.

Code:

```
ServicePlacementRequireDomainDistributionPolicyDescription distributeDomain = new
ServicePlacementRequireDomainDistributionPolicyDescription();
serviceDescription.PlacementPolicies.Add(distributeDomain);
```

Powershell:

```
New-ServiceFabricService -ApplicationName $applicationName -ServiceName $serviceName -ServiceTypeName
$serviceTypeName –Stateful -MinReplicaSetSize 2 -TargetReplicaSetSize 3 -PartitionSchemeSingleton -
PlacementPolicy @("RequiredDomainDistribution")
```

Now, would it be possible to use these configurations for services in a cluster which was not geographically spanned? Sure you could! But there’s not a great reason too – especially the required, invalid, and preferred domain configurations should be avoided unless you’re actually running a geographically spanned cluster - it doesn't make any sense to try to force a given workload to run in a single rack, or to prefer some segment of your local cluster over another unless there's different types of hardware or workload segmentation going on, and those cases can be handled via normal placement constraints.

Next steps

- For more information about the other options available for configuring services check out the topic on the other Cluster Resource Manager configurations available Learn about configuring Services

Cluster resource manager integration with Service Fabric cluster management

11/15/2016 • 11 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Milan Micic

The Service Fabric Cluster Resource Manager isn't the main component of Service Fabric that handles management operations (like application upgrades) but it is involved. The first way that the Cluster Resource Manager helps with management is by tracking the desired state of the cluster and the services inside it from a resourcing and balance perspective and sending out health reports when it cannot put the cluster into the desired configuration (and example would be if there is insufficient capacity, or conflicting rules about where a service should be placed). Another piece of integration has to do with how upgrades work: during upgrades the Cluster Resource Manager alters its behavior. We'll talk about both of these below.

Health integration

The Cluster Resource Manager constantly tracks the rules you have defined for your services, as well as the capacities available on the nodes and in the cluster, and emits health warnings and errors if it cannot satisfy those rules or if there is insufficient capacity. For example, if a node is over capacity and the Cluster Resource Manager can't correct the situation it will emit a health warning indicating which node is over capacity, and for which metrics.

Another example of a time when you'll see the Resource Manager emit health warnings is if you have defined a placement constraint (such as "NodeColor == Blue") and the Resource Manager detects a violation of that constraint. We do this both for custom constraints as well as the default constraints (like fault and upgrade domain constraints) that are enforced automatically.

Here's an example of one such health report. In this case the health report is for one of the system service's partitions because the replicas of that partition are temporarily packed into too few Upgrade Domains, like could happen if there were a string of failures:

```
PS C:\Users\User > Get-WindowsFabricPartitionHealth -PartitionId '00000000-0000-0000-0000-000000000001'


PartitionId           : 00000000-0000-0000-0000-000000000001
AggregatedHealthState : Warning
UnhealthyEvaluations  :
                        Unhealthy event: SourceId='System.PLB',
Property='ReplicaConstraintViolation_UpgradeDomain', HealthState='Warning', ConsiderWarningAsError=false.

ReplicaHealthStates   :
                        ReplicaId             : 130766528804733380
                        AggregatedHealthState : Ok

                        ReplicaId             : 130766528804577821
                        AggregatedHealthState : Ok

                        ReplicaId             : 130766528854889931
                        AggregatedHealthState : Ok

                        ReplicaId             : 130766528804577822
                        AggregatedHealthState : Ok

                        ReplicaId             : 130837073190680024
                        AggregatedHealthState : Ok

HealthEvents          :
                        SourceId              : System.PLB
                        Property              : ReplicaConstraintViolation_UpgradeDomain
                        HealthState           : Warning
                        SequenceNumber        : 130837100116930204
                        SentAt                : 8/10/2015 7:53:31 PM
                        ReceivedAt            : 8/10/2015 7:53:33 PM
                        TTL                   : 00:01:05
                        Description           : The Load Balancer has detected a Constraint Violation for this
Replica: fabric:/System/FailoverManagerService Secondary Partition 00000000-0000-0000-0000-000000000001 is
                        violating the Constraint: UpgradeDomain Details: Node --
3d1a4a68b2592f55125328cd0f8ed477  Policy -- Packing
                        RemoveWhenExpired     : True
                        IsExpired             : False
                        Transitions           : Ok->Warning = 8/10/2015 7:13:02 PM, LastError = 1/1/0001
12:00:00 AM
```

Here's what this health message is telling us is:

1. All the replicas themselves are healthy (this is Service Fabric's first priority)
2. That the Upgrade Domain distribution constraint is currently being violated (meaning that a particular Upgrade Domain has more of the replicas for this partition than it should)
3. Which node contains the replica causing the violation (The node with ID: 3d1a4a68b2592f55125328cd0f8ed477)
4. When this all happened (8/10/2015 7:13:02 PM)

This is great data for an alert that fires in production to let you know something has gone wrong and you probably want to go take a look. In this case, for example, we'd want to see if we can figure out why the Resource Manager didn't feel like it had any choice but to pack the replicas into the Upgrade Domain. This could be because all of the nodes in the other Upgrade Domains were down and there weren't enough spare other domains, or if there were enough domains up something else which caused the nodes in those other Upgrade Domains to be invalid (like some placement policy on the service or insufficient capacity, for example).

Let's say however that you want to create a service, or the Resource Manager is trying to find a place to place some services, but there doesn't appear to be any solutions that work. This could be for many reasons, but usually it is due to one of the two following conditions:

1. Some transient condition has made it impossible to place this service instance or replica correctly
2. The service's requirements are misconfigured in a way that causes its requirements to be unsatisfiable.

In each of these conditions you'll see a health report from the Cluster Resource Manager that provides information to help you determine what is going on and why the service can't be placed. We call this process the "Constraint Elimination Sequence". During it, the system walks through the configured constraints affecting the service and records what they eliminate. This way when services aren't able to be placed, you can see which nodes were eliminated and why.

Constraint types

Let's talk about each of the different constraints you can see in these health reports and what it is checking for. Note that most of the time you won't see some these constraints eliminate nodes since the constraints are at the soft or optimize level by default (there's more about constraint priorities further along in this article). You could however see health messages related to these constrains if they are configured as hard constraints or in the rare cases that they do cause nodes to be eliminated, so we present them here for completeness:

- ReplicaExclusionStatic and ReplicaExclusionDynamic – This is an internal constraint that indicates that during the search we ran into a situation where two stateful replicas or stateless instances from the same partition would have to be placed on the same node (which isn't allowed). ReplicaExclusionStatic and ReplicaExclusionDynamic are almost exactly the same rule. The ReplicaExclusionDynamic constraint says "we couldn't place this replica here because the only proposed solution already had placed a replica here". This is different from the ReplicaExclusionStatic exclusion which indicates not a proposed conflict but an actual one – there is a replica already on the node. Is this confusing? Yes. Does it matter a lot? No. Suffice it to say that if you are seeing a constraint elimination sequence containing either the ReplicaExclusionStatic or ReplicaExclusionDynamic constraint that the Cluster Resource Manager thinks that there aren't enough nodes to place all of the replicas. The further constraints can usually tell us how we're ending up with too few in the first place.
- PlacementConstraint: If you see this message, it means that we eliminated some nodes because they didn't match the service's placement constraints. We trace out the currently configured placement constraints as a part of this message. This is usually normal if you have any placement constraints provided, however if there is a bug in the placement constraint causing too many nodes to be eliminated this is where you could see that result.
- NodeCapacity: If you see this constraint it means that we couldn't place the replicas on the indicated nodes because doing so would cause the node to go over capacity
- Affinity: This constraint indicates that we couldn't place the replica on the affected nodes since it would cause a violation of the affinity constraint.
- FaultDomain & UpgradeDomain: This constraint eliminates nodes if placing the replica on the indicated nodes would cause packing in a particular fault or upgrade domain. Several examples discussing this constraint are presented in the topic on fault and upgrade domain constraints and resulting behavior
- PreferredLocation: You shouldn't normally see this constraint causing nodes to get removed from the solution since it is optimization only by default. Further, the preferred location constraint is usually only present during upgrades (when it is used to move replicas back to where they were when the upgrade started), however it is possible.

Constraint priorities

In all of these constraints, you may have been thinking "Hey – I think that placement constraints are the most important thing in my system. I'm willing to violate other constraints, even things like affinity and capacity, if it ensures that the placement constraints aren't ever violated."

Well it turns out we can do that! Constraints can be configured with a few different levels of enforcement, but they boil down to "hard" (0), "soft" (1), "optimization" (2), and "off" (-1). Most of the constraints we've defined as hard by default (since, for example, most people don't normally think about capacity as something they are willing to

relax), and almost all are either hard or soft. However in advanced situations these can be changed. For example, what if you wanted to ensure that affinity would always be violated in order to solve node capacity issues, you could priority of the affinity constraint to “soft” (1) and leave the capacity constraint set to “Hard” (0). The different constraint priorities are also why you'll see some constraint violation warnings more often than others - because there are certain constraints that we're willing to relax (violate) temporarily. Note that these levels don't really mean that a given constraint will or will not always be violated, just that there's an order in which they are preferentially enforced so that we can make the right tradeoffs if it is impossible to satisfy all of them.

The configuration and default priority values for the different constraints are listed below:

ClusterManifest.xml

```
<Section Name="PlacementAndLoadBalancing">
    <Parameter Name="PlacementConstraintPriority" Value="0" />
    <Parameter Name="CapacityConstraintPriority" Value="0" />
    <Parameter Name="AffinityConstraintPriority" Value="0" />
    <Parameter Name="FaultDomainConstraintPriority" Value="0" />
    <Parameter Name="UpgradeDomainConstraintPriority" Value="1" />
    <Parameter Name="PreferredLocationConstraintPriority" Value="2" />
</Section>
```

You’ll notice here that there are constraints defined for Upgrade and Fault domains, and also that Upgrade Domain constraint is Soft. Also there’s this weird “PreferredLocation” constraint with a priority. What is all this stuff?

First off, we model the desire to keep services spread out amongst fault and upgrade domains as a constraint inside the Resource Manager’s engine. Historically there have been a couple of times where we needed either to get really strict about placement with regard to fault and upgrade domains, and also a couple cases where there was some issue where we just needed to ignore them entirely (though briefly!), so generally we’ve been glad both for that design choice and for the flexibility of the constraint infrastructure. Most of the time though, Upgrade domain constraint sits as a soft constraint, meaning that if the Resource Manager temporarily needs to pack a couple replicas into an upgrade domain in order to deal with, say, an upgrade going on, or a bunch of concurrent failures or other constraint violations (from the hard constraints) then that is ok. This normally doesn’t happen unless there are a lot of failures or other churn in the system preventing correct placement, and if the environment is configured correctly the stable state is always upgrade domain being fully respected.

The PreferredLocation constraint is a little different, and hence it is the only constraint set to “Optimization”. We use this constraint while upgrades in flight in order to try to prefer putting services back where we found them before the upgrade. There’s all sorts of reasons why this may not work in practice, but it’s a nice optimization, so here it sits. We’ll talk more about it when we talk about how the Cluster Resource Manager assists with upgrades.

Upgrades

The Cluster Resource Manager also helps during application and cluster upgrades in order to ensure that the upgrade goes smoothly, but also to ensure that the rules and performance of the cluster are not compromised.

Keep enforcing the rules

The main thing to be aware of is that the rules – the strict constraints around things like placement constraints are still enforced during upgrades. You’d think that this goes without saying, but we’re saying it anyway just to be explicit about it. The positive side of this is that you can ensure that if you wanted to be sure that certain workloads don’t run on certain nodes those rules will be enforced even during the upgrade automatically. If your environment is highly constrained, this can cause upgrades to take a long time since there are only a few options for where a service can go if it (or the node it sits on) needs to be brought down for an update.

Smart replacements

When an upgrade starts the Resource Manager takes a snapshot of the current arrangement of the cluster and

attempts to return things to that state when the upgrade is done. The reasoning behind this is simple – first it ensures that there are only a couple transitions for each service replica or instance as a part of the upgrade (the move out of the affected node and the move back in). Secondly it ensures that the upgrade itself doesn't have much impact on the layout of the cluster; if the cluster was arranged well before the upgrade it will be arranged well after it, or at least no worse.

Reduced churn

Another thing that happens during upgrades is that the Cluster Resource Manager turns off balancing for the entity being upgraded. So if you have two different application instances and then start an upgrade on one of them, then balancing is paused for that application instance, but not the other one. Preventing reactive balancing prevents unnecessary reactions to the upgrade itself ("Oh no! An empty node! Better fill it with all sorts of stuff!") and consequently prevents in a lot of extra movements for services in the cluster that would just have to be undone when the services need to move back to the nodes after the upgrade is finished. If the upgrade in question is a Cluster upgrade, then the entire cluster is paused for balancing for the duration of the upgrade (constraint checks – ensuring the rules are enforced – stay active, just proactive rebalancing is disabled).

Relaxed rules

One of the things that comes up during upgrades is generally that you want the upgrade to complete even if the cluster is overall rather constrained or full. During upgrades being able to manage the capacity of the cluster is even more important than usual as there is typically between 5 and 20 percent of the capacity down at a time as the upgrade rolls through the cluster, and that workload usually has to go somewhere. This is where the notion of buffered capacities really comes into play – while the buffered capacity is respected during normal operation (leaving some overhead), the Cluster Resource Manager will fill up to the total capacity (taking up the buffer) during upgrades.

Next steps

- Start from the beginning and get an Introduction to the Service Fabric Cluster Resource Manager

Defragmentation of metrics and load in Service Fabric

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

The Service Fabric Cluster Resource Manager mainly is concerned with balancing in terms of distributing the load – making sure that all of the nodes in the cluster are equally utilized. This is usually the safest and smartest layout in terms of surviving failures since it makes sure that any given failure doesn't take out the some large percentage of a given workload. The Service Fabric Cluster Resource Manager does support a different strategy as well, which is defragmentation. Defragmentation generally means that instead of trying to distribute the utilization of a metric across the cluster, we should actually try to consolidate it. This is a fortunate inversion of our normal strategy – instead of optimizing the cluster based on minimizing the average standard deviation of metric load for a given metric, we start optimizing for increases in deviation. But why would you want this strategy?

Well, if you've spread the load out evenly among the nodes in the cluster then you've eaten up some of the resources that the nodes have to offer. Normally this isn't a problem, but sometimes some workloads create services which are exceptionally large and consume the vast majority of a node – say 75% to 95% of a node's resources would end up dedicated to a single service instance or replica. This isn't a problem, the Cluster Resource Manager will detect at service creation time that it needs to reorganize the cluster in order to make room for this large workload and set about making it happen, but in the meantime that workload has to wait to be scheduled in the cluster.

Given that the scheduling of new workloads is usually at least a little latency sensitive, if we don't do anything differently we can sometimes blow right by those SLAs if there's a lot of services and state to move around, particularly if workloads in the cluster are generally large (and hence taking longer to move around in the cluster). Indeed, when we measured creation times in simulations based on real cluster data, we saw that if services were large enough and the cluster was fairly utilized that the creation of those large services would be slowed down, and that we could improve this by introducing the policy of defragmentation metrics.

Just like file creation or access could get slowed down if a computer's hard disk was fragmented and could be sped up by defragmenting the drive so that there were larger contiguous blocks available, you can configure defragmentation metrics to have the Cluster Resource Manager to proactively try to condense the load of the services into fewer nodes so that there is (almost) always room for even large services, enabling them to be created quickly. Most people won't need this, because services should usually be small and hence it's not hard to find room for them, but if you have large services and need them created quickly (and are willing to accept the other tradeoffs such as increased impactfulness of failures and some resources being left unutilized while they wait for workloads to be scheduled) then the defragmentation strategy is for you.

The diagram below gives a visual representation of two different clusters, one which is defragmented and one which is not. In the balanced case, consider the movements which would be necessary to place one of the largest service objects, if a new one were to be created, compared to the defragmented cluster, where it could be immediately placed on nodes 4 or 5.

Defragmentation pros and cons

So what are those other conceptual tradeoffs? We recommend thorough measurement of your workloads before turning on defragmentation metrics. Here's a quick table of things to think about:

DEFRAGMENTATION PROS	DEFRAGMENTATION CONS
Allows faster creation of large services	Concentrates load onto fewer nodes, increasing contention
Enables lower data movement during creation	Failures can impact more services and cause more churn
Allows rich description of requirements and reclamation of space	More complex overall Resource Management configuration

You can mix defragmented and normal metrics in the same cluster and the Resource Manager will do it's best to ensure that you get a layout that consolidates as much of the defragmentation metrics as it can while trying to spread out the rest. The exact results you'll get will depend on the number of balancing metrics compared to the number of defragmentation metrics and their weights, current loads, etc.

Configuring defragmentation metrics

Configuring defragmentation metrics is a global decision in the cluster, and individual metrics can be selected for defragmentation:

ClusterManifest.xml:

```
<Section Name="DefragmentationMetrics">
    <Parameter Name="Disk" Value="true" />
    <Parameter Name="CPU" Value="false" />
</Section>
```

Next steps

- The Cluster Resource Manager has a lot of options for describing the cluster. To find out more about them check out this article on describing a Service Fabric cluster
- Metrics are how the Service Fabric Cluster Resource Manger manages consumption and capacity in the cluster. To learn more about them and how to configure them check out this article

Balancing your service fabric cluster

11/15/2016 • 8 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ashish Kishan Mansukhani

The Service Fabric Cluster Resource Manager allows reporting dynamic load, reacting to changes in the cluster, correcting constraint violations, and rebalancing the cluster if necessary. But how often does it do these things, and what triggers it? There are several controls related to this.

The first set of controls around balancing are a set of timers. These timers govern how often the Cluster Resource Manager examines the state of the cluster for things that need to be addressed. There are three different categories of work, each with their own corresponding timer. They are:

1. Placement – this stage deals with placing any stateful replicas or stateless instances which are missing. This covers both new services and handling stateful replicas or stateless instances which have failed and need to be recreated. Deleting and dropping replicas or instances is also handled here.
2. Constraint Checks – this stage checks for and corrects violations of the different placement constraints (rules) within the system. Examples of rules are things like ensuring that nodes are not over capacity and that a service's placement constraints (more on these later) are met.
3. Balancing – this stage checks to see if proactive rebalancing is necessary based on the configured desired level of balance for different metrics, and if so attempts to find an arrangement in the cluster that is more balanced.

Configuring Cluster Resource Manager Steps and Timers

Each of these different types of corrections the Cluster Resource Manager can make is controlled by a different timer which governs its frequency. So for example, if you only want to deal with placing new service workloads in the cluster every hour (to batch them up), but want regular balancing checks every few seconds, you can configure that behavior. When each timer fires, the task is scheduled. By default the Resource Manager scans its state and applies updates (batching all the changes that have occurred since the last scan, like noticing that a node is down) every 1/10th of a second, sets the placement and constraint check flags every second, and the balancing flag every 5 seconds.

ClusterManifest.xml:

```
<Section Name="PlacementAndLoadBalancing">
    <Parameter Name="PLBRefreshGap" Value="0.1" />
    <Parameter Name="MinPlacementInterval" Value="1.0" />
    <Parameter Name="MinConstraintCheckInterval" Value="1.0" />
    <Parameter Name="MinLoadBalancingInterval" Value="5.0" />
</Section>
```

Today we only perform one of these actions at a time, sequentially (that's why we refer to these configurations as "minimum intervals")). This is so that, for example, we've already responded to any pending requests to create new replicas before we move on to balancing the cluster. As you can see by the default time intervals specified, we can scan and check for anything we need to do very frequently, meaning that the set of changes we make at the end of each step is usually smaller: we're not scanning through hours of changes in the cluster and trying to correct them all at once, we are trying to handle things more or less as they happen but with some batching when many things happen at the same time. This makes the Service Fabric resource manager very responsive to things that happen in the cluster.

While most of these tasks are straightforward (if there are constraint violations, fix them, if there are services to be created, create them), the Cluster Resource Manager also needs some additional information to determine if the cluster imbalanced. For that we have two other pieces of configuration: *Balancing Thresholds* and *Activity Thresholds*.

Balancing thresholds

A Balancing Threshold is the main control for triggering proactive rebalancing (remember that the timer is just for how often the Cluster Resource Manager should check - it doesn't mean that anything will happen). The Balancing Threshold defines how imbalanced the cluster needs to be for a specific metric in order for the Cluster Resource Manager to consider it imbalanced and trigger balancing.

Balancing Thresholds are defined on a per-metric basis as a part of the cluster definition. For more information on metrics check out this article.

ClusterManifest.xml

```
<Section Name="MetricBalancingThresholds">
  <Parameter Name="MetricName1" Value="2"/>
  <Parameter Name="MetricName2" Value="3.5"/>
</Section>
```

The Balancing Threshold for a metric is a ratio. If the amount of load on the most loaded node divided by the amount of load on the least loaded node exceeds this number, then the cluster is considered imbalanced and balancing will be triggered the next time the Cluster Resource Manager checks (by default, ever 5 seconds, as governed by the MinLoadBalancingInterval, shown above).

In this simple example each service is consuming one unit of some metric. In the top example, the maximum load on a node is 5 and the minimum is 2. Let's say that the balancing threshold for this metric is 3. Therefore, in the top example, the cluster is considered balanced and no balancing will be triggered when the Cluster Resource Manager checks (since the ratio in the cluster is 5/2 = 2.5 and that is less than the specified balancing threshold of 3).

In the bottom example, the max load on a node is 10, while the minimum is 2, resulting in a ratio of 5. This puts the cluster over the designed balancing threshold of 3 for that metric. As a result, a global rebalancing run will be scheduled next time the balancing timer fires. Note that just because a balancing search is kicked off doesn't mean anything will move - sometimes the cluster is imbalanced but the situation can't be improved - but in a situation like this one (at least by default) some the load will almost certainly be distributed to Node3. Note that since we are not using a greedy approach some load could also be distributed to Node2 since that would result in minimization of the overall differences between nodes, but we would expect that the majority of the load would flow to Node3.

Note that getting below the balancing threshold is not an explicit goal – Balancing Thresholds are just a *trigger* that tells the Cluster Resource Manager that it should look into the cluster to determine what improvements it can make, if any.

Activity thresholds

Sometimes, although nodes are relatively imbalanced, the *total* amount of load in the cluster is low. This could be just because of the time of day, or because the cluster is new and just getting bootstrapped. In either case, you may not want to spend time balancing the cluster because there's actually very little to be gained – you'll just be spending network and compute resources to move things around, without making any absolute difference. Because we want to avoid doing this, there's another control inside of the Resource Manager, known as Activity Thresholds, which allows you to specify some absolute lower bound for activity – if no node has at least this much load then balancing will not be triggered even if the Balancing Threshold is met.

As an example let's say that we have reports with the following totals for consumption on these nodes. Let's also say that we retain our Balancing Threshold of 3 for this metric, but now we also have an Activity Threshold of 1536. In the first case, while the cluster is imbalanced per the Balancing Threshold no node meets that minimum Activity Threshold, so we leave things alone. In the bottom example, Node1 is way over the Activity Threshold, so balancing will be performed (since both the Balancing Threshold and the Activity Threshold for the metric are exceeded)

Just like Balancing Thresholds, Activity Thresholds are defined per-metric via the cluster definition:

ClusterManifest.xml

```
<Section Name="MetricActivityThresholds">
  <Parameter Name="Memory" Value="1536"/>
</Section>
```

Note that balancing and activity thresholds are both tied to the metric - balancing will only be triggered if both balancing and activity thresholds are exceeded for the same metric. Thus, if we exceed the Balancing Threshold for Memory and the Activity Threshold for CPU, balancing will not trigger as long as the remaining thresholds (Balancing Threshold for CPU and Activity Threshold for Memory) are not exceeded.

Balancing services together

Something that's interesting to note is that whether the cluster is imbalanced or not is a cluster-wide decision, but the way we go about fixing it is moving individual service replicas and instances around. This makes sense, right? If memory is stacked up on one node, multiple replicas or instances could be contributing to it, so it could require moving any of the stateful replicas or stateless instances that use the affected, imbalanced metric.

Occasionally though a customer will call us up or file a ticket saying that a service that wasn't imbalanced got moved. How could it happen that a service gets moved around even if all of that service's metrics were balanced, even perfectly so, at the time of the other imbalance? Let's see!

Take for example four services, Service1, Service2, Service3, and Service4. Service1 reports against metrics Metric1 and Metric2, Service2 against Metric2 and Mmetric3, Service3 against Metric3 and Metric4, and Service4 against some metric Metric99. Surely you can see where we're going here. We have a chain! From the perspective of the Cluster Resource Manager, we don't really have four independent services, we have a bunch of services that are related (Service1, Service2, and Service3) and one that is off on its own.

So it is possible that an imbalance in Metric1 can cause replicas or instances belonging to Service3 to move around. Usually these movements are pretty limited, but can be larger depending on exactly how imbalanced Metric1 got and what changes were necessary in the cluster in order to correct it. We can also say with certainty that an imbalance in Metrics 1, 2, or 3 will never cause movements in Service4 – there'd be no point since moving the replicas or instances belonging to Service4 around can do absolutely nothing to impact the balance of Metrics 1, 2, or 3.

The Cluster Resource Manager automatically figures out what services are related, since services may have been added, removed, or had their metric configuration change – for example, between two runs of balancing Service2 may have been reconfigured to remove Metric2. This breaks the chain between Service1 and Service2. Now instead of two groups of services, you have three:

Next steps

- Metrics are how the Service Fabric Cluster Resource Manger manages consumption and capacity in the cluster. To learn more about them and how to configure them check out this article
- Movement Cost is one way of signaling to the Cluster Resource Manager that certain services are more expensive to move than others. To learn more about movement cost, refer to this article
- The Cluster Resource Manager has several throttles that you can configure to slow down churn in the cluster. They're not normally necessary, but if you need them you can learn about them here

Throttling the behavior of the Service Fabric Cluster Resource Manager

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

Even if you've configured the Cluster Resource Manager correctly, the cluster can get disrupted. For example there could be simultaneous node or fault domain failures - what would happen if that occurred during an upgrade? The Resource Manager will try its best to fix everything, but in times like this you may want to consider a backstop so that the cluster itself has a chance to stabilize (the nodes which are going to come back do, the network conditions heal themselves, corrected bits get deployed). To help with these sorts of situations, the Service Fabric Cluster Resource Manager does include several throttles. Note that these throttles are fairly disruptive and generally shouldn't be used unless there's been some careful math done around the amount of parallel work that can actually be done in the cluster, as well as a frequent need to respond to these sorts of (ahem) unplanned macroscopic reconfiguration events (AKA: "Very Bad Days").

Generally, we recommend avoiding very bad days through other options (like regular code updates and avoiding overscheduling the cluster to begin with) rather than throttling your cluster to prevent it from using resources when it is trying to fix itself). The throttles do have default values that we've found through experience to be ok defaults, but you should probably take a look and tune them to your expected actual load. While not overly constraining or loading the cluster is a best practice you may determine that there are cases which (until you can remedy them) where you need to have a couple of throttles in place, even if it means the cluster will take longer to stabilize.

Configuring the throttles

The throttles that are included by default are:

- GlobalMovementThrottleThreshold – this controls the total number of movements in the cluster over some time (defined as the GlobalMovementThrottleCountingInterval, value in seconds)
- MovementPerPartitionThrottleThreshold – this controls the total number of movements for any service partition over some time (the MovementPerPartitionThrottleCountingInterval, value in seconds)

```
<Section Name="PlacementAndLoadBalancing">
     <Parameter Name="GlobalMovementThrottleThreshold" Value="1000" />
     <Parameter Name="GlobalMovementThrottleCountingInterval" Value="600" />
     <Parameter Name="MovementPerPartitionThrottleThreshold" Value="50" />
     <Parameter Name="MovementPerPartitionThrottleCountingInterval" Value="600" />
</Section>
```

Be aware that most of the time we've seen customers use these throttles it has been because they were already in a resource constrained environment (such as limited network bandwidth into individual nodes or disks which weren't up to the requirements of parallel replica builds which were being placed on them) which meant that such operations wouldn't succeed or would be slow anyway. In these situations customers were comfortable knowing that they were potentially extending the amount of time it would take the cluster to reach a stable state, including knowing that they could end up running at lower overall reliability while they were throttled.

Next steps

- To find out about how the Cluster Resource Manager manages and balances load in the cluster, check out the article on balancing load
- The Cluster Resource Manager has a lot of options for describing the cluster. To find out more about them check out this article on describing a Service Fabric cluster

Service movement cost for influencing Cluster Resource Manager choices

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Matt • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • ShawnJackson

An important factor to consider when you're trying to determine what changes to make to a cluster and the score of a solution is the overall cost of achieving that solution.

Moving service instances or replicas costs CPU time and network bandwidth at a minimum. For stateful services, it also costs the amount of space on disk that you need to create a copy of the state before shutting down old replicas. Clearly you'd want to minimize the cost of any solution that Azure Service Fabric Cluster Resource Manager comes up with. But you also don't want to ignore solutions that would significantly improve the allocation of resources in the cluster.

Cluster Resource Manager has two ways of computing costs and limiting them, even while it tries to manage the cluster according to its other goals. The first is that when Cluster Resource Manager is planning a new layout for the cluster, it counts every move that it would make. In a simple case, if you get two solutions with about the same overall balance (score) at the end, then take the one with the lowest cost (total number of moves).

This works pretty well. But as with default or static loads, it's unlikely in any complex system that all moves are equal. Some are likely to be much more expensive.

Changing a replica's move cost and factors to consider

As with reporting load (another feature of Cluster Resource Manager), you give the service a way of self-reporting how costly the service is to move at a particular time.

Code:

```
this.ServicePartition.ReportMoveCost(MoveCost.Medium);
```

MoveCost has four levels: Zero, Low, Medium, and High. These are relative to each other, except for Zero. Zero means that moving a replica is free and should not count against the score of the solution. Setting your move cost to High is *not* a guarantee that the replica won't move, just that it won't be moved unless there's a good reason to.

MoveCost helps you find the solutions that cause the least disruption overall and are easiest to achieve while still arriving at equivalent balance. A service's notion of cost can be relative to many things. The most common factors in calculating your move cost are:

- The amount of state or data that the service has to move.
- The cost of disconnection of clients. The cost of moving a primary replica is usually higher than the cost of moving a secondary replica.
- The cost of interrupting an in-flight operation. Some operations at the data store level or operations performed in response to a client call are costly. After a certain point, you don't want to stop them if you don't have to. So for the duration of the operation, you bump up the cost to reduce the likelihood that the service replica or instance will move. When the operation is done, you put it back to normal.

Next steps

- Service Fabric Cluster Resource Manger uses metrics to manage consumption and capacity in the cluster. To learn more about metrics and how to configure them, check out Managing resource consumption and load in Service Fabric with metrics.
- To learn about how the Cluster Resource Manager manages and balances load in the cluster, check out Balancing your Service Fabric cluster.

Service Fabric application lifecycle

11/22/2016 • 6 min to read • Edit on GitHub

Contributors

unknown • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • MarkDeichmiller • Jesse Benson • Mani Ramaswamy

As with other platforms, an application on Azure Service Fabric usually goes through the following phases: design, development, testing, deployment, upgrading, maintenance, and removal. Service Fabric provides first-class support for the full application lifecycle of cloud applications, from development through deployment, daily management, and maintenance to eventual decommissioning. The service model enables several different roles to participate independently in the application lifecycle. This article provides an overview of the APIs and how they are used by the different roles throughout the phases of the Service Fabric application lifecycle.

Service model roles

The service model roles are:

- **Service developer**: Develops modular and generic services that can be re-purposed and used in multiple applications of the same type or different types. For example, a queue service can be used for creating a ticketing application (helpdesk) or an e-commerce application (shopping cart).
- **Application developer**: Creates applications by integrating a collection of services to satisfy certain specific requirements or scenarios. For example, an e-commerce website might integrate "JSON Stateless Front-End Service," "Auction Stateful Service," and "Queue Stateful Service" to build an auctioning solution.
- **Application administrator**: Makes decisions on the application configuration (filling in the configuration template parameters), deployment (mapping to available resources), and quality of service. For example, an application administrator decides the language locale (English for the United States or Japanese for Japan, for example) of the application. A different deployed application can have different settings.
- **Operator**: Deploys applications based on the application configuration and requirements specified by the application administrator. For example, an operator provisions and deploys the application and ensures that it is running in Azure. Operators monitor application health and performance information and maintain the physical infrastructure as needed.

Develop

1. A *service developer* develops different types of services using the Reliable Actors or Reliable Services programming model.
2. A *service developer* declaratively describes the developed service types in a service manifest file consisting of one or more code, configuration, and data packages.
3. An *application developer* then builds an application using different service types.
4. An *application developer* declaratively describes the application type in an application manifest by referencing the service manifests of the constituent services and appropriately overriding and parameterizing different configuration and deployment settings of the constituent services.

See Get started with Reliable Actors and Get started with Reliable Services for examples.

Deploy

1. An *application administrator* tailors the application type to a specific application to be deployed to a Service

Fabric cluster by specifying the appropriate parameters of the **ApplicationType** element in the application manifest.

2. An *operator* uploads the application package to the cluster image store by using the **CopyApplicationPackage** method or the **Copy-ServiceFabricApplicationPackage** cmdlet. The application package contains the application manifest and the collection of service packages. Service Fabric deploys applications from the application package stored in the image store, which can be an Azure blob store or the Service Fabric system service.
3. The *operator* then provisions the application type in the target cluster from the uploaded application package using the **ProvisionApplicationAsync** method, the **Register-ServiceFabricApplicationType** cmdlet, or the **Provision an Application** REST operation.
4. After provisioning the application, an *operator* starts the application with the parameters supplied by the *application administrator* using the **CreateApplicationAsync** method, the **New-ServiceFabricApplication** cmdlet, or the **Create Application** REST operation.
5. After the application has been deployed, an *operator* uses the **CreateServiceAsync** method, the **New-ServiceFabricService** cmdlet, or the **Create Service** REST operation to create new service instances for the application based on available service types.
6. The application is now running in the Service Fabric cluster.

See Deploy an application for examples.

Test

1. After deploying to the local development cluster or a test cluster, a *service developer* runs the built-in failover test scenario by using the **FailoverTestScenarioParameters** and **FailoverTestScenario** classes, or the **Invoke-ServiceFabricFailoverTestScenario** cmdlet. The failover test scenario runs a specified service through important transitions and failovers to ensure that it's still available and working.
2. The *service developer* then runs the built-in chaos test scenario using the **ChaosTestScenarioParameters** and **ChaosTestScenario** classes, or the **Invoke-ServiceFabricChaosTestScenario** cmdlet. The chaos test scenario randomly induces multiple node, code package, and replica faults into the cluster.
3. The *service developer* tests service-to-service communication by authoring test scenarios that move primary replicas around the cluster.

See Introduction to the Fault Analysis Service for more information.

Upgrade

1. A *service developer* updates the constituent services of the instantiated application and/or fixes bugs and provides a new version of the service manifest.
2. An *application developer* overrides and parameterizes the configuration and deployment settings of the consistent services and provides a new version of the application manifest. The application developer then incorporates the new versions of the service manifests into the application and provides a new version of the application type in an updated application package.
3. An *application administrator* incorporates the new version of the application type into the target application by updating the appropriate parameters.
4. An *operator* uploads the updated application package to the cluster image store using the **CopyApplicationPackage** method or the **Copy-ServiceFabricApplicationPackage** cmdlet. The application package contains the application manifest and the collection of service packages.
5. An *operator* provisions the new version of the application in the target cluster by using the **ProvisionApplicationAsync** method, the **Register-ServiceFabricApplicationType** cmdlet, or the **Provision an Application** REST operation.
6. An *operator* upgrades the target application to the new version using the **UpgradeApplicationAsync** method, the **Start-ServiceFabricApplicationUpgrade** cmdlet, or the **Upgrade an Application** REST

operation.

7. An *operator* checks the progress of upgrade using the **GetApplicationUpgradeProgressAsync** method, the **Get-ServiceFabricApplicationUpgrade** cmdlet, or the **Get Application Upgrade Progress** REST operation.
8. If necessary, the *operator* modifies and reapplies the parameters of the current application upgrade using the **UpdateApplicationUpgradeAsync** method, the **Update-ServiceFabricApplicationUpgrade** cmdlet, or the **Update Application Upgrade** REST operation.
9. If necessary, the *operator* rolls back the current application upgrade using the **RollbackApplicationUpgradeAsync** method, the **Start-ServiceFabricApplicationRollback** cmdlet, or the **Rollback Application Upgrade** REST operation.
10. Service Fabric upgrades the target application running in the cluster without losing the availability of any of its constituent services.

See the Application upgrade tutorial for examples.

Maintain

1. For operating system upgrades and patches, Service Fabric interfaces with the Azure infrastructure to guarantee availability of all the applications running in the cluster.
2. For upgrades and patches to the Service Fabric platform, Service Fabric upgrades itself without losing availability of any of the applications running on the cluster.
3. An *application administrator* approves the addition or removal of nodes from a cluster after analyzing historical capacity utilization data and projected future demand.
4. An *operator* adds and removes nodes specified by the *application administrator*.
5. When new nodes are added to or existing nodes are removed from the cluster, Service Fabric automatically load-balances the running applications across all nodes in the cluster to achieve optimal performance.

Remove

1. An *operator* can delete a specific instance of a running service in the cluster without removing the entire application using the **DeleteServiceAsync** method, the **Remove-ServiceFabricService** cmdlet, or the **Delete Service** REST operation.
2. An *operator* can also delete an application instance and all of its services using the **DeleteApplicationAsync** method, the **Remove-ServiceFabricApplication** cmdlet, or the **Delete Application** REST operation.
3. Once the application and services have stopped, the *operator* can unprovision the application type using the **UnprovisionApplicationAsync** method, the **Unregister-ServiceFabricApplicationType** cmdlet, or the **Unprovision an Application** REST operation. Unprovisioning the application type does not remove the application package from the ImageStore. You must remove the application package manually.
4. An *operator* removes the application package from the ImageStore using the **RemoveApplicationPackage** method or the **Remove-ServiceFabricApplicationPackage** cmdlet.

See Deploy an application for examples.

Next steps

For more information on developing, testing, and managing Service Fabric applications and services, see:

- Reliable Actors
- Reliable Services
- Deploy an application
- Application upgrade
- Testability overview

- REST-based application lifecycle sample

Set up continuous integration for a Service Fabric application by using Visual Studio Team Services

11/15/2016 • 8 min to read • Edit on GitHub

Contributors

Matt Thalman • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • loekd • Aaron Dandy • Matt Kotsenas • Steven Moschidis • Matthew Pirocchi • ShawnJackson • AlisonParks • stevehug • Andy Pasic • Andy De George • Catherine Wang

This article describes the steps to set up continuous integration for an Azure Service Fabric application by using Visual Studio Team Services (VSTS), to ensure that your application is built, packaged, and deployed in an automated fashion.

This document reflects the current procedure and is expected to change over time.

Prerequisites

To get started, follow these steps:

1. Ensure that you have access to a Team Services account or create one yourself.
2. Ensure that you have access to a Team Services team project or create one yourself.
3. Ensure that you have a Service Fabric cluster to which you can deploy your application or create one using the Azure Portal, an Azure Resource Manager template, or Visual Studio.
4. Ensure that you have already created a Service Fabric Application (.sfproj) project. You must have a project that was created or upgraded with Service Fabric SDK 2.1 or higher (the .sfproj file should contain a ProjectVersion property value of 1.1 or higher).

> **NOTE**
> Custom build agents are no longer required. Team Services hosted agents now come pre-installed with Service Fabric cluster management software, allowing for deployment of your applications directly from those agents.

Configure and share your source files

The first thing you'll want to do is prepare a publish profile for use by the deployment process that will execute within Team Services. The publish profile should be configured to target the cluster that you've previously prepared:

1. Choose a publish profile within your Application project that you want to use for your continuous integration workflow and follow the publish instructions on how to publish an application to a remote cluster. You don't actually need to publish your application though. You can simply click the **Save** hyperlink in the publish dialog once you've configured things appropriately.
2. If you want your application to be upgraded for each deployment that occurs within Team Services, you'll want to configure the publish profile to enable upgrade. In the same publish dialog used in step 1, ensure that the **Upgrade the Application** checkbox is checked. Learn more about configuring additional upgrade settings. Click the **Save** hyperlink to save the settings to the publish profile.
3. Ensure that you've saved your changes to the publish profile and cancel the publish dialog.
4. Now it's time to share your Application project source files with Team Services. Once your source files are accessible in Team Services, you can now move on to the next step of generating builds.

Create a build definition

A Team Services build definition describes a workflow that is composed of a set of build steps that are executed sequentially. The goal of the build definition that you'll be creating is to produce a Service Fabric application package, as well as including some other supplemental files, that can be used to eventually deploy the application to a cluster. Learn more about Team Services build definitions.

Create a definition from the build template

1. Open your team project in Visual Studio Team Services.
2. Select the **Build** tab.
3. Select the green **+** sign to create a new build definition.
4. In the dialog that opens, select **Azure Service Fabric Application** within the **Build** template category.
5. Select **Next**.
6. Select the repository and branch associated with your Service Fabric application.
7. Select the agent queue you wish to use. Hosted agents are supported.
8. Select **Create**.
9. Save the build definition and provide a name.
10. The following is a description of the build steps generated by the template:

BUILD STEP	DESCRIPTION
Nuget restore	Restores the NuGet packages for the solution.
Build solution *.sln	Builds the entire solution.
Build solution *.sfproj	Generates the Service Fabric application package that will be used to deploy the application. Note that the application package location is specified to be within the build's artifact directory.
Update Service Fabric App Versions	Updates the version values contained in the application package's manifest files to allow for upgrade support. See the task documentation page for more information.
Copy Files	Copies the publish profile and application parameters files to the build's artifacts in order to be consumed for deployment.
Publish Artifact	Publishes the build's artifacts. This allows a release definition to consume the build's artifacts.

Verify the default set of tasks

1. Verify the **Solution** input field for the **NuGet restore** and **Build solution** build steps. By default, these build steps will execute upon all solution files that are contained in the associated repository. If you only want the build definition to operate on one of those solution files, you need to explicitly update the path to that file.
2. Verify the **Solution** input field for the **Package application** build step. By default, this build step assumes only one Service Fabric Application project (.sfproj) exists in the repository. If you have multiple such files in your repository and want to target only one of them for this build definition, you need to explicitly update the path to that file. If you want to package multiple Application projects in your repository, you need to create additional **Visual Studio Build** steps in the build definition that each target an Application project. You would then also need to update the **MSBuild Arguments** field for each of those build steps so that the package location is unique for each of them.
3. Verify the versioning behavior defined in the **Update Service Fabric App Versions** build step. By default, this build step appends the build number to all version values in the application package's manifest files. See the

task documentation page for more information. This is useful for supporting upgrade of your application since each upgrade deployment requires different version values from the previous deployment. If you're not intending to use application upgrade in your workflow, you may consider disabling this build step. In fact, it must be disabled if your intention is to produce a build that can be used to overwrite an existing Service Fabric application because deployment will fail if the version of the application produced by the build does not match the version of the application in the cluster.

4. If your solution contains a .NET Core project, you must ensure that your build definition contains a build step that restores the dependencies defined by any project.json files. To do this, follow these steps:
   a. Select **Add build step...**.
   b. Locate the **Command Line** task within the Utility tab and click its Add button.
   c. For the task's input fields, use the following values:
   d. Tool: dotnet
   e. Arguments: restore
   f. Drag the task so that it is immediately after the **NuGet restore** step.
5. Save any changes you've made to the build definition.

Try it

Select **Queue Build** to manually start a build. Builds will also be triggered upon push or check-in. Once you've verified that the build is executing successfully, you can now move on to defining a release definition that will deploy your application to a cluster.

Create a release definition

A Team Services release definition describes a workflow that is composed of a set of tasks that are executed sequentially. The goal of the release definition that you'll be creating is to take an application package and deploy it to a cluster. When used together, the build definition and release definition can execute the entire workflow from starting with source files to ending with a running application in your cluster. Learn more about Team Services release definitions.

Create a definition from the release template

1. Open your project in Visual Studio Team Services.
2. Select the **Release** tab.
3. Select the green **+** sign to create a new release definition and select **Create release definition** in the menu.
4. In the dialog that opens, select **Azure Service Fabric Deployment** within the **Deployment** template category.
5. Select **Next**.
6. Select the build definition you want to use as the source of this release definition. The release definition will reference the artifacts that were produced by the selected build definition.
7. Check the **Continuous deployment** check box if you wish to have Team Services automatically create a new release and deploy the Service Fabric application whenever a build completes.
8. Select the agent queue you wish to use. Hosted agents are supported.
9. Select **Create**.
10. Edit the definition name by clicking the pencil icon at the top of the page.
11. Select the cluster to which your application should be deployed from the **Cluster Connection** input field of the task. The cluster connection provides the necessary information that allows the deployment task to connect to the cluster. If you do not yet have a cluster connection for your cluster, select the **Manage** hyperlink next to the field to add one. On the page that opens, perform the following steps:
    a. Select **New Service Endpoint** and then select **Azure Service Fabric** from the menu.
    b. Select the type of authentication being used by the cluster targeted by this endpoint.
    c. Define a name for your connection in the **Connection Name** field. Typically, you would use the name of

your cluster.

d. Define the client connection endpoint URL in the **Cluster Endpoint** field. Example: https://contoso.westus.cloudapp.azure.com:19000.

e. For Azure Active Directory credentials, define the credentials you want to use to connect to the cluster in the **Username** and **Password** fields.

f. For Certificate Based authentication, define the Base64 encoding of the client certificate file in the **Client Certificate** field. See the help pop-up on that field for info on how to get that value. If your certificate is password-protected, define the password in the **Password** field.

g. Confirm your changes by clicking **OK**. After navigating back to your release definition, click the refresh icon on the **Cluster Connection** field to see the endpoint you just added.

12. Save the release definition.

The definition that is created consists of one task of type **Service Fabric Application Deployment**. See the task documentation page for more information about this task.

Verify the template defaults

1. Verify the **Publish Profile** input field for the **Deploy Service Fabric Application** task. By default, this field references a publish profile named Cloud.xml contained in the build's artifacts. If you want to reference a different publish profile or if the build contains multiple application packages in its artifacts, you need to update the path appropriately.
2. Verify the **Application Package** input field for the **Deploy Service Fabric Application** task. By default, this references the default application package path used in the build definition template. If you've modified the default application package path in the build definition, you need to update the path appropriately here as well.

Try it

Select **Create Release** from the **Release** button menu to manually create a release. In the dialog that opens, select the build that you want to base the release on and then click **Create**. If you enabled continuous deployment, releases will also be created automatically when the associated build definition completes a build.

Next steps

To learn more about continuous integration with Service Fabric applications, read the following articles:

- Team Services documentation home
- Build management in Team Services
- Release management in Team Services

Deploy and remove applications using PowerShell

11/22/2016 • 4 min to read • Edit on GitHub

Contributors

unknown • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Dhanashri Kshirsagar • Sean McKenna • Andy De George • ShawnJackson • alexwun

Once an application type has been packaged, it's ready for deployment into an Azure Service Fabric cluster. Deployment involves the following three steps:

1. Upload the application package
2. Register the application type
3. Create the application instance

> **NOTE**
>
> If you use Visual Studio for deploying and debugging applications on your local development cluster, all the following steps are handled automatically through a PowerShell script found in the Scripts folder of the application project. This article provides background on what those scripts are doing so that you can perform the same operations outside of Visual Studio.

Upload the application package

Uploading the application package puts it in a location that's accessible by internal Service Fabric components. You can use PowerShell to perform the upload. Before you run any PowerShell commands in this article, always start by using Connect-ServiceFabricCluster to connect to the Service Fabric cluster.

Suppose you have a folder named *MyApplicationType* that contains the necessary application manifest, service manifests, and code/config/data packages. The Copy-ServiceFabricApplicationPackage command uploads the package to the cluster Image Store. The **Get-ImageStoreConnectionStringFromClusterManifest** cmdlet, which is part of the Service Fabric SDK PowerShell module, is used to get the image store connection string. To import the SDK module, run:

```
Import-Module "$ENV:ProgramFiles\Microsoft SDKs\Service
Fabric\Tools\PSModule\ServiceFabricSDK\ServiceFabricSDK.psm1"
```

You can copy an application package from *C:\users\ryanwi\Documents\Visual Studio 2015\Projects\MyApplication\myapplication\pkg\debug* to *c:\temp\MyApplicationType* (rename the "debug" directory to "MyApplicationType"). The following example uploads the package:

```
PS C:\temp> dir

    Directory: c:\temp


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        8/12/2016  10:23 AM                MyApplicationType

PS C:\temp> tree /f .\Stateless1Pkg

C:\TEMP\MyApplicationType
    ApplicationManifest.xml
|
    Stateless1Pkg
    |   ServiceManifest.xml
    
    Code
    |   |   Microsoft.ServiceFabric.Data.dll
    |   |   Microsoft.ServiceFabric.Data.Interfaces.dll
    |   |   Microsoft.ServiceFabric.Internal.dll
    |   |   Microsoft.ServiceFabric.Internal.Strings.dll
    |   |   Microsoft.ServiceFabric.Services.dll
    |   |   ServiceFabricServiceModel.dll
    |   |   MyService.exe
    |   |   MyService.exe.config
    |   |   MyService.pdb
    |   |   System.Fabric.dll
    |   |   System.Fabric.Strings.dll
    |   |
    |   |   en-us
    |         Microsoft.ServiceFabric.Internal.Strings.resources.dll
    |         System.Fabric.Strings.resources.dll
    |
    Config
    |     Settings.xml
    |
    Data
          init.dat

PS C:\temp> Copy-ServiceFabricApplicationPackage -ApplicationPackagePath MyApplicationType -
ImageStoreConnectionString (Get-ImageStoreConnectionStringFromClusterManifest(Get-
ServiceFabricClusterManifest))
Copy application package succeeded

PS D:\temp>
```

Register the application package

Registering the application package makes the application type and version declared in the application manifest available for use. The system reads the package uploaded in the previous step, verify the package (equivalent to running Test-ServiceFabricApplicationPackage locally), process the package contents, and copy the processed package to an internal system location.

```
PS D:\temp> Register-ServiceFabricApplicationType MyApplicationType
Register application type succeeded

PS D:\temp> Get-ServiceFabricApplicationType

ApplicationTypeName    : MyApplicationType
ApplicationTypeVersion : AppManifestVersion1
DefaultParameters      : {}

PS D:\temp>
```

The Register-ServiceFabricApplicationType command returns only after the system has successfully copied the application package. How long this takes depends on the contents of the application package. If needed, the **-TimeoutSec** parameter can be used to supply a longer timeout. (The default timeout is 60 seconds.)

The Get-ServiceFabricApplicationType command lists all successfully registered application type versions.

Create the application

You can instantiate an application by using any application type version that has been registered successfully through the New-ServiceFabricApplication command. The name of each application must start with the *fabric:* scheme and be unique for each application instance. Any default services defined in the application manifest of the target application type are created at this time.

```
PS D:\temp> New-ServiceFabricApplication fabric:/MyApp MyApplicationType AppManifestVersion1

ApplicationName        : fabric:/MyApp
ApplicationTypeName    : MyApplicationType
ApplicationTypeVersion : AppManifestVersion1
ApplicationParameters  : {}

PS D:\temp> Get-ServiceFabricApplication

ApplicationName        : fabric:/MyApp
ApplicationTypeName    : MyApplicationType
ApplicationTypeVersion : AppManifestVersion1
ApplicationStatus      : Ready
HealthState            : OK
ApplicationParameters  : {}

PS D:\temp> Get-ServiceFabricApplication | Get-ServiceFabricService

ServiceName            : fabric:/MyApp/MyService
ServiceKind            : Stateless
ServiceTypeName        : MyServiceType
IsServiceGroup         : False
ServiceManifestVersion : SvcManifestVersion1
ServiceStatus          : Active
HealthState            : Ok

PS D:\temp>
```

The Get-ServiceFabricApplication command lists all application instances that were successfully created, along with their overall status.

The Get-ServiceFabricService command lists all service instances that were successfully created within a given application instance. Default services (if any) are listed here.

Multiple application instances can be created for any given version of a registered application type. Each application instance runs in isolation, with its own work directory and process.

Remove an application

When an application instance is no longer needed, you can permanently remove it by using the Remove-ServiceFabricApplication command. This command automatically removes all services that belong to the application as well, permanently removing all service state. This operation cannot be reversed, and application state cannot be recovered.

```
PS D:\temp> Remove-ServiceFabricApplication fabric:/MyApp

Confirm
Continue with this operation?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"):
Remove application instance succeeded

PS D:\temp> Get-ServiceFabricApplication
PS D:\temp>
```

When a particular version of an application type is no longer needed, you should unregister it by using the Unregister-ServiceFabricApplicationType command. Unregistering unused types releases storage space used by the application package contents of that type on the image store. An application type can be unregistered as long as no applications are instantiated against it and no pending application upgrades are referencing it.

```
PS D:\temp> Get-ServiceFabricApplicationType

ApplicationTypeName    : DemoAppType
ApplicationTypeVersion : v1
DefaultParameters      : {}

ApplicationTypeName    : DemoAppType
ApplicationTypeVersion : v2
DefaultParameters      : {}

ApplicationTypeName    : MyApplicationType
ApplicationTypeVersion : AppManifestVersion1
DefaultParameters      : {}

PS D:\temp> Unregister-ServiceFabricApplicationType MyApplicationType AppManifestVersion1
Unregister application type succeeded

PS D:\temp> Get-ServiceFabricApplicationType

ApplicationTypeName    : DemoAppType
ApplicationTypeVersion : v1
DefaultParameters      : {}

ApplicationTypeName    : DemoAppType
ApplicationTypeVersion : v2
DefaultParameters      : {}

PS D:\temp>
```

Troubleshooting

Copy-ServiceFabricApplicationPackage asks for an ImageStoreConnectionString

The Service Fabric SDK environment should already have the correct defaults set up. But if needed, the ImageStoreConnectionString for all commands should match the value that the Service Fabric cluster is using. You can find this in the cluster manifest retrieved through the Get-ServiceFabricClusterManifest command:

```
PS D:\temp> Copy-ServiceFabricApplicationPackage .\MyApplicationType

cmdlet Copy-ServiceFabricApplicationPackage at command pipeline position 1
Supply values for the following parameters:
ImageStoreConnectionString:

PS D:\temp> Get-ServiceFabricClusterManifest
<ClusterManifest xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" Name="Server-Default-SingleNode" Version="1.0" xmlns="http://schemas.microsoft.com/2011/01/fabric">

    [...]

    <Section Name="Management">
      <Parameter Name="ImageStoreConnectionString" Value="file:D:\ServiceFabric\Data\ImageStore" />
    </Section>

    [...]

PS D:\temp> Copy-ServiceFabricApplicationPackage .\MyApplicationType -ImageStoreConnectionString
file:D:\ServiceFabric\Data\ImageStore
Copy application package succeeded

PS D:\temp>
```

Next steps

Service Fabric application upgrade

Service Fabric health introduction

Diagnose and troubleshoot a Service Fabric service

Model an application in Service Fabric

Publish an application to a remote cluster by using Visual Studio

11/15/2016 • 5 min to read • Edit on GitHub

Contributors

Catherine Wang • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Dhanashri Kshirsagar • stevehug • Andy De George • Sean McKenna

The Azure Service Fabric extension for Visual Studio provides an easy, repeatable, and scriptable way to publish an application to a Service Fabric cluster.

The artifacts required for publishing

Deploy-FabricApplication.ps1

This is a PowerShell script that uses a publish profile path as a parameter for publishing Service Fabric applications. Since this script is part of your application, you are welcome to modify it as necessary for your application.

Publish profiles

A folder in the Service Fabric application project called **PublishProfiles** contains XML files that store essential information for publishing an application, such as:

- Service Fabric cluster connection parameters
- Path to an application parameter file
- Upgrade settings

By default, your application will include two publish profiles: Local.xml and Cloud.xml. You can add more profiles by copying and pasting one of the default files.

Application parameter files

A folder in the Service Fabric application project called **ApplicationParameters** contains XML files for user-specified application manifest parameter values. Application manifest files can be parameterized so that you can use different values for deployment settings. To learn more about parameterizing your application, see Manage multiple environments in Service Fabric.

> **NOTE**
>
> For actor services, you should build the project first before attempting to edit the file in an editor or through the publish dialog box. This is because part of the manifest files will be generated during the build.

To publish an application by using the Publish Service Fabric Application dialog box

The following steps demonstrate how to publish an application by using the **Publish Service Fabric Application** dialog box provided by the Visual Studio Service Fabric Tools.

1. On the shortcut menu of the Service Fabric Application project, choose **Publish...** to view the **Publish Service Fabric Application** dialog box.

The file selected in the **Target profile** dropdown list box is where all of the settings, except **Manifest versions**, are saved. You can either reuse an existing profile or create a new one by choosing in the **Target profile** dropdown list box. When you choose a publish profile, its contents appear in the corresponding fields of the dialog box. To save your changes at any time, choose the **Save Profile** link.

2. In the **Connection endpoint** section, specify a local or remote Service Fabric cluster's publishing endpoint. To add or change the connection endpoint, click on the **Connection Endpoint** dropdown list. The list shows the available Service Fabric cluster connection endpoints to which you can publish based on your Azure subscription(s). Note that if you are not already logged in to Visual Studio, you will be prompted to do so.

   Use the cluster selection dialog box to choose from the set of available subscriptions and clusters.

> **NOTE**
> If you would like to publish to an arbitrary endpoint (such as a party cluster), see the **Publishing to an arbitrary cluster endpoint** section below.

Once you choose an endpoint, Visual Studio validates the connection to the selected Service Fabric cluster. If the cluster isn't secure, Visual Studio can connect to it immediately. However, if the cluster is secure, you'll need to install a certificate on your local computer before proceeding. See How to configure secure connections for more information. When you're done, choose the **OK** button. The selected cluster appears in the **Publish Service Fabric Application** dialog box.

3. In the **Application Parameter File** dropdown list box, navigate to an application parameter file. An application parameter file holds user-specified values for parameters in the application manifest file. To add or change a parameter, choose the **Edit** button. Enter or change the parameter's value in the **Parameters** grid. When you're done, choose the **Save** button.

4. Use the **Upgrade the Application** checkbox to specify whether this publish action is an upgrade. Upgrade publish actions differ from normal publish actions. See Service Fabric Application Upgrade for a list of differences. To configure upgrade settings, choose the **Configure Upgrade Settings** link. The upgrade parameter editor appears. See Configure the upgrade of a Service Fabric application to learn more about upgrade parameters.
5. Choose the **Manifest Versions...** button to view the **Edit Versions** dialog box. You need to update application and service versions for an upgrade to take place. See Service Fabric application upgrade tutorial to learn how application and service manifest versions impact an upgrade process.

If the application and service versions use semantic versioning such as 1.0.0 or numerical values in the

format of 1.0.0.0, select the **Automatically update application and service versions** option. When you choose this option, the service and application version numbers are automatically updated whenever a code, config, or data package version is updated. If you prefer to edit the versions manually, clear the checkbox to disable this feature.

> **NOTE**
>
> For all package entries to appear for an actor project, first build the project to generate the entries in the Service Manifest files.

6. When you're done specifying all of the necessary settings, choose the **Publish** button to publish your application to the selected Service Fabric cluster. The settings that you specified are applied to the publish process.

Publish to an arbitrary cluster endpoint (including party clusters)

The Visual Studio publishing experience is optimized for publishing to remote clusters associated with one of your Azure subscriptions. However, it is possible to publish to arbitrary endpoints (such as Service Fabric party clusters) by directly editing the publish profile XML. As described above, two publish profiles are provided by default--**Local.xml** and **Cloud.xml**--but you are welcome to create additional profiles for different environments. For instance, you might want to create a profile for publishing to party clusters, perhaps named **Party.xml**.

If you are connecting to a unsecured cluster, all that's required is the cluster connection endpoint, such as `partycluster1.eastus.cloudapp.azure.com:19000`. In that case, the connection endpoint in the publish profile would look something like this:

```
<ClusterConnectionParameters ConnectionEndpoint="partycluster1.eastus.cloudapp.azure.com:19000" />
```

If you are connecting to a secured cluster, you will also need to provide the details of the client certificate from the local store to be used for authentication. For more details, see Configuring secure connections to a Service Fabric cluster.

Once your publish profile is set up, you can reference it in the publish dialog box as shown below.

Note that in this case, the new publish profile points to one of the default application parameter files. This is

appropriate if you want to publish the same application configuration to a number of environments. By contrast, in cases where you want to have different configurations for each environment that you want to publish to, it would make sense to create a corresponding application parameter file.

Next steps

To learn how to automate the publishing process in a continuous integration environment, see Set up Service Fabric continuous integration.

Service Fabric application upgrade

11/22/2016 • 4 min to read • Edit on GitHub

Contributors

Mani Ramaswamy • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Rebecca Agiewich • Andy De George • Jesse Benson

An Azure Service Fabric application is a collection of services. During an upgrade, Service Fabric compares the new application manifest with the previous version and determines which services in the application require updates. Service Fabric compares the version numbers in the service manifests with the version numbers in the previous version. If a service has not changed, that service is not upgraded.

Rolling upgrades overview

In a rolling application upgrade, the upgrade is performed in stages. At each stage, the upgrade is applied to a subset of nodes in the cluster, called an update domain. As a result, the application remains available throughout the upgrade. During the upgrade, the cluster may contain a mix of the old and new versions.

For that reason, the two versions must be forward and backward compatible. If they are not compatible, the application administrator is responsible for staging a multiple-phase upgrade to maintain availability. In a multiple-phase upgrade, the first step is upgrading to an intermediate version of the application that is compatible with the previous version. The second step is to upgrade the final version that breaks compatibility with the pre-update version, but is compatible with the intermediate version.

Update domains are specified in the cluster manifest when you configure the cluster. Update domains do not receive updates in a particular order. An update domain is a logical unit of deployment for an application. Update domains allow the services to remain at high availability during an upgrade.

Non-rolling upgrades are possible if the upgrade is applied to all nodes in the cluster, which is the case when the application has only one update domain. This approach is not recommended, since the service goes down and isn't available at the time of upgrade. Additionally, Azure doesn't provide any guarantees when a cluster is set up with only one update domain.

Health checks during upgrades

For an upgrade, health policies have to be set (or default values may be used). An upgrade is termed successful when all update domains are upgraded within the specified time-outs, and when all update domains are deemed healthy. A healthy update domain means that the update domain passed all the health checks specified in the health policy. For example, a health policy may mandate that all services within an application instance must be *healthy*, as health is defined by Service Fabric.

Health policies and checks during upgrade by Service Fabric are service and application agnostic. That is, no service-specific tests are done. For example, your service might have a throughput requirement, but Service Fabric does not have the information to check throughput. Refer to the health articles for the checks that are performed. The checks that happen during an upgrade include tests for whether the application package was copied correctly, whether the instance was started, and so on.

The application health is an aggregation of the child entities of the application. In short, Service Fabric evaluates the health of the application through the health that is reported on the application. It also evaluates the health of all the services for the application this way. Service Fabric further evaluates the health of the application services by aggregating the health of their children, such as the service replica. Once the application health policy is

satisfied, the upgrade can proceed. If the health policy is violated, the application upgrade fails.

Upgrade modes

The mode that we recommend for application upgrade is the monitored mode, which is the commonly used mode. Monitored mode performs the upgrade on one update domain, and if all health checks pass (per the policy specified), moves on to the next update domain automatically. If health checks fail and/or time-outs are reached, the upgrade is either rolled back for the update domain, or the mode is changed to unmonitored manual. You can configure the upgrade to choose one of those two modes for failed upgrades.

Unmonitored manual mode needs manual intervention after every upgrade on an update domain, to kick off the upgrade on the next update domain. No Service Fabric health checks are performed. The administrator performs the health or status checks before starting the upgrade in the next update domain.

Application upgrade flowchart

The flowchart following this paragraph can help you understand the upgrade process of a Service Fabric application. In particular, the flow describes how the time-outs, including *HealthCheckStableDuration*, *HealthCheckRetryTimeout*, and *UpgradeHealthCheckInterval*, help control when the upgrade in one update domain is considered a success or a failure.

Next steps

Upgrading your Application Using Visual Studio walks you through an application upgrade using Visual Studio.

Upgrading your Application Using Powershell walks you through an application upgrade using PowerShell.

Control how your application upgrades by using Upgrade Parameters.

Make your application upgrades compatible by learning how to use Data Serialization.

Learn how to use advanced functionality while upgrading your application by referring to Advanced Topics.

Fix common problems in application upgrades by referring to the steps in Troubleshooting Application Upgrades.

Configure the upgrade of a Service Fabric application in Visual Studio

11/22/2016 • 2 min to read • Edit on GitHub

Contributors

Catherine Wang • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Wike • stevehug

Visual Studio tools for Azure Service Fabric provide upgrade support for publishing to local or remote clusters. There are two advantages to upgrading your application to a newer version instead of replacing the application during testing and debugging:

- Application data won't be lost during the upgrade.
- Availability remains high so there won't be any service interruption during the upgrade, if there are enough service instances spread across upgrade domains.

Tests can be run against an application while it's being upgraded.

Parameters needed to upgrade

You can choose from two types of deployment: regular or upgrade. A regular deployment erases any previous deployment information and data on the cluster, while an upgrade deployment preserves it. When you upgrade a Service Fabric application in Visual Studio, you need to provide application upgrade parameters and health check policies. Application upgrade parameters help control the upgrade, while health check policies determine whether the upgrade was successful. See Service Fabric application upgrade: upgrade parameters for more details.

There are three upgrade modes: *Monitored*, *UnmonitoredAuto*, and *UnmonitoredManual*.

- A Monitored upgrade automates the upgrade and application health check.
- An UnmonitoredAuto upgrade automates the upgrade, but skips the application health check.
- When you do an UnmonitoredManual upgrade, you need to manually upgrade each upgrade domain.

Each upgrade mode requires different sets of parameters. See Application upgrade parameters to learn more about the available upgrade options.

Upgrade a Service Fabric application in Visual Studio

If you're using the Visual Studio Service Fabric tools to upgrade a Service Fabric application, you can specify a publish process to be an upgrade rather than a regular deployment by checking the **Upgrade the application** check box.

To configure the upgrade parameters

1. Click the **Settings** button next to the check box. The **Edit Upgrade Parameters** dialog box appears. The **Edit Upgrade Parameters** dialog box supports the Monitored, UnmonitoredAuto, and UnmonitoredManual upgrade modes.
2. Select the upgrade mode that you want to use and then fill out the parameter grid.

   Each parameter has default values. The optional parameter *DefaultServiceTypeHealthPolicy* takes a hash table input. Here's an example of the hash table input format for *DefaultServiceTypeHealthPolicy*:

```
@{ ConsiderWarningAsError = "false"; MaxPercentUnhealthyDeployedApplications = 0;
MaxPercentUnhealthyServices = 0; MaxPercentUnhealthyPartitionsPerService = 0;
MaxPercentUnhealthyReplicasPerPartition = 0 }
```

ServiceTypeHealthPolicyMap is another optional parameter that takes a hash table input in the following format:

```
@ {"ServiceTypeName" :
"MaxPercentUnhealthyPartitionsPerService,MaxPercentUnhealthyReplicasPerPartition,MaxPercentUnhealthyServi
ces"}
```

Here's a real-life example:

```
@{ "ServiceTypeName01" = "5,10,5"; "ServiceTypeName02" = "5,5,5" }
```

3. If you select UnmonitoredManual upgrade mode, you must manually start a PowerShell console to continue and finish the upgrade process. Refer to Service Fabric application upgrade: advanced topics to learn how manual upgrade works.

Upgrade an application by using PowerShell

You can use PowerShell cmdlets to upgrade a Service Fabric application. See Service Fabric application upgrade tutorial and Start-ServiceFabricApplicationUpgrade for detailed information.

Specify a health check policy in the application manifest file

Every service in a Service Fabric application can have its own health policy parameters that override the default values. You can provide these parameter values in the application manifest file.

The following example shows how to apply a unique health check policy for each service in the application manifest.

```
<Policies>
    <HealthPolicy ConsiderWarningAsError="false" MaxPercentUnhealthyDeployedApplications="20">
        <DefaultServiceTypeHealthPolicy MaxPercentUnhealthyServices="20"
                MaxPercentUnhealthyPartitionsPerService="20"
                MaxPercentUnhealthyReplicasPerPartition="20" />
        <ServiceTypeHealthPolicy ServiceTypeName="ServiceTypeName1"
                MaxPercentUnhealthyServices="20"
                MaxPercentUnhealthyPartitionsPerService="20"
                MaxPercentUnhealthyReplicasPerPartition="20" />
    </HealthPolicy>
</Policies>
```

Next steps

For more information about deploying an application, see Deploy an existing application in Azure Service Fabric.

Contributors

Mani Ramaswamy • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Cody Mansfield • loekd • Rebecca Agiewich • Andy De George • Jesse Benson

title: Application upgrade: upgrade parameters | Microsoft Docs description: Describes parameters related to upgrading a Service Fabric application, including health checks to perform and policies to automatically undo the upgrade. services: service-fabric documentationcenter: .net author: mani-ramaswamy manager: timlt editor: ''

ms.assetid: a4170ac6-192e-44a8-b93d-7e39c92a347e ms.service: service-fabric ms.devlang: dotnet ms.topic: article ms.tgt_pltfrm: NA ms.workload: NA ms.date: 11/15/2016 ms.author: subramar

Application upgrade parameters

This article describes the various parameters that apply during the upgrade of an Azure Service Fabric application. The parameters include the name and version of the application. They are knobs that control the time-outs and health checks that are applied during the upgrade, and they specify the policies that must be applied when an upgrade fails.

PARAMETER	DESCRIPTION
ApplicationName	Name of the application that is being upgraded. Examples: fabric:/VisualObjects, fabric:/ClusterMonitor
TargetApplicationTypeVersion	The version of the application type that the upgrade targets.
FailureAction	The action taken by Service Fabric when the upgrade fails. The application may be rolled back to the pre-update version (rollback), or the upgrade may be stopped at the current upgrade domain. In the latter case, the upgrade mode is also changed to Manual. Allowed values are Rollback and Manual.
HealthCheckWaitDurationSec	The time to wait (in seconds) after the upgrade has finished on the upgrade domain before Service Fabric evaluates the health of the application. This duration can also be considered as the time an application should be running before it can be considered healthy. If the health check passes, the upgrade process proceeds to the next upgrade domain. If the health check fails, Service Fabric waits for an interval (the UpgradeHealthCheckInterval) before retrying the health check again until the HealthCheckRetryTimeout is reached. The default and recommended value is 0 seconds.
HealthCheckRetryTimeoutSec	The duration (in seconds) that Service Fabric continues to perform health evaluation before declaring the upgrade as failed. The default is 600 seconds. This duration starts after HealthCheckWaitDuration is reached. Within this HealthCheckRetryTimeout, Service Fabric might perform multiple health checks of the application health. The default value is 10 minutes and should be customized appropriately for your application.

PARAMETER	DESCRIPTION
HealthCheckStableDurationSec	The duration (in seconds) to verify that the application is stable before moving to the next upgrade domain or completing the upgrade. This wait duration is used to prevent undetected changes of health right after the health check is performed. The default value is 120 seconds, and should be customized appropriately for your application.
UpgradeDomainTimeoutSec	Maximum time (in seconds) for upgrading a single upgrade domain. If this time-out is reached, the upgrade stops and proceeds based on the setting for UpgradeFailureAction. The default value is never (Infinite) and should be customized appropriately for your application.
UpgradeTimeout	A time-out (in seconds) that applies for the entire upgrade. If this time-out is reached, the upgrade stops and UpgradeFailureAction is triggered. The default value is never (Infinite) and should be customized appropriately for your application.
UpgradeHealthCheckInterval	The frequency that the health status is checked. This parameter is specified in the ClusterManager section of the *cluster manifest*, and is not specified as part of the upgrade cmdlet. The default value is 60 seconds.

Service health evaluation during application upgrade

The health evaluation criteria are optional. If the health evaluation criteria are not specified when an upgrade starts, Service Fabric uses the application health policies specified in the ApplicationManifest.xml of the application instance.

PARAMETER	DESCRIPTION
ConsiderWarningAsError	Default value is False. Treat the warning health events for the application as errors when evaluating the health of the application during upgrade. By default, Service Fabric does not evaluate warning health events to be failures (errors), so the upgrade can proceed even if there are warning events.
MaxPercentUnhealthyDeployedApplications	Default and recommended value is 0. Specify the maximum number of deployed applications (see the Health section) that can be unhealthy before the application is considered unhealthy and fails the upgrade. This parameter defines the application health on the node and helps detect issues during upgrade. Typically, the replicas of the application get load-balanced to the other node, which allows the application to appear healthy, thus allowing the upgrade to proceed. By specifying a strict MaxPercentUnhealthyDeployedApplications health, Service Fabric can detect a problem with the application package quickly and help produce a fail fast upgrade.

PARAMETER	DESCRIPTION
MaxPercentUnhealthyServices	Default and recommended value is 0. Specify the maximum number of services in the application instance that can be unhealthy before the application is considered unhealthy and fails the upgrade.
MaxPercentUnhealthyPartitionsPerService	Default and recommended value is 0. Specify the maximum number of partitions in a service that can be unhealthy before the service is considered unhealthy.
MaxPercentUnhealthyReplicasPerPartition	Default and recommended value is 0. Specify the maximum number of replicas in partition that can be unhealthy before the partition is considered unhealthy.
UpgradeReplicaSetCheckTimeout	**Stateless service**--Within a single upgrade domain, Service Fabric tries to ensure that additional instances of the service are available. If the target instance count is more than one, Service Fabric waits for more than one instance to be available, up to a maximum time-out value. This time-out is specified by using the UpgradeReplicaSetCheckTimeout property. If the time-out expires, Service Fabric proceeds with the upgrade, regardless of the number of service instances. If the target instance count is one, Service Fabric does not wait, and immediately proceeds with the upgrade. **Stateful service**--Within a single upgrade domain, Service Fabric tries to ensure that the replica set has a quorum. Service Fabric waits for a quorum to be available, up to a maximum time-out value (specified by the UpgradeReplicaSetCheckTimeout property). If the time-out expires, Service Fabric proceeds with the upgrade, regardless of quorum. This setting is set as never (infinite) when rolling forward, and 900 seconds when rolling back.
ForceRestart	If you update a configuration or data package without updating the service code, the service is restarted only if the ForceRestart property is set to true. When the update is complete, Service Fabric notifies the service that a new configuration package or data package is available. The service is responsible for applying the changes. If necessary, the service can restart itself.

The MaxPercentUnhealthyServices, MaxPercentUnhealthyPartitionsPerService, and MaxPercentUnhealthyReplicasPerPartition criteria can be specified per service type for an application instance. Setting these parameters per-service allows for an application to contain different services types with different evaluation policies. For example, a stateless gateway service type can have a MaxPercentUnhealthyPartitionsPerService that is different from a stateful engine service type for a particular application instance.

Next steps

Upgrading your Application Using Visual Studio walks you through an application upgrade using Visual Studio.

Upgrading your Application Using Powershell walks you through an application upgrade using PowerShell.

Make your application upgrades compatible by learning how to use Data Serialization.

Learn how to use advanced functionality while upgrading your application by referring to Advanced Topics.

Fix common problems in application upgrades by referring to the steps in Troubleshooting Application Upgrades.

Service Fabric application upgrade using PowerShell

11/22/2016 • 6 min to read • Edit on GitHub

Contributors

Mani Ramaswamy • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Dhanashri Kshirsagar • Rebecca Agiewich

The most frequently used and recommended upgrade approach is the monitored rolling upgrade. Azure Service Fabric monitors the health of the application being upgraded based on a set of health policies. Once an update domain (UD) is upgraded, Service Fabric evaluates the application health and either proceeds to the next update domain or fails the upgrade depending on the health policies.

A monitored application upgrade can be performed using the managed or native APIs, PowerShell, or REST. For instructions on performing an upgrade using Visual Studio, see Upgrading your application using Visual Studio.

With Service Fabric monitored rolling upgrades, the application administrator can configure the health evaluation policy that Service Fabric uses to determine if the application is healthy. In addition, the administrator can configure the action to be taken when the health evaluation fails (for example, doing an automatic rollback.) This section walks through a monitored upgrade for one of the SDK samples that uses PowerShell.

Step 1: Build and deploy the Visual Objects sample

Build and publish the application by right-clicking on the application project, **VisualObjectsApplication**, and selecting the **Publish** command. For more information, see Service Fabric application upgrade tutorial. Alternatively, you can use PowerShell to deploy your application.

> **NOTE**
> Before any of the Service Fabric commands may be used in PowerShell, you first need to connect to the cluster by using the `Connect-ServiceFabricCluster` cmdlet. Similarly, it is assumed that the Cluster has already been set up on your local machine. See the article on setting up your Service Fabric development environment.

After building the project in Visual Studio, you can use the PowerShell command **Copy-ServiceFabricApplicationPackage** to copy the application package to the ImageStore. The next step is to register the application to the Service Fabric runtime using the **Register-ServiceFabricApplicationPackage** cmdlet. The final step is to start an instance of the application by using the **New-ServiceFabricApplication** cmdlet. These three steps are analogous to using the **Deploy** menu item in Visual Studio.

Now, you can use Service Fabric Explorer to view the cluster and the application. The application has a web service that can be navigated to in Internet Explorer by typing http://localhost:8081/visualobjects in the address bar. You should see some floating visual objects moving around in the screen. Additionally, you can use **Get-ServiceFabricApplication** to check the application status.

Step 2: Update the Visual Objects sample

You might notice that with the version that was deployed in Step 1, the visual objects do not rotate. Let's upgrade this application to one where the visual objects also rotate.

Select the VisualObjects.ActorService project within the VisualObjects solution, and open the

StatefulVisualObjectActor.cs file. Within that file, navigate to the method `MoveObject`, comment out `this.State.Move()`, and uncomment `this.State.Move(true)`. This change rotates the objects after the service is upgraded.

We also need to update the *ServiceManifest.xml* file (under PackageRoot) of the project **VisualObjects.ActorService**. Update the *CodePackage* and the service version to 2.0, and the corresponding lines in the *ServiceManifest.xml* file. You can use the Visual Studio *Edit Manifest Files* option after you right-click on the solution to make the manifest file changes.

After the changes are made, the manifest should look like the following (highlighted portions show the changes):

```
<ServiceManifestName="VisualObjects.ActorService" Version="2.0"
xmlns="http://schemas.microsoft.com/2011/01/fabric" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<CodePackageName="Code" Version="2.0">
```

Now the *ApplicationManifest.xml* file (found under the **VisualObjects** project under the **VisualObjects** solution) is updated to version 2.0 of the **VisualObjects.ActorService** project. In addition, the Application version is updated to 2.0.0.0 from 1.0.0.0. The *ApplicationManifest.xml* should look like the following snippet:

```
<ApplicationManifestxmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" ApplicationTypeName="VisualObjects" ApplicationTypeVersion="2.0.0.0"
xmlns="http://schemas.microsoft.com/2011/01/fabric">

 <ServiceManifestRefServiceManifestName="VisualObjects.ActorService" ServiceManifestVersion="2.0" />
```

Now, build the project by selecting just the **ActorService** project, and then right-clicking and selecting the **Build** option in Visual Studio. If you select **Rebuild all**, you should update the versions for all projects, since the code would have changed. Next, let's package the updated application by right-clicking on ***VisualObjectsApplication***, selecting the Service Fabric Menu, and choosing **Package**. This action creates an application package that can be deployed. Your updated application is ready to be deployed.

Step 3: Decide on health policies and upgrade parameters

Familiarize yourself with the application upgrade parameters and the upgrade process to get a good understanding of the various upgrade parameters, time-outs, and health criterion applied. For this walkthrough, the service health evaluation criterion is set to the default (and recommended) values, which means that all services and instances should be *healthy* after the upgrade.

However, let's increase the *HealthCheckStableDuration* to 60 seconds (so that the services are healthy for at least 20 seconds before the upgrade proceeds to the next update domain). Let's also set the *UpgradeDomainTimeout* to be 1200 seconds and the *UpgradeTimeout* to be 3000 seconds.

Finally, let's also set the *UpgradeFailureAction* to rollback. This option requires Service Fabric to roll back the application to the previous version if it encounters any issues during the upgrade. Thus, when starting the upgrade (in Step 4), the following parameters are specified:

FailureAction = Rollback

HealthCheckStableDurationSec = 60

UpgradeDomainTimeoutSec = 1200

UpgradeTimeout = 3000

Step 4: Prepare application for upgrade

Now the application is built and ready to be upgraded. If you open up a PowerShell window as an administrator

and type **Get-ServiceFabricApplication**, it should let you know that it is application type 1.0.0.0 of **VisualObjects** that's been deployed.

The application package is stored under the following relative path where you uncompressed the Service Fabric SDK - *Samples\Services\Stateful\VisualObjects\VisualObjects\obj\x64\Debug*. You should find a "Package" folder in that directory, where the application package is stored. Check the timestamps to ensure that it is the latest build (you may need to modify the paths appropriately as well).

Now let's copy the updated application package to the Service Fabric ImageStore (where the application packages are stored by Service Fabric). The parameter *ApplicationPackagePathInImageStore* informs Service Fabric where it can find the application package. We have put the updated application in "VisualObjects_V2" with the following command (you may need to modify paths again appropriately).

```
Copy-ServiceFabricApplicationPackage  -ApplicationPackagePath
.\Samples\Services\Stateful\VisualObjects\VisualObjects\obj\x64\Debug\Package
-ImageStoreConnectionString fabric:ImageStore   -ApplicationPackagePathInImageStore "VisualObjects\_V2"
```

The next step is to register this application with Service Fabric, which can be performed using the following command:

```
Register-ServiceFabricApplicationType -ApplicationPathInImageStore "VisualObjects\_V2"
```

If the preceding command doesn't succeed, it is likely that you need a rebuild of all services. As mentioned in Step 2, you may have to update your WebService version as well.

Step 5: Start the application upgrade

Now, we're all set to start the application upgrade by using the following command:

```
Start-ServiceFabricApplicationUpgrade -ApplicationName fabric:/VisualObjects -ApplicationTypeVersion 2.0.0.0 -
HealthCheckStableDurationSec 60 -UpgradeDomainTimeoutSec 1200 -UpgradeTimeout 3000   -FailureAction Rollback -
Monitored
```

The application name is the same as it was described in the *ApplicationManifest.xml* file. Service Fabric uses this name to identify which application is getting upgraded. If you set the time-outs to be too short, you may encounter a failure message that states the problem. Refer to the troubleshooting section, or increase the time-outs.

Now, as the application upgrade proceeds, you can monitor it using Service Fabric Explorer, or by using the following PowerShell command: **Get-ServiceFabricApplicationUpgrade fabric:/VisualObjects**.

In a few minutes, the status that you got by using the preceding PowerShell command, should state that all update domains were upgraded (completed). And you should find that the visual objects in your browser window have started rotating!

You can try upgrading from version 2 to version 3, or from version 2 to version 1 as an exercise. Moving from version 2 to version 1 is also considered an upgrade. Play with time-outs and health policies to make yourself familiar with them. When you are deploying to an Azure cluster, the parameters need to be set appropriately. It is good to set the time-outs conservatively.

Next steps

Upgrading your application using Visual Studio walks you through an application upgrade using Visual Studio.

Control how your application upgrades by using upgrade parameters.

Make your application upgrades compatible by learning how to use data serialization.

Learn how to use advanced functionality while upgrading your application by referring to Advanced topics.

Fix common problems in application upgrades by referring to the steps in Troubleshooting application upgrades.

Service Fabric application upgrade tutorial using Visual Studio

11/22/2016 • 3 min to read • Edit on GitHub

Contributors

Mani Ramaswamy • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Dhanashri Kshirsagar • Stephane Lapointe • Rebecca Agiewich • Andy De George • Jesse Benson

Azure Service Fabric simplifies the process of upgrading cloud applications by ensuring that only changed services are upgraded, and that application health is monitored throughout the upgrade process. It also automatically rolls back the application to the previous version upon encountering issues. Service Fabric application upgrades are *Zero Downtime*, since the application can be upgraded with no downtime. This tutorial covers how to complete a rolling upgrade from Visual Studio.

Step 1: Build and publish the Visual Objects sample

First, download the Visual Objects application from GitHub. Then, build and publish the application by right-clicking on the application project, **VisualObjects**, and selecting the **Publish** command in the Service Fabric menu item.

Selecting **Publish** brings up a popup, and you can set the **Target profile** to **PublishProfiles\Local.xml**. The window should look like the following before you click **Publish**.

Now you can click **Publish** in the dialog box. You can use Service Fabric Explorer to view the cluster and the application. The Visual Objects application has a web service that you can go to by typing http://localhost:8082/visualobjects/ in the address bar of your browser. You should see 10 floating visual objects moving around on the screen.

Step 2: Update the Visual Objects sample

You might notice that with the version that was deployed in step 1, the visual objects do not rotate. Let's upgrade this application to one where the visual objects also rotate.

Select the VisualObjects.ActorService project within the VisualObjects solution, and open the **VisualObjectActor.cs** file. Within that file, go to the method `MoveObject`, comment out `visualObject.Move(false)`, and uncomment `visualObject.Move(true)`. This code change rotates the objects after the service is upgraded. **Now you can build (not rebuild) the solution**, which builds the modified projects. If you select *Rebuild all*, you have to update the versions for all the projects.

We also need to version our application. To make the version changes after you right-click on the **VisualObjects** project, you can use the Visual Studio **Edit Manifest Versions** option. Selecting this option brings up the dialog box for edition versions as follows:

Update the versions for the modified projects and their code packages, along with the application to version 2.0.0. After the changes are made, the manifest should look like the following (bold portions show the changes):

The Visual Studio tools can do automatic rollups of versions upon selecting **Automatically update application and service versions**. If you use SemVer, you need to update the code and/or configuration package version alone if that option is selected.

Save the changes, and now check the **Upgrade the Application** box.

Step 3: Upgrade your application

Familiarize yourself with the application upgrade parameters and the upgrade process to get a good understanding of the various upgrade parameters, time-outs, and health criterion that can be applied. For this walkthrough, the service health evaluation criterion is set to the default (unmonitored mode). You can configure these settings by selecting **Configure Upgrade Settings** and then modifying the parameters as desired.

Now we are all set to start the application upgrade by selecting **Publish**. This option upgrades your application to version 2.0.0, in which the objects rotate. Service Fabric upgrades one update domain at a time (some objects are updated first, followed by others), and the service remains accessible during the upgrade. Access to the service can be checked through your client (browser).

Now, as the application upgrade proceeds, you can monitor it with Service Fabric Explorer, by using the **Upgrades in Progress** tab under the applications.

In a few minutes, all update domains should be upgraded (completed), and the Visual Studio output window should also state that the upgrade is completed. And you should find that *all* the visual objects in your browser window are now rotating!

You may want to try changing the versions, and moving from version 2.0.0 to version 3.0.0 as an exercise, or even from version 2.0.0 back to version 1.0.0. Play with time-outs and health policies to make yourself familiar with them. When deploying to an Azure cluster as opposed to a local cluster, the parameters used may have to differ. We recommend that you set the time-outs conservatively.

Next steps

Upgrading your application using PowerShell walks you through an application upgrade using PowerShell.

Control how your application is upgraded by using upgrade parameters.

Make your application upgrades compatible by learning how to use data serialization.

Learn how to use advanced functionality while upgrading your application by referring to Advanced topics.

Fix common problems in application upgrades by referring to the steps in Troubleshooting application upgrades.

Troubleshoot application upgrades

11/22/2016 • 8 min to read • Edit on GitHub

Contributors

Mani Ramaswamy • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Rebecca Agiewich • Andy De George • Jesse Benson

This article covers some of the common issues around upgrading an Azure Service Fabric application and how to resolve them.

Troubleshoot a failed application upgrade

When an upgrade fails, the output of the **Get-ServiceFabricApplicationUpgrade** command contains additional information for debugging the failure. The following list specifies how the additional information can be used:

1. Identify the failure type.
2. Identify the failure reason.
3. Isolate one or more failing components for further investigation.

This information is available when Service Fabric detects the failure regardless of whether the **FailureAction** is to roll back or suspend the upgrade.

Identify the failure type

In the output of **Get-ServiceFabricApplicationUpgrade**, **FailureTimestampUtc** identifies the timestamp (in UTC) at which an upgrade failure was detected by Service Fabric and **FailureAction** was triggered. **FailureReason** identifies one of three potential high-level causes of the failure:

1. UpgradeDomainTimeout - Indicates that a particular upgrade domain took too long to complete and **UpgradeDomainTimeout** expired.
2. OverallUpgradeTimeout - Indicates that the overall upgrade took too long to complete and **UpgradeTimeout** expired.
3. HealthCheck - Indicates that after upgrading an update domain, the application remained unhealthy according to the specified health policies and **HealthCheckRetryTimeout** expired.

These entries only show up in the output when the upgrade fails and starts rolling back. Further information is displayed depending on the type of the failure.

Investigate upgrade timeouts

Upgrade timeout failures are most commonly caused by service availability issues. The output following this paragraph is typical of upgrades where service replicas or instances fail to start in the new code version. The **UpgradeDomainProgressAtFailure** field captures a snapshot of any pending upgrade work at the time of failure.

```
PS D:\temp> Get-ServiceFabricApplicationUpgrade fabric:/DemoApp

ApplicationName                : fabric:/DemoApp
ApplicationTypeName            : DemoAppType
TargetApplicationTypeVersion   : v2
ApplicationParameters          : {}
StartTimestampUtc              : 4/14/2015 9:26:38 PM
FailureTimestampUtc            : 4/14/2015 9:27:05 PM
FailureReason                  : UpgradeDomainTimeout
UpgradeDomainProgressAtFailure : MYUD1

                                 NodeName            : Node4
                                 UpgradePhase        : PostUpgradeSafetyCheck
                                 PendingSafetyChecks :
                                     WaitForPrimaryPlacement - PartitionId: 744c8d9f-1d26-417e-a60e-
cd48f5c098f0

                                 NodeName            : Node1
                                 UpgradePhase        : PostUpgradeSafetyCheck
                                 PendingSafetyChecks :
                                     WaitForPrimaryPlacement - PartitionId: 4b43f4d8-b26b-424e-9307-
7a7a62e79750
UpgradeState                   : RollingBackCompleted
UpgradeDuration                : 00:00:46
CurrentUpgradeDomainDuration   : 00:00:00
NextUpgradeDomain              :
UpgradeDomainsStatus           : { "MYUD1" = "Completed";
                                 "MYUD2" = "Completed";
                                 "MYUD3" = "Completed" }
UpgradeKind                    : Rolling
RollingUpgradeMode             : UnmonitoredAuto
ForceRestart                   : False
UpgradeReplicaSetCheckTimeout  : 00:00:00
```

In this example, the upgrade failed at upgrade domain *MYUD1* and two partitions (*744c8d9f-1d26-417e-a60e-cd48f5c098f0* and *4b43f4d8-b26b-424e-9307-7a7a62e79750*) were stuck. The partitions were stuck because the runtime was unable to place primary replicas (*WaitForPrimaryPlacement*) on target nodes *Node1* and *Node4*.

The **Get-ServiceFabricNode** command can be used to verify that these two nodes are in upgrade domain *MYUD1*. The *UpgradePhase* says *PostUpgradeSafetyCheck*, which means that these safety checks are occurring after all nodes in the upgrade domain have finished upgrading. All this information points to a potential issue with the new version of the application code. The most common issues are service errors in the open or promotion to primary code paths.

An *UpgradePhase* of *PreUpgradeSafetyCheck* means there were issues preparing the upgrade domain before it was performed. The most common issues in this case are service errors in the close or demotion from primary code paths.

The current **UpgradeState** is *RollingBackCompleted*, so the original upgrade must have been performed with a rollback **FailureAction**, which automatically rolled back the upgrade upon failure. If the original upgrade was performed with a manual **FailureAction**, then the upgrade would instead be in a suspended state to allow live debugging of the application.

Investigate health check failures

Health check failures can be triggered by various issues that can happen after all nodes in an upgrade domain finish upgrading and passing all safety checks. The output following this paragraph is typical of an upgrade failure due to failed health checks. The **UnhealthyEvaluations** field captures a snapshot of health checks that failed at the time of the upgrade according to the specified health policy.

```
PS D:\temp> Get-ServiceFabricApplicationUpgrade fabric:/DemoApp

ApplicationName                         : fabric:/DemoApp
ApplicationTypeName                     : DemoAppType
TargetApplicationTypeVersion            : v4
ApplicationParameters                   : {}
StartTimestampUtc                       : 4/24/2015 2:42:31 AM
UpgradeState                            : RollingForwardPending
UpgradeDuration                         : 00:00:27
CurrentUpgradeDomainDuration            : 00:00:27
NextUpgradeDomain                       : MYUD2
UpgradeDomainsStatus                    : { "MYUD1" = "Completed";
                                          "MYUD2" = "Pending";
                                          "MYUD3" = "Pending" }
UnhealthyEvaluations                    :
                                          Unhealthy services: 50% (2/4), ServiceType='PersistedServiceType',
MaxPercentUnhealthyServices=0%.

                                          Unhealthy service: ServiceName='fabric:/DemoApp/Svc3',
AggregatedHealthState='Error'.

                                              Unhealthy partitions: 100% (1/1),
MaxPercentUnhealthyPartitionsPerService=0%.

                                              Unhealthy partition: PartitionId='3a9911f6-a2e5-452d-89a8-
09271e7e49a8', AggregatedHealthState='Error'.

                                                  Error event: SourceId='Replica', Property='InjectedFault'.

                                          Unhealthy service: ServiceName='fabric:/DemoApp/Svc2',
AggregatedHealthState='Error'.

                                              Unhealthy partitions: 100% (1/1),
MaxPercentUnhealthyPartitionsPerService=0%.

                                              Unhealthy partition: PartitionId='744c8d9f-1d26-417e-a60e-
cd48f5c098f0', AggregatedHealthState='Error'.

                                                  Error event: SourceId='Replica', Property='InjectedFault'.

UpgradeKind                             : Rolling
RollingUpgradeMode                      : Monitored
FailureAction                           : Manual
ForceRestart                            : False
UpgradeReplicaSetCheckTimeout           : 49710.06:28:15
HealthCheckWaitDuration                 : 00:00:00
HealthCheckStableDuration               : 00:00:10
HealthCheckRetryTimeout                 : 00:00:10
UpgradeDomainTimeout                    : 10675199.02:48:05.4775807
UpgradeTimeout                          : 10675199.02:48:05.4775807
ConsiderWarningAsError                  :
MaxPercentUnhealthyPartitionsPerService :
MaxPercentUnhealthyReplicasPerPartition :
MaxPercentUnhealthyServices             :
MaxPercentUnhealthyDeployedApplications :
ServiceTypeHealthPolicyMap              :
```

Investigating health check failures first requires an understanding of the Service Fabric health model. But even without such an in-depth understanding, we can see that two services are unhealthy: *fabric:/DemoApp/Svc3* and *fabric:/DemoApp/Svc2*, along with the error health reports ("InjectedFault" in this case). In this example, two out of four services are unhealthy, which is below the default target of 0% unhealthy (*MaxPercentUnhealthyServices*).

The upgrade was suspended upon failing by specifying a **FailureAction** of manual when starting the upgrade. This mode allows us to investigate the live system in the failed state before taking any further action.

Recover from a suspended upgrade

With a rollback **FailureAction**, there is no recovery needed since the upgrade automatically rolls back upon failing. With a manual **FailureAction**, there are several recovery options:

1. trigger a rollback
2. Proceed through the remainder of the upgrade manually
3. Resume the monitored upgrade

The **Start-ServiceFabricApplicationRollback** command can be used at any time to start rolling back the application. Once the command returns successfully, the rollback request has been registered in the system and starts shortly thereafter.

The **Resume-ServiceFabricApplicationUpgrade** command can be used to proceed through the remainder of the upgrade manually, one upgrade domain at a time. In this mode, only safety checks are performed by the system. No more health checks are performed. This command can only be used when the *UpgradeState* shows *RollingForwardPending*, which means that the current upgrade domain has finished upgrading but the next one has not started (pending).

The **Update-ServiceFabricApplicationUpgrade** command can be used to resume the monitored upgrade with both safety and health checks being performed.

```
PS D:\temp> Update-ServiceFabricApplicationUpgrade fabric:/DemoApp -UpgradeMode Monitored

UpgradeMode                              : Monitored
ForceRestart                             :
UpgradeReplicaSetCheckTimeout            :
FailureAction                            :
HealthCheckWaitDuration                  :
HealthCheckStableDuration                :
HealthCheckRetryTimeout                  :
UpgradeTimeout                           :
UpgradeDomainTimeout                     :
ConsiderWarningAsError                   :
MaxPercentUnhealthyPartitionsPerService  :
MaxPercentUnhealthyReplicasPerPartition  :
MaxPercentUnhealthyServices              :
MaxPercentUnhealthyDeployedApplications  :
ServiceTypeHealthPolicyMap               :

PS D:\temp>
```

The upgrade continues from the upgrade domain where it was last suspended and use the same upgrade parameters and health policies as before. If needed, any of the upgrade parameters and health policies shown in the preceding output can be changed in the same command when the upgrade resumes. In this example, the upgrade was resumed in Monitored mode, with the parameters and the health policies unchanged.

Further troubleshooting

Service Fabric is not following the specified health policies

Possible Cause 1:

Service Fabric translates all percentages into actual numbers of entities (for example, replicas, partitions, and services) for health evaluation and always rounds up to whole entities. For example, if the maximum *MaxPercentUnhealthyReplicasPerPartition* is 21% and there are five replicas, then Service Fabric allows up to two unhealthy replicas (that is,`Math.Ceiling (5*0.21)). Thus, health policies should be set accordingly.

Possible Cause 2:

Health policies are specified in terms of percentages of total services and not specific service instances. For example, before an upgrade, if an application has four service instances A, B, C, and D, where service D is

unhealthy but with little impact to the application. We want to ignore the known unhealthy service D during upgrade and set the parameter *MaxPercentUnhealthyServices* to be 25%, assuming only A, B, and C need to be healthy.

However, during the upgrade, D may become healthy while C becomes unhealthy. The upgrade would still succeed because only 25% of the services are unhealthy. However, it might result in unanticipated errors due to C being unexpectedly unhealthy instead of D. In this situation, D should be modeled as a different service type from A, B, and C. Since health policies are specified per service type, different unhealthy percentage thresholds can be applied to different services.

I did not specify a health policy for application upgrade, but the upgrade still fails for some time-outs that I never specified

When health policies aren't provided to the upgrade request, they are taken from the *ApplicationManifest.xml* of the current application version. For example, if you're upgrading Application X from version 1.0 to version 2.0, application health policies specified for in version 1.0 are used. If a different health policy should be used for the upgrade, then the policy needs to be specified as part of the application upgrade API call. The policies specified as part of the API call only apply during the upgrade. Once the upgrade is complete, the policies specified in the *ApplicationManifest.xml* are used.

Incorrect time-outs are specified

You may have wondered about what happens when time-outs are set inconsistently. For example, you may have an *UpgradeTimeout* that's less than the *UpgradeDomainTimeout*. The answer is that an error is returned. Errors are returned if the *UpgradeDomainTimeout* is less than the sum of *HealthCheckWaitDuration* and *HealthCheckRetryTimeout*, or if *UpgradeDomainTimeout* is less than the sum of *HealthCheckWaitDuration* and *HealthCheckStableDuration*.

My upgrades are taking too long

The time for an upgrade to complete depends on the health checks and time-outs specified. Health checks and time-outs depend on how long it takes to copy, deploy, and stabilize the application. Being too aggressive with time-outs might mean more failed upgrades, so we recommend starting conservatively with longer time-outs.

Here's a quick refresher on how the time-outs interact with the upgrade times:

Upgrades for an upgrade domain cannot complete faster than *HealthCheckWaitDuration* + *HealthCheckStableDuration*.

Upgrade failure cannot occur faster than *HealthCheckWaitDuration* + *HealthCheckRetryTimeout*.

The upgrade time for an upgrade domain is limited by *UpgradeDomainTimeout*. If *HealthCheckRetryTimeout* and *HealthCheckStableDuration* are both non-zero and the health of the application keeps switching back and forth, then the upgrade eventually times out on *UpgradeDomainTimeout*. *UpgradeDomainTimeout* starts counting down once the upgrade for the current upgrade domain begins.

Next steps

Upgrading your Application Using Visual Studio walks you through an application upgrade using Visual Studio.

Upgrading your Application Using Powershell walks you through an application upgrade using PowerShell.

Control how your application upgrades by using Upgrade Parameters.

Make your application upgrades compatible by learning how to use Data Serialization.

Learn how to use advanced functionality while upgrading your application by referring to Advanced Topics.

Fix common problems in application upgrades by referring to the steps in Troubleshooting Application Upgrades.

How data serialization affects an application upgrade

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Wike • Jesse Benson • ShawnJackson • Mani Ramaswamy

In a rolling application upgrade, the upgrade is applied to a subset of nodes, one upgrade domain at a time. During this process, some upgrade domains will be on the newer version of your application, and some upgrade domains will be on the older version of your application. During the rollout, the new version of your application must be able to read the old version of your data, and the old version of your application must be able to read the new version of your data. If the data format is not forward and backward compatible, the upgrade may fail, or worse, data may be lost or corrupted. This article discusses what constitutes your data format and offers best practices for ensuring that your data is forward and backward compatible.

What makes up your data format?

In Azure Service Fabric, the data that is persisted and replicated comes from your C# classes. For applications that use Reliable Collections, that is the objects in the reliable dictionaries and queues. For applications that use Reliable Actors, that is the backing state for the actor. These C# classes must be serializable to be persisted and replicated. Therefore, the data format is defined by the fields and properties that are serialized, as well as how they are serialized. For example, in an `IReliableDictionary<int, MyClass>` the data is a serialized `int` and a serialized `MyClass`.

Code changes that result in a data format change

Since the data format is determined by C# classes, changes to the classes may cause a data format change. Care must be taken to ensure that a rolling upgrade can handle the data format change. Examples that may cause data format changes:

- Adding or removing fields or properties
- Renaming fields or properties
- Changing the types of fields or properties
- Changing the class name or namespace

Data Contract as the default serializer

The serializer is generally responsible for reading the data and deserializing it into the current version, even if the data is in an older or *newer* version. The default serializer is the Data Contract serializer, which has well-defined versioning rules. Reliable Collections allow the serializer to be overridden, but Reliable Actors currently do not. The data serializer plays an important role in enabling rolling upgrades. The Data Contract serializer is the serializer that we recommend for Service Fabric applications.

How the data format affects a rolling upgrade

During a rolling upgrade, there are two main scenarios where the serializer may encounter an older or *newer* version of your data:

1. After a node is upgraded and starts back up, the new serializer will load the data that was persisted to disk by the old version.
2. During the rolling upgrade, the cluster will contain a mix of the old and new versions of your code. Since replicas may be placed in different upgrade domains, and replicas send data to each other, the new and/or old

version of your data may be encountered by the new and/or old version of your serializer.

> **NOTE**
> The "new version" and "old version" here refer to the version of your code that is running. The "new serializer" refers to the serializer code that is executing in the new version of your application. The "new data" refers to the serialized C# class from the new version of your application.

The two versions of code and data format must be both forward and backward compatible. If they are not compatible, the rolling upgrade may fail or data may be lost. The rolling upgrade may fail because the code or serializer may throw exceptions or a fault when it encounters the other version. Data may be lost if, for example, a new property was added but the old serializer discards it during deserialization.

Data Contract is the recommended solution for ensuring that your data is compatible. It has well-defined versioning rules for adding, removing, and changing fields. It also has support for dealing with unknown fields, hooking into the serialization and deserialization process, and dealing with class inheritance. For more information, see Using Data Contract.

Next steps

Uprading your Application Using Visual Studio walks you through an application upgrade using Visual Studio.

Uprading your Application Using Powershell walks you through an application upgrade using PowerShell.

Control how your application upgrades by using Upgrade Parameters.

Learn how to use advanced functionality while upgrading your application by referring to Advanced Topics.

Fix common problems in application upgrades by referring to the steps in Troubleshooting Application Upgrades .

Service Fabric application upgrade: advanced topics

11/22/2016 • 4 min to read • Edit on GitHub

Contributors

Mani Ramaswamy • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Rebecca Agiewich • Andy De George • Jesse Benson • Andy Pasic • Stuart Leeks

Adding or removing services during an application upgrade

If a new service is added to an application that is already deployed, and published as an upgrade, the new service is added to the deployed application. Such an upgrade does not affect any of the services that were already part of the application. However, an instance of the service that was added must be started for the new service to be active (using the `New-ServiceFabricService` cmdlet).

Services can also be removed from an application as part of an upgrade. However, all current instances of the to-be-deleted service must be stopped before proceeding with the upgrade (using the `Remove-ServiceFabricService` cmdlet).

Manual upgrade mode

> **NOTE**
> The unmonitored manual mode should be considered only for a failed or suspended upgrade. The monitored mode is the recommended upgrade mode for Service Fabric applications.

Azure Service Fabric provides multiple upgrade modes to support development and production clusters. Deployment options chosen may be different for different environments.

The monitored rolling application upgrade is the most typical upgrade to use in production. When the upgrade policy is specified, Service Fabric ensures that the application is healthy before the upgrade proceeds.

The application administrator can use the manual rolling application upgrade mode to have total control over the upgrade progress through the various upgrade domains. This mode is useful when a customized or complex health evaluation policy is required, or a non-conventional upgrade is happening (for example, the application is already in data loss).

Finally, the automated rolling application upgrade is useful for development or testing environments to provide a fast iteration cycle during service development.

Change to manual upgrade mode

Manual--Stop the application upgrade at the current UD and change the upgrade mode to Unmonitored Manual. The administrator needs to manually call **MoveNextApplicationUpgradeDomainAsync** to proceed with the upgrade or trigger a rollback by initiating a new upgrade. Once the upgrade enters into the Manual mode, it stays in the Manual mode until a new upgrade is initiated. The **GetApplicationUpgradeProgressAsync** command returns FABRIC_APPLICATION_UPGRADE_STATE_ROLLING_FORWARD_PENDING.

Upgrade with a diff package

A Service Fabric application can be upgraded by provisioning with a full, self-contained application package. An

application can also be upgraded by using a diff package that contains only the updated application files, the updated application manifest, and the service manifest files.

A full application package contains all the files necessary to start and run a Service Fabric application. A diff package contains only the files that changed between the last provision and the current upgrade, plus the full application manifest and the service manifest files. Any reference in the application manifest or service manifest that can't be found in the build layout is searched for in the image store.

Full application packages are required for the first installation of an application to the cluster. Subsequent updates can be either a full application package or a diff package.

Occasions when using a diff package would be a good choice:

- A diff package is preferred when you have a large application package that references several service manifest files and/or several code packages, config packages, or data packages.
- A diff package is preferred when you have a deployment system that generates the build layout directly from your application build process. In this case, even though the code hasn't changed, newly built assemblies get a different checksum. Using a full application package would require you to update the version on all code packages. Using a diff package, you only provide the files that changed and the manifest files where the version has changed.

When an application is upgraded using Visual Studio, the diff package is published automatically. To create a diff package manually, the application manifest, and the service manifests must be updated, but only the changed packages should be included in the final application package.

For example, let's start with the following application (version numbers provided for ease of understanding):

```
app1           1.0.0
  service1     1.0.0
    code       1.0.0
    config     1.0.0
  service2     1.0.0
    code       1.0.0
    config     1.0.0
```

Now, let's assume you wanted to update only the code package of service1 using a diff package using PowerShell. Now, your updated application has the following folder structure:

```
app1           2.0.0      <-- new version
  service1     2.0.0      <-- new version
    code       2.0.0      <-- new version
    config     1.0.0
  service2     1.0.0
    code       1.0.0
    config     1.0.0
```

In this case, you update the application manifest to 2.0.0, and the service manifest for service1 to reflect the code package update. The folder for your application package would have the following structure:

```
app1/
  service1/
    code/
```

Next steps

Upgrading your Application Using Visual Studio walks you through an application upgrade using Visual Studio.

Upgrading your Application Using Powershell walks you through an application upgrade using PowerShell.

Control how your application upgrades by using Upgrade Parameters.

Make your application upgrades compatible by learning how to use Data Serialization.

Fix common problems in application upgrades by referring to the steps in Troubleshooting Application Upgrades.

REST-based application lifecycle sample

11/15/2016 • 11 min to read • Edit on GitHub

Contributors

unknown • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • MarkDeichmiller • Jesse Benson

This sample demonstrates the Service Fabric application lifecycle through REST API calls. For more information on the Service Fabric application lifecycle, see Service Fabric application lifecycle.

This sample performs the following:

- Provisions the **WordCount 1.0.0** sample from the WordCount application package in the image store.
- Displays the list of application types, which includes WordCount 1.0.0.
- Creates the WordCount application as **fabric:/WordCount**.
- Displays the list of applications, which includes fabric:/WordCount version 1.0.0.
- Provisions the 1.1.0 version of the WordCount sample from the **WordCountUpgrade** application package in the image store.
- Displays the list of application types, which includes both WordCount 1.0.0 and **WordCount 1.1.0**.
- Upgrades the WordCount application to version 1.1.0.
- Displays the list of applications, which includes WordCount version 1.1.0, but no longer includes WordCount version 1.0.0.
- Deletes the WordCount application.
- Displays the list of applications, which no longer includes fabric:/WordCount.
- Unprovisions the 1.1.0 version of the WordCount sample.
- Displays the list of application types, which includes WordCount 1.0.0, but no longer includes WordCount 1.1.0.
- Unprovisions the 1.0.0 version of the WordCount sample.
- Displays the list of application types, which no longer includes WordCount.

Prerequisites

This sample uses the WordCount sample (found in the **Getting Started** samples). The WordCount sample must be built first, and then two application packages must be copied to the image store.

FOLDER	DESCRIPTION
WordCount	The WordCount sample application. The **ApplicationManifest.xml** file contains **ApplicationTypeVersion="1.0.0"**.
WordCountUpgrade	The WordCount sample application. The ApplicationManifest.xml file must be changed to **ApplicationTypeVersion="1.1.0"** to allow the application upgrade to occur.

To create the application packages and copy them to the image store, take the following steps:

1. Copy **C:\ServiceFabricSamples\Services\WordCount\WordCount\pkg\Debug** to **C:\Temp\WordCount**. This creates the WordCount application package.
2. Copy C:\Temp\WordCount to **C:\Temp\WordCountUpgrade**. This creates the **WordCountUpgrade**

application package.

3. Open **C:\Temp\WordCountUpgrade\ApplicationManifest.xml** in a text editor.
4. In the **ApplicationManifest** element, change the **ApplicationTypeVersion** attribute to **"1.1.0"**. This updates the version number of the application.
5. Save the changed ApplicationManifest.xml file.
6. Run the following PowerShell script as an administrator to copy the applications to the image store:

```
# Deploy the WordCount and upgrade applications
$applicationPathWordCount = "C:\Temp\WordCount"
$applicationPathUpgrade = "C:\Temp\WordCountUpgrade"

# LOCAL:
$imageStoreConnection = "file:C:\SfDevCluster\Data\ImageStoreShare"
$cluster = 'localhost:19000'

Connect-ServiceFabricCluster $cluster

Copy-ServiceFabricApplicationPackage -ApplicationPackagePath $applicationPathWordCount -
ImageStoreConnectionString $imageStoreConnection
Copy-ServiceFabricApplicationPackage -ApplicationPackagePath $applicationPathUpgrade -ImageStoreConnectionString
$imageStoreConnection
```

When the PowerShell script finishes, this application is ready to run.

Example

The following example demonstrates the Service Fabric application lifecycle.

```
using System;
using System.Collections.Generic;
using System.Fabric;
using System.Fabric.Description;
using System.Fabric.Health;
using System.Fabric.Query;
using System.IO;
using System.Net;
using System.Text;
using System.Web.Script.Serialization;

namespace ServiceFabricRestCaller
{
    class Program
    {
        static void Main(string[] args)
        {
            Uri clusterUri = new Uri("http://localhost:19080");
            string buildPathApplication = "WordCount";
            string applicationVersionNumber = "1.0.0";
            string buildPathUpgrade = "WordCountUpgrade";
            string updateVersionNumber = "1.1.0";

            Console.WriteLine("\nProvision the 1.0.0 WordCount application for the first time.");
            ProvisionAnApplication(clusterUri, buildPathApplication);
            Console.WriteLine("\nPress Enter to get the list of application types: ");
            Console.ReadLine();


            Console.WriteLine("\nGet the list of application types.");
            GetListOfApplicationTypes(clusterUri);
            Console.WriteLine("\nPress Enter to create the fabric:/WordCount application: ");
            Console.ReadLine();


            Console.WriteLine("\nCreate the fabric:/WordCount application.");
```

```
Console.WriteLine("\nCreate the fabric:/WordCount application: ");
CreateApplication(clusterUri);
Console.WriteLine("\nPress Enter to get the list of applications: ");
Console.ReadLine();


Console.WriteLine("\nGet the list of applications.");
GetApplicationList(clusterUri);
Console.WriteLine("\nPress Enter to provision the 1.1.0 upgrade to the WordCount application: ");
Console.ReadLine();


Console.WriteLine("\nProvision the 1.1.0 upgrade to the WordCount application.");
ProvisionAnApplication(clusterUri, buildPathUpgrade);
Console.WriteLine("\nPress Enter to get the list of application types: ");
Console.ReadLine();


Console.WriteLine("\nGet the list of application types.");
GetListOfApplicationTypes(clusterUri);
Console.WriteLine("\nPress Enter to upgrade the fabric:/WordCount application: ");
Console.ReadLine();


Console.WriteLine("\nUpgrade the fabric:/WordCount application.");
UpgradeApplicationByApplicationType(clusterUri);
Console.WriteLine("\nPress Enter to get the list of applications: ");
Console.ReadLine();


Console.WriteLine("\nGet the list of applications.");
GetApplicationList(clusterUri);
Console.WriteLine("\nPress Enter to delete the fabric:/WordCount application: ");
Console.ReadLine();


Console.WriteLine("\nDelete the fabric:/WordCount application.");
DeleteApplication(clusterUri);
Console.WriteLine("\nPress Enter to get the list of applications: ");
Console.ReadLine();


Console.WriteLine("\nGet the list of applications.");
GetApplicationList(clusterUri);
Console.WriteLine("\nPress Enter to unprovision the WordCount 1.1.0 application: ");
Console.ReadLine();


Console.WriteLine("\nUnprovision the WordCount 1.1.0 application.");
UnprovisionAnApplication(clusterUri, updateVersionNumber);
Console.WriteLine("\nPress Enter to get the list of application types: ");
Console.ReadLine();


Console.WriteLine("\nGet the list of application types.");
GetListOfApplicationTypes(clusterUri);
Console.WriteLine("\nPress Enter to unprovision the WordCount 1.0.0 application: ");
Console.ReadLine();


Console.WriteLine("\nUnprovision the WordCount 1.0.0 application.");
UnprovisionAnApplication(clusterUri, applicationVersionNumber);
Console.WriteLine("\nPress Enter to get the final list of application types: ");
Console.ReadLine();


Console.WriteLine("\nGet the final list of application types.");
GetListOfApplicationTypes(clusterUri);
Console.WriteLine("\nPress Enter to end this program: ");
Console.ReadLine();
```

```
        Console.ReadLine();
    }

    #region Classes

    /// <summary>
    /// Class similar to ApplicationType. Designed for use with JavaScriptSerializer.
    /// </summary>
    public class AppType
    {
        public string Name { get; set; }
        public string Version { get; set; }
        public List<ApplicationParameter> DefaultParameterList { get; set; }
    }

    /// <summary>
    /// Class designed for use with JavaScriptSerializer.
    /// </summary>
    public class ApplicationInfo
    {
        public string Id { get; set; }
        public string Name { get; set; }
        public string TypeName { get; set; }
        public string TypeVersion { get; set; }
        public ApplicationStatus Status { get; set; }
        public List<Parameter> Parameters { get; set; }
        public HealthState HealthState { get; set; }
    }

    /// <summary>
    /// Class similar to Parameter. Designed for use with JavaScriptSerializer.
    /// </summary>
    public class Parameter
    {
        public string Name { get; set; }
        public string Value { get; set; }
    }

    #endregion


    #region Get List of Application Types (REST API)

    /// <summary>
    /// Gets the list of application types.
    /// </summary>
    /// <param name="clusterUri">The URI to access the cluster.</param>
    /// <returns>Returns true if successful; otherwise false.</returns>
    public static bool GetListOfApplicationTypes(Uri clusterUri)
    {
        // String to capture the response stream.
        string responseString = string.Empty;

        // Create the request and add URL parameters.
        Uri requestUri = new Uri(clusterUri, string.Format("/ApplicationTypes?api-version={0}",
        "1.0"));    // api-version

        HttpWebRequest request = (HttpWebRequest)WebRequest.Create(requestUri);
        request.Method = "GET";

        // Execute the request and obtain the response.
        try
        {
            using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
            {
                using (StreamReader streamReader = new StreamReader(response.GetResponseStream(), true))
                {
                    // Capture the response string.
                    responseString = streamReader.ReadToEnd();
```

```
                }
            }
        }
        catch (WebException e)
        {
            // If there is a web exception, display the error message.
            Console.WriteLine("Error getting the list of application types:");
            Console.WriteLine(e.Message);
            if (e.InnerException != null)
                Console.WriteLine(e.InnerException.Message);
            return false;
        }
        catch (Exception e)
        {
            // If there is another kind of exception, throw it.
            throw (e);
        }

        // Deserialize the response string.
        JavaScriptSerializer jss = new JavaScriptSerializer();
        List<AppType> applicationTypes = jss.Deserialize<List<AppType>>(responseString);

        // Display application type information for each application type.
        Console.WriteLine("Application types:");
        foreach (AppType applicationType in applicationTypes)
        {
            Console.WriteLine("  Application Type:");
            Console.WriteLine("    Name: " + applicationType.Name);
            Console.WriteLine("    Version: " + applicationType.Version);
            Console.WriteLine("    Default Parameter List:");

            foreach (var parameter in applicationType.DefaultParameterList)
            {
                Console.WriteLine("      Name: " + parameter.Name);
                Console.WriteLine("      Value: " + parameter.Value);
            }
        }

        return true;
    }

    #endregion


    #region Provision an Application (REST API)

    /// <summary>
    /// Provisions an application to the image store.
    /// </summary>
    /// <param name="clusterUri">The URI to access the cluster.</param>
    /// <param name="applicationTypeBuildPath">The application type build path ("WordCount" or
"WordCountUpgrade").</param>
    /// <returns>Returns true if successful; otherwise false.</returns>
    public static bool ProvisionAnApplication(Uri clusterUri, string applicationTypeBuildPath)
    {
        // Create the request and add URL parameters.
        Uri requestUri = new Uri(clusterUri, string.Format("/ApplicationTypes/$/Provision?api-version={0}",
            "1.0"));    // api-version

        HttpWebRequest request = (HttpWebRequest)WebRequest.Create(requestUri);
        request.Method = "POST";
        request.ContentType = "application/json; charset=utf-8";

        // Create the byte array that will become the request body.
        string requestBody = "{\"ApplicationTypeBuildPath\":\"" + applicationTypeBuildPath + "\"}";
        byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBody);
        request.ContentLength = requestBodyBytes.Length;

        // Stores the response status code.
```

```
            HttpStatusCode statusCode;

            // Create the request body.
            try
            {
                using (Stream requestStream = request.GetRequestStream())
                {
                    requestStream.Write(requestBodyBytes, 0, requestBodyBytes.Length);
                    requestStream.Close();

                    // Execute the request and obtain the response.
                    using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
                    {
                        statusCode = response.StatusCode;
                    }
                }
            }
            catch (WebException e)
            {
                // If there is a web exception, display the error message.
                Console.WriteLine("Error provisioning the application:");
                Console.WriteLine(e.Message);
                if (e.InnerException != null)
                    Console.WriteLine(e.InnerException.Message);
                return false;
            }
            catch (Exception e)
            {
                // If there is another kind of exception, throw it.
                throw (e);
            }

            Console.WriteLine("Provision an Application response status = " + statusCode.ToString());
            return true;
        }

        #endregion

        #region Unprovision an Application (REST API)

        /// <summary>
        /// Unprovisions an application.
        /// </summary>
        /// <param name="clusterUri">The URI to access the cluster.</param>
        /// <returns>Returns true if successful; otherwise false.</returns>
        public static bool UnprovisionAnApplication(Uri clusterUri, string versionToUnprovision)
        {
            // Create the request and add URL parameters.
            Uri requestUri = new Uri(clusterUri, string.Format("/ApplicationTypes/{0}/$/Unprovision?api-version=
{1}",
                "WordCount",     // Application Type Name
                "1.0"));            // api-version

            HttpWebRequest request = (HttpWebRequest)WebRequest.Create(requestUri);
            request.Method = "POST";
            request.ContentType = "application/json; charset=utf-8";

            // Stores the response status code.
            HttpStatusCode statusCode;

            // Create the byte array that will become the request body.
            string requestBody = "{\"ApplicationTypeVersion\":\"" + versionToUnprovision + "\"}";
            byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBody);
            request.ContentLength = requestBodyBytes.Length;

            // Create the request body.
            try
            {
                using (Stream requestStream = request.GetRequestStream())
```

```
            {
                requestStream.Write(requestBodyBytes, 0, requestBodyBytes.Length);
                requestStream.Close();

                // Execute the request and obtain the response.
                using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
                {
                    statusCode = response.StatusCode;
                }
            }
        }
        catch (WebException e)
        {
            // If there is a web exception, display the error message.
            Console.WriteLine("Error unprovisioning the application:");
            Console.WriteLine(e.Message);
            if (e.InnerException != null)
                Console.WriteLine(e.InnerException.Message);
            return false;
        }
        catch (Exception e)
        {
            // If there is another kind of exception, throw it.
            throw (e);
        }

        Console.WriteLine("Unprovision an Application response status = " + statusCode.ToString());
        return true;
    }

    #endregion

    #region Get Application List (REST API)

    /// <summary>
    /// Gets the list of applications.
    /// </summary>
    /// <param name="clusterUri">The URI to access the cluster.</param>
    /// <returns>Returns true if successful; otherwise false.</returns>
    public static bool GetApplicationList(Uri clusterUri)
    {
        // String to capture the response stream.
        string responseString = string.Empty;

        // Create the request and add URL parameters.
        Uri requestUri = new Uri(clusterUri, string.Format("/Applications?api-version={0}",
            "1.0")); // api-version

        HttpWebRequest request = (HttpWebRequest)WebRequest.Create(requestUri);
        request.Method = "GET";

        // Execute the request and obtain the response.
        try
        {
            using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
            {
                using (StreamReader streamReader = new StreamReader(response.GetResponseStream(), true))
                {
                    // Capture the response string.
                    responseString = streamReader.ReadToEnd();
                }
            }
        }
        catch (WebException e)
        {
            // If there is a web exception, display the error message.
            Console.WriteLine("Error getting the application list:");
            Console.WriteLine(e.Message);
            if (e.InnerException != null)
```

```
                Console.WriteLine(e.InnerException.Message);
            return false;
        }
        catch (Exception e)
        {
            // If there is another kind of exception, throw it.
            throw (e);
        }


        // Deserialize the response string.
        JavaScriptSerializer jss = new JavaScriptSerializer();
        List<ApplicationInfo> applicationInfos = jss.Deserialize<List<ApplicationInfo>>(responseString);

        // Display some application information for each application.
        Console.WriteLine("Application List:");
        foreach (ApplicationInfo applicationInfo in applicationInfos)
        {
            Console.WriteLine("  Application:");
            Console.WriteLine("    Id: " + applicationInfo.Id);
            Console.WriteLine("    Name: " + applicationInfo.Name);
            Console.WriteLine("    TypeName: " + applicationInfo.TypeName);
            Console.WriteLine("    TypeVersion: " + applicationInfo.TypeVersion);
            Console.WriteLine("    Status: " + applicationInfo.Status);
            Console.WriteLine("    HealthState: " + applicationInfo.HealthState);

            Console.WriteLine("    Parameters:");

            foreach (Parameter parameter in applicationInfo.Parameters)
            {
                Console.WriteLine("      Parameter:");
                Console.WriteLine("        Name: " + parameter.Name);
                Console.WriteLine("        Value: " + parameter.Value);
            }
        }

        return true;
    }

    #endregion

    #region Create Application (REST API)

    /// <summary>
    /// Creates an application.
    /// </summary>
    /// <param name="clusterUri">The URI to access the cluster.</param>
    /// <returns>Returns true if successful; otherwise false.</returns>
    public static bool CreateApplication(Uri clusterUri)
    {
        // String to capture the response stream.
        string responseString = string.Empty;

        // Stores the response status code.
        HttpStatusCode statusCode;

        // Create the request and add URL parameters.
        Uri requestUri = new Uri(clusterUri, string.Format("/Applications/$/Create?api-version={0}",
            "1.0"));    // api-version

        HttpWebRequest request = (HttpWebRequest)WebRequest.Create(requestUri);
        request.ContentType = "text/json";
        request.Method = "POST";

        // Create the byte array that will become the request body.
        string requestBody = "{\"Name\":\"fabric:/WordCount\"," +
                                "\"TypeName\":\"WordCount\"," +
                                "\"TypeVersion\":\"1.0.0\"," +
                                "\"ParameterList\":[]}";
```

```
        byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBody);
        request.ContentLength = requestBodyBytes.Length;

        // Create the request body.
        try
        {
            using (Stream requestStream = request.GetRequestStream())
            {
                requestStream.Write(requestBodyBytes, 0, requestBodyBytes.Length);
                requestStream.Close();

                // Execute the request and obtain the response.
                using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
                {
                    statusCode = response.StatusCode;
                }
            }
        }
        catch (WebException e)
        {
            // If there is a web exception, display the error message.
            Console.WriteLine("Error creating application:");
            Console.WriteLine(e.Message);
            if (e.InnerException != null)
                Console.WriteLine(e.InnerException.Message);
            return false;
        }
        catch (Exception e)
        {
            // If there is another kind of exception, throw it.
            throw (e);
        }

        Console.WriteLine("Create Application response status = " + statusCode.ToString());

        return true;
    }

    #endregion


    #region Delete Application (REST API)

    /// <summary>
    /// Deletes an application.
    /// </summary>
    /// <param name="clusterUri">The URI to access the cluster.</param>
    /// <returns>Returns true if successful; otherwise false.</returns>
    public static bool DeleteApplication(Uri clusterUri)
    {
        // Create the request and add URL parameters.
        Uri requestUri = new Uri(clusterUri,
            string.Format("/Applications/{0}/$/Delete?api-version={1}",
            "WordCount",    // Application Name
            "1.0"));        // api-version

        HttpWebRequest request = (HttpWebRequest)WebRequest.Create(requestUri);
        request.Method = "POST";
        request.ContentLength = 0;

        // Stores the response status code.
        HttpStatusCode statusCode;

        // Execute the request and obtain the response.
        try
        {
            using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
            {
                statusCode = response.StatusCode;
```

```
            statusCode = response.StatusCode;
        }
    }
    catch (WebException e)
    {
        // If there is a web exception, display the error message.
        Console.WriteLine("Error deleting application:");
        Console.WriteLine(e.Message);
        if (e.InnerException != null)
            Console.WriteLine(e.InnerException.Message);
        return false;
    }
    catch (Exception e)
    {
        // If there is another kind of exception, throw it.
        throw (e);
    }

    Console.WriteLine("Delete Application response status = " + statusCode.ToString());

    return true;
}

#endregion

#region Upgrade Application by Application Type (REST API)

/// <summary>
/// Upgrades an application by application type.
/// </summary>
/// <param name="clusterUri">The URI to access the cluster.</param>
/// <returns>Returns true if successful; otherwise false.</returns>
public static bool UpgradeApplicationByApplicationType(Uri clusterUri)
{
    // String to capture the response stream.
    string responseString = string.Empty;

    // Stores the response status code.
    HttpStatusCode statusCode;

    // Create the request and add URL parameters.
    Uri requestUri = new Uri(clusterUri, string.Format("/Applications/{0}/$/Upgrade?api-version={1}",
        "WordCount",     // Application Name
        "1.0"));                // api-version

    HttpWebRequest request = (HttpWebRequest)WebRequest.Create(requestUri);
    request.ContentType = "text/json";
    request.Method = "POST";


    // Create the Health Policy.
    string requestBody = "{\"Name\":\"fabric:/WordCount\"," +
                          "\"TargetApplicationTypeVersion\":\"1.1.0\"," +
                          "\"Parameters\":[]," +
                          "\"UpgradeKind\":1," +
                          "\"RollingUpgradeMode\":1," +
                          "\"UpgradeReplicaSetCheckTimeoutInSeconds\":5," +
                          "\"ForceRestart\":true," +
                          "\"MonitoringPolicy\":" +
                          "{\"FailureAction\":1," +
                          "\"HealthCheckWaitDurationInMilliseconds\":\"5000\"," +
                          "\"HealthCheckStableDurationInMilliseconds\":\"10000\"," +
                          "\"HealthCheckRetryTimeoutInMilliseconds\":\"20000\"," +
                          "\"UpgradeTimeoutInMilliseconds\":\"60000\"," +
                          "\"UpgradeDomainTimeoutInMilliseconds\":\"30000\"}}";

    // Create the byte array that will become the request body.
    byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBody);
    request.ContentLength = requestBodyBytes.Length;
```

```
            // Create the request body.
            try
            {
                using (Stream requestStream = request.GetRequestStream())
                {
                    requestStream.Write(requestBodyBytes, 0, requestBodyBytes.Length);
                    requestStream.Close();

                    // Execute the request and obtain the response.
                    using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
                    {
                        statusCode = response.StatusCode;
                    }
                }
            }
            catch (WebException e)
            {
                // If there is a web exception, display the error message.
                Console.WriteLine("Error upgrading application:");
                Console.WriteLine(e.Message);
                if (e.InnerException != null)
                    Console.WriteLine(e.InnerException.Message);
                return false;
            }
            catch (Exception e)
            {
                // If there is another kind of exception, throw it.
                throw (e);
            }

            Console.WriteLine("Update Application response status = " + statusCode.ToString());

            return true;
        }

        #endregion
    }
}
```

Next steps

Service Fabric application lifecycle

Introduction to Service Fabric health monitoring

11/15/2016 • 20 min to read • Edit on GitHub

Contributors

Oana Platon • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • MarkDeichmiller

Azure Service Fabric introduces a health model that provides rich, flexible, and extensible health evaluation and reporting. The model allows near-real-time monitoring of the state of the cluster and the services running in it. You can easily obtain health information and correct potential issues before they cascade and cause massive outages. In the typical model, services send reports based on their local views, and that information is aggregated to provide an overall cluster-level view.

Service Fabric components use this rich health model to report their current state. You can use the same mechanism to report health from your applications. If you invest in high-quality health reporting that captures your custom conditions, you can detect and fix issues for your running application much more easily.

> **NOTE**
> We started the health subsystem to address a need for monitored upgrades. Service Fabric provides monitored application and cluster upgrades that ensure full availability, no downtime and minimal to no user intervention. To achieve these goals, the upgrade checks health based on configured upgrade policies and allows an upgrade to proceed only when health respects desired thresholds. Otherwise, the upgrade is either automatically rolled back or paused to give administrators a chance to fix the issues. To learn more about application upgrades, see this article.

Health store

The health store keeps health-related information about entities in the cluster for easy retrieval and evaluation. It is implemented as a Service Fabric persisted stateful service to ensure high availability and scalability. The health store is part of the **fabric:/System** application, and it is available when the cluster is up and running.

Health entities and hierarchy

The health entities are organized in a logical hierarchy that captures interactions and dependencies among different entities. The entities and hierarchy are automatically built by the health store based on reports received from Service Fabric components.

The health entities mirror the Service Fabric entities. (For example, **health application entity** matches an application instance deployed in the cluster, while **health node entity** matches a Service Fabric cluster node.) The health hierarchy captures the interactions of the system entities, and it is the basis for advanced health evaluation. You can learn about key Service Fabric concepts in Service Fabric technical overview. For more on application, see Service Fabric application model.

The health entities and hierarchy allow the cluster and applications to be effectively reported, debugged, and monitored. The health model provides an accurate, *granular* representation of the health of the many moving pieces in the cluster.

The health entities, organized in a hierarchy based on parent-child relationships.

The health entities are:

- **Cluster**. Represents the health of a Service Fabric cluster. Cluster health reports describe conditions that affect the entire cluster and can't be narrowed down to one or more unhealthy children. Examples include the brain of the cluster splitting due to network partitioning or communication issues.
- **Node**. Represents the health of a Service Fabric node. Node health reports describe conditions that affect the node functionality. They typically affect all the deployed entities running on it. Examples include when a node is out of disk space (or another machine-wide property, such as memory, connections) and when a node is down. The node entity is identified by the node name (string).
- **Application**. Represents the health of an application instance running in the cluster. Application health reports describe conditions that affect the overall health of the application. They can't be narrowed down to individual children (services or deployed applications). Examples include the end-to-end interaction among different services in the application. The application entity is identified by the application name (URI).
- **Service**. Represents the health of a service running in the cluster. Service health reports describe conditions that affect the overall health of the service, and they can't be narrowed down to a partition or a replica. Examples include a service configuration (such as port or external file share) that is causing issues for all partitions. The service entity is identified by the service name (URI).
- **Partition**. Represents the health of a service partition. Partition health reports describe conditions that affect the entire replica set. Examples include when the number of replicas is below target count and when a partition is in quorum loss. The partition entity is identified by the partition ID (GUID).
- **Replica**. Represents the health of a stateful service replica or a stateless service instance. The smallest unit that watchdogs and system components can report on for an application. For stateful services, examples include a primary replica reporting when it can't replicate operations to secondaries and when replication is not proceeding at the expected pace. Also, a stateless instance can report when it is running out of resources or has connectivity issues. The replica entity is identified by the partition ID (GUID) and the replica or instance ID (long).
- **DeployedApplication**. Represents the health of an *application running on a node*. Deployed application health reports describe conditions specific to the application on the node that can't be narrowed down to service packages deployed on the same node. Examples include when the application package can't be downloaded on that node and when there is an issue setting up application security principals on the node.

The deployed application is identified by application name (URI) and node name (string).

- **DeployedServicePackage**. Represents the health of a service package running on a node in the cluster. It describes conditions specific to a service package that do not affect the other service packages on the same node for the same application. Examples include a code package in the service package that cannot be started and a configuration package that cannot be read. The deployed service package is identified by application name (URI), node name (string), and service manifest name (string).

The granularity of the health model makes it easy to detect and correct issues. For example, if a service is not responding, it is feasible to report that the application instance is unhealthy. Reporting at that level is not ideal, however, because the issue might not be affecting all the services within that application. The report should be applied to the unhealthy service or to a specific child partition, if more information points to that partition. The data automatically surfaces through the hierarchy, and an unhealthy partition is made visible at service and application levels. This aggregation helps to pinpoint and resolve the root cause of the issue more quickly.

The health hierarchy is composed of parent-child relationships. A cluster is composed of nodes and applications. Applications have services and deployed applications. Deployed applications have deployed service packages. Services have partitions, and each partition has one or more replicas. There is a special relationship between nodes and deployed entities. If a node is unhealthy as reported by its authority system component (Failover Manager service), it affects the deployed applications, service packages, and replicas deployed on it.

The health hierarchy represents the latest state of the system based on the latest health reports, which is almost real-time information. Internal and external watchdogs can report on the same entities based on application-specific logic or custom monitored conditions. User reports coexist with the system reports.

Plan to invest in how to report and respond to health during the design of a large cloud service, to make the service easier to debug, monitor, and operate.

Health states

Service Fabric uses three health states to describe whether an entity is healthy or not: OK, warning, and error. Any report sent to the health store must specify one of these states. The health evaluation result is one of these states.

The possible health states are:

- **OK**. The entity is healthy. There are no known issues reported on it or its children (when applicable).
- **Warning**. The entity experiences some issues, but it is not yet unhealthy (for example, there are delays, but they do not cause any functional issues yet). In some cases, the warning condition may fix itself without any special intervention, and it is useful to provide visibility into what is going on. In other cases, the warning condition may degrade into a severe problem without user intervention.
- **Error**. The entity is unhealthy. Action should be taken to fix the state of the entity, because it can't function properly.
- **Unknown**. The entity doesn't exist in the health store. This result can be obtained from the distributed queries that merge results from multiple components. For example, get node list query goes to **FailoverManager** and **HealthManager**, get application list query goes to **ClusterManager** and **HealthManager**. These queries merge results from multiple system components. If another system component returns an entity that has not reached or has been cleaned up from the health store, the merged result has unknown health state.

Health policies

The health store applies health policies to determine whether an entity is healthy based on its reports and its children.

> **NOTE**
> Health policies can be specified in the cluster manifest (for cluster and node health evaluation) or in the application manifest (for application evaluation and any of its children). Health evaluation requests can also pass in custom health evaluation policies, which is used only for that evaluation.

By default, Service Fabric applies strict rules (everything must be healthy) for the parent-child hierarchical relationship. If even one of the children has one unhealthy event, the parent is considered unhealthy.

Cluster health policy

The cluster health policy is used to evaluate the cluster health state and node health states. The policy can be defined in the cluster manifest. If it is not present, the default policy (zero tolerated failures) is used. The cluster health policy contains:

- ConsiderWarningAsError. Specifies whether to treat warning health reports as errors during health evaluation. Default: false.
- MaxPercentUnhealthyApplications. Specifies the maximum tolerated percentage of applications that can be unhealthy before the cluster is considered in error.
- MaxPercentUnhealthyNodes. Specifies the maximum tolerated percentage of nodes that can be unhealthy before the cluster is considered in error. In large clusters, some nodes are always down or out for repairs, so this percentage should be configured to tolerate that.
- ApplicationTypeHealthPolicyMap. The application type health policy map can be used during cluster health evaluation to describe special application types. By default, all applications are put into a pool and evaluated with MaxPercentUnhealthyApplications. If some application types should be treated differently, they can be taken out of the global pool. Instead, they are evaluated against the percentages associated with their application type name in the map. For example, in a cluster there are thousands of applications of different types, and a few control application instances of a special application type. The control applications should never be in error. You can specify global MaxPercentUnhealthyApplications to 20% to tolerate some failures, but for the application type "ControlApplicationType" set the MaxPercentUnhealthyApplications to 0. This way, if some of the many applications are unhealthy, but below the global unhealthy percentage, the cluster would be evaluated to Warning. A warning health state does not impact cluster upgrade or other monitoring triggered by Error health state. But even one control application in error would make cluster unhealthy, which triggers roll back or pauses the cluster upgrade, depending on the upgrade configuration. For the application types defined in the map, all application instances are taken out of the global pool of applications. They are evaluated based on the total number of applications of the application type, using the specific MaxPercentUnhealthyApplications from the map. All the rest of the applications remain in the global pool and are evaluated with MaxPercentUnhealthyApplications.

The following example is an excerpt from a cluster manifest. To define entries in the application type map, prefix the parameter name with "ApplicationTypeMaxPercentUnhealthyApplications-", followed by the application type name.

```
<FabricSettings>
  <Section Name="HealthManager/ClusterHealthPolicy">
    <Parameter Name="ConsiderWarningAsError" Value="False" />
    <Parameter Name="MaxPercentUnhealthyApplications" Value="20" />
    <Parameter Name="MaxPercentUnhealthyNodes" Value="20" />
    <Parameter Name="ApplicationTypeMaxPercentUnhealthyApplications-ControlApplicationType" Value="0" />
  </Section>
</FabricSettings>
```

Application health policy

The application health policy describes how the evaluation of events and child-states aggregation is done for applications and their children. It can be defined in the application manifest, **ApplicationManifest.xml**, in the

application package. If no policies are specified, Service Fabric assumes that the entity is unhealthy if it has a health report or a child at the warning or error health state. The configurable policies are:

- ConsiderWarningAsError. Specifies whether to treat warning health reports as errors during health evaluation. Default: false.
- MaxPercentUnhealthyDeployedApplications. Specifies the maximum tolerated percentage of deployed applications that can be unhealthy before the application is considered in error. This percentage is calculated by dividing the number of unhealthy deployed applications over the number of nodes that the applications are currently deployed on in the cluster. The computation rounds up to tolerate one failure on small numbers of nodes. Default percentage: zero.
- DefaultServiceTypeHealthPolicy. Specifies the default service type health policy, which replaces the default health policy for all service types in the application.
- ServiceTypeHealthPolicyMap. Provides a map of service health policies per service type. These policies replace the default service type health policies for each specified service type. For example, if an application has a stateless gateway service type and a stateful engine service type, you can configure the health policies for their evaluation differently. When you specify policy per service type, you can gain more granular control of the health of the service.

Service type health policy

The service type health policy specifies how to evaluate and aggregate the services and the children of services. The policy contains:

- MaxPercentUnhealthyPartitionsPerService. Specifies the maximum tolerated percentage of unhealthy partitions before a service is considered unhealthy. Default percentage: zero.
- MaxPercentUnhealthyReplicasPerPartition. Specifies the maximum tolerated percentage of unhealthy replicas before a partition is considered unhealthy. Default percentage: zero.
- MaxPercentUnhealthyServices. Specifies the maximum tolerated percentage of unhealthy services before the application is considered unhealthy. Default percentage: zero.

The following example is an excerpt from an application manifest:

```
<Policies>
    <HealthPolicy ConsiderWarningAsError="true" MaxPercentUnhealthyDeployedApplications="20">
        <DefaultServiceTypeHealthPolicy
               MaxPercentUnhealthyServices="0"
               MaxPercentUnhealthyPartitionsPerService="10"
               MaxPercentUnhealthyReplicasPerPartition="0"/>
        <ServiceTypeHealthPolicy ServiceTypeName="FrontEndServiceType"
               MaxPercentUnhealthyServices="0"
               MaxPercentUnhealthyPartitionsPerService="20"
               MaxPercentUnhealthyReplicasPerPartition="0"/>
        <ServiceTypeHealthPolicy ServiceTypeName="BackEndServiceType"
               MaxPercentUnhealthyServices="20"
               MaxPercentUnhealthyPartitionsPerService="0"
               MaxPercentUnhealthyReplicasPerPartition="0">
        </ServiceTypeHealthPolicy>
    </HealthPolicy>
</Policies>
```

Health evaluation

Users and automated services can evaluate health for any entity at any time. To evaluate an entity's health, the health store aggregates all health reports on the entity and evaluates all its children (when applicable). The health aggregation algorithm uses health policies that specify how to evaluate health reports and how to aggregate child health states (when applicable).

Health report aggregation

One entity can have multiple health reports sent by different reporters (system components or watchdogs) on different properties. The aggregation uses the associated health policies, in particular the ConsiderWarningAsError member of application or cluster health policy. ConsiderWarningAsError specifies how to evaluate warnings.

The aggregated health state is triggered by the *worst* health reports on the entity. If there is at least one error health report, the aggregated health state is an error.

A health entity that has one or more error health reports is evaluated as Error. The same is true for an expired health report, regardless of its health state.

If there are no error reports and one or more warnings, the aggregated health state is either warning or error, depending on the ConsiderWarningAsError policy flag.

Health report aggregation with warning report and ConsiderWarningAsError set to false (default).

Child health aggregation

The aggregated health state of an entity reflects the child health states (when applicable). The algorithm for aggregating child health states uses the health policies applicable based on the entity type.

Child aggregation based on health policies.

After the health store has evaluated all the children, it aggregates their health states based on the configured maximum percentage of unhealthy children. This percentage is taken from the policy based on the entity and child type.

- If all children have OK states, the child aggregated health state is OK.
- If children have both OK and warning states, the child aggregated health state is warning.
- If there are children with error states that do not respect the maximum allowed percentage of unhealthy children, the aggregated health state is an error.
- If the children with error states respect the maximum allowed percentage of unhealthy children, the aggregated health state is warning.

Health reporting

System components, System Fabric applications, and internal/external watchdogs can report against Service Fabric entities. The reporters make *local* determinations of the health of the monitored entities, based on the conditions they are monitoring. They don't need to look at any global state or aggregate data. The desired behavior is to have simple reporters, and not complex organisms that need to look at many things to infer what information to send.

To send health data to the health store, a reporter needs to identify the affected entity and create a health report. The report can then be sent through the API by using FabricClient.HealthClient.ReportHealth, through PowerShell, or through REST.

Health reports

The health reports for each of the entities in the cluster contain the following information:

- **SourceId**. A string that uniquely identifies the reporter of the health event.
- **Entity identifier**. Identifies the entity where the report is applied. It differs based on the entity type:
  - Cluster. None.
  - Node. Node name (string).
  - Application. Application name (URI). Represents the name of the application instance deployed in the cluster.

  - Service. Service name (URI). Represents the name of the service instance deployed in the cluster.
  - Partition. Partition ID (GUID). Represents the partition unique identifier.
  - Replica. The stateful service replica ID or the stateless service instance ID (INT64).
  - DeployedApplication. Application name (URI) and node name (string).
  - DeployedServicePackage. Application name (URI), node name (string), and service manifest name (string).
- **Property**. A *string* (not a fixed enumeration) that allows the reporter to categorize the health event for a specific property of the entity. For example, reporter A can report the health of the Node01 "storage" property and reporter B can report the health of the Node01 "connectivity" property. In the health store, these reports are treated as separate health events for the Node01 entity.
- **Description**. A string that allows a reporter to provide detailed information about the health event. **SourceId**, **Property**, and **HealthState** should fully describe the report. The description adds human-readable information about the report. The text makes it easier for administrators and users to understand the health report.
- **HealthState**. An enumeration that describes the health state of the report. The accepted values are OK, Warning, and Error.
- **TimeToLive**. A timespan that indicates how long the health report is valid. Coupled with **RemoveWhenExpired**, it lets the health store know how to evaluate expired events. By default, the value is infinite, and the report is valid forever.
- **RemoveWhenExpired**. A Boolean. If set to true, the expired health report is automatically removed from the health store, and the report doesn't impact entity health evaluation. Used when the report is valid for a specified period of time only, and the reporter doesn't need to explicitly clear it out. It's also used to delete reports from the health store (for example, a watchdog is changed and stops sending reports with previous source and property). It can send a report with a brief TimeToLive along with RemoveWhenExpired to clear up any previous state from the health store. If the value is set to false, the expired report is treated as an error on the health evaluation. The false value signals to the health store that the source should report periodically on this property. If it doesn't, then there must be something wrong with the watchdog. The watchdog's health is captured by considering the event as an error.
- **SequenceNumber**. A positive integer that needs to be ever-increasing, it represents the order of the reports. It is used by the health store to detect stale reports that are received late because of network delays or other issues. A report is rejected if the sequence number is less than or equal to the most recently applied number for the same entity, source, and property. If it is not specified, the sequence number is generated automatically. It is necessary to put in the sequence number only when reporting on state transitions. In this situation, the source needs to remember which reports it sent and keep the information for recovery on failover.

These four pieces of information--SourceId, entity identifier, Property, and HealthState--are required for every health report. The SourceId string is not allowed to start with the prefix "**System.**", which is reserved for system reports. For the same entity, there is only one report for the same source and property. Multiple reports for the same source and property override each other, either on the health client side (if they are batched) or on the health store side. The replacement is based on sequence numbers; newer reports (with higher sequence numbers) replace older reports.

Health events

Internally, the health store keeps health events, which contain all the information from the reports, and additional metadata. The metadata includes the time the report was given to the health client and the time it was modified on the server side. The health events are returned by health queries.

The added metadata contains:

- **SourceUtcTimestamp**. The time the report was given to the health client (Coordinated Universal Time).
- **LastModifiedUtcTimestamp**. The time the report was last modified on the server side (Coordinated

Universal Time).

- **IsExpired**. A flag to indicate whether the report was expired when the query was executed by the health store. An event can be expired only if RemoveWhenExpired is false. Otherwise, the event is not returned by query and is removed from the store.
- **LastOkTransitionAt**, **LastWarningTransitionAt**, **LastErrorTransitionAt**. The last time for OK/warning/error transitions. These fields give the history of the health state transitions for the event.

The state transition fields can be used for smarter alerts or "historical" health event information. They enable scenarios such as:

- Alert when a property has been at warning/error for more than X minutes. Checking the condition for a period of time avoids alerts on temporary conditions. For example, an alert if the health state has been warning for more than five minutes can be translated into (HealthState == Warning and Now - LastWarningTransitionTime > 5 minutes).
- Alert only on conditions that have changed in the last X minutes. If a report was already at error before the specified time, it can be ignored because it was already signaled previously.
- If a property is toggling between warning and error, determine how long it has been unhealthy (that is, not OK). For example, an alert if the property hasn't been healthy for more than five minutes can be translated into (HealthState != Ok and Now - LastOkTransitionTime > 5 minutes).

Example: Report and evaluate application health

The following example sends a health report through PowerShell on the application **fabric:/WordCount** from the source **MyWatchdog**. The health report contains information about the health property "availability" in an error health state, with infinite TimeToLive. Then it queries the application health, which returns aggregated health state errors and the reported health events in the list of health events.

```
PS C:\> Send-ServiceFabricApplicationHealthReport –ApplicationName fabric:/WordCount –SourceId "MyWatchdog" –
HealthProperty "Availability" –HealthState Error

PS C:\> Get-ServiceFabricApplicationHealth fabric:/WordCount


ApplicationName                 : fabric:/WordCount
AggregatedHealthState           : Error
UnhealthyEvaluations            :
                                  Error event: SourceId='MyWatchdog', Property='Availability'.

ServiceHealthStates             :
                                  ServiceName           : fabric:/WordCount/WordCountService
                                  AggregatedHealthState : Error

                                  ServiceName           : fabric:/WordCount/WordCountWebService
                                  AggregatedHealthState : Ok

DeployedApplicationHealthStates :
                                  ApplicationName       : fabric:/WordCount
                                  NodeName              : _Node_0
                                  AggregatedHealthState : Ok

                                  ApplicationName       : fabric:/WordCount
                                  NodeName              : _Node_2
                                  AggregatedHealthState : Ok

                                  ApplicationName       : fabric:/WordCount
                                  NodeName              : _Node_3
                                  AggregatedHealthState : Ok

                                  ApplicationName       : fabric:/WordCount
                                  NodeName              : _Node_4
                                  AggregatedHealthState : Ok

                                  ApplicationName       : fabric:/WordCount
                                  NodeName              : _Node_1
                                  AggregatedHealthState : Ok

HealthEvents                    :
                                  SourceId              : System.CM
                                  Property              : State
                                  HealthState           : Ok
                                  SequenceNumber        : 360
                                  SentAt                : 3/22/2016 7:56:53 PM
                                  ReceivedAt            : 3/22/2016 7:56:53 PM
                                  TTL                   : Infinite
                                  Description           : Application has been created.
                                  RemoveWhenExpired     : False
                                  IsExpired             : False
                                  Transitions           : Error->Ok = 3/22/2016 7:56:53 PM, LastWarning =
1/1/0001 12:00:00 AM

                                  SourceId              : MyWatchdog
                                  Property              : Availability
                                  HealthState           : Error
                                  SequenceNumber        : 131032204762818013
                                  SentAt                : 3/23/2016 3:27:56 PM
                                  ReceivedAt            : 3/23/2016 3:27:56 PM
                                  TTL                   : Infinite
                                  Description           :
                                  RemoveWhenExpired     : False
                                  IsExpired             : False
                                  Transitions           : Ok->Error = 3/23/2016 3:27:56 PM, LastWarning =
1/1/0001 12:00:00 AM
```

Health model usage

The health model allows cloud services and the underlying Service Fabric platform to scale, because monitoring and health determinations are distributed among the different monitors within the cluster. Other systems have a single, centralized service at the cluster level that parses all the *potentially* useful information emitted by services. This approach hinders their scalability. It also doesn't allow them to collect very specific information to help identify issues and potential issues as close to the root cause as possible.

The health model is used heavily for monitoring and diagnosis, for evaluating cluster and application health, and for monitored upgrades. Other services use health data to perform automatic repairs, build cluster health history, and issue alerts on certain conditions.

Next steps

View Service Fabric health reports

Use system health reports for troubleshooting

How to report and check service health

Add custom Service Fabric health reports

Monitor and diagnose services locally

Service Fabric application upgrade

Report and check service health

11/15/2016 • 5 min to read • Edit on GitHub

Contributors

Todd Abel • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Todd Abel • Joseph Molnar • Pushkar Newaskar • Mark Fussell • Rebecca Agiewich • kunaldsingh

When your services encounter problems, your ability to respond to and fix incidents and outages depends on your ability to detect the issues quickly. If you report problems and failures to the Azure Service Fabric health manager from your service code, you can use standard health monitoring tools that Service Fabric provides to check the health status.

There are two ways that you can report health from the service:

- Use Partition or CodePackageActivationContext objects.
  You can use the `Partition` and `CodePackageActivationContext` objects to report the health of elements that are part of the current context. For example, code that runs as part of a replica can report health only on that replica, the partition that it belongs to, and the application that it is a part of.
- Use `FabricClient`.
  You can use `FabricClient` to report health from the service code if the cluster is not secure or if the service is running with admin privileges. This won't be true in most real-world scenarios. With `FabricClient`, you can report health on any entity that is a part of the cluster. Ideally, however, service code should only send reports that are related to its own health.

This article walks you through an example that reports health from the service code. The example also shows how the tools that Service Fabric provides can be used to check the health status. This article is intended to be a quick introduction to the health monitoring capabilities of Service Fabric. For more detailed information, you can read the series of in-depth articles about health that start with the link at the end of this article.

Prerequisites

You must have the following installed:

- Visual Studio 2015
- Service Fabric SDK

To create a local secure dev cluster

- Open PowerShell with admin privileges, and run the following commands.

```
C:\Users\punewa> cd "C:\Program Files\Microsoft SDKs\Service Fabric\ClusterSetup"
C:\Program Files\Microsoft SDKs\Service Fabric\ClusterSetup> .\DevClusterSetup.ps1 -AsSecureCluster
```

To deploy an application and check its health

1. Open Visual Studio as an administrator.
2. Create a project by using the **Stateful Service** template.

3. Press **F5** to run the application in debug mode. The application will be deployed to the local cluster.
4. After the application is running, right-click the Local Cluster Manager icon in the notification area and select **Manage Local Cluster** from the shortcut menu to open Service Fabric Explorer.

5. The application health should be displayed as in this image. At this time, the application should be healthy with no errors.

6. You can also check the health by using PowerShell. You can use `Get-ServiceFabricApplicationHealth` to check an application's health, and you can use `Get-ServiceFabricServiceHealth` to check a service's health. The health report for the same application in PowerShell is in this image.

```
PS C:\Users\kunalds> Get-ServiceFabricApplicationHealth -ApplicationName fabric:/Application10


ApplicationName                : fabric:/Application10
AggregatedHealthState          : Ok
ServiceHealthStates            :
                                 ServiceName           : fabric:/Application10/Stateful1
                                 AggregatedHealthState : Ok

DeployedApplicationHealthStates :
                                 ApplicationName       : fabric:/Application10
                                 NodeName              : Node.2
                                 AggregatedHealthState : Ok

                                 ApplicationName       : fabric:/Application10
                                 NodeName              : Node.1
                                 AggregatedHealthState : Ok

                                 ApplicationName       : fabric:/Application10
                                 NodeName              : Node.4
                                 AggregatedHealthState : Ok

HealthEvents                   :
                                 SourceId              : System.CM
                                 Property              : State
                                 HealthState           : Ok
                                 SequenceNumber        : 264255
                                 SentAt                : 10/28/2015 5:48:18 AM
                                 ReceivedAt            : 10/28/2015 5:48:18 AM
                                 TTL                   : Infinite
                                 Description           : Application has been created.
                                 RemoveWhenExpired     : False
                                 IsExpired             : False
                                 Transitions           : ->Ok = 10/28/2015 5:48:18 AM
```

To add custom health events to your service code

The Service Fabric project templates in Visual Studio contain sample code. The following steps show how you can report custom health events from your service code. Such reports will automatically show up in the standard tools for health monitoring that Service Fabric provides, such as Service Fabric Explorer, Azure portal health view, and PowerShell.

1. Reopen the application that you created previously in Visual Studio, or create a new application by using the **Stateful Service** Visual Studio template.

2. Open the Stateful1.cs file, and find the `myDictionary.TryGetValueAsync` call in the `RunAsync` method. You can see that this method returns a `result` that holds the current value of the counter because the key logic in this application is to keep a count running. If this were a real application, and if the lack of result represented a failure, you would want to flag that event.
3. To report a health event when the lack of result represents a failure, add the following steps.

   a. Add the `System.Fabric.Health` namespace to the Stateful1.cs file.

   ```
   using System.Fabric.Health;
   ```

   b. Add the following code after the `myDictionary.TryGetValueAsync` call

   ```
   if (!result.HasValue)
   {
       HealthInformation healthInformation = new HealthInformation("ServiceCode", "StateDictionary",
   HealthState.Error);
       this.Partition.ReportReplicaHealth(healthInformation);
   }
   ```

   We report replica health because it's being reported from a stateful service. The `HealthInformation` parameter stores information about the health issue that's being reported.

   If you had created a stateless service, use the following code

   ```
   if (!result.HasValue)
   {
       HealthInformation healthInformation = new HealthInformation("ServiceCode", "StateDictionary",
   HealthState.Error);
       this.Partition.ReportInstanceHealth(healthInformation);
   }
   ```

4. If your service is running with admin privileges or if the cluster is not secure, you can also use `FabricClient` to report health as shown in the following steps.

   a. Create the `FabricClient` instance after the `var myDictionary` declaration.

   ```
   var fabricClient = new FabricClient(new FabricClientSettings() { HealthReportSendInterval =
   TimeSpan.FromSeconds(0) });
   ```

   b. Add the following code after the `myDictionary.TryGetValueAsync` call.

   ```
   if (!result.HasValue)
   {
      var replicaHealthReport = new StatefulServiceReplicaHealthReport(
           this.ServiceInitializationParameters.PartitionId,
           this.ServiceInitializationParameters.ReplicaId,
           new HealthInformation("ServiceCode", "StateDictionary", HealthState.Error));
       fabricClient.HealthManager.ReportHealth(replicaHealthReport);
   }
   ```

5. Let's simulate this failure and see it show up in the health monitoring tools. To simulate the failure, comment out the first line in the health reporting code that you added earlier. After you comment out the first line, the code will look like the following example.

```
//if(!result.HasValue)
{
    HealthInformation healthInformation = new HealthInformation("ServiceCode", "StateDictionary",
HealthState.Error);
    this.Partition.ReportReplicaHealth(healthInformation);
}
```

This code will now fire this health report each time `RunAsync` executes. After you make the change, press **F5** to run the application.

6. After the application is running, open Service Fabric Explorer to check the health of the application. This time, Service Fabric Explorer will show that the application is unhealthy. This is because of the error that was reported from the code that we added previously.

7. If you select the primary replica in the tree view of Service Fabric Explorer, you will see that **Health State** indicates an error, too. Service Fabric Explorer also displays the health report details that were added to the `HealthInformation` parameter in the code. You can see the same health reports in PowerShell and the Azure portal.

This report will remain in the health manager until it is replaced by another report or until this replica is deleted. Because we did not set `TimeToLive` for this health report in the `HealthInformation` object, the report will never expire.

We recommend that health should be reported on the most granular level, which in this case is the replica. You can also report health on `Partition`.

```
HealthInformation healthInformation = new HealthInformation("ServiceCode", "StateDictionary",
HealthState.Error);
this.Partition.ReportPartitionHealth(healthInformation);
```

To report health on `Application`, `DeployedApplication`, and `DeployedServicePackage`, use `CodePackageActivationContext`.

```
HealthInformation healthInformation = new HealthInformation("ServiceCode", "StateDictionary",
HealthState.Error);
var activationContext = FabricRuntime.GetActivationContext();
activationContext.ReportApplicationHealth(healthInformation);
```

Next steps

Deep dive on Service Fabric health

Add custom Service Fabric health reports

11/15/2016 • 19 min to read • Edit on GitHub

Contributors

Oana Platon • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Wike • MarkDeichmiller

Azure Service Fabric introduces a health model designed to flag unhealthy cluster and application conditions on specific entities. The health model uses **health reporters** (system components and watchdogs). The goal is easy and fast diagnosis and repair. Service writers need to think upfront about health. Any condition that can impact health should be reported on, especially if it can help flag problems close to the root. The health information can save a lot of time and effort on debugging and investigation once the service is up and running at scale in the cloud (private or Azure).

The Service Fabric reporters monitor identified conditions of interest. They report on those conditions based on their local view. The health store aggregates health data sent by all reporters to determine whether entities are globally healthy. The model is intended to be rich, flexible, and easy to use. The quality of the health reports determines the accuracy of the health view of the cluster. False positives that wrongly show unhealthy issues can negatively impact upgrades or other services that use health data. Examples of such services are repair services and alerting mechanisms. Therefore, some thought is needed to provide reports that capture conditions of interest in the best possible way.

To design and implement health reporting, watchdogs, and system components must:

- Define the condition they are interested in, the way it is monitored, and the impact on the cluster or application functionality. Based on this information, decide on the health report property and health state.
- Determine the entity that the report applies to.
- Determine where the reporting is done, from within the service or from an internal or external watchdog.
- Define a source used to identify the reporter.
- Choose a reporting strategy, either periodically or on transitions. The recommended way is periodically, as it requires simpler code and is less prone to errors.
- Determine how long the report for unhealthy conditions should stay in the health store and how it should be cleared. Using this information, decide the report's time to live and remove-on-expiration behavior.

As mentioned, reporting can be done from:

- The monitored Service Fabric service replica.
- Internal watchdogs deployed as a Service Fabric service (for example, a Service Fabric stateless service that monitors conditions and issues reports). The watchdogs can be deployed an all nodes or can be affinitized to the monitored service.
- Internal watchdogs that run on the Service Fabric nodes but are *not* implemented as Service Fabric services.
- External watchdogs that probe the resource from *outside* the Service Fabric cluster (for example, monitoring service like Gomez).

> **NOTE**
>
> Out of the box, the cluster is populated with health reports sent by the system components. Read more at Using system health reports for troubleshooting. The user reports must be sent on health entities that have already been created by the system.

Once the health reporting design is clear, health reports can be sent easily. You can use FabricClient to report health if the cluster is not secure or if the fabric client has admin privileges. This can be done through the API by using FabricClient.HealthManager.ReportHealth, through PowerShell, or through REST. Configuration knobs batch reports for improved performance.

> **NOTE**
>
> Report health is synchronous, and it represents only the validation work on the client side. The fact that the report is accepted by the health client or the `Partition` or `CodePackageActivationContext` objects doesn't mean that it is applied in the store. It is sent asynchronously and possibly batched with other reports. The processing on the server may still fail: the sequence number could be stale, the entity on which the report must be applied has been deleted, etc.

Health client

The health reports are sent to the health store through a health client, which lives inside the fabric client. The health client can be configured with the following:

- **HealthReportSendInterval**: The delay between the time the report is added to the client and the time it is sent to the health store. Used to batch reports into a single message, rather than sending one message for each report. The batching improves performance. Default: 30 seconds.
- **HealthReportRetrySendInterval**: The interval at which the health client resends accumulated health reports to the health store. Default: 30 seconds.
- **HealthOperationTimeout**: The timeout period for a report message sent to the health store. If a message times out, the health client retries it until the health store confirms that the report has been processed. Default: two minutes.

> **NOTE**
>
> When the reports are batched, the fabric client must be kept alive for at least the HealthReportSendInterval to ensure that they are sent. If the message is lost or the health store cannot apply them due to transient errors, the fabric client must be kept alive longer to give it a chance to retry.

The buffering on the client takes the uniqueness of the reports into consideration. For example, if a particular bad reporter is reporting 100 reports per second on the same property of the same entity, the reports are replaced with the last version. Only one such report exists in the client queue at most. If batching is configured, the number of reports sent to the health store is just one per send interval. This report is the last added report, which reflects the most current state of the entity. All configuration parameters can be specified when `FabricClient` is created by passing FabricClientSettings with the desired values for health-related entries.

The following creates a fabric client and specifies that the reports should be sent when they are added. On timeouts and errors that can be retried, retries happen every 40 seconds.

```
var clientSettings = new FabricClientSettings()
{
    HealthOperationTimeout = TimeSpan.FromSeconds(120),
    HealthReportSendInterval = TimeSpan.FromSeconds(0),
    HealthReportRetrySendInterval = TimeSpan.FromSeconds(40),
};
var fabricClient = new FabricClient(clientSettings);
```

Same parameters can be specified when a connection to a cluster is created through PowerShell. The following starts a connection to a local cluster:

```
PS C:\> Connect-ServiceFabricCluster -HealthOperationTimeoutInSec 120 -HealthReportSendIntervalInSec 0 -
HealthReportRetrySendIntervalInSec 40
True

ConnectionEndpoint   :
FabricClientSettings : {
                       ClientFriendlyName                    : PowerShell-1944858a-4c6d-465f-89c7-9021c12ac0bb
                       PartitionLocationCacheLimit           : 100000
                       PartitionLocationCacheBucketCount     : 1024
                       ServiceChangePollInterval             : 00:02:00
                       ConnectionInitializationTimeout       : 00:00:02
                       KeepAliveInterval                     : 00:00:20
                       HealthOperationTimeout                : 00:02:00
                       HealthReportSendInterval              : 00:00:00
                       HealthReportRetrySendInterval         : 00:00:40
                       NotificationGatewayConnectionTimeout : 00:00:00
                       NotificationCacheUpdateTimeout        : 00:00:00
                       }
GatewayInformation   : {
                       NodeAddress                           : localhost:19000
                       NodeId                                : 1880ec88a3187766a6da323399721f53
                       NodeInstanceId                        : 130729063464981219
                       NodeName                              : Node.1
                       }
```

> **NOTE**
> To ensure that unauthorized services can't report health against the entities in the cluster, the server can be configured to accept requests only from secured clients. The `FabricClient` used for reporting must have security enabled to be able to communicate with the cluster (for example, with Kerberos or certificate authentication). Read more about cluster security.

Report from within low privilege services

From within Service Fabric services that do not have admin access to the cluster, you can report health on entities from the current context through `Partition` or `CodePackageActivationContext`.

- For stateless services, use IStatelessServicePartition.ReportInstanceHealth to report on the current service instance.
- For stateful services, use IStatefulServicePartition.ReportReplicaHealth to report on current replica.
- Use IServicePartition.ReportPartitionHealth to report on the current partition entity.
- Use CodePackageActivationContext.ReportApplicationHealth to report on current application.
- Use CodePackageActivationContext.ReportDeployedApplicationHealth to report on the current application deployed on the current node.
- Use CodePackageActivationContext.ReportDeployedServicePackageHealth to report on a service package for the current application deployed on the current node.

> **NOTE**
> Internally, the `Partition` and the `CodePackageActivationContext` hold a health client which is configured with default settings. The same considerations explained for health client apply - reports are batched and sent on a timer, so the objects should be kept alive to have a chance to send the report.

Design health reporting

The first step in generating high-quality reports is identifying the conditions that can impact the health of the service. Any condition that can help flag problems in the service or cluster when it starts--or even better, before a

problem happens--can potentially save billions of dollars. The benefits include less down time, fewer night hours spent investigating and repairing issues, and higher customer satisfaction.

Once the conditions are identified, watchdog writers need to figure out the best way to monitor them for balance between overhead and usefulness. For example, consider a service that does complex calculations that use some temporary files on a share. A watchdog could monitor the share to ensure that enough space is available. It could listen for notifications of file or directory changes. It could report a warning if an upfront threshold is reached and report an error if the share is full. On a warning, a repair system could start cleaning up older files on the share. On an error, a repair system could move the service replica to another node. Note how the condition states are described in terms of health: the state of the condition that can be considered healthy or unhealthy (warning or error).

Once the monitoring details are set, a watchdog writer needs to figure out how to implement the watchdog. If the conditions can be determined from within the service, the watchdog can be part of the monitored service itself. For example, the service code can check the share usage, and then report by using a local fabric client every time it tries to write a file. The advantage of this approach is that reporting is simple. Care must be taken to prevent watchdog bugs from impacting the service functionality, though.

Reporting from within the monitored service is not always an option. A watchdog within the service may not be able to detect the conditions. It may not have the logic or data to make the determination. The overhead of monitoring the conditions may be high. The conditions also may not be specific to a service, but instead affect interactions between services. Another option is to have watchdogs in the cluster as separate processes. The watchdogs simply monitor the conditions and report, without affecting the main services in any way. For example, these watchdogs could be implemented as stateless services in the same application, deployed on all nodes or on the same nodes as the service.

Sometimes, a watchdog running in the cluster is not an option either. If the monitored condition is the availability or functionality of the service as users see it, it's best to have the watchdogs in the same place as the user clients. There, they can test the operations in the same way users call them. For example, you can have a watchdog that lives outside the cluster and issues requests to the service, and then checks the latency and correctness of the result. (For a calculator service, for example, does 2+2 return 4 in a reasonable amount of time?)

Once the watchdog details have been finalized, you should decide on a source ID that uniquely identifies it. If multiple watchdogs of the same type are living in the cluster, either they must report on different entities, or, if they report on the same entity, they must ensure that the source ID or the property is different. This way, their reports can coexist. The property of the health report should capture the monitored condition. (For the example above, the property could be **ShareSize**.) If multiple reports apply to the same condition, the property should contain some dynamic information that allows reports to coexist. For example, if multiple shares need to be monitored, the property name can be **ShareSize-sharename**.

> **NOTE**
>
> The health store should *not* be used to keep status information. Only health-related information should be reported as health, as this information impacts the health evaluation of an entity. The health store was not designed as a general-purpose store. It uses health evaluation logic to aggregate all data into the health state. Sending information unrelated to health (like reporting status with a health state of OK) will not impact the aggregated health state, but it can negatively affect the performance of the health store.

The next decision point is which entity to report on. Most of the time, this is obvious, based on the condition. You should choose the entity with best possible granularity. If a condition impacts all replicas in a partition, report on the partition, not on the service. There are corner cases where more thought is needed, though. If the condition impacts an entity, such as a replica, but the desire is to have the condition flagged for more than the duration of replica life, then it should be reported on the partition. Otherwise, when the replica is deleted, all reports associated with it will be cleaned up from the store. This means that watchdog writers must also think about the lifetimes of

the entity and the report. It must be clear when a report should be cleaned up from a store (for example, when an error reported on an entity no longer applies).

Let's look at an example that puts together the points I described. Consider a Service Fabric application composed of a master stateful persistent service and secondary stateless services deployed on all nodes (one secondary service type for each type of task). The master has a processing queue that contains commands to be executed by secondaries. The secondaries execute the incoming requests and send back acknowledgement signals. One condition that could be monitored is the length of the master processing queue. If the master queue length reaches a threshold, a warning is reported. The warning indicates that the secondaries can't handle the load. If the queue reaches the maximum length and commands are dropped, an error is reported, as the service can't recover. The reports can be on the property **QueueStatus**. The watchdog lives inside the service, and it's sent periodically on the master primary replica. The time to live is two minutes, and it's sent periodically every 30 seconds. If the primary goes down, the report is cleaned up automatically from store. If the service replica is up, but it is deadlocked or having other issues, the report expires in the health store. In this case, the entity is evaluated at error.

Another condition that can be monitored is task execution time. The master distributes tasks to the secondaries based on the task type. Depending on the design, the master could poll the secondaries for task status. It could also wait for secondaries to send back acknowledgement signals when they are done. In the second case, care must be taken to detect situations where secondaries die or messages are lost. One option is for the master to send a ping request to the same secondary, which sends back its status. If no status is received, the master considers it a failure and reschedules the task. This behavior assumes that the tasks are idempotent.

The monitored condition can be translated as a warning if the task is not done in a certain time (**t1**, for example 10 minutes). If the task is not completed in time (**t2**, for example 20 minutes), the monitored condition can be translated as Error. This reporting can be done in multiple ways:

- The master primary replica reports on itself periodically. You can have one property for all pending tasks in the queue. If at least one task takes longer, the report status on the property **PendingTasks** is a warning or error, as appropriate. If there are no pending tasks or all tasks started execution, the report status is OK. The tasks are persistent. If the primary goes down, the newly promoted primary can continue to report properly.
- Another watchdog process (in the cloud or external) checks the tasks (from outside, based on the desired task result) to see if they are completed. If they do not respect the thresholds, a report is sent on the master service. A report is also sent on each task that includes the task identifier, like **PendingTask+taskId**. Reports should be sent only on unhealthy states. Set time to live to a few minutes, and mark the reports to be removed when they expire to ensure cleanup.
- The secondary that is executing a task reports when it takes longer than expected to run it. It reports on the service instance on the property **PendingTasks**. The report pinpoints the service instance that has issues, but it doesn't capture the situation where the instance dies. The reports are cleaned up then. It could report on the secondary service. If the secondary completes the task, the secondary instance clears the report from the store. The report doesn't capture the situation where the acknowledgement message is lost and the task is not finished from the master's point of view.

However the reporting is done in the cases described above, the reports are captured in application health when health is evaluated.

Report periodically vs. on transition

By using the health reporting model, watchdogs can send reports periodically or on transitions. The recommended way for watchdog reporting is periodically, because the code is much simpler and less prone to errors. The watchdogs must strive to be as simple as possible to avoid bugs that trigger incorrect reports. Incorrect *unhealthy* reports impact health evaluations and scenarios based on health, including upgrades. Incorrect *healthy* reports hide issues in the cluster, which is not desired.

For periodic reporting, the watchdog can be implemented with a timer. On a timer callback, the watchdog can check the state and send a report based on the current state. There is no need to see which report was sent

previously or make any optimizations in terms of messaging. The health client has batching logic to help with performance. While the health client is kept alive, it retries internally until the report is acknowledged by the health store or the watchdog generates a newer report with the same entity, property, and source.

Reporting on transitions requires careful handling of state. The watchdog monitors some conditions and reports only when the conditions change. The upside of this approach is that fewer reports are needed. The downside is that the logic of the watchdog is complex. The watchdog must maintain the conditions or the reports, so that they can be inspected to determine state changes. On failover, care must be taken when a report is sent that may have not been sent previously (queued, but not yet sent to the health store). The sequence number must be ever-increasing. If not, the reports are rejected as stale. In the rare cases where data loss is incurred, synchronization may be needed between the state of the reporter and the state of the health store.

Reporting on transitions makes sense for services reporting on themselves, through `Partition` or `CodePackageActivationContext`. When the local object (replica or deployed service package / deployed application) is removed, all its reports are also removed. This automatic cleanup relaxes the need for synchronization between reporter and health store. If the report is for parent partition or parent application, care must be taken on failover to avoid stale reports in the health store. Logic must be added to maintain the correct state and clear the report from store when not needed anymore.

Implement health reporting

Once the entity and report details are clear, sending health reports can be done through the API, PowerShell, or REST.

API

To report through the API, you need to create a health report specific to the entity type they want to report on. Give the report to a health client. Alternatively, create a health information and pass it to correct reporting methods on `Partition` or `CodePackageActivationContext` to report on current entities.

The following example shows periodic reporting from a watchdog within the cluster. The watchdog checks whether an external resource can be accessed from within a node. The resource is needed by a service manifest within the application. If the resource is unavailable, the other services within the application can still function properly. Therefore, the report is sent on the deployed service package entity every 30 seconds.

```
private static Uri ApplicationName = new Uri("fabric:/WordCount");
private static string ServiceManifestName = "WordCount.Service";
private static string NodeName = FabricRuntime.GetNodeContext().NodeName;
private static Timer ReportTimer = new Timer(new TimerCallback(SendReport), null, 30 * 1000, 30 * 1000);
private static FabricClient Client = new FabricClient(new FabricClientSettings() { HealthReportSendInterval =
TimeSpan.FromSeconds(0) });

public static void SendReport(object obj)
{
    // Test whether the resource can be accessed from the node
    HealthState healthState = this.TestConnectivityToExternalResource();

    // Send report on deployed service package, as the connectivity is needed by the specific service manifest
    // and can be different on different nodes
    var deployedServicePackageHealthReport = new DeployedServicePackageHealthReport(
        ApplicationName,
        ServiceManifestName,
        NodeName,
        new HealthInformation("ExternalSourceWatcher", "Connectivity", healthState));

    // TODO: handle exception. Code omitted for snippet brevity.
    // Possible exceptions: FabricException with error codes
    // FabricHealthStaleReport (non-retryable, the report is already queued on the health client),
    // FabricHealthMaxReportsReached (retryable; user should retry with exponential delay until the report is
accepted).
    Client.HealthManager.ReportHealth(deployedServicePackageHealthReport);
}
```

PowerShell

Send health reports with **Send-ServiceFabric*EntityType*HealthReport**.

The following example shows periodic reporting on CPU values on a node. The reports should be sent every 30 seconds, and they have a time to live of two minutes. If they expire, the reporter has issues, so the node is evaluated at error. When the CPU is above a threshold, the report has a health state of warning. When the CPU remains above a threshold for more than the configured time, it's reported as an error. Otherwise, the reporter sends a health state of OK.

```
PS C:\> Send-ServiceFabricNodeHealthReport -NodeName Node.1 -HealthState Warning -SourceId PowershellWatcher -
HealthProperty CPU -Description "CPU is above 80% threshold" -TimeToLiveSec 120

PS C:\> Get-ServiceFabricNodeHealth -NodeName Node.1
NodeName              : Node.1
AggregatedHealthState : Warning
UnhealthyEvaluations  :
                        Unhealthy event: SourceId='PowershellWatcher', Property='CPU', HealthState='Warning',
ConsiderWarningAsError=false.

HealthEvents          :
                        SourceId              : System.FM
                        Property              : State
                        HealthState           : Ok
                        SequenceNumber        : 5
                        SentAt                : 4/21/2015 8:01:17 AM
                        ReceivedAt            : 4/21/2015 8:02:12 AM
                        TTL                   : Infinite
                        Description           : Fabric node is up.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : ->Ok = 4/21/2015 8:02:12 AM

                        SourceId              : PowershellWatcher
                        Property              : CPU
                        HealthState           : Warning
                        SequenceNumber        : 130741236814913394
                        SentAt                : 4/21/2015 9:01:21 PM
                        ReceivedAt            : 4/21/2015 9:01:21 PM
                        TTL                   : 00:02:00
                        Description           : CPU is above 80% threshold
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : ->Warning = 4/21/2015 9:01:21 PM
```

The following example reports a transient warning on a replica. It first gets the partition ID and then the replica ID for the service it is interested in. It then sends a report from **PowershellWatcher** on the property **ResourceDependency**. The report is of interest for only two minutes, and it is removed from the store automatically.

```
PS C:\> $partitionId = (Get-ServiceFabricPartition -ServiceName fabric:/WordCount/WordCount.Service).PartitionId

PS C:\> $replicaId = (Get-ServiceFabricReplica -PartitionId $partitionId | where {$_.ReplicaRole -eq
"Primary"}).ReplicaId

PS C:\> Send-ServiceFabricReplicaHealthReport -PartitionId $partitionId -ReplicaId $replicaId -HealthState
Warning -SourceId PowershellWatcher -HealthProperty ResourceDependency -Description "The external resource that
the primary is using has been rebooted at 4/21/2015 9:01:21 PM. Expect processing delays for a few minutes." -
TimeToLiveSec 120 -RemoveWhenExpired

PS C:\> Get-ServiceFabricReplicaHealth  -PartitionId $partitionId -ReplicaOrInstanceId $replicaId


PartitionId           : 8f82daff-eb68-4fd9-b631-7a37629e08c0
ReplicaId             : 130740415594605869
AggregatedHealthState : Warning
UnhealthyEvaluations  :
                        Unhealthy event: SourceId='PowershellWatcher', Property='ResourceDependency',
HealthState='Warning', ConsiderWarningAsError=false.

HealthEvents          :
                        SourceId              : System.RA
                        Property              : State
                        HealthState           : Ok
                        SequenceNumber        : 130740768777734943
                        SentAt                : 4/21/2015 8:01:17 AM
                        ReceivedAt            : 4/21/2015 8:02:12 AM
                        TTL                   : Infinite
                        Description           : Replica has been created.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : ->Ok = 4/21/2015 8:02:12 AM

                        SourceId              : PowershellWatcher
                        Property              : ResourceDependency
                        HealthState           : Warning
                        SequenceNumber        : 130741243777723555
                        SentAt                : 4/21/2015 9:12:57 PM
                        ReceivedAt            : 4/21/2015 9:12:57 PM
                        TTL                   : 00:02:00
                        Description           : The external resource that the primary is using has been
rebooted at 4/21/2015 9:01:21 PM. Expect processing delays for a few minutes.
                        RemoveWhenExpired     : True
                        IsExpired             : False
                        Transitions           : ->Warning = 4/21/2015 9:12:32 PM
```

REST

Send health reports using REST with POST requests that go to the desired entity and have in the body the health report description. For example, see how to send REST cluster health reports or service health reports. All entities are supported.

Next steps

Based on the health data, service writers and cluster/application administrators can think of ways to consume the information. For example, they can set up alerts based on health status to catch severe issues before they provoke outages. Administrators can also set up repair systems to fix issues automatically.

Introduction to Service Fabric health Monitoring

View Service Fabric health reports

How to report and check service health

Use system health reports for troubleshooting

Monitor and diagnose services locally

Service Fabric application upgrade

Use system health reports to troubleshoot

11/15/2016 • 17 min to read • Edit on GitHub

Contributors

Oana Platon • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • v-anacga • MarkDeichmiller

Azure Service Fabric components report out of the box on all entities in the cluster. The health store creates and deletes entities based on the system reports. It also organizes them in a hierarchy that captures entity interactions.

> **NOTE**
>
> To understand health-related concepts, read more at Service Fabric health model.

System health reports provide visibility into cluster and application functionality and flag issues through health. For applications and services, system health reports verify that entities are implemented and are behaving correctly from the Service Fabric perspective. The reports do not provide any health monitoring of the business logic of the service or detection of hung processes. User services can enrich the health data with information specific to their logic.

> **NOTE**
>
> Watchdogs health reports are visible only *after* the system components create an entity. When an entity is deleted, the health store automatically deletes all health reports associated with it. The same is true when a new instance of the entity is created (for example, a new service replica instance is created). All reports associated with the old instance are deleted and cleaned up from the store.

The system component reports are identified by the source, which starts with the "**System.**" prefix. Watchdogs can't use the same prefix for their sources, as reports with invalid parameters are rejected. Let's look at some system reports to understand what triggers them and how to correct the possible issues they represent.

> **NOTE**
>
> Service Fabric continues to add reports on conditions of interest that improve visibility into what is happening in the cluster and application.

Cluster system health reports

The cluster health entity is created automatically in the health store. If everything works properly, it doesn't have a system report.

Neighborhood loss

System.Federation reports an error when it detects a neighborhood loss. The report is from individual nodes, and the node ID is included in the property name. If one neighborhood is lost in the entire Service Fabric ring, you can typically expect two events (both sides of the gap report). If more neighborhoods are lost, there are more events.

The report specifies the global lease timeout as the time to live. The report is resent every half of the TTL duration for as long as the condition remains active. The event is automatically removed when it expires. Remove when

expired behavior ensures that the report is cleaned up from the health store correctly, even if the reporting node is down.

- **SourceId**: System.Federation
- **Property**: Starts with **Neighborhood** and includes node information
- **Next steps**: Investigate why the neighborhood is lost (for example, check the communication between cluster nodes).

Node system health reports

System.FM, which represents the Failover Manager service, is the authority that manages information about cluster nodes. Each node should have one report from System.FM showing its state. The node entities are removed when the node state is removed (see RemoveNodeStateAsync).

Node up/down

System.FM reports as OK when the node joins the ring (it's up and running). It reports an error when the node departs the ring (it's down, either for upgrading or simply because it has failed). The health hierarchy built by the health store takes action on deployed entities in correlation with System.FM node reports. It considers the node a virtual parent of all deployed entities. The deployed entities on that node are exposed through queries if the node is reported as up by System.FM, with the same instance as the instance associated with the entities. When System.FM reports that the node is down or restarted (a new instance), the health store automatically cleans up the deployed entities that can exist only on the down node or on the previous instance of the node.

- **SourceId**: System.FM
- **Property**: State
- **Next steps**: If the node is down for an upgrade, it should come back up once it has been upgraded. In this case, the health state should be switched back to OK. If the node doesn't come back or it fails, the problem needs more investigation.

The following example shows the System.FM event with a health state of OK for node up:

```
PS C:\> Get-ServiceFabricNodeHealth -NodeName Node.1
NodeName              : Node.1
AggregatedHealthState : Ok
HealthEvents          :
                        SourceId              : System.FM
                        Property              : State
                        HealthState           : Ok
                        SequenceNumber        : 2
                        SentAt                : 4/24/2015 5:27:33 PM
                        ReceivedAt            : 4/24/2015 5:28:50 PM
                        TTL                   : Infinite
                        Description           : Fabric node is up.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : ->Ok = 4/24/2015 5:28:50 PM
```

Certificate expiration

System.FabricNode reports a warning when certificates used by the node are near expiration. There are three certificates per node: **Certificate_cluster**, **Certificate_server**, and **Certificate_default_client**. When the expiration is at least two weeks away, the report health state is OK. When the expiration is within two weeks, the report type is a warning. TTL of these events is infinite, and they are removed when a node leaves the cluster.

- **SourceId**: System.FabricNode
- **Property**: Starts with **Certificate** and contains more information about the certificate type
- **Next steps**: Update the certificates if they are near expiration.

Load capacity violation

The Service Fabric Load Balancer reports a warning if it detects a node capacity violation.

- **SourceId**: System.PLB
- **Property**: Starts with **Capacity**
- **Next steps**: Check provided metrics and view the current capacity on the node.

Application system health reports

System.CM, which represents the Cluster Manager service, is the authority that manages information about an application.

State

System.CM reports as OK when the application has been created or updated. It informs the health store when the application has been deleted, so that it can be removed from store.

- **SourceId**: System.CM
- **Property**: State
- **Next steps**: If the application has been created, it should include the Cluster Manager health report. Otherwise, check the state of the application by issuing a query (for example, the PowerShell cmdlet **Get-ServiceFabricApplication -ApplicationName *applicationName***).

The following example shows the state event on the **fabric:/WordCount** application:

```
PS C:\> Get-ServiceFabricApplicationHealth fabric:/WordCount -ServicesFilter None -DeployedApplicationsFilter
None

ApplicationName                 : fabric:/WordCount
AggregatedHealthState           : Ok
ServiceHealthStates             : None
DeployedApplicationHealthStates : None
HealthEvents                    :
                                  SourceId              : System.CM
                                  Property              : State
                                  HealthState           : Ok
                                  SequenceNumber        : 82
                                  SentAt                : 4/24/2015 6:12:51 PM
                                  ReceivedAt            : 4/24/2015 6:12:51 PM
                                  TTL                   : Infinite
                                  Description           : Application has been created.
                                  RemoveWhenExpired     : False
                                  IsExpired             : False
                                  Transitions           : ->Ok = 4/24/2015 6:12:51 PM
```

Service system health reports

System.FM, which represents the Failover Manager service, is the authority that manages information about services.

State

System.FM reports as OK when the service has been created. It deletes the entity from the health store when the service has been deleted.

- **SourceId**: System.FM
- **Property**: State

The following example shows the state event on the service **fabric:/WordCount/WordCountService**:

```
PS C:\> Get-ServiceFabricServiceHealth fabric:/WordCount/WordCountService

ServiceName           : fabric:/WordCount/WordCountService
AggregatedHealthState : Ok
PartitionHealthStates :
                        PartitionId           : 875a1caa-d79f-43bd-ac9d-43ee89a9891c
                        AggregatedHealthState : Ok

HealthEvents          :
                        SourceId              : System.FM
                        Property              : State
                        HealthState           : Ok
                        SequenceNumber        : 3
                        SentAt                : 4/24/2015 6:12:51 PM
                        ReceivedAt            : 4/24/2015 6:13:01 PM
                        TTL                   : Infinite
                        Description           : Service has been created.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : ->Ok = 4/24/2015 6:13:01 PM
```

Unplaced replicas violation

System.PLB reports a warning if it cannot find a placement for one or more service replicas. The report is removed when it expires.

- **SourceId**: System.FM
- **Property**: State
- **Next steps**: Check the service constraints and the current state of the placement.

The following example shows a violation for a service configured with 7 target replicas in a cluster with 5 nodes:

```
PS C:\> Get-ServiceFabricServiceHealth fabric:/WordCount/WordCountService


ServiceName           : fabric:/WordCount/WordCountService
AggregatedHealthState : Warning
UnhealthyEvaluations  :
                        Unhealthy event: SourceId='System.PLB',
                        Property='ServiceReplicaUnplacedHealth_Secondary_a1f83a35-d6bf-4d39-b90d-28d15f39599b',
HealthState='Warning',
                        ConsiderWarningAsError=false.

PartitionHealthStates :
                        PartitionId           : a1f83a35-d6bf-4d39-b90d-28d15f39599b
                        AggregatedHealthState : Warning

HealthEvents          :
                        SourceId              : System.FM
                        Property              : State
                        HealthState           : Ok
                        SequenceNumber        : 10
                        SentAt                : 3/22/2016 7:56:53 PM
                        ReceivedAt            : 3/22/2016 7:57:18 PM
                        TTL                   : Infinite
                        Description           : Service has been created.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : Error->Ok = 3/22/2016 7:57:18 PM, LastWarning = 1/1/0001
12:00:00 AM

                        SourceId              : System.PLB
                        Property              : ServiceReplicaUnplacedHealth_Secondary_a1f83a35-d6bf-4d39-b90d-
28d15f39599b
                        HealthState           : Warning
                        SequenceNumber        : 131032232425505477
```

```
                        SentAt                : 3/23/2016 4:14:02 PM
                        ReceivedAt            : 3/23/2016 4:14:03 PM
                        TTL                   : 00:01:05
                        Description           : The Load Balancer was unable to find a placement for one or
more of the Service's Replicas:
                        fabric:/WordCount/WordCountService Secondary Partition a1f83a35-d6bf-4d39-b90d-
28d15f39599b could not be placed, possibly,
                        due to the following constraints and properties:
                        Placement Constraint: N/A
                        Depended Service: N/A

                        Constraint Elimination Sequence:
                        ReplicaExclusionStatic eliminated 4 possible node(s) for placement -- 1/5 node(s)
remain.
                        ReplicaExclusionDynamic eliminated 1 possible node(s) for placement -- 0/5 node(s)
remain.

                        Nodes Eliminated By Constraints:

                        ReplicaExclusionStatic:
                        FaultDomain:fd:/0 NodeName:_Node_0 NodeType:NodeType0 UpgradeDomain:0 UpgradeDomain:
ud:/0 Deactivation Intent/Status:
                        None/None
                        FaultDomain:fd:/1 NodeName:_Node_1 NodeType:NodeType1 UpgradeDomain:1 UpgradeDomain:
ud:/1 Deactivation Intent/Status:
                        None/None
                        FaultDomain:fd:/3 NodeName:_Node_3 NodeType:NodeType3 UpgradeDomain:3 UpgradeDomain:
ud:/3 Deactivation Intent/Status:
                        None/None
                        FaultDomain:fd:/4 NodeName:_Node_4 NodeType:NodeType4 UpgradeDomain:4 UpgradeDomain:
ud:/4 Deactivation Intent/Status:
                        None/None

                        ReplicaExclusionDynamic:
                        FaultDomain:fd:/2 NodeName:_Node_2 NodeType:NodeType2 UpgradeDomain:2 UpgradeDomain:
ud:/2 Deactivation Intent/Status:
                        None/None


                        RemoveWhenExpired     : True
                        IsExpired             : False
                        Transitions           : Error->Warning = 3/22/2016 7:57:48 PM, LastOk = 1/1/0001
12:00:00 AM
```

Partition system health reports

System.FM, which represents the Failover Manager service, is the authority that manages information about service partitions.

State

System.FM reports as OK when the partition has been created and is healthy. It deletes the entity from the health store when the partition is deleted.

If the partition is below the minimum replica count, it reports an error. If the partition is not below the minimum replica count, but it is below the target replica count, it reports a warning. If the partition is in quorum loss, System.FM reports an error.

Other important events include a warning when the reconfiguration takes longer than expected and when the build takes longer than expected. The expected times for the build and reconfiguration are configurable based on service scenarios. For example, if a service has a terabyte of state, such as SQL Database, the build takes longer than for a service with a small amount of state.

- **SourceId**: System.FM

- **Property**: State
- **Next steps**: If the health state is not OK, it's possible that some replicas have not been created, opened, or promoted to primary or secondary correctly. In many instances, the root cause is a service bug in the open or change-role implementation.

The following example shows a healthy partition:

```
PS C:\> Get-ServiceFabricPartition fabric:/StatelessPiApplication/StatelessPiService | Get-
ServiceFabricPartitionHealth
PartitionId           : 29da484c-2c08-40c5-b5d9-03774af9a9bf
AggregatedHealthState : Ok
ReplicaHealthStates   : None
HealthEvents          :
                        SourceId              : System.FM
                        Property              : State
                        HealthState           : Ok
                        SequenceNumber        : 38
                        SentAt                : 4/24/2015 6:33:10 PM
                        ReceivedAt            : 4/24/2015 6:33:31 PM
                        TTL                   : Infinite
                        Description           : Partition is healthy.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : ->Ok = 4/24/2015 6:33:31 PM
```

The following example shows the health of a partition that is below target replica count. The next step is to get the partition description, which shows how it is configured: **MinReplicaSetSize** is two and **TargetReplicaSetSize** is seven. Then get the number of nodes in the cluster: five. So in this case, two replicas can't be placed.

```
PS C:\> Get-ServiceFabricPartition fabric:/WordCount/WordCountService | Get-ServiceFabricPartitionHealth -
ReplicasFilter None

PartitionId           : 875a1caa-d79f-43bd-ac9d-43ee89a9891c
AggregatedHealthState : Warning
UnhealthyEvaluations  :
                        Unhealthy event: SourceId='System.FM', Property='State', HealthState='Warning',
ConsiderWarningAsError=false.

ReplicaHealthStates   : None
HealthEvents          :
                        SourceId              : System.FM
                        Property              : State
                        HealthState           : Warning
                        SequenceNumber        : 37
                        SentAt                : 4/24/2015 6:13:12 PM
                        ReceivedAt            : 4/24/2015 6:13:31 PM
                        TTL                   : Infinite
                        Description           : Partition is below target replica or instance count.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : Ok->Warning = 4/24/2015 6:13:31 PM

PS C:\> Get-ServiceFabricPartition fabric:/WordCount/WordCountService

PartitionId            : 875a1caa-d79f-43bd-ac9d-43ee89a9891c
PartitionKind          : Int64Range
PartitionLowKey        : 1
PartitionHighKey       : 26
PartitionStatus        : Ready
LastQuorumLossDuration : 00:00:00
MinReplicaSetSize      : 2
TargetReplicaSetSize   : 7
HealthState            : Warning
DataLossNumber         : 130743727710830900
ConfigurationNumber    : 8589934592


PS C:\> @(Get-ServiceFabricNode).Count
5
```

Replica constraint violation

System.PLB reports a warning if it detects a replica constraint violation and can't place replicas of the partition.

- **SourceId**: System.PLB
- **Property**: Starts with **ReplicaConstraintViolation**

Replica system health reports

System.RA, which represents the reconfiguration agent component, is the authority for the replica state.

State

System.RA reports as OK when the replica has been created.

- **SourceId**: System.RA
- **Property**: State

The following example shows a healthy replica:

```
PS C:\> Get-ServiceFabricPartition fabric:/WordCount/WordCountService | Get-ServiceFabricReplica | where
{$_.ReplicaRole -eq "Primary"} | Get-ServiceFabricReplicaHealth
PartitionId           : 875a1caa-d79f-43bd-ac9d-43ee89a9891c
ReplicaId             : 130743727717237310
AggregatedHealthState : Ok
HealthEvents          :
                        SourceId              : System.RA
                        Property              : State
                        HealthState           : Ok
                        SequenceNumber        : 130743727718018580
                        SentAt                : 4/24/2015 6:12:51 PM
                        ReceivedAt            : 4/24/2015 6:13:02 PM
                        TTL                   : Infinite
                        Description           : Replica has been created.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : ->Ok = 4/24/2015 6:13:02 PM
```

Replica open status

The description of this health report contains the start time (Coordinated Universal Time) when the API call was invoked.

System.RA reports a warning if the replica open takes longer than the configured period (default: 30 minutes). If the API impacts service availability, the report is issued much faster (a configurable interval, with a default of 30 seconds). The time measured includes the time taken for the replicator open and the service open. The property changes to OK if the open completes.

- **SourceId**: System.RA
- **Property**: **ReplicaOpenStatus**
- **Next steps**: If the health state is not OK, investigate why the replica open takes longer than expected.

Slow service API call

System.RAP and **System.Replicator** report a warning if a call to the user service code takes longer than the configured time. The warning is cleared when the call completes.

- **SourceId**: System.RAP or System.Replicator
- **Property**: The name of the slow API. The description provides more details about the time the API has been pending.
- **Next steps**: Investigate why the call takes longer than expected.

The following example shows a partition in quorum loss, and the investigation steps done to figure out why. One of the replicas has a warning health state, so you get its health. It shows that the service operation takes longer than expected, an event reported by System.RAP. After this information is received, the next step is to look at the service code and investigate there. For this case, the **RunAsync** implementation of the stateful service throws an unhandled exception. The replicas are recycling, so you may not see any replicas in the warning state. You can retry getting the health state and look for any differences in the replica ID. In certain cases, the retries can give you clues.

```
PS C:\> Get-ServiceFabricPartition fabric:/HelloWorldStatefulApplication/HelloWorldStateful | Get-
ServiceFabricPartitionHealth

PartitionId           : 72a0fb3e-53ec-44f2-9983-2f272aca3e38
AggregatedHealthState : Error
UnhealthyEvaluations  :
                        Error event: SourceId='System.FM', Property='State'.

ReplicaHealthStates   :
                        ReplicaId             : 130743748372546446
                        AggregatedHealthState : Ok
```

```
                        ReplicaId             : 130743746168084332
                        AggregatedHealthState : Ok

                        ReplicaId             : 130743746195428808
                        AggregatedHealthState : Warning

                        ReplicaId             : 130743746195428807
                        AggregatedHealthState : Ok

HealthEvents          :
                        SourceId              : System.FM
                        Property              : State
                        HealthState           : Error
                        SequenceNumber        : 182
                        SentAt                : 4/24/2015 7:00:17 PM
                        ReceivedAt            : 4/24/2015 7:00:31 PM
                        TTL                   : Infinite
                        Description           : Partition is in quorum loss.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : Warning->Error = 4/24/2015 6:51:31 PM

PS C:\> Get-ServiceFabricPartition fabric:/HelloWorldStatefulApplication/HelloWorldStateful

PartitionId            : 72a0fb3e-53ec-44f2-9983-2f272aca3e38
PartitionKind          : Int64Range
PartitionLowKey        : -9223372036854775808
PartitionHighKey       : 9223372036854775807
PartitionStatus        : InQuorumLoss
LastQuorumLossDuration : 00:00:13
MinReplicaSetSize      : 2
TargetReplicaSetSize   : 3
HealthState            : Error
DataLossNumber         : 130743746152927699
ConfigurationNumber    : 227633266688

PS C:\> Get-ServiceFabricReplica 72a0fb3e-53ec-44f2-9983-2f272aca3e38 130743746195428808

ReplicaId           : 130743746195428808
ReplicaAddress      : PartitionId: 72a0fb3e-53ec-44f2-9983-2f272aca3e38, ReplicaId: 130743746195428808
ReplicaRole         : Primary
NodeName            : Node.3
ReplicaStatus       : Ready
LastInBuildDuration : 00:00:01
HealthState         : Warning

PS C:\> Get-ServiceFabricReplicaHealth 72a0fb3e-53ec-44f2-9983-2f272aca3e38 130743746195428808

PartitionId           : 72a0fb3e-53ec-44f2-9983-2f272aca3e38
ReplicaId             : 130743746195428808
AggregatedHealthState : Warning
UnhealthyEvaluations  :
                        Unhealthy event: SourceId='System.RAP', Property='ServiceOpenOperationDuration',
HealthState='Warning', ConsiderWarningAsError=false.

HealthEvents          :
                        SourceId              : System.RA
                        Property              : State
                        HealthState           : Ok
                        SequenceNumber        : 130743756170185892
                        SentAt                : 4/24/2015 7:00:17 PM
                        ReceivedAt            : 4/24/2015 7:00:33 PM
                        TTL                   : Infinite
                        Description           : Replica has been created.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : ->Ok = 4/24/2015 7:00:33 PM
```

```
SourceId              : System.RAP
Property              : ServiceOpenOperationDuration
HealthState           : Warning
SequenceNumber        : 130743756399407044
SentAt                : 4/24/2015 7:00:39 PM
ReceivedAt            : 4/24/2015 7:00:59 PM
TTL                   : Infinite
Description           : Start Time (UTC): 2015-04-24 19:00:17.019
RemoveWhenExpired     : False
IsExpired             : False
Transitions           : ->Warning = 4/24/2015 7:00:59 PM
```

When you start the faulty application under the debugger, the diagnostic events windows show the exception thrown from RunAsync:

Visual Studio 2015 diagnostic events: RunAsync failure in **fabric:/HelloWorldStatefulApplication**.

Replication queue full

System.Replicator reports a warning if the replication queue is full. On the primary, this usually happens because one or more secondary replicas are slow to acknowledge operations. On the secondary, this usually happens when the service is slow to apply the operations. The warning is cleared when the queue is no longer full.

- **SourceId**: System.Replicator
- **Property**: **PrimaryReplicationQueueStatus** or **SecondaryReplicationQueueStatus**, depending on the replica role

Slow Naming operations

System.NamingService reports health on its primary replica when a Naming operation takes longer than acceptable. Examples of Naming operations are CreateServiceAsync or DeleteServiceAsync. More methods can be found under FabricClient, for example under service management methods or property management methods.

> **NOTE**
> The Naming service resolves service names to a location in the cluster and enables users to manage service names and properties. It is a Service Fabric partitioned persisted service. One of the partitions represents the Authority Owner, which contains metadata about all Service Fabric names and services. The Service Fabric names are mapped to different partitions, called Name Owner partitions, so the service is extensible. Read more about Naming service.

When a Naming operation takes longer than expected, the operation is flagged with a Warning report on the *primary replica of the Naming service partition that serves the operation*. If the operation completes successfully, the Warning is cleared. If the operation completes with an error, the health report includes details about the error.

- **SourceId**: System.NamingService
- **Property**: Starts with prefix **Duration_** and identifies the slow operation and the Service Fabric name on which the operation is applied. For example, if create service at name fabric:/MyApp/MyService takes too long, the property is Duration_AOCreateService.fabric:/MyApp/MyService. AO points to the role of the Naming partition for this name and operation.
- **Next steps**: Check why the Naming operation fails. Each operation can have different root causes. For example, delete service may be stuck on a node because the application host keeps crashing on a node due to a user bug in the service code.

The following example shows a create service operation. The operation took longer than the configured duration. AO retries and sends work to NO. NO completed the last operation with Timeout. In this case, the same replica is primary for both the AO and NO roles.

```
PartitionId           : 00000000-0000-0000-0000-000000001000
ReplicaId             : 131064359253133577
AggregatedHealthState : Warning
UnhealthyEvaluations  :
                        Unhealthy event: SourceId='System.NamingService',
Property='Duration_AOCreateService.fabric:/MyApp/MyService', HealthState='Warning',
ConsiderWarningAsError=false.

HealthEvents          :
                        SourceId              : System.RA
                        Property              : State
                        HealthState           : Ok
                        SequenceNumber        : 131064359308715535
                        SentAt                : 4/29/2016 8:38:50 PM
                        ReceivedAt            : 4/29/2016 8:39:08 PM
                        TTL                   : Infinite
                        Description           : Replica has been created.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : Error->Ok = 4/29/2016 8:39:08 PM, LastWarning = 1/1/0001
12:00:00 AM

                        SourceId              : System.NamingService
                        Property              : Duration_AOCreateService.fabric:/MyApp/MyService
                        HealthState           : Warning
                        SequenceNumber        : 131064359526778775
                        SentAt                : 4/29/2016 8:39:12 PM
                        ReceivedAt            : 4/29/2016 8:39:38 PM
                        TTL                   : 00:05:00
                        Description           : The AOCreateService started at 2016-04-29 20:39:08.677 is
taking longer than 30.000.
                        RemoveWhenExpired     : True
                        IsExpired             : False
                        Transitions           : Error->Warning = 4/29/2016 8:39:38 PM, LastOk = 1/1/0001
12:00:00 AM

                        SourceId              : System.NamingService
                        Property              : Duration_NOCreateService.fabric:/MyApp/MyService
                        HealthState           : Warning
                        SequenceNumber        : 131064360657607311
                        SentAt                : 4/29/2016 8:41:05 PM
                        ReceivedAt            : 4/29/2016 8:41:08 PM
                        TTL                   : 00:00:15
                        Description           : The NOCreateService started at 2016-04-29 20:39:08.689
completed with FABRIC_E_TIMEOUT in more than 30.000.
                        RemoveWhenExpired     : True
                        IsExpired             : False
                        Transitions           : Error->Warning = 4/29/2016 8:39:38 PM, LastOk = 1/1/0001
12:00:00 AM
```

DeployedApplication system health reports

System.Hosting is the authority on deployed entities.

Activation

System.Hosting reports as OK when an application has been successfully activated on the node. Otherwise, it reports an error.

- **SourceId**: System.Hosting
- **Property**: Activation, including the rollout version
- **Next steps**: If the application is unhealthy, investigate why the activation failed.

The following example shows successful activation:

```
PS C:\> Get-ServiceFabricDeployedApplicationHealth -NodeName Node.1 -ApplicationName fabric:/WordCount

ApplicationName                    : fabric:/WordCount
NodeName                           : Node.1
AggregatedHealthState              : Ok
DeployedServicePackageHealthStates :
                                     ServiceManifestName   : WordCountServicePkg
                                     NodeName              : Node.1
                                     AggregatedHealthState : Ok

HealthEvents                       :
                                     SourceId              : System.Hosting
                                     Property              : Activation
                                     HealthState           : Ok
                                     SequenceNumber        : 130743727751144415
                                     SentAt                : 4/24/2015 6:12:55 PM
                                     ReceivedAt            : 4/24/2015 6:13:03 PM
                                     TTL                   : Infinite
                                     Description           : The application was activated successfully.
                                     RemoveWhenExpired     : False
                                     IsExpired             : False
                                     Transitions           : ->Ok = 4/24/2015 6:13:03 PM
```

Download

System.Hosting reports an error if the application package download fails.

- **SourceId**: System.Hosting
- **Property**: **Download:*RolloutVersion***
- **Next steps**: Investigate why the download failed on the node.

DeployedServicePackage system health reports

System.Hosting is the authority on deployed entities.

Service package activation

System.Hosting reports as OK if the service package activation on the node is successful. Otherwise, it reports an error.

- **SourceId**: System.Hosting
- **Property**: Activation
- **Next steps**: Investigate why the activation failed.

Code package activation

System.Hosting reports as OK for each code package if the activation is successful. If the activation fails, it reports a warning as configured. If **CodePackage** fails to activate or terminates with an error greater than the configured **CodePackageHealthErrorThreshold**, hosting reports an error. If a service package contains multiple code packages, an activation report is generated for each one.

- **SourceId**: System.Hosting
- **Property**: Uses the prefix **CodePackageActivation** and contains the name of the code package and the entry point as **CodePackageActivation:*CodePackageName*:*SetupEntryPoint/EntryPoint*** (for example, **CodePackageActivation:Code:SetupEntryPoint**)

Service type registration

System.Hosting reports as OK if the service type has been registered successfully. It reports an error if the registration wasn't done in time (as configured by using **ServiceTypeRegistrationTimeout**). If the service type is unregistered from the node, this is because the run time has been closed. Hosting reports a warning.

- **SourceId**: System.Hosting
- **Property**: Uses the prefix **ServiceTypeRegistration** and contains the service type name (for example, **ServiceTypeRegistration:FileStoreServiceType**)

The following example shows a healthy deployed service package:

```
PS C:\> Get-ServiceFabricDeployedServicePackageHealth -NodeName Node.1 -ApplicationName fabric:/WordCount -
ServiceManifestName WordCountServicePkg


ApplicationName       : fabric:/WordCount
ServiceManifestName   : WordCountServicePkg
NodeName              : Node.1
AggregatedHealthState : Ok
HealthEvents          :
                        SourceId              : System.Hosting
                        Property              : Activation
                        HealthState           : Ok
                        SequenceNumber        : 130743727751456915
                        SentAt                : 4/24/2015 6:12:55 PM
                        ReceivedAt            : 4/24/2015 6:13:03 PM
                        TTL                   : Infinite
                        Description           : The ServicePackage was activated successfully.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : ->Ok = 4/24/2015 6:13:03 PM

                        SourceId              : System.Hosting
                        Property              : CodePackageActivation:Code:EntryPoint
                        HealthState           : Ok
                        SequenceNumber        : 130743727751613185
                        SentAt                : 4/24/2015 6:12:55 PM
                        ReceivedAt            : 4/24/2015 6:13:03 PM
                        TTL                   : Infinite
                        Description           : The CodePackage was activated successfully.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : ->Ok = 4/24/2015 6:13:03 PM

                        SourceId              : System.Hosting
                        Property              : ServiceTypeRegistration:WordCountServiceType
                        HealthState           : Ok
                        SequenceNumber        : 130743727753644473
                        SentAt                : 4/24/2015 6:12:55 PM
                        ReceivedAt            : 4/24/2015 6:13:03 PM
                        TTL                   : Infinite
                        Description           : The ServiceType was registered successfully.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : ->Ok = 4/24/2015 6:13:03 PM
```

Download

System.Hosting reports an error if the service package download fails.

- **SourceId**: System.Hosting
- **Property**: **Download:*RolloutVersion***
- **Next steps**: Investigate why the download failed on the node.

Upgrade validation

System.Hosting reports an error if validation during the upgrade fails or if the upgrade fails on the node.

- **SourceId**: System.Hosting
- **Property**: Uses the prefix **FabricUpgradeValidation** and contains the upgrade version

- **Description**: Points to the error encountered

Next steps

View Service Fabric health reports

How to report and check service health

Monitor and diagnose services locally

Service Fabric application upgrade

View Service Fabric health reports

11/15/2016 • 27 min to read • Edit on GitHub

Contributors

Oana Platon • Theano Petersen • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • rmcmurray • MarkDeichmiller

Azure Service Fabric introduces a health model that comprises health entities on which system components and watchdogs can report local conditions that they are monitoring. The health store aggregates all health data to determine whether entities are healthy.

Out of the box, the cluster is populated with health reports sent by the system components. Read more at Use system health reports to troubleshoot.

Service Fabric provides multiple ways to get the aggregated health of the entities:

- Service Fabric Explorer or other visualization tools
- Health queries (through PowerShell, the API, or REST)
- General queries that return a list of entities that have health as one of the properties (through PowerShell, the API, or REST)

To demonstrate these options, let's use a local cluster with five nodes. Next to the **fabric:/System** application (which exists out of the box), some other applications are deployed. One of these applications is **fabric:/WordCount**. This application contains a stateful service configured with seven replicas. Because there are only five nodes, the system components show a warning that the partition is below the target count.

```
<Service Name="WordCountService">
    <StatefulService ServiceTypeName="WordCountServiceType" TargetReplicaSetSize="7" MinReplicaSetSize="2">
      <UniformInt64Partition PartitionCount="1" LowKey="1" HighKey="26" />
    </StatefulService>
</Service>
```

Health in Service Fabric Explorer

Service Fabric Explorer provides a visual view of the cluster. In the image below, you can see that:

- The application **fabric:/WordCount** is red (in error) because it has an error event reported by **MyWatchdog** for the property **Availability**.
- One of its services, **fabric:/WordCount/WordCountService** is yellow (in warning). Since the service is configured with seven replicas, they can't all be placed, as there are only five nodes. Although it's not shown here, the service partition is yellow because of the system report. The yellow partition triggers the yellow service.
- The cluster is red because of the red application.

The evaluation uses default policies from the cluster manifest and application manifest. They are strict policies and do not tolerate any failure.

View of the cluster with Service Fabric Explorer:

> **NOTE**
>
> Read more about Service Fabric Explorer.

Health queries

Service Fabric exposes health queries for each of the supported entity types. They can be accessed through the API (the methods can be found on **FabricClient.HealthManager**), PowerShell cmdlets, and REST. These queries return complete health information about the entity: the aggregated health state, entity health events, child health states (when applicable), and unhealthy evaluations when the entity is not healthy.

> **NOTE**
>
> A health entity is returned when it is fully populated in the health store. The entity must be active (not deleted) and have a system report. Its parent entities on the hierarchy chain must also have system reports. If any of these conditions is not satisfied, the health queries return an exception that shows why the entity is not returned.

The health queries must pass in the entity identifier, which depends on the entity type. The queries accept optional health policy parameters. If no health policies are specified, the health policies from the cluster or application manifest are used for evaluation. The queries also accept filters for returning only partial children or events--the ones that respect the specified filters.

> **NOTE**
>
> The output filters are applied on the server side, so the message reply size is reduced. We recommended that you use the output filters to limit the data returned, rather than apply filters on the client side.

An entity's health contains:

- The aggregated health state of the entity. Computed by the health store based on entity health reports, child health states (when applicable), and health policies. Read more about entity health evaluation.
- The health events on the entity.
- The collection of health states of all children for the entities that can have children. The health states contain entity identifiers and the aggregated health state. To get complete health for a child, call the query health for the child entity type and pass in the child identifier.
- The unhealthy evaluations that point to the report that triggered the state of the entity, if the entity is not healthy.

Get cluster health

Returns the health of the cluster entity and contains the health states of applications and nodes (children of the cluster). Input:

- [Optional] The cluster health policy used to evaluate the nodes and the cluster events.
- [Optional] The application health policy map, with the health policies used to override the application manifest policies.
- [Optional] Filters for events, nodes, and applications that specify which entries are of interest and should be returned in the result (for example, errors only, or both warnings and errors). All events, nodes, and applications are used to evaluate the entity aggregated health, regardless of the filter.

API

To get cluster health, create a `FabricClient` and call the GetClusterHealthAsync method on its **HealthManager**.

The following call gets the cluster health:

```
ClusterHealth clusterHealth = await fabricClient.HealthManager.GetClusterHealthAsync();
```

The following code gets the cluster health by using a custom cluster health policy and filters for nodes and applications. It creates ClusterHealthQueryDescription, which contains the input information.

```
var policy = new ClusterHealthPolicy()
{
    MaxPercentUnhealthyNodes = 20
};
var nodesFilter = new NodeHealthStatesFilter()
{
    HealthStateFilterValue = HealthStateFilter.Error | HealthStateFilter.Warning
};
var applicationsFilter = new ApplicationHealthStatesFilter()
{
    HealthStateFilterValue = HealthStateFilter.Error
};
var queryDescription = new ClusterHealthQueryDescription()
{
    HealthPolicy = policy,
    ApplicationsFilter = applicationsFilter,
    NodesFilter = nodesFilter,
};

ClusterHealth clusterHealth = await fabricClient.HealthManager.GetClusterHealthAsync(queryDescription);
```

PowerShell

The cmdlet to get the cluster health is Get-ServiceFabricClusterHealth. First, connect to the cluster by using the Connect-ServiceFabricCluster cmdlet.

The state of the cluster is five nodes, the system application, and fabric:/WordCount configured as described.

The following cmdlet gets cluster health by using default health policies. The aggregated health state is in warning, because the fabric:/WordCount application is in warning. Note how the unhealthy evaluations provide details on the conditions that triggered the aggregated health.

```
PS C:\> Get-ServiceFabricClusterHealth

AggregatedHealthState   : Warning
UnhealthyEvaluations    :
                          Unhealthy applications: 100% (1/1), MaxPercentUnhealthyApplications=0%.

                          Unhealthy application: ApplicationName='fabric:/WordCount',
AggregatedHealthState='Warning'.

                              Unhealthy services: 100% (1/1), ServiceType='WordCountServiceType',
MaxPercentUnhealthyServices=0%.

                              Unhealthy service: ServiceName='fabric:/WordCount/WordCountService',
AggregatedHealthState='Warning'.

                                  Unhealthy event: SourceId='System.PLB',
                          Property='ServiceReplicaUnplacedHealth_Secondary_a1f83a35-d6bf-4d39-b90d-
28d15f39599b', HealthState='Warning',
                          ConsiderWarningAsError=false.


NodeHealthStates        :
                          NodeName              : _Node_2
                          AggregatedHealthState : Ok

                          NodeName              : _Node_0
                          AggregatedHealthState : Ok

                          NodeName              : _Node_1
                          AggregatedHealthState : Ok

                          NodeName              : _Node_3
                          AggregatedHealthState : Ok

                          NodeName              : _Node_4
                          AggregatedHealthState : Ok

ApplicationHealthStates :
                          ApplicationName       : fabric:/System
                          AggregatedHealthState : Ok

                          ApplicationName       : fabric:/WordCount
                          AggregatedHealthState : Warning

HealthEvents            : None
```

The following PowerShell cmdlet gets the health of the cluster by using a custom application policy. It filters results to get only error or warning applications and nodes. As a result, no nodes are returned, as they are all healthy. Only the fabric:/WordCount application respects the applications filter. Because the custom policy specifies to consider warnings as errors for the fabric:/WordCount application, the application is evaluated as in error, and so is the cluster.

```
PS c:\> $appHealthPolicy = New-Object -TypeName System.Fabric.Health.ApplicationHealthPolicy
$appHealthPolicy.ConsiderWarningAsError = $true
$appHealthPolicyMap = New-Object -TypeName System.Fabric.Health.ApplicationHealthPolicyMap
$appUri1 = New-Object -TypeName System.Uri -ArgumentList "fabric:/WordCount"
$appHealthPolicyMap.Add($appUri1, $appHealthPolicy)
Get-ServiceFabricClusterHealth -ApplicationHealthPolicyMap $appHealthPolicyMap -ApplicationsFilter
"Warning,Error" -NodesFilter "Warning,Error"


AggregatedHealthState   : Error
UnhealthyEvaluations    :
                          Unhealthy applications: 100% (1/1), MaxPercentUnhealthyApplications=0%.

                          Unhealthy application: ApplicationName='fabric:/WordCount',
AggregatedHealthState='Error'.

                              Unhealthy services: 100% (1/1), ServiceType='WordCountServiceType',
MaxPercentUnhealthyServices=0%.

                              Unhealthy service: ServiceName='fabric:/WordCount/WordCountService',
AggregatedHealthState='Error'.

                                  Unhealthy event: SourceId='System.PLB',
                          Property='ServiceReplicaUnplacedHealth_Secondary_a1f83a35-d6bf-4d39-b90d-
28d15f39599b', HealthState='Warning',
                          ConsiderWarningAsError=true.


NodeHealthStates        : None
ApplicationHealthStates :
                          ApplicationName       : fabric:/WordCount
                          AggregatedHealthState : Error

HealthEvents            : None
```

REST

You can get cluster health with a GET request or a POST request that includes health policies described in the body.

Get node health

Returns the health of a node entity and contains the health events reported on the node. Input:

- [Required] The node name that identifies the node.
- [Optional] The cluster health policy settings used to evaluate health.
- [Optional] Filters for events that specify which entries are of interest and should be returned in the result (for example, errors only, or both warnings and errors). All events are used to evaluate the entity aggregated health, regardless of the filter.

API

To get node health through the API, create a `FabricClient` and call the GetNodeHealthAsync method on its HealthManager.

The following code gets the node health for the specified node name:

```
NodeHealth nodeHealth = await fabricClient.HealthManager.GetNodeHealthAsync(nodeName);
```

The following code gets the node health for the specified node name and passes in events filter and custom policy through NodeHealthQueryDescription:

```
var queryDescription = new NodeHealthQueryDescription(nodeName)
{
    HealthPolicy = new ClusterHealthPolicy() {  ConsiderWarningAsError = true },
    EventsFilter = new HealthEventsFilter() { HealthStateFilterValue = HealthStateFilter.Warning },
};

NodeHealth nodeHealth = await fabricClient.HealthManager.GetNodeHealthAsync(queryDescription);
```

PowerShell

The cmdlet to get the node health is Get-ServiceFabricNodeHealth. First, connect to the cluster by using the Connect-ServiceFabricCluster cmdlet. The following cmdlet gets the node health by using default health policies:

```
PS C:\> Get-ServiceFabricNodeHealth _Node_1


NodeName              : _Node_1
AggregatedHealthState : Ok
HealthEvents          :
                        SourceId              : System.FM
                        Property              : State
                        HealthState           : Ok
                        SequenceNumber        : 6
                        SentAt                : 3/22/2016 7:47:56 PM
                        ReceivedAt            : 3/22/2016 7:48:19 PM
                        TTL                   : Infinite
                        Description           : Fabric node is up.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : Error->Ok = 3/22/2016 7:48:19 PM, LastWarning = 1/1/0001
12:00:00 AM
```

The following cmdlet gets the health of all nodes in the cluster:

```
PS C:\> Get-ServiceFabricNode | Get-ServiceFabricNodeHealth | select NodeName, AggregatedHealthState | ft -
AutoSize

NodeName AggregatedHealthState
-------- ---------------------
_Node_2                     Ok
_Node_0                     Ok
_Node_1                     Ok
_Node_3                     Ok
_Node_4                     Ok
```

REST

You can get node health with a GET request or a POST request that includes health policies described in the body.

Get application health

Returns the health of an application entity. It contains the health states of the deployed application and service children. Input:

- [Required] The application name (URI) that identifies the application.
- [Optional] The application health policy used to override the application manifest policies.
- [Optional] Filters for events, services, and deployed applications that specify which entries are of interest and should be returned in the result (for example, errors only, or both warnings and errors). All events, services, and deployed applications are used to evaluate the entity aggregated health, regardless of the filter.

API

To get application health, create a `FabricClient` and call the GetApplicationHealthAsync method on its HealthManager.

The following code gets the application health for the specified application name (URI):

```
ApplicationHealth applicationHealth = await
fabricClient.HealthManager.GetApplicationHealthAsync(applicationName);
```

The following code gets the application health for the specified application name (URI), with filters and custom policies specified via ApplicationHealthQueryDescription.

```
HealthStateFilter warningAndErrors = HealthStateFilter.Error | HealthStateFilter.Warning;
var serviceTypePolicy = new ServiceTypeHealthPolicy()
{
    MaxPercentUnhealthyPartitionsPerService = 0,
    MaxPercentUnhealthyReplicasPerPartition = 5,
    MaxPercentUnhealthyServices = 0,
};
var policy = new ApplicationHealthPolicy()
{
    ConsiderWarningAsError = false,
    DefaultServiceTypeHealthPolicy = serviceTypePolicy,
    MaxPercentUnhealthyDeployedApplications = 0,
};

var queryDescription = new ApplicationHealthQueryDescription(applicationName)
{
    HealthPolicy = policy,
    EventsFilter = new HealthEventsFilter() { HealthStateFilterValue = warningAndErrors },
    ServicesFilter = new ServiceHealthStatesFilter() { HealthStateFilterValue = warningAndErrors },
    DeployedApplicationsFilter = new DeployedApplicationHealthStatesFilter() { HealthStateFilterValue =
warningAndErrors },
};

ApplicationHealth applicationHealth = await
fabricClient.HealthManager.GetApplicationHealthAsync(queryDescription);
```

PowerShell

The cmdlet to get the application health is Get-ServiceFabricApplicationHealth. First, connect to the cluster by using the Connect-ServiceFabricCluster cmdlet.

The following cmdlet returns the health of the **fabric:/WordCount** application:

```
PS c:\>
PS C:\WINDOWS\system32>  Get-ServiceFabricApplicationHealth fabric:/WordCount


ApplicationName                 : fabric:/WordCount
AggregatedHealthState           : Warning
UnhealthyEvaluations            :
                                  Unhealthy services: 100% (1/1), ServiceType='WordCountServiceType',
MaxPercentUnhealthyServices=0%.

                                  Unhealthy service: ServiceName='fabric:/WordCount/WordCountService',
AggregatedHealthState='Warning'.

                                      Unhealthy event: SourceId='System.PLB',
                                  Property='ServiceReplicaUnplacedHealth_Secondary_a1f83a35-d6bf-4d39-b90d-
28d15f39599b', HealthState='Warning',
                                  ConsiderWarningAsError=false.

ServiceHealthStates             :
                                  ServiceName           : fabric:/WordCount/WordCountService
                                  AggregatedHealthState : Warning
```

```
                                  ServiceName           : fabric:/WordCount/WordCountWebService
                                  AggregatedHealthState : Ok

DeployedApplicationHealthStates :
                                  ApplicationName       : fabric:/WordCount
                                  NodeName              : _Node_0
                                  AggregatedHealthState : Ok

                                  ApplicationName       : fabric:/WordCount
                                  NodeName              : _Node_2
                                  AggregatedHealthState : Ok

                                  ApplicationName       : fabric:/WordCount
                                  NodeName              : _Node_3
                                  AggregatedHealthState : Ok

                                  ApplicationName       : fabric:/WordCount
                                  NodeName              : _Node_4
                                  AggregatedHealthState : Ok

                                  ApplicationName       : fabric:/WordCount
                                  NodeName              : _Node_1
                                  AggregatedHealthState : Ok

HealthEvents                    :
                                  SourceId              : System.CM
                                  Property              : State
                                  HealthState           : Ok
                                  SequenceNumber        : 360
                                  SentAt                : 3/22/2016 7:56:53 PM
                                  ReceivedAt            : 3/22/2016 7:56:53 PM
                                  TTL                   : Infinite
                                  Description           : Application has been created.
                                  RemoveWhenExpired     : False
                                  IsExpired             : False
                                  Transitions           : Error->Ok = 3/22/2016 7:56:53 PM, LastWarning =
1/1/0001 12:00:00 AM

                                  SourceId              : MyWatchdog
                                  Property              : Availability
                                  HealthState           : Ok
                                  SequenceNumber        : 131031545225930951
                                  SentAt                : 3/22/2016 9:08:42 PM
                                  ReceivedAt            : 3/22/2016 9:08:42 PM
                                  TTL                   : Infinite
                                  Description           : Availability checked successfully, latency ok
                                  RemoveWhenExpired     : False
                                  IsExpired             : False
                                  Transitions           : Error->Ok = 3/22/2016 8:55:39 PM, LastWarning =
1/1/0001 12:00:00 AM
```

The following PowerShell cmdlet passes in custom policies. It also filters children and events.

```
PS C:\> Get-ServiceFabricApplicationHealth -ApplicationName fabric:/WordCount -ConsiderWarningAsError $true -
ServicesFilter Error -EventsFilter Error -DeployedApplicationsFilter Error

ApplicationName                : fabric:/WordCount
AggregatedHealthState          : Error
UnhealthyEvaluations           :
                                 Unhealthy services: 100% (1/1), ServiceType='WordCountServiceType',
MaxPercentUnhealthyServices=0%.

                                 Unhealthy service: ServiceName='fabric:/WordCount/WordCountService',
AggregatedHealthState='Error'.

                                     Unhealthy event: SourceId='System.PLB',
                                 Property='ServiceReplicaUnplacedHealth_Secondary_a1f83a35-d6bf-4d39-b90d-
28d15f39599b', HealthState='Warning',
                                 ConsiderWarningAsError=true.

ServiceHealthStates            :
                                 ServiceName           : fabric:/WordCount/WordCountService
                                 AggregatedHealthState : Error

DeployedApplicationHealthStates : None
HealthEvents                   : None
```

REST

You can get application health with a GET request or a POST request that includes health policies described in the body.

Get service health

Returns the health of a service entity. It contains the partition health states. Input:

- [Required] The service name (URI) that identifies the service.
- [Optional] The application health policy used to override the application manifest policy.
- [Optional] Filters for events and partitions that specify which entries are of interest and should be returned in the result (for example, errors only, or both warnings and errors). All events and partitions are used to evaluate the entity aggregated health, regardless of the filter.

API

To get service health through the API, create a `FabricClient` and call the GetServiceHealthAsync method on its HealthManager.

The following example gets the health of a service with specified service name (URI):

```
ServiceHealth serviceHealth = await fabricClient.HealthManager.GetServiceHealthAsync(serviceName);
```

The following code gets the service health for the specified service name (URI), specifying filters and custom policy via ServiceHealthQueryDescription:

```
var queryDescription = new ServiceHealthQueryDescription(serviceName)
{
    EventsFilter = new HealthEventsFilter() { HealthStateFilterValue = HealthStateFilter.All },
    PartitionsFilter = new PartitionHealthStatesFilter() { HealthStateFilterValue = HealthStateFilter.Error },
};

ServiceHealth serviceHealth = await fabricClient.HealthManager.GetServiceHealthAsync(queryDescription);
```

PowerShell

The cmdlet to get the service health is Get-ServiceFabricServiceHealth. First, connect to the cluster by using the

Connect-ServiceFabricCluster cmdlet.

The following cmdlet gets the service health by using default health policies:

```
PS C:\> Get-ServiceFabricServiceHealth -ServiceName fabric:/WordCount/WordCountService


ServiceName           : fabric:/WordCount/WordCountService
AggregatedHealthState : Warning
UnhealthyEvaluations  :
                        Unhealthy event: SourceId='System.PLB',
                        Property='ServiceReplicaUnplacedHealth_Secondary_a1f83a35-d6bf-4d39-b90d-28d15f39599b',
HealthState='Warning',
                        ConsiderWarningAsError=false.

PartitionHealthStates :
                        PartitionId           : a1f83a35-d6bf-4d39-b90d-28d15f39599b
                        AggregatedHealthState : Warning

HealthEvents          :
                        SourceId              : System.FM
                        Property              : State
                        HealthState           : Ok
                        SequenceNumber        : 10
                        SentAt                : 3/22/2016 7:56:53 PM
                        ReceivedAt            : 3/22/2016 7:57:18 PM
                        TTL                   : Infinite
                        Description           : Service has been created.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : Error->Ok = 3/22/2016 7:57:18 PM, LastWarning = 1/1/0001
12:00:00 AM

                        SourceId              : System.PLB
                        Property              : ServiceReplicaUnplacedHealth_Secondary_a1f83a35-d6bf-4d39-b90d-
28d15f39599b
                        HealthState           : Warning
                        SequenceNumber        : 131031547693687021
                        SentAt                : 3/22/2016 9:12:49 PM
                        ReceivedAt            : 3/22/2016 9:12:49 PM
                        TTL                   : 00:01:05
                        Description           : The Load Balancer was unable to find a placement for one or more
of the Service's Replicas:
                        fabric:/WordCount/WordCountService Secondary Partition a1f83a35-d6bf-4d39-b90d-
28d15f39599b could not be placed, possibly,
                        due to the following constraints and properties:
                        Placement Constraint: N/A
                        Depended Service: N/A

                        Constraint Elimination Sequence:
                        ReplicaExclusionStatic eliminated 4 possible node(s) for placement -- 1/5 node(s)
remain.
                        ReplicaExclusionDynamic eliminated 1 possible node(s) for placement -- 0/5 node(s)
remain.

                        Nodes Eliminated By Constraints:

                        ReplicaExclusionStatic:
                        FaultDomain:fd:/0 NodeName:_Node_0 NodeType:NodeType0 UpgradeDomain:0 UpgradeDomain:
ud:/0 Deactivation Intent/Status:
                        None/None
                        FaultDomain:fd:/1 NodeName:_Node_1 NodeType:NodeType1 UpgradeDomain:1 UpgradeDomain:
ud:/1 Deactivation Intent/Status:
                        None/None
                        FaultDomain:fd:/3 NodeName:_Node_3 NodeType:NodeType3 UpgradeDomain:3 UpgradeDomain:
ud:/3 Deactivation Intent/Status:
                        None/None
                        FaultDomain:fd:/4 NodeName:_Node_4 NodeType:NodeType4 UpgradeDomain:4 UpgradeDomain:
```

```
ud:/4 Deactivation Intent/Status:
                        None/None

                        ReplicaExclusionDynamic:
                        FaultDomain:fd:/2 NodeName:_Node_2 NodeType:NodeType2 UpgradeDomain:2 UpgradeDomain:
ud:/2 Deactivation Intent/Status:
                        None/None


                        RemoveWhenExpired     : True
                        IsExpired             : False
```

REST

You can get service health with a GET request or a POST request that includes health policies described in the body.

Get partition health

Returns the health of a partition entity. It contains the replica health states. Input:

- [Required] The partition ID (GUID) that identifies the partition.
- [Optional] The application health policy used to override the application manifest policy.
- [Optional] Filters for events and replicas that specify which entries are of interest and should be returned in the result (for example, errors only, or both warnings and errors). All events and replicas are used to evaluate the entity aggregated health, regardless of the filter.

API

To get partition health through the API, create a `FabricClient` and call the GetPartitionHealthAsync method on its HealthManager. To specify optional parameters, create PartitionHealthQueryDescription.

```
PartitionHealth partitionHealth = await fabricClient.HealthManager.GetPartitionHealthAsync(partitionId);
```

PowerShell

The cmdlet to get the partition health is Get-ServiceFabricPartitionHealth. First, connect to the cluster by using the Connect-ServiceFabricCluster cmdlet.

The following cmdlet gets the health for all partitions of the **fabric:/WordCount/WordCountService** service:

```
PS C:\> Get-ServiceFabricPartition fabric:/WordCount/WordCountService | Get-ServiceFabricPartitionHealth


PartitionId           : a1f83a35-d6bf-4d39-b90d-28d15f39599b
AggregatedHealthState : Warning
UnhealthyEvaluations  :
                        Unhealthy event: SourceId='System.FM', Property='State', HealthState='Warning',
ConsiderWarningAsError=false.

ReplicaHealthStates   :
                        ReplicaId             : 131031502143040223
                        AggregatedHealthState : Ok

                        ReplicaId             : 131031502346844060
                        AggregatedHealthState : Ok

                        ReplicaId             : 131031502346844059
                        AggregatedHealthState : Ok

                        ReplicaId             : 131031502346844061
                        AggregatedHealthState : Ok

                        ReplicaId             : 131031502346844058
                        AggregatedHealthState : Ok

HealthEvents          :
                        SourceId              : System.FM
                        Property              : State
                        HealthState           : Warning
                        SequenceNumber        : 76
                        SentAt                : 3/22/2016 7:57:26 PM
                        ReceivedAt            : 3/22/2016 7:57:48 PM
                        TTL                   : Infinite
                        Description           : Partition is below target replica or instance count.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : Error->Warning = 3/22/2016 7:57:48 PM, LastOk = 1/1/0001
12:00:00 AM
```

REST

You can get partition health with a GET request or a POST request that includes health policies described in the body.

Get replica health

Returns the health of a stateful service replica or a stateless service instance. Input:

- [Required] The partition ID (GUID) and replica ID that identifies the replica.
- [Optional] The application health policy parameters used to override the application manifest policies.
- [Optional] Filters for events that specify which entries are of interest and should be returned in the result (for example, errors only, or both warnings and errors). All events are used to evaluate the entity aggregated health, regardless of the filter.

API

To get the replica health through the API, create a `FabricClient` and call the GetReplicaHealthAsync method on its HealthManager. To specify advanced parameters, use ReplicaHealthQueryDescription.

```
ReplicaHealth replicaHealth = await fabricClient.HealthManager.GetReplicaHealthAsync(partitionId, replicaId);
```

PowerShell

The cmdlet to get the replica health is Get-ServiceFabricReplicaHealth. First, connect to the cluster by using the Connect-ServiceFabricCluster cmdlet.

The following cmdlet gets the health of the primary replica for all partitions of the service:

```
PS C:\> Get-ServiceFabricPartition fabric:/WordCount/WordCountService | Get-ServiceFabricReplica | where
{$_.ReplicaRole -eq "Primary"} | Get-ServiceFabricReplicaHealth


PartitionId           : a1f83a35-d6bf-4d39-b90d-28d15f39599b
ReplicaId             : 131031502143040223
AggregatedHealthState : Ok
HealthEvents          :
                        SourceId              : System.RA
                        Property              : State
                        HealthState           : Ok
                        SequenceNumber        : 131031502145556748
                        SentAt                : 3/22/2016 7:56:54 PM
                        ReceivedAt            : 3/22/2016 7:57:12 PM
                        TTL                   : Infinite
                        Description           : Replica has been created.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : Error->Ok = 3/22/2016 7:57:12 PM, LastWarning = 1/1/0001
12:00:00 AM
```

REST

You can get replica health with a GET request or a POST request that includes health policies described in the body.

Get deployed application health

Returns the health of an application deployed on a node entity. It contains the deployed service package health states. Input:

- [Required] The application name (URI) and node name (string) that identify the deployed application.
- [Optional] The application health policy used to override the application manifest policies.
- [Optional] Filters for events and deployed service packages that specify which entries are of interest and should be returned in the result (for example, errors only, or both warnings and errors). All events and deployed service packages are used to evaluate the entity aggregated health, regardless of the filter.

API

To get the health of an application deployed on a node through the API, create a `FabricClient` and call the GetDeployedApplicationHealthAsync method on its HealthManager. To specify optional parameters, use DeployedApplicationHealthQueryDescription.

```
DeployedApplicationHealth health = await fabricClient.HealthManager.GetDeployedApplicationHealthAsync(
    new DeployedApplicationHealthQueryDescription(applicationName, nodeName));
```

PowerShell

The cmdlet to get the deployed application health is Get-ServiceFabricDeployedApplicationHealth. First, connect to the cluster by using the Connect-ServiceFabricCluster cmdlet. To find out where an application is deployed, run Get-ServiceFabricApplicationHealth and look at the deployed application children.

The following cmdlet gets the health of the **fabric:/WordCount** application deployed on **_Node_2**.

```
PS C:\> Get-ServiceFabricDeployedApplicationHealth -ApplicationName fabric:/WordCount -NodeName _Node_2


ApplicationName                    : fabric:/WordCount
NodeName                           : _Node_2
AggregatedHealthState              : Ok
DeployedServicePackageHealthStates :
                                     ServiceManifestName   : WordCountServicePkg
                                     NodeName              : _Node_2
                                     AggregatedHealthState : Ok

                                     ServiceManifestName   : WordCountWebServicePkg
                                     NodeName              : _Node_2
                                     AggregatedHealthState : Ok

HealthEvents                       :
                                     SourceId              : System.Hosting
                                     Property              : Activation
                                     HealthState           : Ok
                                     SequenceNumber        : 131031502143710698
                                     SentAt                : 3/22/2016 7:56:54 PM
                                     ReceivedAt            : 3/22/2016 7:57:12 PM
                                     TTL                   : Infinite
                                     Description           : The application was activated successfully.
                                     RemoveWhenExpired     : False
                                     IsExpired             : False
                                     Transitions           : Error->Ok = 3/22/2016 7:57:12 PM, LastWarning =
1/1/0001 12:00:00 AM
```

REST

You can get deployed application health with a GET request or a POST request that includes health policies described in the body.

Get deployed service package health

Returns the health of a deployed service package entity. Input:

- [Required] The application name (URI), node name (string), and service manifest name (string) that identify the deployed service package.
- [Optional] The application health policy used to override the application manifest policy.
- [Optional] Filters for events that specify which entries are of interest and should be returned in the result (for example, errors only, or both warnings and errors). All events are used to evaluate the entity aggregated health, regardless of the filter.

API

To get the health of a deployed service package through the API, create a `FabricClient` and call the GetDeployedServicePackageHealthAsync method on its HealthManager. To specify optional parameters, use DeployedServicePackageHealthQueryDescription.

```
DeployedServicePackageHealth health = await fabricClient.HealthManager.GetDeployedServicePackageHealthAsync(
    new DeployedServicePackageHealthQueryDescription(applicationName, nodeName, serviceManifestName));
```

PowerShell

The cmdlet to get the deployed service package health is Get-ServiceFabricDeployedServicePackageHealth. First, connect to the cluster by using the Connect-ServiceFabricCluster cmdlet. To see where an application is deployed, run Get-ServiceFabricApplicationHealth and look at the deployed applications. To see which service packages are in an application, look at the deployed service package children in the Get-ServiceFabricDeployedApplicationHealth output.

The following cmdlet gets the health of the **WordCountServicePkg** service package of the **fabric:/WordCount** application deployed on **_Node_2**. The entity has **System.Hosting** reports for successful service-package and entry-point activation, and successful service-type registration.

```
PS C:\> Get-ServiceFabricDeployedApplication -ApplicationName fabric:/WordCount -NodeName _Node_2 | Get-
ServiceFabricDeployedServicePackageHealth -ServiceManifestName WordCountServicePkg


ApplicationName       : fabric:/WordCount
ServiceManifestName   : WordCountServicePkg
NodeName              : _Node_2
AggregatedHealthState : Ok
HealthEvents          :
                        SourceId              : System.Hosting
                        Property              : Activation
                        HealthState           : Ok
                        SequenceNumber        : 131031502301306211
                        SentAt                : 3/22/2016 7:57:10 PM
                        ReceivedAt            : 3/22/2016 7:57:12 PM
                        TTL                   : Infinite
                        Description           : The ServicePackage was activated successfully.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : Error->Ok = 3/22/2016 7:57:12 PM, LastWarning = 1/1/0001
12:00:00 AM

                        SourceId              : System.Hosting
                        Property              : CodePackageActivation:Code:EntryPoint
                        HealthState           : Ok
                        SequenceNumber        : 131031502301568982
                        SentAt                : 3/22/2016 7:57:10 PM
                        ReceivedAt            : 3/22/2016 7:57:12 PM
                        TTL                   : Infinite
                        Description           : The CodePackage was activated successfully.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : Error->Ok = 3/22/2016 7:57:12 PM, LastWarning = 1/1/0001
12:00:00 AM

                        SourceId              : System.Hosting
                        Property              : ServiceTypeRegistration:WordCountServiceType
                        HealthState           : Ok
                        SequenceNumber        : 131031502314788519
                        SentAt                : 3/22/2016 7:57:11 PM
                        ReceivedAt            : 3/22/2016 7:57:12 PM
                        TTL                   : Infinite
                        Description           : The ServiceType was registered successfully.
                        RemoveWhenExpired     : False
                        IsExpired             : False
                        Transitions           : Error->Ok = 3/22/2016 7:57:12 PM, LastWarning = 1/1/0001
12:00:00 AM
```

REST

You can get deployed service package health with a GET request or a POST request that includes health policies described in the body.

Health chunk queries

The health chunk queries can return multi-level cluster children (recursively), per input filters. It supports advanced filters that allow much flexibility to express which specific children to be returned, identified by their unique identifier or other group identifier and/or health state. By default, no children are included, as opposed to health commands that always include first-level children.

The health queries return only first-level children of the specified entity per required filters. To get the children of the children, you must call additional health APIs for each entity of interest. Similarly, to get the health of specific entities, you must call one health API for each desired entity. The chunk query advanced filtering allows you to request multiple items of interest in one query, minimizing the message size and the number of messages.

The value of the chunk query is that you can get health state for more cluster entities (potentially all cluster entities starting at required root) in one call. You can express complex health query such as:

- Return only applications at error, and for those applications include all services at warning|error. For returned services, include all partitions.
- Return only the health of 4 applications, specified by their names.
- Return only the health of applications of a desired application type.
- Return all deployed entities on a node. Returns all applications, all deployed applications on the specified node and all the deployed service packages on that node.
- Return all replicas at error. Returns all applications, services, partitions, and only replicas at error.
- Return all applications. For a specified service, include all partitions.

Currently, the health chunk query is exposed only for the cluster entity. It returns a cluster health chunk, which contains:

- The cluster aggregated health state.
- The health state chunk list of nodes that respect input filters.
- The health state chunk list of applications that respect input filters. Each application health state chunk contains a chunk list with all services that respect input filters and a chunk list with all deployed applications that respect the filters. Same for the children of services and deployed applications. This way, all entities in the cluster can be potentially returned if requested, in a hierarchical fashion.

Cluster health chunk query

Returns the health of the cluster entity and contains the hierarchical health state chunks of required children. Input:

- [Optional] The cluster health policy used to evaluate the nodes and the cluster events.
- [Optional] The application health policy map, with the health policies used to override the application manifest policies.
- [Optional] Filters for nodes and applications that specify which entries are of interest and should be returned in the result. The filters are specific to an entity/group of entities or are applicable to all entities at that level. The list of filters can contain one general filter and/or filters for specific identifiers to fine-grain entities returned by the query. If empty, the children are not returned by default. Read more about the filters at NodeHealthStateFilter and ApplicationHealthStateFilter. The application filters can recursively specify advanced filters for children.

The chunk result includes the children that respect the filters.

Currently, the chunk query does not return unhealthy evaluations or entity events. That extra information can be obtained using the existing cluster health query.

API

To get cluster health chunk, create a `FabricClient` and call the GetClusterHealthChunkAsync method on its **HealthManager**. You can pass in ClusterHealthQueryDescription to describe health policies and advanced filters.

The following code gets cluster health chunk with advanced filters.

```
var queryDescription = new ClusterHealthChunkQueryDescription();
queryDescription.ApplicationFilters.Add(new ApplicationHealthStateFilter()
    {
        // Return applications only if they are in error
        HealthStateFilter = HealthStateFilter.Error
    });

// Return all replicas
var wordCountServiceReplicaFilter = new ReplicaHealthStateFilter()
    {
        HealthStateFilter = HealthStateFilter.All
    };

// Return all replicas and all partitions
var wordCountServicePartitionFilter = new PartitionHealthStateFilter()
    {
        HealthStateFilter = HealthStateFilter.All
    };
wordCountServicePartitionFilter.ReplicaFilters.Add(wordCountServiceReplicaFilter);

// For specific service, return all partitions and all replicas
var wordCountServiceFilter = new ServiceHealthStateFilter()
{
    ServiceNameFilter = new Uri("fabric:/WordCount/WordCountService"),
};
wordCountServiceFilter.PartitionFilters.Add(wordCountServicePartitionFilter);

// Application filter: for specific application, return no services except the ones of interest
var wordCountApplicationFilter = new ApplicationHealthStateFilter()
    {
        // Always return fabric:/WordCount application
        ApplicationNameFilter = new Uri("fabric:/WordCount"),
    };
wordCountApplicationFilter.ServiceFilters.Add(wordCountServiceFilter);

queryDescription.ApplicationFilters.Add(wordCountApplicationFilter);

var result = await fabricClient.HealthManager.GetClusterHealthChunkAsync(queryDescription);
```

PowerShell

The cmdlet to get the cluster health is Get-ServiceFabricClusterChunkHealth. First, connect to the cluster by using the Connect-ServiceFabricCluster cmdlet.

The following code gets nodes only if they are in Error except for a specific node, which should always be returned.

```
PS C:\> $errorFilter = [System.Fabric.Health.HealthStateFilter]::Error;
$allFilter = [System.Fabric.Health.HealthStateFilter]::All;

$nodeFilter1 = New-Object System.Fabric.Health.NodeHealthStateFilter -Property @{HealthStateFilter=$errorFilter}
$nodeFilter2 = New-Object System.Fabric.Health.NodeHealthStateFilter -Property
@{NodeNameFilter="_Node_1";HealthStateFilter=$allFilter}
# Create node filter list that will be passed in the cmdlet
$nodeFilters = New-Object System.Collections.Generic.List[System.Fabric.Health.NodeHealthStateFilter]
$nodeFilters.Add($nodeFilter1)
$nodeFilters.Add($nodeFilter2)

Get-ServiceFabricClusterHealthChunk -NodeFilters $nodeFilters

HealthState                  : Error
NodeHealthStateChunks        :
                               TotalCount            : 1

                               NodeName              : _Node_1
                               HealthState           : Ok

ApplicationHealthStateChunks : None
```

The following cmdlet gets cluster chunk with application filters.

```
$errorFilter = [System.Fabric.Health.HealthStateFilter]::Error;
$allFilter = [System.Fabric.Health.HealthStateFilter]::All;

# All replicas
$replicaFilter = New-Object System.Fabric.Health.ReplicaHealthStateFilter -Property
@{HealthStateFilter=$allFilter}

# All partitions
$partitionFilter = New-Object System.Fabric.Health.PartitionHealthStateFilter -Property
@{HealthStateFilter=$allFilter}
$partitionFilter.ReplicaFilters.Add($replicaFilter)

# For WordCountService, return all partitions and all replicas
$svcFilter1 = New-Object System.Fabric.Health.ServiceHealthStateFilter -Property
@{ServiceNameFilter="fabric:/WordCount/WordCountService"}
$svcFilter1.PartitionFilters.Add($partitionFilter)

$svcFilter2 = New-Object System.Fabric.Health.ServiceHealthStateFilter -Property
@{HealthStateFilter=$errorFilter}

$appFilter = New-Object System.Fabric.Health.ApplicationHealthStateFilter -Property
@{ApplicationNameFilter="fabric:/WordCount"}
$appFilter.ServiceFilters.Add($svcFilter1)
$appFilter.ServiceFilters.Add($svcFilter2)

$appFilters = New-Object System.Collections.Generic.List[System.Fabric.Health.ApplicationHealthStateFilter]
$appFilters.Add($appFilter)

Get-ServiceFabricClusterHealthChunk -ApplicationFilters $appFilters

HealthState                  : Error
NodeHealthStateChunks        : None
ApplicationHealthStateChunks :
                               TotalCount            : 1

                               ApplicationName       : fabric:/WordCount
                               ApplicationTypeName   : WordCount
                               HealthState           : Error
                               ServiceHealthStateChunks :
                                   TotalCount            : 1

                                   ServiceName           : fabric:/WordCount/WordCountService
                                   HealthState           : Error
                                   PartitionHealthStateChunks :
                                       TotalCount            : 1

                                       PartitionId           : a1f83a35-d6bf-4d39-b90d-28d15f39599b
                                       HealthState           : Error
                                       ReplicaHealthStateChunks :
                                           TotalCount            : 5

                                           ReplicaOrInstanceId   : 131031502143040223
                                           HealthState           : Ok

                                           ReplicaOrInstanceId   : 131031502346844060
                                           HealthState           : Ok

                                           ReplicaOrInstanceId   : 131031502346844059
                                           HealthState           : Ok

                                           ReplicaOrInstanceId   : 131031502346844061
                                           HealthState           : Ok

                                           ReplicaOrInstanceId   : 131031502346844058
                                           HealthState           : Error
```

The following cmdlet returns all deployed entities on a node.

```
$errorFilter = [System.Fabric.Health.HealthStateFilter]::Error;
$allFilter = [System.Fabric.Health.HealthStateFilter]::All;

$dspFilter = New-Object System.Fabric.Health.DeployedServicePackageHealthStateFilter -Property
@{HealthStateFilter=$allFilter}
$daFilter =  New-Object System.Fabric.Health.DeployedApplicationHealthStateFilter -Property
@{HealthStateFilter=$allFilter;NodeNameFilter="_Node_2"}
$daFilter.DeployedServicePackageFilters.Add($dspFilter)

$appFilter = New-Object System.Fabric.Health.ApplicationHealthStateFilter -Property
@{HealthStateFilter=$allFilter}
$appFilter.DeployedApplicationFilters.Add($daFilter)

$appFilters = New-Object System.Collections.Generic.List[System.Fabric.Health.ApplicationHealthStateFilter]
$appFilters.Add($appFilter)
Get-ServiceFabricClusterHealthChunk -ApplicationFilters $appFilters


HealthState                  : Error
NodeHealthStateChunks        : None
ApplicationHealthStateChunks :
                               TotalCount            : 2

                               ApplicationName       : fabric:/System
                               HealthState           : Ok
                               DeployedApplicationHealthStateChunks :
                                   TotalCount            : 1

                                   NodeName              : _Node_2
                                   HealthState           : Ok
                                   DeployedServicePackageHealthStateChunks :
                                       TotalCount            : 1

                                       ServiceManifestName   : FAS
                                       HealthState           : Ok



                               ApplicationName       : fabric:/WordCount
                               ApplicationTypeName   : WordCount
                               HealthState           : Error
                               DeployedApplicationHealthStateChunks :
                                   TotalCount            : 1

                                   NodeName              : _Node_2
                                   HealthState           : Ok
                                   DeployedServicePackageHealthStateChunks :
                                       TotalCount            : 2

                                       ServiceManifestName   : WordCountServicePkg
                                       HealthState           : Ok

                                       ServiceManifestName   : WordCountWebServicePkg
                                       HealthState           : Ok
```

REST

You can get cluster health chunk with a GET request or a POST request that includes health policies and advanced filters described in the body.

General queries

General queries return a list of Service Fabric entities of a specified type. They are exposed through the API (via the methods on **FabricClient.QueryManager**), PowerShell cmdlets, and REST. These queries aggregate subqueries from multiple components. One of them is the health store, which populates the aggregated health state for each

query result.

> **NOTE**
> General queries return the aggregated health state of the entity and do not contain rich health data. If an entity is not healthy, you can follow up with health queries to get all its health information, including events, child health states, and unhealthy evaluations.

If general queries return an unknown health state for an entity, it's possible that the health store doesn't have complete data about the entity. It's also possible that a subquery to the health store wasn't successful (for example, there was a communication error, or the health store was throttled). Follow up with a health query for the entity. If the subquery encountered transient errors, such as network issues, this follow-up query may succeed. It may also give you more details from the health store about why the entity is not exposed.

The queries that contain **HealthState** for entities are:

- Node list: Returns the list nodes in the cluster (paged).
  - API: FabricClient.QueryClient.GetNodeListAsync
  - PowerShell: Get-ServiceFabricNode
- Application list: Returns the list of applications in the cluster (paged).
  - API: FabricClient.QueryClient.GetApplicationListAsync
  - PowerShell: Get-ServiceFabricApplication
- Service list: Returns the list of services in an application (paged).
  - API: FabricClient.QueryClient.GetServiceListAsync
  - PowerShell: Get-ServiceFabricService
- Partition list: Returns the list of partitions in a service (paged).
  - API: FabricClient.QueryClient.GetPartitionListAsync
  - PowerShell: Get-ServiceFabricPartition
- Replica list: Returns the list of replicas in a partition (paged).
  - API: FabricClient.QueryClient.GetReplicaListAsync
  - PowerShell: Get-ServiceFabricReplica
- Deployed application list: Returns the list of deployed applications on a node.
  - API: FabricClient.QueryClient.GetDeployedApplicationListAsync
  - PowerShell: Get-ServiceFabricDeployedApplication
- Deployed service package list: Returns the list of service packages in a deployed application.
  - API: FabricClient.QueryClient.GetDeployedServicePackageListAsync
  - PowerShell: Get-ServiceFabricDeployedApplication

> **NOTE**
> Some of the queries return paged results. The return of these queries is a list derived from PagedList. If the results do not fit a message, only a page is returned and a ContinuationToken that tracks where enumeration stopped. You should continue to call the same query and pass in the continuation token from the previous query to get next results.

Examples

The following code gets the unhealthy applications in the cluster:

```
var applications = fabricClient.QueryManager.GetApplicationListAsync().Result.Where(
  app => app.HealthState == HealthState.Error);
```

The following cmdlet gets the application details for the fabric:/WordCount application. Notice that health state is

at warning.

```
PS C:\> Get-ServiceFabricApplication -ApplicationName fabric:/WordCount

ApplicationName        : fabric:/WordCount
ApplicationTypeName    : WordCount
ApplicationTypeVersion : 1.0.0
ApplicationStatus      : Ready
HealthState            : Warning
ApplicationParameters  : { "WordCountWebService_InstanceCount" = "1";
                         "_WFDebugParams_" = "
[{"ServiceManifestName":"WordCountWebServicePkg","CodePackageName":"Code","EntryPointType":"Main","Debug
                         ExePath":"C:\\Program Files (x86)\\Microsoft Visual Studio
                         14.0\\Common7\\Packages\\Debugger\\VsDebugLaunchNotify.exe","DebugArguments":"
{74f7e5d5-71a9-47e2-a8cd-1878ec4734f1} -p
                         [ProcessId] -tid [ThreadId]","EnvironmentBlock":"_NO_DEBUG_HEAP=1\u0000"},
{"ServiceManifestName":"WordCountServicePkg","CodeP
                         ackageName":"Code","EntryPointType":"Main","DebugExePath":"C:\\Program Files
(x86)\\Microsoft Visual Studio
                         14.0\\Common7\\Packages\\Debugger\\VsDebugLaunchNotify.exe","DebugArguments":"
{2ab462e6-e0d1-4fda-a844-972f561fe751} -p
                         [ProcessId] -tid [ThreadId]","EnvironmentBlock":"_NO_DEBUG_HEAP=1\u0000"}]" }
```

The following cmdlet gets the services with a health state of warning:

```
PS C:\> Get-ServiceFabricApplication | Get-ServiceFabricService | where {$_.HealthState -eq "Warning"}


ServiceName            : fabric:/WordCount/WordCountService
ServiceKind            : Stateful
ServiceTypeName        : WordCountServiceType
IsServiceGroup         : False
ServiceManifestVersion : 1.0.0
HasPersistedState      : True
ServiceStatus          : Active
HealthState            : Warning
```

Cluster and application upgrades

During a monitored upgrade of the cluster and application, Service Fabric checks health to ensure that everything remains healthy. If an entity is unhealthy as evaluated by using configured health policies, the upgrade applies upgrade-specific policies to determine the next action. The upgrade may be paused to allow user interaction (such as fixing error conditions or changing policies), or it may automatically roll back to the previous good version.

During a *cluster* upgrade, you can get the cluster upgrade status. The upgrade status includes unhealthy evaluations, which point to what is unhealthy in the cluster. If the upgrade is rolled back due to health issues, the upgrade status remembers the last unhealthy reasons. This information can help administrators investigate what went wrong after the upgrade rolled back or stopped.

Similarly, during an *application* upgrade, any unhealthy evaluations are contained in the application upgrade status.

The following shows the application upgrade status for a modified fabric:/WordCount application. A watchdog reported an error on one of its replicas. The upgrade is rolling back because the health checks are not respected.

```
PS C:\> Get-ServiceFabricApplicationUpgrade fabric:/WordCount

ApplicationName               : fabric:/WordCount
ApplicationTypeName           : WordCount
TargetApplicationTypeVersion  : 1.0.0.0
ApplicationParameters         : {}
StartTimestampUtc             : 4/21/2015 5:23:26 PM
FailureTimestampUtc           : 4/21/2015 5:23:37 PM
FailureReason                 : HealthCheck
UpgradeState                  : RollingBackInProgress
UpgradeDuration               : 00:00:23
CurrentUpgradeDomainDuration  : 00:00:00
CurrentUpgradeDomainProgress  : UD1

                                NodeName            : _Node_1
                                UpgradePhase        : Upgrading

                                NodeName            : _Node_2
                                UpgradePhase        : Upgrading

                                NodeName            : _Node_3
                                UpgradePhase        : PreUpgradeSafetyCheck
                                PendingSafetyChecks :
                                EnsurePartitionQuorum - PartitionId: 30db5be6-4e20-4698-8185-4bd7ca744020
NextUpgradeDomain             : UD2
UpgradeDomainsStatus          : { "UD1" = "Completed";
                                "UD2" = "Pending";
                                "UD3" = "Pending";
                                "UD4" = "Pending" }
UnhealthyEvaluations          :
                                Unhealthy services: 100% (1/1), ServiceType='WordCountServiceType',
MaxPercentUnhealthyServices=0%.

                                  Unhealthy service: ServiceName='fabric:/WordCount/WordCountService',
AggregatedHealthState='Error'.

                                      Unhealthy partitions: 100% (1/1),
MaxPercentUnhealthyPartitionsPerService=0%.

                                      Unhealthy partition: PartitionId='a1f83a35-d6bf-4d39-b90d-28d15f39599b',
AggregatedHealthState='Error'.

                                          Unhealthy replicas: 20% (1/5),
MaxPercentUnhealthyReplicasPerPartition=0%.

                                          Unhealthy replica: PartitionId='a1f83a35-d6bf-4d39-b90d-28d15f39599b',
                                  ReplicaOrInstanceId='131031502346844058', AggregatedHealthState='Error'.

                                              Error event: SourceId='DiskWatcher', Property='Disk'.

UpgradeKind                   : Rolling
RollingUpgradeMode            : UnmonitoredAuto
ForceRestart                  : False
UpgradeReplicaSetCheckTimeout : 00:15:00
```

Read more about the Service Fabric application upgrade.

Use health evaluations to troubleshoot

Whenever there is an issue with the cluster or an application, look at the cluster or application health to pinpoint what is wrong. The unhealthy evaluations provide details about what triggered the current unhealthy state. If you need to, you can drill down into unhealthy child entities to identify the root cause.

> **NOTE**
> The unhealthy evaluations show the first reason the entity is evaluated to current health state. There may be multiple other events that trigger this state, but they are not be reflected in the evaluations. To get more information, drill down into the health entities to figure out all the unhealthy reports in the cluster.

Next steps

Use system health reports to troubleshoot

Add custom Service Fabric health reports

How to report and check service health

Monitor and diagnose services locally

Service Fabric application upgrade

Monitor and diagnose services in a local machine development setup

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

Todd Abel • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Mark Fussell • Ryan Wike • Rebecca Agiewich • Matt Thalman • kunaldsingh • Jamie Strusz • Andy De George • Sean McKenna

Monitoring, detecting, diagnosing, and troubleshooting allow for services to continue with minimal disruption to the user experience. While monitoring and diagnostics are critical in an actual deployed production environment, the efficacy will depend on adopting a similar model during development of services to ensure they work when you move to a real-world setup. Service Fabric makes it easy for service developers to implement diagnostics that can seamlessly work across both single-machine local development setups and real-world production cluster setups.

The benefits of Event Tracing for Windows

Event Tracing for Windows (ETW) is the recommended technology for tracing messages in Service Fabric. Reasons for this are:

- **ETW is fast.** It was built as a tracing technology that has minimal impact on code execution times.
- **ETW tracing works seamlessly across local development environments and also real-world cluster setups.** This means you don't have to rewrite your tracing code when you are ready to deploy your code to a real cluster.
- **Service Fabric system code also uses ETW for internal tracing.** This allows you to view your application traces interleaved with Service Fabric system traces. It also helps you to more easily understand the sequences and interrelationships between your application code and events in the underlying system.
- **There is built-in support in Service Fabric Visual Studio tools to view ETW events.**

View Service Fabric system events in Visual Studio

Service Fabric emits ETW events to help application developers understand what's happening in the platform. If you haven't already done so, go ahead and follow the steps in Creating your first application in Visual Studio. This information will help you get an application up and running with the Diagnostics Events Viewer showing the trace messages.

1. If the diagnostics events window does not automatically show, Go to the **View** tab in Visual Studio, choose **Other Windows** and then **Diagnostic Events Viewer**.
2. Each event has standard metadata information that tells you the node, application and service the event is coming from. You can also filter the list of events by using the **Filter events** box at the top of the events window. For example, you can filter on **Node Name** or **Service Name.** And when you're looking at event details, you can also pause by using the **Pause** button at the top of the events window and resume later without any loss of events.

Add your own custom traces to the application code

The Service Fabric Visual Studio project templates contain sample code. The code shows how to add custom application code ETW traces that show up in the Visual Studio ETW viewer alongside system traces from Service Fabric. The advantage of this method is that metadata is automatically added to traces, and the Visual Studio Diagnostic Events Viewer is already configured to display them.

For projects created from the **service templates** (stateless or stateful) just search for the `RunAsync` implementation:

1. The call to `ServiceEventSource.Current.ServiceMessage` in the `RunAsync` method shows an example of a custom ETW trace from the application code.
2. In the **ServiceEventSource.cs** file, you will find an overload for the `ServiceEventSource.ServiceMessage` method that should be used for high-frequency events due to performance reasons.

For projects created from the **actor templates** (stateless or stateful):

1. Open the **"ProjectName".cs** file where *ProjectName* is the name you chose for your Visual Studio project.
2. Find the code `ActorEventSource.Current.ActorMessage(this, "Doing Work");` in the *DoWorkAsync* method. This is an example of a custom ETW trace written from application code.
3. In file **ActorEventSource.cs**, you will find an overload for the `ActorEventSource.ActorMessage` method that should be used for high-frequency events due to performance reasons.

After adding custom ETW tracing to your service code, you can build, deploy, and run the application again to see your event(s) in the Diagnostic Events Viewer. If you debug the application with **F5**, the Diagnostic Events Viewer will open automatically.

Next steps

The same tracing code that you added to your application above for local diagnostics will work with tools that you can use to view these events when running your application on an Azure cluster. Check out these articles that discuss the different options for the tools and describe how you can set them up.

- How to collect logs with Azure Diagnostics
- Using ElasticSearch as a Service Fabric application trace store

Monitor and diagnose services in a local machine development setup

11/22/2016 • 3 min to read • Edit on GitHub

Contributors

Mani Ramaswamy • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • PRmerger • carolinacmoravia

Monitoring, detecting, diagnosing, and troubleshooting allow for services to continue with minimal disruption to the user experience. Monitoring and diagnostics are critical in an actual deployed production environment. Adopting a similar model during development of services ensures that the diagnostic pipeline works when you move to a production environment. Service Fabric makes it easy for service developers to implement diagnostics that can seamlessly work across both single-machine local development setups and real-world production cluster setups.

Debugging Service Fabric Java applications

For Java applications, multiple logging frameworks are available. Since `java.util.logging` is the default option with the JRE, it is also used for the code examples in github. The following discussion explains how to configure the `java.util.logging` framework.

Using java.util.logging you can redirect your application logs to memory, output streams, console files, or sockets. For each of these options, there are default handlers already provided in the framework. You can create a `app.properties` file to configure the file handler for your application to redirect all logs to a local file.

The following code snippet contains an example configuration:

```
handlers = java.util.logging.FileHandler

java.util.logging.FileHandler.level = ALL
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.limit = 1024000
java.util.logging.FileHandler.count = 10
java.util.logging.FileHandler.pattern = /tmp/servicefabric/logs/mysfapp%u.%g.log
```

The folder pointed to by the `app.properties` file must exist. After the `app.properties` file is created, you need to also modify your entry point script, `entrypoint.sh` in the `<applicationfolder>/<servicePkg>/Code/` folder to set the property `java.util.logging.config.file` to `app.propertes` file. The entry should look like the following snippet:

```
java -Djava.library.path=$LD_LIBRARY_PATH -Djava.util.logging.config.file=<path to app.properties> -jar <service
name>.jar
```

This configuration results in logs being collected in a rotating fashion at `/tmp/servicefabric/logs/`. The **%u** and **%g** allow for creating more files, with filenames mysfapp0.log, mysfapp1.log, and so on. By default if no handler is explicitly configured, the console handler is registered. One can view the logs in syslog under /var/log/syslog.

For more information, see the code examples in github.

Debugging Service Fabric C# applications

Multiple frameworks are available for tracing CoreCLR applications on Linux. For more information, see GitHub:

logging. Since EventSource is familiar to C# developers,`this article uses EventSource for tracing in CoreCLR samples on Linux.

The first step is to include System.Diagnostics.Tracing so that you can write your logs to memory, output streams, or console files. For logging using EventSource, add the following project to your project.json:

```
    "System.Diagnostics.StackTrace": "4.0.1"
```

You can use a custom EventListener to listen for the service event and then appropriately redirect them to trace files. The following code snippet shows a sample implementation of logging using EventSource and a custom EventListener:

```
 public class ServiceEventSource : EventSource
 {
        public static ServiceEventSource Current = new ServiceEventSource();

        [NonEvent]
        public void Message(string message, params object[] args)
        {
            if (this.IsEnabled())
            {
                var finalMessage = string.Format(message, args);
                this.Message(finalMessage);
            }
        }

        // TBD: Need to add method for sample event.

}
```

```
   internal class ServiceEventListener : EventListener
   {

        protected override void OnEventSourceCreated(EventSource eventSource)
        {
            EnableEvents(eventSource, EventLevel.LogAlways, EventKeywords.All);
        }
        protected override void OnEventWritten(EventWrittenEventArgs eventData)
        {
            using (StreamWriter Out = new StreamWriter( new FileStream("/tmp/MyServiceLog.txt",
FileMode.Append)))
        {
                 // report all event information
          Out.Write(" {0} ",  Write(eventData.Task.ToString(), eventData.EventName,
eventData.EventId.ToString(), eventData.Level,""));
                if (eventData.Message != null)
            Out.WriteLine(eventData.Message, eventData.Payload.ToArray());
            else
        {
                    string[] sargs = eventData.Payload != null ? eventData.Payload.Select(o =>
o.ToString()).ToArray() : null;
                    Out.WriteLine("({0}).", sargs != null ? string.Join(", ", sargs) : "");
        }
           }
        }
    }
```

The preceding snippet outputs the logs to a file in `/tmp/MyServiceLog.txt`. This file name needs to be appropriately updated. In case you want to redirect the logs to console, use the following snippet in your customized EventListener class:

```
public static TextWriter Out = Console.Out;
```

The samples at C# Samples use EventSource and a custom EventListener to log events to a file.

Next steps

The same tracing code added to your application also works with the diagnostics of your application on an Azure cluster. Check out these articles that discuss the different options for the tools and describe how to set them up.

- How to collect logs with Azure Diagnostics

Collect logs by using Azure Diagnostics

11/15/2016 • 7 min to read • Edit on GitHub

Contributors

Todd Abel • Ralph Squillace • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • ShawnJackson

---

When you're running an Azure Service Fabric cluster, it's a good idea to collect the logs from all the nodes in a central location. Having the logs in a central location helps you analyze and troubleshoot issues in your cluster, or issues in the applications and services running in that cluster.

One way to upload and collect logs is to use the Azure Diagnostics extension, which uploads logs to Azure Storage. The logs are not that useful directly in storage. But you can use an external process to read the events from storage and place them in a product such as Log Analytics, Elastic Search, or another log-parsing solution.

Prerequisites

You'll use these tools to perform some of the operations in this document:

- Azure Diagnostics (related to Azure Cloud Services but has good information and examples)
- Azure Resource Manager
- Azure PowerShell
- Azure Resource Manager client
- Azure Resource Manager template

Log sources that you might want to collect

- **Service Fabric logs**: Emitted from the platform to standard Event Tracing for Windows (ETW) and EventSource channels. Logs can be one of several types:
  - Operational events: Logs for operations that the Service Fabric platform performs. Examples include creation of applications and services, node state changes, and upgrade information.
  - Reliable Actors programming model events
  - Reliable Services programming model events
- **Application events**: Events emitted from your service's code and written out by using the EventSource helper class provided in the Visual Studio templates. For more information on how to write logs from your application, see Monitor and diagnose services in a local machine development setup.

Deploy the Diagnostics extension

The first step in collecting logs is to deploy the Diagnostics extension on each of the VMs in the Service Fabric cluster. The Diagnostics extension collects logs on each VM and uploads them to the storage account that you specify. The steps vary a little based on whether you use the Azure portal or Azure Resource Manager. The steps also vary based on whether the deployment is part of cluster creation or is for a cluster that already exists. Let's look at the steps for each scenario.

Deploy the Diagnostics extension as part of cluster creation through the portal

To deploy the Diagnostics extension to the VMs in the cluster as part of cluster creation, you use the Diagnostics settings panel shown in the following image. To enable Reliable Actors or Reliable Services event collection, ensure that Diagnostics is set to **On** (the default setting). After you create the cluster, you can't change these settings by

using the portal.

The Azure support team *requires* support logs to help resolve any support requests that you create. These logs are collected in real time and are stored in one of the storage accounts created in the resource group. The Diagnostics settings configure application-level events. These events include Reliable Actors events, Reliable Services events, and some system-level Service Fabric events to be stored in Azure Storage.

Products such as Elastic Search or your own process can get the events from the storage account. There is currently no way to filter or groom the events that are sent to the table. If you don't implement a process to remove events from the table, the table will continue to grow.

When you're creating a cluster by using the portal, we highly recommend that you download the template *before you click **OK** to create the cluster. For details, refer to Set up a Service Fabric cluster by using an Azure Resource Manager template. You'll need the template to make changes later, because you can't make some changes by using the portal.

You can export templates from the portal by using the following steps. However, these templates can be more difficult to use because they might have null values that are missing required information.

1. Open your resource group.
2. Select **Settings** to display the settings panel.
3. Select **Deployments** to display the deployment history panel.
4. Select a deployment to display the details of the deployment.
5. Select **Export Template** to display the template panel.
6. Select **Save to file** to export a .zip file that contains the template, parameter, and PowerShell files.

After you export the files, you need to make a modification. Edit the parameters.json file and remove the **adminPassword** element. This will cause a prompt for the password when the deployment script is run. When you're running the deployment script, you might have to fix null parameter values.

To use the downloaded template to update a configuration:

1. Extract the contents to a folder on your local computer.
2. Modify the contents to reflect the new configuration.
3. Start PowerShell and change to the folder where you extracted the content.

4. Run **deploy.ps1** and fill in the subscription ID, the resource group name (use the same name to update the configuration), and a unique deployment name.

Deploy the Diagnostics extension as part of cluster creation by using Azure Resource Manager

To create a cluster by using Resource Manager, you need to add the Diagnostics configuration JSON to the full cluster Resource Manager template before you create the cluster. We provide a sample five-VM cluster Resource Manager template with Diagnostics configuration added to it as part of our Resource Manager template samples. You can see it at this location in the Azure Samples gallery: Five-node cluster with Diagnostics Resource Manager template sample.

To see the Diagnostics setting in the Resource Manager template, open the azuredeploy.json file and search for **IaaSDiagnostics**. To create a cluster by using this template, select the **Deploy to Azure** button available at the previous link.

Alternatively, you can download the Resource Manager sample, make changes to it, and create a cluster with the modified template by using the `New-AzureRmResourceGroupDeployment` command in an Azure PowerShell window. See the following code for the parameters that you pass in to the command. For detailed information on how to deploy a resource group by using PowerShell, see the article Deploy a resource group with the Azure Resource Manager template.

```
New-AzureRmResourceGroupDeployment -ResourceGroupName $resourceGroupName -Name $deploymentName -TemplateFile
$pathToARMConfigJsonFile -TemplateParameterFile $pathToParameterFile –Verbose
```

Deploy the Diagnostics extension to an existing cluster

If you have an existing cluster that doesn't have Diagnostics deployed, or if you want to modify an existing configuration, you can add or update it. Modify the Resource Manager template that's used to create the existing cluster or download the template from the portal as described earlier. Modify the template.json file by performing the following tasks.

Add a new storage resource to the template by adding to the resources section.

```
{
  "apiVersion": "2015-05-01-preview",
  "type": "Microsoft.Storage/storageAccounts",
  "name": "[parameters('applicationDiagnosticsStorageAccountName')]",
  "location": "[parameters('computeLocation')]",
  "properties": {
    "accountType": "[parameters('applicationDiagnosticsStorageAccountType')]"
  },
  "tags": {
    "resourceType": "Service Fabric",
    "clusterName": "[parameters('clusterName')]"
  }
},
```

Next, add to the parameters section just after the storage account definitions, between `supportLogStorageAccountName` and `vmNodeType0Name`. Replace the placeholder text *storage account name goes here* with the name of the storage account.

```
    "applicationDiagnosticsStorageAccountType": {
      "type": "string",
      "allowedValues": [
        "Standard_LRS",
        "Standard_GRS"
      ],
      "defaultValue": "Standard_LRS",
      "metadata": {
        "description": "Replication option for the application diagnostics storage account"
      }
    },
    "applicationDiagnosticsStorageAccountName": {
      "type": "string",
      "defaultValue": "storage account name goes here",
      "metadata": {
        "description": "Name for the storage account that contains application diagnostics data from the
cluster"
      }
    },
```

Then, update the `VirtualMachineProfile` section of the template.json file by adding the following code within the extensions array. Be sure to add a comma at the beginning or the end, depending on where it's inserted.

```
{
    "name": "[concat(parameters('vmNodeType0Name'),'_Microsoft.Insights.VMDiagnosticsSettings')]",
    "properties": {
        "type": "IaaSDiagnostics",
        "autoUpgradeMinorVersion": true,
        "protectedSettings": {
        "storageAccountName": "[parameters('applicationDiagnosticsStorageAccountName')]",
        "storageAccountKey": "[listKeys(resourceId('Microsoft.Storage/storageAccounts',
parameters('applicationDiagnosticsStorageAccountName')),'2015-05-01-preview').key1]",
        "storageAccountEndPoint": "https://core.windows.net/"
        },
        "publisher": "Microsoft.Azure.Diagnostics",
        "settings": {
        "WadCfg": {
            "DiagnosticMonitorConfiguration": {
            "overallQuotaInMB": "50000",
            "EtwProviders": {
                "EtwEventSourceProviderConfiguration": [
                {
                    "provider": "Microsoft-ServiceFabric-Actors",
                    "scheduledTransferKeywordFilter": "1",
                    "scheduledTransferPeriod": "PT5M",
                    "DefaultEvents": {
                    "eventDestination": "ServiceFabricReliableActorEventTable"
                    }
                },
                {
                    "provider": "Microsoft-ServiceFabric-Services",
                    "scheduledTransferPeriod": "PT5M",
                    "DefaultEvents": {
                    "eventDestination": "ServiceFabricReliableServiceEventTable"
                    }
                }
                ],
                "EtwManifestProviderConfiguration": [
                {
                    "provider": "cbd93bc2-71e5-4566-b3a7-595d8eeca6e8",
                    "scheduledTransferLogLevelFilter": "Information",
                    "scheduledTransferKeywordFilter": "4611686018427387904",
                    "scheduledTransferPeriod": "PT5M",
                    "DefaultEvents": {
                    "eventDestination": "ServiceFabricSystemEventTable"
                    }
                }
                ]
            }
            }
        },
        "StorageAccount": "[parameters('applicationDiagnosticsStorageAccountName')]"
        },
        "typeHandlerVersion": "1.5"
    }
}
```

After you modify the template.json file as described, republish the Resource Manager template. If the template was exported, running the deploy.ps1 file republishes the template. After you deploy, ensure that **ProvisioningState** is **Succeeded**.

Update Diagnostics to collect and upload logs from new EventSource channels

To update Diagnostics to collect logs from new EventSource channels that represent a new application that you're about to deploy, perform the same steps as in the previous section for setup of Diagnostics for an existing cluster.

Update the `EtwEventSourceProviderConfiguration` section in the template.json file to add entries for the new EventSource channels before you apply the configuration update by using the `New-AzureRmResourceGroupDeployment` PowerShell command. The name of the event source is defined as part of your code in the Visual Studio-generated ServiceEventSource.cs file.

For example, if your event source is named My-Eventsource, add the following code to place the events from My-Eventsource into a table named MyDestinationTableName.

```
        {
            "provider": "My-Eventsource",
            "scheduledTransferPeriod": "PT5M",
            "DefaultEvents": {
            "eventDestination": "MyDestinationTableName"
            }
        }
```

To collect performance counters or event logs, modify the Resource Manager template by using the examples provided in Create a Windows virtual machine with monitoring and diagnostics by using an Azure Resource Manager template. Then, republish the Resource Manager template.

Next steps

To understand in more detail what events you should look for while troubleshooting issues, see the diagnostic events emitted for Reliable Actors and Reliable Services.

Related articles

- Learn how to collect performance counters or logs by using the Diagnostics extension
- Service Fabric solution in Log Analytics

Collect logs by using Azure Diagnostics

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Mani Ramaswamy • Ralph Squillace • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • ShawnJackson • Todd Abel

When you're running an Azure Service Fabric cluster, it's a good idea to collect the logs from all the nodes in a central location. Having the logs in a central location makes it easy to analyze and troubleshoot issues, whether they are in your services, your application, or the cluster itself. One way to upload and collect logs is to use the Azure Diagnostics extension, which uploads logs to Azure Storage. You can read the events from storage and place them in a product such as Elastic Search or another log-parsing solution.

Log sources that you might want to collect

- **Service Fabric logs**: Emitted from the platform via LTTng and uploaded to your storage account. Logs can be operational events or runtime events that the platform emits. These logs are stored in the location that the cluster manifest specifies. (To get the storage account details, search for the tag **AzureTableWinFabETWQueryable** and look for **StoreConnectionString**.)
- **Application events**: Emitted from your service's code. You can use any logging solution that writes text-based log files--for example, LTTng. For more information, see the LTTng documentation on tracing your application.

Deploy the Diagnostics extension

The first step in collecting logs is to deploy the Diagnostics extension on each of the VMs in the Service Fabric cluster. The Diagnostics extension collects logs on each VM and uploads them to the storage account that you specify. The steps vary based on whether you use the Azure portal or Azure Resource Manager.

To deploy the Diagnostics extension to the VMs in the cluster as part of cluster creation, set **Diagnostics** to **On**. After you create the cluster, you can't change this setting by using the portal.

Then, configure Linux Azure Diagnostics (LAD) to collect the files and place them into your storage account. This process is explained as scenario 3 ("Upload your own log files") in the article Using LAD to monitor and diagnose Linux VMs. Following this process gets you access to the traces. You can upload the traces to a visualizer of your choice.

You can also deploy the Diagnostics extension by using Azure Resource Manager. The process is similar for Windows and Linux and is documented for Windows clusters in How to collect logs with Azure Diagnostics.

You can also use Operations Management Suite, as described in Operations Management Suite Log Analytics with Linux.

After you finish this configuration, the LAD agent monitors the specified log files. Whenever a new line is appended to the file, it creates a syslog entry that is sent to the storage that you specified.

Next steps

To understand in more detail what events you should examine while troubleshooting issues, see LTTng documentation and Using LAD.

Use Elasticsearch as a Service Fabric application trace store

11/15/2016 • 12 min to read • Edit on GitHub

Contributors

Karol Zadora-Przylecki • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ralph Squillace • MarkDeichmiller • Ryan Wike

Introduction

This article describes how Azure Service Fabric applications can use **Elasticsearch** and **Kibana** for application trace storage, indexing, and search. Elasticsearch is an open-source, distributed, and scalable real-time search and analytics engine that is well-suited for this task. It can be installed on Windows and Linux virtual machines running in Microsoft Azure. Elasticsearch can efficiently process *structured* traces produced using technologies such as **Event Tracing for Windows (ETW)**.

ETW is used by Service Fabric runtime to source diagnostic information (traces). It is the recommended method for Service Fabric applications to source their diagnostic information, too. Using the same mechanism allows for correlation between runtime-supplied and application-supplied traces and makes troubleshooting easier. Service Fabric project templates in Visual Studio include a logging API (based on the .NET **EventSource** class) that emits ETW traces by default. For a general overview of Service Fabric application tracing using ETW, see Monitoring and diagnosing services in a local machine development setup.

For the traces to show up in Elasticsearch, they need to be captured at the Service Fabric cluster nodes in real time (while the application is running) and sent to an Elasticsearch endpoint. There are two major options for trace capturing:

- **In-process trace capturing**
  The application, or more precisely, service process, is responsible for sending out the diagnostic data to the trace store (Elasticsearch).
- **Out-of-process trace capturing**
  A separate agent is capturing traces from the service process or processes and sending them to the trace store.

Below, we describe how to set up Elasticsearch on Azure, discuss the pros and cons for both capture options, and explain how to configure a Service Fabric service to send data to Elasticsearch.

Set up Elasticsearch on Azure

The most straightforward way to set up the Elasticsearch service on Azure is through **Azure Resource Manager templates**. A comprehensive Quickstart Azure Resource Manager template for Elasticsearch is available from Azure Quickstart templates repository. This template uses separate storage accounts for scale units (groups of nodes). It can also provision separate client and server nodes with different configurations and various numbers of data disks attached.

Here, we use another template, called **ES-MultiNode** from the Azure diagnostic tools repository. This template is easier to use, and it creates an Elasticsearch cluster protected by HTTP basic authentication. Before you proceed, download the repository from GitHub to your machine (by either cloning the repository or downloading a zip file). The ES-MultiNode template is located in the folder with the same name.

Prepare a machine to run Elasticsearch installation scripts

The easiest way to use the ES-MultiNode template is through a provided Azure PowerShell script called `CreateElasticSearchCluster`. To use this script, you need to install PowerShell modules and a tool called **openssl**. The latter is needed for creating an SSH key that can be used to administer your Elasticsearch cluster remotely.

`CreateElasticSearchCluster` script is designed for ease of use with the ES-MultiNode template from a Windows machine. It is possible to use the template on a non-Windows machine, but that scenario is beyond the scope of this article.

1. If you haven't installed them already, install **Azure PowerShell modules**. When prompted, click **Run**, then **Install**. Azure PowerShell 1.3 or newer is required.
2. The **openssl** tool is included in the distribution of **Git for Windows**. If you have not done so already, install Git for Windows now. (The default installation options are OK.)
3. Assuming that Git has been installed but not included in the system path, open a Microsoft Azure PowerShell window and run the following commands:

   ```
   $ENV:PATH += ";<Git installation folder>\usr\bin"
   $ENV:OPENSSL_CONF = "<Git installation folder>\usr\ssl\openssl.cnf"
   ```

   Replace the `<Git installation folder>` with the Git location on your machine; the default is **"C:\Program Files\Git"**. Note the semicolon character at the beginning of the first path.

4. Ensure that you are logged on to Azure (via `Add-AzureRmAccount` cmdlet) and that you have selected the subscription that should be used to create your Elastic Search cluster. You can verify that correct subscription is selected using `Get-AzureRmContext` and `Get-AzureRmSubscription` cmdlets.
5. If you haven't done so already, change the current directory to the ES-MultiNode folder.

Run the CreateElasticSearchCluster script

Before you run the script, open the `azuredeploy-parameters.json` file and verify or provide values for the script parameters. The following parameters are provided:

PARAMETER NAME	DESCRIPTION
dnsNameForLoadBalancerIP	The name that is used to create the publicly visible DNS name for the Elastic Search cluster (by appending the Azure region domain to the provided name). For example, if this parameter value is "myBigCluster" and the chosen Azure region is West US, the resulting DNS name for the cluster is myBigCluster.westus.cloudapp.azure.com. This name also serves as a root name for many artifacts associated with the Elastic Search cluster, such as data node names.
adminUsername	The name of the administrator account for managing the Elastic Search cluster (corresponding SSH keys are generated automatically).
dataNodeCount	The number of nodes in the Elastic Search cluster. The current version of the script does not distinguish between data and query nodes; all nodes play both roles. Defaults to 3 nodes.
dataDiskSize	The size of data disks (in GB) that is allocated for each data node. Each node receives 4 data disks, exclusively dedicated to Elastic Search service.

PARAMETER NAME	DESCRIPTION
region	The name of Azure region where the Elastic Search cluster should be located.
esUserName	The user name of the user that is configured to have access to ES cluster (subject to HTTP basic authentication). The password is not part of parameters file and must be provided when `CreateElasticSearchCluster` script is invoked.
vmSizeDataNodes	The Azure virtual machine size for Elastic Search cluster nodes. Defaults to Standard_D2.

Now you are ready to run the script. Issue the following command:

```
CreateElasticSearchCluster -ResourceGroupName <es-group-name> -Region <azure-region> -EsPassword <es-password>
```

where

SCRIPT PARAMETER NAME	DESCRIPTION
`<es-group-name>`	The name of the Azure resource group that will contain all Elastic Search cluster resources.
`<azure-region>`	The name of the Azure region where the Elastic Search cluster should be created.
`<es-password>`	The password for the Elastic Search user.

> **NOTE**
> If you get a NullReferenceException from the Test-AzureResourceGroup cmdlet, you have forgotten to log on to Azure (`Add-AzureRmAccount`).

If you get an error from running the script and you determine that the error was caused by a wrong template parameter value, correct the parameter file and run the script again with a different resource group name. You can also reuse the same resource group name and have the script clean up the old one by adding the `-RemoveExistingResourceGroup` parameter to the script invocation.

Result of running the CreateElasticSearchCluster script

After you run the `CreateElasticSearchCluster` script, the following main artifacts will be created. For this example we assume that you have used "myBigCluster" as the value of the `dnsNameForLoadBalancerIP` parameter and that the region where you created the cluster is West US.

ARTIFACT	NAME, LOCATION, AND REMARKS
SSH key for remote administration	myBigCluster.key file (in the directory from which the CreateElasticSearchCluster was run). This key file can be used to connect to the admin node and (through the admin node) to data nodes in the cluster.

ARTIFACT	NAME, LOCATION, AND REMARKS
Admin node	myBigCluster-admin.westus.cloudapp.azure.com A dedicated VM for remote Elasticsearch cluster administration--the only one that allows external SSH connections. It runs on the same virtual network as all the Elasticsearch cluster nodes, but it does not run any Elasticsearch services.
Data nodes	myBigCluster1 ... myBigCluster*N* Data nodes that are running Elasticsearch and Kibana services. You can connect via SSH to each node, but only via the admin node.
Elasticsearch cluster	http://myBigCluster.westus.cloudapp.azure.com/es/ The primary endpoint for the Elasticsearch cluster (note the /es suffix). It is protected by basic HTTP authentication (the credentials were the specified esUserName/esPassword parameters of the ES-MultiNode template). The cluster has also the head plug-in installed (http://myBigCluster.westus.cloudapp.azure.com/es/_plugin/head) for basic cluster administration.
Kibana service	http://myBigCluster.westus.cloudapp.azure.com The Kibana service is set up to show data from the created Elasticsearch cluster. It is protected by the same authentication credentials as the cluster itself.

In-process versus out-of-process trace capturing

In the introduction, we mentioned two fundamental ways of collecting diagnostic data: in-process and out-of-process. Each has strengths and weaknesses.

Advantages of the **in-process trace capturing** include:

1. *Easy configuration and deployment*
   - The configuration of diagnostic data collection is just part of the application configuration. It is easy to always keep it "in sync" with the rest of the application.
   - Per-application or per-service configuration is easily achievable.
   - Out-of-process trace capturing usually requires a separate deployment and configuration of the diagnostic agent, which is an extra administrative task and a potential source of errors. The particular agent technology often allows only one instance of the agent per virtual machine (node). This means that configuration for the collection of the diagnostic configuration is shared among all applications and services running on that node.
2. *Flexibility*
   - The application can send the data wherever it needs to go, as long as there is a client library that supports the targeted data storage system. New sinks can be added as desired.
   - Complex capture, filtering, and data-aggregation rules can be implemented.
   - An out-of-process trace capturing is often limited by the data sinks that the agent supports. Some agents are extensible.
3. *Access to internal application data and context*

- The diagnostic subsystem running inside the application/service process can easily augment the traces with contextual information.
- In the out-of-process approach, the data must be sent to an agent via some inter-process communication mechanism, such as Event Tracing for Windows. This mechanism could impose additional limitations.

Advantages of the **out-of-process trace capturing** include:

1. *The ability to monitor the application and collect crash dumps*

   - In-process trace capturing may be unsuccessful if the application fails to start or crashes. An independent agent has a much better chance of capturing crucial troubleshooting information.

2. *Maturity, robustness, and proven performance*

   - An agent developed by a platform vendor (such as a Microsoft Azure Diagnostics agent) has been subject to rigorous testing and battle-hardening.
   - With in-process trace capturing, care must be taken to ensure that the activity of sending diagnostic data from an application process does not interfere with the application's main tasks or introduce timing or performance problems. An independently running agent is less prone to these issues and is specifically designed to limit its impact on the system.

It is possible to combine and benefit from both approaches. Indeed, it might be the best solution for many applications.

Here, we use the **Microsoft.Diagnostic.Listeners library** and the in-process trace capturing to send data from a Service Fabric application to an Elasticsearch cluster.

Use the Listeners library to send diagnostic data to Elasticsearch

The Microsoft.Diagnostic.Listeners library is part of PartyCluster sample Service Fabric application. To use it:

1. Download the PartyCluster sample from GitHub.
2. Copy the Microsoft.Diagnostics.Listeners and Microsoft.Diagnostics.Listeners.Fabric projects (whole folders) from the PartyCluster sample directory to the solution folder of the application that is supposed to send the data to Elasticsearch.
3. Open the target solution, right-click the solution node in the Solution Explorer, and choose **Add Existing Project**. Add the Microsoft.Diagnostics.Listeners project to the solution. Repeat the same for the Microsoft.Diagnostics.Listeners.Fabric project.
4. Add a project reference from your service project(s) to the two added projects. (Each service that is supposed to send data to Elasticsearch should reference Microsoft.Diagnostics.EventListeners and Microsoft.Diagnostics.EventListeners.Fabric).

Service Fabric General Availability release and Microsoft.Diagnostics.Tracing Nuget package

Applications built with Service Fabric General Availability release (2.0.135, released March 31, 2016) target **.NET Framework 4.5.2**. This version is the highest version of the .NET Framework supported by Azure at the time of the GA release. Unfortunately, this version of the framework lacks certain EventListener APIs that the Microsoft.Diagnostics.Listeners library needs. Because EventSource (the component that forms the basis of logging APIs in Fabric applications) and EventListener are tightly coupled, every project that uses the Microsoft.Diagnostics.Listeners library must use an alternative implementation of EventSource. This implementation is provided by the **Microsoft.Diagnostics.Tracing Nuget package**, authored by Microsoft. The package is fully backward-compatible with EventSource included in the framework, so no code changes should be necessary other than referenced namespace changes.

To start using the Microsoft.Diagnostics.Tracing implementation of the EventSource class, follow these steps for each service project that needs to send data to Elasticsearch:

1. Right-click on the service project and choose **Manage Nuget Packages**.
2. Switch to the nuget.org package source (if it is not already selected) and search for "**Microsoft.Diagnostics.Tracing**".
3. Install the `Microsoft.Diagnostics.Tracing.EventSource` package (and its dependencies).
4. Open the **ServiceEventSource.cs** or **ActorEventSource.cs** file in your service project and replace the `using System.Diagnostics.Tracing` directive on top of the file with the `using Microsoft.Diagnostics.Tracing` directive.

These steps will not be necessary once the **.NET Framework 4.6** is supported by Microsoft Azure.

Elasticsearch listener instantiation and configuration

The final step for sending diagnostic data to Elasticsearch is to create an instance of `ElasticSearchListener` and configure it with Elasticsearch connection data. The listener automatically captures all events raised via EventSource classes defined in the service project. It needs to be alive during the lifetime of the service, so the best place to create it is in the service initialization code. Here is how the initialization code for a stateless service could look after the necessary changes (additions pointed out in comments starting with `****`):

```
using System;
using System.Diagnostics;
using System.Fabric;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.ServiceFabric.Services.Runtime;

// **** Add the following directives
using Microsoft.Diagnostics.EventListeners;
using Microsoft.Diagnostics.EventListeners.Fabric;

namespace Stateless1
{
    internal static class Program
    {
        /// <summary>
        /// This is the entry point of the service host process.
        /// </summary>
        private static void Main()
        {
            try
            {
                // **** Instantiate ElasticSearchListener
                var configProvider = new FabricConfigurationProvider("ElasticSearchEventListener");
                ElasticSearchListener esListener = null;
                if (configProvider.HasConfiguration)
                {
                    esListener = new ElasticSearchListener(configProvider);
                }

                // The ServiceManifest.XML file defines one or more service type names.
                // Registering a service maps a service type name to a .NET type.
                // When Service Fabric creates an instance of this service type,
                // an instance of the class is created in this host process.

                ServiceRuntime.RegisterServiceAsync("Stateless1Type",
                    context => new Stateless1(context)).GetAwaiter().GetResult();

                ServiceEventSource.Current.ServiceTypeRegistered(Process.GetCurrentProcess().Id,
typeof(Stateless1).Name);

                // Prevents this host process from terminating so services keep running.
                Thread.Sleep(Timeout.Infinite);

                // **** Ensure that the ElasticSearchListner instance is not garbage-collected prematurely
                GC.KeepAlive(esListener);
            }
            catch (Exception e)
            {
                ServiceEventSource.Current.ServiceHostInitializationFailed(e);
                throw;
            }
        }
    }
}
```

Elasticsearch connection data should be put in a separate section in the service configuration file (**PackageRoot\Config\Settings.xml**). The name of the section must correspond to the value passed to the `FabricConfigurationProvider` constructor, for example:

```
<Section Name="ElasticSearchEventListener">
  <Parameter Name="serviceUri" Value="http://myBigCluster.westus.cloudapp.azure.com/es/" />
  <Parameter Name="userName" Value="(ES user name)" />
  <Parameter Name="password" Value="(ES user password)" />
  <Parameter Name="indexNamePrefix" Value="myapp" />
</Section>
```

The values of `serviceUri`, `userName` and `password` parameters correspond to the Elasticsearch cluster endpoint address, Elasticsearch user name, and password, respectively. `indexNamePrefix` is the prefix for Elasticsearch indexes; the Microsoft.Diagnostics.Listeners library creates a new index for its data daily.

Verification

That's it! Now, whenever the service is run, it starts sending traces to the Elasticsearch service specified in the configuration. You can verify this by opening the Kibana UI associated with the target Elasticsearch instance. In our example, the page address is http://myBigCluster.westus.cloudapp.azure.com/. Check that indexes with the name prefix chosen for the `ElasticSearchListener` instance have indeed been created and populated with data.

Next steps

- Learn more about diagnosing and monitoring a Service Fabric service

Diagnostics and performance monitoring for Reliable Actors

11/15/2016 • 7 min to read • Edit on GitHub

Contributors

Abhishek Ram • Theano Petersen • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Vaclav Turecek • Ryan Wike • ShawnJackson • Matt Trower • jessebenson

The Reliable Actors runtime emits EventSource events and performance counters. These provide insights into how the runtime is operating and help with troubleshooting and performance monitoring.

EventSource events

The EventSource provider name for the Reliable Actors runtime is "Microsoft-ServiceFabric-Actors". Events from this event source appear in the Diagnostics Events window when the actor application is being debugged in Visual Studio.

Examples of tools and technologies that help in collecting and/or viewing EventSource events are PerfView, Azure Diagnostics, Semantic Logging, and the Microsoft TraceEvent Library.

Keywords

All events that belong to the Reliable Actors EventSource are associated with one or more keywords. This enables filtering of events that are collected. The following keyword bits are defined.

BIT	DESCRIPTION
0x1	Set of important events that summarize the operation of the Fabric Actors runtime.
0x2	Set of events that describe actor method calls. For more information, see the introductory topic on actors.
0x4	Set of events related to actor state. For more information, see the topic on actor state management.
0x8	Set of events related to turn-based concurrency in the actor. For more information, see the topic on concurrency.

Performance counters

The Reliable Actors runtime defines the following performance counter categories.

CATEGORY	DESCRIPTION
Service Fabric Actor	Counters specific to Azure Service Fabric actors, e.g. time taken to save actor state
Service Fabric Actor Method	Counters specific to methods implemented by Service Fabric actors, e.g. how often an actor method is invoked

Each of the above categories has one or more counters.

The Windows Performance Monitor application that is available by default in the Windows operating system can be used to collect and view performance counter data. Azure Diagnostics is another option for collecting performance counter data and uploading it to Azure tables.

Performance counter instance names

A cluster that has a large number of actor services or actor service partitions will have a large number of actor performance counter instances. The performance counter instance names can help in identifying the specific partition and actor method (if applicable) that the performance counter instance is associated with.

Service Fabric Actor category

For the category `Service Fabric Actor`, the counter instance names are in the following format:

```
ServiceFabricPartitionID_ActorsRuntimeInternalID
```

ServiceFabricPartitionID is the string representation of the Service Fabric partition ID that the performance counter instance is associated with. The partition ID is a GUID, and its string representation is generated through the `Guid.ToString` method with format specifier "D".

ActorRuntimeInternalID is the string representation of a 64-bit integer that is generated by the Fabric Actors runtime for its internal use. This is included in the performance counter instance name to ensure its uniqueness and avoid conflict with other performance counter instance names. Users should not try to interpret this portion of the performance counter instance name.

The following is an example of a counter instance name for a counter that belongs to the `Service Fabric Actor` category:

```
2740af29-78aa-44bc-a20b-7e60fb783264_635650083799324046
```

In the example above, `2740af29-78aa-44bc-a20b-7e60fb783264` is the string representation of the Service Fabric partition ID, and `635650083799324046` is the 64-bit ID that is generated for the runtime's internal use.

Service Fabric Actor Method category

For the category `Service Fabric Actor Method`, the counter instance names are in the following format:

```
MethodName_ActorsRuntimeMethodId_ServiceFabricPartitionID_ActorsRuntimeInternalID
```

MethodName is the name of the actor method that the performance counter instance is associated with. The format of the method name is determined based on some logic in the Fabric Actors runtime that balances the readability of the name with constraints on the maximum length of the performance counter instance names on Windows.

ActorsRuntimeMethodId is the string representation of a 32-bit integer that is generated by the Fabric Actors runtime for its internal use. This is included in the performance counter instance name to ensure its uniqueness and avoid conflict with other performance counter instance names. Users should not try to interpret this portion of the performance counter instance name.

ServiceFabricPartitionID is the string representation of the Service Fabric partition ID that the performance counter instance is associated with. The partition ID is a GUID, and its string representation is generated through the `Guid.ToString` method with format specifier "D".

ActorRuntimeInternalID is the string representation of a 64-bit integer that is generated by the Fabric Actors runtime for its internal use. This is included in the performance counter instance name to ensure its uniqueness and avoid conflict with other performance counter instance names. Users should not try to interpret this portion of the performance counter instance name.

The following is an example of a counter instance name for a counter that belongs to the `Service Fabric Actor Method` category:

```
ivoicemailboxactor.leavemessageasync_2_89383d32-e57e-4a9b-a6ad-57c6792aa521_635650083804480486
```

In the example above, `ivoicemailboxactor.leavemessageasync` is the method name, `2` is the 32-bit ID generated for the runtime's internal use, `89383d32-e57e-4a9b-a6ad-57c6792aa521` is the string representation of the Service Fabric partition ID, and `635650083804480486` is the 64-bit ID generated for the runtime's internal use.

List of events and performance counters

Actor method events and performance counters

The Reliable Actors runtime emits the following events related to actor methods.

EVENT NAME	EVENT ID	LEVEL	KEYWORD	DESCRIPTION
ActorMethodStart	7	Verbose	0x2	Actors runtime is about to invoke an actor method.
ActorMethodStop	8	Verbose	0x2	An actor method has finished executing. That is, the runtime's asynchronous call to the actor method has returned, and the task returned by the actor method has finished.
ActorMethodThrewException	9	Warning	0x3	An exception was thrown during the execution of an actor method, either during the runtime's asynchronous call to the actor method or during the execution of the task returned by the actor method. This event indicates some sort of failure in the actor code that needs investigation.

The Reliable Actors runtime publishes the following performance counters related to the execution of actor methods.

CATEGORY NAME	COUNTER NAME	DESCRIPTION
Service Fabric Actor Method	Invocations/Sec	Number of times that the actor service method is invoked per second
Service Fabric Actor Method	Average milliseconds per invocation	Time taken to execute the actor service method in milliseconds
Service Fabric Actor Method	Exceptions thrown/Sec	Number of times that the actor service method threw an exception per second

Concurrency events and performance counters

The Reliable Actors runtime emits the following events related to concurrency.

EVENT NAME	EVENT ID	LEVEL	KEYWORD	DESCRIPTION
ActorMethodCallsWaitingForLock	12	Verbose	0x8	This event is written at the start of each new turn in an actor. It contains the number of pending actor calls that are waiting to acquire the per-actor lock that enforces turn-based concurrency.

The Reliable Actors runtime publishes the following performance counters related to concurrency.

CATEGORY NAME	COUNTER NAME	DESCRIPTION
Service Fabric Actor	# of actor calls waiting for actor lock	Number of pending actor calls waiting to acquire the per-actor lock that enforces turn-based concurrency
Service Fabric Actor	Average milliseconds per lock wait	Time taken (in milliseconds) to acquire the per-actor lock that enforces turn-based concurrency
Service Fabric Actor	Average milliseconds actor lock held	Time (in milliseconds) for which the per-actor lock is held

Actor state management events and performance counters

The Reliable Actors runtime emits the following events related to actor state management.

EVENT NAME	EVENT ID	LEVEL	KEYWORD	DESCRIPTION
ActorSaveStateStart	10	Verbose	0x4	Actors runtime is about to save the actor state.
ActorSaveStateStop	11	Verbose	0x4	Actors runtime has finished saving the actor state.

The Reliable Actors runtime publishes the following performance counters related to actor state management.

CATEGORY NAME	COUNTER NAME	DESCRIPTION
Service Fabric Actor	Average milliseconds per save state operation	Time taken to save actor state in milliseconds
Service Fabric Actor	Average milliseconds per load state operation	Time taken to load actor state in milliseconds

Events related to actor replicas

The Reliable Actors runtime emits the following events related to actor replicas.

EVENT NAME	EVENT ID	LEVEL	KEYWORD	DESCRIPTION
ReplicaChangeRoleToPrimary	1	Informational	0x1	Actor replica changed role to Primary. This implies that the actors for this partition will be created inside this replica.
ReplicaChangeRoleFromPrimary	2	Informational	0x1	Actor replica changed role to non-Primary. This implies that the actors for this partition will no longer be created inside this replica. No new requests will be delivered to actors already created within this replica. The actors will be destroyed after any in-progress requests are completed.

Actor activation and deactivation events and performance counters

The Reliable Actors runtime emits the following events related to actor activation and deactivation.

EVENT NAME	EVENT ID	LEVEL	KEYWORD	DESCRIPTION
ActorActivated	5	Informational	0x1	An actor has been activated.
ActorDeactivated	6	Informational	0x1	An actor has been deactivated.

The Reliable Actors runtime publishes the following performance counters related to actor activation and deactivation.

CATEGORY NAME	COUNTER NAME	DESCRIPTION
Service Fabric Actor	Average OnActivateAsync milliseconds	Time taken to execute OnActivateAsync method in milliseconds

Actor request processing performance counters

When a client invokes a method via an actor proxy object, it results in a request message being sent over the network to the actor service. The service processes the request message and sends a response back to the client. The Reliable Actors runtime publishes the following performance counters related to actor request processing.

CATEGORY NAME	COUNTER NAME	DESCRIPTION
Service Fabric Actor	# of outstanding requests	Number of requests being processed in the service

CATEGORY NAME	COUNTER NAME	DESCRIPTION
Service Fabric Actor	Average milliseconds per request	Time taken (in milliseconds) by the service to process a request
Service Fabric Actor	Average milliseconds for request deserialization	Time taken (in milliseconds) to deserialize actor request message when it is received at the service
Service Fabric Actor	Average milliseconds for response serialization	Time taken (in milliseconds) to serialize the actor response message at the service before the response is sent to the client

Next steps

- How Reliable Actors use the Service Fabric platform
- Actor API reference documentation
- Sample code

Diagnostic functionality for Stateful Reliable Services

11/15/2016 • 1 min to read • Edit on GitHub

Contributors

AlanWarwick • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Wike • katiecumming • Matt Trower

The Stateful Reliable Services StatefulServiceBase class emits EventSource events that can be used to debug the service, provide insights into how the runtime is operating, and help with troubleshooting.

EventSource events

The EventSource name for the Stateful Reliable Services StatefulServiceBase class is "Microsoft-ServiceFabric-Services". Events from this event source appear in the Diagnostics Events window when the service is being debugged in Visual Studio.

Examples of tools and technologies that help in collecting and/or viewing EventSource events are PerfView, Microsoft Azure Diagnostics, and the Microsoft TraceEvent Library.

Events

EVENT NAME	EVENT ID	LEVEL	EVENT DESCRIPTION
StatefulRunAsyncInvocation	1	Informational	Emitted when service RunAsync task is started
StatefulRunAsyncCancellation	2	Informational	Emitted when service RunAsync task is cancelled
StatefulRunAsyncCompletion	3	Informational	Emitted when service RunAsync task is completed
StatefulRunAsyncSlowCancellation	4	Warning	Emitted when service RunAsync task takes too long to complete cancellation
StatefulRunAsyncFailure	5	Error	Emitted when service RunAsync task throws an exception

Interpret events

StatefulRunAsyncInvocation, StatefulRunAsyncCompletion, and StatefulRunAsyncCancellation events are useful to the service writer to understand the lifecycle of a service--as well as the timing for when a service is started, cancelled, or completed. This can be useful when debugging service issues or understanding the service lifecycle.

Service writers should pay close attention to StatefulRunAsyncSlowCancellation and StatefulRunAsyncFailure events because they indicate issues with the service.

StatefulRunAsyncFailure is emitted whenever the service RunAsync() task throws an exception. Typically, an exception thrown indicates an error or bug in the service. Additionally, the exception causes the service to fail, so it

is moved to a different node. This can be an expensive operation and can delay incoming requests while the service is moved. Service writers should determine the cause of the exception and, if possible, mitigate it.

StatefulRunAsyncSlowCancellation is emitted whenever a cancellation request for the RunAsync task takes longer than four seconds. When a service takes too long to complete cancellation, it impacts the ability for the service to be quickly restarted on another node. This may impact the overall availability of the service.

Troubleshoot your local development cluster setup

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Sean McKenna • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • katiecumming • Andy De George

If you run into an issue while interacting with your local Azure Service Fabric development cluster, review the following suggestions for potential solutions.

Cluster setup failures

Cannot clean up Service Fabric logs

Problem

While running the DevClusterSetup script, you see an error like this:

```
Cannot clean up C:\SfDevCluster\Log fully as references are likely being held to items in it. Please remove those
and run this script again.
At line:1 char:1 + .\DevClusterSetup.ps1
+ ~~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo : NotSpecified: (:) [Write-Error], WriteErrorException
+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException,DevClusterSetup.ps1
```

Solution

Close the current PowerShell window and open a new PowerShell window as an administrator. You should now be able to successfully run the script.

Cluster connection failures

Service Fabric PowerShell cmdlets are not recognized in Azure PowerShell

Problem

If you try to run any of the Service Fabric PowerShell cmdlets, such as `Connect-ServiceFabricCluster` in an Azure PowerShell window, it fails, saying that the cmdlet is not recognized. The reason for this is that Azure PowerShell uses the 32-bit version of Windows PowerShell (even on 64-bit OS versions), whereas the Service Fabric cmdlets only work in 64-bit environments.

Solution

Always run Service Fabric cmdlets directly from Windows PowerShell.

> **NOTE**
>
> The latest version of Azure PowerShell does not create a special shortcut, so this should no longer occur.

Type Initialization exception

Problem

When you are connecting to the cluster in PowerShell, you see the error TypeInitializationException for System.Fabric.Common.AppTrace.

Solution

Your path variable was not correctly set during installation. Please sign out of Windows and sign back in. This will fully refresh your path.

Cluster connection fails with "Object is closed"

Problem

A call to Connect-ServiceFabricCluster fails with an error like this:

```
Connect-ServiceFabricCluster : The object is closed.
At line:1 char:1
+ Connect-ServiceFabricCluster
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo : InvalidOperation: (:) [Connect-ServiceFabricCluster], FabricObjectClosedException
+ FullyQualifiedErrorId : CreateClusterConnectionErrorId,Microsoft.ServiceFabric.Powershell.ConnectCluster
```

Solution

Close the current PowerShell window and open a new PowerShell window as an administrator. You should now be able to successfully connect.

Fabric Connection Denied exception

Problem

When debugging from Visual Studio, you get a FabricConnectionDeniedException error.

Solution

This error usually occurs when you try to try to start a service host process manually, rather than allowing the Service Fabric runtime to start it for you.

Ensure that you do not have any service projects set as startup projects in your solution. Only Service Fabric application projects should be set as startup projects.

> **TIP**
>
> If, following setup, your local cluster begins to behave abnormally, you can reset it using the local cluster manager system tray application. This will remove the existing cluster and set up a new one. Please note that all deployed applications and associated data will be removed.

Next steps

- Understand and troubleshoot your cluster with system health reports
- Visualize your cluster with Service Fabric Explorer

Troubleshoot common issues when you deploy services on Azure Service Fabric

11/15/2016 • 1 min to read • Edit on GitHub

Contributors

Matt Trower • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Todd Abel • Matt Barry

When you're running services on your developer computer, it is easy to use Visual Studio's debugging tools. For remote clusters, health reports are always a good place to start. The easiest ways to access these reports are through PowerShell or SFX. This article assumes that you are debugging a remote cluster and have a basic understanding of how to use either of these tools.

Application crash

The "Partition is below target replica or instance count" report is a good indication that your service is crashing. To find out where your service is crashing takes a little more investigation. When your service is running at scale, your best friend will be a set of well-thought-out traces. We suggest that you try Azure Diagnostics for collecting those traces and using a solution such as Elastic Search for viewing and searching the traces.

UNHEALTHY EVALUATIONS

Kind	Health State	Description
Partitions	Error	Unhealthy partitions: 100% (1/1), MaxPercentUnhealthyPartitionsPerService=0%.
Partition	Error	Unhealthy partition: PartitionId='d62ed901-95f8-40c3-b011-26955768d14f', AggregatedHealthState='Error'.
Event	Error	Error event: SourceId='System.FM', Property='State'. Partition is below target replica or instance count.

During service or actor initialization

Any exceptions before the service type is initialized will cause the process to crash. For these types of crashes, the application event log will show the error from your service. These are the most common exceptions to see before the service is initialized.

System.IO.FileNotFoundException

This error is often due to missing assembly dependencies. Check the CopyLocal property in Visual Studio or the global assembly cache for the node.

System.Runtime.InteropServices.COMException at System.Fabric.Interop.NativeRuntime+IFabricRuntime.RegisterStatefulServiceFactory(IntPtr, IFabricStatefulServiceFactory)

This indicates that the registered service type name does not match the service manifest.

Azure Diagnostics can be configured to upload the application event log for all your nodes automatically.

RunAsync() or OnActivateAsync()

If the crash happens during the initialization or running of your registered service type or actor, the exception will be caught by Azure Service Fabric. You can view these from the EventSource providers detailed in the "Next steps" section.

Next steps

Learn more about existing diagnostics provided by Service Fabric:

- Reliable Actors diagnostics
- Reliable Services diagnostics

Partition Service Fabric reliable services

11/15/2016 • 15 min to read • Edit on GitHub

Contributors

Mark Fussell • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • unknown • katiecumming • Aprameya Rao

This article provides an introduction to the basic concepts of partitioning Azure Service Fabric reliable services. The source code used in the article is also available on GitHub.

Partitioning

Partitioning is not unique to Service Fabric. In fact, it is a core pattern of building scalable services. In a broader sense, we can think about partitioning as a concept of dividing state (data) and compute into smaller accessible units to improve scalability and performance. A well-known form of partitioning is data partitioning, also known as sharding.

Partition Service Fabric stateless services

For stateless services, you can think about a partition being a logical unit that contains one or more instances of a service. Figure 1 shows a stateless service with five instances distributed across a cluster using one partition.

There are really two types of stateless service solutions. The first one is a service that persists its state externally, for example in an Azure SQL database (like a website that stores the session information and data). The second one is computation-only services (like a calculator or image thumbnailing) that do not manage any persistent state.

In either case, partitioning a stateless service is a very rare scenario--scalability and availability are normally achieved by adding more instances. The only time you want to consider multiple partitions for stateless service instances is when you need to meet special routing requests.

As an example, consider a case where users with IDs in a certain range should only be served by a particular service instance. Another example of when you could partition a stateless service is when you have a truly partitioned backend (e.g. a sharded SQL database) and you want to control which service instance should write to the database shard--or perform other preparation work within the stateless service that requires the same

partitioning information as is used in the backend. Those types of scenarios can also be solved in different ways and do not necessarily require service partitioning.

The remainder of this walkthrough focuses on stateful services.

Partition Service Fabric stateful services

Service Fabric makes it easy to develop scalable stateful services by offering a first-class way to partition state (data). Conceptually, you can think about a partition of a stateful service as a scale unit that is highly reliable through replicas that are distributed and balanced across the nodes in a cluster.

Partitioning in the context of Service Fabric stateful services refers to the process of determining that a particular service partition is responsible for a portion of the complete state of the service. (As mentioned before, a partition is a set of replicas). A great thing about Service Fabric is that it places the partitions on different nodes. This allows them to grow to a node's resource limit. As the data needs grow, partitions grow, and Service Fabric rebalances partitions across nodes. This ensures the continued efficient use of hardware resources.

To give you an example, say you start with a 5-node cluster and a service that is configured to have 10 partitions and a target of three replicas. In this case, Service Fabric would balance and distribute the replicas across the cluster--and you would end up with two primary replicas per node. If you now need to scale out the cluster to 10 nodes, Service Fabric would rebalance the primary replicas across all 10 nodes. Likewise, if you scaled back to 5 nodes, Service Fabric would rebalance all the replicas across the 5 nodes.

Figure 2 shows the distribution of 10 partitions before and after scaling the cluster.

As a result, the scale-out is achieved since requests from clients are distributed across computers, overall performance of the application is improved, and contention on access to chunks of data is reduced.

Plan for partitioning

Before implementing a service, you should always consider the partitioning strategy that is required to scale out. There are different ways, but all of them focus on what the application needs to achieve. For the context of this article, let's consider some of the more important aspects.

A good approach is to think about the structure of the state that needs to be partitioned, as the first step.

Let's take a simple example. If you were to build a service for a countywide poll, you could create a partition for each city in the county. Then, you could store the votes for every person in the city in the partition that corresponds to that city. Figure 3 illustrates a set of people and the city in which they reside.

Data:

UserID	FirstName	LastName	City	Vote
bscholl	Boris	Scholl	Kirkland	Y
chackdan	Chacko	Daniel	Redmond	N
jeffreyr	Jeffrey	Richter	Redmond	Y
kunalds	Kunal	Deep Singh	Seattle	Y
mfussell	Mark	Fussell	Sammamish	N
masnider	Matt	Snider	Seattle	Y
subramar	Mani	Ramaswamy	Redmond	Y
seanmck	Sean	McKenna	Seattle	Y
vturecek	Vaclav	Turecek	Seattle	N

As the population of cities varies widely, you may end up with some partitions that contain a lot of data (e.g. Seattle) and other partitions with very little state (e.g. Kirkland). So what is the impact of having partitions with uneven amounts of state?

If you think about the example again, you can easily see that the partition that holds the votes for Seattle will get more traffic than the Kirkland one. By default, Service Fabric makes sure that there is about the same number of primary and secondary replicas on each node. So you may end up with nodes that hold replicas that serve more traffic and others that serve less traffic. You would preferably want to avoid hot and cold spots like this in a cluster.

In order to avoid this, you should do two things, from a partitioning point of view:

- Try to partition the state so that it is evenly distributed across all partitions.
- Report load from each of the replicas for the service. (For information on how, check out this article on Metrics and Load). Service Fabric provides the capability to report load consumed by services, such as amount of memory or number of records. Based on the metrics reported, Service Fabric detects that some partitions are serving higher loads than others and rebalances the cluster by moving replicas to more suitable nodes, so that overall no node is overloaded.

Sometimes, you cannot know how much data will be in a given partition. So a general recommendation is to do both--first, by adopting a partitioning strategy that spreads the data evenly across the partitions and second, by reporting load. The first method prevents situations described in the voting example, while the second helps smooth out temporary differences in access or load over time.

Another aspect of partition planning is to choose the correct number of partitions to begin with. From a Service Fabric perspective, there is nothing that prevents you from starting out with a higher number of partitions than anticipated for your scenario. In fact, assuming the maximum number of partitions is a valid approach.

In rare cases, you may end up needing more partitions than you have initially chosen. As you cannot change the partition count after the fact, you would need to apply some advanced partition approaches, such as creating a new service instance of the same service type. You would also need to implement some client-side logic that routes the requests to the correct service instance, based on client-side knowledge that your client code must maintain.

Another consideration for partitioning planning is the available computer resources. As the state needs to be

accessed and stored, you are bound to follow:

- Network bandwidth limits
- System memory limits
- Disk storage limits

So what happens if you run into resource constraints in a running cluster? The answer is that you can simply scale out the cluster to accommodate the new requirements.

The capacity planning guide offers guidance for how to determine how many nodes your cluster needs.

Get started with partitioning

This section describes how to get started with partitioning your service.

Service Fabric offers a choice of three partition schemes:

- Ranged partitioning (otherwise known as UniformInt64Partition).
- Named partitioning. Applications using this model usually have data that can be bucketed, within a bounded set. Some common examples of data fields used as named partition keys would be regions, postal codes, customer groups, or other business boundaries.
- Singleton partitioning. Singleton partitions are typically used when the service does not require any additional routing. For example, stateless services use this partitioning scheme by default.

Named and Singleton partitioning schemes are special forms of ranged partitions. By default, the Visual Studio templates for Service Fabric use ranged partitioning, as it is the most common and useful one. The remainder of this article focuses on the ranged partitioning scheme.

Ranged partitioning scheme

This is used to specify an integer range (identified by a low key and high key) and a number of partitions (n). It creates n partitions, each responsible for a non-overlapping subrange of the overall partition key range. For example, a ranged partitioning scheme with a low key of 0, a high key of 99, and a count of 4 would create four partitions, as shown below.

A common approach is to create a hash based on a unique key within the data set. Some common examples of keys would be a vehicle identification number (VIN), an employee ID, or a unique string. By using this unique key, you would then generate a hash code, modulus the key range, to use as your key. You can specify the upper and lower bounds of the allowed key range.

Select a hash algorithm

An important part of hashing is selecting your hash algorithm. A consideration is whether the goal is to group

similar keys near each other (locality sensitive hashing)--or if activity should be distributed broadly across all partitions (distribution hashing), which is more common.

The characteristics of a good distribution hashing algorithm are that it is easy to compute, it has few collisions, and it distributes the keys evenly. A good example of an efficient hash algorithm is the FNV-1 hash algorithm.

A good resource for general hash code algorithm choices is the Wikipedia page on hash functions.

Build a stateful service with multiple partitions

Let's create your first reliable stateful service with multiple partitions. In this example, you will build a very simple application where you want to store all last names that start with the same letter in the same partition.

Before you write any code, you need to think about the partitions and partition keys. You need 26 partitions (one for each letter in the alphabet), but what about the low and high keys? As we literally want to have one partition per letter, we can use 0 as the low key and 25 as the high key, as each letter is its own key.

> **NOTE**
> This is a simplified scenario, as in reality the distribution would be uneven. Last names starting with the letters "S" or "M" are more common than the ones starting with "X" or "Y".

1. Open **Visual Studio** > **File** > **New** > **Project**.
2. In the **New Project** dialog box, choose the Service Fabric application.
3. Call the project "AlphabetPartitions".
4. In the **Create a Service** dialog box, choose **Stateful** service and call it "Alphabet.Processing" as shown in the image below.

5. Set the number of partitions. Open the Applicationmanifest.xml file located in the ApplicationPackageRoot folder of the AlphabetPartitions project and update the parameter Processing_PartitionCount to 26 as shown below.

```
<Parameter Name="Processing_PartitionCount" DefaultValue="26" />
```

You also need to update the LowKey and HighKey properties of the StatefulService element in the ApplicationManifest.xml as shown below.

```
<Service Name="Processing">
  <StatefulService ServiceTypeName="ProcessingType" TargetReplicaSetSize="
[Processing_TargetReplicaSetSize]" MinReplicaSetSize="[Processing_MinReplicaSetSize]">
    <UniformInt64Partition PartitionCount="[Processing_PartitionCount]" LowKey="0" HighKey="25" />
  </StatefulService>
</Service>
```

6. For the service to be accessible, open up an endpoint on a port by adding the endpoint element of ServiceManifest.xml (located in the PackageRoot folder) for the Alphabet.Processing service as shown below:

```
<Endpoint Name="ProcessingServiceEndpoint" Port="8089" Protocol="http" Type="Internal" />
```

Now the service is configured to listen to an internal endpoint with 26 partitions.

7. Next, you need to override the `CreateServiceReplicaListeners()` method of the Processing class.

> **NOTE**
> For this sample, we assume that you are using a simple HttpCommunicationListener. For more information on reliable service communication, see The Reliable Service communication model.

8. A recommended pattern for the URL that a replica listens on is the following format: `{scheme}://{nodeIp}:{port}/{partitionid}/{replicaid}/{guid}`. So you want to configure your communication listener to listen on the correct endpoints and with this pattern.

   Multiple replicas of this service may be hosted on the same computer, so this address needs to be unique to the replica. This is why partition ID + replica ID are in the URL. HttpListener can listen on multiple addresses on the same port as long as the URL prefix is unique.

   The extra GUID is there for an advanced case where secondary replicas also listen for read-only requests. When that's the case, you want to make sure that a new unique address is used when transitioning from primary to secondary to force clients to re-resolve the address. '+' is used as the address here so that the replica listens on all available hosts (IP, FQDM, localhost, etc.) The code below shows an example.

```
 protected override IEnumerable<ServiceReplicaListener> CreateServiceReplicaListeners()
 {
      return new[] { new ServiceReplicaListener(context => this.CreateInternalListener(context))};
 }
 private ICommunicationListener CreateInternalListener(ServiceContext context)
 {

      EndpointResourceDescription internalEndpoint =
context.CodePackageActivationContext.GetEndpoint("ProcessingServiceEndpoint");
      string uriPrefix = String.Format(
             "{0}://+:{1}/{2}/{3}-{4}/",
             internalEndpoint.Protocol,
             internalEndpoint.Port,
             context.PartitionId,
             context.ReplicaOrInstanceId,
             Guid.NewGuid());

      string nodeIP = FabricRuntime.GetNodeContext().IPAddressOrFQDN;

      string uriPublished = uriPrefix.Replace("+", nodeIP);
      return new HttpCommunicationListener(uriPrefix, uriPublished, this.ProcessInternalRequest);
 }
```

It's also worth noting that the published URL is slightly different from the listening URL prefix. The listening URL is given to HttpListener. The published URL is the URL that is published to the Service Fabric Naming Service, which is used for service discovery. Clients will ask for this address through that discovery service. The address that clients get needs to have the actual IP or FQDN of the node in order to connect. So you need to replace '+' with the node's IP or FQDN as shown above.

9. The last step is to add the processing logic to the service as shown below.

```
 private async Task ProcessInternalRequest(HttpListenerContext context, CancellationToken cancelRequest)
 {
     string output = null;
     string user = context.Request.QueryString["lastname"].ToString();

     try
     {
         output = await this.AddUserAsync(user);
     }
     catch (Exception ex)
     {
         output = ex.Message;
     }

     using (HttpListenerResponse response = context.Response)
     {
         if (output != null)
         {
             byte[] outBytes = Encoding.UTF8.GetBytes(output);
             response.OutputStream.Write(outBytes, 0, outBytes.Length);
         }
     }
 }
 private async Task<string> AddUserAsync(string user)
 {
     IReliableDictionary<String, String> dictionary = await
this.StateManager.GetOrAddAsync<IReliableDictionary<String, String>>("dictionary");

     using (ITransaction tx = this.StateManager.CreateTransaction())
     {
         bool addResult = await dictionary.TryAddAsync(tx, user.ToUpperInvariant(), user);

         await tx.CommitAsync();

         return String.Format(
             "User {0} {1}",
             user,
             addResult ? "sucessfully added" : "already exists");
     }
 }
```

`ProcessInternalRequest` reads the values of the query string parameter used to call the partition and calls `AddUserAsync` to add the lastname to the reliable dictionary `dictionary`.

10. Let's add a stateless service to the project to see how you can call a particular partition.

    This service serves as a simple web interface that accepts the lastname as a query string parameter, determines the partition key, and sends it to the Alphabet.Processing service for processing.

11. In the **Create a Service** dialog box, choose **Stateless** service and call it "Alphabet.Web" as shown below.

.

12. Update the endpoint information in the ServiceManifest.xml of the Alphabet.WebApi service to open up a port as shown below.

```
<Endpoint Name="WebApiServiceEndpoint" Protocol="http" Port="8081"/>
```

13. You need to return a collection of ServiceInstanceListeners in the class Web. Again, you can choose to implement a simple HttpCommunicationListener.

```
protected override IEnumerable<ServiceInstanceListener> CreateServiceInstanceListeners()
{
    return new[] {new ServiceInstanceListener(context => this.CreateInputListener(context))};
}
private ICommunicationListener CreateInputListener(ServiceContext context)
{
    // Service instance's URL is the node's IP & desired port
    EndpointResourceDescription inputEndpoint =
context.CodePackageActivationContext.GetEndpoint("WebApiServiceEndpoint")
    string uriPrefix = String.Format("{0}://+:{1}/alphabetpartitions/", inputEndpoint.Protocol,
inputEndpoint.Port);
    var uriPublished = uriPrefix.Replace("+", FabricRuntime.GetNodeContext().IPAddressOrFQDN);
    return new HttpCommunicationListener(uriPrefix, uriPublished, this.ProcessInputRequest);
}
```

14. Now you need to implement the processing logic. The HttpCommunicationListener calls `ProcessInputRequest` when a request comes in. So let's go ahead and add the code below.

```
private async Task ProcessInputRequest(HttpListenerContext context, CancellationToken cancelRequest)
{
    String output = null;
    try
    {
        string lastname = context.Request.QueryString["lastname"];
        char firstLetterOfLastName = lastname.First();
        ServicePartitionKey partitionKey = new ServicePartitionKey(Char.ToUpper(firstLetterOfLastName) -
'A');

        ResolvedServicePartition partition = await
this.servicePartitionResolver.ResolveAsync(alphabetServiceUri, partitionKey, cancelRequest);
        ResolvedServiceEndpoint ep = partition.GetEndpoint();

        JObject addresses = JObject.Parse(ep.Address);
        string primaryReplicaAddress = (string)addresses["Endpoints"].First();

        UriBuilder primaryReplicaUriBuilder = new UriBuilder(primaryReplicaAddress);
        primaryReplicaUriBuilder.Query = "lastname=" + lastname;

        string result = await this.httpClient.GetStringAsync(primaryReplicaUriBuilder.Uri);

        output = String.Format(
                "Result: {0}. <p>Partition key: '{1}' generated from the first letter '{2}' of input
value '{3}'. <br>Processing service partition ID: {4}. <br>Processing service replica address: {5}",
                result,
                partitionKey,
                firstLetterOfLastName,
                lastname,
                partition.Info.Id,
                primaryReplicaAddress);
    }
    catch (Exception ex) { output = ex.Message; }

    using (var response = context.Response)
    {
        if (output != null)
        {
            output = output + "added to Partition: " + primaryReplicaAddress;
            byte[] outBytes = Encoding.UTF8.GetBytes(output);
            response.OutputStream.Write(outBytes, 0, outBytes.Length);
        }
    }
}
```

Let's walk through it step by step. The code reads the first letter of the query string parameter `lastname` into a char. Then, it determines the partition key for this letter by subtracting the hexadecimal value of `A` from the hexadecimal value of the last names' first letter.

```
string lastname = context.Request.QueryString["lastname"];
char firstLetterOfLastName = lastname.First();
ServicePartitionKey partitionKey = new ServicePartitionKey(Char.ToUpper(firstLetterOfLastName) - 'A');
```

Remember, for this example, we are using 26 partitions with one partition key per partition. Next, we obtain the service partition `partition` for this key by using the `ResolveAsync` method on the `servicePartitionResolver` object. `servicePartitionResolver` is defined as

```
private readonly ServicePartitionResolver servicePartitionResolver =
ServicePartitionResolver.GetDefault();
```

The `ResolveAsync` method takes the service URI, the partition key, and a cancellation token as parameters. The service URI for the processing service is `fabric:/AlphabetPartitions/Processing`. Next, we get the endpoint of the partition.

```
ResolvedServiceEndpoint ep = partition.GetEndpoint()
```

Finally, we build the endpoint URL plus the querystring and call the processing service.

```
JObject addresses = JObject.Parse(ep.Address);
string primaryReplicaAddress = (string)addresses["Endpoints"].First();

UriBuilder primaryReplicaUriBuilder = new UriBuilder(primaryReplicaAddress);
primaryReplicaUriBuilder.Query = "lastname=" + lastname;

string result = await this.httpClient.GetStringAsync(primaryReplicaUriBuilder.Uri);
```

Once the processing is done, we write the output back.

15. The last step is to test the service. Visual Studio uses application parameters for local and cloud deployment. To test the service with 26 partitions locally, you need to update the `Local.xml` file in the ApplicationParameters folder of the AlphabetPartitions project as shown below:

```
<Parameters>
  <Parameter Name="Processing_PartitionCount" Value="26" />
  <Parameter Name="WebApi_InstanceCount" Value="1" />
</Parameters>
```

16. Once you finish deployment, you can check the service and all of its partitions in the Service Fabric Explorer.

17. In a browser, you can test the partitioning logic by entering `http://localhost:8081/?lastname=somename`. You will see that each last name that starts with the same letter is being stored in the same partition.

The entire source code of the sample is available on GitHub.

Next steps

For information on Service Fabric concepts, see the following:

- Availability of Service Fabric services
- Scalability of Service Fabric services
- Capacity planning for Service Fabric applications

Availability of Service Fabric services

11/15/2016 • 2 min to read • Edit on GitHub

Contributors

Aprameya Rao • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Wike • MarkDeichmiller

Azure Service Fabric services can be either stateful or stateless. This article gives an overview of how Service Fabric maintains availability of a service in the event of failures.

Availability of Service Fabric stateless services

A stateless service is an application service that does not have any local persistent state.

Creating a stateless service requires defining an instance count, which is the number of instances of the stateless service that should be running in the cluster. This is the number of copies of the application logic that will be instantiated in the cluster. Increasing the number of instances is the recommended way of scaling up a stateless service.

When a fault is detected on any instance of a stateless service, a new instance is created on some other eligible node in the cluster.

Availability of Service Fabric stateful services

A stateful service has some state associated with it. In Service Fabric, a stateful service is modeled as a set of replicas. Each replica is an instance of the code of the service that has a copy of the state. Read and write operations are performed at one replica (called the primary). Changes to state from write operations are *replicated* to multiple other replicas (called active secondaries). The combination of primary and active secondary replicas is the replica set of the service.

There can be only one primary replica servicing read and write requests, but there can be multiple active secondary replicas. The number of active secondary replicas is configurable, and a higher number of replicas can tolerate a greater number of concurrent software and hardware failures.

In the event of a fault (when the primary replica goes down), Service Fabric makes one of the active secondary replicas the new primary replica. This active secondary replica already has the updated version of the state (via *replication*), and it can continue processing further read and write operations.

This concept--of a replica being either a primary or active secondary--is known as the replica role.

Replica roles

The role of a replica is used to manage the life cycle of the state being managed by that replica. A replica whose role is primary services read requests. It also services write requests by updating its state and replicating the changes to the active secondaries in its replica set. The role of an active secondary is to receive state changes that the primary replica has replicated and update its view of the state.

> **NOTE**
>
> Higher-level programming models such as the reliable actors framework abstract away the concept of replica role from the developer.

Next steps

For more information on Service Fabric concepts, see the following:

- Scalability of Service Fabric services
- Partitioning Service Fabric services
- Defining and managing state

Scaling Service Fabric applications

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

Aprameya Rao • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Ryan Wike • MarkDeichmiller

Azure Service Fabric makes it easy to build scalable applications by load-balancing services, partitions, and replicas on all the nodes in a cluster. This enables maximum resource utilization.

High scale for Service Fabric applications can be achieved in two ways:

1. Scaling at the partition level
2. Scaling at the service name level

Scaling at the partition level

Service Fabric supports partitioning an individual service into multiple smaller partitions. The partitioning overview provides information on the types of partitioning schemes that are supported. The replicas of each partition are spread across the nodes in a cluster. Consider a service that uses a ranged partitioning scheme with a low key of 0, a high key of 99, and four partitions. In a three-node cluster, the service might be laid out with four replicas that share the resources on each node as shown here:

Increasing the number of nodes allows Service Fabric to utilize the resources on the new nodes by moving some of the replicas to empty nodes. By increasing the number of nodes to four, the service now has three replicas running on each node (of different partitions), allowing for better resource utilization and performance.

Scaling at the service name level

A service instance is a specific instance of an application name and a service type name (see Service Fabric application life cycle). During the creation of a service, you specify the partition scheme (see Partitioning Service Fabric services) to be used.

The first level of scaling is by service name. You can create new instances of a service, with different levels of partitioning, as your older service instances become busy. This allows new service consumers to use less-busy service instances, rather than busier ones.

One option for increasing capacity, as well as increasing or decreasing partition counts, is to create a new service instance with a new partition scheme. This adds complexity, though, as any consuming clients need to know when and how to use the differently named service.

Example scenario: Embedded dates

One possible scenario would be to use date information as part of the service name. For example, you could use a service instance with a specific name for all customers who joined in 2013 and another name for customers who joined in 2014. This naming scheme allows for programmatically increasing the names depending on the date (as 2014 approaches, the service instance for 2014 can be created on demand).

However, this approach is based on the clients using application-specific naming information that is outside the scope of Service Fabric knowledge.

- *Using a naming convention*: In 2013, when your application goes live, you create a service called fabric:/app/service2013. Near the second quarter of 2013, you create another service, called fabric:/app/service2014. Both of these services are of the same service type. In this approach, your client will need to employ logic to construct the appropriate service name based on the year.
- *Using a lookup service*: Another pattern is to provide a secondary lookup service, which can provide the name of the service for a desired key. New service instances can then be created by the lookup service. The lookup service itself doesn't retain any application data, only data about the service names that it creates. Thus, for the year-based example above, the client would first contact the lookup service to find out the name of the service handling data for a given year, and then use that service name for performing the actual operation. The result of the first lookup can be cached.

Next steps

For more information on Service Fabric concepts, see the following:

- Availability of Service Fabric services
- Partitioning Service Fabric services

- Defining and managing state

Capacity planning for Service Fabric applications

11/15/2016 • 6 min to read • Edit on GitHub

Contributors

Mani Ramaswamy • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Rebecca Agiewich

This document teaches you how to estimate the amount of resources (CPUs, RAM, disk storage) you need to run your Azure Service Fabric applications. It is common for your resource requirements to change over time. You typically require few resources as you develop/test your service, and then require more resources as you go into production and your application grows in popularity. When you design your application, think through the long-term requirements and make choices that allow your service to scale to meet high customer demand.

When you create a Service Fabric cluster, you decide what kinds of virtual machines (VMs) make up the cluster. Each VM comes with a limited amount of resources in the form of CPUs (cores and speed), network bandwidth, RAM, and disk storage. As your service grows over time, you can upgrade to VMs that offer greater resources and/or add more VMs to your cluster. To do the latter, you must architect your service initially so it can take advantage of new VMs that get dynamically added to the cluster.

Some services manage little to no data on the VMs themselves. Therefore, capacity planning for these services should focus primarily on performance, which means selecting the appropriate CPUs (cores and speed) of the VMs. In addition, you should consider network bandwidth, including how frequently network transfers are occurring and how much data is being transferred. If your service needs to perform well as service usage increases, you can add more VMs to the cluster and load balance the network requests across all the VMs.

For services that manage large amounts of data on the VMs, capacity planning should focus primarily on size. Thus, you should carefully consider the capacity of the VM's RAM and disk storage. The virtual memory management system in Windows makes disk space look like RAM to application code. In addition, the Service Fabric runtime provides smart paging keeping only hot data in memory and moving the cold data to disk. Applications can thus use more memory than is physically available on the VM. Having more RAM simply increases performance, since the VM can keep more disk storage in RAM. The VM you select should have a disk large enough to store the data that you want on the VM. Similarly, the VM should have enough RAM to provide you with the performance you desire. If your service's data grows over time, you can add more VMs to the cluster and partition the data across all the VMs.

Determine how many nodes you need

Partitioning your service allows you to scale out your service's data. For more information on partitioning, see Partitioning Service Fabric. Each partition must fit within a single VM, but multiple (small) partitions can be placed on a single VM. So, having more small partitions gives you greater flexibility than having a few larger partitions. The trade-off is that having lots of partitions increases Service Fabric overhead and you cannot perform transacted operations across partitions. There is also more potential network traffic if your service code frequently needs to access pieces of data that live in different partitions. When designing your service, you should carefully consider these pros and cons to arrive at an effective partitioning strategy.

Let's assume your application has a single stateful service that has a store size that you expect to grow to DB_Size GB in a year. You are willing to add more applications (and partitions) as you experience growth beyond that year. The replication factor (RF), which determines the number of replicas for your service impacts the total DB_Size. The total DB_Size across all replicas is the Replication Factor multiplied by DB_Size. Node_Size represents the disk space/RAM per node you want to use for your service. For best performance, the DB_Size should fit into memory across the cluster, and a Node_Size that is around the RAM of the VM should be chosen. By allocating a Node_Size

that is larger than the RAM capacity, you are relying on the paging provided by the Service Fabric runtime. Thus, your performance may not be optimal if your entire data is considered to be hot (since then the data is paged in/out). However, for many services where only a fraction of the data is hot, it is more cost-effective.

The number of nodes required for maximum performance can be computed as follows:

```
Number of Nodes = (DB_Size * RF)/Node_Size
```

Account for growth

You may want to compute the number of nodes based on the DB_Size that you expect your service to grow to, in addition to the DB_Size that you began with. Then, grow the number of nodes as your service grows so that you are not over-provisioning the number of nodes. But the number of partitions should be based on the number of nodes that are needed when you're running your service at maximum growth.

It is good to have some extra machines available at any time so that you can handle any unexpected spikes or failure (for example, if a few VMs go down). While the extra capacity should be determined by using your expected spikes, a starting point is to reserve a few extra VMs (5-10 percent extra).

The preceding assumes a single stateful service. If you have more than one stateful service, you have to add the DB_Size associated with the other services into the equation. Alternatively, you can compute the number of nodes separately for each stateful service. Your service may have replicas or partitions that aren't balanced. Keep in mind that partitions may also have more data than others. For more information on partitioning, see partitioning article on best practices. However, the preceding equation is partition and replica agnostic, because Service Fabric ensures that the replicas are spread out among the nodes in an optimized manner.

Use a spreadsheet for cost calculation

Now let's put some real numbers in the formula. An example spreadsheet shows how to plan the capacity for an application that contains three types of data objects. For each object, we approximate its size and how many objects we expect to have. We also select how many replicas we want of each object type. The spreadsheet calculates the total amount of memory to be stored in the cluster.

Then we enter a VM size and monthly cost. Based on the VM size, the spreadsheet tells you the minimum number of partitions you must use to split your data to physically fit on the nodes. You may desire a larger number of partitions to accommodate your application's specific computation and network traffic needs. The spreadsheet shows the number of partitions that are managing the user profile objects has increased from one to six.

Now, based on all this information, the spreadsheet shows that you could physically get all the data with the desired partitions and replicas on a 26-node cluster. However, this cluster would be densely packed, so you may want some additional nodes to accommodate node failures and upgrades. The spreadsheet also shows that having more than 57 nodes provides no additional value because you would have empty nodes. Again, you may want to go above 57 nodes anyway to accommodate node failures and upgrades. You can tweak the spreadsheet to match your application's specific needs.

Data Object Type		Inventory Item	User Profile	Order
Size of Object (KB)		4	2	1
Number of Objects		10,000	3,000,000	45,000,000
Total Size of Object (KB)		*40,000*	*6,000,000*	*45,000,000*
Number of Desired Replicas		3	5	3
Total size of Replicas (KB)		*120,000*	*30,000,000*	*135,000,000*
Total size of data to store in the Cluster (GB)	*157*			
Node Available RAM/Storage (GB)	6	(Put available RAM or disk size based on your performance needs)		
Node Cost/Month	$ 229.00	(Ex: Azure A3 pricing)		
Minimum Partitions Required		1	1	8
Number of Desired Partitions		1	6	8
Minimum Number of Nodes	26			
Cost of Nodes/Month	$ 6,010			
Number of Nodes (1 replica per node)	57			
Cost of Nodes/Month	$ 13,053			

Next steps

Check out Partitioning Service Fabric services to learn more about partitioning your service.

Introduction to the Fault Analysis Service

11/15/2016 • 6 min to read • Edit on GitHub

Contributors

rishirsinha • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • John Bergant • Vaclav Turecek • Tanvir Tanviruzzaman • stevehug • Jesse Benson • Dene Hager

The Fault Analysis Service is designed for testing services that are built on Microsoft Azure Service Fabric. With the Fault Analysis Service you can induce meaningful faults and run complete test scenarios against your applications. These faults and scenarios exercise and validate the numerous states and transitions that a service will experience throughout its lifetime, all in a controlled, safe, and consistent manner.

Actions are the individual faults targeting a service for testing it. A service developer can use these as building blocks to write complicated scenarios. For example:

- Restart a node to simulate any number of situations where a machine or VM is rebooted.
- Move a replica of your stateful service to simulate load balancing, failover, or application upgrade.
- Invoke quorum loss on a stateful service to create a situation where write operations can't proceed because there aren't enough "back-up" or "secondary" replicas to accept new data.
- Invoke data loss on a stateful service to create a situation where all in-memory state is completely wiped out.

Scenarios are complex operations composed of one or more actions. The Fault Analysis Service provides two built-in complete scenarios:

- Chaos Scenario
- Failover Scenario

Testing as a service

The Fault Analysis Service is a Service Fabric system service that is automatically started with a Service Fabric cluster. This is service acts as the host for fault injection, test scenario execution, and health analysis.

SYSTEM

Search list Reset All

Name	Service Type	Version	Service Kind	Health State	Status
fabric:/System/ClusterManagerService	ClusterManagerServiceType	5.0.135.9590	Stateful	OK	Active
fabric:/System/FailoverManagerService	FMServiceType	5.0.135.9590	Stateful	OK	Active
fabric:/System/FaultAnalysisService	FaultAnalysisServiceType	5.0.135.9590	Stateful	OK	Active
fabric:/System/NamingService	NamingStoreService	5.0.135.9590	Stateful	OK	Active

When a fault action or test scenario is initiated, a command is sent to the Fault Analysis Service to run the fault action or test scenario. The Fault Analysis Service is stateful so that it can reliable run faults and scenarios and validate results. For example, a long-running test scenario can be reliably executed by the Fault Analysis Service. And because tests are being executed inside the cluster, the service can examine the state of the cluster and your services to provide more in-depth information about failures.

Testing distributed systems

Service Fabric makes the job of writing and managing distributed scalable applications significantly easier. The Fault Analysis Service makes testing a distributed application similarly easier. There are three main issues that need to be solved while testing:

1. Simulating/generating failures that might occur in real-world scenarios: One of the important aspects of Service Fabric is that it enables distributed applications to recover from various failures. However, to test that the application is able to recover from these failures, we need a mechanism to simulate/generate these real-world failures in a controlled test environment.
2. The ability to generate correlated failures: Basic failures in the system, such as network failures and machine failures, are easy to produce individually. Generating a significant number of scenarios that can happen in the real world as a result of the interactions of these individual failures is non-trivial.
3. Unified experience across various levels of development and deployment: There are many fault injection systems that can do various types of failures. However, the experience in all of these is poor when moving from one-box developer scenarios, to running the same tests in large test environments, to using them for tests in production.

While there are many mechanisms to solve these problems, a system that does the same with required guarantees--all the way from a one-box developer environment, to test in production clusters--is missing. The Fault Analysis Service helps the application developers concentrate on testing their business logic. The Fault Analysis Service provides all the capabilities needed to test the interaction of the service with the underlying distributed system.

Simulating/generating real-world failure scenarios

To test the robustness of a distributed system against failures, we need a mechanism to generate failures. While in theory, generating a failure like a node down seems easy, it starts hitting the same set of consistency problems that Service Fabric is trying to solve. As an example, if we want to shut down a node, the required workflow is the following:

1. From the client, issue a shutdown node request.
2. Send the request to the right node.

   a. If the node is not found, it should fail.

   b. If the node is found, it should return only if the node is shut down.

To verify the failure from a test perspective, the test needs to know that when this failure is induced, the failure actually happens. The guarantee that Service Fabric provides is that either the node will go down or was already down when the command reached the node. In either case the test should be able to correctly reason about the state and succeed or fail correctly in its validation. A system implemented outside of Service Fabric to do the same set of failures could hit many network, hardware, and software issues, which would prevent it from providing the preceding guarantees. In the presence of the issues stated before, Service Fabric will reconfigure the cluster state to work around the issues, and hence the Fault Analysis Service will still be able to give the right set of guarantees.

Generating required events and scenarios

While simulating a real-world failure consistently is tough to start with, the ability to generate correlated failures is even tougher. For example, a data loss happens in a stateful persisted service when the following things happen:

1. Only a write quorum of the replicas are caught up on replication. All the secondary replicas lag behind the primary.
2. The write quorum goes down because of the replicas going down (due to a code package or node going down).
3. The write quorum cannot come back up because the data for the replicas is lost (due to disk corruption or machine reimaging).

These correlated failures do happen in the real world, but not as frequently as individual failures. The ability to test for these scenarios before they happen in production is critical. Even more important is the ability to simulate these scenarios with production workloads in controlled circumstances (in the middle of the day with all engineers on deck). That is much better than having it happen for the first time in production at 2:00 A.M.

Unified experience across different environments

The practice traditionally has been to create three different sets of experiences, one for the development environment, one for tests, and one for production. The model was:

1. In the development environment, produce state transitions that allow unit tests of individual methods.
2. In the test environment, produce failures to allow end-to-end tests that exercise various failure scenarios.
3. Keep the production environment pristine to prevent any non-natural failures and to ensure that there is extremely quick human response to failure.

In Service Fabric, through the Fault Analysis Service, we are proposing to turn this around and use the same methodology from developer environment to production. There are two ways to achieve this:

1. To induce controlled failures, use the Fault Analysis Service APIs from a one-box environment all the way to production clusters.
2. To give the cluster a fever that causes automatic induction of failures, use the Fault Analysis Service to generate automatic failures. Controlling the rate of failures through configuration enables the same service to be tested differently in different environments.

With Service Fabric, though the scale of failures would be different in the different environments, the actual mechanisms would be identical. This allows for a much quicker code-to-deployment pipeline and the ability to test the services under real-world loads.

Using the Fault Analysis Service

C#

Fault Analysis Service features are in the System.Fabric namespace in the Microsoft.ServiceFabric NuGet package. To use the Fault Analysis Service features, include the nuget package as a reference in your project.

PowerShell

To use PowerShell, you must install the Service Fabric SDK. After the SDK is installed, the ServiceFabric PowerShell module is auto loaded for you to use.

Next steps

To create truly cloud-scale services, it is critical to ensure, both before and after deployment, that services can withstand real world failures. In the services world today, the ability to innovate quickly and move code to production quickly is very important. The Fault Analysis Service helps service developers to do precisely that.

Begin testing your applications and services using the built-in test scenarios, or author your own test scenarios using the fault actions provided by the Fault Analysis Service.

Service Fabric testability scenarios: Service communication

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Vaclav Turecek • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Tanvir Tanviruzzaman • MarkDeichmiller • Jesse Benson

Microservices and service-oriented architectural styles surface naturally in Azure Service Fabric. In these types of distributed architectures, componentized microservice applications are typically composed of multiple services that need to talk to each other. In even the simplest cases, you generally have at least a stateless web service and a stateful data storage service that need to communicate.

Service-to-service communication is a critical integration point of an application, because each service exposes a remote API to other services. Working with a set of API boundaries that involves I/O generally requires some care, with a good amount of testing and validation.

There are numerous considerations to make when these service boundaries are wired together in a distributed system:

- *Transport protocol*. Will you use HTTP for increased interoperability, or a custom binary protocol for maximum throughput?
- *Error handling*. How will permanent and transient errors be handled? What will happen when a service moves to a different node?
- *Timeouts and latency*. In multitiered applications, how will each service layer handle latency through the stack and to the user?

Whether you use one of the built-in service communication components provided by Service Fabric or you build your own, testing the interactions between your services is critical to ensuring resiliency in your application.

Prepare for services to move

Service instances may move around over time. This is especially true when they are configured with load metrics for custom-tailored optimal resource balancing. Service Fabric moves your service instances to maximize their availability even during upgrades, failovers, scale-out, and other situations that occur over the lifetime of a distributed system.

As services move around in the cluster, your clients and other services should be prepared to handle two scenarios when they talk to a service:

- The service instance or partition replica has moved since the last time you talked to it. This is a normal part of a service lifecycle, and it should be expected to happen during the lifetime of your application.
- The service instance or partition replica is in the process of moving. Although failover of a service from one node to another occurs very quickly in Service Fabric, there may be a delay in availability if the communication component of your service is slow to start.

Handling these scenarios gracefully is important for a smooth-running system. To do so, keep in mind that:

- Every service that can be connected to has an *address* that it listens on (for example, HTTP or WebSockets). When a service instance or partition moves, its address endpoint changes. (It moves to a different node with a different IP address.) If you're using the built-in communication components, they will handle re-resolving

service addresses for you.

- There may be a temporary increase in service latency as the service instance starts up its listener again. This depends on how quickly the service opens the listener after the service instance is moved.
- Any existing connections need to be closed and reopened after the service opens on a new node. A graceful node shutdown or restart allows time for existing connections to be shut down gracefully.

Test it: Move service instances

By using Service Fabric's testability tools, you can author a test scenario to test these situations in different ways:

1. Move a stateful service's primary replica.

   The primary replica of a stateful service partition can be moved for any number of reasons. Use this to target the primary replica of a specific partition to see how your services react to the move in a very controlled manner.

   ```
   PS > Move-ServiceFabricPrimaryReplica -PartitionId 6faa4ffa-521a-44e9-8351-dfca0f7e0466 -ServiceName
   fabric:/MyApplication/MyService
   ```

2. Stop a node.

   When a node is stopped, Service Fabric moves all of the service instances or partitions that were on that node to one of the other available nodes in the cluster. Use this to test a situation where a node is lost from your cluster and all of the service instances and replicas on that node have to move.

   You can stop a node by using the PowerShell **Stop-ServiceFabricNode** cmdlet:

   ```
   PS > Restart-ServiceFabricNode -NodeName Node_1
   ```

Maintain service availability

As a platform, Service Fabric is designed to provide high availability of your services. But in extreme cases, underlying infrastructure problems can still cause unavailability. It is important to test for these scenarios, too.

Stateful services use a quorum-based system to replicate state for high availability. This means that a quorum of replicas needs to be available to perform write operations. In rare cases, such as a widespread hardware failure, a quorum of replicas may not be available. In these cases, you will not be able to perform write operations, but you will still be able to perform read operations.

Test it: Write operation unavailability

By using the testability tools in Service Fabric, you can inject a fault that induces quorum loss as a test. Although such a scenario is rare, it is important that clients and services that depend on a stateful service are prepared to handle situations where they cannot make write requests to it. It is also important that the stateful service itself is aware of this possibility and can gracefully communicate it to callers.

You can induce quorum loss by using the PowerShell **Invoke-ServiceFabricPartitionQuorumLoss** cmdlet:

```
PS > Invoke-ServiceFabricPartitionQuorumLoss -ServiceName fabric:/Myapplication/MyService -QuorumLossMode
QuorumReplicas -QuorumLossDurationInSeconds 20
```

In this example, we set `QuorumLossMode` to `QuorumReplicas` to indicate that we want to induce quorum loss without taking down all replicas. This way, read operations are still possible. To test a scenario where an entire partition is unavailable, you can set this switch to `AllReplicas`.

Next steps

Learn more about testability actions

Learn more about testability scenarios

Induce controlled Chaos in Service Fabric clusters

11/15/2016 • 4 min to read • Edit on GitHub

Contributors

Tanvir Tanviruzzaman • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • ShawnJackson • katiecumming

Large-scale distributed systems like cloud infrastructures are inherently unreliable. Azure Service Fabric enables developers to write reliable services on top of an unreliable infrastructure. To write robust services, developers need to be able to induce faults against such unreliable infrastructure to test the stability of their services.

The Fault Injection and Cluster Analysis Service (also known as the Fault Analysis Service) gives developers the ability to induce fault actions to test services. However, targeted simulated faults get you only so far. To take the testing further, you can use Chaos.

Chaos simulates continuous, interleaved faults (both graceful and ungraceful) throughout the cluster over extended periods of time. After you configure Chaos with the rate and the kind of faults, you can start or stop it through either C# APIs or PowerShell to generate faults in the cluster and your service.

While Chaos is running, it produces different events that capture the state of the run at the moment. For example, an ExecutingFaultsEvent contains all the faults that are being executed in that iteration. A ValidationFailedEvent contains the details of a failure that was found during cluster validation. You can invoke the GetChaosReportAsync API to get the report of Chaos runs.

Faults induced in Chaos

Chaos generates faults across the entire Service Fabric cluster and compresses faults that are seen in months or years into a few hours. The combination of interleaved faults with the high fault rate finds corner cases that are otherwise missed. This Chaos exercise leads to a significant improvement in the code quality of the service.

Chaos induces faults from the following categories:

- Restart a node
- Restart a deployed code package
- Remove a replica
- Restart a replica
- Move a primary replica (configurable)
- Move a secondary replica (configurable)

Chaos runs in multiple iterations. Each iteration consists of faults and cluster validation for the specified period. You can configure the time spent for the cluster to stabilize and for validation to succeed. If a failure is found in cluster validation, Chaos generates and persists a ValidationFailedEvent with the UTC timestamp and the failure details.

For example, consider an instance of Chaos that is set to run for an hour with a maximum of three concurrent faults. Chaos induces three faults, and then validates the cluster health. It iterates through the previous step until it is explicitly stopped through the StopChaosAsync API or one-hour passes. If the cluster becomes unhealthy in any iteration (that is, it does not stabilize within a configured time), Chaos generates a ValidationFailedEvent. This event indicates that something has gone wrong and might need further investigation.

In its current form, Chaos induces only safe faults. This implies that, in the absence of external faults, a quorum loss or data loss never occurs.

Important configuration options

- **TimeToRun**: Total time that Chaos runs before it finishes with success. You can stop Chaos before it has run for the TimeToRun period through the StopChaos API.
- **MaxClusterStabilizationTimeout**: The maximum amount of time to wait for the cluster to become healthy before checking on it again. This wait is to reduce the load on the cluster while it is recovering. The checks performed are:
  - If the cluster health is OK
  - If the service health is OK
  - If the target replica set size is achieved for the service partition
  - That no InBuild replicas exist
- **MaxConcurrentFaults**: The maximum number of concurrent faults that are induced in each iteration. The higher the number, the more aggressive Chaos is. This results in more complex failovers and transition combinations. Chaos guarantees that, in the absence of external faults, there is no quorum loss or data loss, regardless of how high a value this configuration has.
- **EnableMoveReplicaFaults**: Enables or disables the faults that cause the primary or secondary replicas to move. These faults are disabled by default.
- **WaitTimeBetweenIterations**: The amount of time to wait between iterations, that is, after a round of faults and corresponding validation.
- **WaitTimeBetweenFaults**: The amount of time to wait between two consecutive faults in an iteration.

How to run Chaos

C#:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using System.Fabric;

using System.Diagnostics;
using System.Fabric.Chaos.DataStructures;

class Program
{
    private class ChaosEventComparer : IEqualityComparer<ChaosEvent>
    {
        public bool Equals(ChaosEvent x, ChaosEvent y)
        {
            return x.TimeStampUtc.Equals(y.TimeStampUtc);
        }

        public int GetHashCode(ChaosEvent obj)
        {
            return obj.TimeStampUtc.GetHashCode();
        }
    }

    static void Main(string[] args)
    {
        var clusterConnectionString = "localhost:19000";
        using (var client = new FabricClient(clusterConnectionString))
        {
            var startTimeUtc = DateTime.UtcNow;
            var stabilizationTimeout = TimeSpan.FromSeconds(30.0);
            var timeToRun = TimeSpan.FromMinutes(60.0);
            var maxConcurrentFaults = 3;

            var parameters = new ChaosParameters(
                stabilizationTimeout,
```

```
                maxConcurrentFaults,
                true, /* EnableMoveReplicaFault */
                timeToRun);

            try
            {
                client.TestManager.StartChaosAsync(parameters).GetAwaiter().GetResult();
            }
            catch (FabricChaosAlreadyRunningException)
            {
                Console.WriteLine("An instance of Chaos is already running in the cluster.");
            }

            var filter = new ChaosReportFilter(startTimeUtc, DateTime.MaxValue);

            var eventSet = new HashSet<ChaosEvent>(new ChaosEventComparer());

            while (true)
            {
                var report = client.TestManager.GetChaosReportAsync(filter).GetAwaiter().GetResult();

                foreach (var chaosEvent in report.History)
                {
                    if (eventSet.add(chaosEvent))
                    {
                        Console.WriteLine(chaosEvent);
                    }
                }

                // When Chaos stops, a StoppedEvent is created.
                // If a StoppedEvent is found, exit the loop.
                var lastEvent = report.History.LastOrDefault();

                if (lastEvent is StoppedEvent)
                {
                    break;
                }

                Task.Delay(TimeSpan.FromSeconds(1.0)).GetAwaiter().GetResult();
            }
        }
    }
}
```

PowerShell:

```
$connection = "localhost:19000"
$timeToRun = 60
$maxStabilizationTimeSecs = 180
$concurrentFaults = 3
$waitTimeBetweenIterationsSec = 60

Connect-ServiceFabricCluster $connection

$events = @{}
$now = [System.DateTime]::UtcNow

Start-ServiceFabricChaos -TimeToRunMinute $timeToRun -MaxConcurrentFaults $concurrentFaults -
MaxClusterStabilizationTimeoutSec $maxStabilizationTimeSecs -EnableMoveReplicaFaults -
WaitTimeBetweenIterationsSec $waitTimeBetweenIterationsSec

while($true)
{
    $stopped = $false
    $report = Get-ServiceFabricChaosReport -StartTimeUtc $now -EndTimeUtc ([System.DateTime]::MaxValue)

    foreach ($e in $report.History) {

        if(-Not ($events.Contains($e.TimeStampUtc.Ticks)))
        {
            $events.Add($e.TimeStampUtc.Ticks, $e)
            if($e -is [System.Fabric.Chaos.DataStructures.ValidationFailedEvent])
            {
                Write-Host -BackgroundColor White -ForegroundColor Red $e
            }
            else
            {
                if($e -is [System.Fabric.Chaos.DataStructures.StoppedEvent])
                {
                    $stopped = $true
                }

                Write-Host $e
            }
        }
    }

    if($stopped -eq $true)
    {
        break
    }

    Start-Sleep -Seconds 1
}

Stop-ServiceFabricChaos
```

Testability actions

11/15/2016 • 8 min to read • Edit on GitHub

Contributors

Tanvir Tanviruzzaman • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Vaclav Turecek • ShawnJackson • Ryan Wike • heeldin

In order to simulate an unreliable infrastructure, Azure Service Fabric provides you, the developer, with ways to simulate various real-world failures and state transitions. These are exposed as testability actions. The actions are the low-level APIs that cause a specific fault injection, state transition, or validation. By combining these actions, you can write comprehensive test scenarios for your services.

Service Fabric provides some common test scenarios composed of these actions. We highly recommend that you utilize these built-in scenarios, which are carefully chosen to test common state transitions and failure cases. However, actions can be used to create custom test scenarios when you want to add coverage for scenarios that are not covered by the built-in scenarios yet or that are custom tailored for your application.

C# implementations of the actions are found in the System.Fabric.dll assembly. The System Fabric PowerShell module is found in the Microsoft.ServiceFabric.Powershell.dll assembly. As part of runtime installation, the ServiceFabric PowerShell module is installed to allow for ease of use.

Graceful vs. ungraceful fault actions

Testability actions are classified into two major buckets:

- Ungraceful faults: These faults simulate failures like machine restarts and process crashes. In such cases of failures, the execution context of process stops abruptly. This means no cleanup of the state can run before the application starts up again.
- Graceful faults: These faults simulate graceful actions like replica moves and drops triggered by load balancing. In such cases, the service gets a notification of the close and can clean up the state before exiting.

For better quality validation, run the service and business workload while inducing various graceful and ungraceful faults. Ungraceful faults exercise scenarios where the service process abruptly exits in the middle of some workflow. This tests the recovery path once the service replica is restored by Service Fabric. This will help test data consistency and whether the service state is maintained correctly after failures. The other set of failures (the graceful failures) test that the service correctly reacts to replicas being moved around by Service Fabric. This tests handling of cancellation in the RunAsync method. The service needs to check for the cancellation token being set, correctly save its state, and exit the RunAsync method.

Testability actions list

ACTION	DESCRIPTION	MANAGED API	POWERSHELL CMDLET	GRACEFUL/UNGRACEFUL FAULTS
CleanTestState	Removes all the test state from the cluster in case of a bad shutdown of the test driver.	CleanTestStateAsync	Remove-ServiceFabricTestState	Not applicable

ACTION	DESCRIPTION	MANAGED API	POWERSHELL CMDLET	GRACEFUL/UNGRACEFUL FAULTS
InvokeDataLoss	Induces data loss into a service partition.	InvokeDataLossAsync	Invoke-ServiceFabricPartitionDataLoss	Graceful
InvokeQuorumLoss	Puts a given stateful service partition into quorum loss.	InvokeQuorumLossAsync	Invoke-ServiceFabricQuorumLoss	Graceful
Move Primary	Moves the specified primary replica of a stateful service to the specified cluster node.	MovePrimaryAsync	Move-ServiceFabricPrimaryReplica	Graceful
Move Secondary	Moves the current secondary replica of a stateful service to a different cluster node.	MoveSecondaryAsync	Move-ServiceFabricSecondaryReplica	Graceful
RemoveReplica	Simulates a replica failure by removing a replica from a cluster. This will close the replica and will transition it to role 'None', removing all of its state from the cluster.	RemoveReplicaAsync	Remove-ServiceFabricReplica	Graceful
RestartDeployedCodePackage	Simulates a code package process failure by restarting a code package deployed on a node in a cluster. This aborts the code package process, which will restart all the user service replicas hosted in that process.	RestartDeployedCodePackageAsync	Restart-ServiceFabricDeployedCodePackage	Ungraceful
RestartNode	Simulates a Service Fabric cluster node failure by restarting a node.	RestartNodeAsync	Restart-ServiceFabricNode	Ungraceful
RestartPartition	Simulates a datacenter blackout or cluster blackout scenario by restarting some or all replicas of a partition.	RestartPartitionAsync	Restart-ServiceFabricPartition	Graceful

ACTION	DESCRIPTION	MANAGED API	POWERSHELL CMDLET	GRACEFUL/UNGRACEFUL FAULTS
RestartReplica	Simulates a replica failure by restarting a persisted replica in a cluster, closing the replica and then reopening it.	RestartReplicaAsync	Restart-ServiceFabricReplica	Graceful
StartNode	Starts a node in a cluster that is already stopped.	StartNodeAsync	Start-ServiceFabricNode	Not applicable
StopNode	Simulates a node failure by stopping a node in a cluster. The node will stay down until StartNode is called.	StopNodeAsync	Stop-ServiceFabricNode	Ungraceful
ValidateApplication	Validates the availability and health of all Service Fabric services within an application, usually after inducing some fault into the system.	ValidateApplicationAsync	Test-ServiceFabricApplication	Not applicable
ValidateService	Validates the availability and health of a Service Fabric service, usually after inducing some fault into the system.	ValidateServiceAsync	Test-ServiceFabricService	Not applicable

Running a testability action using PowerShell

This tutorial shows you how to run a testability action by using PowerShell. You will learn how to run a testability action against a local (one-box) cluster or an Azure cluster. Microsoft.Fabric.Powershell.dll--the Service Fabric PowerShell module--is installed automatically when you install the Microsoft Service Fabric MSI. The module is loaded automatically when you open a PowerShell prompt.

Tutorial segments:

- Run an action against a one-box cluster
- Run an action against an Azure cluster

Run an action against a one-box cluster

To run a testability action against a local cluster, first connect to the cluster and open the PowerShell prompt in administrator mode. Let us look at the **Restart-ServiceFabricNode** action.

```
Restart-ServiceFabricNode -NodeName Node1 -CompletionMode DoNotVerify
```

Here the action **Restart-ServiceFabricNode** is being run on a node named "Node1". The completion mode specifies that it should not verify whether the restart-node action actually succeeded. Specifying the completion mode as "Verify" will cause it to verify whether the restart action actually succeeded. Instead of directly specifying the node by its name, you can specify it via a partition key and the kind of replica, as follows:

```
Restart-ServiceFabricNode -ReplicaKindPrimary  -PartitionKindNamed -PartitionKey Partition3 -CompletionMode
Verify
```

```
$connection = "localhost:19000"
$nodeName = "Node1"

Connect-ServiceFabricCluster $connection
Restart-ServiceFabricNode -NodeName $nodeName -CompletionMode DoNotVerify
```

Restart-ServiceFabricNode should be used to restart a Service Fabric node in a cluster. This will stop the Fabric.exe process, which will restart all of the system service and user service replicas hosted on that node. Using this API to test your service helps uncover bugs along the failover recovery paths. It helps simulate node failures in the cluster.

The following screenshot shows the **Restart-ServiceFabricNode** testability command in action.

The output of the first **Get-ServiceFabricNode** (a cmdlet from the Service Fabric PowerShell module) shows that the local cluster has five nodes: Node.1 to Node.5. After the testability action (cmdlet) **Restart-ServiceFabricNode** is executed on the node, named Node.4, we see that the node's uptime has been reset.

Run an action against an Azure cluster

Running a testability action (by using PowerShell) against an Azure cluster is similar to running the action against a local cluster. The only difference is that before you can run the action, instead of connecting to the local cluster, you need to connect to the Azure cluster first.

Running a testability action using C#

To run a testability action by using C#, first you need to connect to the cluster by using FabricClient. Then obtain the parameters needed to run the action. Different parameters can be used to run the same action. Looking at the RestartServiceFabricNode action, one way to run it is by using the node information (node name and node instance ID) in the cluster.

```
RestartNodeAsync(nodeName, nodeInstanceId, completeMode, operationTimeout, CancellationToken.None)
```

Parameter explanation:

- **CompleteMode** specifies that the mode should not verify whether the restart action actually succeeded. Specifying the completion mode as "Verify" will cause it to verify whether the restart action actually succeeded.
- **OperationTimeout** sets the amount of time for the operation to finish before a TimeoutException exception is thrown.
- **CancellationToken** enables a pending call to be canceled.

Instead of directly specifying the node by its name, you can specify it via a partition key and the kind of replica.

For further information, see PartitionSelector and ReplicaSelector.

```
// Add a reference to System.Fabric.Testability.dll and System.Fabric.dll
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Fabric.Testability;
using System.Fabric;
using System.Threading;
using System.Numerics;

class Test
{
    public static int Main(string[] args)
    {
        string clusterConnection = "localhost:19000";
        Uri serviceName = new Uri("fabric:/samples/PersistentToDoListApp/PersistentToDoListService");
        string nodeName = "N0040";
        BigInteger nodeInstanceId = 130743013389060139;

        Console.WriteLine("Starting RestartNode test");
        try
        {
            //Restart the node by using ReplicaSelector
            RestartNodeAsync(clusterConnection, serviceName).Wait();

            //Another way to restart node is by using nodeName and nodeInstanceId
            RestartNodeAsync(clusterConnection, nodeName, nodeInstanceId).Wait();
        }
        catch (AggregateException exAgg)
        {
            Console.WriteLine("RestartNode did not complete: ");
            foreach (Exception ex in exAgg.InnerExceptions)
            {
                if (ex is FabricException)
                {
                    Console.WriteLine("HResult: {0} Message: {1}", ex.HResult, ex.Message);
                }
            }
            return -1;
        }

        Console.WriteLine("RestartNode completed.");
        return 0;
    }

    static async Task RestartNodeAsync(string clusterConnection, Uri serviceName)
    {
        PartitionSelector randomPartitionSelector = PartitionSelector.RandomOf(serviceName);
        ReplicaSelector primaryofReplicaSelector = ReplicaSelector.PrimaryOf(randomPartitionSelector);

        // Create FabricClient with connection and security information here
        FabricClient fabricclient = new FabricClient(clusterConnection);
        await fabricclient.FaultManager.RestartNodeAsync(primaryofReplicaSelector, CompletionMode.Verify);
    }

    static async Task RestartNodeAsync(string clusterConnection, string nodeName, BigInteger nodeInstanceId)
    {
        // Create FabricClient with connection and security information here
        FabricClient fabricclient = new FabricClient(clusterConnection);
        await fabricclient.FaultManager.RestartNodeAsync(nodeName, nodeInstanceId, CompletionMode.Verify);
    }
}
```

PartitionSelector and ReplicaSelector

PartitionSelector

PartitionSelector is a helper exposed in testability and is used to select a specific partition on which to perform any of the testability actions. It can be used to select a specific partition if the partition ID is known beforehand. Or, you can provide the partition key and the operation will resolve the partition ID internally. You also have the option of selecting a random partition.

To use this helper, create the PartitionSelector object and select the partition by using one of the Select* methods. Then pass in the PartitionSelector object to the API that requires it. If no option is selected, it defaults to a random partition.

```
Uri serviceName = new Uri("fabric:/samples/InMemoryToDoListApp/InMemoryToDoListService");
Guid partitionIdGuid = new Guid("8fb7ebcc-56ee-4862-9cc0-7c6421e68829");
string partitionName = "Partition1";
Int64 partitionKeyUniformInt64 = 1;

// Select a random partition
PartitionSelector randomPartitionSelector = PartitionSelector.RandomOf(serviceName);

// Select a partition based on ID
PartitionSelector partitionSelectorById = PartitionSelector.PartitionIdOf(serviceName, partitionIdGuid);

// Select a partition based on name
PartitionSelector namedPartitionSelector = PartitionSelector.PartitionKeyOf(serviceName, partitionName);

// Select a partition based on partition key
PartitionSelector uniformIntPartitionSelector = PartitionSelector.PartitionKeyOf(serviceName,
partitionKeyUniformInt64);
```

ReplicaSelector

ReplicaSelector is a helper exposed in testability and is used to help select a replica on which to perform any of the testability actions. It can be used to select a specific replica if the replica ID is known beforehand. In addition, you have the option of selecting a primary replica or a random secondary. ReplicaSelector derives from PartitionSelector, so you need to select both the replica and the partition on which you wish to perform the testability operation.

To use this helper, create a ReplicaSelector object and set the way you want to select the replica and the partition. You can then pass it into the API that requires it. If no option is selected, it defaults to a random replica and random partition.

```
Guid partitionIdGuid = new Guid("8fb7ebcc-56ee-4862-9cc0-7c6421e68829");
PartitionSelector partitionSelector = PartitionSelector.PartitionIdOf(serviceName, partitionIdGuid);
long replicaId = 130559876481875498;

// Select a random replica
ReplicaSelector randomReplicaSelector = ReplicaSelector.RandomOf(partitionSelector);

// Select the primary replica
ReplicaSelector primaryReplicaSelector = ReplicaSelector.PrimaryOf(partitionSelector);

// Select the replica by ID
ReplicaSelector replicaByIdSelector = ReplicaSelector.ReplicaIdOf(partitionSelector, replicaId);

// Select a random secondary replica
ReplicaSelector secondaryReplicaSelector = ReplicaSelector.RandomSecondaryOf(partitionSelector);
```

Next steps

- Testability scenarios
- How to test your service

- Simulate failures during service workloads
- Service-to-service communication failures

Simulate failures during service workloads

11/15/2016 • 3 min to read • Edit on GitHub

Contributors

Anmol Ahuja • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Tanvir Tanviruzzaman • ShawnJackson • Jesse Benson

The testability scenarios in Azure Service Fabric enable developers to not worry about dealing with individual faults. There are scenarios, however, where an explicit interleaving of client workload and failures might be needed. The interleaving of client workload and faults ensures that the service is actually performing some action when failure happens. Given the level of control that testability provides, these could be at precise points of the workload execution. This induction of faults at different states in the application can find bugs and improve quality.

Sample custom scenario

This test shows a scenario that interleaves the business workload with graceful and ungraceful failures. The faults should be induced in the middle of service operations or compute for best results.

Let's walk through an example of a service that exposes four workloads: A, B, C, and D. Each corresponds to a set of workflows and could be compute, storage, or a mix. For the sake of simplicity, we will abstract out the workloads in our example. The different faults executed in this example are:

- RestartNode: Ungraceful fault to simulate a machine restart.
- RestartDeployedCodePackage: Ungraceful fault to simulate service host process crashes.
- RemoveReplica: Graceful fault to simulate replica removal.
- MovePrimary: Graceful fault to simulate replica moves triggered by the Service Fabric load balancer.

```
// Add a reference to System.Fabric.Testability.dll and System.Fabric.dll.

using System;
using System.Fabric;
using System.Fabric.Testability.Scenario;
using System.Threading;
using System.Threading.Tasks;

class Test
{
    public static int Main(string[] args)
    {
        // Replace these strings with the actual version for your cluster and application.
        string clusterConnection = "localhost:19000";
        Uri applicationName = new Uri("fabric:/samples/PersistentToDoListApp");
        Uri serviceName = new Uri("fabric:/samples/PersistentToDoListApp/PersistentToDoListService");

        Console.WriteLine("Starting Workload Test...");
        try
        {
            RunTestAsync(clusterConnection, applicationName, serviceName).Wait();
        }
        catch (AggregateException ae)
        {
            Console.WriteLine("Workload Test failed: ");
            foreach (Exception ex in ae.InnerExceptions)
            {
                if (ex is FabricException)
                {
                    Console.WriteLine("HResult: {0} Message: {1}", ex.HResult, ex.Message);
```

```
                Console.WriteLine("ExitCode: {0}, Message: {1}", exitCode, exitMessage);
                }
            }
            return -1;
        }

        Console.WriteLine("Workload Test completed successfully.");
        return 0;
    }

    public enum ServiceWorkloads
    {
        A,
        B,
        C,
        D
    }

    public enum ServiceFabricFaults
    {
        RestartNode,
        RestartCodePackage,
        RemoveReplica,
        MovePrimary,
    }

    public static async Task RunTestAsync(string clusterConnection, Uri applicationName, Uri serviceName)
    {
        // Create FabricClient with connection and security information here.
        FabricClient fabricClient = new FabricClient(clusterConnection);
        // Maximum time to wait for a service to stabilize.
        TimeSpan maxServiceStabilizationTime = TimeSpan.FromSeconds(120);

        // How many loops of faults you want to execute.
        uint testLoopCount = 20;
        Random random = new Random();

        for (var i = 0; i < testLoopCount; ++i)
        {
            var workload = SelectRandomValue<ServiceWorkloads>(random);
            // Start the workload.
            var workloadTask = RunWorkloadAsync(workload);

            // While the task is running, induce faults into the service. They can be ungraceful faults like
            // RestartNode and RestartDeployedCodePackage or graceful faults like RemoveReplica or MovePrimary.
            var fault = SelectRandomValue<ServiceFabricFaults>(random);

            // Create a replica selector, which will select a primary replica from the given service to test.
            var replicaSelector = ReplicaSelector.PrimaryOf(PartitionSelector.RandomOf(serviceName));
            // Run the selected random fault.
            await RunFaultAsync(applicationName, fault, replicaSelector, fabricClient);
            // Validate the health and stability of the service.
            await fabricClient.ServiceManager.ValidateServiceAsync(serviceName, maxServiceStabilizationTime);

            // Wait for the workload to finish successfully.
            await workloadTask;
        }
    }

    private static async Task RunFaultAsync(Uri applicationName, ServiceFabricFaults fault, ReplicaSelector
selector, FabricClient client)
    {
        switch (fault)
        {
            case ServiceFabricFaults.RestartNode:
                await client.ClusterManager.RestartNodeAsync(selector, CompletionMode.Verify);
                break;
            case ServiceFabricFaults.RestartCodePackage:
                await client.ApplicationManager.RestartDeployedCodePackageAsync(applicationName, selector,
CompletionMode.Verify);
```

```
CompletionMode.Verify);
                break;
            case ServiceFabricFaults.RemoveReplica:
                await client.ServiceManager.RemoveReplicaAsync(selector, CompletionMode.Verify, false);
                break;
            case ServiceFabricFaults.MovePrimary:
                await client.ServiceManager.MovePrimaryAsync(selector.PartitionSelector);
                break;
        }
    }

    private static Task RunWorkloadAsync(ServiceWorkloads workload)
    {
        throw new NotImplementedException();
        // This is where you trigger and complete your service workload.
        // Note that the faults induced while your service workload is running will
        // fault the primary service. Hence, you will need to reconnect to complete or check
        // the status of the workload.
    }

    private static T SelectRandomValue<T>(Random random)
    {
        Array values = Enum.GetValues(typeof(T));
        T workload = (T)values.GetValue(random.Next(values.Length));
        return workload;
    }
}
```

How to Invoke Data Loss on Services

11/15/2016 • 5 min to read • Edit on GitHub

Contributors

LMWF • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil

WARNING

This document describe how to cause data loss in your services, and should be used with care.

Introduction

You can invoke data loss on a partition of your Service Fabric Service by calling StartPartitionDataLossAsync(). This api uses the Fault Injection and Analysis Service to perform the work to cause data loss conditions.

Using the Fault Injection and Analysis Service

The Fault Injection and Analysis Service currently supports the following APIs in the chart below. The right side of the chart shows the corresponding PowerShell cmdlet. Please refer to the msdn documentation on each API for more information on each one.

C# API	POWERSHELL CMDLET
StartPartitionDataLossAsync	Start-ServiceFabricPartitionDataLoss
StartPartitionQuorumLossAsync	Start-ServiceFabricPartitionQuorumLoss
StartPartitionRestartAsync	Start-ServiceFabricPartitionRestart

Conceptual Overview of Running a Command

The Fault Injection and Analysis Service uses an asynchronous model where you start the command with one API, referred to as the "Start" API in this document, then checks the progress of this command using a "GetProgress" API until it has reached a terminal state, or until you cancel it. To start a command, call the "Start" API for the corresponding API. This API returns when the Fault Injection and Analysis Service has accepted the request. However, it does not indicate how far a command has run, or even if it has started yet. In order to check progress of a command, call the "GetProgress" API that corresponds to the "Start" API previously called. The "GetProgress" API will return an object indicating the current status of the command inside its State property. A command runs indefinitely until:

1. It completes successfully. If you call "GetProgress" on it in this case, the progress object's State will be Completed.
2. It encounters a fatal error. If you call "GetProgress" on it in this case, the progress object's State will be Faulted
3. You cancel it through the CancelTestCommandAsync API, or Stop-ServiceFabricTestCommand PowerShell cmdlet. If you call "GetProgress" on it in this case, the progress object's State will be either Cancelled or ForceCancelled, depending on an argument to that API. See the documentation for CancelTestCommandAsync for more details.

Details of Running a Command

In order to start a command, call the Start API with the expected arguments. All Start APIs have a Guid argument named operationId. You should keep track of the operationId argument, since it is used to track progress of this command. This must be passed into the "GetProgress" API in order to track progress of the command. The operationId must be unique.

After successfully calling the Start API, the GetProgress API should be called in a loop until the returned progress object's State property is Completed. All FabricTransientException's and OperationCanceledException's should be retried. When the command has reached a terminal state (Completed, Faulted, or Cancelled), the returned progress object's Result property will have additional information. If the state is Completed, Result.SelectedPartition.PartitionId will contain the partition id that was selected. Result.Exception will be null. If the state is Faulted, Result.Exception will have the reason the Fault Injection and Analysis Service faulted the command. Result.SelectedPartition.PartitionId will have the partition id that was selected. In some situations, the command may not have proceeded far enough to choose a partition. In that case, the PartitionId will be 0. If the state is Cancelled, Result.Exception will be null. Like the Faulted case, Result.SelectedPartition.PartitionId will have the partition id that was chosen, but if the command has not proceeded far enough to do so, it will be 0. Please also refer to the sample below.

The sample code below shows how to start then check progress on a command to cause data loss on a specific partition.

```
    static async Task PerformDataLossSample()
    {
        // Create a unique operation id for the command below
        Guid operationId = Guid.NewGuid();

        // Note: Use the appropriate overload for your configuration
        FabricClient fabricClient = new FabricClient();

        // The name of the target service
        Uri targetServiceName = new Uri("fabric:/MyService");

        // The id of the target partition inside the target service
        Guid targetPartitionId = new Guid("00000000-0000-0000-0000-000002233445");

        PartitionSelector partitionSelector = PartitionSelector.PartitionIdOf(targetServiceName,
targetPartitionId);

        // Start the command.  Retry OperationCanceledException and all FabricTransientException's.  Note when
StartPartitionDataLossAsync completes
        // successfully it only means the Fault Injection and Analysis Service has saved the intent to perform
this work.  It does not say anything about the progress
        // of the command.
        while (true)
        {
            try
            {
                await fabricClient.TestManager.StartPartitionDataLossAsync(operationId, partitionSelector,
DataLossMode.FullDataLoss).ConfigureAwait(false);
                break;
            }
            catch (OperationCanceledException)
            {
            }
            catch (FabricTransientException)
            {
            }

            await Task.Delay(TimeSpan.FromSeconds(1.0d)).ConfigureAwait(false);
        }

        PartitionDataLossProgress progress = null;
```

```
        // Poll the progress using GetPartitionDataLossProgressAsync until it is either Completed or Faulted.  In
this example, we're assuming
        // the command won't be cancelled.

        while (true)
        {
            try
            {
                progress = await
fabricClient.TestManager.GetPartitionDataLossProgressAsync(operationId).ConfigureAwait(false);
            }
            catch (OperationCanceledException)
            {
                continue;
            }
            catch (FabricTransientException)
            {
                continue;
            }

            if (progress.State == TestCommandProgressState.Completed)
            {
                Console.WriteLine("Command '{0}' completed successfully", operationId);

                // In a terminal state .Result.SelectedPartition.PartitionId will have the chosen partition
                Console.WriteLine("  Printing selected partition='{0}'",
progress.Result.SelectedPartition.PartitionId);
                break;
            }
            else if (progress.State == TestCommandProgressState.Faulted)
            {
                // If State is Faulted, the progress object's Result property's Exception property will have the
reason why.
                Console.WriteLine("Command '{0}' failed with '{1}'", operationId, progress.Result.Exception);
                break;
            }
            else
            {
                Console.WriteLine("Command '{0}' is currently Running", operationId);
            }

            await Task.Delay(TimeSpan.FromSeconds(5.0d)).ConfigureAwait(false);
        }
    }
```

The sample below shows how to use the PartitionSelector to choose a random partition of a specified service:

```
    static async Task PerformDataLossUseSelectorSample()
    {
        // Create a unique operation id for the command below
        Guid operationId = Guid.NewGuid();

        // Note: Use the appropriate overload for your configuration
        FabricClient fabricClient = new FabricClient();

        // The name of the target service
        Uri targetServiceName = new Uri("fabric:/SampleService ");

        // Use a PartitionSelector that will have the Fault Injection and Analysis Service choose a random
partition of “targetServiceName”
        PartitionSelector partitionSelector = PartitionSelector.RandomOf(targetServiceName);

        // Start the command.  Retry OperationCanceledException and all FabricTransientException's.  Note when
StartPartitionDataLossAsync completes
        // successfully it only means the Fault Injection and Analysis Service has saved the intent to perform
this work.  It does not say anything about the progress
        // of the command.
```

```
        while (true)
        {
            try
            {
                await fabricClient.TestManager.StartPartitionDataLossAsync(operationId, partitionSelector,
DataLossMode.FullDataLoss).ConfigureAwait(false);
                break;
            }
            catch (OperationCanceledException)
            {
            }
            catch (FabricTransientException)
            {
            }
            catch (Exception e)
            {
                Console.WriteLine("Unexpected exception '{0}'", e);
                throw;
            }

            await Task.Delay(TimeSpan.FromSeconds(1.0d)).ConfigureAwait(false);
        }

        PartitionDataLossProgress progress = null;

        // Poll the progress using GetPartitionDataLossProgressAsync until it is either Completed or Faulted.  In
this example, we're assuming
        // the command won't be cancelled.

        while (true)
        {
            try
            {
                progress = await
fabricClient.TestManager.GetPartitionDataLossProgressAsync(operationId).ConfigureAwait(false);
            }
            catch (OperationCanceledException)
            {
                continue;
            }
            catch (FabricTransientException)
            {
                continue;
            }

            if (progress.State == TestCommandProgressState.Completed)
            {
                Console.WriteLine("Command '{0}' completed successfully", operationId);

                Console.WriteLine("Printing progress.Result:");
                Console.WriteLine("  Printing selected partition='{0}'",
progress.Result.SelectedPartition.PartitionId);

                break;
            }
            else if (progress.State == TestCommandProgressState.Faulted)
            {
                // If State is Faulted, the progress object's Result property's Exception property will have the
reason why.
                Console.WriteLine("Command '{0}' failed with '{1}', SelectedPartition {2}", operationId,
progress.Result.Exception, progress.Result.SelectedPartition);
                break;
            }
            else
            {
                Console.WriteLine("Command '{0}' is currently Running", operationId);
            }

            await Task.Delay(TimeSpan.FromSeconds(1.0d)).ConfigureAwait(false);
```

```
            await Task.Delay(TimeSpan.FromSeconds(1.0d).ConfigureAwait(false);
        }
    }
```

History and Truncation

After a command has reached a terminal state, its metadata will remain in the Fault Injection and Analysis Service for a certain time, before it will be removed to save space. If “GetProgress” is called using the operationId of a command after it has been removed, it will return a FabricException with an ErrorCode of KeyNotFound.

Testability scenarios

11/15/2016 • 6 min to read • Edit on GitHub

Contributors

Tanvir Tanviruzzaman • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Vaclav Turecek • Anmol Ahuja • ShawnJackson • Jesse Benson

Large distributed systems like cloud infrastructures are inherently unreliable. Azure Service Fabric gives developers the ability to write services to run on top of unreliable infrastructures. In order to write high-quality services, developers need to be able to induce such unreliable infrastructure to test the stability of their services.

The Fault Analysis Service gives developers the ability to induce fault actions to test services in the presence of failures. However, targeted simulated faults will get you only so far. To take the testing further, you can use the test scenarios in Service Fabric: a chaos test and a failover test. These scenarios simulate continuous interleaved faults, both graceful and ungraceful, throughout the cluster over extended periods of time. Once a test is configured with the rate and kind of faults, it can be started through either C# APIs or PowerShell, to generate faults in the cluster and your service.

> **WARNING**
> ChaosTestScenario is being replaced by a more resilient, service-based Chaos. Please refer to the new article Controlled Chaos for more details.

Chaos test

The chaos scenario generates faults across the entire Service Fabric cluster. The scenario compresses faults generally seen in months or years to a few hours. The combination of interleaved faults with the high fault rate finds corner cases that are otherwise missed. This leads to a significant improvement in the code quality of the service.

Faults simulated in the chaos test

- Restart a node
- Restart a deployed code package
- Remove a replica
- Restart a replica
- Move a primary replica (optional)
- Move a secondary replica (optional)

The chaos test runs multiple iterations of faults and cluster validations for the specified period of time. The time spent for the cluster to stabilize and for validation to succeed is also configurable. The scenario fails when you hit a single failure in cluster validation.

For example, consider a test set to run for one hour with a maximum of three concurrent faults. The test will induce three faults, and then validate the cluster health. The test will iterate through the previous step till the cluster becomes unhealthy or one hour passes. If the cluster becomes unhealthy in any iteration, i.e. it does not stabilize within a configured time, the test will fail with an exception. This exception indicates that something has gone wrong and needs further investigation.

In its current form, the fault generation engine in the chaos test induces only safe faults. This means that in the

absence of external faults, a quorum or data loss will never occur.

Important configuration options

- **TimeToRun**: Total time that the test will run before finishing with success. The test can finish earlier in lieu of a validation failure.
- **MaxClusterStabilizationTimeout**: Maximum amount of time to wait for the cluster to become healthy before failing the test. The checks performed are whether cluster health is OK, service health is OK, the target replica set size is achieved for the service partition, and no InBuild replicas exist.
- **MaxConcurrentFaults**: Maximum number of concurrent faults induced in each iteration. The higher the number, the more aggressive the test, hence resulting in more complex failovers and transition combinations. The test guarantees that in absence of external faults there will not be a quorum or data loss, irrespective of how high this configuration is.
- **EnableMoveReplicaFaults**: Enables or disables the faults that are causing the move of the primary or secondary replicas. These faults are disabled by default.
- **WaitTimeBetweenIterations**: Amount of time to wait between iterations, i.e. after a round of faults and corresponding validation.

How to run the chaos test

C# sample

```
using System;
using System.Fabric;
using System.Fabric.Testability.Scenario;
using System.Threading;
using System.Threading.Tasks;

class Test
{
    public static int Main(string[] args)
    {
        string clusterConnection = "localhost:19000";

        Console.WriteLine("Starting Chaos Test Scenario...");
        try
        {
            RunChaosTestScenarioAsync(clusterConnection).Wait();
        }
        catch (AggregateException ae)
        {
            Console.WriteLine("Chaos Test Scenario did not complete: ");
            foreach (Exception ex in ae.InnerExceptions)
            {
                if (ex is FabricException)
                {
                    Console.WriteLine("HResult: {0} Message: {1}", ex.HResult, ex.Message);
                }
            }
            return -1;
        }

        Console.WriteLine("Chaos Test Scenario completed.");
        return 0;
    }

    static async Task RunChaosTestScenarioAsync(string clusterConnection)
    {
        TimeSpan maxClusterStabilizationTimeout = TimeSpan.FromSeconds(180);
        uint maxConcurrentFaults = 3;
        bool enableMoveReplicaFaults = true;

        // Create FabricClient with connection and security information here.
        FabricClient fabricClient = new FabricClient(clusterConnection);
```

```
        // The chaos test scenario should run at least 60 minutes or until it fails.
        TimeSpan timeToRun = TimeSpan.FromMinutes(60);
        ChaosTestScenarioParameters scenarioParameters = new ChaosTestScenarioParameters(
          maxClusterStabilizationTimeout,
          maxConcurrentFaults,
          enableMoveReplicaFaults,
          timeToRun);

        // Other related parameters:
        // Pause between two iterations for a random duration bound by this value.
        // scenarioParameters.WaitTimeBetweenIterations = TimeSpan.FromSeconds(30);
        // Pause between concurrent actions for a random duration bound by this value.
        // scenarioParameters.WaitTimeBetweenFaults = TimeSpan.FromSeconds(10);

        // Create the scenario class and execute it asynchronously.
        ChaosTestScenario chaosScenario = new ChaosTestScenario(fabricClient, scenarioParameters);

        try
        {
            await chaosScenario.ExecuteAsync(CancellationToken.None);
        }
        catch (AggregateException ae)
        {
            throw ae.InnerException;
        }
    }
}
```

PowerShell

```
$connection = "localhost:19000"
$timeToRun = 60
$maxStabilizationTimeSecs = 180
$concurrentFaults = 3
$waitTimeBetweenIterationsSec = 60

Connect-ServiceFabricCluster $connection

Invoke-ServiceFabricChaosTestScenario -TimeToRunMinute $timeToRun -MaxClusterStabilizationTimeoutSec
$maxStabilizationTimeSecs -MaxConcurrentFaults $concurrentFaults -EnableMoveReplicaFaults -
WaitTimeBetweenIterationsSec $waitTimeBetweenIterationsSec
```

Failover test

The failover test scenario is a version of the chaos test scenario that targets a specific service partition. It tests the effect of failover on a specific service partition while leaving the other services unaffected. Once it's configured with the target partition information and other parameters, it runs as a client-side tool that uses either C# APIs or PowerShell to generate faults for a service partition. The scenario iterates through a sequence of simulated faults and service validation while your business logic runs on the side to provide a workload. A failure in service validation indicates an issue that needs further investigation.

Faults simulated in the failover test

- Restart a deployed code package where the partition is hosted
- Remove a primary/secondary replica or stateless instance
- Restart a primary secondary replica (if a persisted service)
- Move a primary replica
- Move a secondary replica
- Restart the partition

The failover test induces a chosen fault and then runs validation on the service to ensure its stability. The failover

test induces only one fault at a time, as opposed to possible multiple faults in the chaos test. If the service partition does not stabilize within the configured timeout after each fault, the test fails. The test induces only safe faults. This means that in absence of external failures, a quorum or data loss will not occur.

Important configuration options

- **PartitionSelector**: Selector object that specifies the partition that needs to be targeted.
- **TimeToRun**: Total time that the test will run before finishing.
- **MaxServiceStabilizationTimeout**: Maximum amount of time to wait for the cluster to become healthy before failing the test. The checks performed are whether service health is OK, the target replica set size is achieved for all partitions, and no InBuild replicas exist.
- **WaitTimeBetweenFaults**: Amount of time to wait between every fault and validation cycle.

How to run the failover test

C#

```
using System;
using System.Fabric;
using System.Fabric.Testability.Scenario;
using System.Threading;
using System.Threading.Tasks;

class Test
{
    public static int Main(string[] args)
    {
        string clusterConnection = "localhost:19000";
        Uri serviceName = new Uri("fabric:/samples/PersistentToDoListApp/PersistentToDoListService");

        Console.WriteLine("Starting Chaos Test Scenario...");
        try
        {
            RunFailoverTestScenarioAsync(clusterConnection, serviceName).Wait();
        }
        catch (AggregateException ae)
        {
            Console.WriteLine("Chaos Test Scenario did not complete: ");
            foreach (Exception ex in ae.InnerExceptions)
            {
                if (ex is FabricException)
                {
                    Console.WriteLine("HResult: {0} Message: {1}", ex.HResult, ex.Message);
                }
            }
            return -1;
        }

        Console.WriteLine("Chaos Test Scenario completed.");
        return 0;
    }

    static async Task RunFailoverTestScenarioAsync(string clusterConnection, Uri serviceName)
    {
        TimeSpan maxServiceStabilizationTimeout = TimeSpan.FromSeconds(180);
        PartitionSelector randomPartitionSelector = PartitionSelector.RandomOf(serviceName);

        // Create FabricClient with connection and security information here.
        FabricClient fabricClient = new FabricClient(clusterConnection);

        // The chaos test scenario should run at least 60 minutes or until it fails.
        TimeSpan timeToRun = TimeSpan.FromMinutes(60);
        FailoverTestScenarioParameters scenarioParameters = new FailoverTestScenarioParameters(
          randomPartitionSelector,
          timeToRun,
          maxServiceStabilizationTimeout);
```

```
        // Other related parameters:
        // Pause between two iterations for a random duration bound by this value.
        // scenarioParameters.WaitTimeBetweenIterations = TimeSpan.FromSeconds(30);
        // Pause between concurrent actions for a random duration bound by this value.
        // scenarioParameters.WaitTimeBetweenFaults = TimeSpan.FromSeconds(10);

        // Create the scenario class and execute it asynchronously.
        FailoverTestScenario failoverScenario = new FailoverTestScenario(fabricClient, scenarioParameters);

        try
        {
            await failoverScenario.ExecuteAsync(CancellationToken.None);
        }
        catch (AggregateException ae)
        {
            throw ae.InnerException;
        }
    }
}
```

PowerShell

```
$connection = "localhost:19000"
$timeToRun = 60
$maxStabilizationTimeSecs = 180
$waitTimeBetweenFaultsSec = 10
$serviceName = "fabric:/SampleApp/SampleService"

Connect-ServiceFabricCluster $connection

Invoke-ServiceFabricFailoverTestScenario -TimeToRunMinute $timeToRun -MaxServiceStabilizationTimeoutSec
$maxStabilizationTimeSecs -WaitTimeBetweenFaultsSec $waitTimeBetweenFaultsSec -ServiceName $serviceName -
PartitionKindSingleton
```

Load test your application by using Visual Studio Team Services

11/22/2016 • 5 min to read • Edit on GitHub

Contributors

Catherine Wang • Andy Pasic • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • stevehug • Andy De George

This article shows how to use Microsoft Visual Studio load test features to stress test an application. It uses an Azure Service Fabric stateful service back end and a stateless service web front end. The example application used here is an airplane location simulator. You provide an airplane ID, departure time, and destination. The application's back end processes the requests, and the front end displays on a map the airplane that matches the criteria.

The following diagram illustrates the Service Fabric application that you'll be testing.

Prerequisites

Before getting started, you need to do the following:

- Get a Visual Studio Team Services account. You can get one for free at Visual Studio Team Services.
- Get and install Visual Studio 2013 or Visual Studio 2015. This article uses Visual Studio 2015 Enterprise edition, but Visual Studio 2013 and other editions should work similarly.
- Deploy your application to a staging environment. See How to deploy applications to a remote cluster using Visual Studio for information about this.
- Understand your application's usage pattern. This information is used to simulate the load pattern.
- Understand the goal for your load testing. This helps you interpret and analyze the load test results.

Create and run the Web Performance and Load Test project

Create a Web Performance and Load Test project

1. Open Visual Studio 2015. Choose **File** > **New** > **Project** on the menu bar to open the **New Project** dialog box.

2. Expand the **Visual C#** node and choose **Test** > **Web Performance and Load Test project**. Give the project a name and then choose the **OK** button.

You should see a new Web Performance and Load Test project in Solution Explorer.

Record a web performance test

1. Open the .webtest project.
2. Choose the **Add Recording** icon to start a recording session in your browser.

3. Browse to the Service Fabric application. The recording panel should show the web requests.

4. Perform a sequence of actions that you expect the users to perform. These actions are used as a pattern to generate the load.
5. When you're done, choose the **Stop** button to stop recording.

The .webtest project in Visual Studio should have captured a series of requests. Dynamic parameters are replaced automatically. At this point, you can delete any extra, repeated dependency requests that are not part of your test scenario.

6. Save the project and then choose the **Run Test** command to run the web performance test locally and make sure everything works correctly.

Parameterize the web performance test

You can parameterize the web performance test by converting it to a coded web performance test and then editing the code. As an alternative, you can bind the web performance test to a data list so that the test iterates through the data. See Generate and run a coded web performance test for details about how to convert the web performance test to a coded test. See Add a data source to a web performance test for information about how to bind data to a web performance test.

For this example, we'll convert the web performance test to a coded test so you can replace the airplane ID with a generated GUID and add more requests to send flights to different locations.

Create a load test project

A load test project is composed of one or more scenarios described by the web performance test and unit test, along with additional specified load test settings. The following steps show how to create a load test project:

1. On the shortcut menu of your Web Performance and Load Test project, choose **Add** > **Load Test**. In the **Load Test** wizard, choose the **Next** button to configure the test settings.
2. In the **Load Pattern** section, choose whether you want a constant user load or a step load, which starts with a few users and increases the users over time.

   If you have a good estimate of the amount of user load and want to see how the current system performs, choose **Constant Load**. If your goal is to learn whether the system performs consistently under various loads, choose **Step Load**.

3. In the **Test Mix** section, choose the **Add** button and then select the test that you want to include in the load test. You can use the **Distribution** column to specify the percentage of total tests run for each test.
4. In the **Run Settings** section, specify the load test duration.

   > **NOTE**
   >
   > The **Test Iterations** option is available only when you run a load test locally using Visual Studio.

5. In the **Location** section of **Run Settings**, specify the location where load test requests are generated. The wizard may prompt you to log in to your Team Services account. Log in and then choose a geographic location. When you're done, choose the **Finish** button.
6. After the load test is created, open the .loadtest project and choose the current run setting, such as **Run Settings** > **Run Settings1 [Active]**. This opens the run settings in the **Properties** window.
7. In the **Results** section of the **Run Settings** properties window, the **Timing Details Storage** setting should have **None** as its default value. Change this value to **All Individual Details** to get more information on the load test results. See Load Testing for more information on how to connect to Visual Studio Team Services and run a load test.

Run the load test by using Visual Studio Team Services

Choose the **Run Load Test** command to start the test run.

View and analyze the load test results

As the load test progresses, the performance information is graphed. You should see something similar to the following graph.

1. Choose the **Download report** link near the top of the page. After the report is downloaded, choose the **View report** button.

   On the **Graph** tab you can see graphs for various performance counters. On the **Summary** tab, the overall test results appear. The **Tables** tab shows the total number of passed and failed load tests.

2. Choose the number links on the **Test** > **Failed** and the **Errors** > **Count** columns to see error details.

   The **Detail** tab shows virtual user and test scenario information for failed requests. This data can be useful if the load test includes multiple scenarios.

See Analyzing Load Test Results in the Graphs View of the Load Test Analyzer for more information on viewing load test results.

Automate your load test

Visual Studio Team Services Load Test provides APIs to help you manage load tests and analyze results in a Team Services account. See Cloud Load Testing Rest APIs for more information.

Next steps

- Monitoring and diagnosing services in a local machine development setup

Service Fabric terminology overview

11/15/2016 • 8 min to read • Edit on GitHub

Contributors

unknown • Kim Whitlatch (Beyondsoft Corporation) • Tyson Nevil • Alexander Zeitler • Adriano Arce • Rick Gregory • Mark Fussell • Gutemberg Ribeiro • MarkDeichmiller • Jesse Benson

Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices. This topic details the terminology used by Service Fabric in order to understand the terms used in the documentation.

Infrastructure concepts

Cluster: A network-connected set of virtual or physical machines into which your microservices are deployed and managed. Clusters can scale to thousands of machines.

Node: A machine or VM that is part of a cluster is called a node. Each node is assigned a node name (a string). Nodes have characteristics such as placement properties. Each machine or VM has an auto-start Windows service, `FabricHost.exe`, which starts running upon boot and then starts two executables: `Fabric.exe` and `FabricGateway.exe`. These two executables make up the node. For testing scenarios, you can host multiple nodes on a single machine or VM by running multiple instances of `Fabric.exe` and `FabricGateway.exe`.

Application concepts

Application Type: The name/version assigned to a collection of service types. Defined in an `ApplicationManifest.xml` file, embedded in an application package directory, which is then copied to the Service Fabric cluster's image store. You can then create a named application from this application type within the cluster.

Read the Application Model article for more information.

Application Package: A disk directory containing the application type's `ApplicationManifest.xml` file. References the service packages for each service type that makes up the application type. The files in the application package directory are copied to Service Fabric cluster's image store. For example, an application package for an email application type could contain references to a queue service package, a frontend service package, and a database service package.

Named Application: After an application package is copied to the image store, you create an instance of the application within the cluster by specifying the application package's application type (using its name/version). Each application type instance is assigned a URI name that looks like this: `"fabric:/MyNamedApp"`. Within a cluster, you can create multiple named applications from a single application type. You can also create named applications from different application types. Each named application is managed and versioned independently.

Service Type: The name/version assigned to a service's code packages, data packages, and configuration packages. Defined in a `ServiceManifest.xml` file, embedded in a service package directory and the service package directory is then referenced by an application package's `ApplicationManifest.xml` file. Within the cluster, after creating a named application, you can create a named service from one of the application type's service types. The service type's `ServiceManifest.xml` file describes the service.

Read the Application Model article for more information.

There are two types of services:

- **Stateless:** Use a stateless service when the service's persistent state is stored in an external storage service such as Azure Storage, Azure SQL Database, or Azure DocumentDB. Use a stateless service when the service has no persistent storage at all. For example, a calculator service where values are passed to the service, a computation is performed using these values, and a result is returned.
- **Stateful:** Use a stateful service when you want Service Fabric to manage your service's state via its Reliable Collections or Reliable Actors programming models. Specify how many partitions you want to spread your state over (for scalability) when creating a named service. Also specify how many times to replicate your state across nodes (for reliability). Each named service has a single primary replica and multiple secondary replicas. You modify your named service's state by writing to the primary replica. Service Fabric then replicates this state to all the secondary replicas keeping your state in sync. Service Fabric automatically detects when a primary replica fails and promotes an existing secondary replica to a primary replica. Service Fabric then creates a new secondary replica.

Service Package: A disk directory containing the service type's `ServiceManifest.xml` file. This file references the code, static data, and configuration packages for the service type. The files in the service package directory are referenced by the application type's `ApplicationManifest.xml` file. For example, a service package could refer to the code, static data, and configuration packages that make up a database service.

Named Service: After creating a named application, you can create an instance of one of its service types within the cluster by specifying the service type (using its name/version). Each service type instance is assigned a URI name scoped under its named application's URI. For example, if you create a "MyDatabase" named service within a "MyNamedApp" named application, the URI looks like: `"fabric:/MyNamedApp/MyDatabase"`. Within a named application, you can create several named services. Each named service can have its own partition scheme and instance/replica counts.

Code Package: A disk directory containing the service type's executable files (typically EXE/DLL files). The files in the code package directory are referenced by the service type's `ServiceManifest.xml` file. When a named service is created, the code package is copied to the one or more nodes selected to run the named service. Then the code starts running. There are two types of code package executables:

- **Guest executables**: Executables that run as-is on the host operating system (Windows or Linux). That is, these executables do not link to or reference any Service Fabric runtime files and therefore do not use any Service Fabric programming models. These executables are unable to use some Service Fabric features such as the naming service for endpoint discovery. Guest executables cannot report load metrics specific to each service instance.
- **Service Host Executables**: Executables that use Service Fabric programming models by linking to Service Fabric runtime files, enabling Service Fabric features. For example, a named service instance can register endpoints with Service Fabric's Naming Service and can also report load metrics.

Data Package: A disk directory containing the service type's static, read-only data files (typically photo, sound, and video files). The files in the data package directory are referenced by the service type's `ServiceManifest.xml` file. When a named service is created, the data package is copied to the one or more nodes selected to run the named service. The code starts running and can now access the data files.

Configuration Package: A disk directory containing the service type's static, read-only configuration files (typically text files). The files in the configuration package directory are referenced by the service type's `ServiceManifest.xml` file. When a named service is created, the files in the configuration package are copied to the one or more nodes selected to run the named service. Then the code starts running and can now access the configuration files.

Partition Scheme: When creating a named service, you specify a partition scheme. Services with large amounts of state split the data across partitions which spreads it across the cluster's nodes. This allows your named service's state to scale. Within a partition, stateless named services have instances while stateful named services have replicas. Usually, stateless named services only ever have one partition since they have no internal state. The partition instances provide for availability; if one instance fails, other instances continue to operate normally and

then Service Fabric will create a new instance. Stateful named services maintain their state within replicas and each partition has its own replica set with all the state being kept in sync. Should a replica fail, Service Fabric builds a new replica from the existing replicas.

Read the Partition Service Fabric reliable services article for more information.

System services

There are system services that are created in every cluster that provide the platform capabilities of Service Fabric.

Naming Service: Each Service Fabric cluster has a Naming service, which resolves service names to a location in the cluster. You manage the service names and properties, similar to an internet Domain Name Service (DNS) for the cluster. Clients securely communicate with any node in the cluster using the Naming Service to resolve a service name and its location. Clients obtain the actual machine IP address and port where it is currently running. You can develop services and clients capable of resolving the current network location despite applications being moved within the cluster for example due to failures, resource balancing, or the resizing of the cluster.

Read Communicate with services for more information on the client and service communication APIs that work with the Naming service.

Image Store Service: Each Service Fabric cluster has an Image Store service where deployed, versioned application packages are kept. Copy an application package to the Image Store and then register the application type contained within that application package. After the application type is provisioned, you create a named applications from it. You can unregister an application type from the Image Store service after all its named applications have been deleted.

Read the Deploy an application article for more information on deploying applications to the Image store service.

Built-in programming models

There are .NET Framework programming models available for you to build Service Fabric services:

Reliable Services: An API to build stateless and stateful services. Stateful service store their state in Reliable Collections (such as a dictionary or a queue). You also get to plug in a variety of communication stacks such as Web API and Windows Communication Foundation (WCF).

Reliable Actors: An API to build stateless and stateful objects through the virtual Actor programming model. This model can be useful when you have lots of independent units of computation/state. Because this model uses a turn-based threading model, it is best to avoid code that calls out to other actors or services since an individual actor cannot process other incoming requests until all its outbound requests have completed.

Read the Choose a Programming Model for your service article for more information.

Next steps

To learn more about Service Fabric:

- Overview of Service Fabric
- Why a microservices approach to building applications?
- Application scenarios