Namit Tanasseri, Rahul Rai

Microservices with Azure

Software architecture patterns to build enterprise-grade Microservices using Microsoft Azure Service Fabric

Packt

Microservices with Azure

Software architecture patterns to build enterprise-grade Microservices using Microsoft Azure Service Fabric

Namit Tanasseri
Rahul Rai

Packt

BIRMINGHAM - MUMBAI

Microservices with Azure

First published: June 2017

Production reference: 1240617

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78712-114-0

www.packtpub.com

Credits

Authors
Namit Tanasseri
Rahul Rai

Copy Editor
Dipti Mankame

Reviewers
Manish Sharma
Paul Glavich
Roberto Freato

Project Coordinator
Judie Jose

Proofreader
Safis Editing

Acquisition Editor
Divya Poojari

Indexer
Rekha Nair

Content Development Editor
Abhishek Jadhav

Graphics
Kirk D'Penha

Technical Editor
Mohd Riyan Khan

Production Coordinator
Aparna Bhagat

About the Authors

Namit Tanasseri is a certified Microsoft Cloud Solutions Architect with more than 11 years of work experience. He started his career as a Software Development Engineer with Microsoft Research and Development Center in 2005. During the first five years of his career, he had opportunities to work with major Microsoft product groups such as Microsoft Office and Windows. The experience working with the largest development teams within Microsoft helped him strengthen his knowledge of agile software development methodologies and processes. He also earned a patent award for a technology invention during this tenure.

Namit joined Microsoft Services chasing his passion for delivering solutions on cloud technologies and for gaining better exposure to technology consulting. As a technology consultant with Microsoft, Namit worked with Microsoft Azure Services for four years. Namit was a worldwide subject matter expert in Microsoft Azure, and he actively contributed to the Microsoft cloud community, apart from delivering top quality solutions for Microsoft customers. Namit also led the Windows Azure community in Microsoft Services India. Namit currently serves as a Microsoft Cloud Solutions Architect from Sydney, Australia working on large and medium sized enterprise engagements.

To my mom, Leena, and my wife, Poonam, for their patience during the writing of this book.

I would like to portray my gratitude to Microsoft for cultivating the curiosity about technology and innovation within me.

Special thanks to my friends for all the good days, the memories of which help me sail through difficult times.
Finally, I cannot forget to thank God and my grandpa (who sits beside him in Heaven) for gifting me the skills and patience for completing this book.

Rahul Rai is a technology consultant based in Sydney, Australia with over nine years of professional experience. He has been at the forefront of cloud consulting for government organizations and businesses around the world.

Rahul has been working on Microsoft Azure since the service was in its infancy, delivering an ITSM tool built for and on Azure in 2008. Since then, Rahul has played the roles of a developer, a consultant, and an architect for enterprises ranging from small start-ups to multinational corporations.

Rahul has worked for over five years with Microsoft Services, where he worked with diverse teams to deliver innovative solutions on Microsoft Azure. In Microsoft, Rahul was a worldwide Subject Matter Expert in Microsoft cloud technologies. Rahul has also worked as a Cloud Solution Architect for Microsoft, for which he worked closely with some established Microsoft partners to drive joint customer transformations to cloud-based architectures. He loves contributing to community initiatives and speaks at some renowned conferences such as Tech Days and Ignite. Rahul is a contributor to Azure Open Source initiatives and has authored several MSDN articles and publications on Azure.

I dedicate this book to my mother and my father, Indra and Ghanshyam Rai, who have always been there for me.

I would like to thank my wife, Neha, without whose support this book would not have been possible. I would also like to thank my lovely sister, Mohini, my brother-in-law, Ajay and, my friends who have supported me in this authorial journey.

Thank you, God, for taking care of me every day and giving me the gift of writing. Finally, I would like to thank the brilliant technical community which has never failed to inspire and drive me with its incredible knowledge and insatiable thirst for learning.

About the Reviewers

Roberto Freato has been an independent IT consultant since he started to work for small software factories while he was studying, after the MSc in computer science. With a thesis about Consumer Cloud Computing, he got specialization on Cloud and Azure. Today, he works as a freelance consultant for major companies in Italy, helping clients to design and kick-off their distributed software solutions. He trains for the developer community in his free time, speaking in many conferences. He is a Microsoft MVP since 2010.

Manish Sharma works with Microsoft as a solution architect as part of Microsoft Services. As a solution architect, he is responsible for leading large enterprise transformational engagements defining technology/solution roadmaps, conducting architecture and technical evaluations of enterprise engagements and architecting mutli-million-dollar solution developments and maintenance outsourcing managements. He is also a technology evangelist and speaker in prestigious events such as Microsoft TechEd, on latest and cutting-edge technologies such as HoloLens, Internet of Things, Connected Car, and Cloud technologies such as Azure.

Paul Glavich was an ASP.NET MVP for 13 years and currently works as a principal consultant for Readify. Previously, he was a chief technology officer (CTO) for Saasu (saasu.com), solution architect at Datacom, then senior consultant for readify, and prior to that a technical architect for EDS, Australia. He has over 20 years of industry experience ranging from PICK, C, C++, Delphi, and Visual Basic 3/4/5/6 to his current specialty in .Net with C#, ASP.NET, Azure, Cloud and DevOps.

Paul has been developing in .Net technologies since .Net was first in Beta and was the technical architect for one of the world's first Internet banking solutions using .Net technology.

Paul can be seen on various .Net-related newsgroups, has presented at the Sydney .Net user group (www.sdnug.org) and TechEd, and is also a member of ASPInsiders (www.aspinsiders.com). He has also written some technical articles, which can be seen on community sites such as ASPAlliance.com (www.aspalliance.com). Paul has authored a total of three books, Beginning AJAX in ASP.NET, Beginning Microsoft ASP.NET AJAX, and the latest book on .Net Performance Testing and Optimization. He is currently focusing on overall architecture, solution design, and Microsoft Cloud solutions.

On a more personal note, Paul is married with three children, three grandkids, holds a fifth degree black belt in Budo-Jitsu, and also practices Wing Chun Kung fu.

There are so many people who have helped me get to where I am today, but it would take another book. So to keep things short, I would like to thank my three children Kristy, Marc, and Elizabeth for being awesome, my parents for buying me that first computer, my nerd friends for nerding out with me, but mostly I would like to thank my wife, Michele, for supporting me in my life and career, and enduring my never-ending stream of technobabble and Dad jokes.

Customer Feedback

Table of Contents

Preface

Let’s take a drive down the memory lane. Imagine it is the 90’s, and you have been asked to build an application for an enterprise. You go to an IT shop and procure a server (or ask someone to do it for you), after several days or weeks the server shows up. Next, you configure the server, the network and the many settings it has. Finally, you deploy your application on the server and turn on the lights, and everyone is happy.

Business requirements change and now your customer needs some changes to be made to the application. So you go back make the changes, take the now upgraded application package and dump it on the server. You shut down the machine and bring it back up to bring up your refreshed application. You make compromises with downtime but are happy with the convenience of bundling everything into a single package and pushing it to the server.

So, in this early period, the agility of hardware limited the agility of software. It wasn’t easy to scale infrastructure and therefore altering the systems at the snap of fingers wasn’t a possibility. Moreover, during those times internet applications were more of a value addition than a necessity. Thus, traditional hardware agility supported monolithic architecture, which requires building applications as one big bundle of tightly coupled services.

Fast forward a couple of years and the advent of cloud changed the game. With the cloud, infrastructure has become easy to provision on-demand, and it also eliminates any waste of developer time that traditionally went into preparing the servers to host applications. Today software is a differentiator and not a value addition for any enterprise. Owing to the competition, high frequency of change, and continually changing customer requirements require decoupling an application to make it easy to scale, test and deploy.

The advent of cloud and the need for applications to be decoupled into smaller components that fulfill a single responsibility (Single Responsibility Principle) for the ease of development, deployment, and scale lead to the rise of the Microservice architecture.

However, breaking down a monolith into small independent services comes with the overhead of management of these services. This need of managing the Microservices gave birth to Azure Service Fabric which apart from its many services acts as a cluster manager. Azure Service Fabric Cluster Manager governs a set of virtual machines and handles Microservice deployment, placement, and failover. Azure Service Fabric also governs the application model and has capabilities to drive Application Lifecycle Management (ALM).

Microsoft itself has used Service Fabric for many years to host its planet-scale services such as Azure SQL Database, Azure IoT hub, Skype for Business, Cortana, and Intune.

This book introduces its readers to the concept of Microservices and Microsoft Azure Service Fabric as a distributed platform to host enterprise-grade Microservices. Learning the concepts of technology is often not sufficient. In practical scenarios, developers and architects often search for guidance on some of the most common design challenges. This book addresses common architectural challenges associated with the Microservice architecture, using proven architectural patterns.

What this book covers

`Chapter 1`, *Microservices – Getting to Know the Buzzword*, lays the foundation of concepts of Microservices and explores the scenarios, where Microservices are best suited for your application.

`Chapter 2`, *Microsoft Azure Platform and Services Primer*, provides a very fast-paced overview of Microsoft Azure as a platform for hosting internet-scale applications.

`Chapter 3`, *Understanding Azure Service Fabric*, explains the basic concepts and architecture of Azure Service Fabric.

`Chapter 4`, *Hands-on with Service Fabric – Guest Executables*, talks about building and deploying applications as Guest Executables on a Service Fabric cluster.

`Chapter 5`, *Hands on with Service Fabric – Reliable Services*, explains the concept of Reliable Services programming model for building Microservices hosted on Service Fabric.

`Chapter 6`, *Reliable Actors*, introduces Actor programming model on Service Fabric and the ways to build and deploy actors on a Service Fabric cluster.

`Chapter 7`, *Microservices Architecture Patterns Motivation*, provides an overview of the motivation behind driving Microservices architectural patterns. The chapter also talks about the classification of the patterns that are discussed in this book.

`Chapter 8`, *Microservices Architectural Patterns*, introduces a catalog of design patterns categorized by its application. Each design pattern explains the problem and the proven solution for that problem. The pattern concludes with considerations that should be taken while applying the pattern and the use cases where the pattern can be applied.

`Chapter 9`, *Securing and Managing Your Microservices*, will guide you on securing your Microservices deployment on a Service Fabric cluster.

`Chapter 10`, *Diagnostics and Monitoring*, covers how to set up diagnostics and monitoring in your Service Fabric application. You will also learn how to use Service Fabric Explorer to monitor the cluster.

`Chapter 11`, *Continuous Integration and Continuous Deployment*, takes you through the process of deploying your Microservices application on a Service Fabric cluster using Visual Studio Team Services.

`Chapter 12`, *Serverless Microservices*, helps you understand the concept of Serverless Computing and building Microservices using Azure functions.

What you need for this book

The examples found in this book require a Microsoft Azure subscription. You can sign up for a free trial account via the Azure website: `https://azure.microsoft.com/`.

You will need Windows 7+, latest Service Fabric SDK, latest Azure SDK, latest Azure PowerShell, 4GB RAM, 30 GB available Hard Disk space, Visual Studio 2017, and Visual Studio Team Service for executing the practical examples in this book.

Who this book is for

The book is aimed at IT architects, system administrators, and DevOps engineers who have a basic knowledge of the Microsoft Azure platform and are working on, or are curious about, the concepts of Microservices and the Microservice architecture.

This book assumes that you are proficient in .NET, especially in C# development. A little knowledge of Azure SDK, Azure Management Portal, Azure PowerShell, and Azure Command Line Interface (Azure CLI) will help you navigate through this book easily.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "the argument `cancellationToken` is an object of type `CancellationToken`"

A block of code is set as follows:

```
<Resources>
  <Endpoints>
     <Endpoint Name="ServiceEndpoint" Type="Input" Protocol="http"
        Port="80" />
  </Endpoints>
</Resources>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<Resources>
  <Endpoints>
    <Endpoint Name="ServiceEndpoint" Type="Input" Protocol="http"
      Port="80" />
  </Endpoints>
</Resources>
```

Any command-line input or output is written as follows:

```
PS C:\> Register-ServiceFabricApplicationType <Application name>
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: " In the **Security** section, set the **Security mode** to **Secure**"

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The complete set of code can also be downloaded from the following GitHub repository: `https://github.com/PacktPublishing/Microservices-with-Azure`. We also have other code bundles from our rich catalog of books and videos available at: `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/MicroserviceswithAzure_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

This book is divided into three parts so that you can choose an appropriate starting point in the book according to your interest:

1. **Laying The Foundation**: The chapters in this part of the book introduce you to the concept of Microservices and Microsoft Azure.
2. **Microsoft Azure Service Fabric**: The chapters in this part of the book introduce you to the various programming models available in Azure Service Fabric.
3. **Microservice Architecture Pattern**: The chapters in this part of the book give you an introduction to the Microservice architecture patterns and discuss several practical architecture patterns that you can use while designing your applications.
4. **Supplementary Learning**: The chapters in this part of the book will walk you through some of the essential but supplementary concepts of Service Fabric such as DevOps, security and, Nanoservices.

Continue learning at the companion website

https://microserviceswithazure.com/

`https://microserviceswithazure.com/`

Part 1 – Laying The Foundation

The chapters in this part of the book introduce you to the concept of Microservices and Microsoft Azure.

`Chapter 1`, *Microservices – Getting to Know the Buzzword*, lays the foundation of concepts of Microservices and explores the scenarios, where Microservices are best suited for your application.

`Chapter 2`, *Microsoft Azure Platform and Services Primer*, provides a very fast-paced overview of Microsoft Azure as a platform for hosting internet-scale applications.

Part 2 – Microsoft Azure Service Fabric

The chapters in this part of the book introduce you to the various programming models available in Azure Service Fabric.

`Chapter 3`, *Understanding Azure Service Fabric*, explains the basic concepts and architecture of Azure Service Fabric.

`Chapter 4`, *Hands-on with Service Fabric – Guest Executables*, talks about building and deploying applications as Guest Executables on a Service Fabric cluster.

`Chapter 5`, *Hands on with Service Fabric – Reliable Services*, explains the concept of Reliable Services programming model for building Microservices hosted on Service Fabric.

`Chapter 6`, *Reliable Actors*, introduces Actor programming model on Service Fabric and the ways to build and deploy actors on a Service Fabric cluster.

Part 3 – Microservice Architecture Patterns

The chapters in this part of the book give you an introduction to the Microservice architecture patterns and discuss several practical architecture patterns that you can use while designing your applications.

`Chapter 7`, *Microservices Architecture Patterns Motivation*, provides an overview of the motivation behind driving Microservices architectural patterns. The chapter also talks about the classification of the patterns that are discussed in this book.

`Chapter 8`, *Microservices Architectural Patterns*, introduces a catalog of design patterns categorized by its application. Each design pattern explains the problem and the proven solution for that problem. The pattern concludes with considerations that should be taken while applying the pattern and the use cases where the pattern can be applied.

Part 4 – Supplementary Learning

The chapters in this part of the book will walk you through some of the essential but supplementary concepts of Service Fabric such as DevOps, security and, Nanoservices.

`Chapter 9`, *Securing and Managing Your Microservices*, will guide you on securing your Microservices deployment on a Service Fabric cluster.

`Chapter 10`, *Diagnostics and Monitoring*, covers how to set up diagnostics and monitoring in your Service Fabric application. You will also learn how to use Service Fabric Explorer to monitor the cluster.

`Chapter 11`, *Continuous Integration and Continuous Deployment*, takes you through the process of deploying your Microservices application on a Service Fabric cluster using Visual Studio Team Services.

`Chapter 12`, *Serverless Microservices*, helps you understand the concept of Serverless Computing and building Microservices using Azure functions.

1
Microservices – Getting to Know the Buzzword

The world of information technology today is witnessing a revolution influenced by cloud computing. Agile, inexpensive, scalable infrastructure which is completely self-serviced and pay-per-use has a critical part to play in optimizing the operational efficiency and time-to-market for software applications enabling all major industries. With the changing nature of underlying hardware and operational strategies, many companies find it challenging to meet competitive business requirements of delivering applications or application features which are highly scalable, highly available, and continuously evolving by nature.

The agility of this change has also compelled solution architects and software developers to constantly rethink their approach of architecting a software solution. Often, a new architecture model is inspired by learnings from the past. Microservices-driven architecture is one such example which is inspired by **Service-Oriented Architecture** (**SOA**). The idea behind Microservices-based architecture is heavily based on componentization, abstraction, and object-oriented design, which is not new to a software engineer.

In a traditional application, this factorization is achieved by using classes and interfaces defined in shared libraries accessed across multiple tiers of the application. The cloud revolution encourages developers to distribute their application logic across services to better cater to changing business demands such as faster delivery of capabilities, increased reach to customers across geographies, and improved resource utilization.

What are Microservices?

In simple words, a Microservice can be defined as an autonomous software service which is built to perform a single, specific, and granular task.

The word *autonomous* in the preceding definition stands for the ability of the Microservice to execute within isolated process boundaries. Every Microservice is a separate entity which can be developed, deployed, instantiated, scaled, and managed discretely.

The language, framework, or platform used for developing a Microservice should not impact its invocation. This is achieved by defining communication contracts which adhere to industry standards. Commonly, Microservices are invoked using network calls over popular internet protocols such as REST.

On cloud platforms, Microservices are usually deployed on a **Platform as a Service** (**PaaS**) or **Infrastructure as a Service** (**IaaS**) stack. It is recommended to employ a management software to regulate the lifecycle of Microservices on a cloud stack. This is especially desirable in solutions which require high density deployment, automatic failover, predictive healing, and rolling updates. Microsoft Azure Service Fabric is a good example of a distributed cluster management software which can be used for this purpose. More about this is covered in later sections of this book.

Microservices are also highly decoupled by nature and follow the principle of minimum knowledge. The details about the implementation of the service and the business logic used to achieve the task are abstracted from the consuming application. This property of the service enables it to be independently updated without impacting dependent applications or services. Decoupling also empowers distributed development as separate teams can focus on delivering separate Microservices simultaneously with minimal interdependency.

It is critical for a Microservice to focus on the task it is responsible for. This property is popularly known as the **Single Responsibility Principle** (**SRP**) in software engineering. This task ideally should be elementary by nature. Defining the term *elementary* is a key challenge involved in designing a Microservice. There is more than one way of doing this:

- Restricting the cyclomatic complexity of the code module defining the Microservice is one way of achieving this. Cyclomatic complexity indicates the complexity of a code block by measuring the linear independent paths of execution within it.
- Logical isolation of functionality based on the bounded context that the Microservice is a part of.
- Another simpler way is to estimate the duration of delivering a Microservice.

Irrespective of the approach, it is also important to set both minimum and maximum complexity for Microservices before designing them. Services which are too small, also known as **Nanoservices**, can also introduce crucial performance and maintenance hurdles.

Microservices can be developed using any programming language or framework driven by the skills of the development team and the capability of the tools. Developers can choose a performance-driven programming language such as C or C++ or pick a modern managed programming language such as C# or Java. Cloud hosting providers such as Azure and Amazon offer native support for most of the popular tools and frameworks for developing Microservices.

A Microservice typically has three building blocks – **code**, **state**, and **configuration**. The ability to independently deploy, scale, and upgrade them is critical for the scalability and maintainability of the system. This can be a challenging problem to solve. The choice of technology used to host each of these blocks will play an important role in addressing this complexity. For instance, if the code is developed using .NET Web API and the state is externalized on an Azure SQL Database, the scripts used for upgrading or scaling will have to handle compute, storage, and network capabilities on both these platforms simultaneously. Modern Microservice platforms such as Azure Service Fabric offer solutions by co-locating state and code for the ease of management, which simplifies this problem to a great extent.

Co-location, or having code and state exist together, for a Microservice has many advantages. Support for versioning is one of them. In a typical enterprise environment, it's a common requirement to have side-by-side deployments of services serving in parallel. Every upgrade to a service is usually treated as a different version which can be deployed and managed separately. Co-locating code and state helps build a clear logical and physical separation across multiple versions of Microservices. This will simplify the tasks around managing and troubleshooting services.

A Microservice is always associated with a unique address. In the case of a web-hosted Microservice, this address is usually a URL. This unique address is required for discovering and invoking a Microservice. The discoverability of a Microservice must be independent of the infrastructure hosting it. This calls for a requirement of a service registry which keeps track of where each service is hosted and how it can be reached. Modern registry services also capture health information of Microservices, acting like a circuit breaker for the consuming applications.

Microservices natively demands hyperscale deployments. In simpler words, Microservices should scale to handle increasing demands. This involves seamless provisioning of compute, storage, and network infrastructure. It also involves challenges around lifecycle management and cluster management. A Microservices hosting platform typically has the features to address these challenges.

Microservices hosting platform

The primary objective of a Microservices hosting platform is to simplify the tasks around developing, deploying, and maintaining Microservices while optimizing the infrastructure resource consumption. Together, these tasks can be called *Microservice lifecycle management tasks*.

The journey starts with the hosting platform supporting development of the Microservices by providing means for integrating with platform features and application framework. This is critical to enable the hosting platform to manage the lifecycle of a service hosted on it. Integration is usually achieved by the hosting platform exposing APIs (application programming interfaces) which can be consumed by the development team. These APIs are generally compatible with popular programming languages.

Co-locating code and state is desirable for improving the efficiency of a Microservice. While this is true, storing state locally introduces challenges around maintaining the integrity of data across multiple instances of a service. Hosting platforms such as Service Fabric come with rich features for maintaining consistency of state across multiple instances of a Microservice there by abstracting the complexity of synchronizing state from the developer.

The hosting platform is also responsible for abstracting the complexity around physical deployment of Microservices from the development team. One way this is achieved is by containerizing the deployment. Containers are operating system-level virtualized environments. This means that the kernel of the operating system is shared across multiple isolated virtual environments. Container-based deployment makes possible an order-of-magnitude increase in density of the Microservice deployed. This is aligned with the recommended cloud design pattern called **compute resource consolidation**. A good example to discuss in this context, as mentioned by Mark Fussell from Microsoft, is the deployment model for Azure SQL Databases hosted on Azure Service Fabric. A SQL Azure Database cluster comprises hundreds of machines running tens of thousands of containers hosting a total of hundreds of thousands of databases. Each of these containers hosts code and state associated with multiple Microservices. This is an inspiring example of how a good hosting platform can handle hyperscale deployment of Microservices.

A good hosting platform will also support deployment of services across heterogeneous hardware configurations and operating systems. This is significant for meeting demands of services which have specific requirements around high-performance hardware. An example would be a service which performs **GPU** (**graphics processing unit**) intensive tasks.

Once the Microservices are deployed, management overhead should be delegated to the hosting platform. This includes reliability management, health monitoring, managing updates, and so on. The hosting platform is responsible for the placement of a Microservice on a cluster of virtual machines. The placement is driven by a highly optimized algorithm which considers multiple constraints at runtime to efficiently pick the right host virtual machine for a Microservice.

The following diagram illustrates a sample placement strategy of Microservices in a cluster:

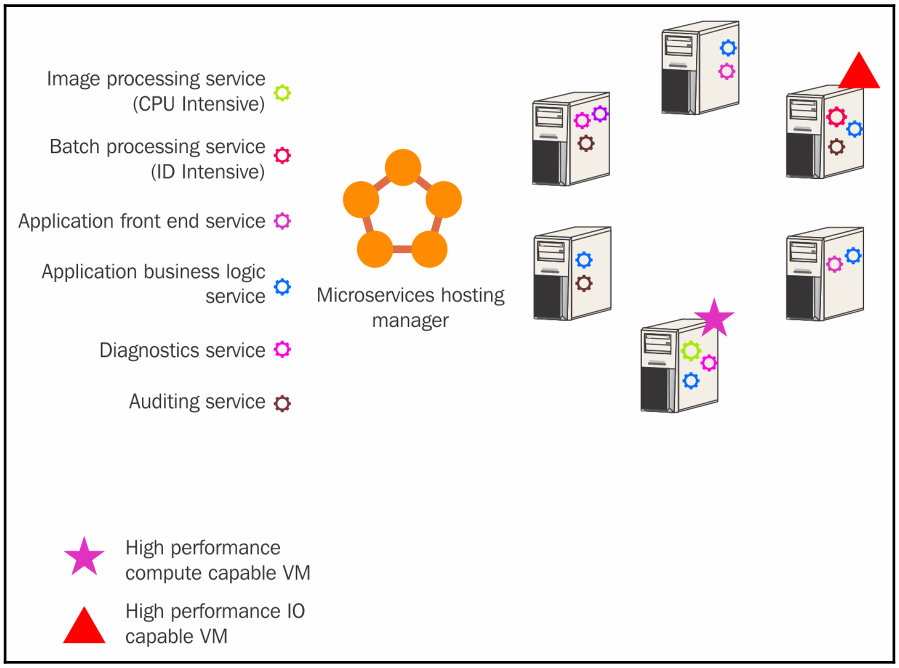


Image processing service (CPU Intensive)
Batch processing service (ID Intensive)
Application front end service
Application business logic service
Diagnostics service
Auditing service
Microservices hosting manager
High performance compute capable VM
High performance IO capable VM

Microservice placement strategy

As the number of Microservices grows, so does the demand for automating monitoring, and diagnostics systems which takes care of the health of these services. The hosting platform is responsible for capturing the monitoring information from every Microservice and then aggregating it and storing it in a centralized health store. The health information is then exposed to the consumers and also ingested by the hosting platform itself, to take corrective measures. Modern hosting platforms support features such as preventive healing, which uses machine learning to predict future failures of a virtual machine and take preventive actions to avoid service outages. This information is also used by the failover manager subsystem of the hosting platform to identify failure of a virtual machine and to automatically reconfigure the service replicas to maintain availability. The failover manager also ensures that when nodes are added or removed from the cluster, the load is automatically redistributed across the available nodes. This is a critical feature of a hosting platform considering the nature of the cloud resources to fail, as they are running on commodity hardware.

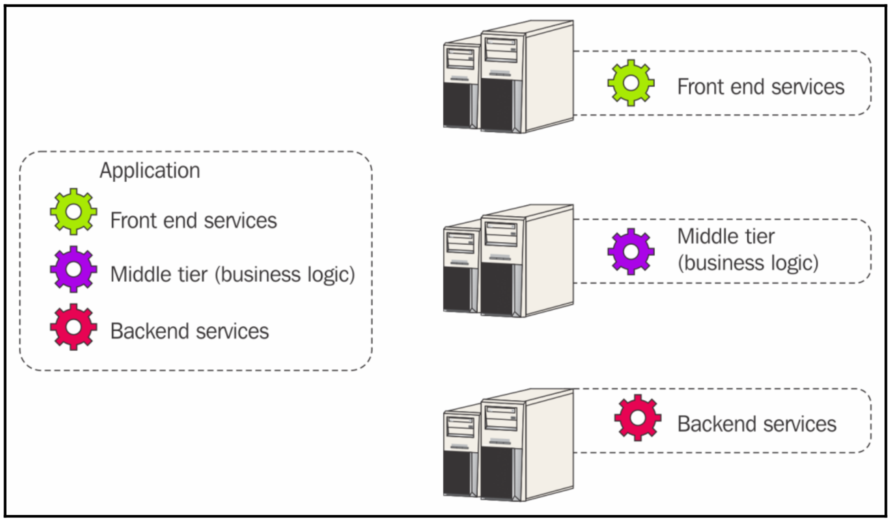
Considering the fact that migrating to a Microservices architecture can be a significant change in terms of the programming paradigm, deployment model, and operational strategy, a question which usually rises is *why adopt a Microservice architecture?*

The Microservice advantage

Every application has a shelf life, after which it is either upgraded or replaced with another application with evolved capabilities or which is a better fit for changing business needs. The agility in businesses has reduced this shelf life further by a significant factor. For instance, if you are building an application for distributing news feeds among employees within a company, you would want to build quicker prototypes and get feedback on the application sooner than executing an elaborate design and plan phase. This, of course, will be with the cognizance that the application can be further optimized and revised iteratively. This technique also comes in handy when you are building a consumer application where you are unsure of the scale of growth in the user base. An application such as Facebook, which grew its user base from a couple of million to 1,500 million in a few years would have been impossible to plan for, if the architecture was not well architected to accommodate future needs. In short, modern-day applications demand architectural patterns which can adapt, scale and gracefully handle changes in workload.

To understand the benefits of Microservices architecture for such systems, we will require a brief peek at its predecessor, monolithic architecture. The term monolith stands for a single large structure. A typical client-server application for the previous era would use a tiered architecture. Tiers would be decoupled from one another and would use contracts to communicate with each other. Within a tier, components or services would be packed with high cohesion, making them interdependent on each other.

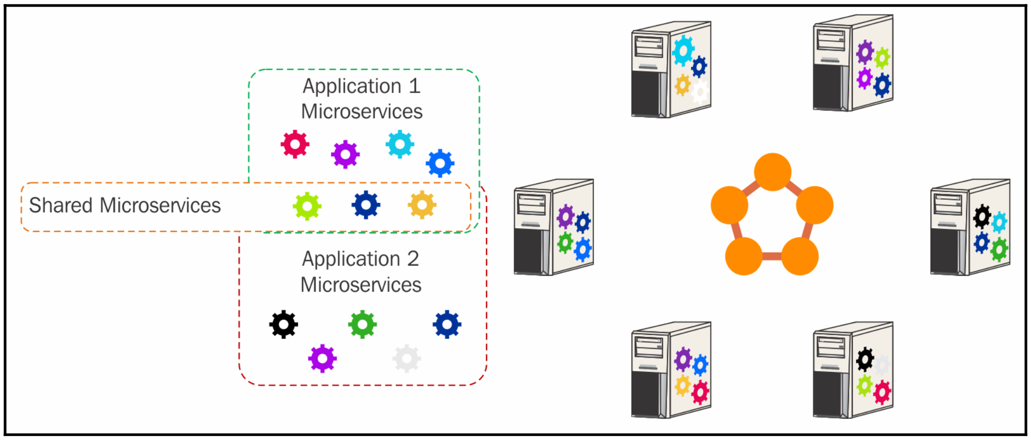
The following diagram illustrates a typical monolithic application architecture:



Monolithic application deployment topology

This works fine in simpler systems which are aimed to solve a static problem catering to a constant user base. The downside is that the components within a tier cannot scale independently, neither can they be upgraded or deployed separately. The tight coupling also prevents the components from being reused across tiers. These limitations introduce major roadblocks when a solution is expected to be agile by nature.

Microservices architecture addresses these problems by decomposing tightly coupled monolithic ties to smaller services. Every Microservice can be developed, tested, deployed, reused, scaled, and managed independently. Each of these services will align to a single business functionality. The development team authoring a service can work independently with the customer to elicit business requirements and build the service with the technology best suited to the implementation of that particular business scenario. This means that there are no overarching constraints around the choice of technology to be used or implementation patterns to be followed. This is perfect for an agile environment where the focus is more on delivering the business value over long-term architectural benefits. A typical set of enterprise applications may also share Microservices between them. The following diagram illustrates the architecture of such a Microservices-driven solution:



Microservice application deployment topology

The following are a few key advantages of a Microservice architecture:

Fault tolerance

As the system is decomposed to granular services, failure of a service will not impact other parts of the system. This is important for a large, business-critical application. For instance, if a service logging events of the system fails, it will not impact the functioning of the whole system.

The decomposed nature of the services also helps fault isolation and troubleshooting. With proper health monitoring systems in place, a failure of a Microservice can be easily identified and rectified without causing downtime to the rest of the application. This also applies to application upgrades. If a newer version of a service is not stable, it can be rolled back to an older version with minimal impact to the overall system. Advanced Microservice hosting platforms such as Service Fabric also come with features such as predictive healing, which uses machine learning to foresee failures and takes preventive measures to avoid service downtime.

Technology-agnostic

In today's world, when the technology is changing fast, eliminating long-term commitment to a single technology stack is a significant advantage. Every Microservice can be built on a separate technology stack and can be redesigned, replaced, or upgraded independently as they execute in isolation. This means that every Microservice can be built using a different programming language and use a different type of data store which best suits the solution. This decreases the dependency concerns compared to the monolithic designs, and makes replacing services much easier.

A good example where this ability of a Microservice maximizes its effect is a scenario where different data stores can be used by different services in alignment with the business scenario they address. A logging service can use a slower and cheaper data store, whereas a real-time service can use a faster and more performant data store. As the consuming services are abstracted from the implementation of the service, they are not concerned about the compatibility with the technology used to access the data.

Development agility

Microservices being handled by separate logical development streams makes it easier for a new developer to understand the functionality of a service and ramp up to speed. This is particularly useful in an agile environment where the team can constantly change and there is minimal dependency on an individual developer. It also makes code maintenance related tasks simpler as smaller services are much more readable and easily testable.

Often, large-scale systems have specific requirements which require specialized services. An example of this is a service which processes graphical data which requires specialized skills to build and test the service. If a development team does not have the domain knowledge to deliver this service, it can be easily outsourced or offloaded to a different team which has the required skill sets. This would be very hard in a monolithic system because of the interdependency of services.

Heterogeneous deployment

The ability of Microservices to be executed as an isolated process decouples it from the constraints around a specific hosting environment. For instance, services can be deployed across multiple cloud stacks such as IaaS and PaaS and across different operating systems such as Windows and Linux hosted on private data centers or on cloud. This decouples the technology limitations from the business requirements.

Most of the mid and large sized companies are now going through a cloud transformation. These companies have already invested significant resources on their on-premises data centers. This forces cloud vendors to support hybrid computing models where the IT infrastructure can coexist across cloud and on-premises data centers. In this case, the infrastructure configuration available on-premises may not match the one provisioned on cloud. The magnitude of application tiers in a monolithic architecture may prevent it from being deployed on less capable server machines, making efficient resource utilization a challenge. Microservices, on the other hand, being smaller, decoupled deployment units, can easily be deployed on heterogeneous environments.

Manageability

Each Microservice can be separately versioned, upgraded, and scaled without impacting the rest of the system. This enables running multiple development streams in parallel with independent delivery cycles aligned with the business demands. If we take a system which distributes news to the employees of a company as an example, and the notification service needs an upgrade to support push notifications to mobile phones, it can be upgraded without any downtime in the system and without impacting the rest of the application. The team delivering the notification service can function at its own pace without having a dependency on a big bang release or a product release cycle.

The ability to scale each service independently is also a key advantage in distributed systems. This lets the operations team increase or decrease the number of instances of a service dynamically to handle varying loads. A good example is systems which require batch processing. Batch jobs which run periodically, say once in a day, only require the batch processing service to be running for a few hours. This service can be turned on and scaled up for the duration of batch processing and then turned off to better utilize the computing resources among other services.

Reusability

Granularity is the key for reuse. Microservices, being small and focused on a specific business scenario, improve the opportunity for them to be reused across multiple subsystems within an organization. This in turn reflects as momentous cost savings.

The factor of reuse is proportional to the size of the organization and its IT applications. Bigger companies have more number of applications developed by multiple development teams, each of which may run their own delivery cycles. Often, the lack of ability to share code across these teams forces software components to be duplicated, causing a considerable impact on development and maintenance cost. Although service duplication across applications may not always be bad, with proper service cataloging and communication, Microservices can easily solve this problem by enabling service reuse across business units.

The SOA principle

SOA has multiple definitions that vary with the vendors that provide platforms to host SOA services. One of the commonly accepted SOA definitions was coined by Don Box of Microsoft. His definition is essentially a set of design guidelines which a service-oriented system should adhere to.

> *Boundaries are Explicit*
> *Services are Autonomous*
> *Services share Schema and Contract, not Class*
> *Compatibility is based upon Policy*
> *– Don Box, Microsoft*

Although this definition was originally explained in relation to Microsoft Indigo (now WCF), the tenets still hold true for other SOA platforms as well. An understanding of this principle is that all the services should be available in the network. This tenet dictates that no modules, routines, or procedures can be considered as participants in SOA. Let's take a look at the original tenets in a little detail. The first tenet says that a service should implement a domain functionality and should be discoverable by the other services making up the system. The discovery of service is generally done by registering each service in a directory. The clients of the services can discover each service at runtime. The second tenet explains that the services should be independent of the other services that make up the system. Since the services are independent of each other, they may also enjoy independence of platform and programming language. The third tenet advices that each service should expose an interface through which the rest of the services can communicate with it. The knowledge of this contract should be sufficient to operate with the service. The fourth tenet dictates that the services define the boundaries in which they would work. An example of such a boundary can be a range of integers within which a service that performs arithmetic operations would operate. Such policies should be mentioned in the form of policy expressions and should be machine readable. In WCF, the policies are implemented by the **Web Services Policy** (**WS-Policy**) framework.

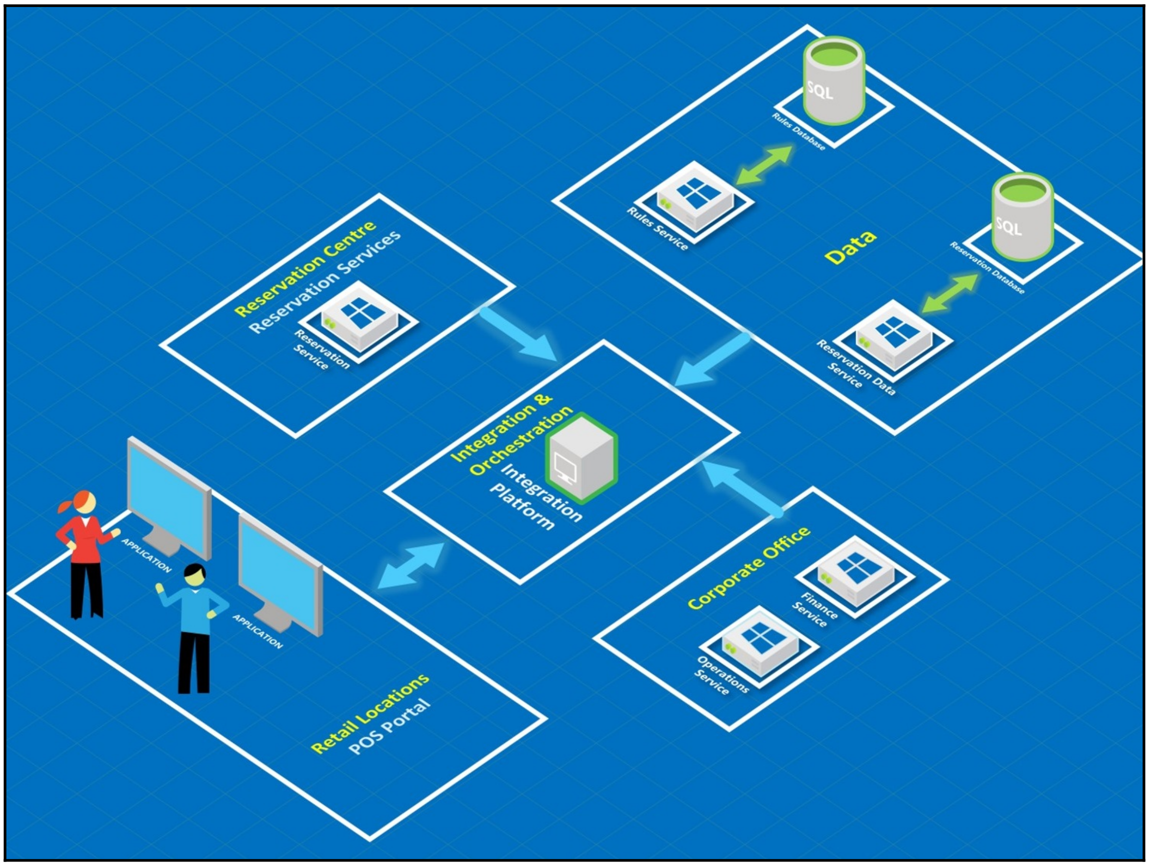
Although none of the original tenets dictate the size of individual services built using SOA architecture, to obtain independence from other services in the system, an individual service in SOA needs to be coarse-grained. To minimize interaction between services, each service should implement functionalities that work together.

In essence, both the Microservices architecture and the SOA architecture try to solve the problems of monolithic design by modularizing the components. In fact, a system already designed using the SOA architecture is a step in the right direction to realize Microservices architecture.

Issues with SOA

An inherent problem in the SOA architecture is that it tries to mimic the communication levels in an enterprise. SOA principles take a holistic look at the various communication channels in an enterprise and try to normalize them. To understand this problem in a better manner, let us take a look at a real-world SOA implementation done for an organization.

The following is the architecture of a real-life SOA-based application of a car rental company. The architecture diagram presented below has intentionally been simplified to ease understanding:



SOA architecture

This model is a classic example of an SOA-based system. The various participants in this SOA landscape are as follows:

- **The corporate office services**: These services provides data pertaining to fleet management, finances, data warehouse, and so on.
- **The reservation services**: These services help manage bookings and cancellations.
- **Backend services**: These services interface the systems that supply rules to the system and that supply reservation data to the system. There might be additional systems involved in the application, but we will consider only two of them at the moment.
- **Integration platform**: The various services of the system need to interact with each other. The integration platform is responsible for orchestrating the communication between the various services. This system understands the data that it receives from the various systems and responds to the various commands that it receives from the portal.
- **The Point of Sale portal**: The portal is responsible for providing an interface for the users to interact with the services. The technology to realize the frontend of the application is not important. The frontend might be a web portal, a rich client, or a mobile application.

The various systems involved in the application may be developed by different teams. In the preceding example, there can be a team responsible for the backend systems, one for the reservation center, one for the corporate office, one for the portal, and one for the integration services. Any change in the hierarchy of communication may lead to a change in the architecture of the system and thus drive up the costs. For instance, if the organization decides to externalize its finance systems and offload some of the information to another system, then the existing orchestrations would need to be modified. This would lead to increased testing efforts and also redeployment of the entire application.

Another aspect worth noting here is that the integration system forms the backbone of SOA. This concept is generally wrongly interpreted and **Enterprise Service Bus** (**ESB**) is used to hook up multiple monoliths which may communicate over complicated, inefficient, and inflexible protocols. This not only adds the overhead of complex transformations to the system but also makes the system resilient to change. Any change in contract would lead to composing of new transformations.

Typical SOA implementations also impede agility. Implementing a change in application is slow because multiple teams need to coordinate with each other. For example, in the preceding scenario, if the application needs to accept a new means of payment, then the portal team would need to make changes in the user interface, the payment team would need to make changes in their service, the backend team would need to add new fields in the database to capture the payment details, and the orchestration team would need to make changes to tie the communication together.

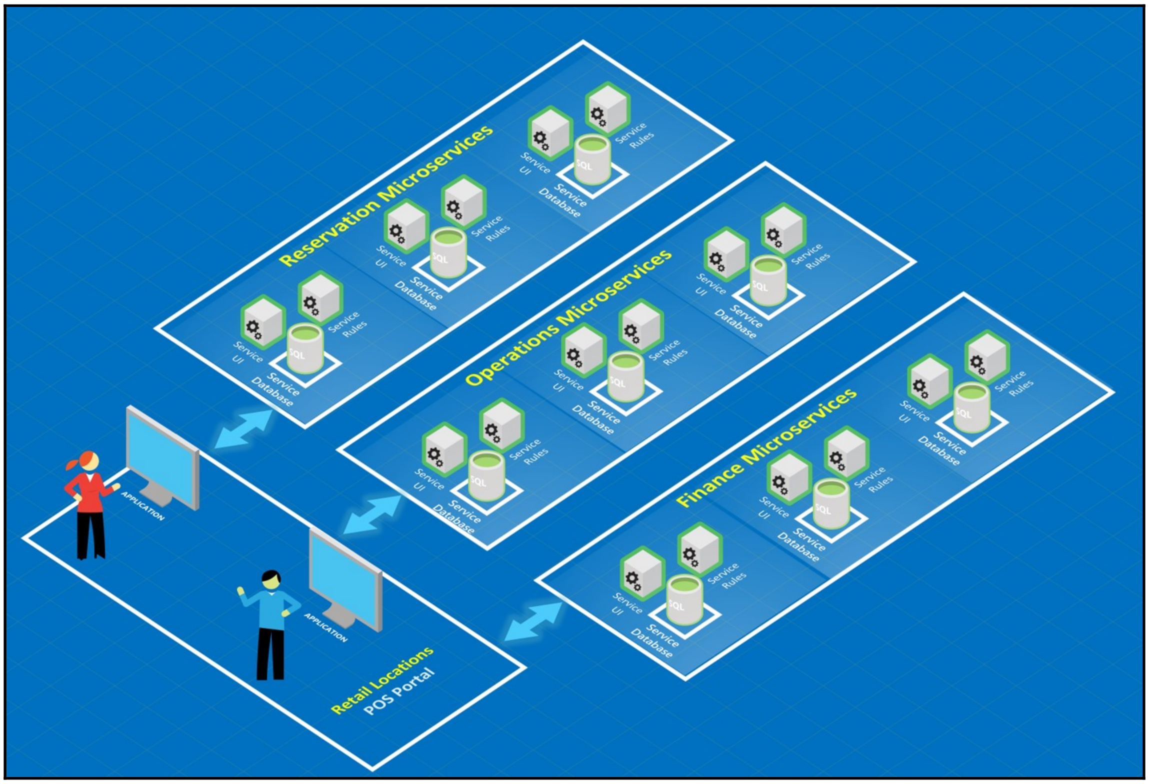
The participant services in SOA also face versioning issues. If any of the services modify their contract, then the orchestrator systems would need to undergo changes as well. In case the changes are too expensive to make, the new version of service would need to maintain backward compatibility with the old contract, which may not always be possible. The deployment of modified services requires more coordination as the modified service needs to be deployed before the affected services get deployed, leading to the formation of deployment monoliths.

The orchestration and integration system runs the risk of becoming a monolith itself. As most of the business logic is concentrated in the orchestration system, the services might just be administering data whereas the orchestration system contains all the business logics of the entire application. Even in a domain-driven design setting, any change in an entity that leads to a change in the user interface would require redeployment of many services. This makes SOA lose its flexibility.

The Microservices solution

Unlike SOA, which promotes cohesion of services, Microservices promote the principle of isolation of services. Each Microservice should have minimal interaction with other Microservices that are part of the system. This gives the advantage of independent scale and deployment to the Microservices.

Let's redraw the architecture of the car rental company using the Microservices architecture principle:



Microservices architecture

In the revised architecture, we have created a Microservice corresponding to each domain of the original system. This architecture does away with the integration and orchestration component. Unlike SOA, which requires all services to be connected to an ESB, Microservices can communicate with each other through simple message passing. We will soon look at how Microservices can communicate.

Also, note that we have used the principles of **Domain-Driven Design** (**DDD**), which is the principle that should be used for designing a Microservices-based system. A Microservice should never spawn across domains. However, each domain can have multiple Microservices. Microservices avoid communicating with each other and for the most part use the user interface for communication.

In the revised setup, each team can develop and manage a Microservice. Rather than distributing teams around technologies and creating multiple channels of communication, this distribution can increase agility. For instance, adding a new form of payment requires making a change in the payment Microservice and therefore requires communication with only a single team.

Isolation between services makes adoption of Continuous Delivery much simpler. This allows you to safely deploy applications and roll out changes and revert deployments in case of failures.

Since services can be individually versioned and deployed, significant savings are attained in the deployment and testing of Microservices.

Inter-Microservice communication

Microservices can rarely be designed in a manner that they do not need to communicate with each other. However, if you base your Microservices system on the DDD principle, there should be minimal communication required between the participant Microservices.

Cross-domain interactions of Microservices help reduce the complexity of individual services and duplication of code. We will take a look at some of the communication patterns in `Chapter 8`, *Microservices Architectural Patterns*. However, let us look at the various types of communication.

Communication through user interface

In most cases, the usability of a system is determined through the frontend. A system designed using Microservices should avoid using a monolithic user interface. There are several proponents of the idea that Microservices should contain a user interface and we agree with that.

Tying a service with a user interface gives high flexibility to the system to incorporate changes and add new features. This also ensures that distribution of teams is not by the communication hierarchy of the organization but by domains that Microservices are a part of. This practice also has the benefit of ensuring that the user interface will not become a deployment monolith at any point in time.

Although there are several challenges associated with integrating the user interface of Microservices, there are several ways to enable this integration. Let's take a look at a few.

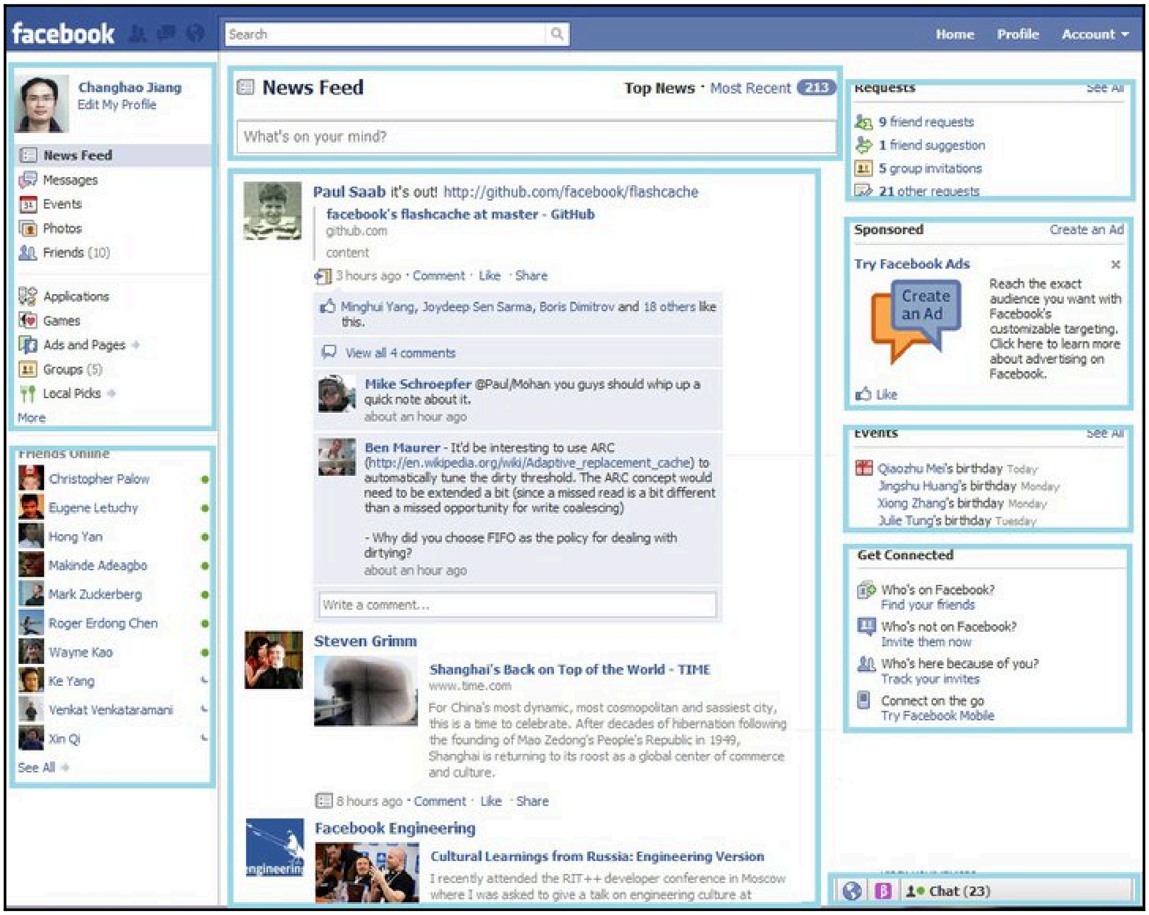
Sharing common code

To ensure a consistent look and feel of the end user portal, code that ensures consistency can be shared with the other frontends. However, care should be taken to ensure that no business logic, binding logic, or any other logic creeps into the shared code.

Your shared code should always be in a state of being released publicly. This will ensure that no breaking changes or business logic gets added to the shared library.

Composite user interface for the web

Several high-scale websites such as Facebook and MSN combine data from multiple services on their page. Such websites compose their frontend out of multiple components. Each of these components could be the user interface provided by individual Microservices. A great example of this approach is Facebook's **BigPipe** technology, which composes its web page from small reusable chunks called *pagelets* and *pipes* them through several executing stages inside web servers and browsers:



Paul Saab it's out! http://github.com/facebook/flashcache
facebook's flashcache at master - GitHub
github.com
content
3 hours ago · Comment · Like · Share
Minghui Yang, Joydeep Sen Sarma, Boris Dimitrov and 18 others like this.
View all 4 comments
Mike Schroepfer @Paul/Mohan you guys should whip up a quick note about it.
about an hour ago
Ben Maurer - It'd be interesting to use ARC (http://en.wikipedia.org/wiki/Adaptive_replacement_cache) to automatically tune the dirty threshold. The ARC concept would need to be extended a bit (since a missed read is a bit different than a missed opportunity for write coalescing)

- Why did you choose FIFO as the policy for dealing with dirtying?
about an hour ago
Write a comment...

Steven Grimm
Shanghai's Back on Top of the World - TIME
www.time.com
For China's most dynamic, most cosmopolitan and sassiest city, this is a time to celebrate. After decades of hibernation following the founding of Mao Zedong's People's Republic in 1949, Shanghai is returning to its roost as a global center of commerce and culture.
8 hours ago · Comment · Like · Share

Facebook Engineering
Cultural Learnings from Russia: Engineering Version
I recently attended the RIT++ developer conference in Moscow where I was asked to give a talk on engineering culture at

Reach the exact audience you want with Facebook's customizable targeting. Click here to learn more about advertising on Facebook.

Facebook BigPipe (source: https://www.facebook.com/notes/facebook-engineering/bigpipe-pipelining-web-pages-for-high-performance/389414033919/)

The composition of a user interface can take place at multiple levels, ranging from development to execution. The flexibility of such integrations varies with the level they are carried out at.

The most primitive form of composition can be the sharing of code, which can be done at the time of development. However, using this integration, you have to rely on deployment monoliths as the various versions of user interface can't be deployed in parallel.

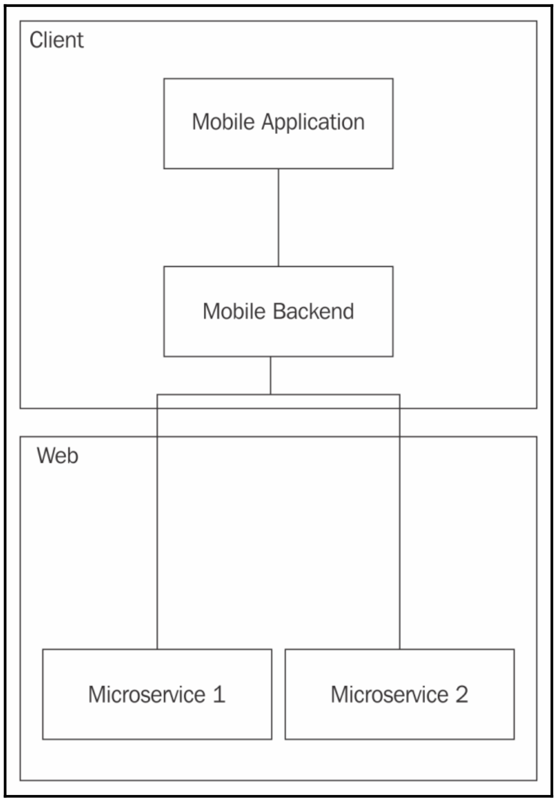
A much more flexible integration can also take place at runtime. For instance, Asynchronous JavaScript and XML (AJAX), HTML, and other dependencies can be loaded in the browser. Several JavaScript frameworks, such as Angular.js, Ember.js, and Ext.js, can help realize composition in single-page applications.

In cases where integration through JavaScript is not feasible, middleware may be used which fetches the HTML component of each Microservice and composes them to return a single HTML document to the client. Some typical examples of such compositions are the edge side includes of varnish or squid, which are proxies and caches. Server-side includes such as those available on Apache and NGINX can also be used to carry out transformations on servers rather than on caches.

Thin backend for rich clients

Unlike web applications, rich clients need to be deployed as monoliths. Any change in the Microservices would require a fresh deployment of the client application. Unlike web applications where each Microservice consists of a user interface, it is not the case for mobile or desktop applications. Moreover, structuring the teams in a manner that each team has a frontend developer for each rich client that the application can be deployed to is not feasible.

A way in which this dependency can be minimized is by having a backend for the rich client applications which is deployed with the application:



Microservices for rich clients

Although this approach is not perfect, it does ensure that part of the system conforms to Microservices architecture. Care should be taken to not alter any Microservice to encapsulate the business logic of the rich client. The mobile and desktop clients should optimize content delivery as per their needs.

Synchronous communication

A simple solution for synchronous communication between services is to use REST and transfer JSON data over HTTP. REST can also help in service discovery by using **Hypermedia as the Engine of Application State** (**HATEOAS**). HATEOAS is a component of REST which models relationships between resources by using links. Once the client queries the entry point of the service, it can use the links it receives to navigate to other Microservices.

If text-based transfers are not desired, protocol buffers (Google's data interchange format) may be used to transmit data. This protocol has been implemented in several languages to increase its adoption, for example, Ruby protobuf.

A protocol that can be used to transmit structured data across a network is **Simple Object Access Protocol** (**SOAP**). It can be used to make calls to different Microservices using various transport mechanisms such as JMS, TCP, or UDP. SOAP is language-neutral and highly extensible.

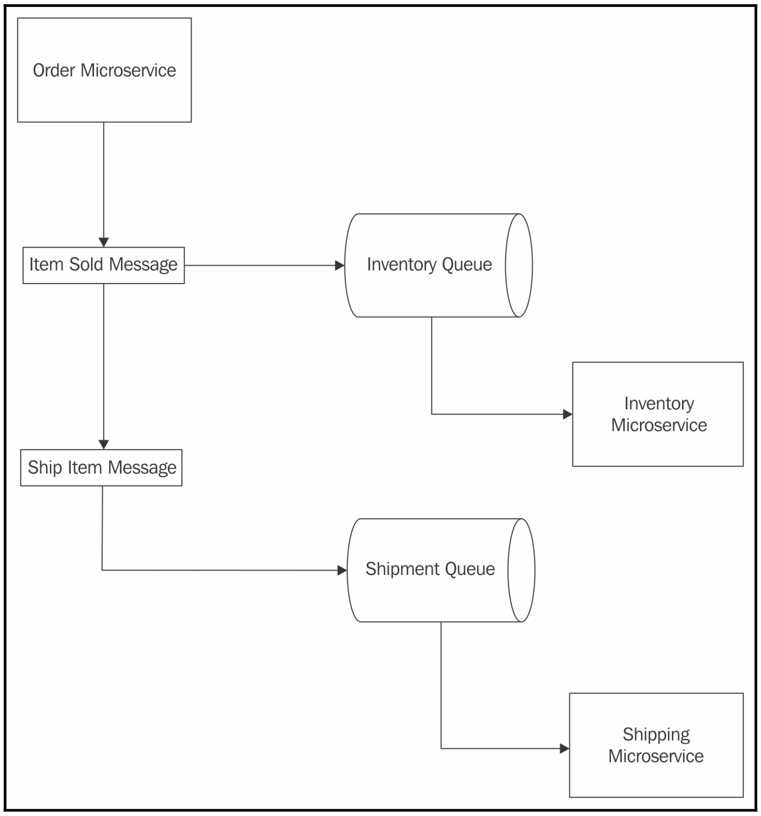
Asynchronous communication

Asynchronous message passing has the benefit of truly decoupling Microservices from each other. Since the communication is carried out by a broker, individual services need not be aware of the location of the receiver of the request. This also gives individual services the ability to scale independently and recover and respond to messages in case of failure. However, this communication pattern lacks the feature of immediate feedback and is slower than the synchronous communication format.

There are several tools available for such communication, such as MSMQ and Rabbit MQ. Microsoft Azure offers Service Bus Queues and Microsoft Azure Storage Queue for asynchronous messaging on cloud. Amazon SQS provides similar functionality in Amazon Web Services.

Orchestrated communication

This process is similar to the asynchronous communication process that we discussed earlier. Orchestrated communication still uses message stores to transmit data; however, the Microservice sending the data would insert different messages in different queues in order to complete the action. For example, an **Order Microservice** would insert the message in the queue consumed by the **Inventory Microservice** and another message in the queue consumed by the **Shipment Microservice**:



Orchestrated communications using queues

The orchestration may be carried out by a separate component, which is known as Saga, which we will read more about in `Chapter 8`, *Microservices Architectural Patterns*.

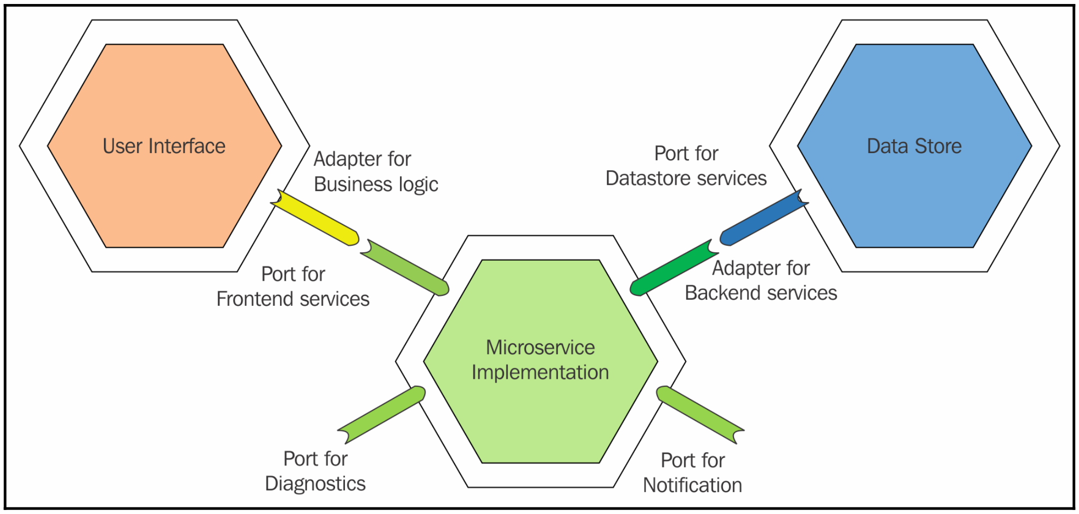
Shared data

Microservices should not share the same data store. Sharing data representation can make altering the database very difficult, and even if done, such a change always runs the risk of causing failure to services that are still using the old data representation. Such challenges ultimately lead to a bloated and complex database and accumulation of lots of dead data over time.

Data replication is a possible solution to sharing data across Microservices. However, data should not be blindly replicated across Microservices as the same problems that are present with shared databases would still remain. A custom transformation process should convert data available from the database to the schema used by the data store of the Microservice. The replication process can be triggered in batches or on certain events in the system.

Architecture of Microservices-based systems

Many of us have been curious about the representation of a Microservice by a hexagon. The reason for this is the inspiration behind the architectural pattern that drives *Microservices – the hexagonal architecture*. This pattern is also popularly known as **ports** and **adapters** in some parts of the globe. In a hexagonal architecture pattern, the code application logic is insulated with an isolation perimeter. This insulation helps a Microservice be unaware of the outside world. The insulation opens specific ports for establishing communication channels to and from the application code. Consuming applications can write adapters against these ports to communicate with the Microservice. The following diagram illustrates a hexagonal pattern for a Microservice:


User Interface
Adapter for Business logic
Port for Frontend services
Port for Datastore services
Data Store
Adapter for Backend services
Microservice Implementation
Port for Diagnostics
Port for Notification

Hexagonal architecture

Protocols in the case of a Microservice architecture are usually APIs. These APIs are exposed using popular protocols for ease of consumption. Hexagonal architecture lets the Microservice treat all of its consumers alike, whether it is a user interface, test suit, monitoring service, or an automation script.

Conway's law

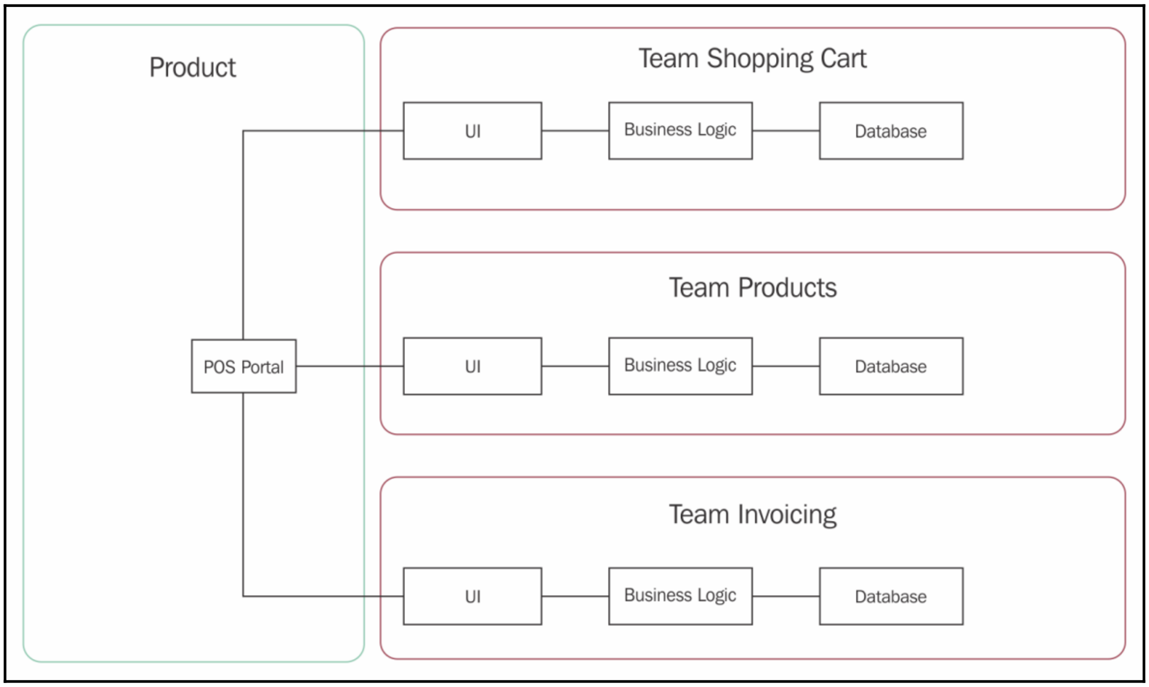
Melvin Edward Conway, an American computer scientist, coined a law that generally guides the design of the applications built by an organization.

> *Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.*
> *– Melvyn Conway 1967*

An important aspect of the law that should be noted is that the communication structure mentioned in the law is not the same as organizational hierarchy but rather how the various teams in the organization communicate with each other. For instance, an e-commerce company might have a product team and an invoicing team. Any application designed by this organization will have a product module and an invoicing module that will communicate with each other through a common interface.

For a large enterprise with many communication channels, the application architecture will be very complex and nearly impossible to maintain.

Using the law in conjunction with principles of domain driven design can actually help an organization enhance agility and design scalable and maintainable solutions. For instance, in an e-commerce company, teams may be structured around the domain components rather than the application layers that they specialize in, for instance, user interface, business logic, and database:



Team structure for Microservices development

Since the domains are clearly defined, the teams across domains will not need to interact too frequently. Also, the interfaces between teams would not be too complex and rigid. Such team layouts are commonly employed by large organizations such as Amazon, where each team is responsible for creating and maintaining a part of a domain.

Amazon practices the *two-pizza* rule to limit the size of teams. According to the rule, no team can be larger in size than what two pizzas can feed. Amazon also does not practice heavy communication between teams and all teams are required to communicate with each other through APIs. For instance, if the marketing team needs statistical data from a product team, they can't ask them for it. They need to hit the product team's API to get the data.

Microservices work better when coupled with the principles of domain driven design rather than communication channels. In the application architecture that we designed earlier, we could have ignored the domains of the application and classified teams by communication structure; for instance, two Microservices may be created, each of which handles product listing and product inventory. Such a distribution might lead to each of the teams to develop components independently of each other and will make moving functionalities between them very difficult if the communication hierarchy changes, such as when the two services need to be merged.

Summary

In this chapter, we learned about the concept of Microservices and its evolution. We also compared it with its predecessors, SOA and monolithic architecture. We then explored the requirement for a Microservices hosting platform and its properties.

We then discussed the various means of communications between Microservices and their advantages and disadvantages, after which we explored the architecture of a Microservices-based system.

To conclude, we also explored the philosophy behind hexagonal architecture and Conway's law.

In the next chapter, we will learn about Microsoft Azure, a cloud platform for hosting Microservices.

2

Microsoft Azure Platform and Services Primer

Organizations today need to be agile and have a fast way to differentiate themselves from the competition. Many organizations that only relied on software to fulfill their support needs are now embracing it to drive their core competency. Satya Nadella, CEO Microsoft, quoted this phenomenon in his speech at the Convergence Conference:

Every business will be a software business
– Satya Nadella, March 2015, Convergence Conference, USA

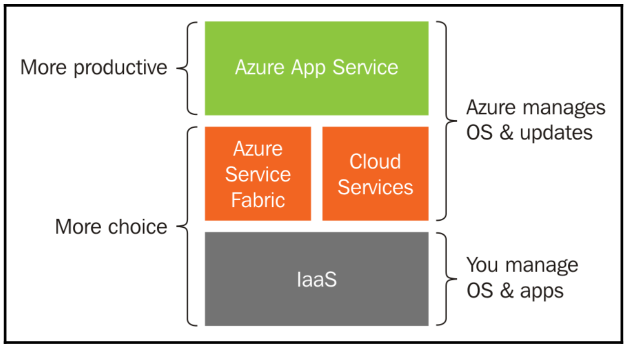
With the advent of software to the core business of organizations, the organizations have started demanding more agility and features out of the application hosting platforms. Organizations today have strict requirements of agility, availability, and DevOps. Since such requirements do not get sufficiently met by monolithic architecture and traditional means of deployment and management, organizations are moving to reliable and scalable cloud applications.

Developer time is a scarce resource for any organization. Every organization strives to reduce the development overheads, such as provisioning of resources and maintenance of infrastructure, by implementing DevOps practices. High agility and low turnaround time can be achieved by using as many platform services as possible in a solution. Modern developers compose solutions by integrating robust services rather than solving harder problems such as scale, failover, and upgrades to deliver solutions in short period of time, which otherwise would've taken a lot of time and resources to build.

There are considerations that need to be made while opting for the right cloud services model for building your applications, namely **Platform as a Service** (**PaaS**) and **Infrastructure as a Service** (**IaaS**). They both vary with respect to the level of flexibility that you will have and amount of responsibility that you would need to fulfill. Microsoft Azure fills the sweet spot between the two models through *Microsoft Azure Service Fabric* and *Microsoft Azure Cloud Services* which enable the developers to deliver highly flexible services without worrying about the underlying infrastructure.

Microsoft Azure offers cloud services as a PaaS offering to support cloud services that are scalable and reliable. Using cloud services, you can host your applications on virtual machines, albeit with higher degree of control than App Services. For example, you can install your own software on a cloud services VM and remote into the machines. However, cloud services are on the verge of becoming obsolete and it is recommended that you use App Services to build your applications. Therefore, we will not cover in detail or recommend using cloud services for your Microservices development.

Following is the image for Microsoft hosting service stack:



Microsoft hosting service stack

A typical Microservices-based system encounters challenges related to loose coupling, separation of concerns, and inter-service communication, and therefore adding the burden of infrastructure would make it very hard to implement. Applications that require Microservices architecture should choose a **Platform as a Service** (**PaaS**) model of development, to really reap the benefits of the architecture.

PaaS for Microservices

Microservices need to be updated, scaled, and managed, both individually and as a cohesive unit. The PaaS environment that supports Microservices development should have the following features.

Abstract infrastructure challenges

A Microservices development platform should not allow **Quality of Service** (**QoS**) challenges such as network latency, messaging formats, scalability, availability, and reliability to propagate to the application.

Scaling is a key enabler of the cloud platform. Scaling can be performed either by migrating the application to a host with higher compute capacity, also known as **scaling up**, or by adding more hosts with the same compute capacity to the pool of resources that the application gets deployed on, also known as **scaling out**. Although scaling up is limited to the maximum size of the available virtual machines, scaling out is virtually unlimited on cloud.

To scale out an application, the workload can either be distributed evenly among all instances or be partitioned between the available instances so that only the instances participating in the partition take up the work. Microsoft Azure Service Fabric supports both the models of scaling.

Microsoft Azure takes care of availability through redundancy of the infrastructure: a backup host gets spawned to resume processing when a host dies. Microsoft Azure Service Fabric can transfer the state of the failed host to the new host so that no data gets lost. Similarly, Microsoft Azure Service Fabric can manage upgrades in a manner that there is no downtime and that recovery is possible if an error is encountered.

The diagnostics and monitoring subsystems of the platform should be robust enough to detect and react to application and platform issues to ensure that services run in a reliable manner. Microsoft Azure Service Fabric provides monitoring and diagnostic features to ensure that Microservices deployed on it have high reliability.

Simplified application lifecycle management

Continuous Integration and Continuous Deployment are now at the core of DevOps. The agile methodology relies on applications being built and delivered in an iterative manner. Microservices support being developed in an agile manner and, because of their isolated nature, can be independently deployed. The PaaS platform should enable features such as automated testing, Continuous Integration and Continuous Deployment, version management, and recovery from failures so that Microservices can be continuously deployed.

Some organizations such as Facebook and Google practice delivering codes at very high velocities. In fact, Amazon is known to deploy code to its servers every 11.6 seconds. Such feats require DevOps pipelines that are continuous in nature, promote flow, small batch size, and continuous improvement.

Simplifying development

Microservices rely little on cooperation and need to discover services in order to communicate. Communication becomes a challenge with features such as replication, failover, and load balancing. The PaaS platform should take care of the challenges so that the developers can focus on delivering features. Microsoft Azure Service Fabric provides all of these features along with an easy programming model to solve these challenges.

As application delivery keeps on getting more agile, developers will keep asking for more features from the platform to deliver solutions that offer business value. There is a need to develop, iterate, and evolve rather than building monoliths that accumulate technical debt over a period and get retired after that.

Microsoft Azure offers several services that can be integrated to compose a solution that realizes a Microservices architecture. Using Service Fabric as a platform and the various Microsoft Azure services as enablers, developers can build unprecedented applications. Let's look at the various service offerings of Microsoft Azure.

Microsoft Azure – the choice of a hosting platform

Microsoft Azure is an Internet-scale cloud platform hosted in data centers, managed and supported by Microsoft. It has been available for public consumption since early 2010. Having grown very rapidly, Azure today provides services on all cloud stacks such as IaaS, PaaS, and SaaS, covering a wide range of offerings around computing, storage, networking, mobile, analytics, and many more.

Choosing a hosting service for your application is an important task. The choice is usually based on factors such as the hardware and software requirements of the application, the required control over the hosting environment, operational cost, options to scale, and supported deployment methods. The hosting services offered by Azure range from fully controllable virtual machines to the capability to host serverless code.

The support for popular enterprise-ready platforms and applications such as Windows Server, Linux, SQL Server, Oracle, IBM, and SAP, makes Azure VMs a popular choice for hosting virtualized workloads. This service is also backed by a gallery of numerous open source virtual machine images, driven by Microsoft partners. *Virtual Machine Scale Sets* can be used to reliably manage, deploy, upgrade, and scale identical virtual machines at a scale spanning to hundreds of instances. The ability to automatically scale a Virtual Machine Scale Set without pre-provisioning optimizes the operational cost significantly.

The support for open source products has opened new opportunities for Microsoft Azure as a cloud platform. *Containers* being the new buzzword, Azure offers an open source enabled container service which optimizes the configuration of popular open source tools and technologies for Azure. *Azure Container Service* is based on the Docker container format to ensure portability of your application containers across environments. It also supports a choice of orchestration solutions like Docker Swarm, Marathon, and DC/OS to enable hyperscale deployment of containers with the ability to scale to tens of thousands of instances.

For customers who would like Microsoft to manage their operating environment by delegating responsibilities such as patching and upgrading, Microsoft offers a fully managed PaaS called c*loud services.* The service supports two types of virtual machines:

- A *web role*, ideal for hosting web applications on **Internet Information Server** (**IIS**)
- A *worker role,* recommended for running services and background tasks

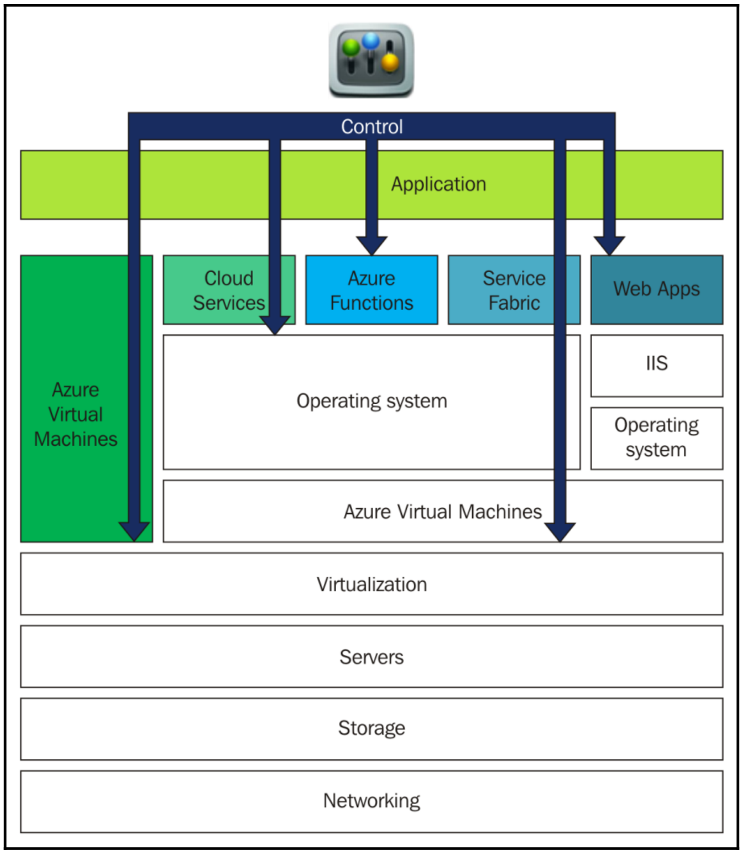
A typical enterprise application deployment will use a collection of web roles and worker roles for hosting application tiers. While the operating system and the underlying hardware is managed by Microsoft, consumers will still have the ability to install custom software on the virtual machine, configure the operating environment, and even remotely access the machine.

For simpler applications which are less concerned about the underlying hosting environment, Microsoft Azure offers App Services, which is a lightweight and powerful platform to host web, mobile, and API applications. This service enables developers to rapidly build, deploy, and manage powerful websites and web apps which can be coded in popular programming languages such as .NET, Node.js, PHP, Python, or Java. Consumers using this service have the ability to run their workload on a dedicated or a shared virtual machine fully managed by Microsoft. Irrespective of this choice, each application is hosted in completely isolated and secured process boundaries.

Azure also offers a service for hosting Nanoservices on a serverless platform. The service is called Azure Functions. Azure Functions supports a list of popular development languages such as C#, Node.js, Python, and PHP. Azure functions also have the capability to scale according to the load imposed on them.

All the hosting services listed above can be used to host Microservices, some with a lot of plumbing and some without any. However, the ability for a hosting platform to support heterogeneous underlying infrastructure and operating environments is critical in today's world. This is what makes Azure Service Fabric unique and the recommended platform to host Microservices. We will cover more about Azure Service Fabric in `Chapter 3`, *Understanding Azure Service Fabric*.

Among all influencers, the control a consumer will have on a hosting platform, in terms of accessing and configuring it, holds maximum weight. This is typically driven by organizational strategies and constrains within a modern-day enterprise. The following diagram illustrates the amount of control a consumer has on each of the hosting services discussed here:



Azure compute services

The preceding diagram illustrates the level of control a user has on a specific Azure service. For example, if cloud services are used, users have the capability of configuring everything above the operating system, such as system registry and environment variables.

Azure Service Fabric offers maximum flexibility by providing configurable managed layers. We will spend the next chapter discussing its caliber in detail.

Summary

In this chapter, we discussed the challenges that an ideal platform for hosting Microservices should address. We then briefly discussed Microservices application lifecycle management and ways to simplify development of Microservices.

In the last section, we compared different Microsoft Azure cloud stacks capable of hosting Microservices.

In the next chapter, we will dive deeper into Microsoft Azure Service Fabric as a platform for hosting enterprise-grade Microservices.

3
Understanding Azure Service Fabric

Microservices architecture portrays efficient mechanisms of solving modern enterprise problems. However, this simplification comes at a cost. Manually managing hyperscale deployments of Microservices is nearly impossible. Automating Microservices lifecycle management becomes an inevitable requirement to achieve enterprise-grade environment stability. This is where the role of Azure Service Fabric becomes significant. To start with, let's try to understand what Service Fabric is.

Mark Fussell, a senior program manager in the Microsoft Azure Service Fabric team, defines Service Fabric as the following:

> *Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable Microservices.*

Let's dig deeper into this definition. The definition categorizes Service Fabric as a platform. A platform, in theory, is a software component capable of hosting other software applications which are built to align with the constraints imposed by the platform and can use the features exposed by the platform for its execution. This is exactly what Service Fabric is for the Microservices services it hosts. Service Fabric is a platform which can host services and offers runtime features to support their execution.

The term distributed in the preceding definition highlights the capability of Service Fabric to host decoupled services. As a hosting platform, Service Fabric provides features to catalog, address, and access these services from other services hosted within the platform as well as from external applications.

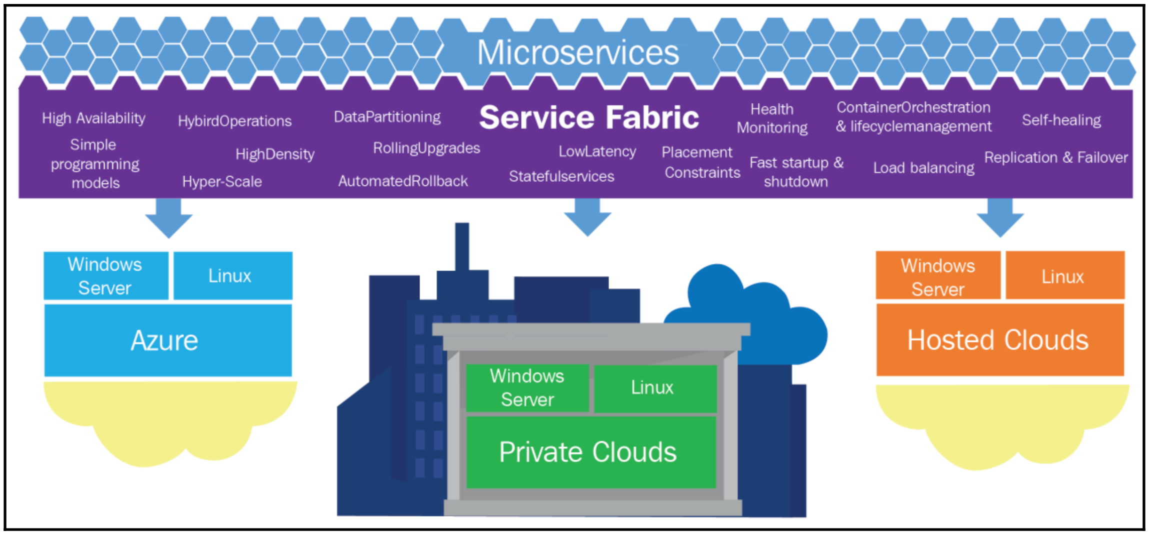
The process involved in end-to-end delivery of a service can be categorized into different phases according to the activities performed during each phase. These phases are design, development, testing, deployment, upgrading, maintenance, and removal. The task of managing all these phases involved in the delivery of a service is commonly known as application lifecycle management. Service Fabric provides first-class support for end-to-end application lifecycle management of applications deployed on cloud as well as for the ones running on-premises data centers.

The last and most important part of the definition emphasizes the capability of Service Fabric to host Microservices. Service Fabric offers capabilities for building and managing scalable and reliable applications composed of distributed, stateless (which do not maintain a session between requests) and stateful (which maintain a session between requests), Microservices running at very high density on a shared pool of machines. Service Fabric hosts Microservices inside containers deployed and activated across the Service Fabric cluster. By using a containerized execution environment for Microservices, Service Fabric is able to provide an increase in the density of deployment and improved portability across heterogeneous environments.

The Service Fabric advantage

Now that we understand what Service Fabric is, let's look at the advantages of using it as a platform to host Microservices.

As discussed in the earlier section in this chapter, Service Fabric is a distributed systems platform used to build hyperscalable, reliable, and easily managed applications for the cloud. The following figure mentioned in MSDN, provides a good overview of capabilities of Service Fabric as a hosting platform for Microservices:



Service Fabric features

Apart from the capabilities of Service Fabric to manage the application lifecycle for Microservices, the preceding diagram provides an important aspect of its ability to be deployed across heterogeneous environments. We will talk about this in detail in later sections of this chapter. Let's now dive deeper in to few of the key features of Service Fabric which make it an ideal platform to build a Microservice-based applications.

Highly scalable

Every Microservice hosted on Service Fabric can be scaled without affecting other Microservices. Service Fabric will support scaling-based on Virtual Machine Scale Sets which means that these services will have the ability to be auto-scaled based on CPU consumption, memory usage, and so on.

Service Fabric enables scaling while maximizing resource utilization with features such as like load balancing, partitions, and replication across all the nodes in the cluster. A Microservice hosted on Service Fabric can be scaled either at partition level or at name level.

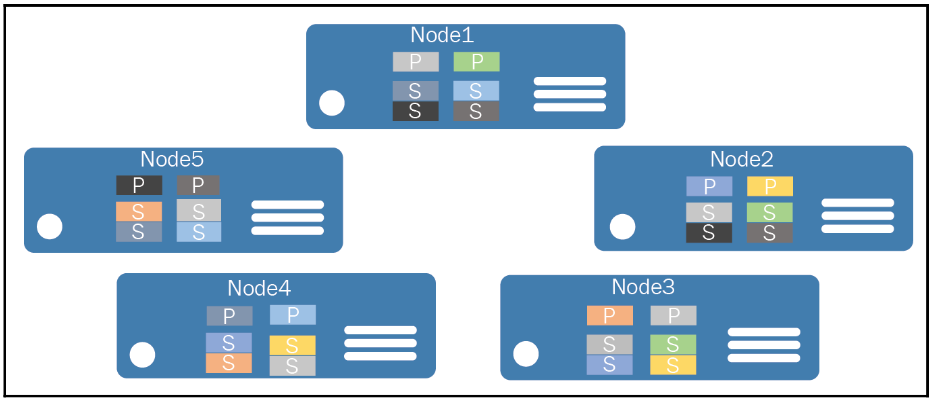
Support for partitioning

Service Fabric supports partitioning of Microservices. Partitioning is the concept of dividing data and compute into smaller units to improve the scalability and performance of a service.

A stateless Microservice can be of two types – one which stores the state externally and two which does not store a state. These Microservices are rarely partitioned as the scalability and the performance of the service can be enhanced by increasing the number of instances running the service. The only reason to partition a stateless service would be to achieve specialized routing requirements.

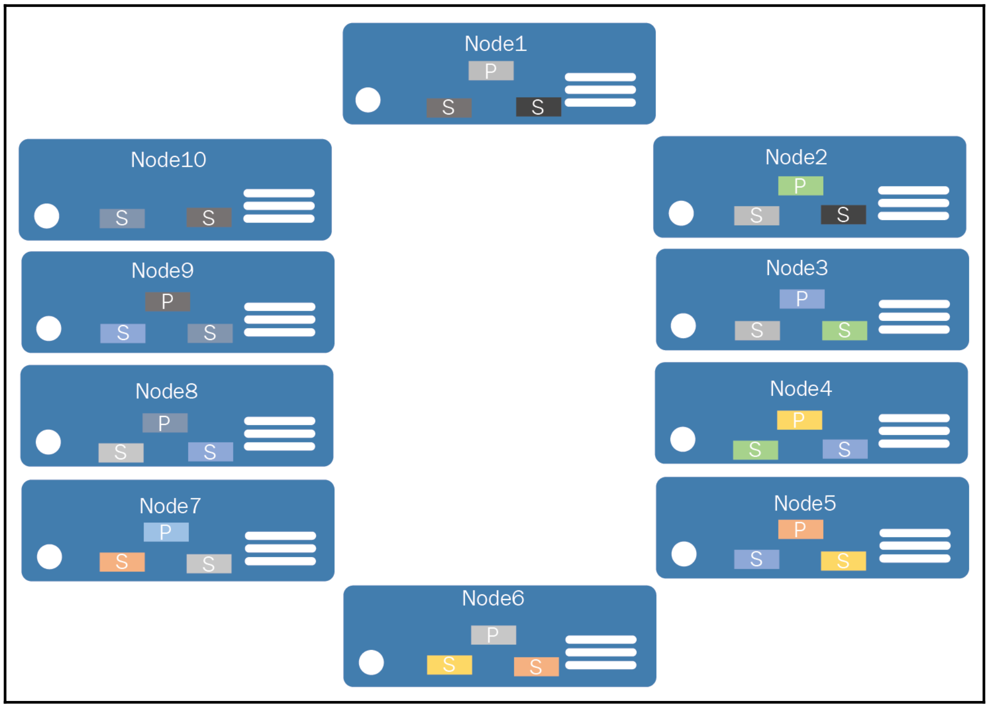
Service Fabric natively supports partitioning of state for a stateful service, thereby reducing the overheads on the developers around maintaining and replicating state across partition replicas. A partition of a stateful service can be thought of as a scale unit that is highly reliable through replicas distributed and load balanced across the nodes in a Service Fabric cluster. Service Fabric controls the optimal distribution of the partitions across multiple nodes, allowing them to grow dynamically. The partitions are rebalanced regularly to ensure resource availability to each service deployed on the cluster.

For instance, if an application with ten partitions each with three replicas is deployed on a five node cluster, Service Fabric will distribute the instances across the nodes with two primary replicas on each node:



Partitioning on a five node cluster

Later, if you scale up the cluster to ten nodes, Service Fabric will rebalance the primary replicas across all the 10 nodes:



Partitioning on 10 node cluster

The same logic will apply when the cluster is scaled down.

Rolling updates

A Service Fabric application is a collection of services. Every service, the ones which are part of the Service Fabric framework or the ones which are hosted on it, will require an upgrade at some point in time. To achieve high availability and low downtime of services during upgrades, Service Fabric supports rolling updates. This means that the upgrade is performed in stages. The concept of update domains is used to divide the nodes in a cluster into logical groups which are updated one at a time.

First, the application manifests of the new and existing deployments are compared to identify services which need an upgrade and only the ones requiring an update is refreshed. During the process of an upgrade, the cluster may contain a version of new and old services running in parallel. This forces the upgrades to be backward-compatible. A multi-phased upgrade can be used as a solution to achieve upgrade of non-compatible versions of services. In a multi-phased upgrade, the service is first upgraded to an intermediate version which is compatible with the old version of the service. Once this is successful, the intermediate version is upgraded to the final version of the service.

Service Fabric also supports non-rolling updates of services deployed in a cluster, a process also known as unmonitored upgrade.

State redundancy

For stateful Microservices, it is efficient to store state near compute for improved performance. Service Fabric natively integrates with a Microsoft technology called **Reliable Collections** to achieve collocation of compute and state for services deployed on it.

Reliable Collections can be thought of as a collection of state stores specifically designed for multi-computer applications. It enables developers to store state locally on a node within a Service Fabric cluster while assuring high availability, scalability, and low latency. For services running multiple instances, the state is replicated across nodes hosting different instances. Replication is the responsibility of the Reliable Services framework. This saves developers a significant amount of time and effort.

Reliable Services is also transactional by nature and supports asynchronous, non-blocking APIs. Presently, this technology supports two types of state stores – **Reliable Dictionary**, for storing key-value pairs, and **Reliable Queues**, a first-in-first-out data structure usually used for message passing.

High-density deployment

The recent revolution of containers has accelerated the ability to improve the density of deployment on a virtual machine. Microservices can be deployed within containers and the containers will provide the logical isolation for the services it hosts. Service Fabric enhances this ability to the next level by offering native support for Microservices. Every Microservice hosted on Service Fabric will be logically isolated and can be managed without impacting other services. This level of granularization in turn makes possible achieving a much higher density of deployment.

Another notable advantage of using Service Fabric is the fact that it is tried and tested. Microsoft runs services such as Azure DocumentDB, Cortana, and many core Azure services on Service Fabric.

Automatic fault tolerance

The cluster manager of Service Fabric ensures failover and resource balancing in case of a hardware failure. This ensures high availability of the services while minimizing manual management and operational overhead.

For a stateless service, Service Fabric lets you define an instance count, which is the number of instances of the stateless service that should be running in the cluster at a given time. The service can be scaled up by increasing the number of instances.

When Service Fabric detects a fault on an instance, it creates a new instance of the service on a healthy node within the cluster to ensure availability. This process is completely automated.

The story becomes a bit more complex for a stateful service. Service Fabric replicates a stateful service on different nodes to achieve high availability. Each replica will contain code used by the service and the state. All write operations are performed on one replica called the **primary replica**. All other replicas are called secondary replicas. Changes to state on the primary replica are automatically replicated to the secondary replicas by the framework. Service Fabric supports the configuration of a number of active secondary replicas. The higher the number of replicas, the better the fault tolerance of a service.

If the primary replica fails, Service Fabric makes one of the secondary replicas the primary replica and spins up a new instance of a service as a secondary replica.

Heterogeneous hosting platforms

It is common for enterprise environments to have heterogeneous hosting environments spread across multiple data centers managed and operated by different vendors. A key advantage of Service Fabric is its ability to manage clusters in and across heterogeneous environments. Service Fabric clusters can run on Azure, AWS, Google Cloud Platform, an on-premises data center, or any other third-party data center. Service Fabric can also manage clusters spread across multiple data centers. This feature is critical for services requiring high availability.

Service Fabric can manage virtual machines or computers running Windows Server or Linux operating systems. It can also operate on a diverse set of hardware configurations. While working with heterogeneous environments, usually there are scenarios where we want to host certain workloads on certain types of nodes. For instance, there may be services which execute GPU-intensive tasks which ideally should be placed on a node with a powerful GPU. In order to support such requirements, Service Fabric offers an option to configure placement constraints. Placement constraints can be used to indicate where certain services should run. These constraints are widely extensible. Nodes can be tagged with custom properties and constraints can be set for every service to be executed on certain types of nodes. Service Fabric also allows imposing constraints around the minimum resource capacity required to host a service on a node in a cluster. These constraints can be based on memory, disk space, and so on.

Technology agnostic

Service Fabric can be considered as a universal deployment environment. Services or applications based on any programming language or even database runtimes such as MongoDB can be deployed on Service Fabric.

Service Fabric supports four types of programming models – Reliable Services, Reliable Actors, Guest Executable, and Guest Containers. These topics are covered in detail in later parts of the book. Services can be written in any programming language and deployed as executables or hosted within containers.

Microsoft ships a rich **Software Development Kit** (**SDK**) for developing, packaging, and deploying services on Service Fabric managed clusters. Apart from .NET, Service Fabric also supports a native Java SDK for Java developers working on Linux. The Java SDK is supported by the Eclipse **integrated development environment** (**IDE**).

Centralized management

Monitoring, diagnosing, and troubleshooting are three key responsibilities of the operations team. Services hosted on Service Fabric can be centrally managed, monitored, and diagnosed outside application boundaries. While monitoring and diagnostics are most important in a production environment, adopting similar tools and processes in development and test environments makes the system more deterministic. The Service Fabric SDK natively supports capabilities around diagnostics which works seamlessly on both local development setups and production cluster setups.

Service Fabric has native support for **Event Tracing for Windows** (**ETW**). Service Fabric code itself uses ETW for internal tracing. This allows developers to centrally access application traces interleaved with Service Fabric system traces, which significantly helps in debugging. ETW is fast and works exactly the same way on development and production environments. However, ETW traces are local to the machine. While running a multi-node cluster, it helps to have a centralized vision on logs produced by all the services running in the cluster. The Azure Diagnostics extension can be used for this purpose. The extension can collect the logs from configured sources and aggregate it in Azure Storage to enable centralized access. External processes can be used to read the events from storage and place them into a product such as Log Analytics or Elastic Search, or another log-parsing solution for better visualization.

Service Fabric as an orchestrator

From a service management point of view, Service Fabric can be thought of as an orchestrator. An orchestrator in general terms is an automated piece of software used to manage service deployments. This piece of software is supposed to abstract the complexities around provisioning, deploying, fault handling, scaling, and optimizing the applications it is managing, from the end user. For instance, an orchestration should be able to consume a configuration which specifies the number of instances of service to run and perform the task of deploying the services-based on multiple complex factors such as resource availability on nodes in a cluster, placement constraints, and so on

Orchestrators are also responsible for fault handling and recovery of services. If a node in a cluster fails, the orchestrator needs to gracefully handle this while ensuring service availability. Updating a service deployment or applying a patch to the underlying framework is also managed by orchestrators. Apart from Service Fabric, there are other orchestration services available in the market, such as Mesosphere, Core OS, Docker Swarm, and Kubernetes. Azure container services open opportunities for some of these powerful orchestration services to be hosted on Azure. More about this is discussed later in this book.

Orchestration as a Service

In Service Fabric, the responsibilities of orchestration are primarily handled by the resource manager subsystem. This service is automatically initiates when a cluster is spun up and it stays awake for the lifetime of the cluster. The key responsibilities of this subsystem include optimization of environment, enforcing constraints, and assisting with other cluster management operations.

Let's dig a little deeper to understand the architecture of the resource manager service.

Is a cluster resource manager similar to an Azure load balancer?

While load balancing is a key part of managing a cluster, this is not what a cluster resource manager does. A traditional load balancer can be of two types, a **network load balancer** (**NLB**) or an **application load balancer** (**ALB**). The primary job of a load balancer is to make sure that all of the services hosted receive a similar amount of work. Some load balancers are also capable of ensuring session stickiness and some are even capable or optimizing the request routing-based on the turnaround time or current machine load.

While some of these strategies are efficient and best suited for a monolithic or tiered architecture, they lack the agility to handle faults and upgrades. A more responsive, integrated solution is required to handle hyperscale deployments of Microservices. The Service Fabric cluster resource manager is not a network load balancer. While a traditional load balancer distributes the traffic to where services are hosted, the Service Fabric cluster resource manager moves services to where there is room for them, or to where they make sense based on other conditions. These conditions can be influenced by the resource utilization on a node, a fault, an upgrade request, or so on.

For instance, the cluster resource manager can move services from a node which is busy to a node which is underutilized based on the CPU and memory use. It can also move a service away from a node which is in the queue for an upgrade.

Architecture of cluster resource manager

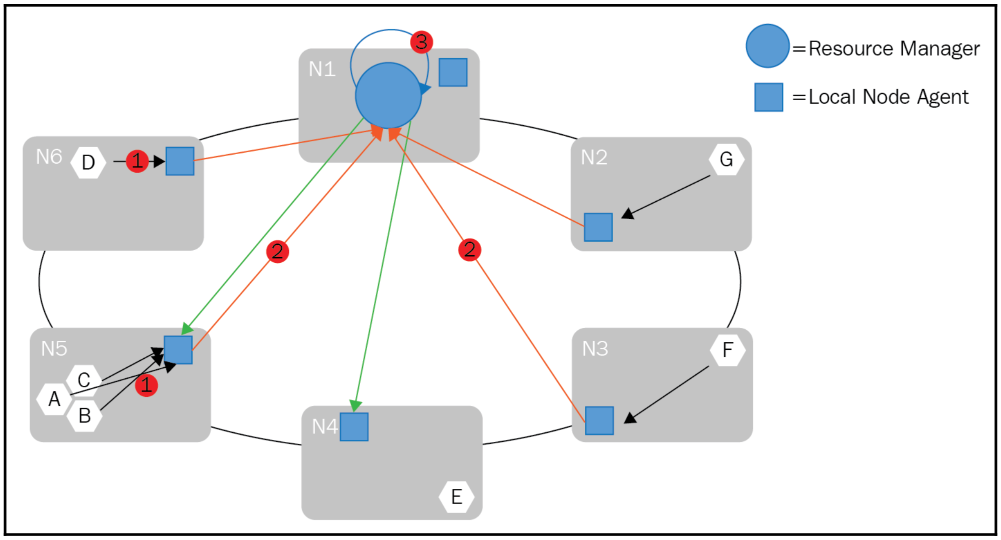
The process of resource management is complex. It is based on multiple parameters which are highly dynamic in nature. To perform resource balancing, the resource manager has to know about all the active services and the resources consumed by each of the services at this point of time. It should also be aware of the actual capacity of every node in the cluster in terms of memory, CPU, and disk space, and the aggregate amount of resources available in the cluster. The resources consumed by a service can change over time, depending on the load it is handling. This also needs to be accounted for before making a decision to move a service from one node to another. To add to the complexity, the cluster resources are not static. The number of nodes in the cluster can increase or decrease at any point of time, which can lead to a change in load distribution. Scheduled or unscheduled upgrades can also roll through the cluster, causing temporal outages of nodes and services. Also, the very fact of cloud resources running on commodity hardware forces the resource manager to be highly fault tolerant.

To achieve these tasks, the Service Fabric cluster resource manager uses two components. The first component is an *agent* which is installed on every node of a cluster. The agent is responsible for collecting information from the hosting node and relaying it to a centralized service. This information will include CPU utilization, memory utilization, remaining disk space, and so on. The agent is also responsible for heartbeat checks for the node.

The second component is a service. Service Fabric is a collection of services. The cluster resource manager service is responsible for aggregating all of the information supplied by the agent and other management services and reacting to changes based on the desired state configuration of the cluster and service. The fault tolerance of the service manager is achieved via replication, similar to how it is done for the services hosted on Service Fabric. The resource manager service runs seven replicas to ensure high availability.

To understand the process of aggregation, let's take an example.

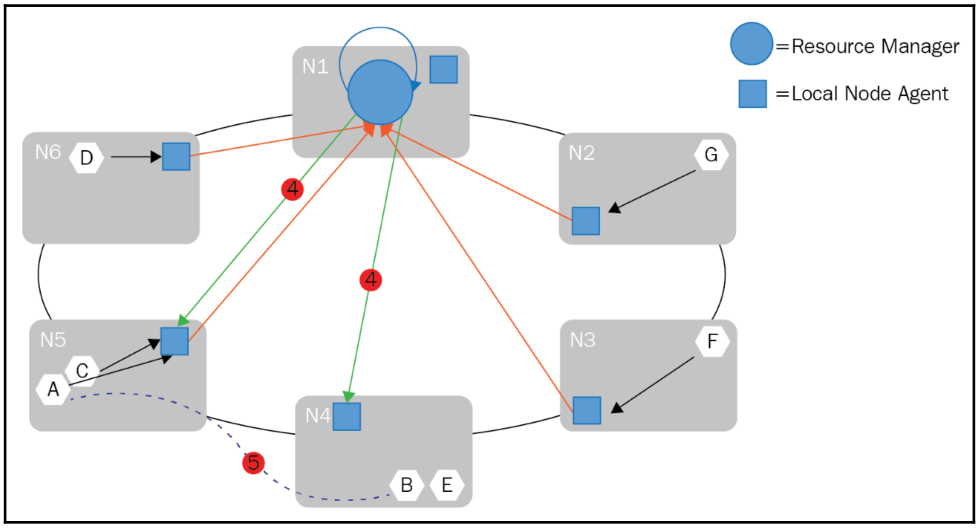
The following figure illustrates a Service Fabric cluster with six nodes. There are seven services deployed on this cluster with the names **A**, **B**, **C**, **D**, **E**, and **F**. The diagram illustrates the initial distribution of the services on the cluster based on placement rules configured for the services. Services **A**, **B**, and **C** are placed on node 5 (**N5**), service **D** on node 6 (**N6**), service **G** on node 2 (**N2**), service **F** on node 3 (**N3**) and service **E** on node 4 (**N4**). The resource manager service itself is hosted on node 1 (**N1**). Every node has a Service Fabric agent running which communicates with the resource manager service hosted on **N1**:



General resource manager functions

During runtime, if the amount of resources consumed by services changes, or if a service fails, or if a new node joins or leaves the cluster, all the changes on a specific node are aggregated and periodically sent to the central resource manager service. This is indicated by lines **1** and **2**. Once aggregated, the results are analyzed before they are persisted by the resource manager service. Periodically, a process within the cluster resource manager service, looks at all of the changes, and determines whether there are any corrective actions required. This process is indicated by the step **3** in the preceding figure.

To understand step **3** in detail, let's consider a scenario where the cluster resource manager determines that **N5** is overloaded. The following diagram illustrates a rebalancing process governed by the resource manager. This case is reported by the agent installed on **N5**. The resource manager service then checks available resources in other nodes of the cluster. Let's assume that **N4** is underutilized as reported by the agent installed on **N4**. The resource manager coordinates with other subsystems to move a service, which is service **B** in this instance, to **N4**. This is indicated by step **5** in the following diagram:

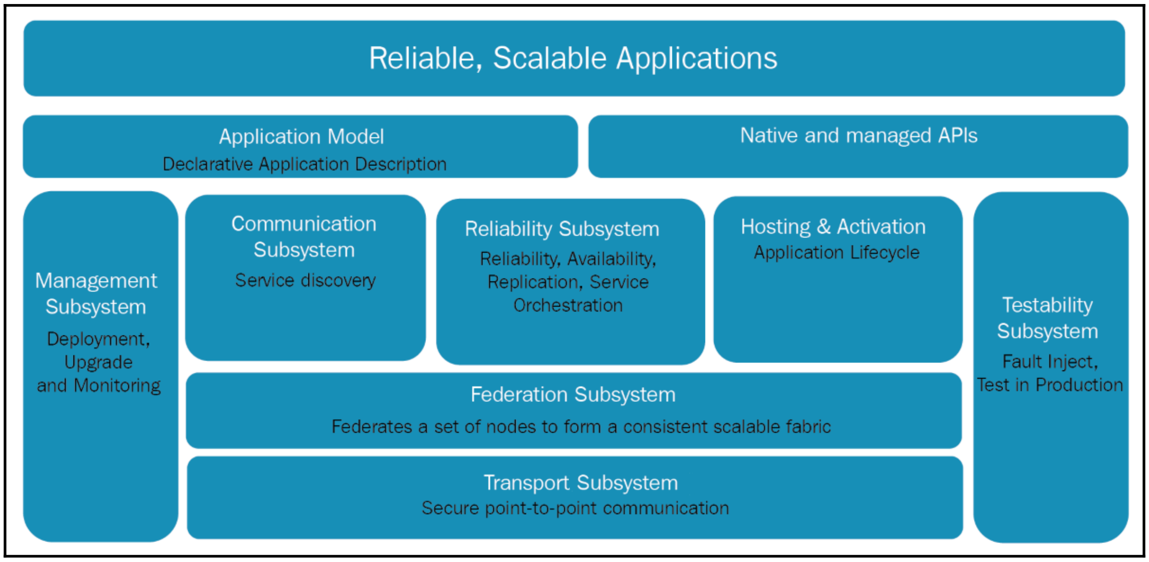


The Resource Manager reconfigures the clusters

This whole process is automated and its complexity is abstracted from the end user. This level of automation is what makes hyperscale deployments possible on Service Fabric.

Architecture of Service Fabric

Service Fabric is a collection of services grouped into different subsystems. These subsystems have specific responsibilities. The following diagram illustrates the major subsystems which form the Service Fabric architecture:


Reliable, Scalable Applications
Application Model
Declarative Application Description
Native and managed APIs
Management Subsystem
Deployment, Upgrade and Monitoring
Communication Subsystem
Service discovery
Reliability Subsystem
Reliability, Availability, Replication, Service Orchestration
Hosting & Activation
Application Lifecycle
Testability Subsystem
Fault Inject, Test in Production
Federation Subsystem
Federates a set of nodes to form a consistent scalable fabric
Transport Subsystem
Secure point-to-point communication

Subsystems of Service Fabric

The first layer from the bottom, the **Transport Subsystem** is responsible for providing secure communication channels between nodes in a Service Fabric cluster. The **Federation Subsystem** above it helps logically group physical or virtual nodes into a cluster so that it can be managed as a unit. This helps Service Fabric with tasks such as failure detection, leader election, and routing. Reliability of the workload hosted on Service Fabric is managed by the **Reliability Subsystem**. It owns the responsibility of replication, resource management, and failover. The **Hosting & Activation** Subsystem manages the lifecycle of the workload on every node and the **Management Subsystem** is responsible for managing the lifecycle of applications. The **Testability Subsystem** helps developers test their workload before and after deployment. Service location of services hosted on Service Fabric is managed by **Communication Subsystem**. The top three boxes capture the application programming models and the application model available for the developers to consume. More about application models is discussed in later parts of this book.

Transport Subsystem

The Transport Subsystem provides a communication channel for intra and inter cluster communication. The channels used for communication are secured by X509 certificate or Windows security. The subsystem supports both one-way and request-response communication patterns. These channels are in turn used by the Federation Subsystem for broadcast and multicast messaging. This subsystem is internal to Service Fabric and cannot be directly used by the developers for application programming.

Federation Subsystem

Federation Subsystem is responsible for logically grouping virtual or physical machines together to form a Service Fabric cluster. This subsystem uses the communication infrastructure provided by the Transport Subsystem to achieve this grouping. Grouping of nodes helps Service Fabric better manage the resources. The key responsibilities of this subsystem includes failure detection, leader election, and routing. The subsystem forms a ring topology over the nodes allocated for the cluster. A token-leasing mechanism along with a heartbeat check is implemented within the system to detect failures, perform leader election, and to achieve consistent routing.

Reliability Subsystem

Reliability of the service hosted on the platform is ensured by the Reliability Subsystem. It achieves these tasks by managing failover, replicating, and balancing resources across nodes in a cluster.

The replicator logic within this subsystem is responsible for replicating the state across multiple instances of a service. Maintaining consistency between the primary and the secondary replicas in a service deployment is its main task. It interacts with the failover unit and the reconfiguration agent to understand what needs to be replicated.

Any changes in the number of nodes in the cluster trigger the failover manager service. This in turn triggers automatic redistribution of services across the active nodes.

The resource manager plays the part of placing service replicas across different failure domains. It is also responsible for balancing the resources across the available nodes in the cluster while optimizing the load distribution and resource consumption.

Management Subsystem

The application lifecycle management of workloads deployed on a Service Fabric cluster is owned by the Management Subsystem. Application developers can access the Management Subsystem functionalities through administrative APIs or PowerShell cmdlets to provision, deploy, upgrade, or de-provision applications. All these operations can be performed without any downtime. The Management Subsystem has three key components – cluster manager, health manager, and image store.

The cluster manager interacts with the failover manager and the resource manager in the Reliability Subsystem for deploying the applications of available nodes considering the placement constraints. It is responsible for the lifecycle of the application, starting from provisioning to de-provisioning. It also integrates with the health manager to perform health checks during service upgrades.

Health manager, as the name suggests, is responsible for monitoring the health of applications, services, nodes, partitions, and replicas. It is also responsible for aggregating the health status and storing it in a centralized health store. APIs are exposed out of this system to query health events to perform corrective actions. The APIs can either return raw events or aggregated health data for a specific cluster resource.

The image store is responsible for persisting and distributing application binaries deployed on a Service Fabric cluster.

Hosting subsystem

The Hosting Subsystem takes care of managing application deployments within the scope of a node. The cluster manager signals the Hosting Subsystem, informing it about the application deployments to be managed on a particular node. The Hosting Subsystem then manages the lifecycle of the application on that node. It interacts with the Reliability Subsystem and Management Subsystem to ensure the health of each deployment.

Communication subsystem

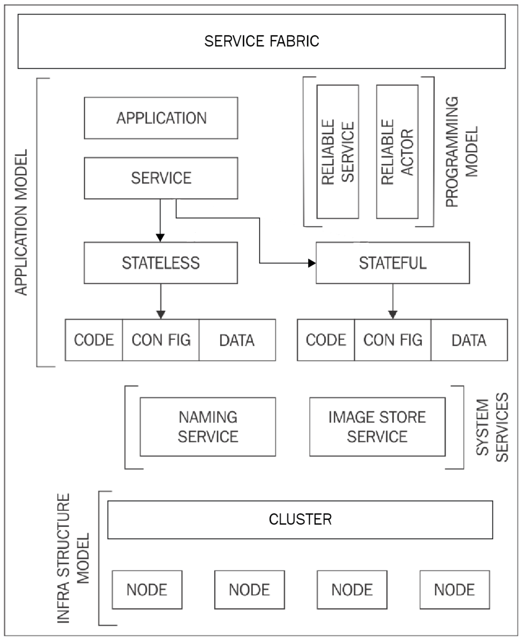
The communication subsystem provides features for service discovery and intra-cluster messaging features using a naming service. The naming service is used to locate a service within a cluster. It also lets users securely communicate with any node on a cluster, retrieve service metadata, and manage service properties. The naming service also exposes APIs, which enables users to resolve network location or each service despite of them being dynamically placed.

Testability Subsystem

The Testability Subsystem provides a list of tools for developers, deployment engineers, and testers to introduce controlled faults and run test scenarios to validate state transitions and behaviors of services deployed on Service Fabric. The fault analysis service is automatically started when a cluster is provisioned. 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.

Deconstructing Service Fabric

A Service Fabric application can be logically decomposed into multiple components. As a pro developer, you write your application using one of the programming models and supply the necessary configurations and data to make your code work, Service Fabric takes care of the rest of the stack and ensures that your code executes in a reliable and highly available environment. Let us take a look at the various components of a Service Fabric application and go through the individual components, starting from the infrastructure model:



Components of Service Fabric application

Infrastructure model

In a distributed computing platform such as Service Fabric, the computing load is distributed over all the available compute units, which are actually a number of virtual machines that work together. Because Service Fabric clusters can be deployed on any platform and on physical or virtual machines, your Service Fabric applications can run on a variety of platforms without any modifications. Let us take a look at the components that make up the infrastructure of Service Fabric.

Cluster

In Service Fabric, a cluster is a network-connected set of virtual or physical machines into which your Microservices are deployed and managed. Clusters can scale to thousands of machines. In traditional data centers, the most deployed clusters are the failover cluster and the load balancing cluster. However, quite unique to the Service Fabric cluster is the Service Fabric cluster resource manager. The Service Fabric cluster resource manager is responsible for deploying the Microservices across the nodes, taking care of the node capacity, fault tolerance, and sufficient replication. Since multiple Microservices can be deployed on a node, the Service Fabric cluster resource manager ensures that there is proper utilization of the compute resources.

Node

A machine or VM that is part of a cluster is called a **node**. A node might be a a physical or virtual machine. A node in Service Fabric is identified by its string name. A cluster may have heterogeneous nodes and more nodes can be added to the existing capacity to scale out the cluster.

Each node on the cluster has Service Fabric runtime binaries installed in it. When the node starts, an auto-start Service Fabric runtime service named `FabricHost.exe` spins up two executables on the node which make up the node:

- `Fabric.exe`: This executable is responsible for managing the lifetime of the Microservices hosted on the node
- `FabricGateway.exe`: This executable is responsible for managing communication between the nodes

System services

A set of system services gets provisioned on every Service Fabric cluster that provides the underlying support for Service Fabric applications.

Naming service

A Service Fabric application is composed of multiple Microservices. The Service Fabric cluster manager may deploy your service instances to any nodes in the cluster, which can make it difficult to discover the service by the client application.

Every Microservice in a Service Fabric application is identified by a string name. A Service Fabric cluster has multiple instances of the naming service, which resolves service names to a location in the cluster. A client can securely communicate with any node in the cluster using the naming service to resolve a service name and its location. When a client requests the location of a Microservice, the naming service responds with the actual machine IP address and port where it is currently running. This makes the Microservice independent of the hardware on which it is hosted.

Image store service

Once your application is ready to be deployed, the application package files are versioned and copied to Service Fabric cluster's image store. The image store is made available to the cluster and other services through a hosted service called the image store service. Once your application package has been uploaded, you need to register the application package to make the application type and version known to the cluster. After the application type is provisioned, you can create named applications from it.

Upgrade service

Azure cloud-based Service Fabric clusters are managed and upgraded by the **Service Fabric Resource Provider** (**SFRP**). The SFRP calls into a cluster through the HTTP gateway port on the management endpoint to get information about nodes and applications in the cluster. SFRP also serves the cluster information that is available to you in the **Azure Management Portal**. The HTTP Gateway ports is also used by the **Service Fabric Explorer** to browse and manage your cluster. The SFRP is an Azure-only service and is not available in the local development environment or any other infrastructure, such as Windows Server.

The upgrade service coordinates upgrading the Service Fabric itself with the SFRP.

Failover manager service

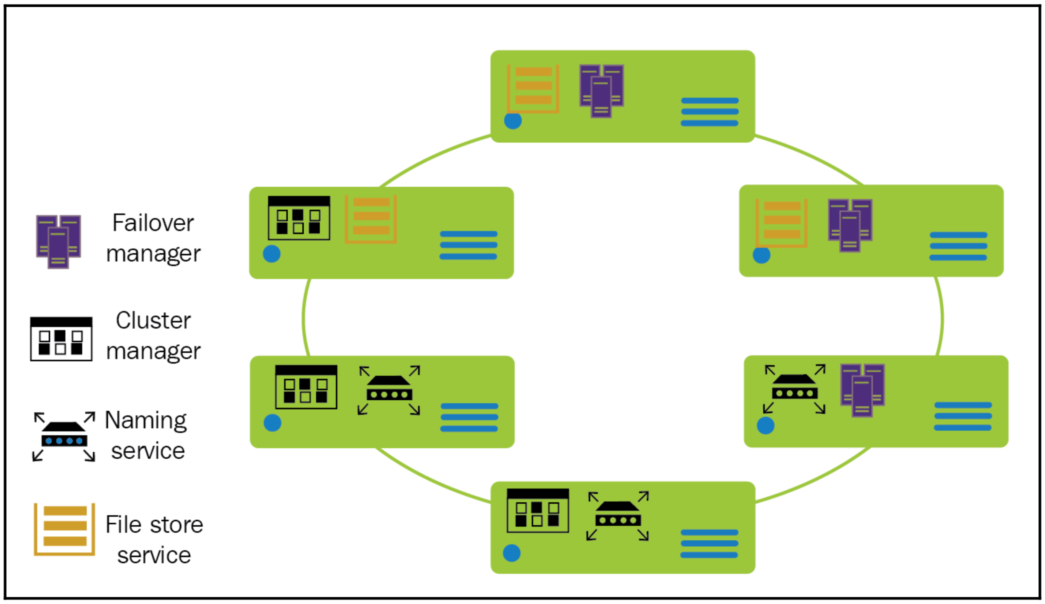
Each Service Fabric cluster node runs an agent that collects and sends load reports to a centralized resource balancer service. The resource balancer service is responsible for generating placement recommendations based on the load on the nodes and the placement requirements. The agent also sends failure reports and other events to a failover manager service. In a event of a change to the available node count, such as when a node fails or when a node is added to the cluster, the failover manager communicates with the resource balancer to get a placement recommendation. Once the failover manager receives a recommendation from the resource balancer service, it places a new service replica on the recommended node.

Cluster manager service

The cluster manager service is available over the HTTP Port `19080` and the TCP port `19000` of the cluster. It allows the Fabric client (available in the `System.Fabric` namespace), REST, and PowerShell clients to perform management operations, such as restarting a node, on the Service Fabric cluster.

Service placement

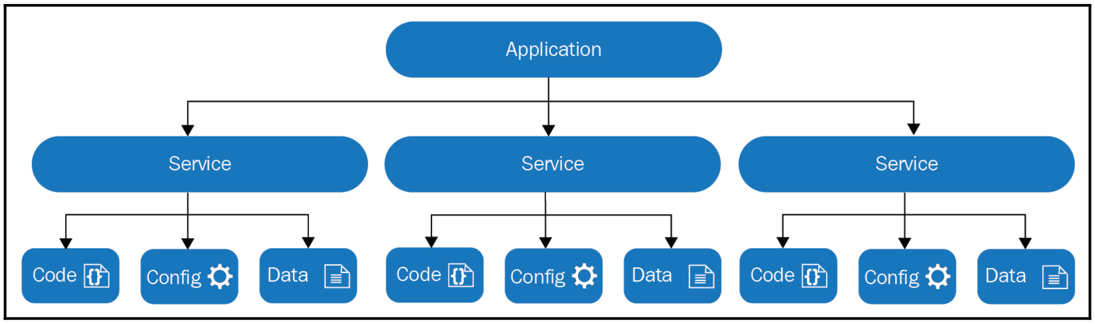
The System Services are deployed in a redundant manner across multiple nodes so that they are highly available. The following diagram shows a possible distribution of the **System Services** in a six node cluster:



System Services

Application model

A Service Fabric application is made up of one or more Microservices. Each Microservice consists of the executable binaries or code of the Microservice, the configuration or setting that is used by the Microservice at runtime and static data used by the Microservice. All the individual components of the Microservices can be versioned and upgraded independently:



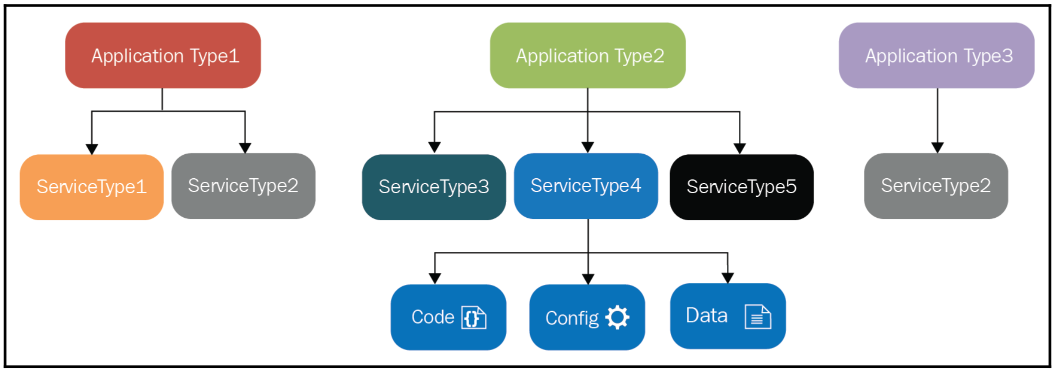
Application model

Applications and Microservices in Service Fabric are denoted by their type names. An application type can have a number of service types. You can create instances of application types (for example, by using the PowerShell command `New-ServiceFabricApplication`) and also service type (for example by using PowerShell command `New-ServiceFabricService`) which can have different settings and configurations, but the same core functionality.

The application types and service types are described through the application manifests (`ApplicationManifest.xml`) and service manifests (`ServiceManifest.xml`) respectively, which are XML files. These manifests serve as the templates against which applications can be instantiated from the cluster's image store.

Once you have installed the Service Fabric SDK, you can find the schema definition for the `ServiceManifest.xml` and `ApplicationManifest.xml` files saved in your developer machine at: `C:\Program Files\Microsoft SDKs\Service Fabric\schemas\ServiceFabricServiceModel.xsd`.

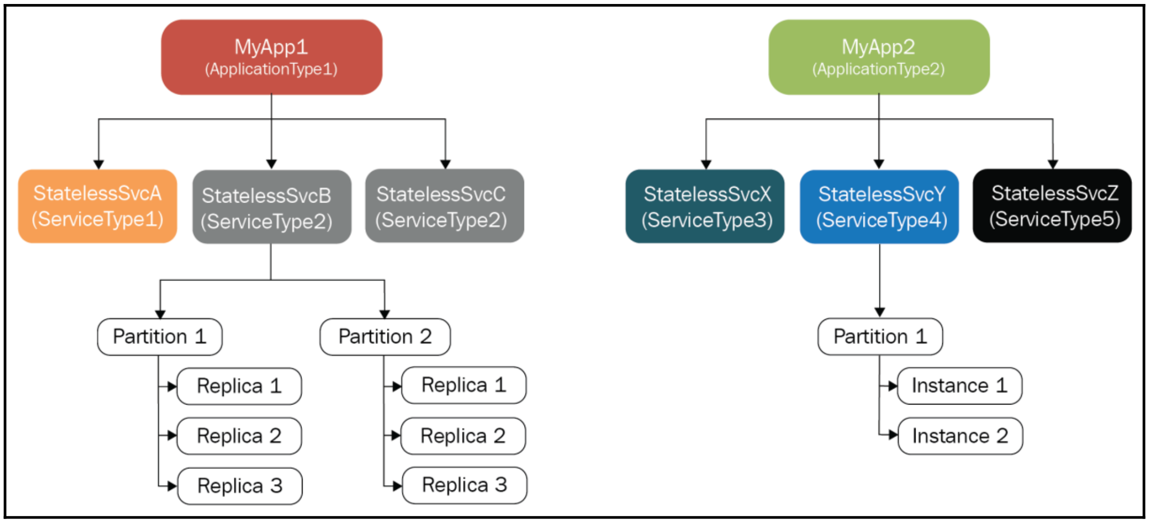
A Service Fabric node can host multiple application instances, some of which might belong to the same application; however, the code for different application instances will run as separate processes. Each application instance can be upgraded and managed independently. The following diagram shows how application types are composed of service types, which in turn are composed of code, configuration, and packages. Each of the service types would include some or all of the code, configuration and data packages:



Service Types

A stateful Microservice can split its state and save its state across several partitions. A partitioned service can be deployed across nodes in a cluster. A partition is made highly available through replication. A partition has a primary replica and may have multiple secondary replicas. The state data of a stateful Microservice is synchronized automatically across replicas. Whenever a primary replica goes down, a secondary replica is promoted to primary to ensure availability. Later, the number of secondary replicas is brought back up to ensure there is enough redundancy available. There can be one or more instances of a service type active in the cluster.

The following diagram represents the relationship between applications and service instances, partitions, and replicas:



Instances, Partitions, and Replicas

Programming model

Service Fabric provides two .NET framework programming models for you to build your Microservices. Both the frameworks provide a minimal set of APIs that allow Service Fabric to manage your Microservices. Additionally, Service Fabric allows you to package your application binaries as a compiled executable program written in any language and host it on a Service Fabric cluster. Let's take a look at the programming models.

Guest Executables

You can package an arbitrary executable, written in any language, such as Node.js, Java, or native applications in Azure Service Fabric, and host it on a Service Fabric cluster. These type of applications are called **Guest Executables**. Guest Executables are treated by Service Fabric like stateless services. Service Fabric handles orchestration and simple execution management of the executable, ensuring it stays up and running according to the service description. However, since the executables do not interact with the Service Fabric platform through Service Fabric APIs, they do not have access to the full set of features the platform offers, such as custom health and load reporting, service endpoint registration, and stateful compute.

Reliable Services

The Reliable Services framework is used for writing Microservices using traditional .NET constructs. The framework helps Service Fabric provide reliability, availability, consistency and scalability to your service. Using the Reliable Services model, you can create both stateless and stateful Microservices.

- **Stateless Reliable Service**: A stateless Microservice in Service Fabric does not contain any state data that needs to be stored reliably or made highly available. There is no affinity of requests to the services, therefore stateless services store any state data in external store such as Azure SQL database or Redis cache.
- **Stateful Reliable Service**: A stateful Microservice in Service Fabric can reliably maintain state that is co-located with the executing code of the Microservice. The Service Fabric platform ensures that the state data is replicated, consistent and highly available. The Service Fabric application programming framework provides a few collection types that can be used to store state data with reliability guarantees. These collection classes are part of `Microsoft.ServiceFabric.Data.Collections` namespace and are called **Reliable Collections**. High availability and strong consistency of state data in these collections is guaranteed by writing transactional state data to a majority quorum of replicas, including the primary replica.

At the time of writing, the namespace `Microsoft.ServiceFabric.Data.Collections` contains three collections:

- `ReliableDictionary`: `ReliableDictionary` is similar to the `ConcurrentDictionary` collection in the `System.Collections.Concurrent` namespace. However, unlike the `ConcurrentDictionary` collection, it 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.
- `ReliableQueue`: `ReliableQueue` is similar to the `ConcurrentQueue` collection in the `System.Collections.Concurrent` namespace. Just like the `ReliableDictionary` collection, it represents a replicated, transactional, and asynchronous collection of values. However, it is a strict **first-in, first-out** (**FIFO**) queue. The value stored in a `ReliableQueue` can be of any type.

- `ReliableConcurrentQueue`: `ReliableConcurrentQueue` is a new Reliable Collection of persisted, replicated values that allows concurrent reads and writes with best-effort. FIFO ordering. `ReliableConcurrentQueue` supports higher throughput and therefore does not guarantee FIFO behavior like the `ReliableQueue` does. Also, while `ReliableQueue` restricts concurrent consumers and producers to a maximum of one each, `ReliableConcurrentQueue` imposes no such restriction, allowing multiple concurrent consumers and producers.

We will learn more about Reliable Services in the next chapter.

Reliable Actors

You must be familiar with the object-oriented programming paradigm which models problems as a number of interacting objects that contain some data which forms the state of the object. The Actor model of computation breaks down problems into a number of Actors which can function independently and interact with each other by messaging.

The Service Fabric Reliable Actors API provides a high-level abstraction for modelling your Microservices as a number of interacting Actors. This framework is based on the Virtual Actor pattern, which was invented by the Microsoft research team and was released with the codename **Orleans**.

The Actor pattern has been implemented in multiple languages through various frameworks such as Akka.NET, ActorKit, and Quasar. However, unlike Actors implemented on other platforms, the Actors in Orleans are virtual. The Orleans runtime manages the location and activation of Actors similarly to the way that the virtual memory manager of an operating system manages memory pages. It activates an Actor by creating an in-memory copy (an activation) on a server, and later it may deactivate that activation if it hasn't been used for some time. If a message is sent to the Actor and there is no activation on any server, then the runtime will pick a location and create a new activation there.
You can read more about the *Orleans* project from the Microsoft Research website at: `https://www.microsoft.com/en-us/research/project/orleans-virtual-actors/`.

As a general guidance, you should consider using the Actor pattern in the following scenarios:

- Your system can be described by a number of independent and interactive units (or Actors), each of which can have its own state and logic
- You do not have significant interaction with external data sources and your queries do not span across the Actors
- Your Actors can execute as single-threaded components and do not execute blocking I/O operations

The Service Fabric Reliable Actors API is built on top of the Service Fabric Reliable Services programming model and each Reliable Actor service you write is actually a partitioned, stateful Reliable Service. The Actor state can be stored in memory, on disk, or in external storage.

Since the Service Fabric Actors are virtual, they have a perpetual lifetime. When a client needs to talk to an Actor, the Service Fabric will activate the Actor if it has not been activated or if it has been deactivated. After an Actor has been lying unused for some time, the Service Fabric will deactivate it to save resources. We will read more about Reliable Actors later in this book.

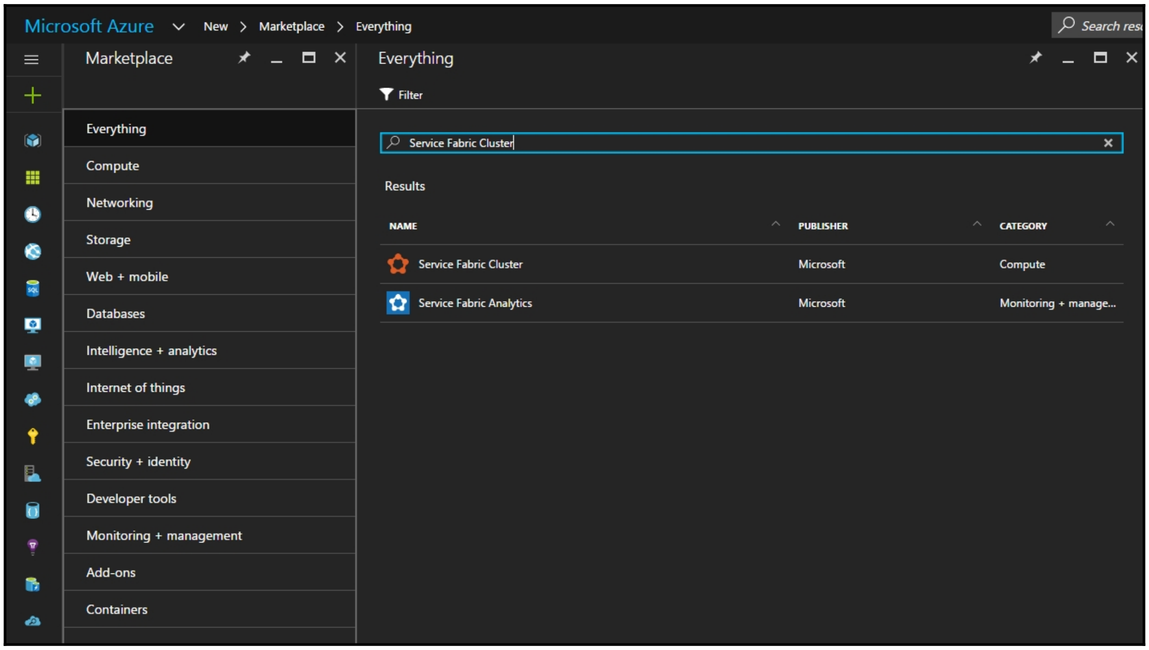
Creating a cluster on Azure

We have discussed quite a few concepts by now. Let's create a Service Fabric cluster in Azure and map the concepts that we learnt with the components. These steps to create a Service Fabric cluster on Azure are also documented on the Microsoft Azure documentation site at: `https://azure.microsoft.com/en-us/documentation/articles/service-fabric-cluster-creation-via-portal/`.

Although you can work with almost all the samples in this book on your system, you would need a Microsoft Azure subscription to deploy your production workload. We recommend that you get a Microsoft Azure subscription now. You can get started with a free one month trial at: `https://azure.microsoft.com/en-us/free/`.

Perform the following steps to create a **Service Fabric Cluster**:

1. Sign in to the Azure Management Portal at: `https://portal.azure.com`.
2. Click **New** to add a new resource template. Search for the **Service Fabric Cluster** template in **Marketplace** under **Everything**:



NAME	PUBLISHER	CATEGORY
Service Fabric Cluster	Microsoft	Compute
Service Fabric Analytics	Microsoft	Monitoring + manage...

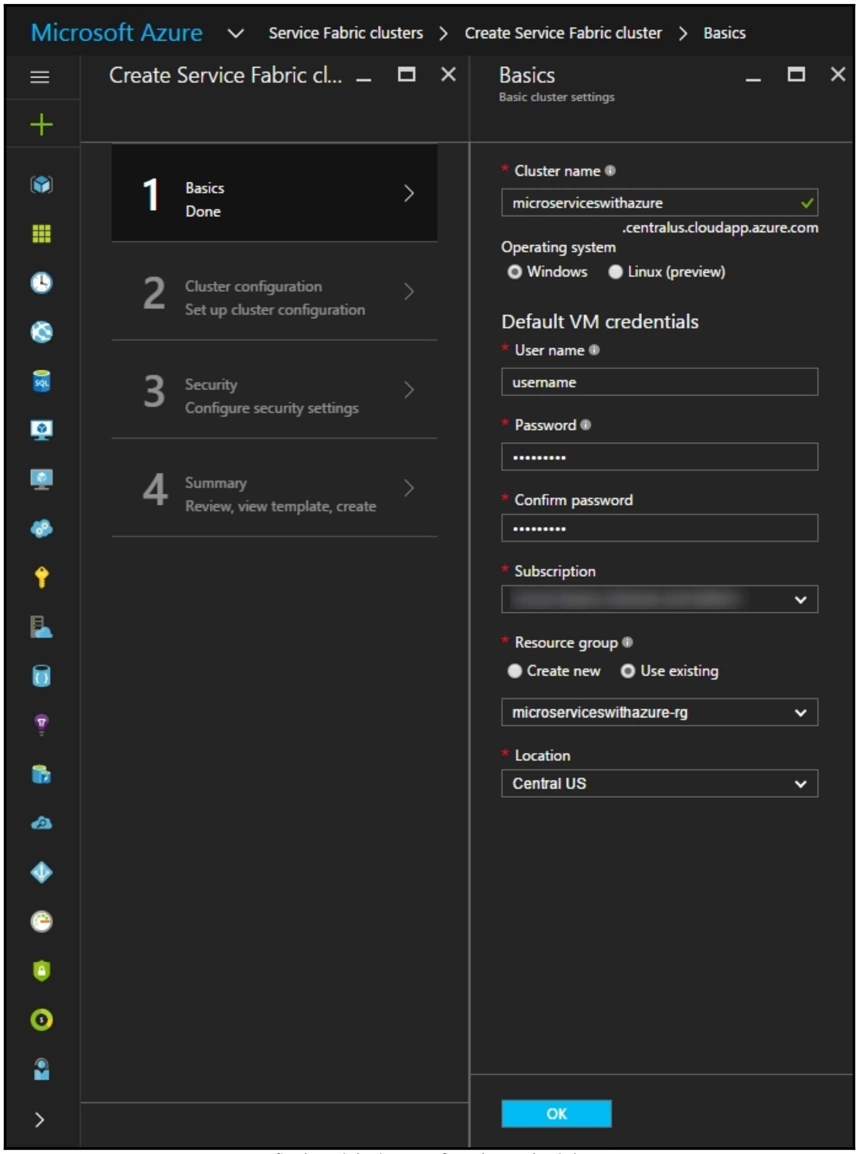
Service Fabric cluster template

3. Select **Service Fabric Cluster** from the list.
4. Navigate to the **Service Fabric Cluster** blade and click **Create**.

The **Create Service Fabric cluster** blade has the following four steps:

Basics

In the **Basics** blade you need to provide the basic details for your cluster:


Microsoft Azure
Service Fabric clusters > Create Service Fabric cluster > Basics
Create Service Fabric cl...
1 Basics Done
2 Cluster configuration Set up cluster configuration
3 Security Configure security settings
4 Summary Review, view template, create
Basics
Basic cluster settings
OK

* Cluster name: microserviceswithazure .centralus.cloudapp.azure.com

Operating system: [✓ Windows] Linux (preview)

Default VM credentials

* User name: username

* Password: •••••••••

* Confirm password: •••••••••

* Subscription:

* Resource group: Create new [✓ Use existing]

microserviceswithazure-rg

* Location: Central US

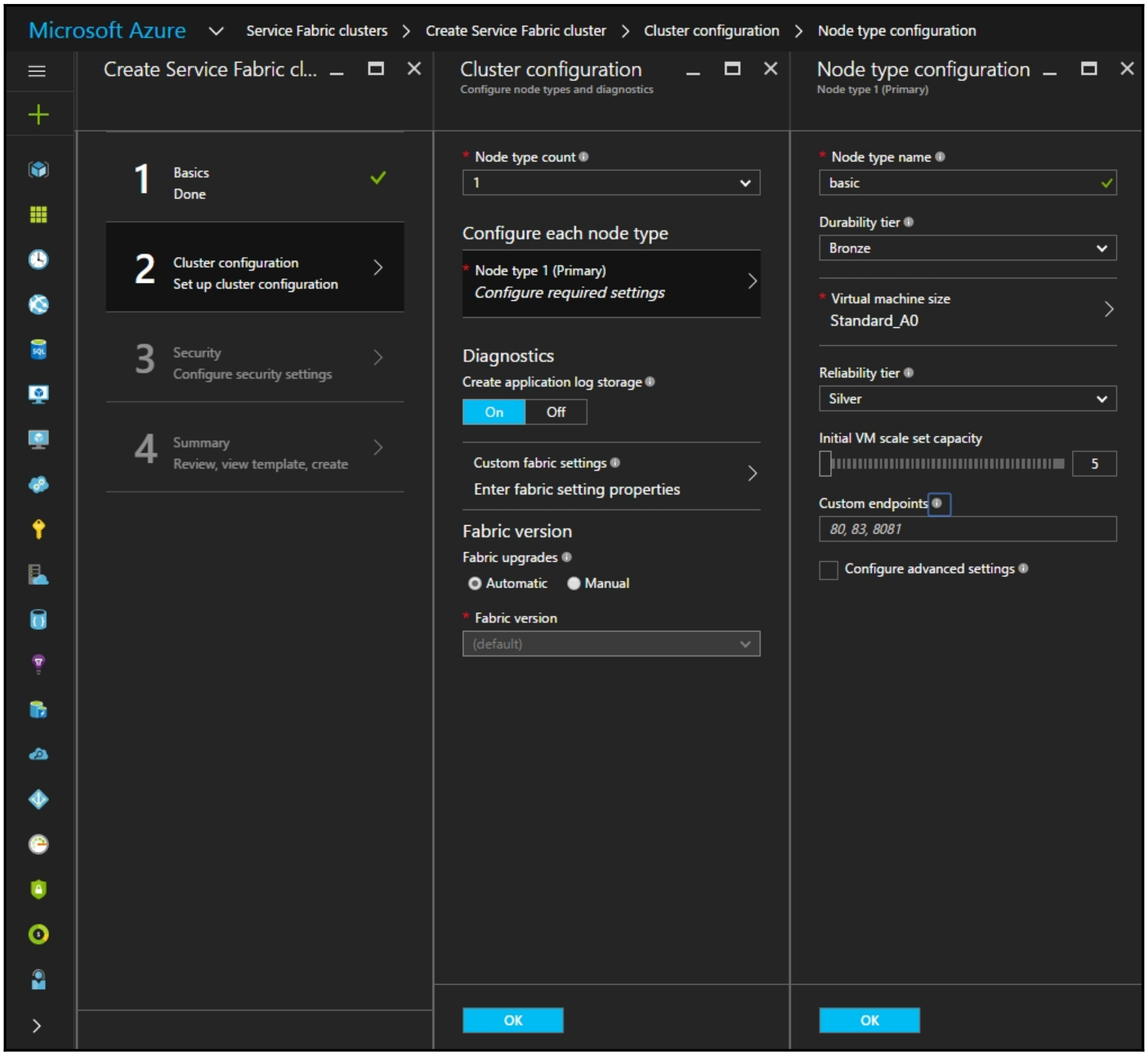
Service Fabric cluster configuration: Basic Blade

You need to provide the following details:

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.
5. Select the region in which you want to create the cluster. You must use the same region that your key vault is in.

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 **PlacementProperties** because a default placement property of **NodeTypeName** is added automatically:



Microsoft Azure > Service Fabric clusters > Create Service Fabric cluster > Cluster configuration > Node type configuration

Create Service Fabric cl...

1 Basics
Done ✓

2 Cluster configuration
Set up cluster configuration

3 Security
Configure security settings

4 Summary
Review, view template, create

Cluster configuration
Configure node types and diagnostics

* Node type count: 1

Configure each node type

* Node type 1 (Primary)
Configure required settings

Diagnostics
Create application log storage: [✓ On] Off

Custom fabric settings
Enter fabric setting properties

Fabric version
Fabric upgrades: [✓ Automatic] Manual

* Fabric version: (default)

OK

Node type configuration
Node type 1 (Primary)

* Node type name: basic ✓

Durability tier: Bronze

* Virtual machine size: Standard_A0

Reliability tier: Silver

Initial VM scale set capacity: 5

Custom endpoints: 80, 83, 8081

Configure advanced settings

OK

Service Fabric cluster configuration: Node Configuration Blade

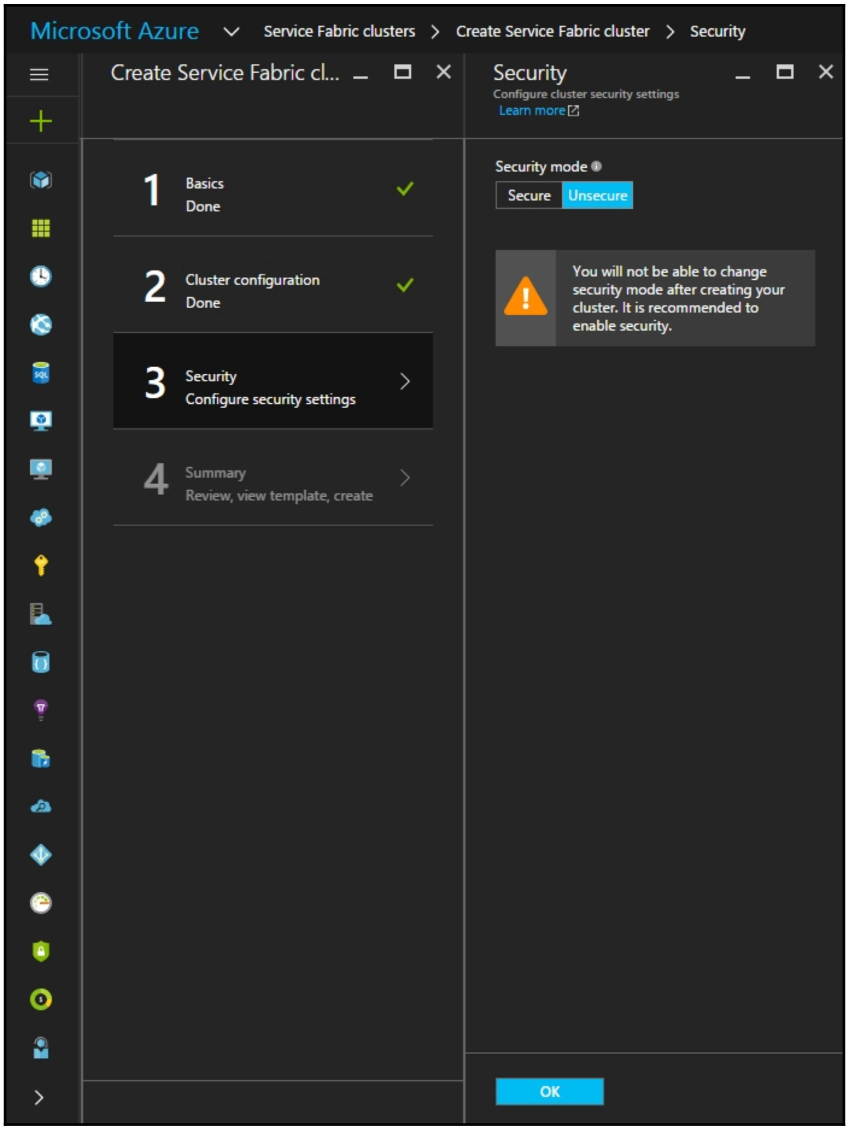
For configuring your cluster, you need to perform the following steps:

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**.

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 has 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**.
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 one 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.
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 upgrades** 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.

Security

The final step is to provide certificate information to secure the cluster. You can secure communication between the clients and the cluster and the communication between the nodes of the cluster using X509 certificates. The communication security can be implemented by adding X509 certificates to a key vault store and configuring the cluster to apply the certificates in this step. To keep this walk-through simple, we will not secure our cluster this time:


Microsoft Azure
Service Fabric clusters > Create Service Fabric cluster > Security
Create Service Fabric cl...
1 Basics Done
2 Cluster configuration Done
3 Security Configure security settings
4 Summary Review, view template, create
Security
Configure cluster security settings
Learn more
Security mode
Secure
Unsecure
OK

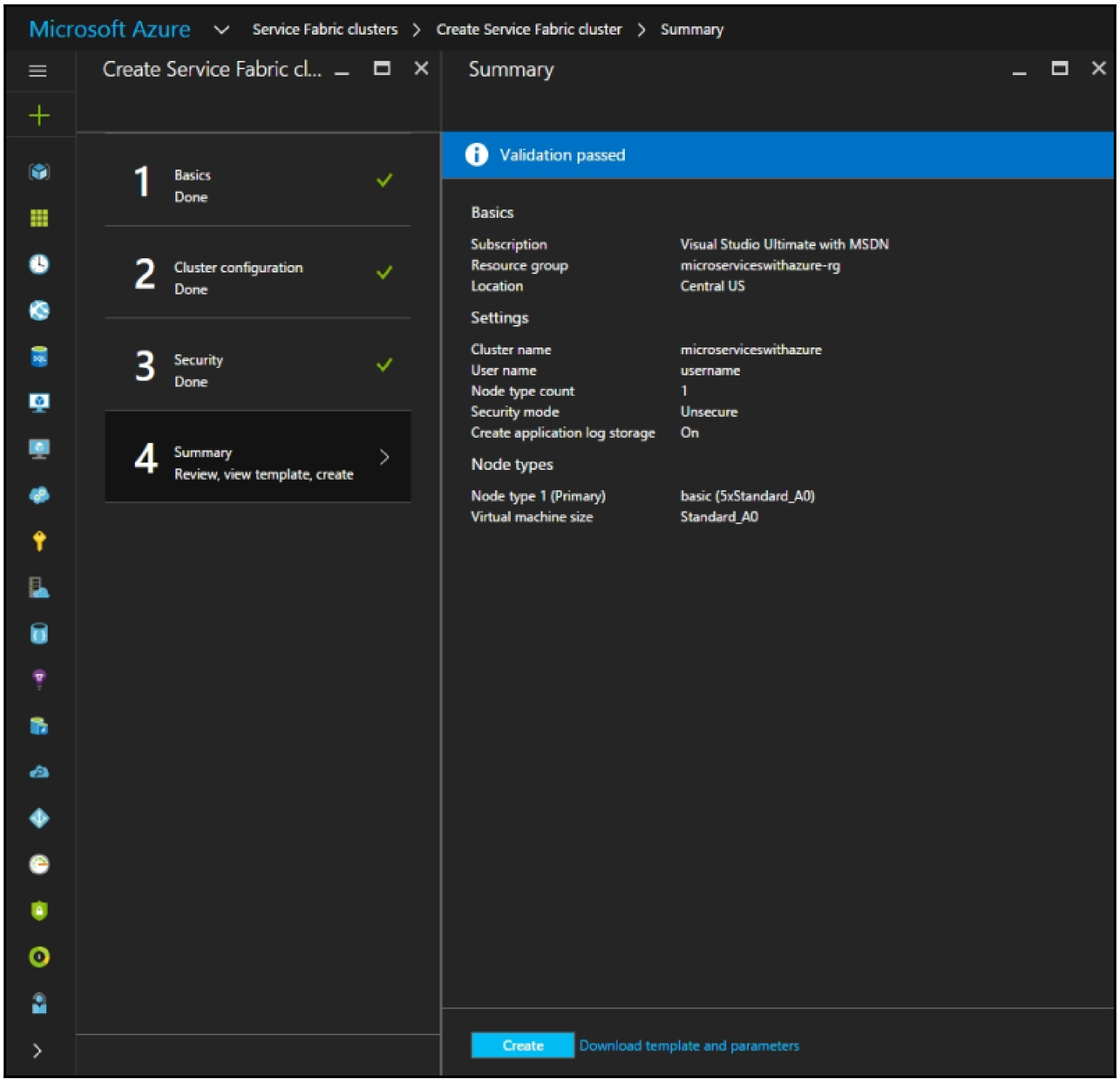
You will not be able to change security mode after creating your cluster. It is recommended to enable security.

Service Fabric cluster configuration: Security Blade

It is highly recommended that you always use a secured cluster for your production workloads. You have to plan for security at initial stages of development as an unsecure cluster cannot be secured later and a new cluster would need to be created if you decide to secure your cluster later. You can read more about securing your cluster in `Chapter 9`, *Securing and Managing Your Microservices*.

Summary

To complete the cluster creation, click **Summary** to see the configurations that you have provided, or download the Azure Resource Manager template that can be 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:


Microsoft Azure
Service Fabric clusters > Create Service Fabric cluster > Summary
Create Service Fabric cl...
1 Basics Done
2 Cluster configuration Done
3 Security Done
4 Summary Review, view template, create
Summary
Validation passed
Create
Download template and parameters

Basics

Subscription	Visual Studio Ultimate with MSDN
Resource group	microserviceswithazure-rg
Location	Central US

Settings

Cluster name	microserviceswithazure
User name	username
Node type count	1
Security mode	Unsecure
Create application log storage	On

Node types

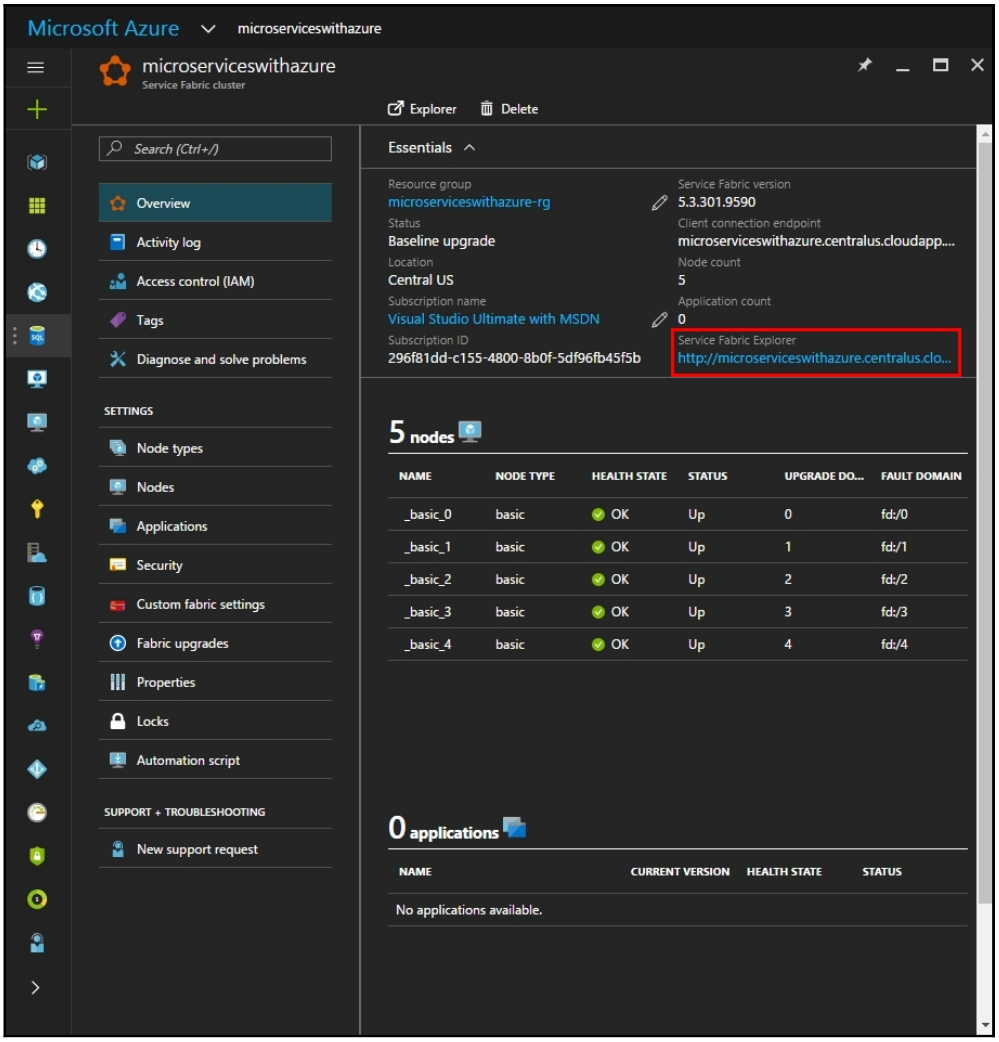
Node type 1 (Primary)	basic (5xStandard_A0)
Virtual machine size	Standard_A0

Service Fabric cluster configuration: Summary

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.

Viewing your cluster status

Once your cluster is created, you can inspect your cluster in the portal:



Resource group: microserviceswithazure-rg
Status: Baseline upgrade
Location: Central US
Subscription name: Visual Studio Ultimate with MSDN
Subscription ID: 296f81dd-c155-4800-8b0f-5df96fb45f5b
Service Fabric version: 5.3.301.9590
Client connection endpoint: microserviceswithazure.centralus.cloudapp....
Node count: 5
Application count: 0
Service Fabric Explorer: http://microserviceswithazure.centralus.clo...

5 nodes

NAME	NODE TYPE	HEALTH STATE	STATUS	UPGRADE DO...	FAULT DOMAIN
_basic_0	basic	✓ OK	Up	0	fd:/0
_basic_1	basic	✓ OK	Up	1	fd:/1
_basic_2	basic	✓ OK	Up	2	fd:/2
_basic_3	basic	✓ OK	Up	3	fd:/3
_basic_4	basic	✓ OK	Up	4	fd:/4

0 applications

NAME	CURRENT VERSION	HEALTH STATE	STATUS
No applications available.			

Service Fabric cluster details

To view your cluster status, perform the following steps:

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**.

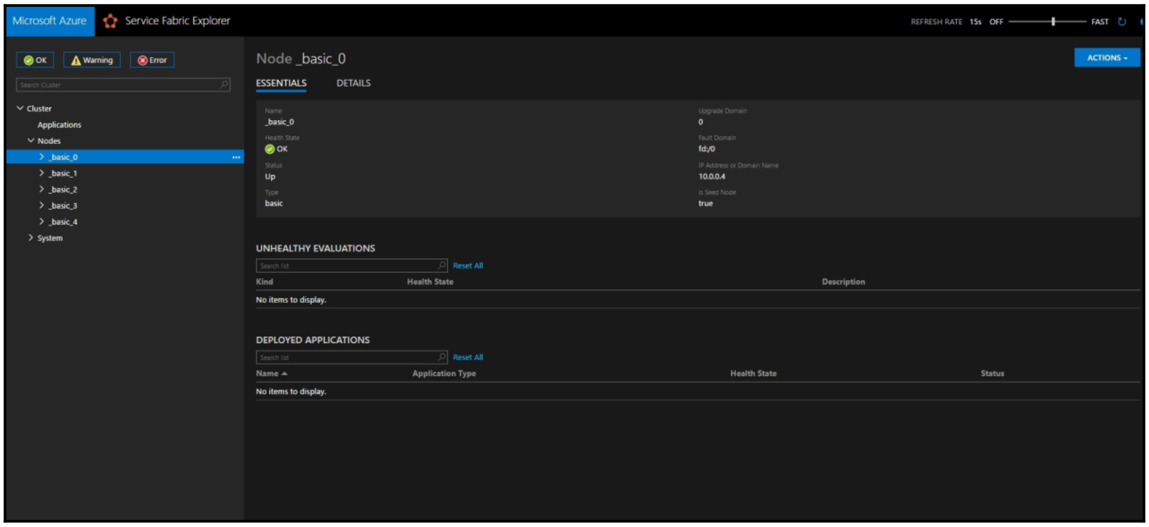
The **nodes** monitor section on the cluster's dashboard blade indicates the number of VMs that are healthy and not healthy. Click on the **Service Fabric Explorer** link now to explore the various components of your cluster.

You can create Service Fabric cluster using Azure Resource Manager. In fact, using ARM is the recommended strategy for deploying workloads. Using Resource Manager, you can repeatedly deploy your solution throughout the development lifecycle and have confidence that your resources are deployed in a consistent state The step-by-step guide to provision a Service Fabric cluster on Azure is available here: `https://azure.microsoft.com/en-us/documentation/articles/service-fabric-cluster-creation-via-arm/`.

Service Fabric Explorer

Service Fabric Explorer is a web-based tool that is built using HTML and AngularJS and is included in every cluster, including the local cluster, at port `19080`. You can access the explorer at `http(s)://clusteraddress:19080/Explorer`.

You can use the Service Fabric Explorer tool for inspecting and managing applications and nodes in an Azure Service Fabric cluster. The left-side section of **Service Fabric Explorer** provides a tree view of your cluster and to the right is a pane showing details of the selected item and an **ACTIONS** button with possible actions you can perform on the item:



Node _basic_0

Name: _basic_0	Upgrade Domain: 0
Health State: OK	Fault Domain: fd:/0
Status: Up	IP Address or Domain Name: 10.0.0.4
Type: basic	Is Seed Node: true

UNHEALTHY EVALUATIONS

Kind	Health State	Description
No items to display.		

DEPLOYED APPLICATIONS

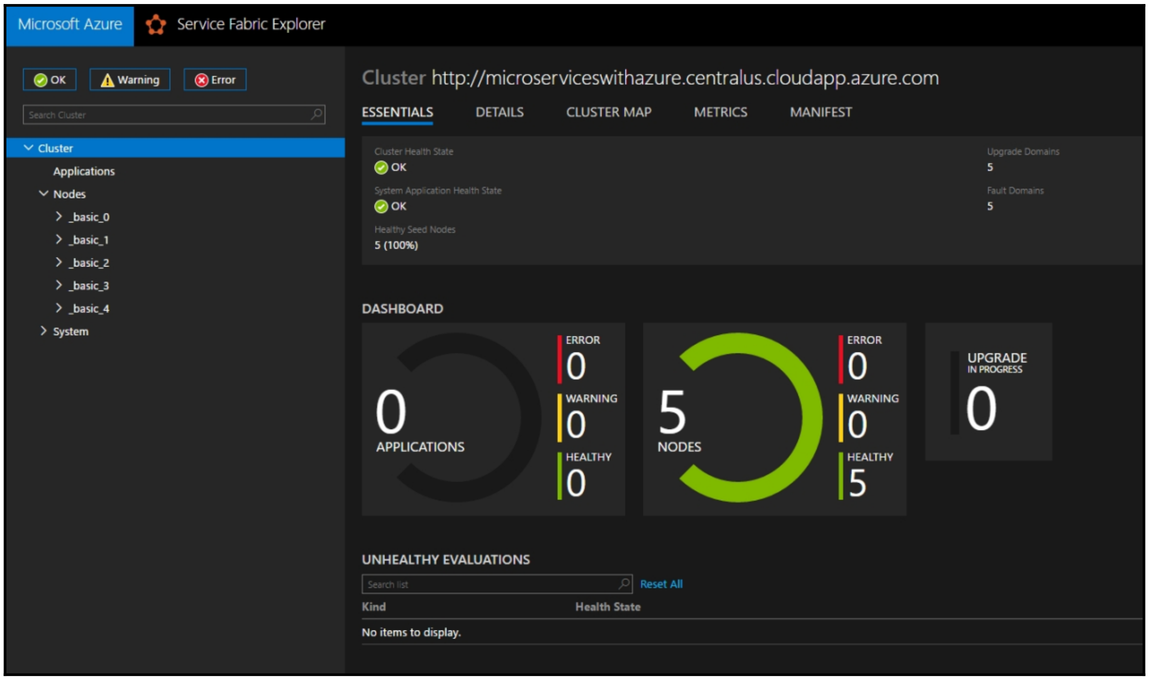
Name ▲	Application Type	Health State	Status
No items to display.			

Service Fabric Explorer

Let's take a look at the layout of Service Fabric Explorer.

Summary view

The cluster dashboard provides an overview of your cluster, including a summary of application and node health:



Cluster http://microserviceswithazure.centralus.cloudapp.azure.com

Cluster Health State: OK
System Application Health State: OK
Healthy Seed Nodes: 5 (100%)
Upgrade Domains: 5
Fault Domains: 5

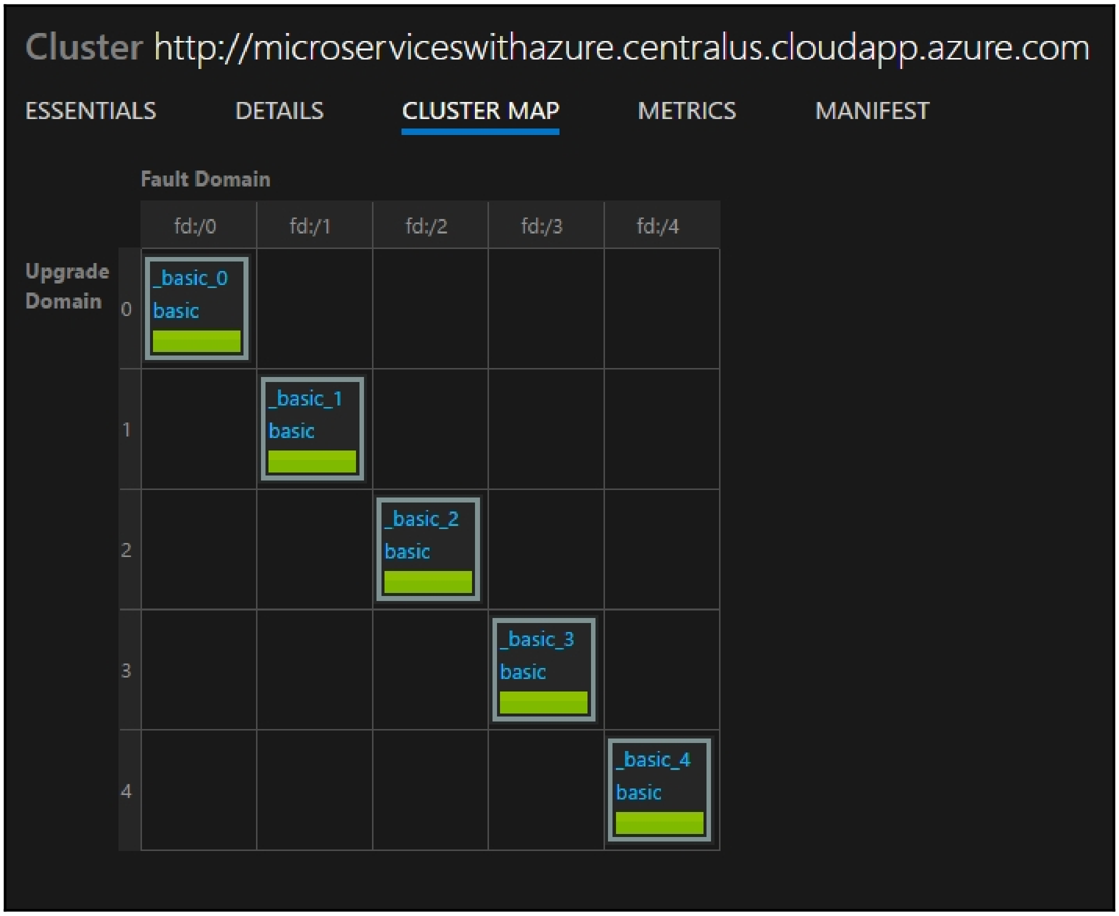
0 APPLICATIONS — ERROR 0, WARNING 0, HEALTHY 0
5 NODES — ERROR 0, WARNING 0, HEALTHY 5
UPGRADE IN PROGRESS 0

Kind	Health State
No items to display.	

Service Fabric Explorer Cluster Summary

Cluster Map

You can also view the placement of nodes in your cluster by clicking on the **Cluster Map** button. To ensure high availability of your services, your cluster nodes are placed in a table of fault domains and upgrade domains:



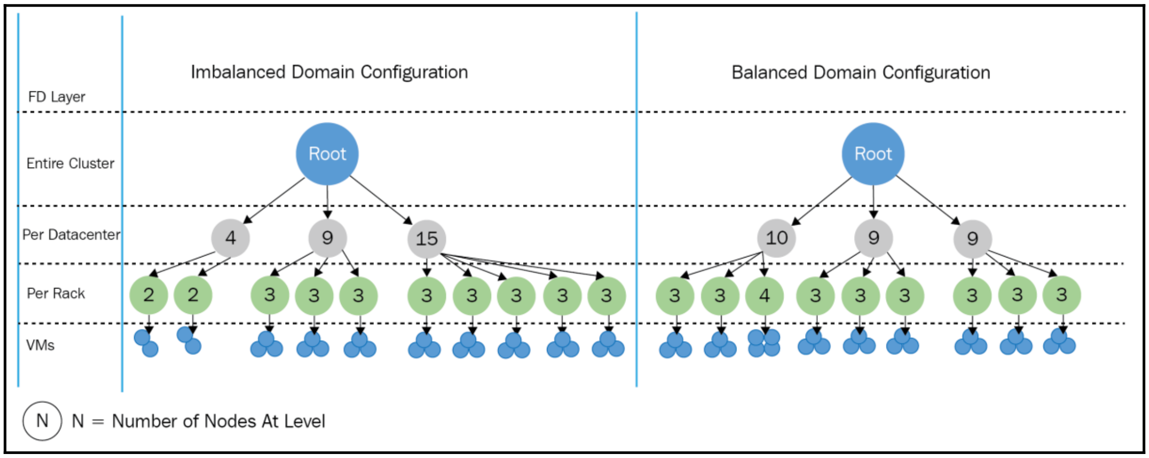
Cluster Map

At this point, you might be interested in knowing what fault domains and upgrade domains are.

Fault domains

A **fault domain** (**FD**) is an area of coordinated failure. FDs are not restricted to a rack of servers. Several other factors are considered while composing a fault domain such as connection to the same ethernet switch and connection to the same power supply. When you provision a Service Fabric cluster on Azure, your nodes are distributed across several Fault Domains. At runtime, the Service Fabric cluster resource manager spreads replicas of each service across nodes in different fault domains so that failure of a node does not bring down the application.

Service Fabric's cluster resource manager prefers to allocate replicas of your service in a balanced tree of fault domains so that failure of a particular domain does not cause failure of the application. The following is a representation of an imbalanced and a balanced domain configuration:


Imbalanced Domain Configuration
Balanced Domain Configuration
FD Layer
Entire Cluster
Per Datacenter
Per Rack
VMs
Root
4
9
15
2
3
Root
10
9
9
3
4
N = Number of Nodes At Level

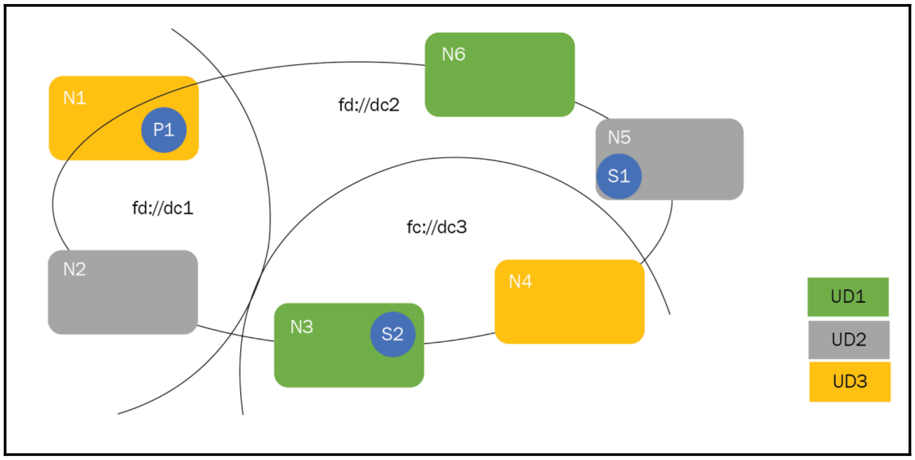
Fault domains

Note that in the imbalanced domain configuration, the extreme right domain may bring down the application if it contains all the replicas of a service. This is something that you don't need to be worry about if you are deploying your cluster in Azure, but something that you need to be aware of when you are deploying your cluster on premises.

Upgrade domains

The **upgrade domains** (**UDs**) are sets of nodes to which an upgrade package (application package, Service Fabric runtime update, or OS update) is applied simultaneously. During an upgrade, all the nodes that are part of the same UDs will go down simultaneously. Unlike fault domains, you can decide the number of upgrade domains that you want and the nodes that should be part of the each upgrade domain.

The following figure shows a setup where we have three upgrade domains spread across three fault domains. The replicas of a stateful service (one primary and two secondary) may be placed across these nodes. 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:



Upgrade domains

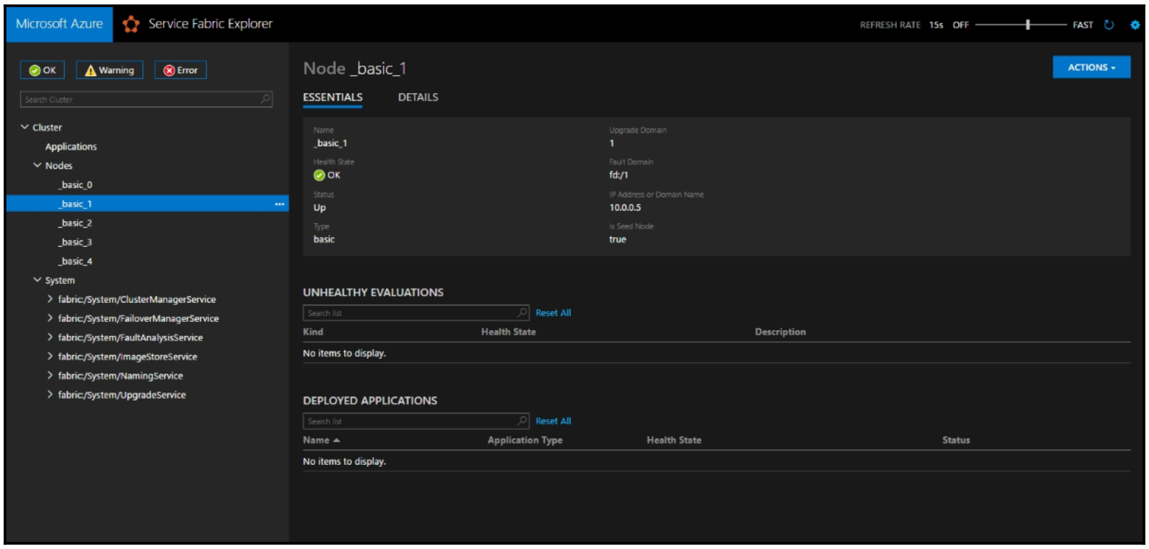
You need to be careful when deciding upon the number of upgrade domains that you require. Too few upgrade domains may affect the capacity of your service, as during an upgrade, the remaining nodes may not be able to cater to the load on application. On the other hand, too many upgrade domains may make the upgrade propagation a slow process.

Viewing applications and services

The cluster node contains two more subtrees, one for applications and one for nodes. Since, we haven't yet deployed an application to the cluster, this node contains no items. We will revisit this part of the explorer once we are done developing our first Service Fabric application.

Cluster nodes

Expanding the nodes tree will list the nodes present in the cluster. The details view will show you, which FD and UD each node is a part of. You can also view which application is currently executing on the node and, more importantly, which replica is running on it:


Microsoft Azure
Service Fabric Explorer
REFRESH RATE 15s OFF FAST
OK
Warning
Error
Search Cluster
Cluster
Applications
Nodes
_basic_0
_basic_1
_basic_2
_basic_3
_basic_4
System
fabric:/System/ClusterManagerService
fabric:/System/FailoverManagerService
fabric:/System/FaultAnalysisService
fabric:/System/ImageStoreService
fabric:/System/NamingService
fabric:/System/UpgradeService
ACTIONS
ESSENTIALS
DETAILS
Search list
Reset All

Node _basic_1

Name: _basic_1
Health State: OK
Status: Up
Type: basic
Upgrade Domain: 1
Fault Domain: fd:/1
IP Address or Domain Name: 10.0.0.5
Is Seed Node: true

UNHEALTHY EVALUATIONS

Kind	Health State	Description
No items to display.		

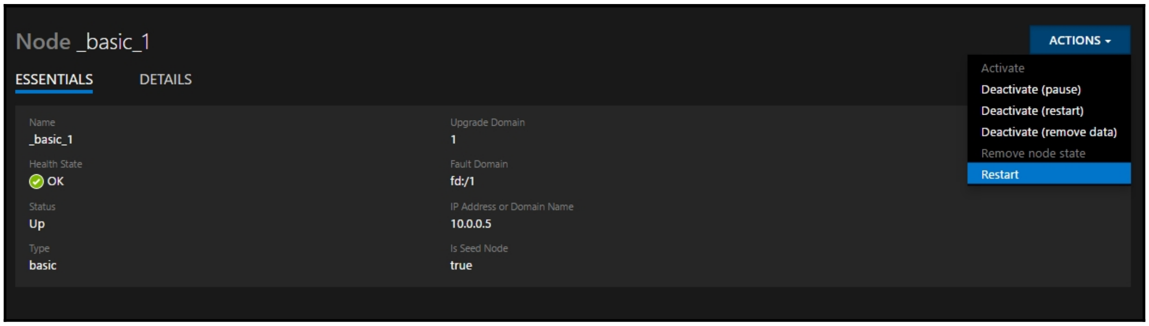
DEPLOYED APPLICATIONS

Name	Application Type	Health State	Status
No items to display.			

Cluster Nodes

Actions

The **ACTIONS** button is available on the nodes, applications, and services view in the explorer. You can invoke actions on the particular element through this button. For instance, in the **Nodes** view, you can restart a node by selecting **Restart** from the **ACTIONS** menu:



Name
_basic_1
Health State
OK
Status
Up
Type
basic
Upgrade Domain
1
Fault Domain
fd:/1
IP Address or Domain Name
10.0.0.5
Is Seed Node
true

Actions Button

The following table lists the actions available for each entity:

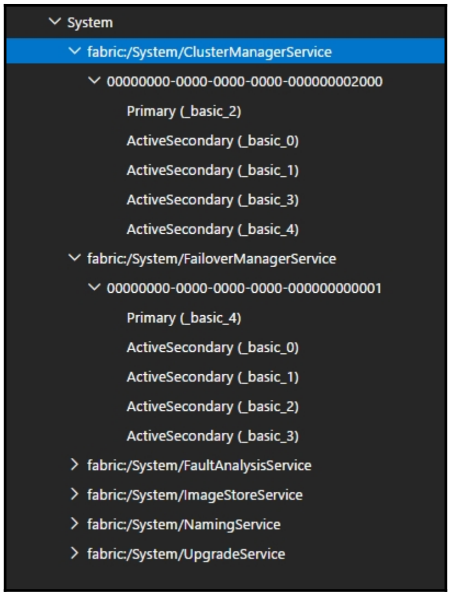
Entity	**Action**	**Description**
Application type	Unprovision type	Remove the application package from the cluster's image store. Requires all applications of that type to be removed first.
Application	Delete application	Delete the application, including all its services and their state (if any).
Service	Delete service	Delete the service and its state (if any).
Node	Activate	Activate the node.
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.

Entity	Action	Description
Node	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.
Node	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.
Node	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.

System

The various system services that we previously discussed are listed in this node:



```
v System
  v fabric:/System/ClusterManagerService
    v 00000000-0000-0000-0000-000000002000
        Primary (_basic_2)
        ActiveSecondary (_basic_0)
        ActiveSecondary (_basic_1)
        ActiveSecondary (_basic_3)
        ActiveSecondary (_basic_4)
  v fabric:/System/FailoverManagerService
    v 00000000-0000-0000-0000-000000000001
        Primary (_basic_4)
        ActiveSecondary (_basic_0)
        ActiveSecondary (_basic_1)
        ActiveSecondary (_basic_2)
        ActiveSecondary (_basic_3)
  > fabric:/System/FaultAnalysisService
  > fabric:/System/ImageStoreService
  > fabric:/System/NamingService
  > fabric:/System/UpgradeService
```

System Services

If you expand the individual service nodes, you will be able to see the placement of the services. Notice that all the services have been deployed to multiple nodes; however, the primary and secondary replicas of each service have been spread across multiple fault and upgrade domains for instance, in the preceding figure, you can see that the primary replica of the **ClusterManagerService** is deployed on the node named (**_basic_2**) while that of the **FailoverManagerService** is deployed on the node named (**_basic_4**).

Preparing your system

To get started with developing applications on Service Fabric, you will need to the following:

1. Install the runtime, SDK, and tools.
2. A Service Fabric cluster to deploy your applications.
3. Configure PowerShell to enable SDK script execution.

We are going to use C#, Visual Studio 2015, and Windows 10 to develop all samples in this book.

You can find the steps to to prepare your development environment on various operating systems at this link: `https://azure.microsoft.com/en-us/documentation/articles/service-fabric-get-started/`.

To install the SDK, tools, and Service Fabric runtime, use the Web Platform Installer (Visual Studio 2015) or enable Service Fabric workload (Visual Studio 2017). You can read more about these options at:

`https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-get-started`.

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
```

After you have completed the installation steps, you should be able to find the Service Fabric local cluster manager installed in your system, along with new Service Fabric project templates in your Visual Studio. Let's get started with building our first application on Service Fabric in the next chapter.

Summary

In this chapter, we covered the features of Service Fabric which make it an ideal platform to host Microservices. We discussed how Service Fabric acts as an orchestrator for managing Microservices.

In the second section of the chapter, we covered the architecture of Service Fabric in detail and studied the cluster services. Next, we discussed the infrastructure model, system services, and programming models available to build and host applications on Service Fabric.

Finally, we walked through the steps for creating a Service Fabric cluster on Azure and preparing our system to get started with developing applications on Service Fabric.

In the next chapter, we will dive deeper into building applications on Service Fabric.

4
Hands-on with Service Fabric – Guest Executables

Service Fabric as a platform supports multiple programming models, each of which is best suited for specific scenarios. Each programming model offers different levels of integration with the underlying management framework. Better integration leads to more automation and fewer overheads. Picking the right programming model for your application or service is the key to efficiently utilizing the capabilities of Service Fabric as a hosting platform. Let's take a deeper look into these programming models.

To start with, let's look at the least integrated hosting option – **Guest Executables**. Native Windows applications or application code using Node.js or Java can be hosted on Service Fabric as a Guest Executable. These executables can be packaged and pushed to a Service Fabric cluster like any other services. As the cluster manager has minimal knowledge about the executable, features such as custom health monitoring, load reporting, state store, and endpoint registration cannot be leveraged by the hosted application. However, from a deployment standpoint, a guest executable is treated like any other service. This means that for a guest executable, Service Fabric cluster manager takes care of high availability, application lifecycle management, rolling updates, automatic failover, high-density deployment, and load balancing.

As an orchestration service, Service Fabric is responsible for deploying and activating an application or application service within a cluster. It is also capable of deploying services within a container image. This programming model is addressed as Guest Containers. The concept of containers is best explained as an implementation of operating system level virtualization. They are encapsulated deployable components running on isolated process boundaries sharing the same kernel. Deployed applications and their runtime dependencies are bundles within the container with an isolated view of all operating system constructs. This makes containers highly portable and secure. The Guest Container programming model is usually chosen when this level of isolation is required for the application. As containers don't have to boot an operating system, they have fast boot up time and are comparatively small in size.

A prime benefit of using Service Fabric as a platform is the fact that it supports heterogeneous operating environments. Service Fabric supports two types of containers to be deployed as Guest Containers:

- Docker Containers on Linux
- Windows Server Containers

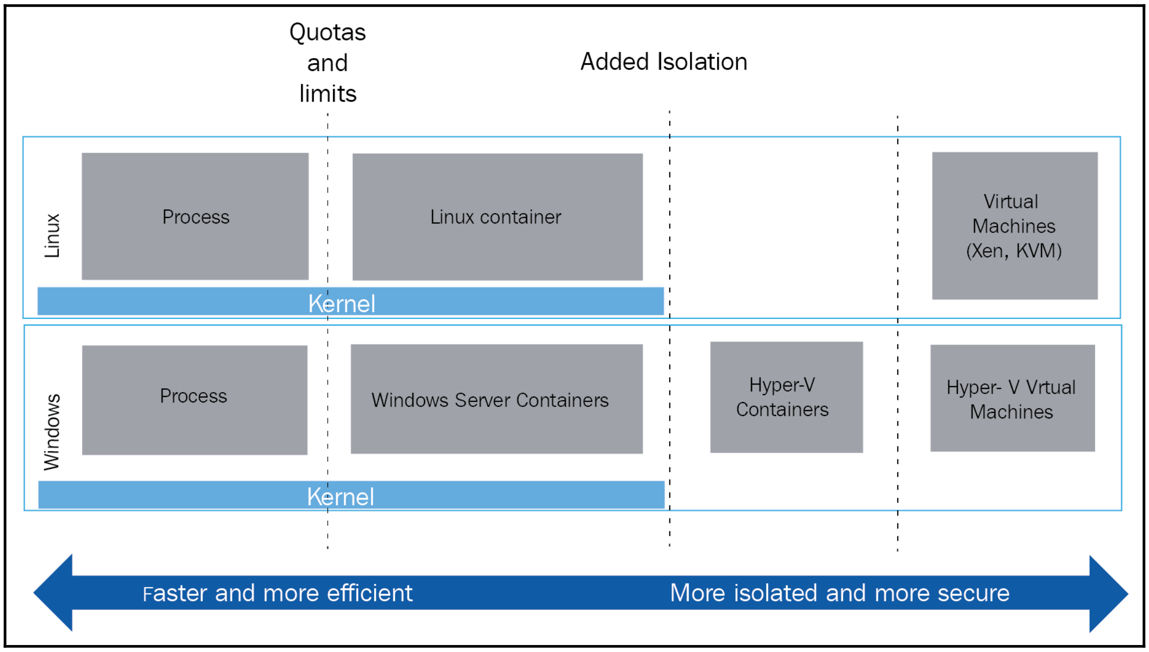
Container images for Docker Containers are stored in Docker Hub and Docker APIs are used to create and manage the containers deployed on a Linux kernel.

Service Fabric supports two different types of containers in Windows Server 2016 with different levels of isolation:

- Windows Server Containers
- Windows Hyper-V Containers

Windows Server Containers are similar to Docker Containers in terms of the isolation they provide. Windows Hyper-V Containers offer a higher degree of isolation and security by not sharing the operating system kernel across instances. These are ideally used when a higher level of security isolation is required, such as systems requiring hostile multitenant hosts.

The following figure illustrates the different isolation levels achieved by using these containers:



Container isolation levels

The Service Fabric application model treats containers as an application host which can in turn host service replicas. There are three ways of utilizing containers within a Service Fabric application mode. Existing applications such as Node.js and JavaScript applications, or other executables can be hosted within a container and deployed on Service Fabric as a Guest Container. A Guest Container is treated similar to a Guest Executable by the Service Fabric runtime. The second scenario supports deploying stateless services inside a container hosted on Service Fabric. Stateless services using Reliable Services and Reliable Actors can be deployed within a container. The third option is to deploy stateful services in containers hosted on Service Fabric. This model also supports Reliable Services and Reliable Actors.

Service Fabric offers several features to manage containerized Microservices. These include container deployment and activation, resource governance, repository authentication, port mapping, container discovery and communication, and the ability to set environment variables. Later in this chapter, we will explore ways of deploying containers on Service Fabric.

While containers offer a good level of isolation, it is still heavy in terms of deployment footprint. Service Fabric offers a simpler, powerful programming model to develop your services, which they call Reliable Services. Reliable Services let you develop stateful and stateless services which can be directly deployed on Service Fabric clusters. For stateful services, the state can be stored close to the compute by using Reliable Collections. High availability of the state store and replication of the state is taken care by the Service Fabric cluster management services. This contributes substantially to the performance of the system by improving the latency of data access. Reliable Services come with a built-in pluggable communication model which supports HTTP with Web API, WebSockets, and custom TCP protocols out-of-the-box.

A Reliable Service is addressed as stateless if it does not maintain any state within it or if the scope of the state stored is limited to a service call and is entirely disposable. This means that a stateless service does not need to persist, synchronize, or replicate state. A good example for this service is a weather service such as MSN weather service. A weather service can be queried to retrieve weather conditions associated with a specific geographical location. The response is totally based on the parameters supplied to the service. This service does not store any state. Although stateless services are simpler to implement, most of the services in real life are not stateless. They either store state in an external state store or an internal one. Web frontend hosting APIs or web applications are good use cases to be hosted as stateless services.

A stateful service typically will utilize various mechanisms to persist its state. The outcome of a service call made to a stateful service is usually influenced by the state persisted by the service. A service exposed by a bank to return the balance on an account is a good example for a stateful service. The state may be stored in an external data store such as Azure SQL Database, Azure Blobs, or Azure Table store. Most services prefer to store the state externally, considering the challenges around reliability, availability, scalability, and consistency of the data store. With Service Fabric, state can be stored close to the compute by using Reliable Collections.

To make things more lightweight, Service Fabric also offers a programming model based on the Virtual Actor pattern. This programming model is called Reliable Actors. The Reliable Actors programming model is built on top of Reliable Services. This guarantees the scalability and reliability of the services. An Actor can be defined as an isolated, independent unit of compute and state with single-threaded execution. Actors can be created, managed, and disposed independent of each other.

A large number of Actors can coexist and execute at a time. Service Fabric Reliable Actors are a good fit for systems which are highly distributed and dynamic by nature. Every Actor is defined as an instance of an Actor type, the same way an object is an instance of a class. Each Actor is uniquely identified by an Actor ID. The lifetime of Service Fabric Actors is not tied to their in-memory state. As a result, Actors are automatically created the first time a request for them is made. Reliable Actor's garbage collector takes care of disposing unused Actors in memory.

Now that we understand the programming models, let's take a look at how the services deployed on Service Fabric are discovered and how the communication between services takes place.

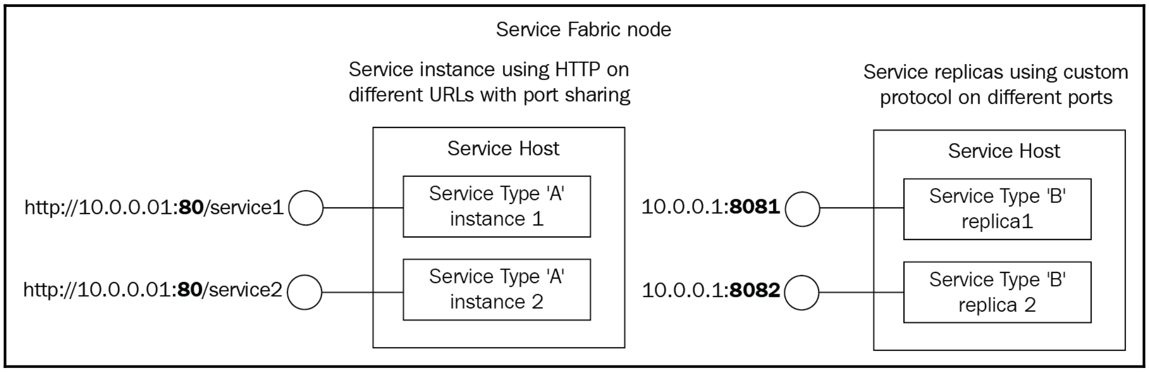
Service Fabric discovery and communication

An application built on top of Microservices is usually composed of multiple services, each of which runs multiple replicas. Each service is specialized in a specific task. To achieve an end-to-end business use case, multiple services will need to be stitched together. This requires services to communicate to each other. A simple example would be a web frontend service communicating with the middle-tier services, which in turn connects to the backend services to handle a single user request. Some of these middle-tier services can also be invoked by external applications.

Services deployed on Service Fabric are distributed across multiple nodes in a cluster of virtual machines. The services can move across dynamically. This distribution of services can either be triggered by a manual action or be result of Service Fabric cluster manager rebalancing services to achieve optimal resource utilization. This makes communication a challenge as services are not tied to a particular machine. Let's understand how Service Fabric solves this challenge for its consumers.

Service protocols

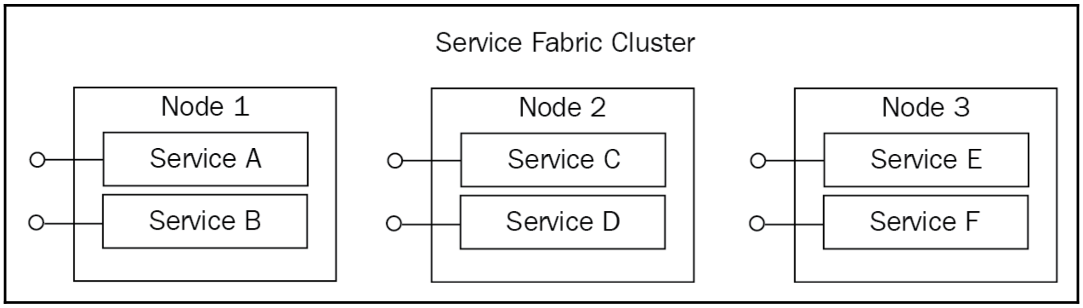
Service Fabric, as a hosting platform for Microservices, does not interfere in the implementation of the service. On top of this, it also lets services decide on the communication channels they want to open. These channels are addressed as service endpoints. During service initiation, Service Fabric provides the opportunity for the services to set up the endpoints for incoming requests on any protocol or communication stack. The endpoints are defined according to common industry standards, that is *IP:Port*. It is possible that multiple service instances share a single host process. In which case, they either have to use different ports or a port sharing mechanism. This will ensure that every service instance is uniquely addressable:


Service Fabric node
Service instance using HTTP on different URLs with port sharing
Service replicas using custom protocol on different ports
Service Host
Service Type 'A' instance 1
Service Type 'A' instance 2
http://10.0.0.01:80/service1
http://10.0.0.01:80/service2
Service Host
Service Type 'B' replica1
Service Type 'B' replica 2
10.0.0.1:8081
10.0.0.1:8082

Service endpoints

Service discovery

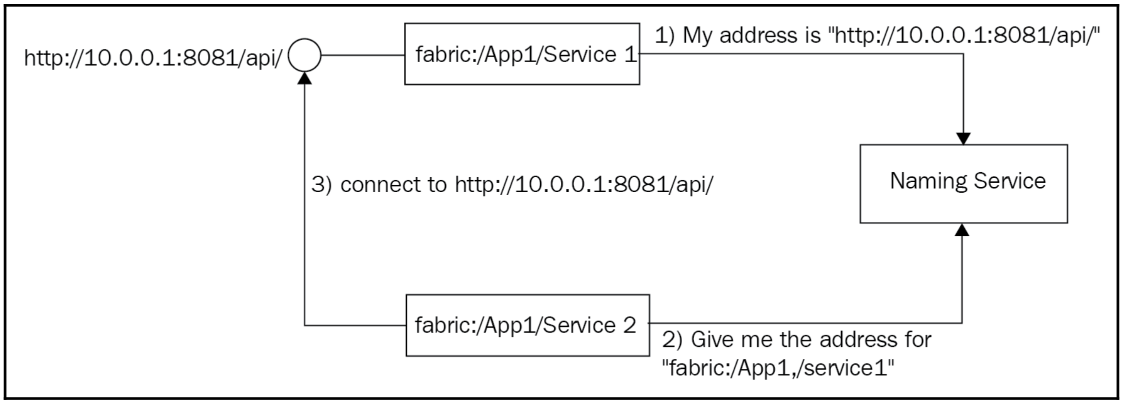
Service Fabric can rebalance services deployed on a cluster as a part of orchestration activities. This can be caused by resource balancing activities, failovers, upgrades, scale-outs, or scale-ins. This will result in changes in service endpoint addresses as the services move across different virtual machines.



Service distribution

The Service Fabric Naming Service is responsible for abstracting this complexity from the consuming service or application. The Naming Service takes care of service discovery and resolution. All service instances in Service Fabric are identified by a unique URL such as `fabric:/MyMicroServiceApp/AppService1`. This name stays constant across the lifetime of the service, although the endpoint addresses which physically host the service may change. Internally, Service Fabric manages a map between the service names and the physical location where the service is hosted. This is similar to the DNS service which is used to resolve website URLs to IP addresses.

The following figure illustrates the name resolution process for a service hosted on Service Fabric:


http://10.0.0.1:8081/api/
fabric:/App1/Service 1
1) My address is "http://10.0.0.1:8081/api/"
Naming Service
3) connect to http://10.0.0.1:8081/api/
fabric:/App1/Service 2
2) Give me the address for "fabric:/App1,/service1"

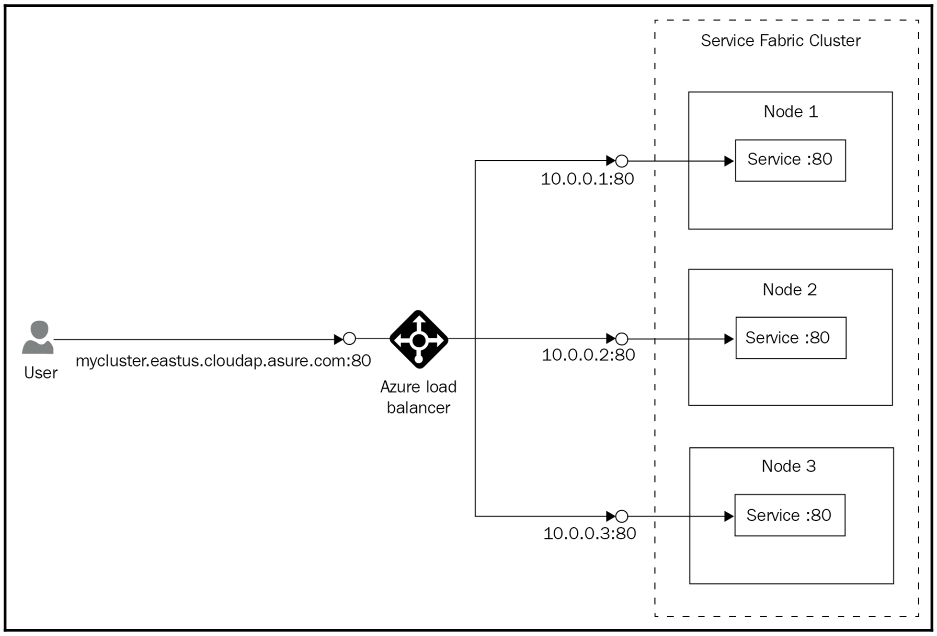
Name resolution

Connections from applications external to Service Fabric

Service communications between services hosted in Service Fabric can be categorized as internal or external. Internal communication between services hosted on Service Fabric is easily achieved using the Naming Service. External communication, originated from an application or a user outside the boundaries of Service Fabric, will need some extra work. To understand how this works, let's dive deeper into the logical network layout of a typical Service Fabric cluster.

A Service Fabric cluster is usually placed behind an Azure load balancer. The load balancer acts like a gateway to all traffic which needs to pass to the Service Fabric cluster. The load balancer is aware of every post open on every node of a cluster. When a request hits the load balancer, it identifies the port the request is looking for and randomly routes the request to one of the nodes which has the requested port open. The load balancer is not aware of the services running on the nodes or the ports associated with the services.

The following figure illustrates request routing in action:



Request routing

Configuring ports and protocols

The protocol and the ports to be opened by a Service Fabric cluster can be easily configured through the portal. Let's take an example to understand the configuration in detail.

If we need a web application to be hosted on a Service Fabric cluster which should have port `80` opened on HTTP to accept incoming traffic, the following steps should be performed.

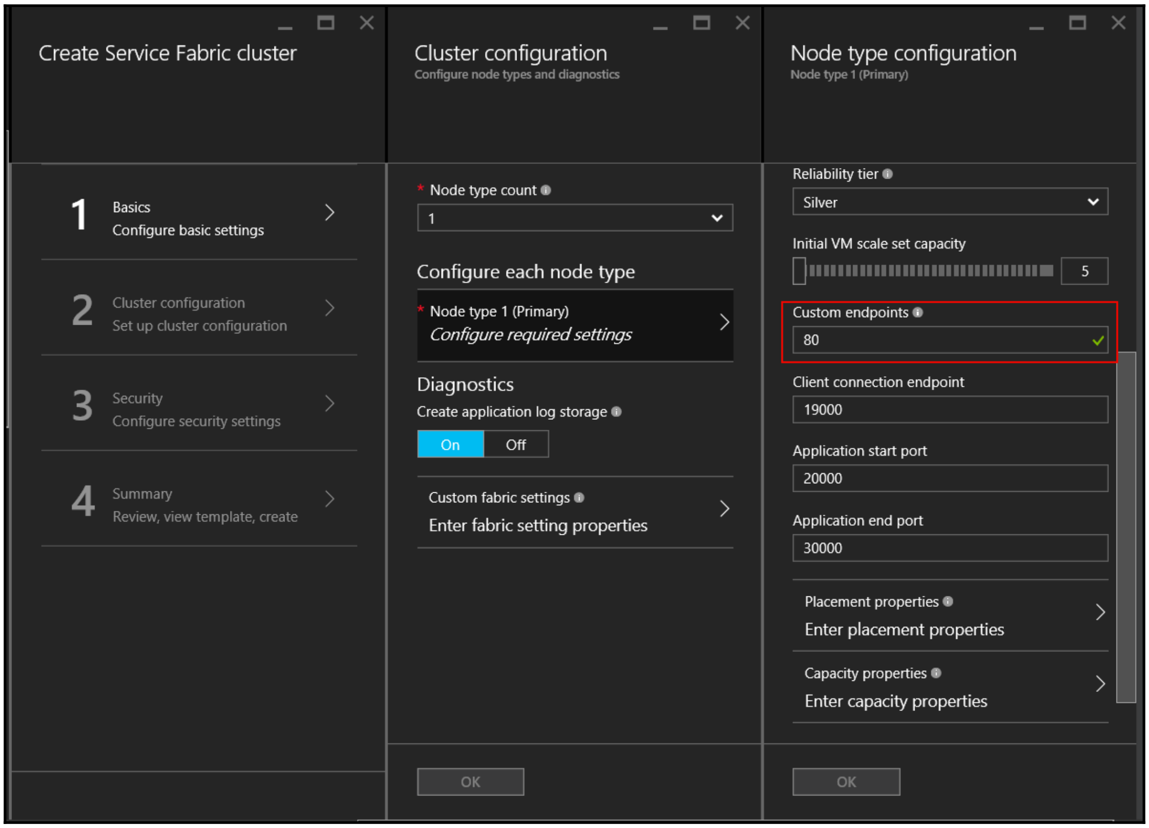
Configuring the service manifest

Once a service listening to port `80` is authored, we need to configure port `80` in the service manifest to open a listener in the service. This can be done by editing `Service Manifest.xml`:

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

Configuring the custom endpoint

On the Service Fabric cluster, configure port `80` as a **Custom endpoint**. This can be easily done through the Azure Management Portal:



Create Service Fabric cluster

1 Basics
Configure basic settings

2 Cluster configuration
Set up cluster configuration

3 Security
Configure security settings

4 Summary
Review, view template, create

Cluster configuration
Configure node types and diagnostics

* Node type count: 1

Configure each node type

* Node type 1 (Primary)
Configure required settings

Diagnostics
Create application log storage: [✓ On] Off

Custom fabric settings
Enter fabric setting properties

OK

Node type configuration
Node type 1 (Primary)

Reliability tier: Silver

Initial VM scale set capacity: 5

Custom endpoints: 80

Client connection endpoint: 19000

Application start port: 20000

Application end port: 30000

Placement properties
Enter placement properties

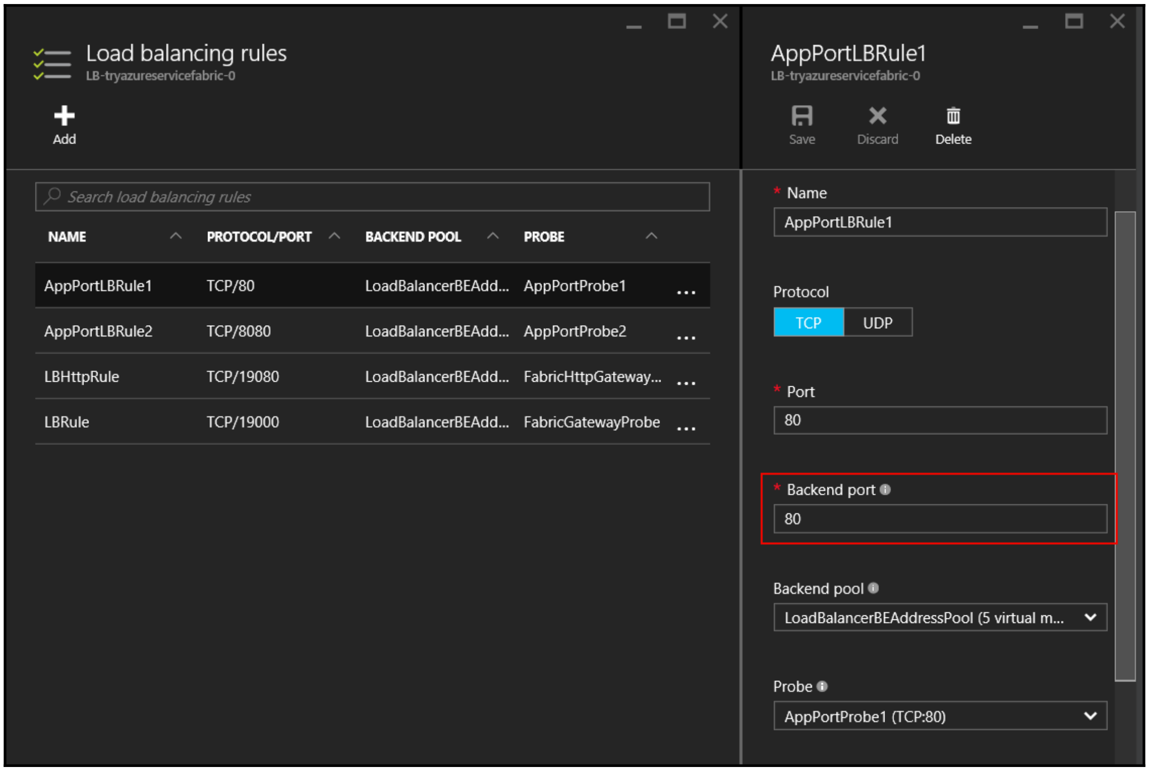
Capacity properties
Enter capacity properties

OK

Configuring custom port

Configuring the Azure load balancer

Once the cluster is configured and created, the Azure load balancer can be instructed to forward the traffic to port `80`. If the Service Fabric cluster is created through the portal, this step is automatically taken care for every port which is configured on the cluster configuration:


Load balancing rules
LB-tryazureservicefabric-0
Add
Search load balancing rules
AppPortLBRule1
LB-tryazureservicefabric-0
Save
Discard
Delete

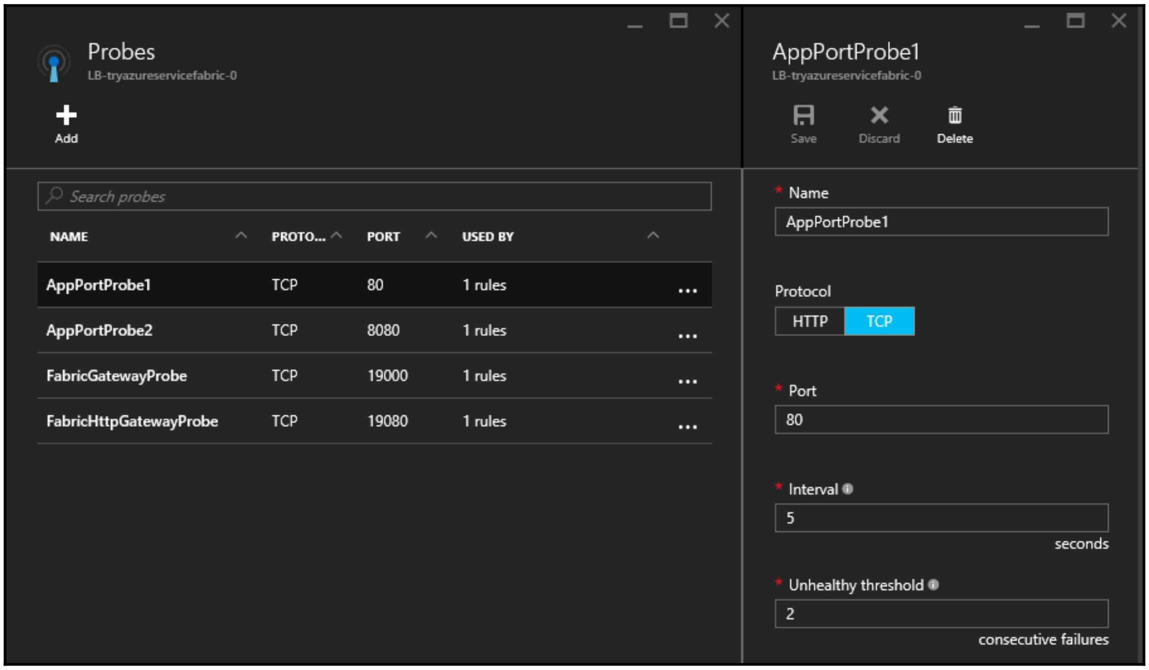
NAME	PROTOCOL/PORT	BACKEND POOL	PROBE
AppPortLBRule1	TCP/80	LoadBalancerBEAdd...	AppPortProbe1
AppPortLBRule2	TCP/8080	LoadBalancerBEAdd...	AppPortProbe2
LBHttpRule	TCP/19080	LoadBalancerBEAdd...	FabricHttpGateway...
LBRule	TCP/19000	LoadBalancerBEAdd...	FabricGatewayProbe

* Name: AppPortLBRule1
Protocol: [✓ TCP] UDP
* Port: 80
* Backend port: 80
Backend pool: LoadBalancerBEAddressPool (5 virtual m...
Probe: AppPortProbe1 (TCP:80)

Configuring Azure Load Balancer

Configuring the health check

Azure load balancer probes the ports on the nodes for their availability to ensure reliability of the service. The probes can be configured on the Azure Portal. This is an optional step as a default probe configuration is applied for each endpoint when a cluster is created:



NAME	PROTO...	PORT	USED BY
AppPortProbe1	TCP	80	1 rules
AppPortProbe2	TCP	8080	1 rules
FabricGatewayProbe	TCP	19000	1 rules
FabricHttpGatewayProbe	TCP	19080	1 rules

* Name: AppPortProbe1

Protocol: HTTP [✓ TCP]

* Port: 80

* Interval: 5 seconds

* Unhealthy threshold: 2 consecutive failures

Configuring the probe

Built-in communication API

Service Fabric offers many built-in communication options to support inter-service communications. Service remoting is one of them. This option allows strong typed remote procedure calls between Reliable Services and Reliable Actors. This option is very easy to set up and operate with as service remoting handles resolution of service addresses, connection, retry, and error handling. Service Fabric also supports HTTP for language-agnostic communication.

Service Fabric SDK exposes `ICommunicationClient` and `ServicePartitionClient` classes for service resolution, HTTP connections, and retry loops. WCF is also supported by Service Fabric as a communication channel to enable legacy workload to be hosted on it. The SDK exposed `WcfCommunicationListener` for the server side and `WcfCommunicationClient` and `ServicePartitionClient` classes for the client to ease programming hurdles.

Deploying a Guest Executable

Service Fabric supports hosting packaged executables developed using .NET, Node.js, Java, or any similar programming languages. These executables are addressed as Guest Executables in the Service Fabric world. Although they are not developed using Service Fabric SDK, Service Fabric still ensures high availability and high-density deployment of Guest Executables on a cluster. Service Fabric is also capable of performing basic health monitoring for these executables and managing their application lifecycle.

Let's explore the details around the steps to be followed to deploy a Guest Executable on a Service Fabric cluster.

Understanding the manifests

Service Fabric uses two XML files - the *Application Manifest* and the *Service Manifest* for the purpose of packaging and deploying the applications. An application in Service Fabric is a unit of deployment. An application can be deployed, upgraded, or even rolled back as a unit. The rollback usually occurs in case of a failure on upgrade and is automatically handled by the Service Fabric orchestrator to ensure system stability.

Application manifest, as the name suggests, is used to describe the application. It defines the list of services and the parameters required to deploy these services. Number of service instances being one of them.

Service manifest defines the components of a service. This includes data, code, and configuration. We will look deeper in these manifests in later sections of this book.

Package structure

Service Fabric expects the application to be packages in a specific directory structure. Let's take an example to understand this further:

```
|-- ApplicationPackageRoot
    |-- GuestService1Pkg
        |-- Code
            |-- existingapp.exe
        |-- Config
            |-- Settings.xml
        |-- Data

        |-- ServiceManifest.xml
    |-- ApplicationManifest.xml
```

`ApplicationPackageRoot`, is the root folder containing the application manifest file - `ApplicationManifest.xml`. Each sub directory under this folder specifies a service which is part of this application. In the preceding example, `GuestService1Pkg` is one such service. Every service folder in turn holds the service components. This includes three sub folders holding the code, configuration, and data. The `Service` folder also holds the service manifest file - `ServiceManifest.xml`. The `Code` directory contains the source code for the service. The `Config` directory contains a settings file - `Settings.xml` which store the service configuration. The `Data` folder stores transient data which is used by the service locally. This data is not replicated across service instances. The `Config` and `Code` folders are optional.

There are two ways to package a Service Fabric Guest Executable. The easiest way is to use Visual Studio template for Guest Executables. The other way is to package it manually. Let's take a look at both these options.

Packaging Guest Executables using Visual Studio

Installing the Service Fabric SDK is a pre-requisite for performing this exercise. The following steps can be followed to create a Guest Executable package using Visual Studio:

1. Open Visual Studio and choose **New Project**.
2. Select **Service Fabric Application**.
3. Choose the **Guest Executable** service template.
4. Click **Browse** and select the folder holding the executable to be packaged.

5. Fill in other parameters:
    - **Code Package Behavior**: This is ideally set to copy all the content of the folder to the Visual Studio project. There is also an option to link a folder if you require the project to dynamically pick up the executables every time it executes.
    - **Program**: Choose the executable that should run to start the service.
    - **Arguments**: Input parameters to be passed in as arguments to the executable should be specified here.
    - **Working Folder**: The working folder can be set to one of these three values:
        - **CodeBase**: If you want to set the working folder to the code directory.
        - **CodePackage**: If you want to set the working folder to the root directory.
        - **Work**: If you want to place the files in a sub directory called **Work**
6. Name your service.
7. Click **OK**.
8. If the service requires an endpoint communication, you need to edit `ServiceManifest.xml` to open an endpoint. We will see how this can be done in the next section.
9. You can now debug the solution and use package and publish action to deploy the Guest Executable on your local cluster.
10. When ready, you can publish the application to a remote cluster using Visual Studio.

Manually packaging a Guest Executable

The process of manually packaging a Guest Executable can be divided into four steps.

Creating the directory structure

To start with, the directory structure for the deployment package needs to be created as per the hierarchy mentioned earlier in this chapter. This package structure will host the code, configuration, and data required for the deployment.

Adding code and configuration

Once the directory structure is ready, the code and the configuration files can be placed under the respective directories. It is allowed to have multiple levels of subdirectories within these folders. It is important to ensure that all the dependencies required for the application to run are included in the folders. Service Fabric replicates the content of these folders across nodes in a cluster.

Updating service manifest

Service manifest file stores the configuration for service deployment. This file needs to be updated to provide a name to the service, to specify the command used to launch the application, and to specify and setup or configure scripts which need to be executed. Following is an example of a service manifest 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>
```

To break this down further, let's go through every section in the service manifest:

```
<ServiceTypes>
  <StatelessServiceType ServiceTypeName="NodeApp" UseImplicitHost="true" />
</ServiceTypes>
```

`ServiceTypeName` is a custom parameter which can be assigned to the deployed service. This value will be later used in the application manifest file. It is important to set the `UseImplicitHost` as true as this specifies the service as self-contained:

```
<CodePackage Name="Code" Version="1.0.0.0">
```

`CodePackage` element specifies the location of the folder holding the code and the version of the application packaged:

```
<SetupEntryPoint>
   <ExeHost>
       <Program>scripts\launchConfig.cmd</Program>
   </ExeHost>
</SetupEntryPoint>
```

`SetupEntryPoint` is an optional element used to specify the executable of batch files which should be executed before the service code is launched. This can be ignored as there are no startup scripts for the application. There can only be one `SetupEntryPoint` for a package:

```
<EntryPoint>
  <ExeHost>
    <Program>node.exe</Program>
    <Arguments>bin/www</Arguments>
    <WorkingFolder>CodeBase</WorkingFolder>
    <ConsoleRedirection FileRetentionCount="5" FileMaxSizeInKb="2048"/>
  </ExeHost>
</EntryPoint>
```

The `EntryPoint` element specifies the details about application launch. The `Program` element specifies the name of the executable to be launched, `Arguments` element specifies the parameters to be passed in as arguments to the executable and the `WorkingFolder` specifies the working directory. `ConsoleRedirection` can be used to setup logging. This element helps redirect console output to a working directory:

```
<Endpoints>
   <Endpoint Name="NodeAppTypeEndpoint" Protocol="http" Port="3000"
Type="Input" />
</Endpoints>
```

The `Endpoints` element specifies the endpoint this application can listen on.

Updating the application manifest

Application manifest defines the list of services and the parameters required to deploy these services. Following is an example:

```
<?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>
```

The `ServiceManifestImport` element specifies the services that are included in the app.

Deployment

A PowerShell script can be used to manually deploy a package to a Service Fabric cluster:

```
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
```

The `InstanceCount` parameter in the preceding script specifies the number of instances of the application to be deployed in the cluster. This value can be set to `-1` if the application needs to be deployed on every node of the Service Fabric cluster.

Deploying a Guest Container

Service Fabric supports hosting containerized Microservices. It offers specialized features to manage containerized workloads. Some of these features are container image deployment and activation, resource governance, repository authentication, container port to host mapping, container-to-container discovery and communication, support for environment variables, and so on. Service Fabric supports two types of container workloads - Windows Containers and Docker Containers. Let's pick them one by one and understand the packaging and deployment process.

Deploying Windows Container

Similar to Guest Executables, Service Fabric will support packaging Guest Containers either through Visual Studio or manually. However, the Visual Studio wizard for Guest Containers is still under development. Let's dive deeper into the process of packaging and deploying a Guest Container manually.

The packaging process consists of four major steps - publishing the container to a repository, creating the package directory structure, updating the service manifest, and updating the application manifest.

Container image deployment and activation

Service Fabric treats a container as an application host capable of hosting multiple service replicas. To deploy and activate a container, the name of the container image must be put into the `ContainerHost` element of the service manifest. The following example deploys a container called `myimage:v1` from a repository `myrepo`:

```
<CodePackage Name="Code" Version="1.0">
    <EntryPoint>
      <ContainerHost>
        <ImageName>myrepo/myimagename:v1</ImageName>
        <Commands></Commands>
      </ContainerHost>
    </EntryPoint>
</CodePackage>
```

The `Commands` element can be used to pass commands to the container image. The element can take comma-separated values as commands.

Resource governance

The `ResourceGovernancePolicy` element is used to restrict the resources consumed by a container on a host. The limits can be set for `Memory`, `MemorySwap`, `CPUShares`, `MemoryReservationInMB`, or `BlkioWeight` (bulk I/O weight). The following sample shows the use of `ResourceGovernancePolicy` element:

```
<ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="FrontendServicePackage"
ServiceManifestVersion="1.0"/>
    <Policies>
        <ContainerHostPolicies CodePackageRef="FrontendService.Code">
            <RepositoryCredentials AccountName="TestUser" Password="12345"
PasswordEncrypted="false"/>
        </ContainerHostPolicies>
    </Policies>
</ServiceManifestImport>
```

The password specified in the `RepositoryCredentials` element should be encrypted using a certificate deployed on the machine.

Container port to host port mapping

`PortBinding` element in the application manifest can be used to configure the ports used for communication by the container. The binding maps the ports internal to the container to the ones open on the hosting machine. Following is an example:

```
<ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="FrontendServicePackage"
ServiceManifestVersion="1.0"/>
    <Policies>
        <ContainerHostPolicies CodePackageRef="FrontendService.Code">
            <PortBinding ContainerPort="8905"/>
        </ContainerHostPolicies>
    </Policies>
</ServiceManifestImport>
```

Container-to-container discovery and communication

The PortBinding element is also used to map a port to an endpoint. The Endpoint element in a PortBinding can be used to specify a fixed port or left blank to choose a random port available to the cluster port range. The Endpoint element in the ServiceManifest enables Service Fabric to automatically publish the element to the Naming Service to enable discovery of the container. Following is an example:

```
<ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="FrontendServicePackage"
ServiceManifestVersion="1.0"/>
    <Policies>
        <ContainerHostPolicies CodePackageRef="FrontendService.Code">
            <PortBinding ContainerPort="8905" EndpointRef="Endpoint1"/>
        </ContainerHostPolicies>
    </Policies>
</ServiceManifestImport>
```

Configuring and setting environment variables

Environment variables for the service can be set within the service manifest. They can also be overridden in the application manifest or by application parameters supplied at runtime. Following is an example explaining how this can be done in a service manifest:

```
<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>
```

The following example shows how this can be overridden in an application manifest:

```
<ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="FrontendServicePackage"
ServiceManifestVersion="1.0"/>
    <EnvironmentOverrides CodePackageRef="FrontendService.Code">
        <EnvironmentVariable Name="BackendServiceName"
Value="[BackendSvc]"/>
        <EnvironmentVariable Name="HttpGatewayPort" Value="19080"/>
    </EnvironmentOverrides>
</ServiceManifestImport>
```

Deploying a Linux container

Containers in Linux can be packaged for Service Fabric either using a Yeoman template or manually. The Service Fabric SDK for Linux comes with a built-in Yeoman generator to ease things for application developers. Yeoman generator facilitates creating an application and adding it to a container image with minimal effort.

Yeoman is a development stack that combines several development tools and runs as a command-line interface. Yeoman has a pluggable architecture and it only orchestrates the development tools plugged into it. The many available Yeoman plugins, called generators in Yeoman terms, create starter templates, manage dependencies, run unit tests, and optimize deployment code, among many other things. You can read more about Yeoman here: `http://yeoman.io/`.

Manifest files can be later updated to add or update services. Once the application is built, the Azure CLI can be used to deploy the local cluster.

Summary

We started this chapter by discussing service discovery and communication protocols used by Service Fabric. Next, we dived deeper into configuring the manifest for custom endpoints, load balancer, and health checks.

Towards the end of the chapter, we briefly discussed methods for deploying Guest executables and Guest Containers on Service Fabric.

In the next chapter, we will discuss another programming model of Service Fabric, known as Reliable Services.

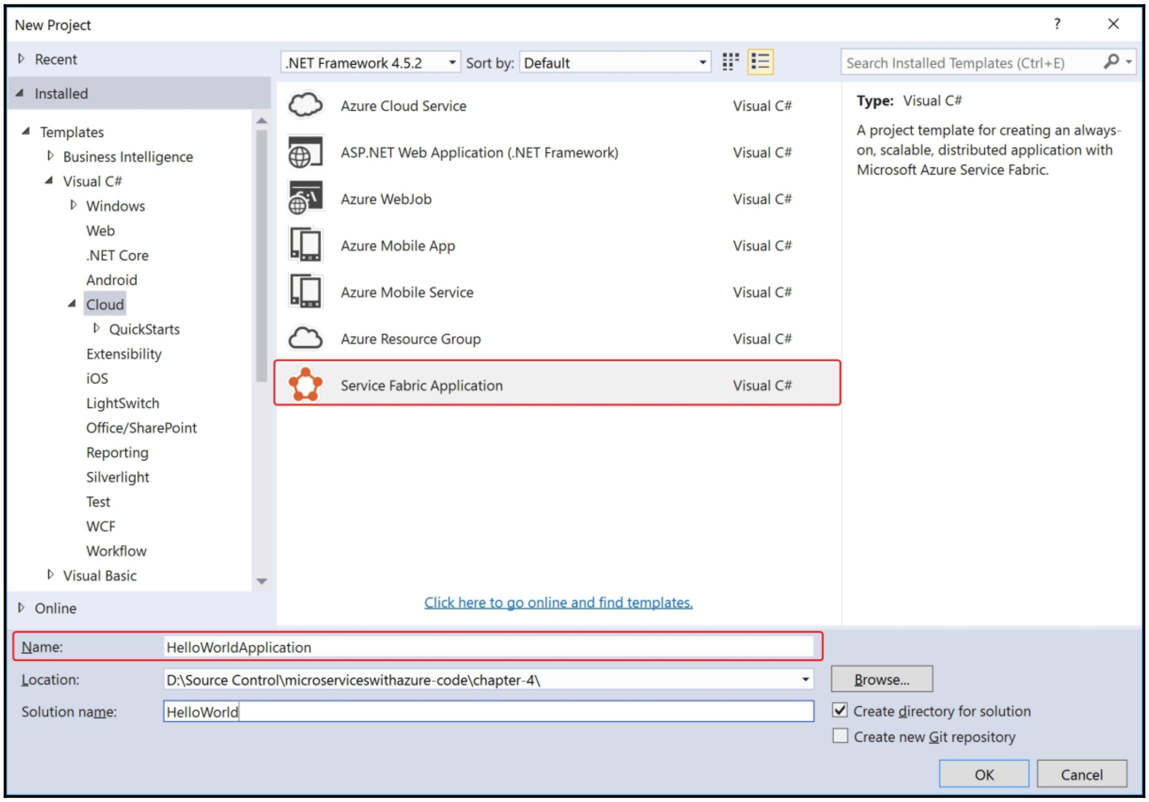
5
Hands on with Service Fabric – Reliable Services

By now, we have set a strong foundation for us to explore the internals of Service Fabric. Let's get started by building our first Service Fabric application. We will build a simple application that will print the customary *Hello World* message.

The companion GitHub repository of this title contains code for all the samples that we have used in this book. You can visit `https://github.com/PacktPublishing/Microservices-with-Azure`to download the samples. Let's start building our application:

1. Launch Visual Studio as an administrator. This is necessary because we are going to test our application in Service Fabric local cluster that needs administrator privileges to work.
2. Click **File**| **New Project** | **Cloud** | **Service Fabric Application**.

3. Name the application `HelloWorldApplication` and click **OK**.


New Project
Recent
Installed
Templates
Business Intelligence
Visual C#
Windows
Web
.NET Core
Android
Cloud
QuickStarts
Extensibility
iOS
LightSwitch
Office/SharePoint
Reporting
Silverlight
Test
WCF
Workflow
Visual Basic
Online
.NET Framework 4.5.2
Sort by: Default
Search Installed Templates (Ctrl+E)
Azure Cloud Service	Visual C#
ASP.NET Web Application (.NET Framework)	Visual C#
Azure WebJob	Visual C#
Azure Mobile App	Visual C#
Azure Mobile Service	Visual C#
Azure Resource Group	Visual C#
Service Fabric Application	Visual C#
Click here to go online and find templates.
Browse...
OK
Cancel

Type: Visual C#

A project template for creating an always-on, scalable, distributed application with Microsoft Azure Service Fabric.

Name: HelloWorldApplication

Location: D:\Source Control\microserviceswithazure-code\chapter-4\

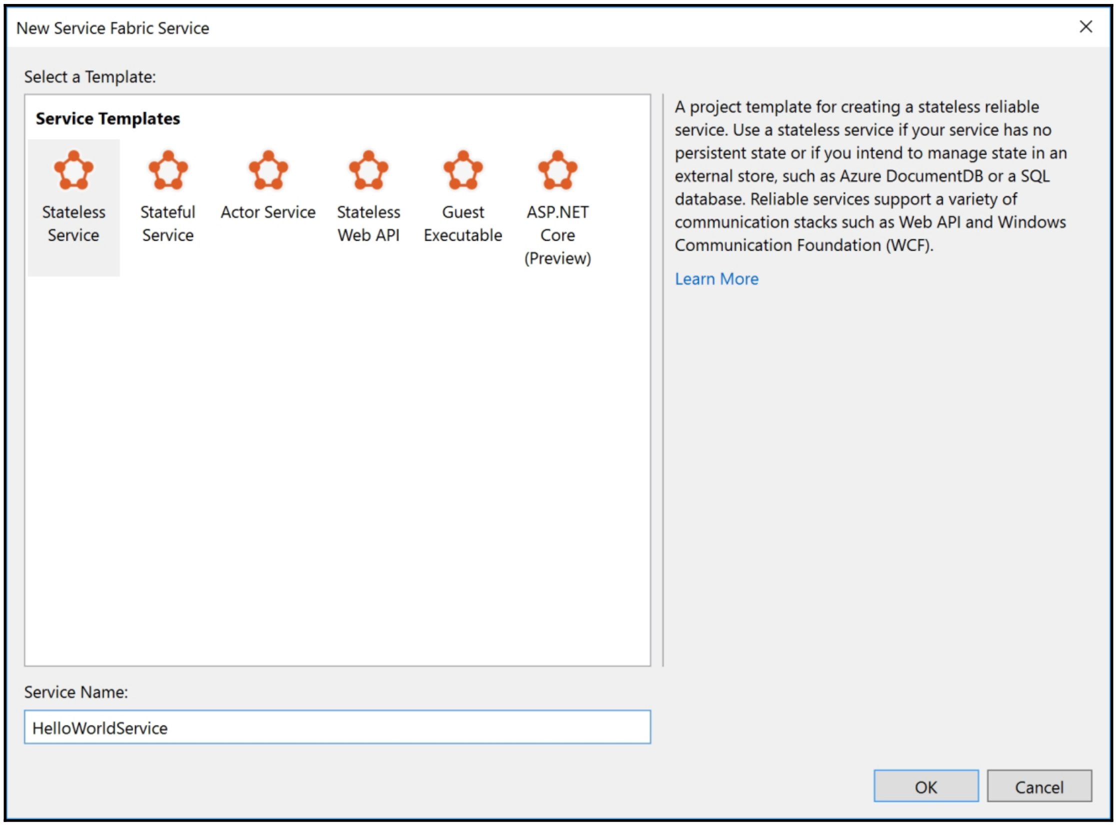
Solution name: HelloWorld

[✓] Create directory for solution

[ ] Create new Git repository

Create new Service Fabric application

4. On the next page, choose **Stateless Service** as the service type to include in your application. Name it `HelloWorldService` and click **OK**.


New Service Fabric Service

Select a Template:

Service Templates

A project template for creating a stateless reliable service. Use a stateless service if your service has no persistent state or if you intend to manage state in an external store, such as Azure DocumentDB or a SQL database. Reliable services support a variety of communication stacks such as Web API and Windows Communication Foundation (WCF).

Learn More

Service Name: HelloWorldService

Create new stateless service dialog

Wait for the project template to unfold. In your solution, you will find two projects – the Service Fabric application project, named `HelloWorldApplication` and the stateless Microservice named `HelloWorldService`. Remember that in Service Fabric an application is a collection of Microservices. We will go through the various components of the project in a little while. For now, let us make our `HelloWorldService` generate some output. Navigate to the `HelloWorldService.cs` file and locate the `HelloWorldService` class.

```
namespace HelloWorldService
{
    /// <summary>
    /// An instance of this class is created for each service
        instance by the Service Fabric runtime.
    /// </summary>
```

```
            internal sealed class HelloWorldService : StatelessService
            {
                public HelloWorldService(StatelessServiceContext
                                                            context)
                    : base(context)
                { }

                /// <summary>
                /// Optional override to create listeners (for example,
                    TCP, HTTP) for this service replica to handle client
                     or user requests.
               /// </summary>
                /// <returns>A collection of listeners.</returns>
                protected override IEnumerable<ServiceInstanceListener>
                  CreateServiceInstanceListeners()
                {
                    return new ServiceInstanceListener[0];
                }

                /// <summary>
                /// This is the main entry point for your
                    service instance.
                /// </summary>
                /// <param name="cancellationToken">Canceled when
                                                                 Service
                      Fabric needs to shut down this service
                      instance.</param>
                protected override async Task RunAsync(CancellationToken
                   cancellationToken)
                {
                    ...
                }
            }
        }
```

A Service Fabric stateless Microservice derives from the `StatelessService` class. Service Fabric internally uses the functions exposed by the `StatelessService` class to govern the lifecycle of your stateless Microservice. The `RunAsync` method is a general-purpose entry point for your service code.

Let's replace the definition of this method with the following one:

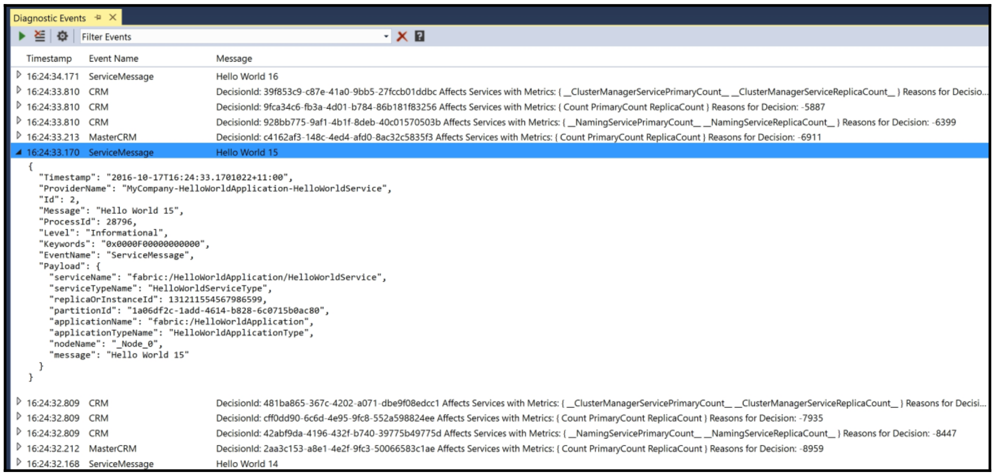
```
protected override async Task RunAsync(CancellationToken
  cancellationToken)
{
    long iterations = 0;
    while (true)
    {
```

```
                cancellationToken.ThrowIfCancellationRequested();
               ServiceEventSource.Current.ServiceMessage(this.Context,
                 $"Hello World {++iterations}");
                await Task.Delay(TimeSpan.FromSeconds(1),
                  cancellationToken);
            }
        }
```

Service Fabric does not force your Microservices to communicate through any specific communication stack. Since we haven't implemented any communication stack for our service in this example, our service will behave as a background service akin to Windows Service, Azure WebJob, or Cloud Service Worker Role.

The Service Fabric solution template also adds an Event Source implementation to create events for **Event Tracing for Windows** (**ETW**) for you. ETW is an efficient kernel-level tracing facility that lets you log kernel or application-defined events. Service Fabric runtime itself logs events using ETW and using the same logging mechanism would help you understand how your service communicates with the Service Fabric runtime. ETW can work across cloud environments and even on your local system. Therefore, using ETW can make your application platform independent.

5. Run the application by clicking on Run or by pressing F5. At this time, visual studio will spin up a local cluster for development. Once your application gets deployed on the cluster, you would be able to see the ETW trace events in **Diagnostic Events** Viewer window of Visual Studio:



```
Timestamp     Event Name      Message
16:24:34.171  ServiceMessage  Hello World 16
16:24:33.810  CRM             DecisionId: 39f853c9-c87e-41a0-9bb5-27fccb01ddbc Affects Services with Metrics: { __ClusterManagerServicePrimaryCount__ __ClusterManagerServiceReplicaCount__ } Reasons for Decisio...
16:24:33.810  CRM             DecisionId: 9fca34c6-fb3a-4d01-b784-86b181f83256 Affects Services with Metrics: { Count PrimaryCount ReplicaCount } Reasons for Decision: -5887
16:24:33.810  CRM             DecisionId: 928bb775-9af1-4b1f-8deb-40c01570503b Affects Services with Metrics: { __NamingServicePrimaryCount__ __NamingServiceReplicaCount__ } Reasons for Decision: -6399
16:24:33.213  MasterCRM       DecisionId: c4162af3-148c-4ed4-afd0-8ac32c5835f3 Affects Services with Metrics: { Count PrimaryCount ReplicaCount } Reasons for Decision: -6911
16:24:33.170  ServiceMessage  Hello World 15
  {
    "Timestamp": "2016-10-17T16:24:33.1701022+11:00",
    "ProviderName": "MyCompany-HelloWorldApplication-HelloWorldService",
    "Id": 2,
    "Message": "Hello World 15",
    "ProcessId": 28796,
    "Level": "Informational",
    "Keywords": "0x0000F00000000000",
    "EventName": "ServiceMessage",
    "Payload": {
      "serviceName": "fabric:/HelloWorldApplication/HelloWorldService",
      "serviceTypeName": "HelloWorldServiceType",
      "replicaOrInstanceId": 131211554567986599,
      "partitionId": "1a06df2c-1add-4614-b828-6c0715b0ac80",
      "applicationName": "fabric:/HelloWorldApplication",
      "applicationTypeName": "HelloWorldApplicationType",
      "nodeName": "_Node_0",
      "message": "Hello World 15"
    }
  }

16:24:32.809  CRM             DecisionId: 481ba865-367c-4202-a071-dbe9f08edcc1 Affects Services with Metrics: { __ClusterManagerServicePrimaryCount__ __ClusterManagerServiceReplicaCount__ } Reasons for Decisi...
16:24:32.809  CRM             DecisionId: cff0dd90-6c6d-4e95-9fc8-552a598824ee Affects Services with Metrics: { Count PrimaryCount ReplicaCount } Reasons for Decision: -7935
16:24:32.809  CRM             DecisionId: 42abf9da-4196-432f-b740-39775b49775d Affects Services with Metrics: { __NamingServicePrimaryCount__ __NamingServiceReplicaCount__ } Reasons for Decision: -8447
16:24:32.212  MasterCRM       DecisionId: 2aa3c153-a8e1-4e2f-9fc3-50066583c1ae Affects Services with Metrics: { Count PrimaryCount ReplicaCount } Reasons for Decision: -8959
16:24:32.168  ServiceMessage  Hello World 14
```

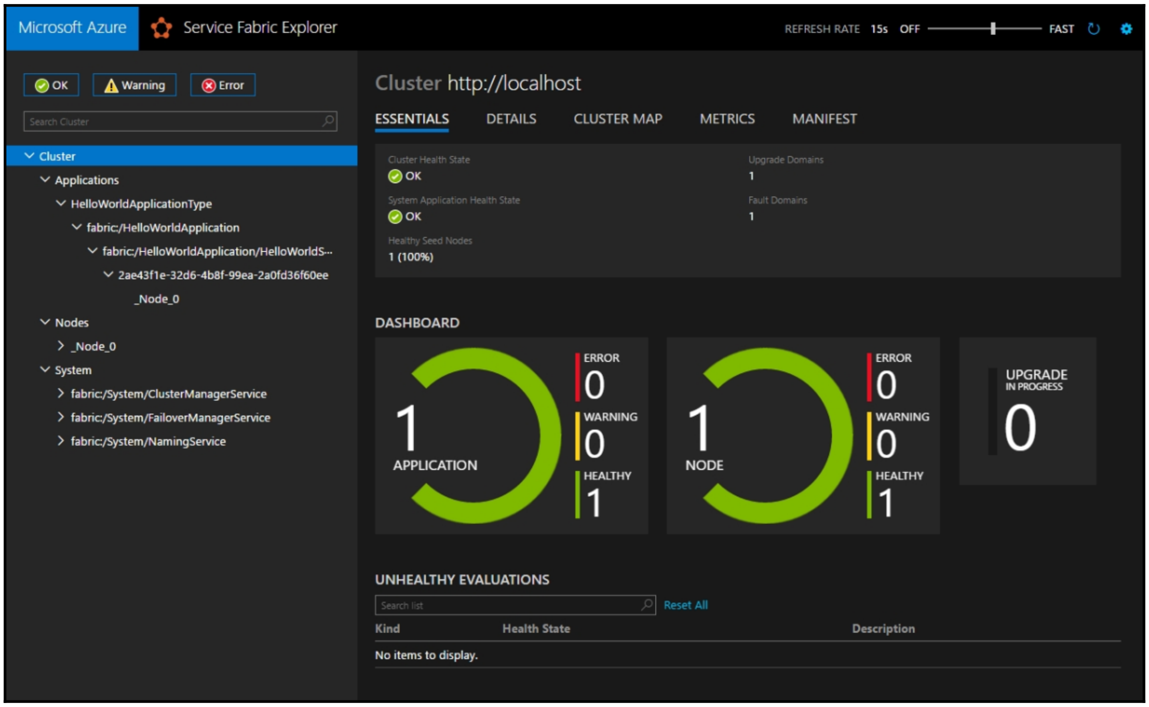
Output of Hello World stateless service

6. You can expand any event to see more details. For instance, in the preceding event, you can view the node and partition on which your Microservice is running.

Exploring the Service Fabric Explorer

Service Fabric SDK installs a *Service Fabric Application management* tool in your system named Service Fabric Explorer (or SFX as Microsoft calls it). You can view the explorer UI by navigating to `http://localhost:19080/Explorer/`.

It is a two-panel web application in which the left panel displays an overview of your cluster in a tree format and the right panel displays detailed information about the currently selected item. Let's take a quick overview of the tree menu on the left:



Service Fabric Explorer overview

Application Type

This node represents the type of your application, which in the case of our sample is `HelloWorldApplicationType`. Defining the application type grants the flexibility for an application administrator to tailor 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.

Application instance

Below the application type node is the application instance node. The application type and application instance are analogous to the class and object concepts in object oriented programming. Your application instance is identified by a name, `fabric:/HelloWorldApplication`, in our case. This name is configurable and can be specified in the format of `fabric:/{name}` where `{name}` is any string.

Service type

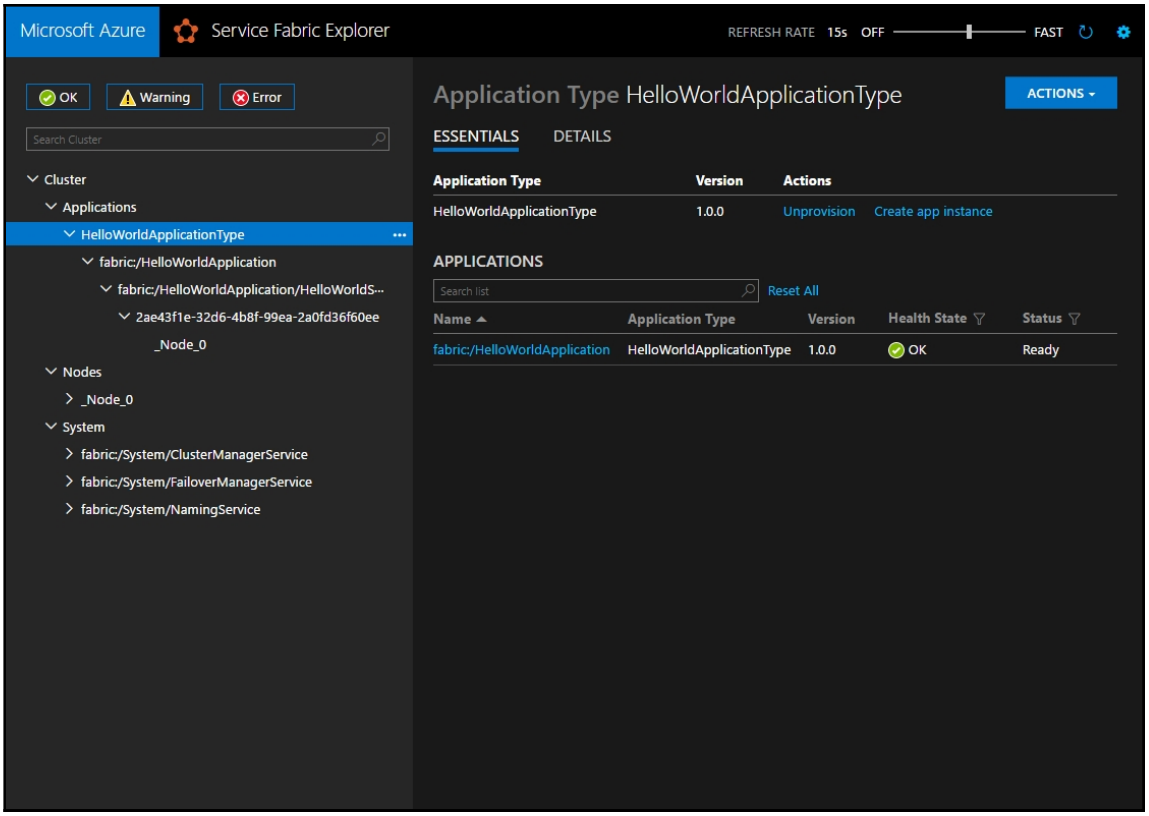
An application type consists of a bundle of service types. Just as the application types are a template for application instances, the service types are templates for your service instances. In Service Fabric Explorer, you will find a service type corresponding to each Microservice that you have in your application. Since in our example, we have added only one service to our application, we can find the `HelloServiceType` representing the registered service type in our application. Each service in a Service Fabric application has a name in the format of `fabric:/{application name}/{service name}`. A Service Fabric cluster named the Naming Service, resolves this name to actual service instance address on every request.

Partition

Right below the Service Type node, you will find a GUID. This GUID represents the partition in which your Microservice instance is deployed. The Naming Service is responsible for routing requests to appropriate partitions and therefore your application should be oblivious to this identifier.

Replica

Each partition of a Microservice can have a configurable number of replicas. These replicas ensure that your service is highly available. The individual replicas that are executing the code of your Microservices are displayed at this level:



```
Cluster
  Applications
    HelloWorldApplicationType
      fabric:/HelloWorldApplication
        fabric:/HelloWorldApplication/HelloWorldS...
          2ae43f1e-32d6-4b8f-99ea-2a0fd36f60ee
            _Node_0
  Nodes
    _Node_0
  System
    fabric:/System/ClusterManagerService
    fabric:/System/FailoverManagerService
    fabric:/System/NamingService
```

Application Type HelloWorldApplicationType

Application Type	Version	Actions
HelloWorldApplicationType	1.0.0	Unprovision Create app instance

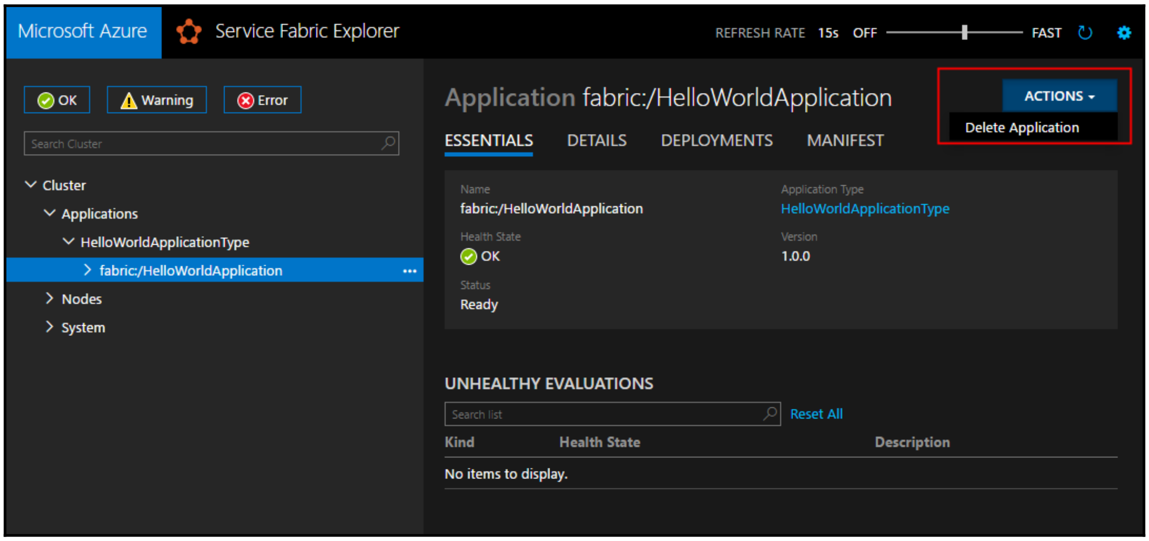
APPLICATIONS

Search list Reset All

Name	Application Type	Version	Health State	Status
fabric:/HelloWorldApplication	HelloWorldApplicationType	1.0.0	✓ OK	Ready

Service Fabric Explorer – application hierarchy

To explore the tool further, click on the application instance node, `fabric:/HelloWorldApplication`, in the left menu and in the **DETAILS** pane, click the **ACTIONS** button and then click the **Delete Application** menu. In the following dialog box, confirm that you want to delete your application instance:



Name
fabric:/HelloWorldApplication

Application Type
HelloWorldApplicationType

Health State
OK

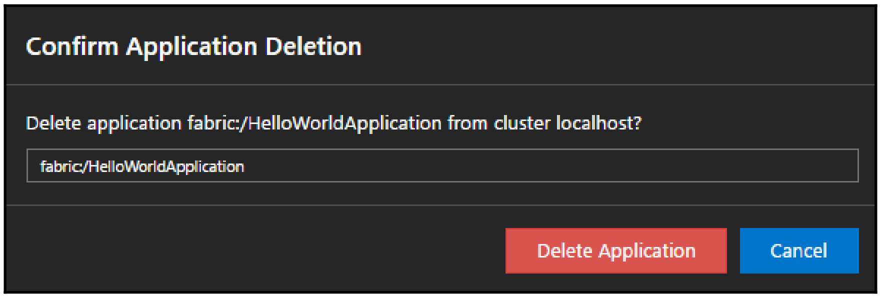
Version
1.0.0

Status
Ready

Kind	Health State	Description
No items to display.		

Service Fabric Explorer – ACTIONS button

Prior to deleting the application, the explorer will prompt you to confirm whether you want to carry out the operation:

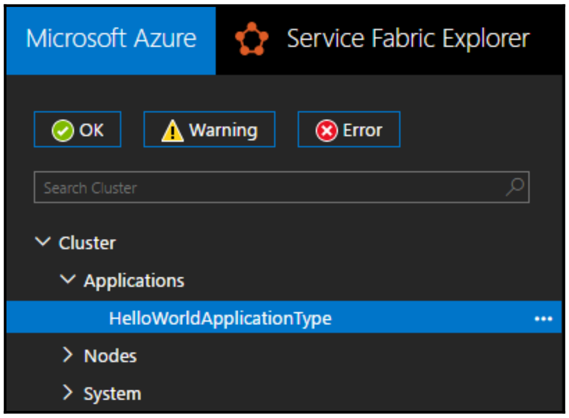


Delete application fabric:/HelloWorldApplication from cluster localhost?

fabric:/HelloWorldApplication

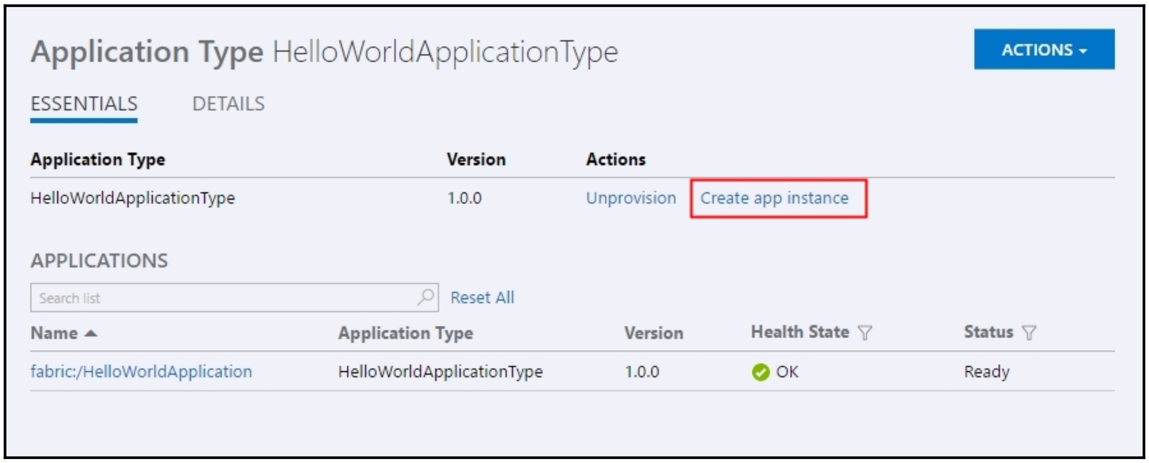
Confirmation to delete application

Once the UI of the application refreshes, your application instance will be removed from the list of available application instances:



Updated list of applications after delete

Note that we have only deleted an application instance, so the application type is still preserved. Click on the **Create app instance** link to provision another instance of the application:



Application Type	Version	Actions
HelloWorldApplicationType	1.0.0	Unprovision Create app instance

Name	Application Type	Version	Health State	Status
fabric:/HelloWorldApplication	HelloWorldApplicationType	1.0.0	OK	Ready

Create application instance

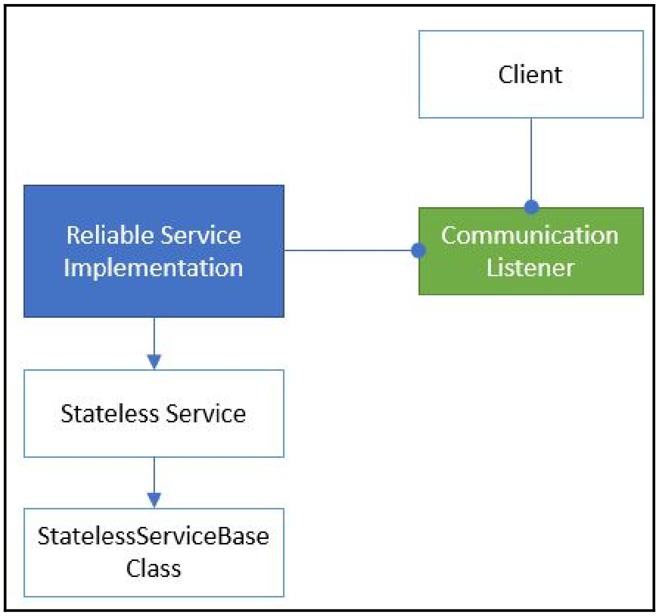
Stateless Reliable Services

A stateless service treats each request as a separate transaction that is independent of any previous request sent to the service. This service can not maintain an internal session store. A stateless service takes all the parameters that it needs to perform an operation at once.

However, you would rarely find applications that are truly stateless. In most of the scenarios, you would find that the state is externalized and stored separately. For instance, a service might use Azure Redis Cache to store state data and not maintain state internally.

Most of the stateless services built on Service Fabric are frontend services that expose the functionality of the underlying system. Users interact with the frontend services, which then forward the call to the correct partition of appropriate stateful services.

Stateless service architecture



Reliable stateless service architecture

A stateless service implementation derives from the `StatelessService` class which manages the lifetime and role of a service.

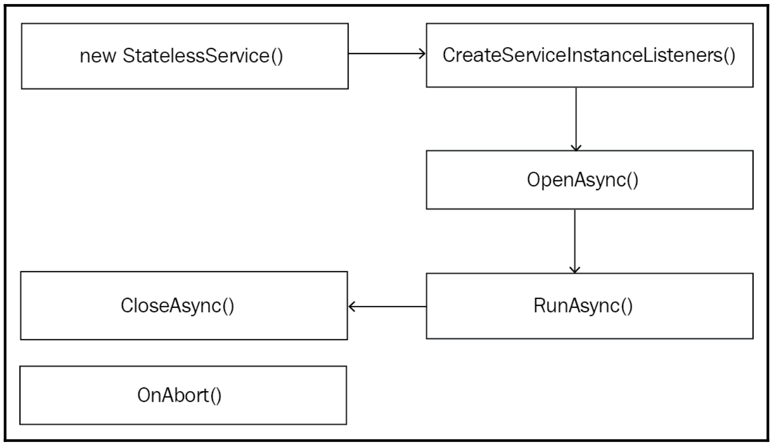
The service implementation may override virtual methods of the base class if the service has work to do at those points in the service lifecycle or if it wants to create a communication listener object. Note that, although the service may implement its own communication listener object exposing `ICommunicationListener`,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

Stateless service lifecycle

Let's take a close look at the lifecycle of a stateless Service Fabric Reliable Service. The lifetime of a stateless Microservice starts as soon as you register a new instance of the service with the Service Fabric runtime. The following are the major lifetime events of a stateless Microservice:

- `CreateServiceInstanceListeners()`: This method is used to setup communication listeners for client requests. Although, Service Fabric provides a default communication listener based on RPC proxy, you can override this method to supply our communication stack of choice. The endpoints returned by the communication listeners are stored as a JSON string of listener name and endpoint string pairs like `{"Endpoints":{"Listener1":"Endpoint1","Listener2":"Endpoint2" ...}}`
- `OnOpenAsync(IStatelessServicePartition, CancellationToken)`: This method is called when the stateless service instance is about to be used. This method should generally be used in conjunction with its counterpart method `OnCloseAsync` to initialize resources that are used by the service such as open connections to external systems, start background processes and setup connections with databases, and so on.
- `RunAsync(CancellationToken)`: This method is a general-purpose entry point for the business logic of your Microservice. This method is invoked when a service instance is ready to begin execution. A cancellation token is provided as input to the method to signal your Microservice that processing should stop. You should only implement this method if your service is continuously processing some data, for example consuming messages from a queue and processing them.

- `OnCloseAsync(CancellationToken)`: In multiple scenarios, your Microservice may be requested to shut down, for example when the code of your service is being upgraded, when the service instance is being moved due to load balancing, or when a transient fault is detected. In such cases, the Service Fabric runtime will trigger the `OnCloseAsync` method to signal your Microservice to start performing the clean-up operations. This method should be used in conjunction with the `OnOpenAsync` method to clean up resources after they have been used by the service.
- `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:



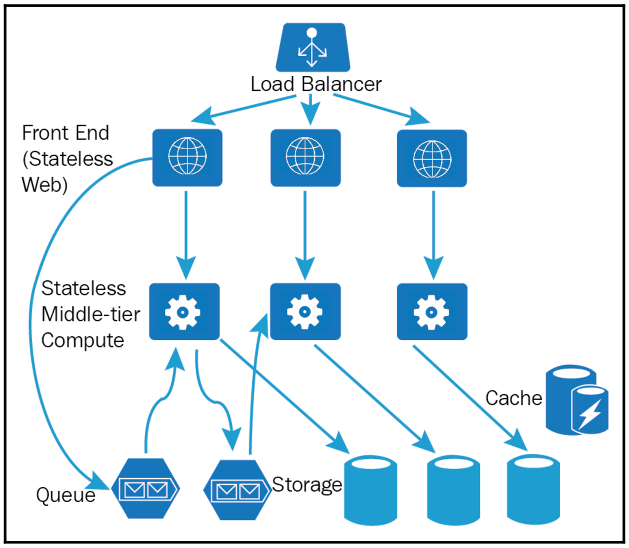
Stateless service lifetime

Scaling stateless services

There are two typical models that are used to build three tier applications using stateless Microservices.

Stateless frontend and stateless middle-tier

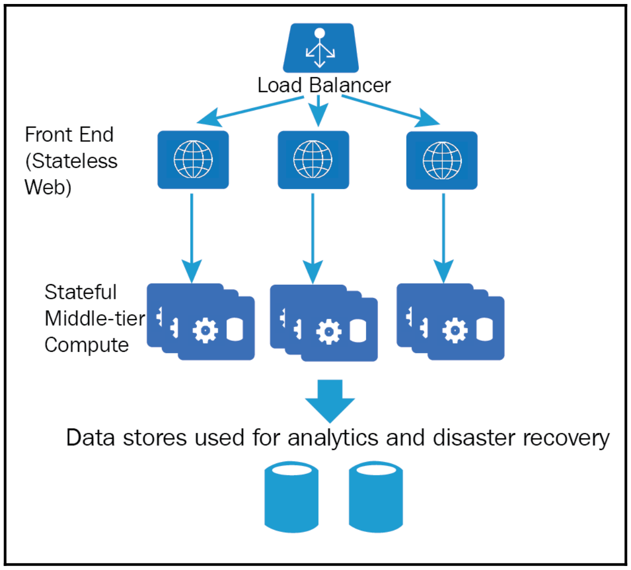
In this model, you build stateless web apps such as ASP.NET and Node.js and deploy your application's frontend as stateless Reliable Services. The **Front End** can communicate with **Stateless Middle-tier Compute** using **Queue** in case asynchronous processing is desired or using HTTP protocol in case synchronous communication is desired. The middle-tier is built using stateless web framework such as Web API or can execute as a continuously executing process. The model can be seen in the following image:



Stateless frontend and stateless middle-tier

Stateless frontend and stateful middle-tier

In this model, you build stateless web apps such as ASP.NET and Node.js and deploy your application's frontend as stateless Reliable Services. The model can be seen in the following image:


Load Balancer
Front End (Stateless Web)
Stateful Middle-tier Compute
Data stores used for analytics and disaster recovery

Stateless front-end and stateful middle-tier

The **Front End** can communicate with **Stateful Middle-tier Compute** using queues, in case asynchronous processing is desired, or using HTTP protocol, in case synchronous communication is desired. This model delivers better performance because the state lives near the compute, which avoids network hops. This model also helps ensure data consistency using transactions to commit data in data stores. In this model, the middle-tier can be built using stateless web framework such as Web API or it can execute as a continuously executing process.

You might have noticed that in both the models we have used a load balancer to route traffic to appropriate stateless service that hosts the frontend of our application. That is because web browsers don't have the ability to interact with the Naming Service to resolve the endpoint of a healthy instance of the service. When you provision your Service Fabric cluster on Azure, you get the ability to add a load balancer to proxy your instances. The load balancer uses a probe to determine the health of the frontend nodes and routes traffic only to the healthy nodes. Addition of a load balancer also makes the process of scaling of frontend invisible to the clients of your application. The load balancer automatically distributes the incoming traffic to the healthy nodes, thus ensuring that your resources get utilized optimally.

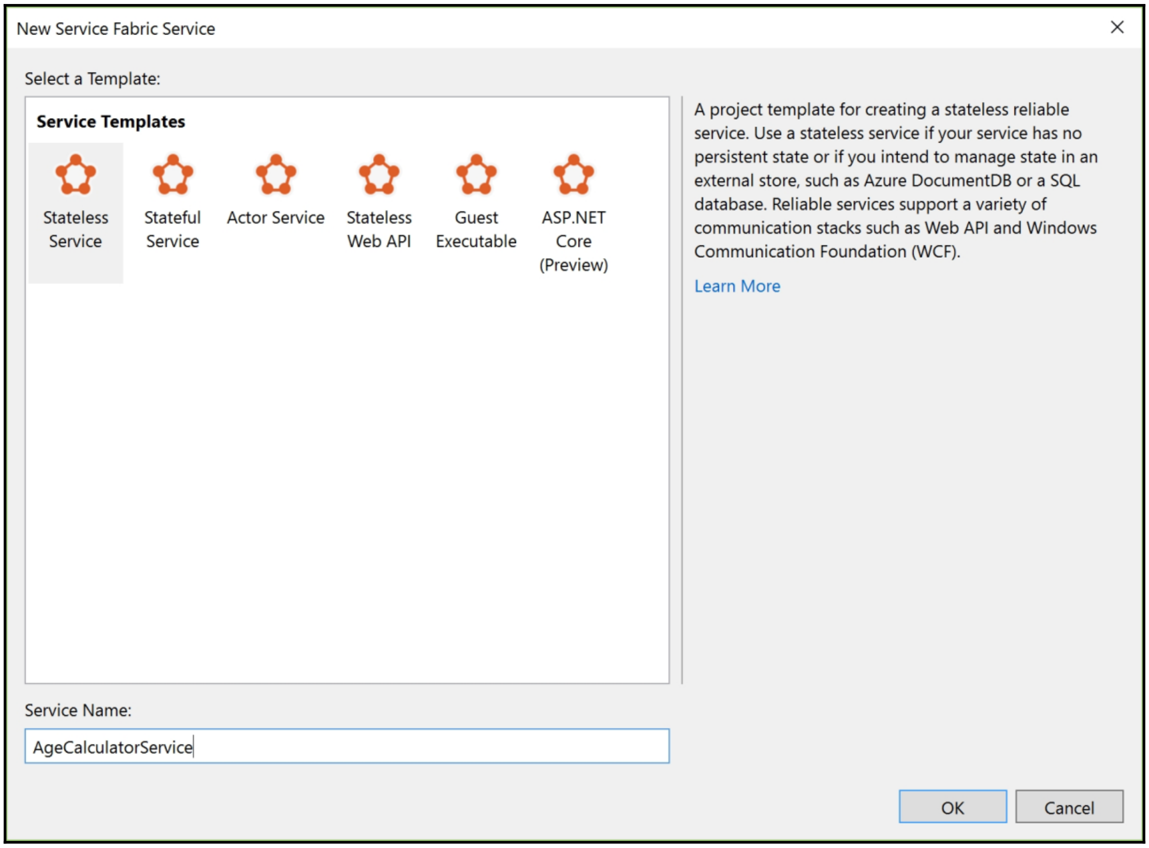
Reliable Services communication

Azure Service Fabric gives you the flexibility to implement custom communication stacks using protocols of your choice. To implement a custom communication stack, you need to implement the `ICommunicationListener` interface. The Reliable Services application framework provides a couple of inbuilt communication stacks that you can use, such as the default stack built on RPC proxy, WCF, REST (Web API), and HTTP (ASP.NET).

Let us build a custom stack using ASP.NET, Web API, and open web interface for .NET (OWIN) self-hosting in Service Fabric stateless Reliable Service.

This sample is inspired from the official *Service Fabric Web API services with OWIN self-hosting* sample from MSDN: `https://azure.microsoft.com/en-us/documentation/articles/service-fabric-reliable-services-communication-webapi/`.
If you are not familiar with Web API. This is a great link to start: `https://www.asp.net/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api`.
You can find an overview of the Katana project here: `https://www.asp.net/aspnet/overview/owin-and-katana/an-overview-of-project-katana`.

Using Visual Studio, create a Service Fabric application with a single stateless service named `AgeCalculatorService`:



A project template for creating a stateless reliable service. Use a stateless service if your service has no persistent state or if you intend to manage state in an external store, such as Azure DocumentDB or a SQL database. Reliable services support a variety of communication stacks such as Web API and Windows Communication Foundation (WCF).

Service Name: AgeCalculatorService

Create stateless service template

Once the project is ready, we will pull in the `Microsoft.AspNet.WebApi.OwinSelfHost` `NuGet` package for Web API in the service project. This package includes all the necessary Web API packages and the host packages.

After installing the package, we will build a basic Web API project structure. Let us add a `Controllers` directory and a simple controller named `AgeController`:

```
public class AgeController : ApiController
{
    public string Get(DateTime dateOfBirth)
    {
        return $"You are {DateTime.UtcNow.Year - dateOfBirth.Year}
        years old.";
    }
}
```

Next, to register routes and other configurations, add a `Startup` class in the project root directory:

```
namespace AgeCalculatorService
{
    using System.Web.Http;

    using Owin;

    public static class Startup
    {
        public static void ConfigureApp(IAppBuilder appBuilder)
        {
            var config = new HttpConfiguration();

            config.Routes.MapHttpRoute(
                "DefaultApi",
                "api/{controller}/{id}",
                new { id = RouteParameter.Optional });

            appBuilder.UseWebApi(config);
        }
    }
}
```

Our application is now ready. Service Fabric executes your service in service host process, which is an executable that runs your service code. You compile your service in the form of an executable file that registers your service type and executes your code. When you open the `Program.cs` file, you will find the `Main` method which is the entry point of the service host process. This method contains the code essential for mapping your service to the related service type. You will find that a parameter named context is passed to your service instance, which, depending on the service type, stateful or stateless, will either be a `StatefulServiceContext` or a `StatelessServiceContext`. Both the context classes are derived from `ServiceContext`. If you want to setup dependency injection in your service, you can utilize the `RegisterServiceAsync` method to do that:

```
    internal static class Program
    {
        private static void Main()
        {
            try
            {
                ServiceRuntime.RegisterServiceAsync(
                    "AgeCalculatorServiceType",
                    context => new
              AgeCalculatorService(context)).GetAwaiter().GetResult();
```

```
                ServiceEventSource.Current.ServiceTypeRegistered(
                    Process.GetCurrentProcess().Id,
                    typeof(AgeCalculatorService).Name);

                Thread.Sleep(Timeout.Infinite);
            }
            catch (Exception e)
            {
ServiceEventSource.Current.ServiceHostInitializationFailed(e.ToString());
                throw;
            }
        }
    }
```

Your application runs in its own process and this code shows just that. Your application is nothing but a console application that is managed by the Service Fabric runtime. **OWIN** decouples ASP.NET from web servers. Therefore, you can start a self-hosted OWIN web server inside your application process. We will implement the `ICommunicationListener` interface to launch an OWIN server, which will use the `IAppBuilder` interface to initialize the ASP.NET Web API. 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

Before we implement the communication stack, we need to configure the endpoints for our service. Service Fabric ensures that the endpoints are available for our service to use. The Service Fabric host process runs under restricted credentials and therefore your application won't have sufficient permissions to setup ports that it needs to listen on. You can setup endpoints for your service in `PackageRoot\ServiceManifest.xml`:

```
<Resources>
  <Endpoints>
    <Endpoint Name="ServiceEndpoint" Type="Input" Protocol="http"
       Port="80" />
  </Endpoints>
</Resources>
```

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 guarantees that only one instance of a stateless service will be deployed on a single node. Therefore, you don't need to worry about multiple applications listening on the same port in this case. In other cases, your service replicas might get deployed to the same host and therefore might be listening on the same port. Therefore, your communication listener must support port sharing. Microsoft recommends that your application communication listeners listen to traffic on an endpoint that is built using combination of partition ID and replica/instance ID, which is guaranteed to be unique.

Create a class named `OwinCommunicationListener` that implements the `ICommunicationListener` interface. We will add private class member variables for values and references that the listener will need to function. These private members will be initialized through the constructor and used later when you set up the listening URL.

```
internal class OwinCommunicationListener : ICommunicationListener
{
    private readonly string appRoot;

    private readonly string endpointName;

    private readonly ServiceEventSource eventSource;

    private readonly ServiceContext serviceContext;

    private readonly Action<IAppBuilder> startup;

    private string listeningAddress;

    private string publishAddress;

    private IDisposable webApp;

    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;
            }
    }
```

Now, let's implement the `OpenAsync` method. In this method, we will start the web server and assign it the URL it will listen on. Since this is a stateless service, we do not need to have the partition and replica identifiers suffixed to the URL to make it unique:

```
        public Task<string> OpenAsync(CancellationToken cancellationToken)
        {
            var serviceEndpoint =
                 this.serviceContext.CodePackageActivationContext.
                    GetEndpoint(this.endpointName);
                var protocol = serviceEndpoint.Protocol;
                var port = serviceEndpoint.Port;

                if (this.serviceContext is StatefulServiceContext)
                {
                    var 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;
            }
        }
```

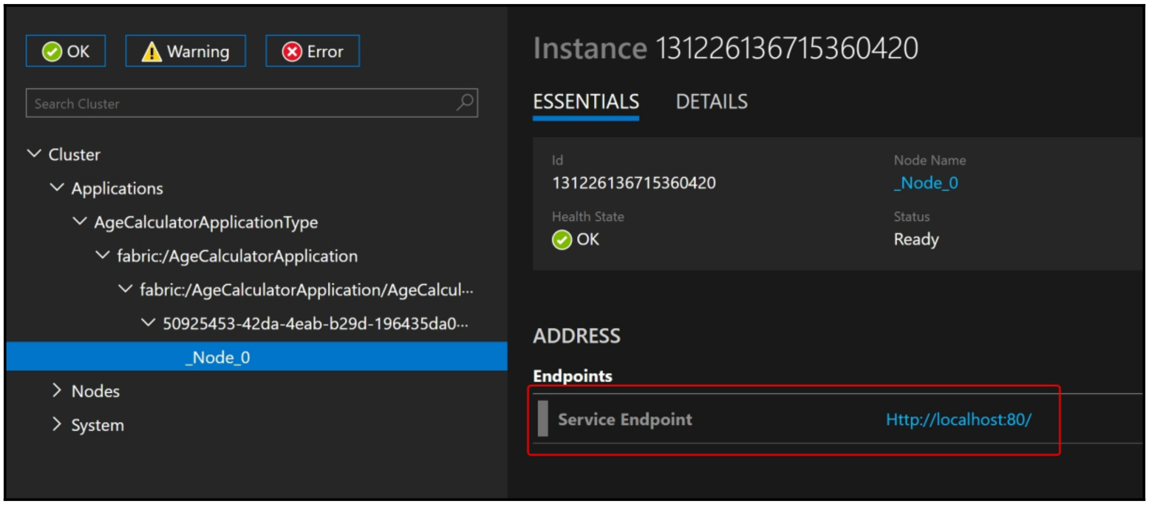
This method starts the web server and returns the address that the server is listening on. This address is registered with the Naming Service, which is a cluster service. A client can use the service name and get this address from the Naming Service of the cluster. Let's implement the `CloseAsync` and the Abort methods to complete the implementation:

```
        public void Abort()
        {
            this.eventSource.Message("Aborting web server on endpoint
              {0}", this.endpointName);

            this.StopWebServer();
        }

        public Task CloseAsync(CancellationToken cancellationToken)
        {
            this.eventSource.Message("Closing web server on endpoint
              {0}", this.endpointName);

            this.StopWebServer();

            return Task.FromResult(true);
        }
        private void StopWebServer()
        {
            if (this.webApp != null)
            {
                try
                {
                    this.webApp.Dispose();
                }
                catch (ObjectDisposedException)
                {
                    // no-op
                }
            }
        }
```

Finally, we need to override the `CreateServiceInstanceListeners` method in our service implementation to create and return an instance of `OwinCommunicationListener` in the `AgeCalculatorService` class.

```
            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));
        }
```

Remove the default implementation of `RunAsync` in the `AgeCalculatorService` class as we don't need to perform any processing in the background. Build and deploy the application on your local cluster. You can find the address of the application in Service Fabric Explorer on which you can send a request to your API:


OK
Warning
Error
Search Cluster
Cluster
Applications
AgeCalculatorApplicationType
fabric:/AgeCalculatorApplication
fabric:/AgeCalculatorApplication/AgeCalcul...
50925453-42da-4eab-b29d-196435da0...
_Node_0
Nodes
System
ESSENTIALS
DETAILS

Instance 131226136715360420

Id: 131226136715360420
Node Name: _Node_0
Health State: OK
Status: Ready

ADDRESS

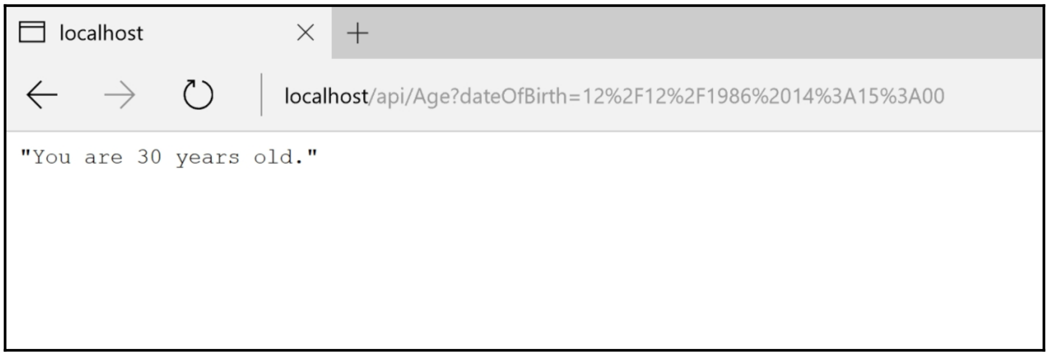
Endpoints

Service Endpoint   Http://localhost:80/

Service endpoint

In the browser, send a request to your application with your date of birth appended as a query string parameter to the request. For example: `http://localhost/api/Age?dateOfBirth=12-12-2001`.

After submitting the request, you should see a result as follows:



```
"You are 30 years old."
```

Output of age calculator

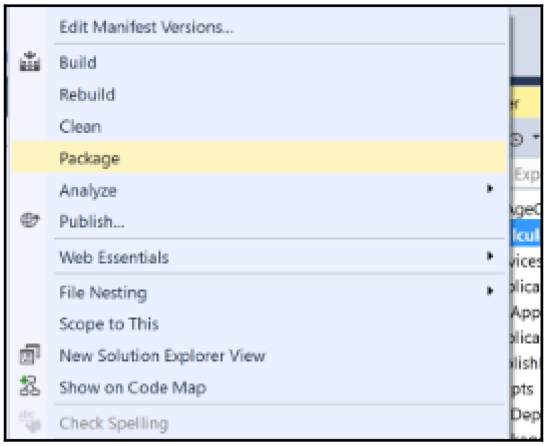
Exploring the application model

A service fabric application is made up of several Microservices. A Microservice is further made up of three parts:

- **Code**: These are the executable binaries of the Microservice. The binaries are packaged in code package.
- **Configuration**: These are the settings that can be used by your Microservices at run time. These are packaged into a configuration package.
- **Data**: Any static data that is used by the Microservice is packaged in data package.

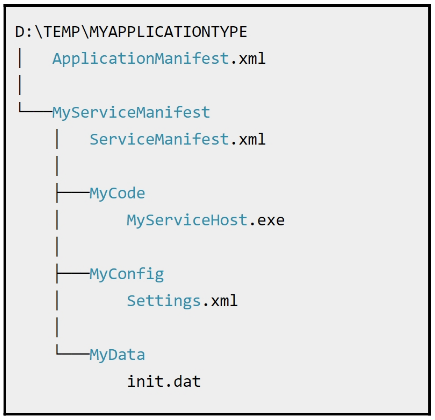
A Service Fabric application is described by a set of versioned manifests, an application manifest, and several service manifests, one each for each of the Microservices that make up the application. Let's explore the components of a Service Fabric application by navigating through the Service Fabric application package.

To package your application, right click on your Service Fabric application project and click on the **Package** option:



Package application

Once the application has finished packaging, open the package folder by navigating to the package location that is shown in the output window. Your application package will have a layout like that shown in the following image:



```
D:\TEMP\MYAPPLICATIONTYPE
│   ApplicationManifest.xml
│
└───MyServiceManifest
    │   ServiceManifest.xml
    │
    ├───MyCode
    │       MyServiceHost.exe
    │
    ├───MyConfig
    │       Settings.xml
    │
    └───MyData
            init.dat
```

Let's discuss the various files available in your package in a bit more detail:

- **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**: There is one service manifest for each reliable service in the application. The service manifest describes the components of a Microservice. 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.
- **Code package**: Each Microservice has a code package that contains the executable and the dependencies that are required by the executable to run.
- **Config Package**: You can package the configurations that are required by your Microservice in this folder. A config package is simply a directory in your project which can contain multiple configuration files. Except for `Settings.xml` file, which is added by default to your reliable service project, the rest of the configurations are not processed by Service Fabric and would need to be processed by the code of your Microservice.

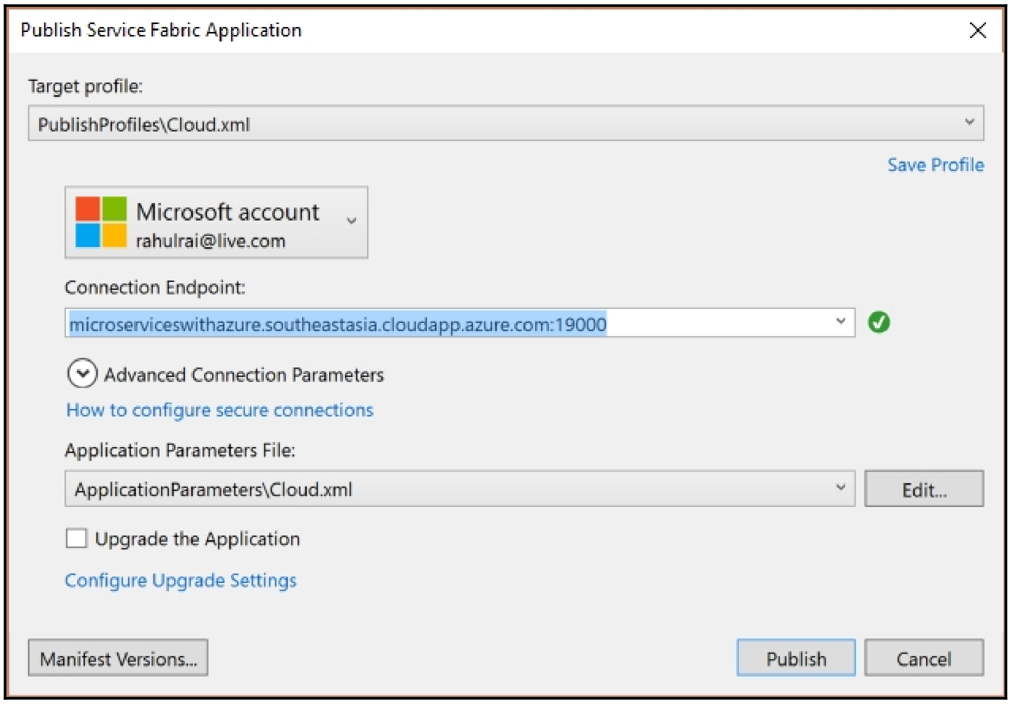
  The `Settings.xml` file can store custom configuration sections and parameters that you can use in your application. For example, you can add a custom configuration section with parameters in your `Settings.xml` file.

```
    <Section Name="CustomConfigSection">
              <Parameter Name="MyParameter" Value="Value1" />
    </Section>
  You can retrieve this configuration value from your service code.
         var configPkg =  context.CodePackageActivationContext.
            GetConfigurationPackageObject("Config");
         var customSection = configPkg.Settings.
            Sections["CustomConfigSection"];
         var value = customSection.
            Parameters["CustomConfigSection"].Value;
```

Service Fabric allows you to override the configurations based on the environment where the application is hosted as well.

- **Data package**: Any static data that is required by your application can be placed inside folders in the `PackageRoot` directory. Each folder represents one data package and can contain multiple data files. Service Fabric does not process any file that you place inside the folders and the code of your service is responsible for parsing and reading the data.

When you deploy your application from Visual Studio, it creates a package and uses the `Deploy-FabricApplication.ps1` script present in the `Script` folder to deploy your application package to Azure. To deploy your application to Azure, right click on the application project and select the `Publish` option in the context menu to bring up the publish dialog.


Publish Service Fabric Application

Target profile: PublishProfiles\Cloud.xml

Microsoft account
rahulrai@live.com

Connection Endpoint: microserviceswithazure.southeastasia.cloudapp.azure.com:19000

Application Parameters File: ApplicationParameters\Cloud.xml

Upgrade the Application [ ]

Save Profile
Advanced Connection Parameters
How to configure secure connections
Edit...
Configure Upgrade Settings
Manifest Versions...
Publish
Cancel

Publish application dialog

In the publish dialog, you can select the relevant application parameter file that you want to get deployed with your application. You can modify the parameters just before deployment by clicking the **Edit** button. In the dialog box that follows, you can enter or change the parameter's value in the **Parameters** grid. When you're done, choose the **Save** button.

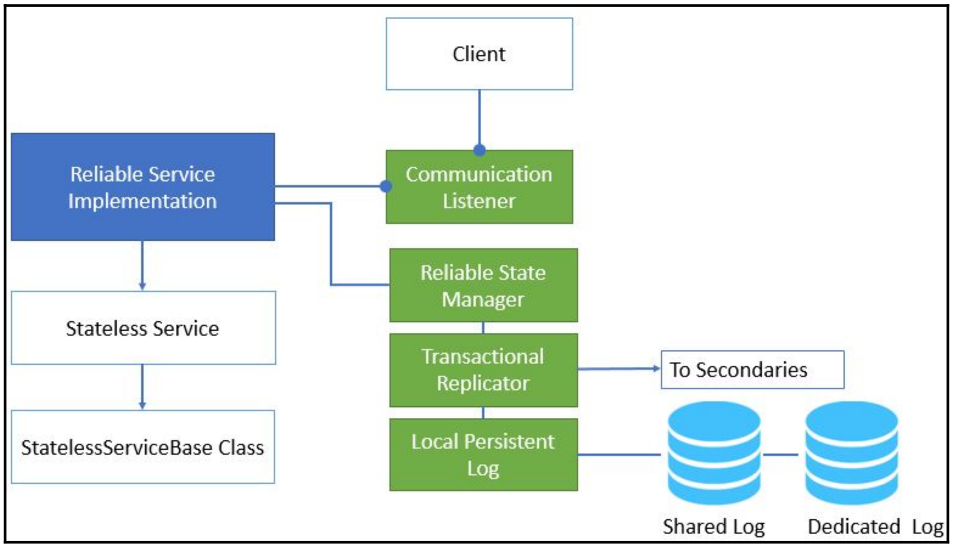
You can deploy your application to the selected cluster by clicking on the **Publish** button.

Stateful service

A stateful Microservice stores state information stored in a consistent and highly available manner using reliable data structure. By default, the state data is saved on the disk of the compute node. However, you can write your own state providers to store state data externally and consume it. Service Fabric ensures that your state data is consistent and highly available so that, in case of failure of primary compute node, a secondary node can resume processing without loss of data.

Stateful service architecture

A stateful service takes requests from clients through the `ICommunicationListener` interface:



Stateful Reliable Service architecture

The implementation of a stateful reliable service derives from `StatefulService` class. Service Fabric runtime manages the lifetime of your stateful Microservice through the `StatefulService` class. The service can use Reliable Collections to persist state information, which are similar to collections in `System.Collections` namespace but with added features for *high availability* and *consistency*. The read and write operations of Reliable Collections go through **Reliable State Manager**. The state is replicated to secondary nodes by a **Transactional Replicator**. This replication ensures that state data is reliable, consistent, and highly available. After ensuring that state data has been copied to secondary replicas, the **Transactional Replicator** invokes the logger. The logger is responsible for persisting state data to disks by using append-only log files. In case of failure, state can be restored by replaying the logs.

There are two types of logs that store the state data. The shared logs are stored under node-level working directory. The logs in shared log are copied lazily to dedicated logs that are stored under service working directory.

Reliable Collections

The `Microsoft.ServiceFabric.Data.Collections` namespace contains various collections that act as reliable state providers and are known as Reliable Collections. You must have used standard data collections such as dictionary and queue present in the `System.Collections` namespace in your applications; the interfaces in Reliable Collections allow you to interact with the state provider in a similar manner. The Reliable Collections differ from the classes in `System.Collections`, in that they are:

- **Replicated**: The state data is replicated across nodes for high availability
- **Persisted**: The state data is persisted both in memory and in disk for durability against large-scale outages
- **Asynchronous**: The Service Fabric API supports asynchronous operations on the Reliable Collections to ensure that threads are not blocked when incurring I/O
- **Transactional**: Service Fabric APIs utilize the abstraction of transactions so you can manage multiple Reliable Collections within a service easily

The Service Fabric API implements the Reliable Collections in a pluggable manner, so you can expect to see new collections being added to the existing offering from time to time. Service Fabric currently provides three Reliable Collections:

- **Reliable Dictionary**: It 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**: It represents a replicated, transactional, and asynchronous strict **first-in**, **first-out** (**FIFO**) queue. Similar to `ConcurrentQueue`, the value can be of any type.
- **Reliable Concurrent Queue**: This queue is similar to Reliable Queue but offer a higher throughput in lieu of doing away with FIFO restrictions that Reliable Queue has.

The Reliable State Manager manages the lifetime of the state providers. Since the data structure needs to be replicated to several nodes, coordination among nodes is required to create instance of the data structure. Using the `IReliableStateManager` interface, your service can access the state manager to create new instances of Reliable Collections. The Reliable State Managers also support transactions. Using transactions, your service can operate with several collections in an atomic manner.

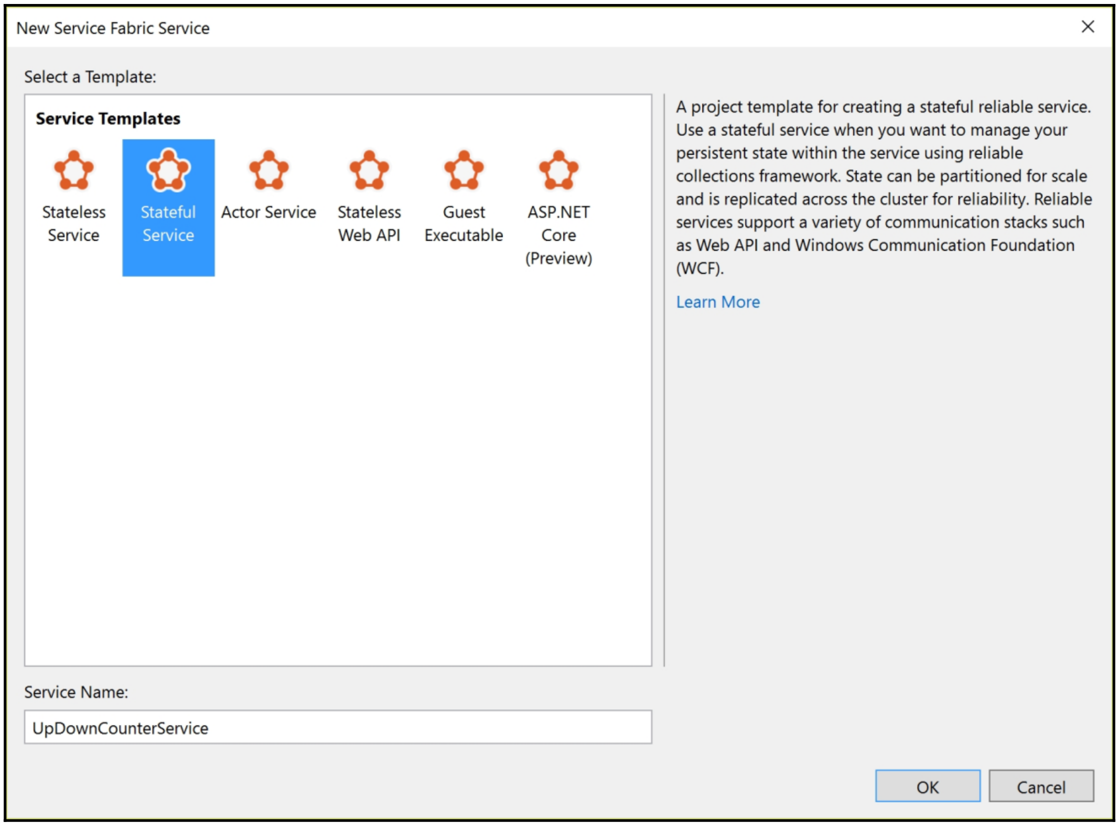
All state data that your site writes or modifies must go through the primary replica. Service Fabric ensures that there are is more than one primary replica of a service available at any point of time. However, data can be read from either the primary replica or secondary replicas.

Let's build a simple application which will help us understand how we can work with Reliable Collections.

Up and down counter application

Let's build a simple stateful application that would help us understand the various aspects of a reliable stateful application. We will build an up and down counter that will increment or decrement a counter value depending on the partition it belongs to. We will save the counter value in a `ReliableDictionary` instance and trace the current state of the dictionary so that we can visualize it in the **Diagnostics Event Viewer** console.

To begin, using Visual Studio create a new Service Fabric application named `StatefulUpDownCounterApplication` and add a Stateful Reliable Service to it. Name the service `UpDownCounterService`:


New Service Fabric Service

Select a Template:

Service Templates

Stateless Service | Stateful Service | Actor Service | Stateless Web API | Guest Executable | ASP.NET Core (Preview)

A project template for creating a stateful reliable service. Use a stateful service when you want to manage your persistent state within the service using reliable collections framework. State can be partitioned for scale and is replicated across the cluster for reliability. Reliable services support a variety of communication stacks such as Web API and Windows Communication Foundation (WCF).

Learn More

Service Name: UpDownCounterService

Create Stateful Service dialog

Let's first create two partitions of this service. For this sample, we will use the named partitioning scheme. Applications that use named partitions usually work with categorized data, for instance an election service that persists vote data by state. Clients of applications that use the named partition scheme need to explicitly specify which partition they want to access at runtime.

To create a named partition, navigate to the `ApplicationManifest.xml` file and update the contents of the `Service` node to the following:

```
       <Service Name="UpDownCounterService">
         <StatefulService ServiceTypeName="UpDownCounterServiceType"
TargetReplicaSetSize="[UpDownCounterService_TargetReplicaSetSize]"
MinReplicaSetSize="[UpDownCounterService_MinReplicaSetSize]">
           <NamedPartition>
             <Partition Name="UpCounter" />
             <Partition Name="DownCounter" />
           </NamedPartition>
         </StatefulService>
```

Using the preceding configuration Service Fabric will create two partitions named `UpCounter` and `DownCounter` for your service.

Next, navigate to the `UpDownCounterService` class and clear the default code present in the `RunAsync` method. Let's start placing our counter code inside this method.

First, we need to identify the partition in which our process is getting executed.

The following code block, creates an instance of `ReliableDictionary` named `counter`. We use the `Partition` property of the base class `StatefulService` to retrieve the partition information.

```
            var myDictionary = await
this.StateManager.GetOrAddAsync<IReliableDictionary<string,
long>>("counter");
            var myPartitionName = (this.Partition.PartitionInfo as
NamedPartitionInformation)?.Name;
```

The variable `myPartitionName` will contain the name of one of the partitions that we had created earlier. Let's consider the case when the value of the `myPartitionName` variable will be `"UpCounter"`. In this case, we will initialize the dictionary with `0` and keep updating the value by one every five seconds. The code is as follows:

```
            switch (myPartitionName)
            {
                case "UpCounter":
                    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);
                            await tx.CommitAsync();
                        }

                        await Task.Delay(TimeSpan.FromSeconds(5),
                          cancellationToken);
                    }
                case "DownCounter":
...
            }
```

All operations on Reliable Collections happen inside transaction scope. This is because the data in the collections need to be actively replicated to majority of secondary nodes to ensure high availability of state data. A failure to replicate data to majority of replicas would lead to failure of transaction.

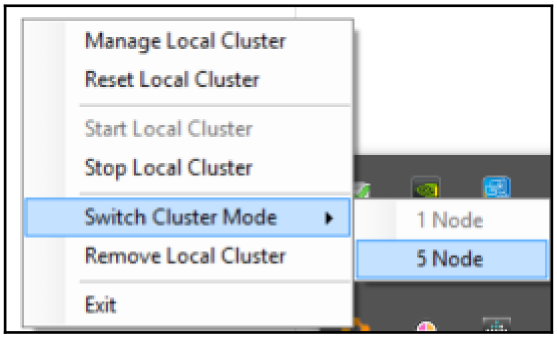
Next, we need to implement the down counter. We will write a similar implementation as we did for up counter, except that the counter value will get initialized with `1000` and it will decrement every five seconds. The code for down counter is as follows:

```
                case "DownCounter":
                    while (true)
                    {
                        cancellationToken.ThrowIfCancellationRequested();
                        using (var tx = this.StateManager.
                                 CreateTransaction())
                        {
                            var result = await myDictionary.
                              TryGetValueAsync(tx, "Counter");
                            ServiceEventSource.Current.ServiceMessage(
                                this.Context,
                                "Current Counter Value: {0}",
                                result.HasValue ? result.Value.
                                 ToString() : "Value does not
                                                      exist.");
                            await myDictionary.AddOrUpdateAsync(tx,
                             "Counter", 1000, (key, value) => --
                                                              value);
                            await tx.CommitAsync();
                        }
                        await Task.Delay(TimeSpan.FromSeconds(5),
                          cancellationToken);
                    }
```

Let's also set the replica count of the service to `3` so that we have multiple secondary replicas of the service. Locate the parameter file `Local.5Node.xml` in the `ApplicationParameters` folder. This file is used to supply parameters when you are running your local cluster in five node mode. This mode spins five host processes on your machine to mimic deployment on five nodes. Validate that the minimum number of replicas and target number of replicas is set to three (at least):

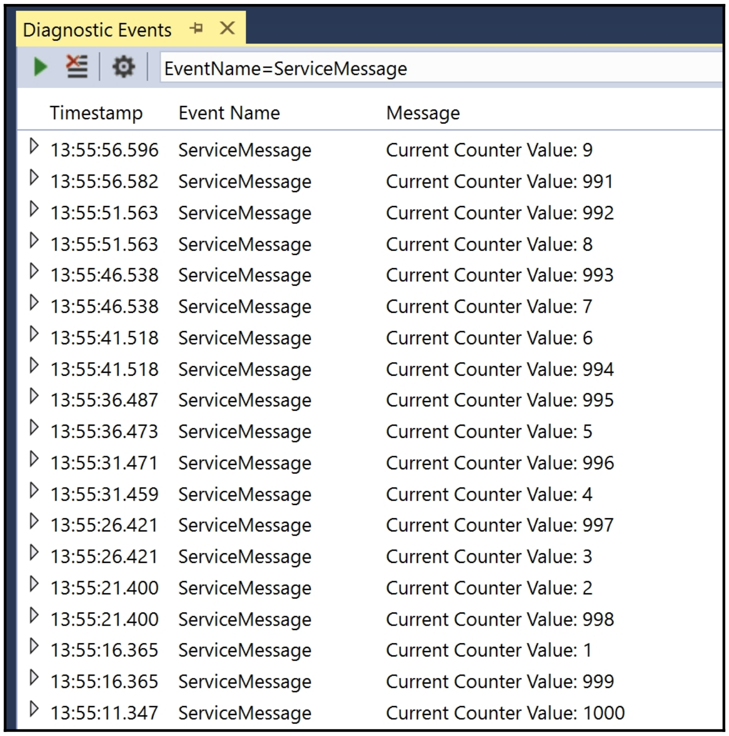
```
<Parameters>
  <Parameter Name="UpDownCounterService_MinReplicaSetSize" Value="3" />
  <Parameter Name="UpDownCounterService_TargetReplicaSetSize" Value="3"
    />
</Parameters>
```

Next, switch the cluster mode to **5 Node**, if not done already, by right clicking the Service Fabric Local Cluster Manager icon in the task bar and selecting the appropriate mode:



Switch cluster mode

Let's quickly deploy our application to the local cluster by pressing *F5*. Open the **Diagnostic Events** window and observe the output of the application:

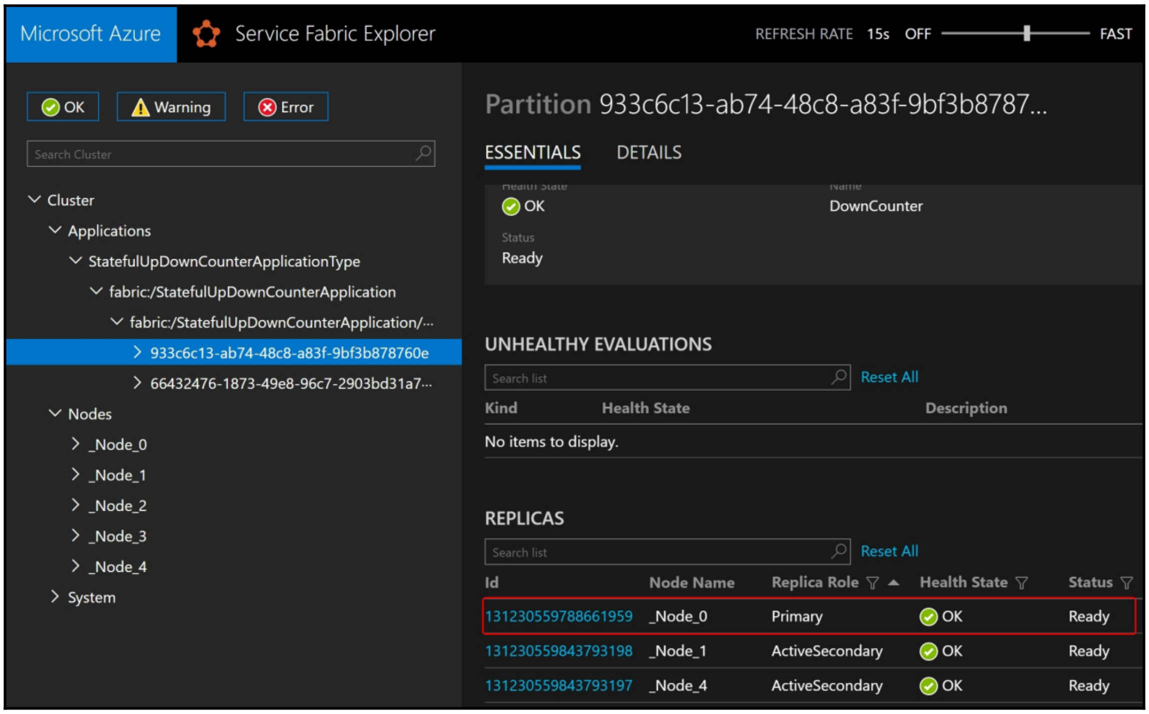


```
EventName=ServiceMessage

Timestamp      Event Name       Message
13:55:56.596   ServiceMessage   Current Counter Value: 9
13:55:56.582   ServiceMessage   Current Counter Value: 991
13:55:51.563   ServiceMessage   Current Counter Value: 992
13:55:51.563   ServiceMessage   Current Counter Value: 8
13:55:46.538   ServiceMessage   Current Counter Value: 993
13:55:46.538   ServiceMessage   Current Counter Value: 7
13:55:41.518   ServiceMessage   Current Counter Value: 6
13:55:41.518   ServiceMessage   Current Counter Value: 994
13:55:36.487   ServiceMessage   Current Counter Value: 995
13:55:36.473   ServiceMessage   Current Counter Value: 5
13:55:31.471   ServiceMessage   Current Counter Value: 996
13:55:31.459   ServiceMessage   Current Counter Value: 4
13:55:26.421   ServiceMessage   Current Counter Value: 997
13:55:26.421   ServiceMessage   Current Counter Value: 3
13:55:21.400   ServiceMessage   Current Counter Value: 2
13:55:21.400   ServiceMessage   Current Counter Value: 998
13:55:16.365   ServiceMessage   Current Counter Value: 1
13:55:16.365   ServiceMessage   Current Counter Value: 999
13:55:11.347   ServiceMessage   Current Counter Value: 1000
```

Output in Diagnostic Events

You would notice that even though both the counters referenced the same dictionary named `counter`, the two counters are progressing independently. That is because the Reliable Collections are not shared across partitions, but only across replicas.

Next, let's kill the primary replicas of both the partitions to see whether the state data is lost. To do so, open the Service Fabric Explorer and find out which node is hosting the primary replica of your service partitions:



Microsoft Azure Service Fabric Explorer REFRESH RATE 15s OFF FAST

OK Warning Error

Search Cluster

- Cluster
  - Applications
    - StatefulUpDownCounterApplicationType
      - fabric:/StatefulUpDownCounterApplication
        - fabric:/StatefulUpDownCounterApplication/...
          - 933c6c13-ab74-48c8-a83f-9bf3b878760e
          - 66432476-1873-49e8-96c7-2903bd31a7...
  - Nodes
    - _Node_0
    - _Node_1
    - _Node_2
    - _Node_3
    - _Node_4
  - System

Partition 933c6c13-ab74-48c8-a83f-9bf3b8787...

ESSENTIALS DETAILS

Health State: OK
Name: DownCounter
Status: Ready

UNHEALTHY EVALUATIONS

Search list Reset All

Kind	Health State	Description
No items to display.		

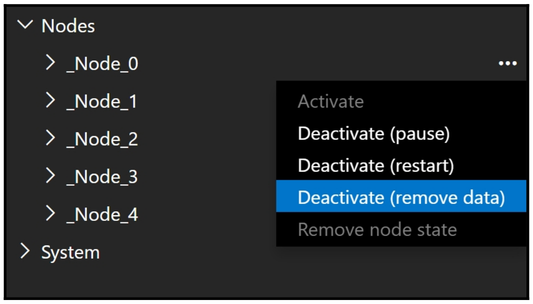
REPLICAS

Search list Reset All

Id	Node Name	Replica Role	Health State	Status
131230559788661959	_Node_0	Primary	OK	Ready
131230559843793198	_Node_1	ActiveSecondary	OK	Ready
131230559843793197	_Node_4	ActiveSecondary	OK	Ready

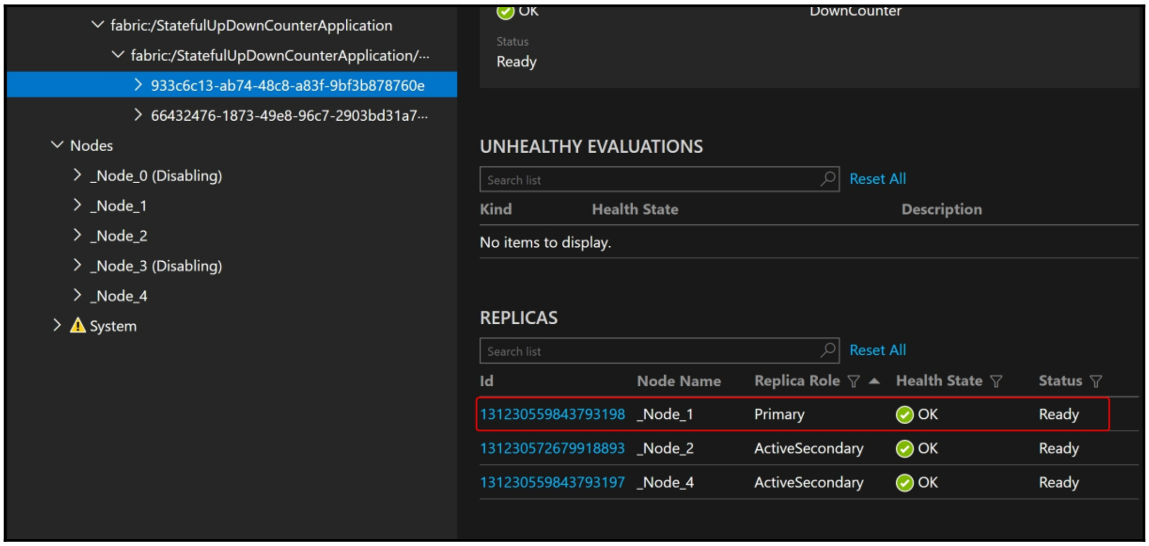
Primary replicas in Service Fabric Explorer

In my case **_Node_0** and **_Node_3** host the primary replicas of the two partitions of the service that we just built. Let's deactivate the nodes and observe the counter value. Click on the ellipsis next to the node to reveal the various node operations. Let's simulate a complete node crash by selecting the **Deactivate (remove data)** option:



Deactivate node operation

You will notice that as soon as you deactivate the node, an active secondary replica on another node, **_Node_1** in my case, gets promoted to primary status and resumes processing without loss of state data which you can verify by looking at the logs accumulated in the **Diagnostic Events** window:



Kind	Health State	Description
No items to display.		

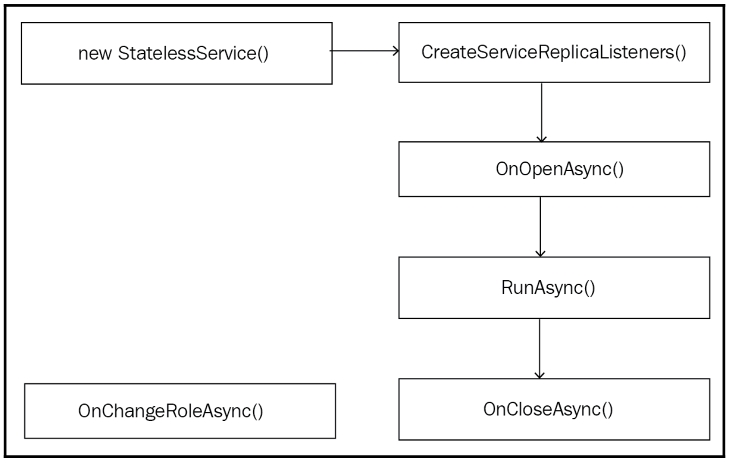
Id	Node Name	Replica Role	Health State	Status
131230559843793198	_Node_1	Primary	✓ OK	Ready
131230572679918893	_Node_2	ActiveSecondary	✓ OK	Ready
131230559843793197	_Node_4	ActiveSecondary	✓ OK	Ready

New primary replica

Stateful service lifecycle

Let's explore the lifecycle of a stateful service. The lifetime of a stateful Microservice replica starts as soon as you register a new instance of the service with the Service Fabric runtime. Most of the lifetime methods are same as that of stateless Microservice. Let's look at those that are different:

- `RunAsync (CancellationToken)`: 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. This method executes only in the active primary replica.
- `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. The stateful service lifetime can be seen in the following image:



Service partitioning

To support scalability and reliability, Service Fabric supports setting up multiple instances of service and supports partitioning a service. Let's see what this concept means for stateless and stateful services.

Stateless services do not store any data locally. When you add a new stateless service in your Service Fabric application, you will find that a configuration is automatically created in `ApplicationManifest.xml` to configure the service to use `SingletonPartition`:

```
<Service Name="Stateless1">
  <StatelessService ServiceTypeName="Stateless1Type"
     InstanceCount="[Stateless1_InstanceCount]">
    <SingletonPartition />
  </StatelessService>
</Service>
```

`SingletonPartition` means that, number of partitions is one, which means that there is a single instance of state. If you set the value of `[Stateless1_InstanceCount]` parameter to `2`, you will have two instances of the service running on different nodes. If you put a breakpoint in the `RunAsymc()` method of `Stateless1` class, you would find that it will be invoked twice, once for each node on which it is deployed.

Due to absence of state data, other partitioning schemes do not make sense for stateless services. To explore the other partitioning schemes, let's add a stateful service to the project. After the template unfolds, you would find the following configuration in the application manifest file:

```
    <Service Name="Stateful1">
      <StatefulService ServiceTypeName="Stateful1Type"
TargetReplicaSetSize="[Stateful1_TargetReplicaSetSize]"
MinReplicaSetSize="[Stateful1_MinReplicaSetSize]">
        <UniformInt64Partition PartitionCount="[Stateful1_PartitionCount]"
LowKey="-9223372036854775808" HighKey="9223372036854775807" />
      </StatefulService>
    </Service>
```

You will notice that instance count is no longer present in the configuration and is replaced with replica count. A replica is a copy of code and state data of a service. Let's simplify the preceding configuration to make it easier to understand:

```
<Service Name="Stateful1">
  <StatefulService ServiceTypeName="Stateful1Type"
     TargetReplicaSetSize="3" MinReplicaSetSize="2">
    <UniformInt64Partition PartitionCount="2" LowKey="1"
       HighKey="10" />
  </StatefulService>
</Service>
```

By default, your stateful services uses uniform partitioning scheme named `UniformInt64Partition`. This partitioning scheme uniformly distributes a continuous key range to the number of partitions that you specify. Using the preceding configuration, the state data will be partitioned in two parts, one partition will serve keys ranging from `1` to `5` and the other partition will serve keys ranging from `6` to `10`. Each of these partitions will have a minimum of two replicas, which can expand up to three. If you deploy your application, you will find that six instances of your application will spin up, three replicas for each partition. However, if you put breakpoint in `RunAsync ()` method, it will be hit only twice, once for each primary replica of a partition because the rest of the replicas won't be in active state.

Next, let's change the partitioning scheme to `NamedPartition`. Named partition scheme is useful in cases where the number of partitions are known in advance and remain static over time, for example partitioning an application by states in a country:

```
<Service Name="Stateful1">
  <StatefulService ServiceTypeName="Stateful1Type"
     TargetReplicaSetSize="3" MinReplicaSetSize="2">
    <NamedPartition>
      <Partition Name="A" />
      <Partition Name="B" />
      <Partition Name="C" />
      <Partition Name="D" />
    </NamedPartition>
  </StatefulService>
</Service>
```

If you deploy the solution now, you will find that twelve instances of your application will get deployed, three for each partition. However, the breakpoint on the `RunAsync` method will only be hit four times, once for each primary replica of a partition.

Service replicas

As discussed previously, the replica is a copy of state data and code of your application. When you deploy a partitioned stateful service, replicas of each partition form a replica set. Each replica in a replica set takes one of the following roles:

1. **Primary**: This replica caters to all the write requests. A replica set can have only one primary replica. Whenever the state data is created or modified, the change is communicated to the secondary replicas that needs to be acknowledged by them. The primary replica waits for acknowledgements from a quorum of replicas to commit the change.
2. **Active secondary**: The replicas in active secondary participate in the write quorum. All participant active secondaries in the write quorum must acknowledge the change in state after committing the change to their own storage.
3. **Idle secondary**: The idle secondaries receive update from the primary but do not participate in the write quorum. In case an active secondary instance fails, an instance of idle secondary will be promoted to the status of active secondary.
4. **None**: A replica with this state does not hold any state. A replica that is being decommissioned holds this state.

You can view the replica states in Service Fabric Explorer by drilling down to the replica level.

Summary

We started this chapter by building our first Service Fabric stateless application. After which we took a thorough look at the Service Fabric Explorer. Next, we studied stateless and stateful service architecture and lifecycles using Service Fabric Reliable Services. Lastly, we looked at service partition and replicas in detail.

In the next chapter we will discuss another programming model of Service Fabric knows as Reliable Actors.

6
Reliable Actors

Actor model

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.

What is an Actor?

An Actor is a specialized service that is much more granular in intent than the typical services that we build. The Actor programming model ensures that individual entities in your solution are highly cohesive and decoupled. An Actor is designed to work within a set of constraints:

- The various Actors of an application can only interact through asynchronous message passing. The messages that are exchanged between Actors should be immutable.
- An Actor can function within the boundary of domain that it is designed for. For instance, a shopping cart actor cannot implement or expose functionality of a product listing Actor.

- An Actor can only change its state when it receives and processes a message.
- An Actor should be single-threaded and it should process only one message at a time. Another message should be picked up by the Actor only when the Actor operation has completed.
- An Actor may spawn new Actors and send a finite number of messages to the other Actors in the application.

Actors in Service Fabric

Each Reliable Actor service you write is a partitioned and stateful Reliable Service. Service Fabric API provides a asynchronous and single-threaded programming model to implement Actors. You only need to work on the Actor implementation, the platform takes care of lifecycle, upgrades, scale, activation, and so on.

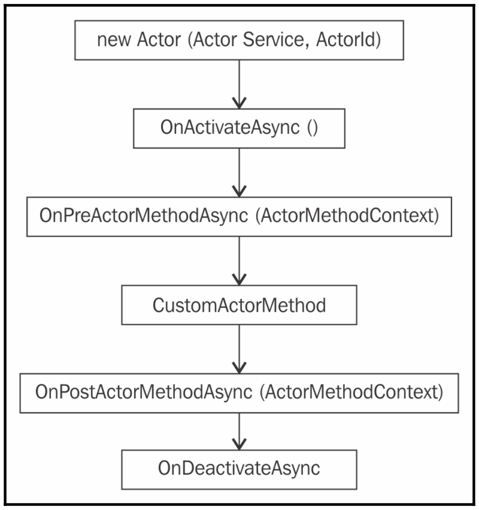
Actors closely resemble objects in object-oriented programming. Similar to the way a .NET object is an instance of a .NET type, every Actor is defined as an instance of an Actor type. For example, there may be an Actor type that implements the functionality of a sensor monitor and there could be many Actors of that type that are distributed on various nodes across a cluster. Each such Actor is uniquely identified by an Actor ID. Repeated calls to the same Actor ID are routed to the same Actor instance. For this reason, Actor services are always stateful services.

Actor lifetime

Unlike Actor implementation on other platforms, such as **Akka.net** (`https://petabridge.com/bootcamp/`), Service Fabric Actors are virtual. What this means is that the Service Fabric Reliable Actors runtime manages the location and activation of Actor instances. The Reliable Actors runtime automatically activates an Actor the first time it receives a request for that Actor ID. If an Actor instance remains unused for a period of time, the Reliable Actors runtime garbage-collects the in-memory object. The Reliable Actors runtime also maintains knowledge of the Actor's existence in the state manager for any future Actor reactivations. Actor instances are activated only when the runtime receives a message that is intended to be served by the Actor. If a message is sent to the Actor instance and there is no activation of that Actor on any cluster node, then the runtime will pick a node and create a new activation there.

Every Actor has a unique identifier and calling any Actor method for an Actor ID makes the runtime activates that Actor. For this reason, the Actor constructors are not called by the client but are called implicitly by the runtime. Therefore, an Actor type's constructor can not accept parameters from the client code, although parameters may be passed to the Actor's constructor by the service itself. There is no single entry point for the activation of an Actor from the client.

Even though the client of an Actor does not have the capability to activate the Actor's constructor, the client does have the ability to explicitly delete an Actor and its state. The following diagram shows the lifecycle of an actor:


new Actor (Actor Service, ActorId)
OnActivateAsync ()
OnPreActorMethodAsync (ActorMethodContext)
CustomActorMethod
OnPostActorMethodAsync (ActorMethodContext)
OnDeactivateAsync

Reliable Actors lifetime

Every Actor derives from the *Actor* class. The runtime is responsible for invoking the Actor constructor. Every time an Actor instance is activated the `OnActivateAsync` method is invoked. You can override this method to initialize your Actor instance. Next, for every request sent to the Actor a pre-handle and a post-handle is available. If you want to log all operations that your Actor performs, then you can override these methods to add logging statements. Finally, the `OnDeactivateAsyc` method can be used to perform cleanup activities in Actor.

Saving state

Actors generally need to persist their internal state so that they can recover it in case an Actor is started or restarted, in case of node crashes, or migrated across nodes in cluster. State persistence is also necessary to build complex Actor workflows that transform and enrich input data and generate resultant data that helps make decisions to carry out further operations.

For example, for an automobile system, a fuel Actor may persist the fuel consumption in its state to later calculate the mileage of the vehicle, which may later help decide whether the vehicle requires a maintenance check.

The Actor base class contains the read only `StateManager` property that can be used to operate with state data. The following lines of code, can save and retrieve state data where the argument `cancellationToken` is an object of type `CancellationToken`:

```
//  Save state data
this.StateManager.TryAddStateAsync("count", 0);
// Read state data
var result = await this.StateManager.GetStateAsync<int>("count",
cancellationToken);
```

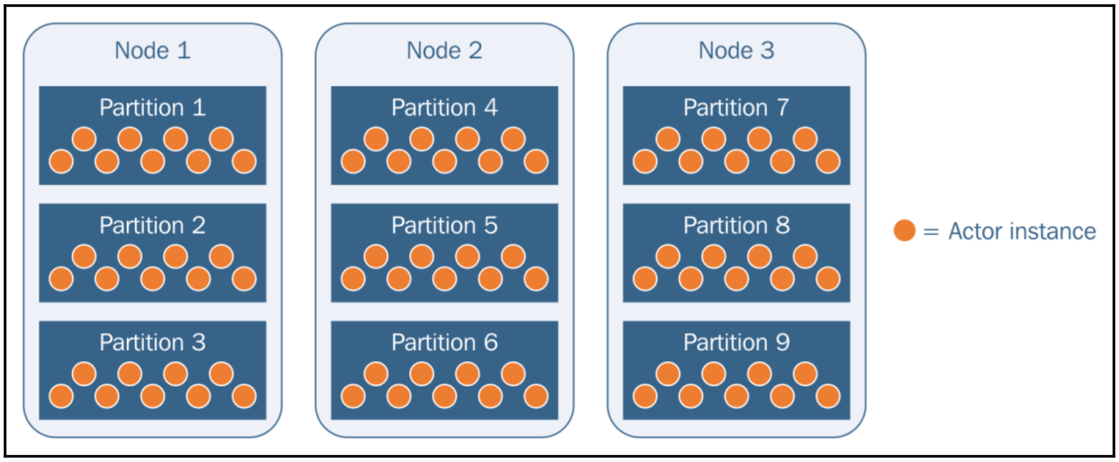
If you want to initialize the state, then you should do so in the `OnActivateAsync` method. Finally, since Actors are single-threaded, we do not need to add concurrency checks while saving or reading state data. Also, since the state is local to the Actor instance, you don't need to add Actor identifier while preserving the state. What this means is that if an Actor named *A* preserves something in state then the information won't be visible to another instance of the same Actor named *B*. The state information can only be operated upon by the Actor named *A*.

Distribution and failover

The Reliable Actors runtime manages scalability and reliability of Actors by distributing Actors throughout the cluster and automatically migrating them from failed nodes to healthy ones when required.

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 like this:



Actor instances distributed among partitions

The partition scheme and key range settings are taken care of by the runtime. Unlike Reliable Services, the Reliable Actors service is restricted to the range partitioning scheme (the uniform Int64 scheme) and requires you to use the full Int64 key range. By default, Actors are randomly placed into partitions resulting in uniform distribution. Communication between Actors happen over the network, which may introduce latency.

Actor communication

Actor interactions can only happen through contracts. Using the contracts, Actors can communicate with each other and also the clients can interact with Actors. Actors are responsible for implementing the interfaces which define these contracts. Using Service Fabric APIs, the client gets a proxy to an Actor via the same set of interfaces. Because this interface is used to invoke Actor methods asynchronously, every method on the interface must be task-returning.

Since any communication needs to take place over network, the communication interfaces should be 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. An Actor proxy requires Actor ID and application name to identify the Actor it should connect to. The proxy does not expose the actual Actor location, which is important because the location of the Actor instance can change from time to time for reasons such as cluster node failure. Another important thing to consider is that Actors are required to be idempotent since they may receive the same message from a client more than once. The following line of code creates a proxy that can communicate with an Actor:

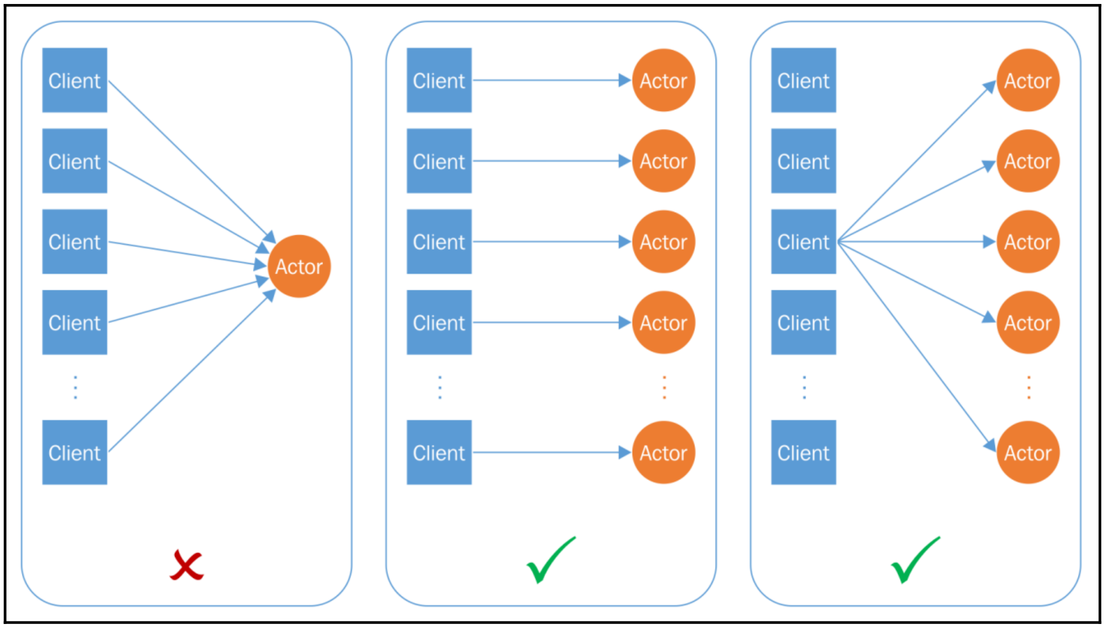
```
var proxy = ActorProxy.Create<IHelloWorldActor>(ActorId.CreateRandom(),
"fabric:/APPLICATION NAME");
```

Here, `IHelloWorldActor` is the contract that the Actor implements. The proxy is generating a random Actor ID to which it wants to communicate with, which is in the Service Fabric application that contains the Actor service hosting the Actor object.

Concurrency

Since Actors are single-threaded, they can process only one request at a time. This means that requests to the same Actor instance need to wait for the previous request to get processed. Turn based concurrency ensures that no other Actor methods or timer/reminder callbacks will be in progress until this callback completes execution.

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:



Actor Concurrency

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. However, reentrancy is a configurable property of Actors and you can disable this feature by decorating your Actor with the `Reentrant` attribute.

```
    [Reentrant(ReentrancyMode.Disallowed)]
```

Asynchronous drivers

Most of the Actor actions are driven as reaction to an input received from the client. However, Actors can update their own state at regular intervals through **timers** and **reminders**. Actors can also post updates to the clients on the progress of operations using **events**. Let's take a brief look at each of these attributes.

Timers

Reliable Actor timers are modelled on timers available in the system.Timers namespace. The timers are usually used to carry out routine operations in the lifetime of an Actor, for example processing input data at a certain rate. You can declare and register a timer in your application using the following code, usually in the `OnActivateAsync` method:

```
private IActorTimer _updateTimer;
protected override Task OnActivateAsync()
    {
        ...

         _updateTimer = RegisterTimer(
            CallBackMethod,                     // Callback method
            ObjectParameter,                   // 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();
    }
```

The `CallBack` method is simply a function that accepts the parameter that you passed while declaring the timer:

```
    private Task CallBackMethod(object state)
    {
        ...
        return Task.FromResult(true);
    }
```

Finally, you can unregister a timer using the `UnregisterTimer` method:

```
protected override Task OnDeactivateAsync()
    {
        if (_updateTimer != null)
        {
            UnregisterTimer(_updateTimer);
        }
        return base.OnDeactivateAsync();
    }
```

Because of the turn wise concurrency feature of Reliable Actors, no two concurrent executions of the callback method will take place at any time. The time will be stopped while the callback is executing and will be restarted when it has completed.

It is important to note that if the Actor doesn't receive external invocations, it will be deactivated and garbage collected after a period of time. When this happens, the timers won't be activated any more. Therefore, it is important that you register the timer again in the `OnActivateAsync` method when the Actor is reactivated.

Actor reminders

Just like Actor timers, Actor reminders are a triggering mechanism to invoke callbacks at periodic intervals. However, unlike Actor timers, Actor reminder callbacks are always triggered until they are explicitly unregistered. If an Actor has been deactivated, an Actor reminder callback will reactivate the Actor and invoke the registered callback. Actor reminders are generally used to carry out asynchronous processing such as data aggregation by applications. The syntax for registering reminders is very similar to that of timers:

```
IActorReminder reminderRegistration = await
  this.RegisterReminderAsync(
    reminderName,                   // Name of reminder

     BitConverter.GetBytes(payload), // Parameter

     TimeSpan.FromDays(3),           // Amount of time to delay
                                    before the callback is invoked

     TimeSpan.FromDays(1)            // Recurrence period
);
```

An Actor implementation that uses reminders needs to implement the `IRemindable` interface. This interface defines a single method named `ReceiveReminderAsync`. This method will be invoked for all the registered reminders, therefore, your application should be able to differentiate between reminders using either the name or the payload.

```
public Task ReceiveReminderAsync(string reminderName, byte[]
  context, TimeSpan dueTime, TimeSpan period)
{
    if (reminderName.Equals("Reminder Name"))
    {
        ...
}
  }
```

To unregister a reminder, if you have not persisted the reminder registration, then use the `GetReminder` method on the Actor base class to get a reference to the reminder registration, and then use the `UnregisterReminder` method to unregister the reminder:

```
IActorReminder reminder = GetReminder("Reminder Name");
Task reminderUnregistration = UnregisterReminderAsync(reminder);
```

Actor events

Actor events provide a mechanism for the application to send notifications to the clients using events. However, this Actor events do not guarantee reliable delivery and therefore if a guaranteed message delivery is desired then other notification mechanisms should be used. Due to its unreliable nature, this mechanism should only be used for Actor-client communication and never for Actor-Actor communication.

To use Actor events, you first need to define an event by extending it from the `IActorEvents` interface. This interface has no members and is only used by the runtime to identify events. All methods in the interface should return void and the parameters should be data contract serializable since they need to be sent to the client over the network:

```
    public interface IReminderActivatedEvent : IActorEvents
    {
        void ReminderActivated(string message);
    }
```

The Actor who publishes this event would need to implement the `IActorEventPublisher<T>` interface. This interface has no members and is only used by the runtime to identify the event publisher:

```
internal class HelloWorldActor : Actor,
IActorEventPublisher<IReminderActivatedEvent>
    {
...
}
```

On the client side, you need to declare an event handler, which is the component that will get invoked when an event is triggered. To define the handler, you simply need to implement the Actor event interface:

```
    public class ReminderHandler : IReminderActivatedEvent
    {
        public void ReminderActivated(string message)
        {
    ...
        }
    }
```

Finally, using the Actor proxy, the client can register the handler with the event:

```
var actorClientProxy = ActorProxy.Create<IHelloWorldActor>(
                new ActorId(userName),
                "fabric:/HelloWorldActorsApplication");
await actorClientProxy.SubscribeAsync<IReminderActivatedEvent>(new
ReminderHandler());
```

The Actor proxy is a smart component and it abstracts fault handling and name resolution from the client. In case of failure of Actor host node, which leads to Actor migrations to new nodes, the proxy will automatically subscribe to the event again.

To unsubscribe an event, the client can use the `UnsubscribeAsync` method on the Actor proxy. To trigger an event, an Actor needs to use the `GetEvent<T >` method to get the event reference and then trigger events by calling methods on the event interface:

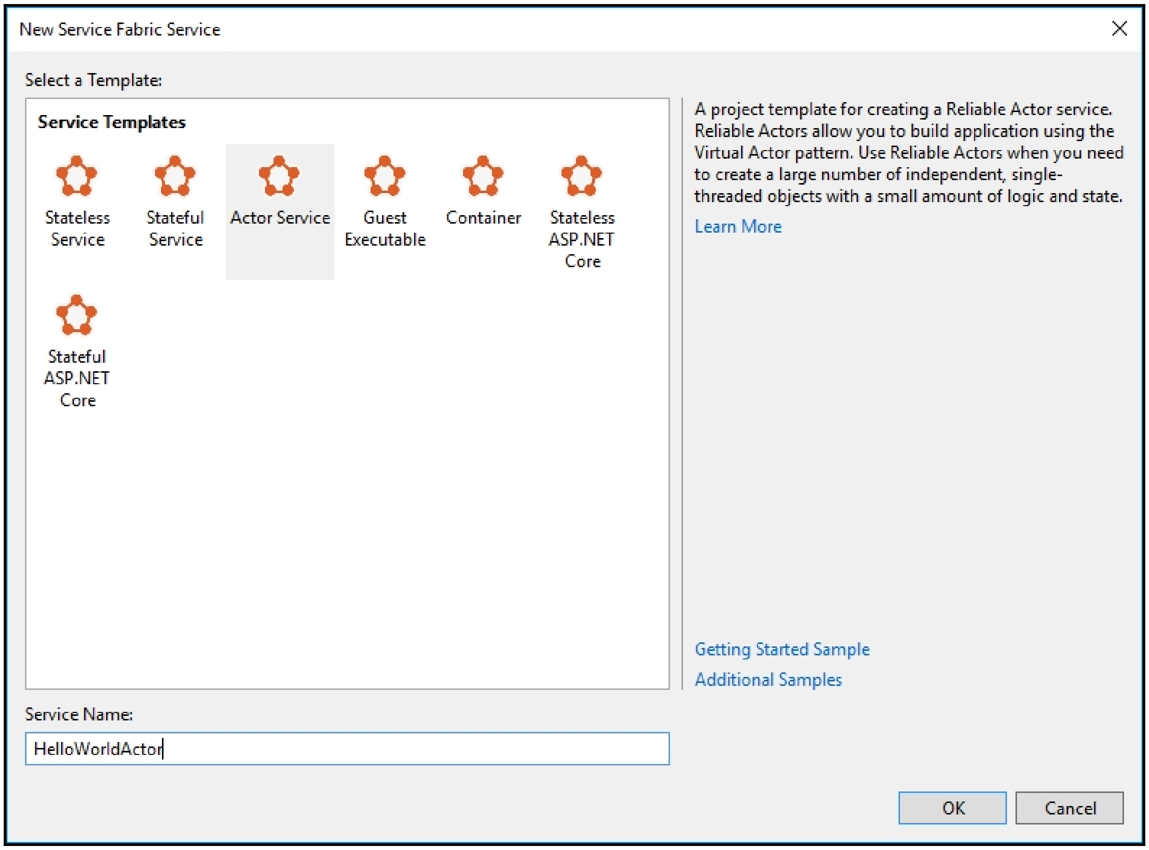
```
    var evt = this.GetEvent<IReminderActivatedEvent>();
    evt.ReminderActivated(reminderMessage.reminderMessage);
```

Your first Reliable Actors application

Now that we have understood all the carious concepts of a Reliable Actors Application in detail, let us cement our learning by building a sample that demonstrates all the key concepts that we have learned till now.

Create a new Service Fabric application named `HelloWorldActorsApplication` in Visual Studio and choose the Reliable Actors template.

You'd need to specify a name for the Actor service in the template dialog, let's name the service `HelloWorldActor`:


New Service Fabric Service

Select a Template:

Service Templates

A project template for creating a Reliable Actor service. Reliable Actors allow you to build application using the Virtual Actor pattern. Use Reliable Actors when you need to create a large number of independent, single-threaded objects with a small amount of logic and state.

Service Name: HelloWorldActor

Creating HelloWorldActor service

After the template finishes unfolding, you would find three projects loaded in your solution as follows:

- `HelloWorldActorsApplication`: This is the application project that packages all the applications together for deployment. This project contains the deployment PowerShell script and the manifest file – `ApplicationManifest.xml`, that contains the name of packages that need to be deployed on Service Fabric cluster among other settings.

- `HelloWorldActor.Interfaces`: This project contains the communication contract for the Actor application and the clients. The clients of an Actor application uses these contracts to communicate with the application and the application implements those contracts. Although, it is not a requirement of Service Fabric to create a separate assembly to store the contracts, designing them in such a manner is useful since this project might be shared between the application and the clients.
- `HelloWorldActor`: This is the Actor service project which defines the Service Fabric service that is going to host our Actor. This project contains implementation of Actor interfaces defined in the Actor interfaces project. Let's take a deeper look into the primary class of this project – `HelloWorldActor` in `HelloWorldActor.cs` file.

  This class derives from the Actor base class and implements the communication contract interfaces defined in the `HelloWorldActor.Interfaces` assembly. This class implements some of the Actor lifecycle events that we previously discussed. This class has a constructor that accepts an `ActorService` instance and an `ActorId` and passes them to the base Actor class.

```
    [StatePersistence(StatePersistence.Persisted)]
    internal class HelloWorldActor : Actor, IHelloWorldActor
    {
        public HelloWorldActor(ActorService actorService, ActorId
          actorId)
            : base(actorService, actorId)
        {
 }
...
}
```

When the Actor service gets activated, the `OnActivateAsync` is invoked. In the default template code, this method instantiates a state variable with count:

```
    protected override Task OnActivateAsync()
    {
        ActorEventSource.Current.ActorMessage(this, "Actor
          activated.");
        return this.StateManager.TryAddStateAsync("count", 0);
    }
```

Let's now navigate to the `Main` method in `Program.cs` file which is responsible for hosting the Actor application. Every Actor service must be associated with a service type in the Service Fabric runtime. The `ActorRuntime.RegisterActorAsync` method registers your Actor type with the Actor service so that the Actor service can run your Actor instances. If you want to host more than one Actor types in an application, then you can add more registrations with the `ActorRuntime` with the following statement:

```
ActorRuntime.RegisterActorAsync<AnotherActor>();
```

Let's modify this application to allow users to save reminders with a message and later notify the user with the message at the scheduled time with an event.

To begin, add a new contract in the Actor interface, `IHelloWorldActor` to set a reminder and retrieve the collection of reminders that have been set:

```
public interface IHelloWorldActor : IActor
{
    Task<List<(string reminderMessage, DateTime
    scheduledReminderTimeUtc)>>
      GetRemindersAsync(CancellationToken
    cancellationToken);

     Task SetReminderAsync(string reminderMessage, DateTime
    scheduledReminderTimeUtc, CancellationToken
     cancellationToken);
}
```

The `SetReminderAsync` method accepts a message that should be displayed at the schceduled time. We will store the message in the Actor state in the form of a list. The `GetRemindersAsync` method will return a list of all the reminders that have been set by a user.

Let's navigate to the `HelloWorldActor` class to implement the members of this interface. First let's implement the `SetReminderAsync` method:

```
public async Task SetReminderAsync(
    string reminderMessage,
    DateTime scheduledReminderTimeUtc,
    CancellationToken cancellationToken)
{
    var existingReminders = await this.StateManager
        .GetStateAsync<List<(string reminderMessage, DateTime
         scheduledReminderTimeUtc)>>(
            "reminders",
            cancellationToken);
    // Add another reminder.
```

```
            existingReminders.Add((reminderMessage,
             scheduledReminderTimeUtc));
        }
```

In this method, using the `StateManager`, we have first retrieved the reminders stored in the state. In the next statement, we added another reminder to the state.

The implementation of `GetReminderAsync` is very straightforward as well. In this method we simply retrive whatever is stored in the state and send it back as a response:

```
        public async Task<List<(
            string reminderMessage,
            DateTime scheduledReminderTimeUtc)>> GetRemindersAsync(
            CancellationToken cancellationToken)
        {
            return await
            this.StateManager.GetStateAsync<List<(string
            reminderMessage, DateTime scheduledReminderTimeUtc)>>
             ("reminders", cancellationToken);

        }
```

The state is initialized in the `OnActivateAsync` method:

```
        protected override Task OnActivateAsync()
        {
            ActorEventSource.Current.ActorMessage(this, "Actor
              activated.");
            return this.StateManager.TryAddStateAsync("reminders", new
               List<(string reminderMessage, DateTime
                    scheduledReminderTimeUtc)>());
        }
```

The application side logic is covered. Let's create a client for our application now. Add a new console application named `HelloWorldActor.Client` to the solution. For this project, make sure that you choose the same .Net framework as your Service Fabric application and set the platform to x64. Since the clients and the application need to share the communication contract, therefore add reference to the `HelloWorldActor.Interfaces` assembly in this project.

Now, let us create a proxy to the Actor objects and invoke the Actor methods using the proxy:

```
private static async Task MainAsync(string[] args, CancellationToken token)
        {
            Console.ForegroundColor = ConsoleColor.White;
            Console.WriteLine("Enter Your Name");
            var userName = Console.ReadLine();
            var actorClientProxy = ActorProxy.Create<IHelloWorldActor>(
                new ActorId(userName),
                "fabric:/HelloWorldActorsApplication");
            await actorClientProxy.SetReminderAsync(
                "Wake me up in 2 minutes.",
                DateTime.UtcNow + TimeSpan.FromMinutes(2),
                token);
            await actorClientProxy.SetReminderAsync(
                "Another reminder to wake up after a minute.",
                DateTime.UtcNow + TimeSpan.FromMinutes(3),
                token);
            Console.WriteLine("Here are your reminders");
            var reminders = await
                            actorClientProxy.GetRemindersAsync(token);
            foreach (var reminder in reminders)
            {
                Console.ForegroundColor = ConsoleColor.Cyan;
                Console.WriteLine($"Reminder at:
                 {reminder.scheduledReminderTimeUtc} with message:
                   {reminder.reminderMessage}");
            }
        }
```

For this example, we will use the name of user to create and identify the Actor object that we want to work with. This will ensure that all our operations activate the same Actor instance and perform operations with it. After we have accepted the name of the user, we will use the `Create` method of `ActorProxy` class to create a proxy to the desired Actor instance. We can later use this proxy to invoke the methods exposed through the Actor interface. The `ActorProxy` abstracts the process to locate an Actor in the cluster and abstracts failure handling and retry mechanism in case of cluster node failures.

Next, we have used the proxy object that we have created to invoke the various methods on the Actor object.

You can run the application and the client now to test the functionalities that we have built till now. Till now the application only stores the reminders in state, however doesn't notify the user at the scheduled time. Now, let us add reminders to the application that will get invoked at the scheduled time.

Head back to the `HelloWorldActor` class and implement the `IRemindable` interface. This interface has a single method `ReceiveReminderAsync` which gets invoked every time a reminder gets triggered. To add a reminder, add the following statement to the `SetReminderAsync` method that we defined earlier:

```
await this.RegisterReminderAsync(
                            $"{this.Id}:{Guid.NewGuid()}",
BitConverter.GetBytes(scheduledReminderTimeUtc.Ticks),
                            scheduledReminderTimeUtc - DateTime.UtcNow,
                            TimeSpan.FromMilliseconds(-1));
```

This statement will register a new reminder with the specified name, payload (which is the scheduled trigger time that we will use to identify the reminder), the scheduled occurrence time and the recurrence interval, which we have set to `-1` to indicate that we don't want the reminder to recur.

Now that we have added a reminder, we need a mechanism to talk back to the client. Actor events give us the capability to do so. However, this mechanism doesn't guarantee message delivery and therefore should be used with caution in enterprise applications where guaranteed delivery might be a requirement. Also, this mechanism is designed for Actor-client communications only and is not supposed to be used for Actor-Actor communications.

To use Actor events, we need to define a class that implements the `IActorEvents` interface. Let's add a class named `IReminderActivatedEvent` in the `HelloWorldActor.Interfaces` which implements the `IActorEvents` interface. This interface doesn't contain any members and is only used by runtime to identify events:

```
    public interface IReminderActivatedEvent : IActorEvents
    {
        void ReminderActivated(string message);
    }
```

Now that we have defined our event, we need to make sure that the runtime knows who is the publisher of the event. The runtime recognizes an event publisher using the `IActorEventPublisher` interface. This interface has no members that require implementation. Let's piggyback this interface on the `IHelloWorldActor` so that both the application and the client know that there are Actor events involved in the communication. The new `IHelloWorldActor` interface declaration should now look like the following:

```
public interface IHelloWorldActor : IActor,
IActorEventPublisher<IReminderActivatedEvent>
```

To complete the application side processing of events, let's implement the `ReceiveReminderAsync` method in the `HelloWorldActor` class which gets triggered every time a reminder is scheduled to trigger. In this function, we will try to extract the reminder from the state that got triggered, raise an Actor event, and finally unregister the reminder:

```
        public async Task ReceiveReminderAsync(string reminderName,
          byte[] state, TimeSpan dueTime, TimeSpan period)
        {
            var payloadTicks = BitConverter.ToInt64(state, 0);
            // Get the reminder from state.
            var cancellationToken = CancellationToken.None;
            var existingReminders = await this.StateManager
                .GetStateAsync<List<(string reminderMessage, DateTime
                  scheduledReminderTimeUtc)>>(
                    "reminders",
                    cancellationToken);
            var reminderMessage =
                existingReminders.FirstOrDefault(
                    reminder => reminder.scheduledReminderTimeUtc ==
                     new DateTime(payloadTicks));
            if (reminderMessage.reminderMessage != string.Empty)
            {
                // Trigger an event for the client.
                var evt = this.GetEvent<IReminderActivatedEvent>();
evt.ReminderActivated(reminderMessage.reminderMessage);
            }

            // Unregister the reminder.
            var thisReminder = this.GetReminder(reminderName);
            await this.UnregisterReminderAsync(thisReminder);
        }
```

We do not need to construct an object of `IReminderActivatedEvent`, the base class Actor provides a `GetEvent` method that does it for us. By invoking the member function of `IReminderActivatedEvent`, we raise an event that gets published to the client.

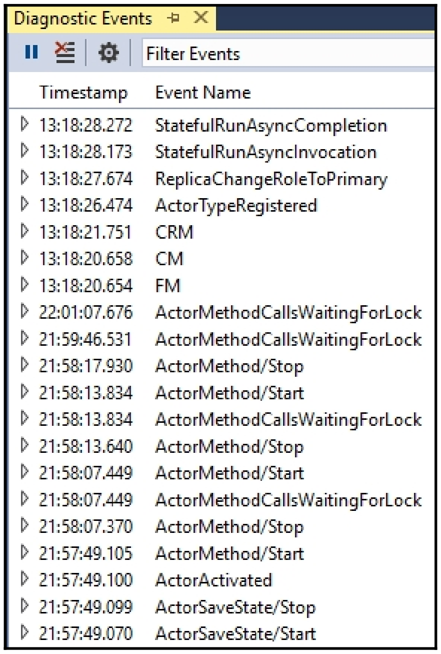
Now let's tie things together on the client side. In the client we need to build an event handler and finally listen to the Actor event that gets raised by the application. It's time to head back to the `HelloWorldActor.Client` application and add a new class in the project named `ReminderHandler`. This class needs to implement the Actor event `IReminderActivatedEvent` which will get invoked when the application raises this event. For this sample, we will simply redirect the message to console when the event is raised:

```
public class ReminderHandler : IReminderActivatedEvent
{
    public void ReminderActivated(string message)
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine($"Reminder Activated: {message}");
        Console.ResetColor();
    }
}
```

To start listening to the event revisit the `Main` method which contains the implementation of the client. In this method, we are going to subscribe to the `IReminderActivatedEvent` and provide it with an object of the event handler that we just defined. The revised `Main` method should look something like the following:

```
    private static async Task MainAsync(string[] args,
      CancellationToken token)
    {
        ...
        Console.ResetColor();
        await
          actorClientProxy.SubscribeAsync<IReminderActivatedEvent>
            (new ReminderHandler());
        ...
    }
```

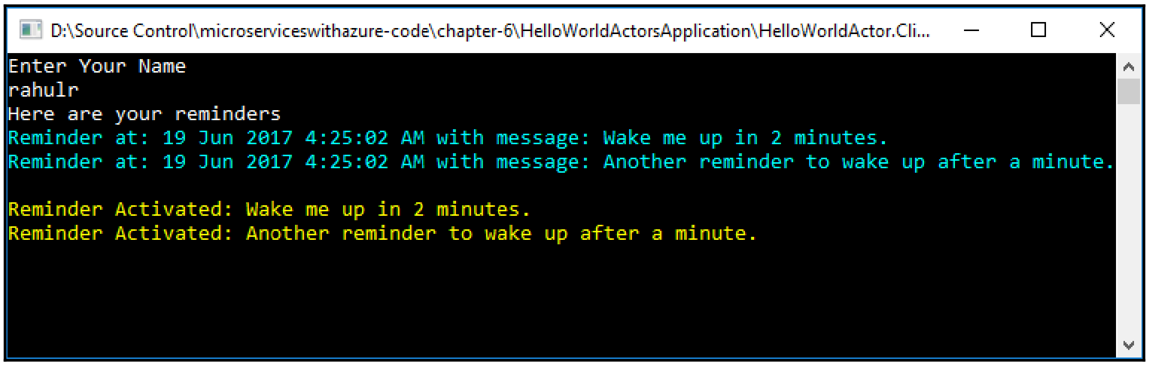
Let's start the application and the client now to see it in action. Once you start the application, you should be able to see messages appearing in the **Diagnostics Events** viewer console:



```
Diagnostic Events
Filter Events
Timestamp      Event Name
13:18:28.272   StatefulRunAsyncCompletion
13:18:28.173   StatefulRunAsyncInvocation
13:18:27.674   ReplicaChangeRoleToPrimary
13:18:26.474   ActorTypeRegistered
13:18:21.751   CRM
13:18:20.658   CM
13:18:20.654   FM
22:01:07.676   ActorMethodCallsWaitingForLock
21:59:46.531   ActorMethodCallsWaitingForLock
21:58:17.930   ActorMethod/Stop
21:58:13.834   ActorMethod/Start
21:58:13.834   ActorMethodCallsWaitingForLock
21:58:13.640   ActorMethod/Stop
21:58:07.449   ActorMethod/Start
21:58:07.449   ActorMethodCallsWaitingForLock
21:58:07.370   ActorMethod/Stop
21:57:49.105   ActorMethod/Start
21:57:49.100   ActorActivated
21:57:49.099   ActorSaveState/Stop
21:57:49.070   ActorSaveState/Start
```

Diagnostic event messages from HelloWorldActor appliction

Finally, when you fire off the client application, you should be able to see new events being registered and reminders being triggered by the application and handled by the client:



```
D:\Source Control\microserviceswithazure-code\chapter-6\HelloWorldActorsApplication\HelloWorldActor.Cli...
Enter Your Name
rahulr
Here are your reminders
Reminder at: 19 Jun 2017 4:25:02 AM with message: Wake me up in 2 minutes.
Reminder at: 19 Jun 2017 4:25:02 AM with message: Another reminder to wake up after a minute.

Reminder Activated: Wake me up in 2 minutes.
Reminder Activated: Another reminder to wake up after a minute.
```

HelloWorldActor client

You might have noticed that we never used the Actor ID while persisting state or while operating with it. That is because the state is local to Actor, which means that no two instances of our application that supply different user names will write data to same store. Another noticeable fact is that we never used thread locking mechanisms in our code. That is because reliable Actors are by nature single-threaded. Which means that you can't invoke multiple methods in the same Actor simultaneously, which avoids thread locking problems.

Summary

We started this chapter with an introduction to the Actor framework and the virtual Actor model that Service Fabric Reliable Actors use. Next, we discussed the lifetime of a Service Fabric Actor and Service Fabric Reliable Actor model helps an Actor application persist state and maintain distribution and failover.

We discussed how Actors communicate among themselves and with the clients and how concurrency and reentrancy governs this behavior. We also discussed asynchronous communication drivers which are timers, reminders, and events.

Finally, to tie all the concepts together, we built a sample application and realized the various aspects of reliable Actors that we discussed.

7
Microservices Architecture Patterns Motivation

Use of patterns in architecting software solutions is not new to the industry. Design patterns and architectural patterns have been popular among application developers since 1990.

An architectural pattern can be defined as a verified and reusable solution to a commonly occurring problem in a software architecture. These patterns have numerous benefits; the most important of these is the ability to solve a recurring problem in a technology agnostic way. This level of abstraction increases the reusability of these patterns across different frameworks and programming paradigms.

Although the definition looks simple, learning the art of leveraging the patterns to define a software solution takes time and effort. Proper understanding of when and where to apply a pattern is the key to a good solution architecture. One way to gain this proficiency is to practice these patterns by designing and developing solutions with them. A good architecture for a solution is often a combination of multiple architectural patterns.

Creating an architecture

The task of creating an architecture for an enterprise-grade system can be organized into three steps:

1. Defining the solution boundaries.
2. Creating the solution structure.
3. Creating the component design.

Let's understand these steps in detail.

Defining the solution boundaries

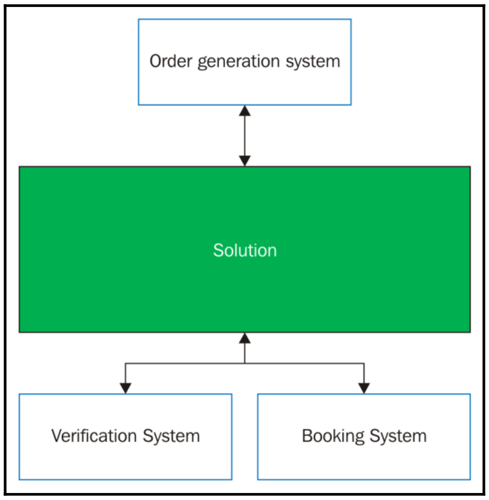
The first step for a solution architect is to understand the boundaries of the proposed solution. This is an important task as, with boundaries, the solution can be a moving target which is difficult to contain. Defining clear solution boundaries helps the delivery team focus on the right scope for the solution to be designed and developed.

The main input for this process of defining the boundaries of a solution would be the business requirements driving the solution. External dependencies such as external systems or processes which needs to be integrated with the solution also influences this process. These inputs help the architect define the context and scope for the solution. It also enables normalization of the solution by abstracting external work flows from the internal ones.

Let's take a simple example to understand this further. Consider the requirement to build a system for booking a movie ticket. Let's assume that the system is driven by a two step process:

1. Verify the availability of a seat.
2. Book the ticket.

Let's also assume that there are two separate services made available for us to consume the preceding mentioned actions. The solution boundaries of such a system would look like the ones illustrated as follows:



Solution boundaries

The box highlighted in green scopes the solution to be built and the other boxes are external systems which the solution is dependent on. This provides a clear understanding of the solution context to the application developers.

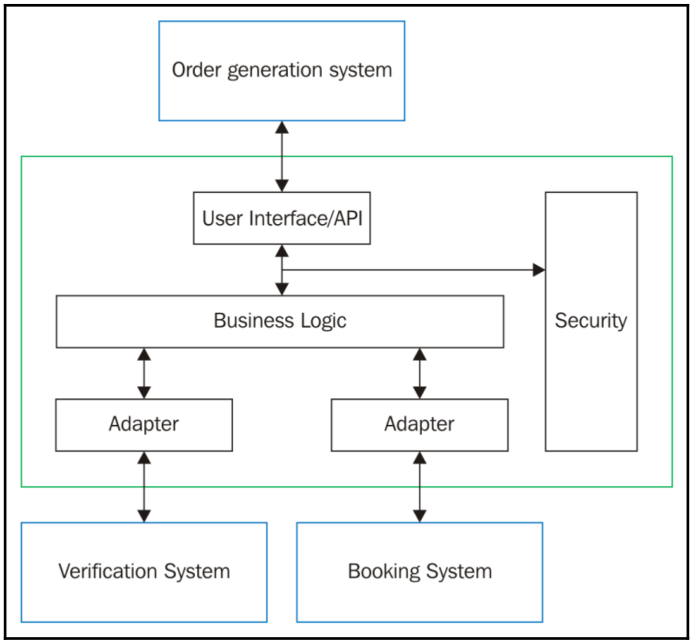
Creating the solution structure

After the solution boundaries are defined, the architect can now start working on the structure of the solution. A solution structure provides conceptual understanding of the solution space. The structure of the solution dictates the architectural framework to be used across the solution. The architectural framework is responsible for ensuring consistency throughout the solution there by making the solution more extensible and maintainable.

The key input for defining the structure of a solution is the system requirements. This will include both functional and non-functional requirements. The non-functional requirements will include requirements around security, performance, availability, reusability, and so on. The technologies used to develop the solution are largely influenced by their ability to deliver these requirements. Each non-functional requirement may force additional software components in the system. For instance, security requirements will require a security module for managing identity, authorization, and authentication of users accessing the system.

This step of the process is of most importance to us. During the definition of the solution structure, the solution architect can use architectural patterns to optimize and improve the solution architecture. We will discuss the common architectural problems surfaced for Microservice-based systems and the patterns which can be used to address these in detail in the next chapter, Microservices architectural patterns.

Let's use the previously described example and understand the solution structure for that system. Let's assume that there is a non-functional requirement to authenticate and authorize the users trying to book the ticket before performing the functionality. The following is a solution structure for this system:



Solution structure

The solution structure defines the components within the solution, each of which specializes in a specific task. The adapters connecting to the external systems act like a bridge for communication to the externally hosted services.

In terms of design patterns, an adapter is a component that allows two incompatible interfaces to work together. The adapter design pattern allows incompatible classes to work together by converting the interface of a class into another interface that the client expects. This eliminates friction between incompatible interfaces and allows them to work together. You can read more about adapters here: `https://en.wikipedia.org/wiki/Adapter_pattern`.

The user interface is used by the users to interact with the system and the core functionality is encapsulated within the component names as **Business Logic**. A separate component is introduced to handle the security requirements of the system. The user interfaces interact with the security module of authenticating and authorizing the user.

Component design

Once the solution structure is defined, every component can be designed in detail. The detailed design is mostly influenced by the functional requirements of the system and the relationship between the components which form the system. Tools such as UML can be used to identify the component relationships. This is where design patterns can be leveraged to improve the quality of the system. Design patterns are categorized into three types – structural, behavioral, and creational. Structural patterns are used to define the composition of objects and behavioral patterns are used to define the communications and algorithms within and across objects. Creational patterns are concerned with the instantiation of objects.

Classification of architectural patterns

Architectural patterns can be classified based on different properties. Componentization is a good example of one such property. Architectural patterns can be applied to monolithic systems or distributed systems. In this book, we will focus on the architectural patterns specific to Microservices-based distributed systems. Within this sub set we have classified the architectural patterns discussed in this book based on the problem type. This will ease the task for a solution architect to map a problem to an architectural pattern. The following are the architectural patterns we address through the patterns discussed in this book.

Optimization or non-functional patterns

Optimization or non-functional architectural patterns focus on enhancing the executional and evolutional quality of a system. The problems addressed by these patterns are oblivious of the system behavior or its functionality. Following are few problem categories addressed by these patterns:

- Resiliency
- Security
- Performance

- Scalability
- Availability

Operational patterns

Challenges in maintaining and managing the execution of a system needs to be thought through and accounted for while architecting a system. These common operational problems can be mapped to proven solutions with the help of architectural patterns. Following are the categories of major operational challenges:

- Orchestration
- Automation
- Centralization
- Deployment

Implementation or functional patterns

Implementation or functional patterns addresses common architectural problems related to a solution structure or framework used by an application. The solutions to these problems define standards for implementing all components in a system which makes these patterns very important. Following are a few challenges addressed by these patterns:

- Communication
- Abstraction
- Persistence

Picking up the right architecture pattern

You should follow a pragmatic approach for picking up the right patterns that address the problem that you are trying to solve. In the world of software development, each problem is unique and therefore building a universal pattern to address all scenarios is impossible. The process of selecting a pattern requires diligence from application architects and developers and should be done in a pragmatic manner. To select the right design pattern, you should first outline the following characteristics of the problem that you are trying to solve.

Context

A pattern documents a recurring problem-solution pairing within a given context. While selecting an architecture pattern for your application, you should be aware of the context of your application. A pattern that solves the messaging challenges of Facebook might not be a good fit for an application that caters to a small number of users. You need to resolve the applicability of a pattern by matching your problem with the associated scenario of the design pattern.

Forces

Forces describe what an effective solution for a problem must consider. An effective solution must balance the forces of a problem. A solution may not perfectly address all the forces. There might be consequences of implementing a solution that balances the forces, however, the benefits of implementation must outweigh the liabilities. For instance, for a system that makes multiple calls to another system on network, a force can be the need to improve application performance. You can solve this problem by batching the calls made by the system. This approach will introduce the liability of increased response time and storage space overhead but if configured precisely to balance response time, performance, and storage, it will adequately address the force without adding high liabilities.

You need to evaluate whether a pattern addresses the forces applicable to your problem, evaluate the benefits and liabilities of applying a pattern to your problem, and validate whether the liabilities arising out of application of a pattern are within acceptable limits of the requirements.

Complementing patterns

Most of the applications do not use a single pattern in isolation. For a pattern to make sense, it should list other patterns with which it competes or cooperates. We have listed related patterns for every pattern that you would be going through. We recommend that you consider the related patterns as well to compliment or compound the pattern under discussion.

Applying a pattern

Once you have selected a pattern to apply to your architecture, you should remodel it to fit your scenario. Due to unique nature of every application, you will need to apply diligence during the process of transforming a pattern to fit your scenario. Let's study the types of variations that you need to consider before applying a pattern to your scenario.

Structural variation

A pattern is not just an interconnection of components. The various components used in a pattern play a distinct role. These roles may be different from the requirements of your scenario. Let's consider the well-known observer design pattern, an observer may receive events from multiple sources or a single source. Also, the observer class may play the role of an observer in one relationship and the role of a subject in another. Therefore, you would need to remodel the observer pattern to fit in the scenario that you are trying to address.

The architecture patterns presented later in the book can be remodeled to fit the requirements of your scenario. You should consider how you can extend the pattern that you wish to apply to add value to your solution.

Behavioral variation

The desired behavior of your application may vary from the pattern. For example, your application may require messages to be processed in the order they are received, whereas the pattern may use concurrency optimization techniques to process messages without considering the message sequence. You should use the guidance of patterns to optimize the behavior of your application for time, space, and behavioral constraints within the application domains and runtime platforms.

Internal variation

The implementation of pattern and your application may vary. For example, a pattern uses *Service Fabric Concurrent Queues*, whereas your application may use *Azure Storage Queues*. Although, the objective of the pattern and your solution may concur, the implementation may vary. You should adapt the pattern to suit the needs of your application and not vice-versa.

Domain dependent variation

Depending on the desired functional, operational, and developmental requirements of your application, the implementation of a pattern may change. According to the principles of **domain-driven design** (**DDD**), within a bounded context a model can have a unique meaning. A pattern may exhibit such a behavior and there may be differences in the meaning of a term in the pattern and the context of the problem that you are trying to address. For instance, asynchrony may differ in meaning by context. For asynchronous communication using queues, your application may poll the queue at regular intervals or get triggered by a message-received event. You should remodel your chosen pattern to align the context of the pattern to the problem that you are addressing.

Summary

In this chapter, we walked through the steps of creating an architecture for a software solution. We then discussed the classification of architectural patterns and a few recommendations about picking the right architectural pattern for your application.

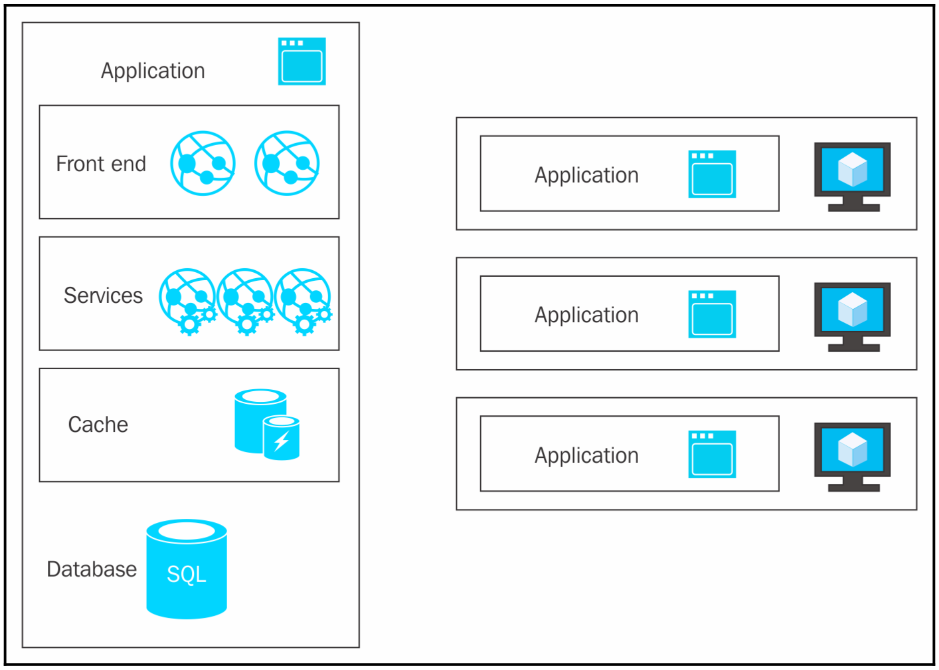
Towards the end of the chapter, we talked about methods of applying an architectural pattern on your application.

In the next chapter, we will dive deeper into common architecture patterns which can be applied on Microservices-based systems.

8
Microservices Architectural Patterns

The idea of componentizing software was first popularized in the world of computing in 1960s. Modern day enterprise distributed systems are build based on this principle of encapsulating functionalities as software components which are consumed by applications to assemble business work flows. These software components which provide functionality to other applications are popularly termed as services. The composite applications that use services can be anything ranging from a mobile application, a web application, or a desktop application.

In a traditional world, applications are monolithic by nature. A monolithic application is composed of many components grouped into multiple tiers bundled together into a single deployable unit. Each tier here can be developed using a specific technology and will have the ability to scale independently.



Monolithic application

Although a monolithic architecture logically simplifies the application, it introduces many challenges as the number of applications in your enterprise increases. Following are few issues with the monolithic design:

- **Scalability**: The unit of scale is scoped to a tier. It is not possible to scale components bundled within an application tier without scaling the whole tier. This introduces massive resource wastage resulting in increase in operational expense.
- **Reuse and maintenance**: The components within an application tier cannot be consumed outside the tier unless exposed as contracts. This forces development teams to replicate code which becomes very difficult to maintain.
- **Updates**: As the whole application is one unit of deployment, updating a component will require updating the whole application which may cause downtime thereby affecting the availability of the application.
- **Low deployment density**: The compute, storage and network requirements of an application, as a bundled deployable unit may not match the infrastructure capabilities of the hosting machine. This may lead to wastage of shortage of resources.

- **Decentralized management**: Due to the redundancy of components across applications, supporting, monitoring, and troubleshooting becomes expensive overheads.
- **Data store bottlenecks**: If there are multiple components accessing the data store, it becomes the single point of failure. This forces the data store to be highly available.
- **Cascading failure**: The hardware dependency of this architecture to ensure availability doesn't work well with cloud hosting platforms where designing for failure is a core principle.

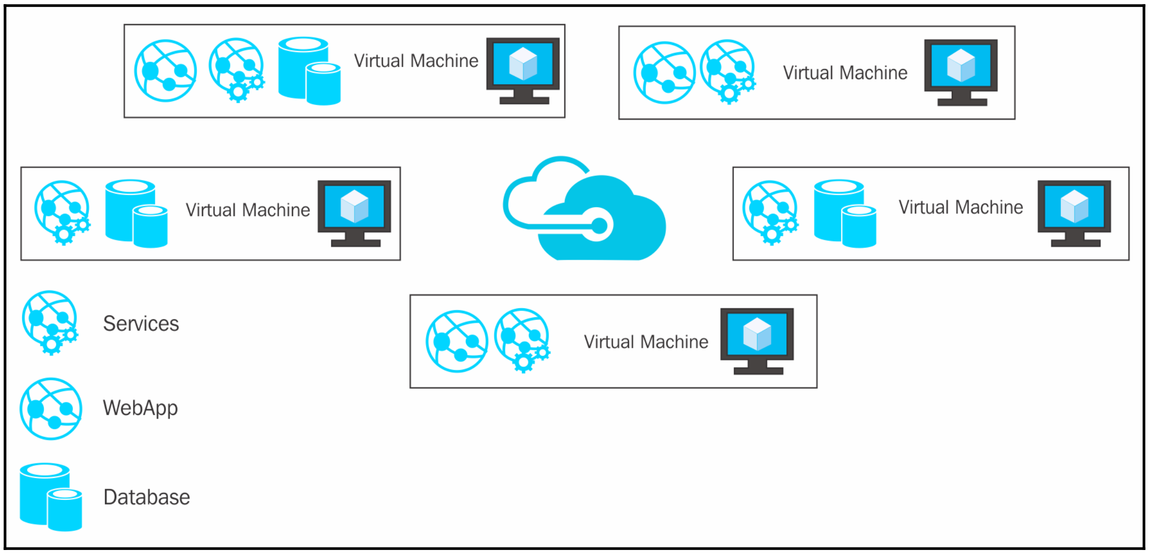
A solution to this problem is to migrate to a distributed system architecture. A distributed system is highly maintainable and far less complex than a monolithic application. There is a clear separation between the various parts of the application in a distributed system. This gives third party service providers an opportunity to plug their applications to the services that the distributed system provides.

A distributed system is generally realized through a **Service-Oriented Architecture** (SOA). The idea behind using SOA is that instead of using modules within an application, use services to provide functionality to the client application. An advantage of SOA architecture is that there can be multiple independent clients of an application. For instance, same set of services may be used by a mobile client and a web client of an e-commerce application. The SOA architecture also supports the independence of client and functionalities. If the contracts of a service do not change, the service may be modified to provide the functionality without altering the clients.

Due to lack of standards and implementation consistency on what functionalities should be provided by each service realized through SOA architecture, traditional architectures resulted in large monolithic services. Because of size of the services, the services became hard to scale. SOA also lacks simplicity as it is generally realized through web services. Traditionally, web services use SOAP and XML, which today have been replaced with REST and JSON. Since services in SOA can encapsulate several functionalities, each of the individual SOA service becomes deployment monoliths. SOA tends to reduce delivery agility and does not gel well with recent DevOps practices such as continuous integration and continuous deployment.

The Microservices architecture is the evolution of SOA. The Microservice architecture is a software architecture pattern where applications are composed of small, independent services which can be aggregated using communication channels to achieve and end-to-end business use case. The services are decoupled from one another in terms of the execution space in which they operate.

Each of these services will have the capability to be scaled, tested and maintained separately:



Microservices architecture

Microservices have all the features of SOA with additional service sizing constraints. Each Microservice has a single focus and it does just one thing and does it well. A distributed system composed of Microservices will have multiple Microservices that communicate with each other to provide a set of functionality for a specific part of the application.

Because the Microservices need to communicate with each other and with the clients, the communication mechanism needs to be quick and platform independent, for example, HTTP REST. A platform independent communication platform also ensures that Microservices and clients developed on different platform can communicate with each other for example a Java client communicating with .NET service.

Microservices are more agile than SOA. This helps development teams change part of systems quickly without affecting the rest of the system. Overall, added agility reduces the time to market of a product and allows the product to be adaptable to changing market conditions.

Microservices can also adapt to changing organizational systems. For instance, a change in account management product of a company would only alter the accounts service of the system without altering the other services such as the HR service or facility management service.

There are complex problems that need to be solved to host Microservices at scale. Such problems include preserving state, rolling upgrades, inter-service communication, and optimal use of machine resources. Such problems lead to slow adoption of Microservices architecture.

Azure Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage, scalable and reliable Microservices thereby addressing significant challenges in developing and managing Microservices. It does so by treating a collection of virtual machines as a worker pool on which Microservices can be deployed. Since, all that Service Fabric needs is a runtime, it can be executed on heterogeneous platforms on any data center.

Service Fabric is a reliable, tried and tested platform to build Microservices. It is already used to host some of the largest Microsoft services such as SQL Azure, Bing, Events hub, Cortana services, and so on.

Following are few advantages of a Service Fabric which makes it the ideal platform to build a Microservice based Application Platform:

- **Highly scalable**: Every service can be scaled without affecting other services. Service Fabric will support scaling based on VM scale sets which means that these services will have the ability to be auto-scale based on CPU consumption, memory usage, and so on.
- **Updates**: Services can be updated separately and different versions of a service can be co-deployed to support backward compatibility. Service Fabric also supports automatic rollback during updates to ensure consistency and stability of an application deployment.
- **State redundancy**: For stateful Microservices, the state can be stored alongside compute of a service. If there are multiple instances of a service running, the state will be replicated for every instance. Service Fabric takes care of replicating the state changes throughout the stores.
- **Centralized management**: The service can be centrally managed, monitored, and diagnosed outside application boundaries.

- **High density deployment**: Service Fabric supports high density deployment on a virtual machine cluster while ensuring even utilization of resources and even distribution of work load.
- **Automatic fault tolerance**: The cluster manager of Service Fabric ensures failover and resource balancing in case of a hardware failure. This ensures that your services are deigned for failure, a compulsory requirement of cloud ready applications.
- **Heterogeneous hosting platforms**: Service Fabric supports hosting your Microservices across public and private cloud environments. The Service Fabric cluster manager is capable of managing service deployments with instances spanning multiple data centers at a time. Apart from Windows operating system, Service Fabric also supports Linux as a host operating system for your Microservices.
- **Technology agnostic**: Services can be written in any programming language and deployed as executable or hosted within containers. Service Fabric also supports a native Java SDK for Java developers.

Architectural patterns

An application built with the Microservices architecture is composed of a set of small and independent services. Due to the granularity, diversity, high density, and scale of Microservices, a Microservices architecture is usually challenged by various problems related to optimization, operations, and implementation of these services. Most of these problems are best addressed with repeatable solutions or patterns which are tried and tested.

In the following sections, we will discuss few such architectural patterns which will help address challenges associated with a system build using Microservices. These patterns can speed up development process by providing tested, proven development paradigms. Each design pattern discussed here consists of the following sections:

- **Category**: The categories of challenges that the design pattern address
- **Problem**: The problem that the design pattern addresses
- **Solution**: How the design pattern solves the problem
- **Considerations**: Rationale that should be applied while applying the pattern in consideration
- **Related patterns**: Other patterns that are either used in conjunction with or are related to the pattern in consideration
- **Use Cases**: Typical scenarios where the pattern may be used

Let's start exploring these patterns one by one.

Each of the patterns discussed below belongs to a set of families, or categories, and is further classified by the problem areas it addresses, or the sub-categories. We have decorated each pattern with the following icons which will help you identify the appropriate categories and sub-categories to which the pattern belongs.

The following table summarizes the categories, which will be used across the chapter:

Category	**Logo**
Implementation	⬡
Operational	⬡
Optimization	⬡

The following table summarizes the sub-category, which will be used across the chapter:

Sub-category	**Logo**
Abstraction	⬡
Automation	⬡

Sub-category	**Logo**
Performance	⬡
Persistence	⬡

Availability	
Centralization	
Communication	
Data Storage	
Deployment	
Maintainability	
Orchestration	

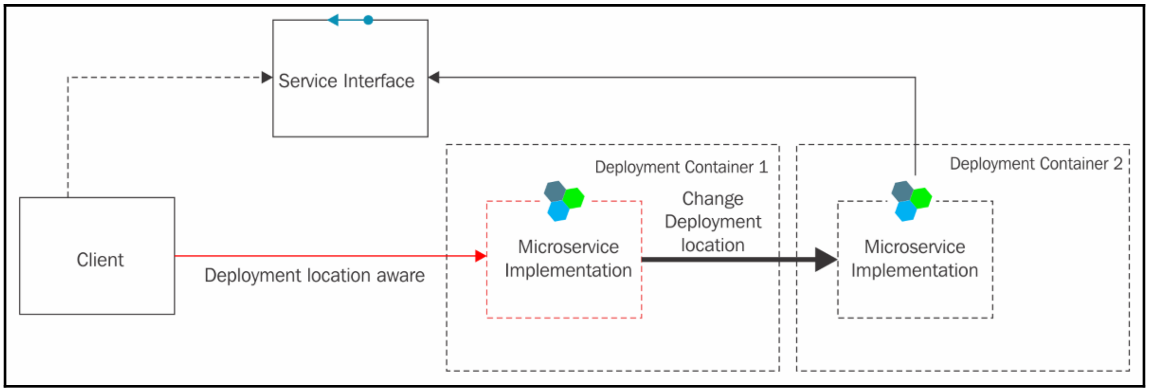
Reliability	
Resiliency	
Resource Utilization	
Reuse	
Scalability	
Security	
Transaction	

Service proxy

Problem

With Microservices hosted remotely, it is inefficient to instantiate a Microservice instance unless and until they are requested for by the client. During design/development time, this demands for a substitute object for the client to integrate with. Also, with cloud offering the ability to host multiple instances of Microservices across the globe, it is best that the deployment and implementation details are hidden from the client. Tightly coupled, deployment location aware communication can often result in service outages if the service deployment is relocated.

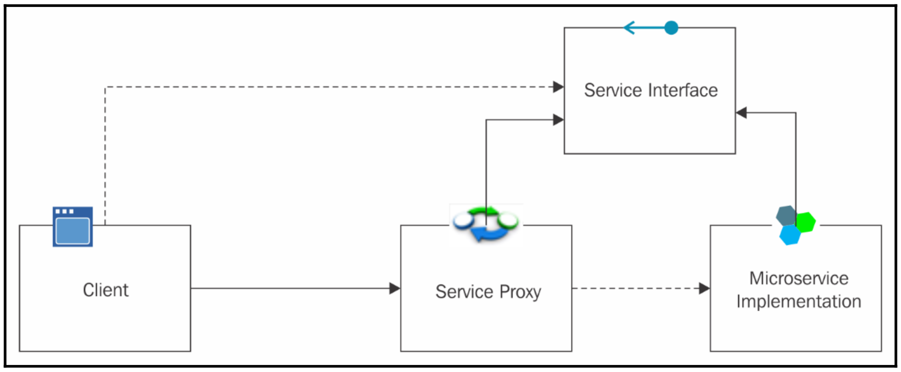
Following is an illustration of this scenario:


Service Interface
Client
Deployment location aware
Deployment Container 1
Microservice Implementation
Change Deployment location
Deployment Container 2
Microservice Implementation

Service proxy (Problem)

Solution

Service proxy is an implementation pattern which solves a communication problem. Service proxy helps in abstracting the service implementation from the interface definitions. This lets you change the service implementation without affecting the contracts with the clients. To enable communication with an Actor or a Microservice, a client will create a proxy object of the Actor that implements the Actor or service interface. All interactions with the Actor or service is performed by invoking methods on the proxy object. The Actor or service proxy can be used for *client-to-Actor/service* and *Actor/service-to-Actor/service* communication. The proxy pattern does not change any interfaces. This defines substitute objects for other objects. Following is an illustration of this pattern in action:



Service proxy (Solution)

Microsoft Azure Service Fabric exposes proxy classes for Reliable Actors and for Reliable Services. The proxy classes are also responsible for location resolution of Microservices.

Considerations

The following considerations should be taken into account while implementing this pattern:

- **Another layer of abstraction**: Adding layers to a system architecture results in some side effects such as increase in communication latency. The maintainability and testability of the code will also be impacted with introduction of additional classes.
- **Exception handling**: As the consuming application interacts with the proxy instead of the service directly, it must cater for scenarios where the backend Microservice is unavailable. The proxy will need to capture any service exceptions and relay it to the client.

- **Logging**: All exceptions and traces should be logged by the proxy to ensure better troubleshooting capabilities.
- **Abstraction**: Implementation or deployment details of the Microservice should be abstracted from the client.
- **Configurable**: The deployment location of the Microservice should be configurable and should have the ability to be updated without code re-compilation.

Related patterns

Following are the related design patterns which can be used in conjunction with this pattern:

- **Retry pattern**: Best used for transient fault handling which should be anticipated for cloud deployment of Microservices
- **Runtime reconfiguration pattern**: Making the configuration changes event-driven can help make the system more dynamic and ensure better availability
- **Gatekeeper pattern**: Best used to minimize the risk of clients gaining direct access to sensitive information and services
- **Circuit breaker pattern**: Handles transient faults to improve the stability and resiliency of an application

Use cases

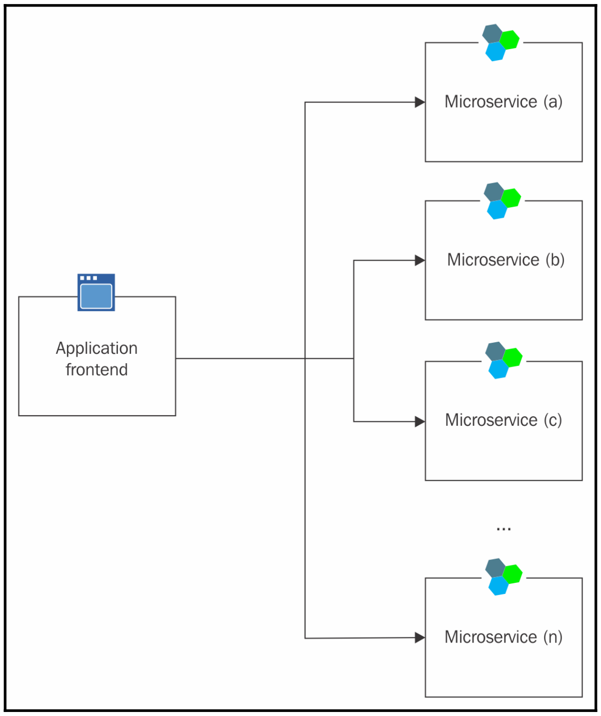
Following are a few use cases where this pattern will be a right fit:

- **Controlling the object lifecycle**: The instantiation of server side objects can be controlled within the proxy pattern thereby supporting scenarios like lazy instantiation, singleton, object pooling, and so on.
- **Security**: Proxy can act like another layer of security as it prevents the client from directly interacting with the Microservice.
- **Transformation**: Proxy can perform certain degree of transformations to abstract these complexities from the client.

Service Façade \ API Gateway

Problem

Microservices architecture recommends services to be decomposed as simpler units which can be knit together to achieve an end-to-end business use case. This results in client being cognizant of services at a granular level thereby increasing the complexity of management overheads related to communication, transformation, and transaction. Any change in the API contracts of the Microservices will also impact the client impacting the stability of the system. Also, with an increase in the number of services in the system, the complexity of client applications increases to accommodate more communication channels and service discovery overheads. The following diagram illustrates such a scenario:



Service Façade (Problem)

Solution

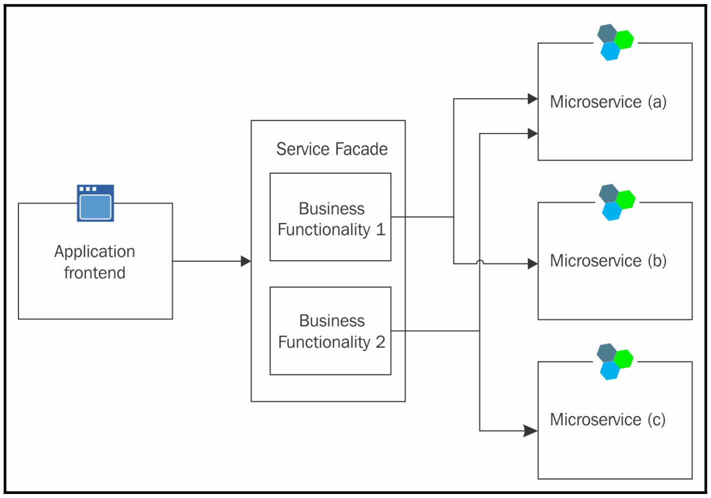
Microservices, forced by the very idea behind the concept, recommend granularization of services to improve reuse and to enforce single responsibility principle. While this improves maintainability, it also introduces the requirement of having an abstraction layer which can aggregate multiple Microservices to expose a business functionality which can be easily consumed by an application. Services Façade will play this role of this abstract layer which a client can integrate with, without knowing the complexity of invoking one or more Microservices behind the scene.

As defined by Gang of Four, the role of the façade pattern is to provide different high-level views of subsystems whose details are abstracted from the client. The subsystems in our case would be specialized Microservices deployed over heterogeneous environments.

Design Patterns: Elements of Reusable Object-Oriented Software is a considered to be an authoritative book describing software design patterns. The book's authors are Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. The book discusses 23 classic software design patterns which are regarded as an important source for object-oriented design theory and practice. The authors of the book are often referred to as the **Gang of Four** (**GoF**).

Service Façade can be visualized as an orchestration layer responsible for integrating one or more Microservices to achieve a specific business use case. The Service Façade either routes the request to a service or fans out the request to multiple services. The business logic of the system is encapsulated in the Service Façade and is thus abstracted from the client. The Façade itself can be a Microservice participating in the application.

The following diagram illustrates this pattern in action:



Service Façade (Solution)

Microsoft Azure API Management is a software as a service which enables orchestration, transformation and documentation of services. The service also provides other functionalities such as metering and analytics. API Management is a potential service to be considered to play the role of a Services Façade for Microservices based applications.

Considerations

The following considerations should be taken into account while implementing this pattern:

- **Cyclomatic complexity**: Cyclomatic complexity is a measure of complexity of a program. This metric measures independent paths through a program source code. An independent path is a path that has not been traversed before in a code execution flow. It is important to ensure that the cyclomatic complexity of the Service Façade is low. This will guarantee maintainability and readability of the code.
- **Transactions**: While integrating Microservices, it's important to ensure transactional integrity. If a business use case terminates before completion, compensatory transactions need to be executed by the Service Façade.
- **Another layer of abstraction**: Adding layers to a system architecture results in some side effects such as increase in latency. The maintainability and testability of the code will also be impacted with introduction of additional components.

- **Exception handling**: As the consuming application interacts with the façade instead of the service directly, it must cater for scenarios where the backend Microservice is unavailable. The Service Façade will need to capture any service exceptions and relay it to the client.
- **Logging**: All exceptions and traces should be logged in the same central location as the application by the Service Façade to ensure better trouble shooting capabilities.
- **Abstraction**: Implementation or deployment details of the Microservice should be abstracted from the client.
- **Configurable**: The deployment location of the Microservice should be configurable and should have the ability to be updated without code re-compilation.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Compensating Transaction pattern**: Used for maintaining transactional integrity of a system by handling workflow failures gracefully
- **Retry pattern**: Best used for transient fault handling which should be anticipated for cloud deployment of Microservices
- **Gatekeeper pattern**: Best used to minimize the risk of clients gaining direct access to sensitive information and services
- **Circuit Breaker pattern**: Handles transient faults to improve the stability and resiliency of an application

Use cases

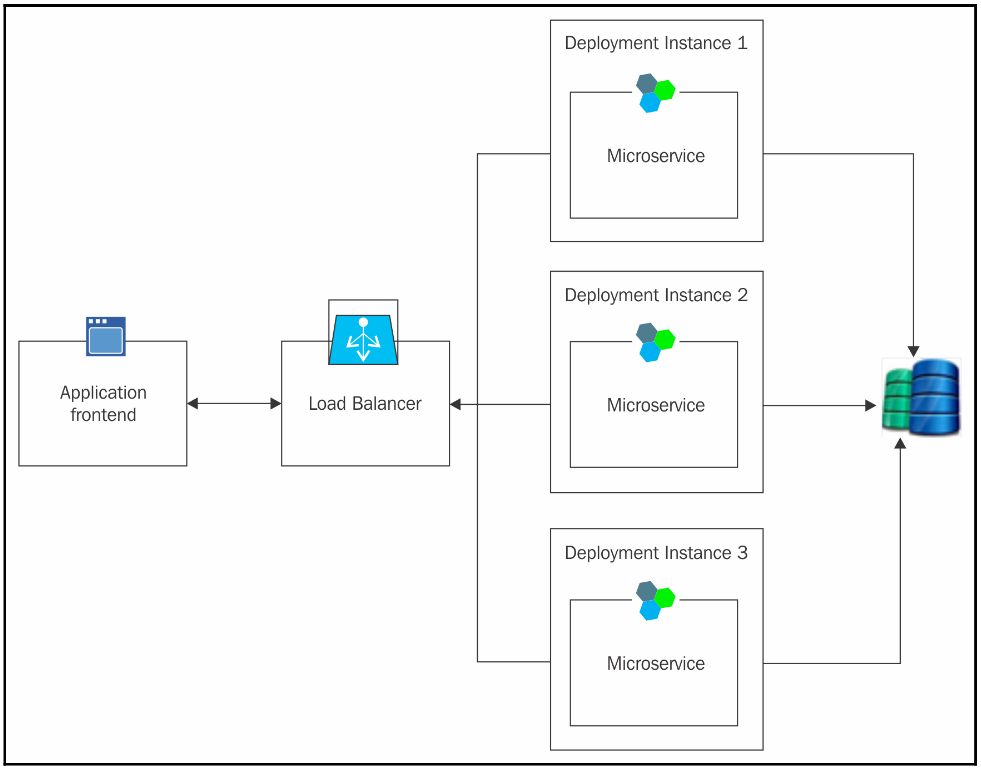
Following are a few use cases where this pattern will be a right fit:

- **Service orchestration**: An ideal use case for this pattern would be to act like an orchestration layer to abstract out the complexity around integration of Microservices such as communication, transaction, deployment details, and so on
- **Security**: Service Façade can act like another layer of security as it prevents the client from directly interacting with Microservices
- **Fault tolerance**: Service Façade can be enriched by implementing a circuit breaker pattern within it to better tolerate transient faults

Reliable Data Store

Problem

Stateful Microservices are commonly associated with dependencies on external data stores. These dependencies often introduce bottle necks in terms of performance and availability as the data source can act like a single point of failure. This problem has enhanced impact where multiple instances of a service co-exists at a given point. Maintaining high availability of the data store becomes expensive as the data grows. The following diagram illustrates a system with a centrally externalized data store:



Reliable data store (Problem)

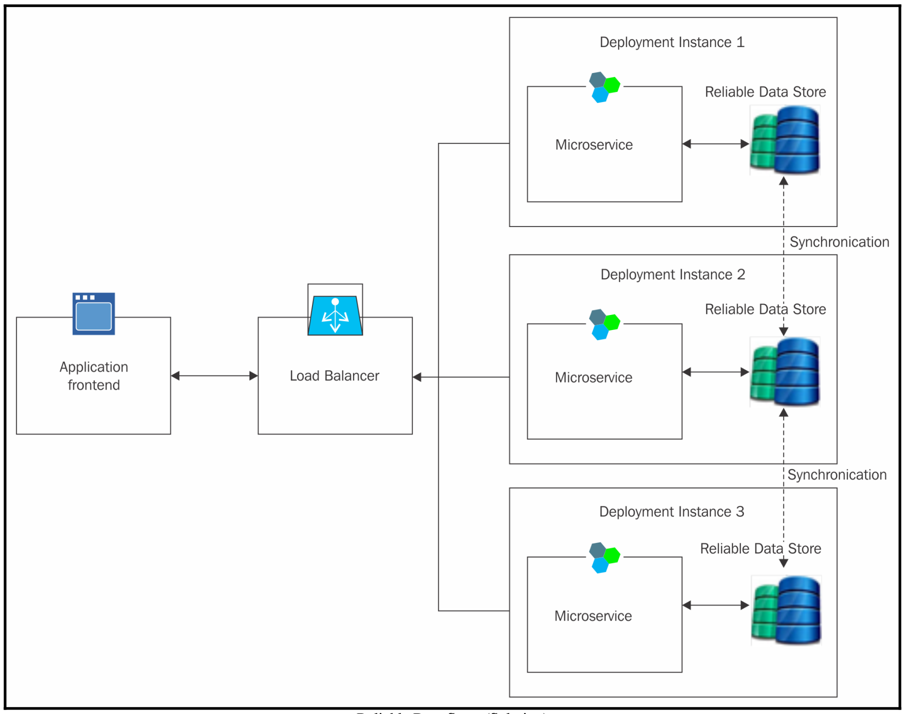
Solution

A solution to this problem is to co-locate Microservice state with compute. This will reduce application's dependency on external data stores (such as cache) for frequently accessed data. Having the data stored on the same deployment instance as the consuming application also contributes positively to the system performance and negates the risk introduced by the data store being a single point of failure.

However, this introduces challenges around maintaining data consistency and synchronization across multiple instances of the same service. A reliable data store with the following characteristics can be used to persist state to address these problems:

- **Replicated**: Data changes can be replicated across multiple instances
- **Persistence**: Data is persisted on a secondary storage to ensure durability
- **Asynchronous**: Supports asynchronous API model to ensure non-blocking calls
- **Transactional**: Should support transactions

The following diagram illustrates a system using Reliable Data Store instead of a centralized data store:



Reliable Data Store (Solution)

Microsoft Azure Service Fabric recommends the use of Reliable Collections for persisting data across instances of a Microservice. Reliable Collection is a natural evolution of the System.Collections library with the added ability to abstract out the complexities of replication, persistence, synchronization, and transaction management from the client. It also offers different isolation levels (repeatable read and snapshot) for accommodating diverse transactional systems.

Considerations

The following considerations should be taken into account while implementing this pattern:

- **Isolation levels**: While using this pattern, the isolation levels suited for the application need to be chosen based on characteristics of transactions. More about isolation levels supported by Reliable Collections can be found at: `https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-reliable-collections-transactions-locks`.
- **Persistence**: The persistence model required by the reliable data sore needs to be accounted for, before implementing this pattern. The frequency of write, replication, batching I/O operations, and so on are few things to consider while choosing a persistence model.
- **Timeouts**: It's important to introduce timeouts for every call associated with a reliable data store to avoid dead locks.
- **High availability**: To ensure high availability, consider running multiple instances of your Microservice at a time. Microsoft recommends at least three instances of a service to be active at any time.
- **Synchronization delay**: Where there are multiple instances of Microservices deployed, the data store on one of the instances will act as a primary and others would be considered as secondary. In case of an update, there will be a delay in reflecting changes across secondary instances of the data store. This may lead to chances of read operations retrieving stale data.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Retry pattern**: Can be used for handling transient faults on the Reliable Data Store.
- **Sharding pattern**: The data can be partitioned horizontally to improve scalability and performance. The pattern is advisable while handling large volumes of equally disperse data sets.
- **Leader Election pattern**: In case of multi-instance deployment, elect a leader to coordinate tasks to avoid conflicts around shared resources.

Use cases

Following are few use cases where this pattern will be a right fit:

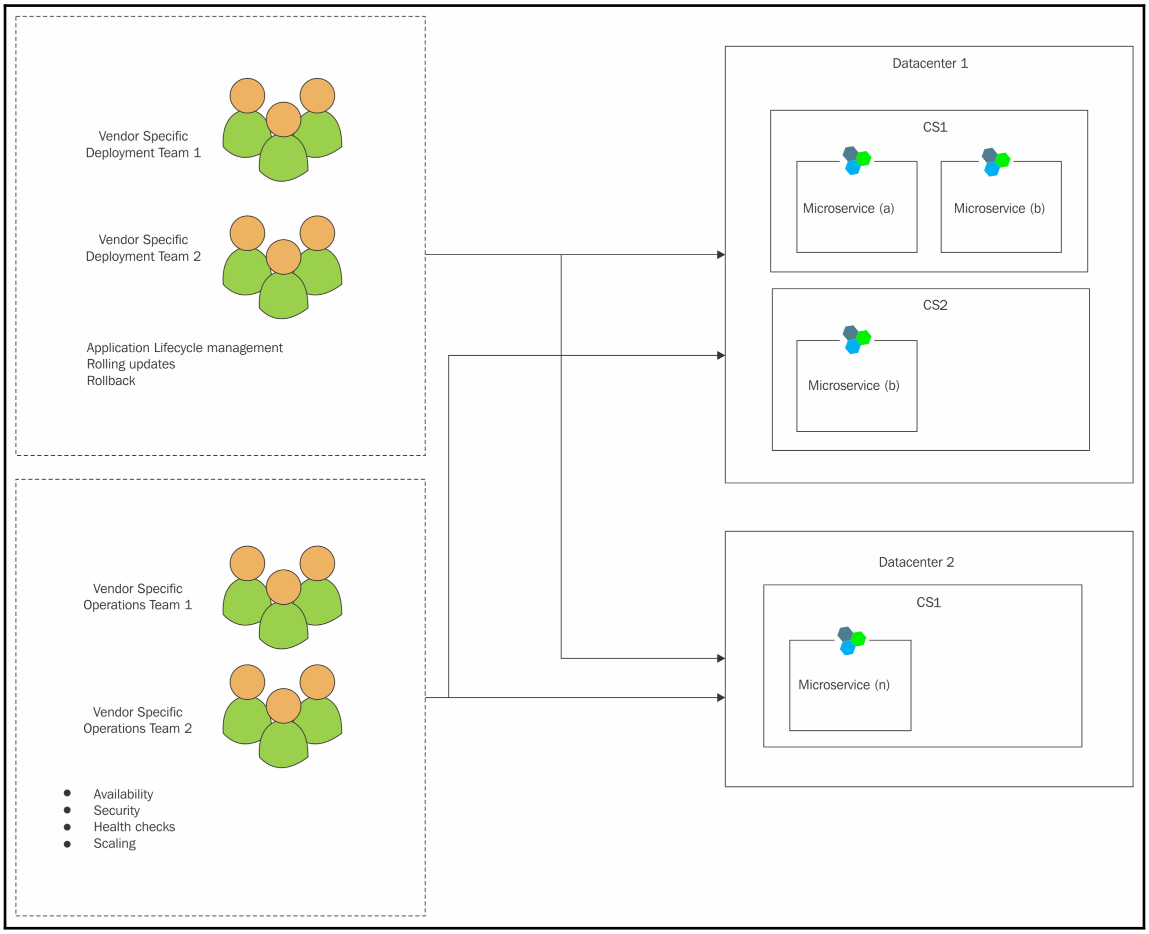
- **Caching**: An ideal use case for this pattern would be application scenarios which require data to be cached. A Reliable Store will offer better performance compared to an external cache while ensuring a higher degree of availability.
- **Fault tolerance**: As the data stores are synchronized and persisted on every node, the system become more tolerant against hardware failures.

Cluster Orchestrator

Problem

Deploying a highly scalable and available set of Microservices on commodity hardware introduces the requirement having an efficient management system. Managing communication between services, monitoring health, maintaining a service catalog, managing application lifecycle, load balancing, scaling instances, data replication, handling failover, managing rolling updates, and so on are few challenges associated with operating enterprise scale Microservices.

It is nearly impossible to efficiently perform these tasks manually. The following diagram illustrates how complex manually managing such a system can become:


Vendor Specific Deployment Team 1
Vendor Specific Deployment Team 2
Application Lifecycle management
Rolling updates
Rollback
Vendor Specific Operations Team 1
Vendor Specific Operations Team 2
Availability
Security
Health checks
Scaling
Datacenter 1
CS1
Microservice (a)
Microservice (b)
CS2
Microservice (b)
Datacenter 2
CS1
Microservice (n)

Cluster orchestrator (Problem)

Solution

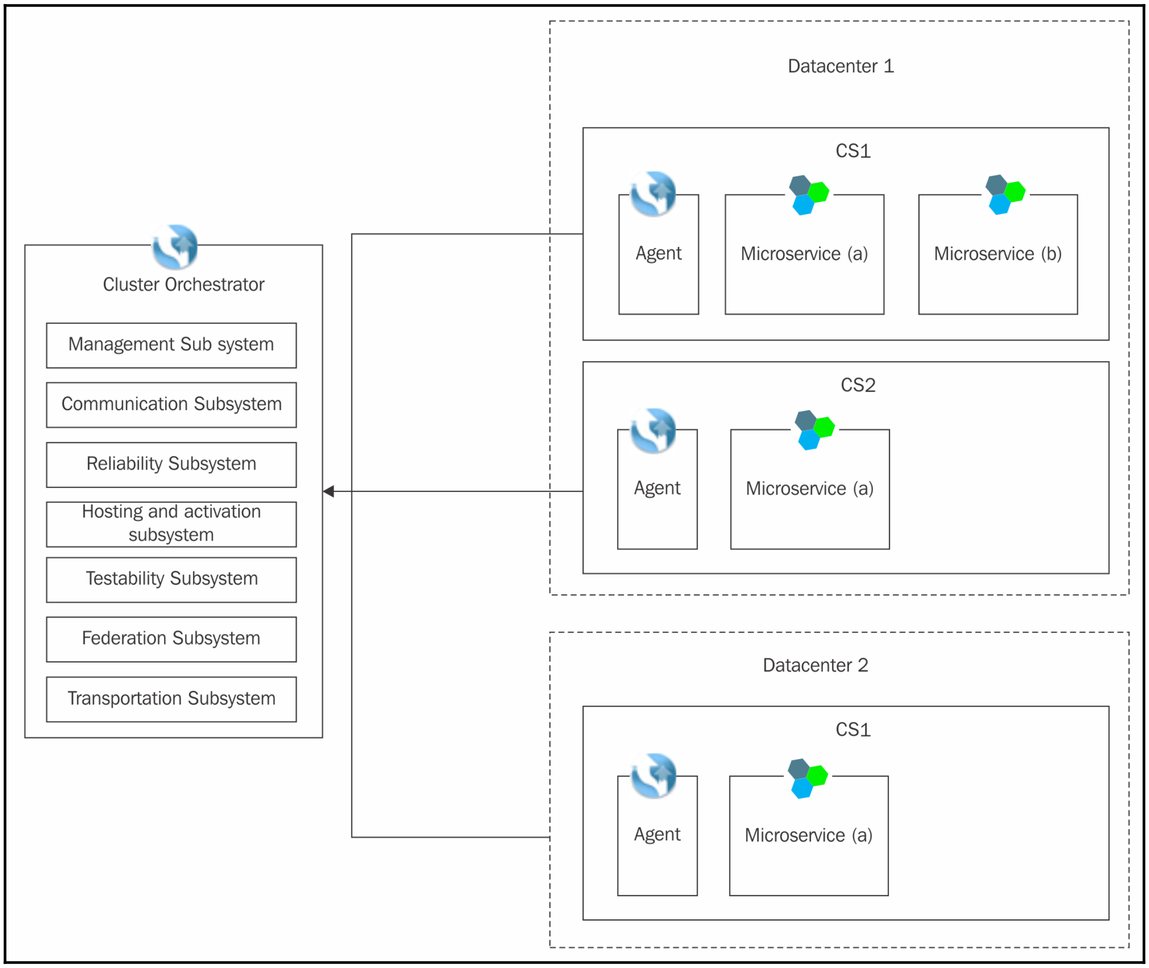
A highly available, cluster orchestration system can be employed to manage Microservice deployments across heterogeneous environments. Cluster orchestrator will own the responsibility of ensuring the availability of Microservices with minimal human intervention. The system will consist of two components:

- **Cluster Orchestration Server(s)**: Centralized management server
- **Cluster Orchestration Agents**: A thin native software deployed on every instance (virtual machine) in a cluster dedicated for hosting the Microservices.

Cluster orchestrator deploys an agent on every hosting instance (virtual machine) in a cluster which is the used to control and communicate with the host operating system and the services deployed on the machine. Once a connection between the agent and the server is established, cluster orchestrator handles the following responsibilities associated with managing the Microservices:

- Availability
- Data replication
- Data partitioning
- Load balancing
- Rolling updates and rollbacks
- Resource placement (high density deployment)
- Failover and recovery
- Health monitoring and healing

The following diagram illustrates a cluster orchestrator in action:



Cluster orchestrator (Solution)

Microsoft Azure Service Fabric has inbuilt orchestration capabilities capable of handling hyper scale Microservice deployments across heterogeneous environments. More about Service Fabric cluster orchestration capabilities can be found here: `https://azure.microsoft.com/en-us/documentation/articles/service-fabric-architecture/`.

Considerations

The following considerations should be taken into account while implementing this pattern:

- **High availability**: It is critical to ensure high availability of the cluster orchestrator as it can act like a single point of failure for your Microservices deployment. A three to five instance cluster is usually recommended for the orchestration software to handle availability.
- **Heterogeneity**: The ability of the orchestration tool to handle virtual machines of different configurations and operating environments is the key to support heterogeneity.
- **Resource allocation**: The orchestration tool should have an efficient resource allocation algorithm to manage high density deployments while optimizing the resource utilization.
- **Security**: The orchestration tool should be responsible to ensure isolation of services when performing high density deployments.
- **Configurable**: Thresholds for rollbacks, update domains, failover, and so on should be configurable to suit specific requirements of the ecosystem.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Compute Resource Consolidation Pattern**: This pattern is ideal to support high density deployments.
- **Health Endpoint Monitoring Pattern**: Employing agents (external processes) to verify that applications and services are performing correctly.
- **Leader Election Pattern**: In case of multi-instance deployment, elect a leader to coordinate tasks to avoid conflicts around shared resources. This is useful in managing a cluster of orchestration servers to ensure high availability.

Use cases

Following are few use cases where this pattern will be a right fit:

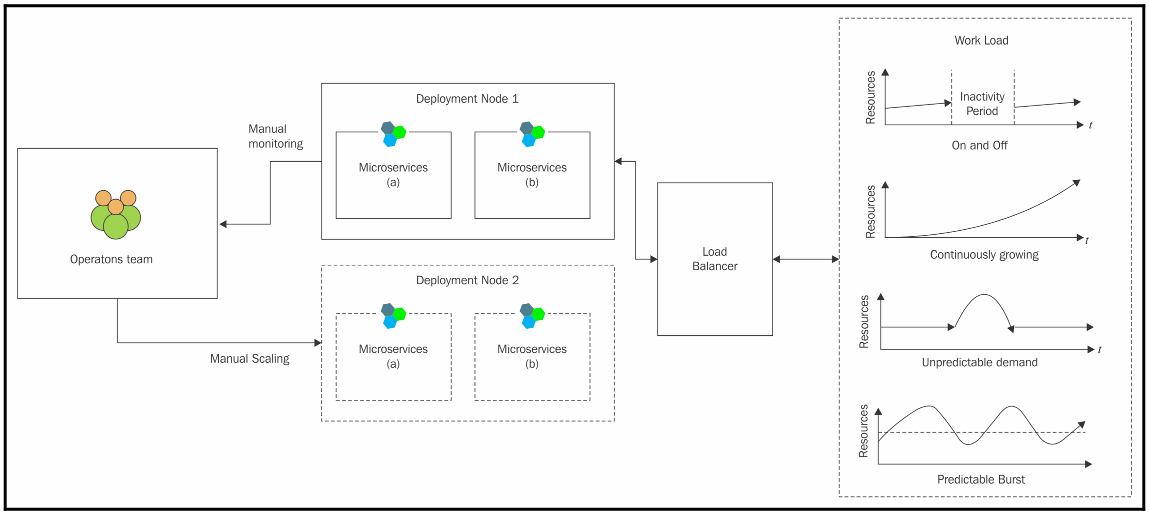
- **Hyper scale deployment of Microservices**: An ideal use case for this pattern would be environments which demand high density. Highly available, hyper scale deployments of Microservices.
- **Automated operations**: Cluster orchestrator minimizes the requirement of manual intervention for managing your Microservices by offering features like self-healing, rolling updates, rollback, and so on.
- **Cross platform deployments**: This pattern is best used to manage Microservice deployments across heterogeneous platforms (like different data centers, different operating systems, different hardware infrastructure, and so on).

Auto-scaling

Problem

The load on the services tier of enterprise system is typically nonlinear. Matching the infrastructure services to the workload is a key challenge faced by operation teams these days. The complexity starts with understanding the pattern around increase/decrease in the workload, mapping the workload with a scale-out / scale-up plan and then executing the plan manually. The challenges multiply if the workload keeps changing frequently altering the scalability requirements dynamically.

The following diagram illustrates a scenario where the operations team manually manages the scaling of services:



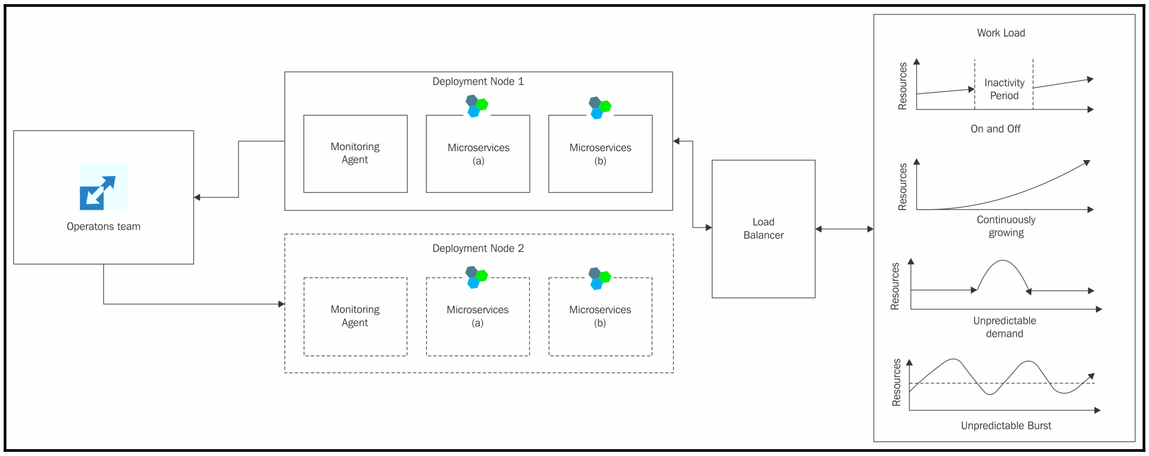
Auto-scaling (Problem)

Solution

The solution is to employ a software system to perform scaling automatically based on parameters such as number of requests, CPU utilization, memory utilization, and buffer size (queue size), and so on to improve the efficiency of the system and to optimize operational expense.

Auto-scaling is an elastic process which will provision resources based on the work load and a preconfigured collection of rules specifying the thresholds, scaling range, scaling factors, and so on. The system should be capable of scaling the resources vertically, by redeploying the service on a more capable hardware or, scaling horizontally, by provisioning additional instances of the services.

The following diagram illustrates an auto-scaling system in action:



Auto-scaling (Solution)

Azure Service Fabric clusters are built on top of virtual machine scale sets and can be scaled either manually or with auto-scale rules. You can read more about building auto-scaling rules at: `https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-programmatic-scaling`.

Considerations

The following considerations should be taken into account while implementing this pattern:

- **System downtime**: Auto-scaling should be achieved without system downtime. This forces the requirement that existing instances hosting the service should not be impacted by the scaling operation.
- **Scale up or scale out**: Depending on the execution model of the application being hosted, the decision whether to scale up/down or scale out/in has to be made.
- **Start-up / shut-down threshold**: Starting up a new node or shutting down a node will take some time. This time needs to be accounted for before a decision on scaling is made. A good input for this decision is consistency of the load which can be evaluated against a threshold before scaling the system.

- **Max and min count for instances**: It is important to set bounds for auto-scaling to ensure that any erroneous/malicious activities are not impacting the availability or the operation expense of the system. Alerts should be configured to monitor any unusual scaling activities.
- **Logging and monitoring**: The scaling activity should be logged and the performance of the system should be monitored post scaling to ensure that the scaling operation was effective.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Throttling pattern**: This pattern can be used to handle graceful degradation of services when the load increases. Throttling can be used with auto-scaling pattern to ensure system availability while scaling out.
- **Health Endpoint Monitoring pattern**: Employing agents (external processes) to verify the load on the system to trigger the auto-scaling process.
- **Competing Consumers pattern**: This pattern helps scale out by enabling multiple concurrent consumers to process messages received on the same messaging channel.

Use cases

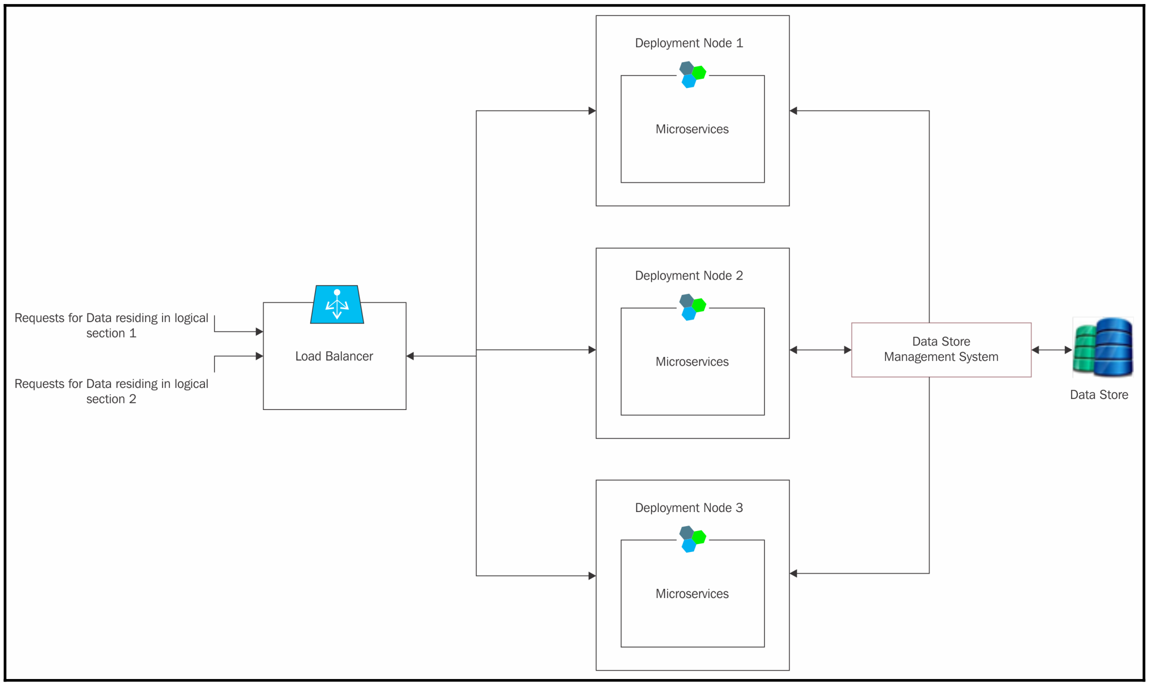
Following are few use cases where this pattern will be a right fit:

- **On and Off workload**: Scenarios where the service needs to be running only for a specific amount of time. Batch jobs and scheduled jobs are apt examples.
- **Growing fast**: When the demand for the service increases continuously. A new production release of a popular service would be an example for this scenario.
- **Unpredicted demand**: Unpredicted increase or decrease in workload demanding a scaling out/in. The load encountered on news site or stock exchanges can be classified in to this category.
- **Predictable burst**: Anticipated increase in load for an indefinite duration. Increase of load on movie ticking systems on a new release would be a perfect example.

Partitioning

Problem

Operational efficiency of a stateful Microservice is highly dependent on the way it stores and retrieves data as part of its operations. As the data store grows, time to access the data increases thereby causing significant impact on the performance, throughput, and scalability of a system. It is a commonly observed pattern that some sections of the data store are accessed more frequently compared to others. In a monolithic architecture, it is very difficult to scale the data store and the compute used to access the data separately for specific segments of the data store. The following figure illustrates such a scenario where the data store becomes a bottleneck:


Deployment Node 1
Microservices
Deployment Node 2
Microservices
Deployment Node 3
Microservices
Requests for Data residing in logical section 1
Requests for Data residing in logical section 2
Load Balancer
Data Store Management System
Data Store

Partitioning (Problem)

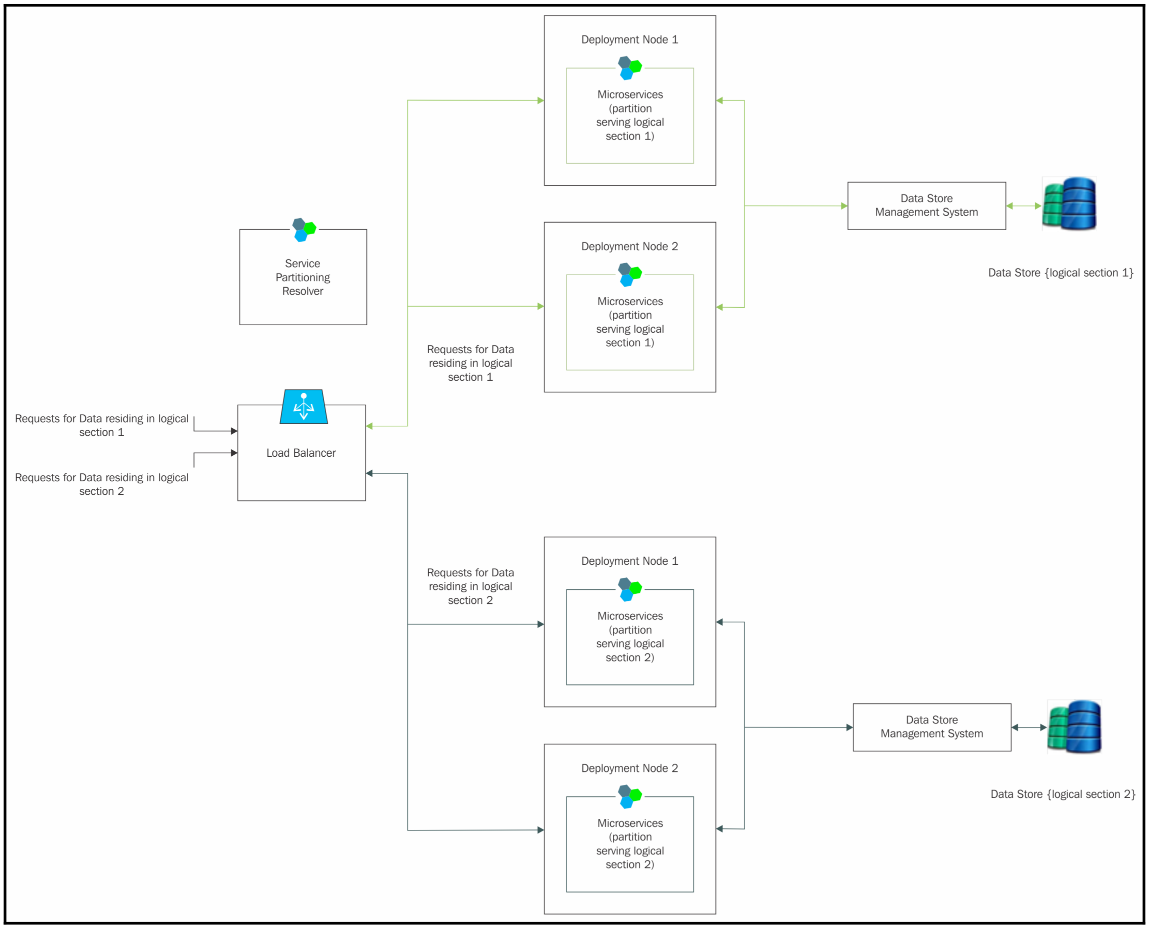
Solution

A solution for this problem is to logically divide the state of the system in to partitions which can then be served separately with minimal dependency on one another.

Partitioning can be defined as the concept of dividing state and compute into smaller units to improve scalability and performance of a system. Partitioning is more suited for stateful Microservices than for stateless Microservices. Partitioning will define logical boundaries to ensure that a particular service partition is responsible for a portion of the complete state of the service. The services will be replicated on every partition.

Before implementing partitioning, it is important to device a partition strategy which supports scale out. Partition strategy should also ensure even distribution of the load across partitions.

The following figure illustrates a system which uses partitioned data store:


Deployment Node 1
Microservices (partition serving logical section 1)
Deployment Node 2
Microservices (partition serving logical section 1)
Data Store Management System
Data Store {logical section 1}
Service Partitioning Resolver
Requests for Data residing in logical section 1
Requests for Data residing in logical section 1
Load Balancer
Requests for Data residing in logical section 2
Requests for Data residing in logical section 2
Deployment Node 1
Microservices (partition serving logical section 2)
Data Store Management System
Deployment Node 2
Microservices (partition serving logical section 2)
Data Store {logical section 2}

Partitioning (Solution)

Microsoft Azure Service Fabric natively supports partitioning for stateful Microservices. Service Fabric offers the following three choices for partitioning Microservices:

- Ranged partitioning
- Named partitioning
- Singleton partitioning

More of Service Fabric partitioning can be found at: `https://azure.microsoft.com/en-us/documentation/articles/service-fabric-concepts-partitioning/`.

Considerations

The following considerations should be taken into account while implementing this pattern:

- **Partitioning Strategy**: Choosing a partition strategy is the key to efficiently balance load across multiple partitions. Following are few of the partitioning strategies which can be considered:
    - Horizontal Partitioning (Sharding)
    - Vertical Partitioning
    - Functional Partitioning

  Data Partitioning guidelines published by Microsoft (`https://msdn.microsoft.com/en-us/library/dn589795.aspx`) is a good reference document to understand these strategies.

- **Data replication**: It is advisable to replicate static data and commonly used data across data partitions to avoid cross partition queries which can be resource consuming.
- **Rebalancing**: Always plan for rebalancing tasks for the partitions which may be required as the partitions age. Rebalancing may cause system downtime.
- **Referential integrity**: Consider moving the ownership of referential integrity to the application layer instead of the database layer to avoid cross partition queries
- **Transactions**: It is advisable to avoid transactions which access data across partitions as this may introduce performance bottlenecks
- **Consistency**: When data is replicated across partitions, it is important to decide on a consistency strategy. You will need to decide if your application requires strong consistency or if it can manage with eventual consistency

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Throttling pattern**: Use this pattern if the load on a partition increases unexpectedly. Throttling is a leading indicator for the requirement of re-partitioning of data
- **Health Endpoint Monitoring pattern**: Continuous monitoring is advisable to regulate the load on each partition. Auto-scaling logic can be applied to alter the capability of a partition based on the load.
- **Sharding pattern**: This pattern can be used to horizontally partition your data to improve scalability when storing and accessing large volumes of data.
- **Command and Query Responsibility Segregation (CQRS) pattern**: Segregating operations that read data from operations that update data by using separate interfaces.

Use cases

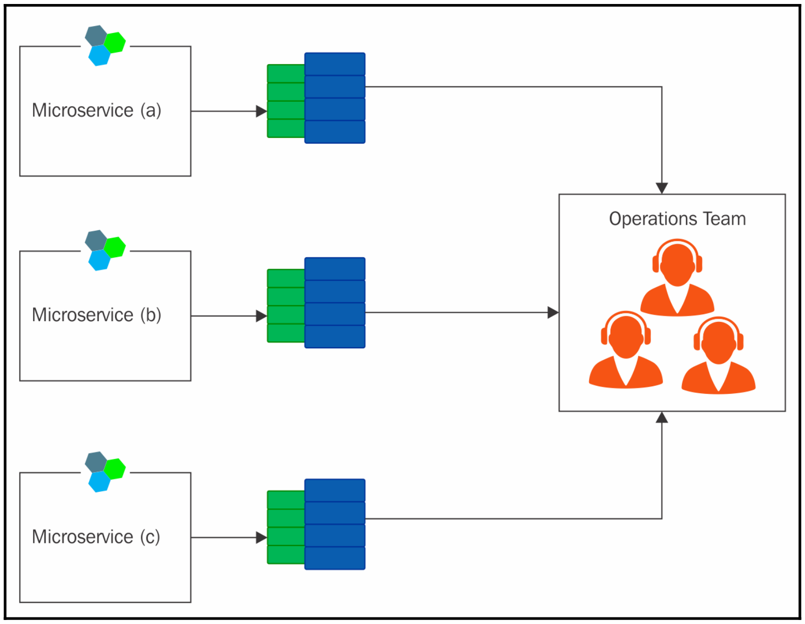
Following are few use cases where this pattern will be a right fit:

- Near real-time systems: Systems which require near real time performance which cannot afford delays in data access from persistent stores.
- Priority workload: Not all operations in a system will hold equal priority. Operations which had to be executed on a higher priority can be separated as a different partition served with more capable hardware to optimize performance.
- Geo dispersed user base: Storing the data close to the users can be beneficial in case of a Geo-dispersed deployment. Partitioning data based on user local can be useful in this scenario.

Centralized Diagnostics Service

Problem

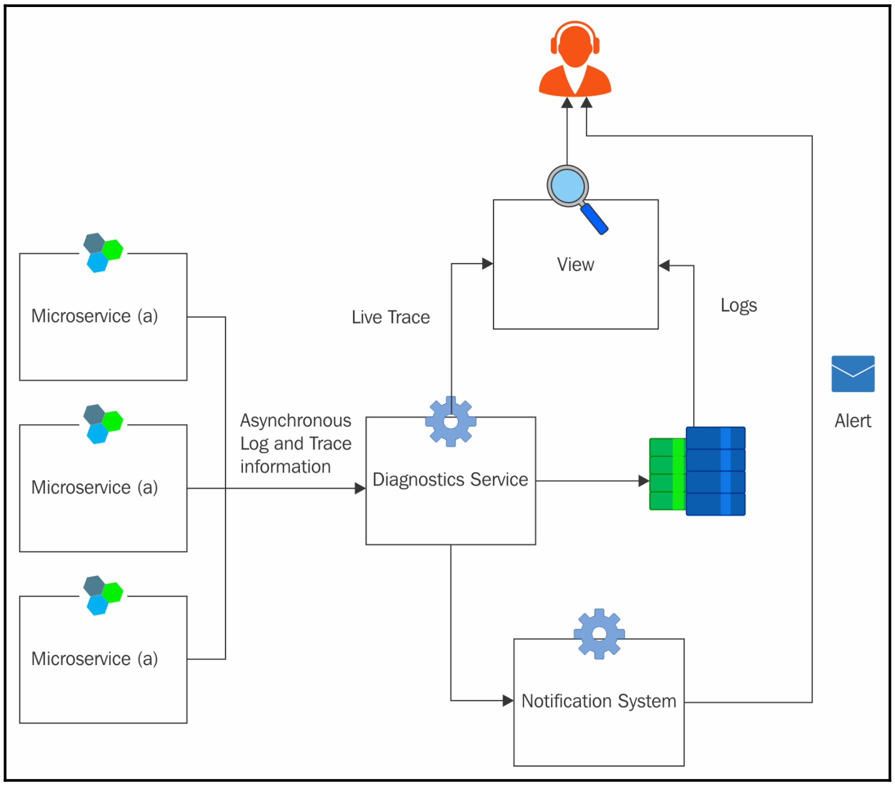
Diagnostics are a critical part of any system which provides insights about the state, health and transactions handled by the system. System logs and call-traces form the primary sources for diagnostics information. With every Microservice instance continuously generating logs and traces, maintaining, querying, and accessing these logs from different sources becomes an operational challenge. When logs are persisted in multiple stores, generating insights, and alerting also becomes complicated. Following is an illustration of an operations team receiving diagnostics information from distributed systems:



Centralized Diagnostics (Problem)

Solution

The solution to this problem is to delegate logging and tracing responsibilities to a separate service which can consolidate diagnostics information from multiple Microservices/Microservice-instances and persist it in a centralized store. This also helps in decoupling the diagnostics activities from business logic there by enabling the ability to scale services independently. The diagnostic service should also be capable of providing a live stream of trace information apart from persisting logs on secondary storage. Centralized logging also enables opportunities for drawing analytics around scenarios like patterns of faults, business trends, and threats. This service can be enhanced by coupling it with notification capabilities to send alerts to the operations team on a specific system state or event. Following is an illustration of a Centralized logging service in action:



Centralized Diagnostics (Solution)

Microsoft Application Insights can be used to consolidate logs and tracing information from Microservices deployments. Application insights has the following inbuilt capabilities which simplify overheads around diagnostics for Microservice deployments:

- Centralized logging
- Proactive detection
- Continuous export (to blobs)
- Ability to search and filter logs
- Performance metrics
- Interactive data analytics
- Multilingual SDKs

More about application insights capabilities can be found at: `https://azure.microsoft.com/en-us/services/application-insights/`.

Considerations

The following considerations should be taken into account while implementing this pattern:

- **Logging agent**: The logging agent which is installed on every instance of a Microservice needs to be a lightweight in terms of resource consumption. Making the logging service asynchronous to ensure that the calls are unblocked is a key factor in implementing a diagnostics service.
- **Permissions**: With the logs stored in a centralized location, the rights to access the logs should be configurable, and Role Based Access Control (RBAC) method should be preferred.
- **Availability**: Diagnostics is an important part of a system. Especially if the system is transactional in nature. It is important to ensure high availability of the diagnostics service. Along with the service front end, the persistent log stores also need to be replicated to meet the availability requirements.
- **Querying**: Ability to search through the logs for an event or a state is an important capability of a diagnostics service. Enabling indexing around logs and traces can improve the efficiency around searching.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Retry pattern**: This pattern should be implemented on the logging agent to handle transient faults on the diagnostics service
- **Queue-Based Load Leveling pattern**: Queues can be introduced as buffers between logging agent and the diagnostics service to prevent data loss
- **Scheduler Agent Supervisor pattern**: A scheduler can be used to purge and archive logs to improve storage efficiency of the system
- **Index Table pattern**: This pattern can be used to index the log entries to improve the efficiency around searching.

Use cases

Following are few use cases where this pattern will be a right fit:

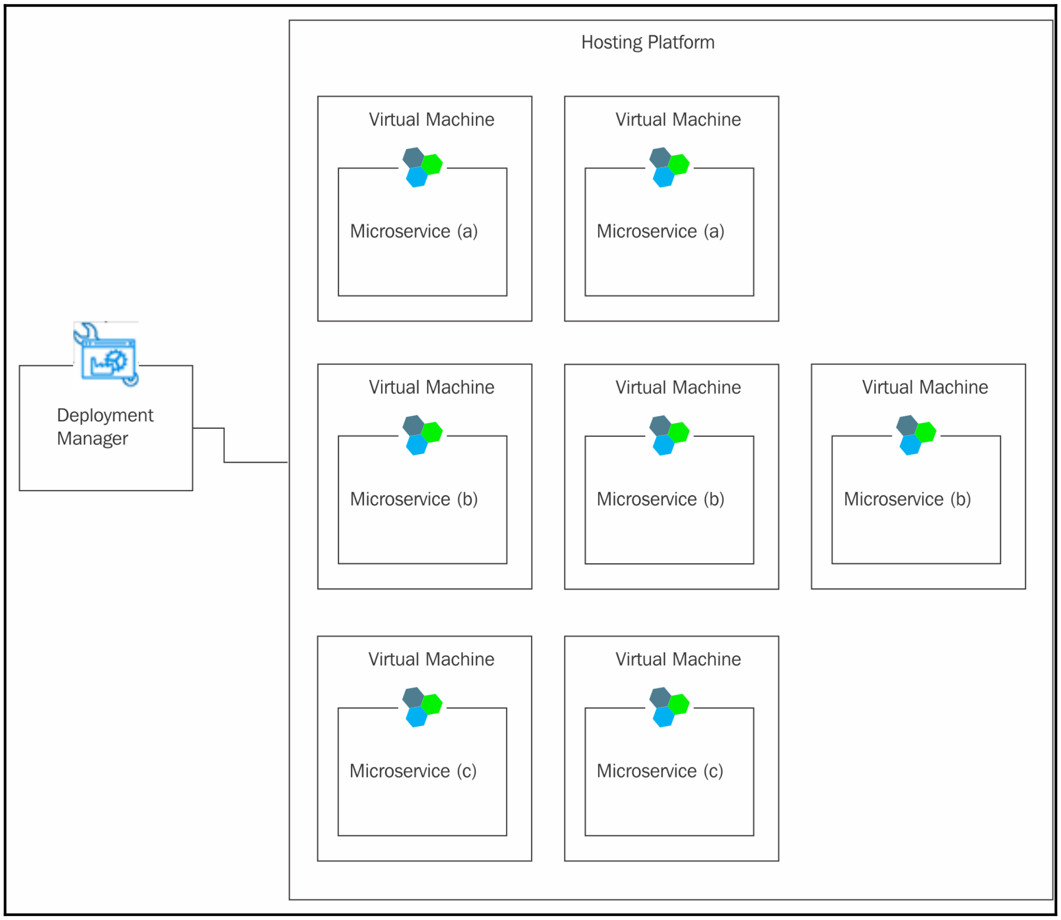
- **Hyper scale deployments**: This pattern is most effective in handling hyper scale deployment of Microservices as it helps in automating most of the diagnostics activity there by reducing operational overhead
- **High Availability systems**: For systems which require high availability, it's important to ensure that the turnaround time in case of a fault is minimal. This requires automation of diagnostics and capabilities around predictive detection and analytics.
- **Geo dispersed system**: Systems which are deployed across the globe usually has a centralized operations team. It is beneficial to equip the operations team with diagnostics capabilities like alerting and analytics to better offer round the clock support.

High Density Deployment

Problem

While architecting an enterprise system, separation of concerns is a commonly used design principle used to maintainability of the system. This leads to isolation of workloads as separate work packages (computational units) which are then deployed on hosting surrogates like web sites, virtual machines, containers, and so on, Often, this principle of isolation extends to the physical hosting infrastructure which may cause work packages to be deployed on separate virtual machines. Although this approach simplified the logical architecture it also causes underutilization of hardware resources there by increasing the operational and hosting cost.

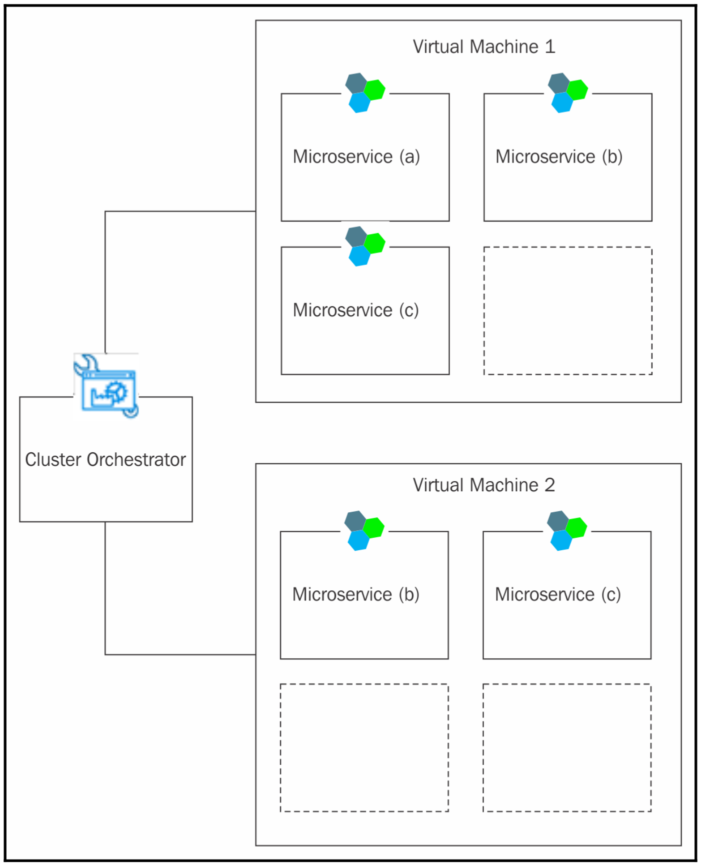
The following diagram illustrates poorly utilized resources for a set of services:



High Density Deployment (Problem)

Solution

Adopting high density deployment by enabling deployment of multiple Microservices on a single computation unit can be used to address this problem. Grouping the Microservice which can be co-deployed on a virtual machine can be based on features, communication patterns, scalability requirements, application lifecycle, resource requirement, and so on. The cluster manager managing the Microservice deployment should be able scale each of the Microservice separately. Cluster manager will also be responsible for the health of a virtual machine and failover to ensure high availability of a Microservice. Following is an illustration of how a cluster orchestrator can be used to achieve high density deployment:



High Density Deployment (Solution)

Microsoft Azure Service Fabric offers natively supports high density deployment of Microservice on a cluster of virtual machines. The Service Fabric cluster manager also possesses rich cluster orchestration capabilities. This requires a detailed cluster design and capacity planning for both the stateful and stateless services. You can read more about capacity planning at:

`https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-capacity`.

More about Service Fabric cluster management can be found at: `https://azure.microsoft.com/en-us/documentation/articles/service-fabric-cluster-resource-manager-cluster-description/`.

Considerations

The following considerations should be taken into account while implementing this pattern:

- **Isolated namespaces**: Deploying multiple Microservices on a single virtual machine introduces challenges around securing resource boundaries around each service. Cluster manager should be responsible for managing the isolation.
- **Resource starvation**: Cluster manager that supports high density deployment should also support enforcing of limits around the resources each Microservice can consume to avoid resource starvation. Throttling logic should be implemented to gracefully handle resource requests after the thresholds are reached.
- **Release management**: Deploying, updating or de-allocating a Microservice should not impact the stability of other Microservices running on the same virtual machine.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Compute Resource Consolidation pattern**: A similar design pattern which supports high density deployment
- **Throttling pattern**: Helps control the consumption of resources used by a Microservice running on a shared hardware
- **Health Endpoint Monitoring pattern**: Continuous health monitoring pattern can be used to ensure requirements around high availability

Use cases

Following are few use cases where this pattern will be a right fit:

- **Enterprise scale Microservice deployment**: This pattern is applicable for any enterprise scale Microservice deployment considering requirements around optimizing operational and hosting expense.
- **Hybrid deployments**: Capability of the cluster manager to deploy Microservices on heterogeneous hardware environments comes handy in case of a hybrid data center where the hardware configurations can be different in different data centers.
- **High performance systems**: Optimizing the deployment based on the resource requirements of a Microservice and ensuring the availability of resources by setting thresholds ensures high performance of Microservices

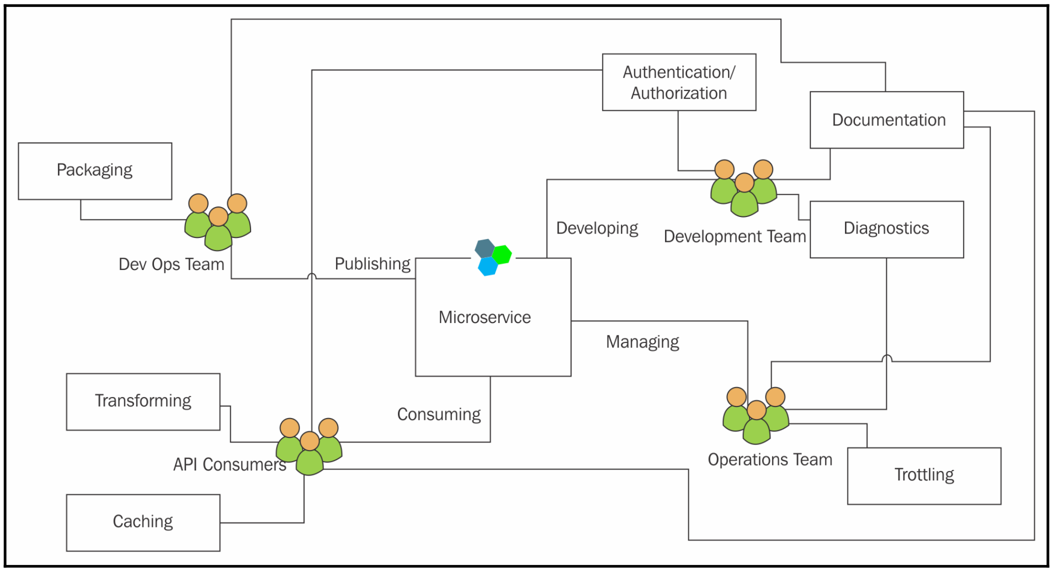
API Gateway

Problem

Decomposing a monolithic architecture into a Microservice architecture exponentially increases the complexity in managing APIs. The challenges around API management can be categorized on to three divisions:

- **Abstraction**: Abstracting the API developers and the consumers from complexities around security, transformation, analytics, diagnostics, throttling and quota management, caching, and so on.
- **Publishing**: Publishing APIs, grouping APIs as bundles which can be associated with a business use case, deriving insights from consumption, and managing consumers.
- **Documentation**: Creating and maintaining a catalog of deployed APIs, documenting the ways they can be consumed, analytics, and so on.

Manually managing these responsibilities become extensively complex especially in hyper scale enterprise ready deployment of Microservices:

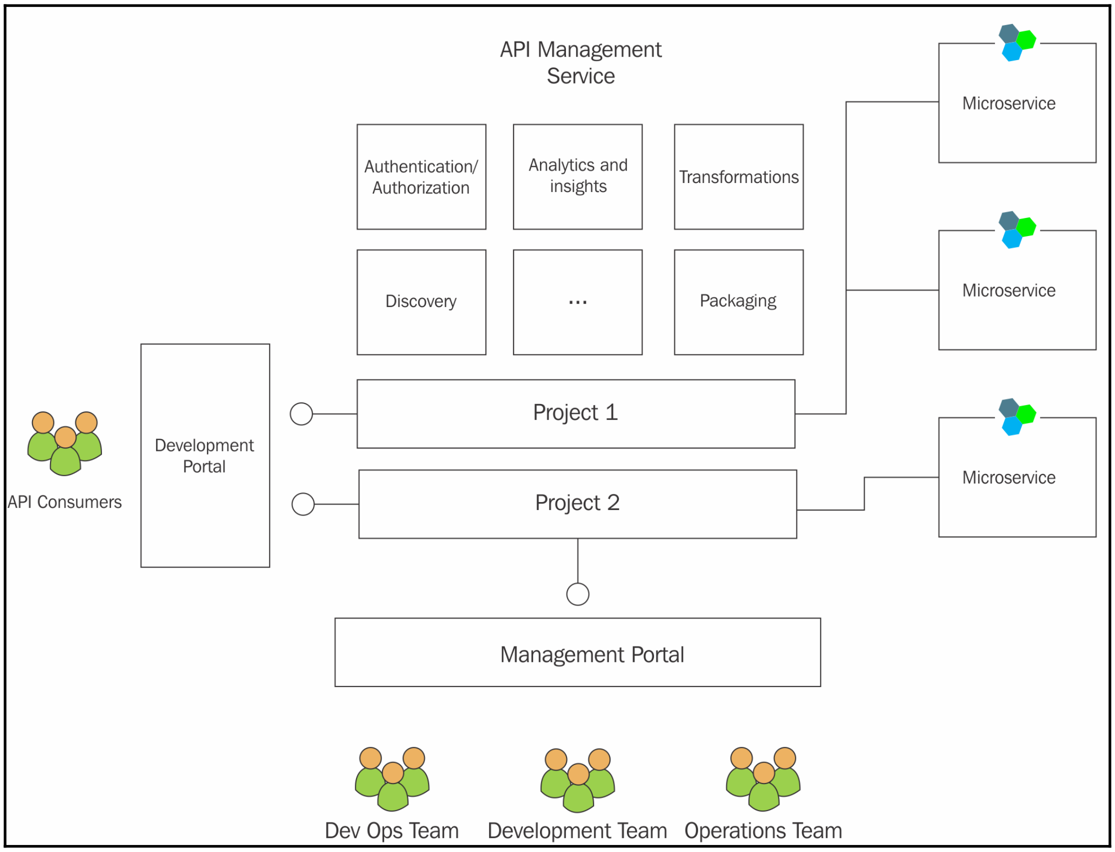


API Gateway (Problem)

Solution

Introducing an API gateway as a management service helps solve complexities around managing Microservices there by letting developers focus better on solving business problems. The API Gateway will be responsible to providing the supporting services for securing, managing, publishing, and consuming Microservices. This expedites consumer onboarding by automating most of the management processes.

API Gateway also have the capability of abstracting the deployment location of the actual APIs:



API Gateway (Solution)

Microsoft API Management is a feature rich service which can act like a gateway for Microservices. API management comes with built-in capabilities around discoverability, security, analytics, business insights, documentation, and transformations which can help enhance the experience around consumption of Microservices to a great extent. More about API management can be found at: `https://azure.microsoft.com/en-us/services/api-management/`.

Considerations

The following considerations should be taken into account while implementing this pattern:

- **Another layer**: While introducing an API gateway, it is important to account for the increase in latency which may be caused in consuming the APIs. In most of the cases this is negligible and can be optimized by caching the API responses.
- **Centralizing diagnostics**: API gateway will have its own diagnostics components which logs information to a store which may be different from the one use by the hosted Microservices. It may be useful to consolidate these logs to improve operational efficiency.
- **Abstraction leads to information filtering**: Abstracting responsibilities will filter out information which may be relevant for the operation of a Microservice. A good example is user (consumer) context which may be of relevance to the Microservice will be filtered out by the API gateway as it is responsible for authentication/authorization.
- **Access control**: It is important that the API gateway supports role based access control as it needs to cater to different teams with different responsibilities.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Cache-Aside pattern**: Caching can be used to buffer API responses to optimize turnaround time
- **Circuit Breaker pattern**: Can be used to improve the stability and resiliency of an application by handling transient faults
- **Gatekeeper pattern**: This design pattern can be used to secure Microservices by preventing direct access
- **Throttling pattern**: Control the consumption of Microservice regulating the calls from a consuming application
- **Retry pattern**: A common pattern used to handle transient faults

Use cases

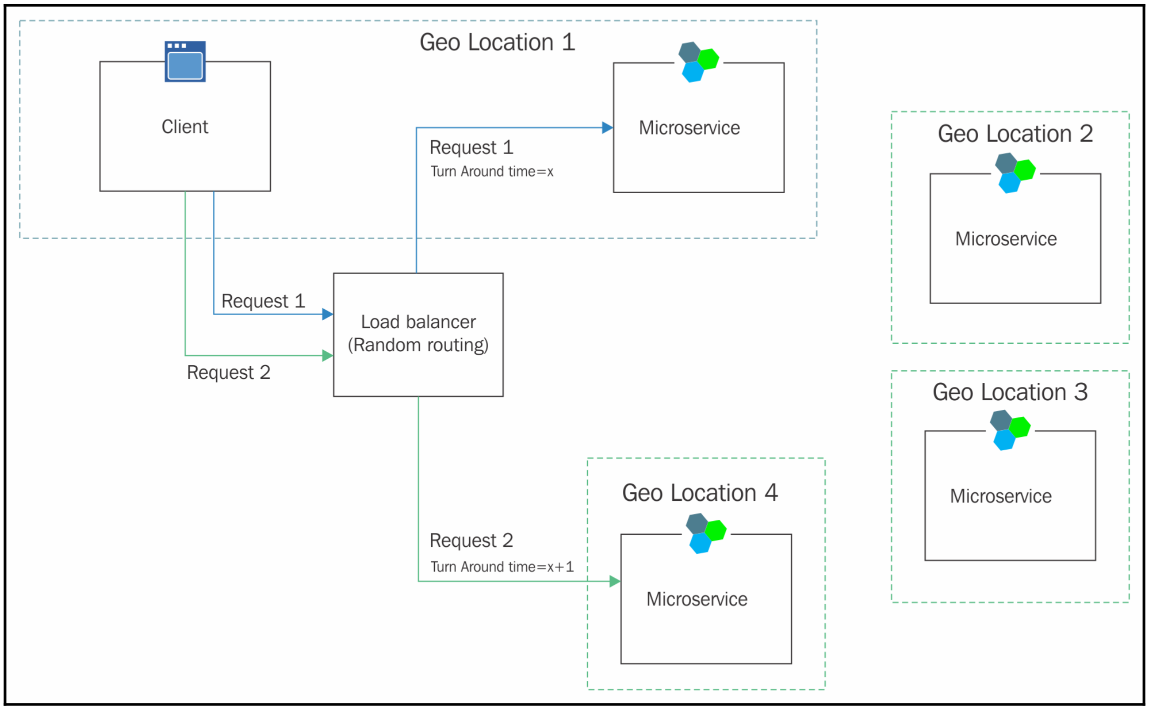
Following are few use cases where this pattern will be a right fit:

- **Multi datacenter deployment**: When your Microservices are deployed across multiple datacenters, API gateway acts as a façade for managing, packaging and consuming Microservices without being aware of the actual deployment location.
- **Developer focused**: With capabilities around packaging, maintaining catalogues and documenting Microservices, API gateway becomes single source of truth for developers to build applications using Microservices.
- **Analytics driven business**: A centralized channel for consuming APIs helps build better analytics which then can be used to build insights to identify trends which may impact the business.
- **Secure systems**: API gateway acts like a gatekeeper preventing consumers from directly accessing the Microservices. Responsibilities like authentication, authorization and throttling are owned by the API gateway there by providing a centralized control over all the Microservices running within an enterprise.

Latency Optimized Load Balancing

Problem

Microservice platforms today, supports geographically dispersed deployments. This means that multiple instances of a Microservice can be hosted in different continents on heterogeneous environments. While abstracting the service hosting location has advantages, it also introduces challenges around offering consistent performance for consumer requests from different parts of the globe. Having a load balancer which uses a round robin or random routing algorithm cannot guarantee the most optimal response in most cases.

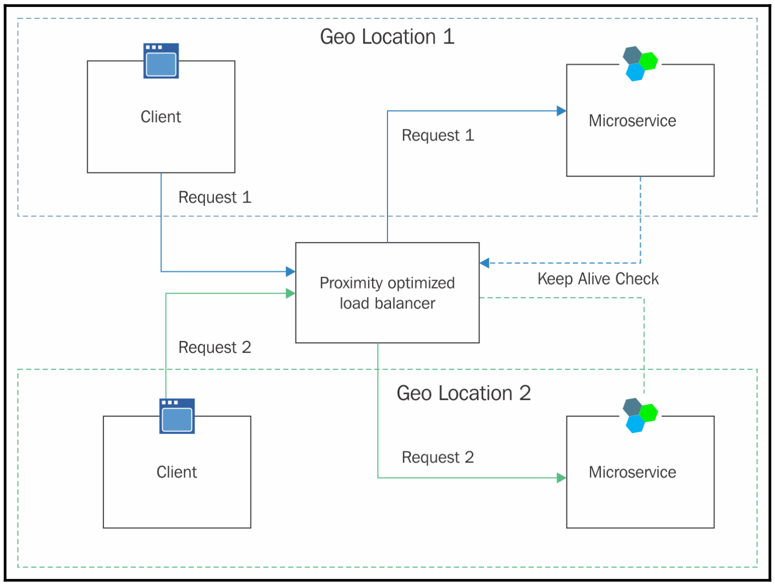


Latency optimized load balancing (Problem)

Solution

Load balancing plays an important role in optimal performance of any distributed system. This pattern optimizes the action of load balancing by considering the proximity of the Microservice deployment and the turnaround time from previous requests along with other parameters such as current load, health of the instance, and so on. This will enable the system to route a request from a consumer to the nearest performant Microservice instance.

Latency optimized load balancing is also effective in hybrid deployments of Microservices across public and private clouds:



Latency optimized load balancing (Solution)

Microsoft Azure Traffic Manager allows you to control the distribution of traffic based on a rule set which can be configured to achieve high availability and performance. More and Traffic manager can be found at: `https://azure.microsoft.com/en-us/documentation/articles/traffic-manager-overview/`.

Apart from routing, traffic manager can also be used to enable automatic failover to improve the availability of your Microservice deployment.

Considerations

The following considerations should be taken into account while implementing this pattern:

- **Route optimization**: Optimizing a route is a complex process. Considering that load handled by an instance of a Microservice can change quickly, the records about previous response time may not lead to the most optimal route. Constant performance monitoring need to happen to analyze the load on virtual machines hosting Microservices.

- **Health checks**: Microservices can be hosted on commodity hardware which are susceptible to failure. Load balancer should be cognizant about the health of the virtual machines hosting Microservices.
- **Point of failure**: Load balancer can act like a single point of failure. It is critical to make load balancing service and its associated state highly available. Geo-replicating instances of the load balancer is advisable to achieve high availability.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Gatekeeper pattern**: The load balancer implicitly acts like a gatekeeper. Load balancing service can be enhanced with capabilities of this pattern to achieve better security.
- **Health Endpoint Monitoring pattern**: A load balancer need to be cognizant of the health of all Microservice instances it is serving. Health monitoring is a recommended strategy to monitor health of the virtual machines.
- **Throttling pattern**: Helps ensure scalability of the service by throttling request based on the capacity of the costed services.

Use cases

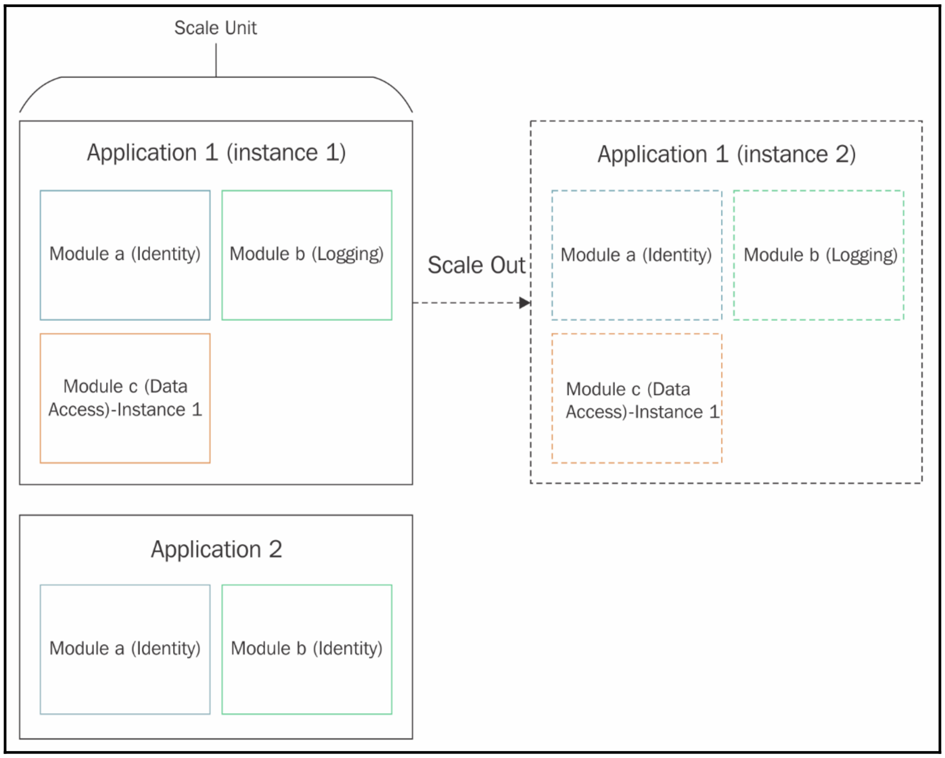
Following are few use cases where this pattern will be a right fit:

- **Near real-time systems**: Latency optimized load balancing is best suited for environments which are very dynamic in terms of the load they encounter and require high performance in terms of turnaround time for requests.
- **Geo-dispersed deployments**: This pattern is highly recommended for Geo-dispersed deployments of Microservices which caters to consumers around the world. Optimizing the routing logic helps improve the turnaround time for responses thereby providing a better performance.
- **Highly available systems**: Apart from optimizing the route, latency optimized load balancer can also be used to enable an automatic failover strategy. The service will start routing the request to a different Geo-location if it sees the performance degrading in the preferred location.

Queue Driven Decomposing Strategy

Problem

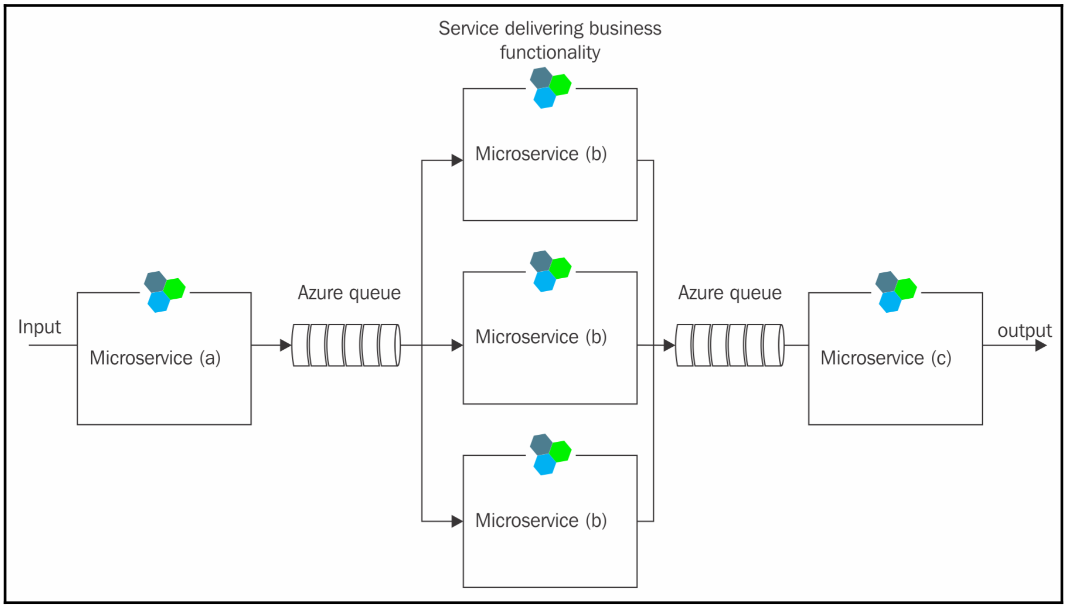
One of the biggest challenges in the Microservice architecture is to decompose services into reusable standalone services which can execute in isolation. The traditional monolithic architecture often contains inseparable redundant implementation of modules which minimizes the opportunities of reuse. With modules tightly coupled, maintaining it, in terms of upgrading, scaling, or troubleshooting it without affecting the whole application becomes impossible:



Queue driven decomposing strategy (Problem)

Solution

Decomposing tasks in to discrete sub tasks which can be encapsulated as standalone services is the fundamental concept behind the Microservice based architecture. These standalone services or Microservices are best reused if their inputs and outputs are standardized. A system comprising of Microservices will have the flexibility to independently scale at the level of each Microservice. This introduces the complexity of balancing load among multiple instances of the Microservice which is easily solved by introducing queues to level the load. A collection of Microservices couples with queues together form a pipeline to implement a business functionality:



Queue driven decomposing strategy (Solution)

Microsoft Azure Service Bus queues or Azure Storage Queues can be used as a buffer between Microservices deployed on Microsoft Azure Service Fabric to implement this design pattern with minimal plumbing work.

Considerations

The following considerations should be taken into account while implementing this pattern:

- **Increase in Complexity**: Decomposing an application in to Microservices, introduces the complexity of managing these services discretely. Also, the communication between the processes becomes less deterministic as they are asynchronous by nature. It is important to account for the overheard in management effort while implementing this pattern.
- **Poison messages**: There is a chance that an instance of a Microservice in a pipeline fails to process a message pipe as it causes an unexpected fault. In this case, the Microservice will retry processing this message for few times and then discards it. These messages usually end up in a dead letter queue which then needs to be handled separately.
- **Transactions**: Incase an end-to-end processing of a pipeline fails in between, the state changes which are committed as a part of the current transaction must be reversed to preserve the integrity of the system.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Retry pattern**: Defines a recommended way of implementing retry logic to handle transient faults
- **Compensating Transaction pattern**: This pattern can be used to reverse state changes caused by a faulty transaction
- **Queue-Based Load Leveling pattern**: Pipes are best implemented using queues. Apart from providing a reliable delivery channel queues can also help in load balancing
- **Pipes and Filters pattern**: A similar pattern recommended for cloud deployed services

Use cases

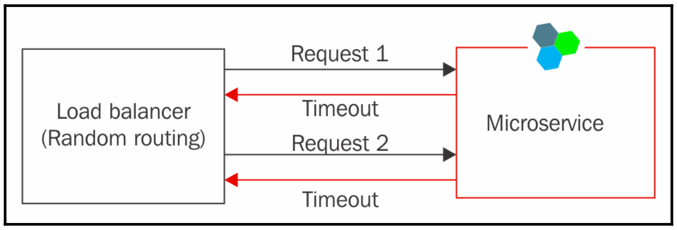
Following are few use cases where this pattern will be a right fit:

- **Enterprise workflows**: This pattern is highly recommended for implementing workflows on top of a Microservice deployment to ensure reliability. Apart from acting as a channel to couple two Microservices, queues also act like a buffer for persisting workflow state.
- **Uneven scalability requirements**: Systems which have requirements around scaling a specific Microservice separately based on variable workload. The Microservices participating in a pipeline can be scaled out/in at every level thereby improving the capability of the system in being elastic.

Circuit Breaker

Problem

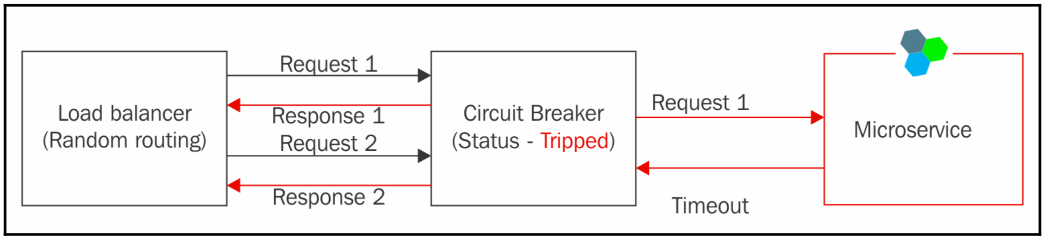
In a Microservice based system, individual services may fail at any point of time. If a client calls a service frequently, then each call would need to wait for a timeout before failing the operation. Making frequent calls to a failing service and waiting for response wastes system resources and slows down the whole application:



Circuit Breaker(Problem)

Solution

The circuit breaker pattern prevents calls to be made to a failing resource once the number of failures cross a particular threshold. To implement the circuit breaker pattern, wrap calls made to a service in a circuit breaker object. The circuit breaker monitors the failures attempts made to the service. Once the number of failed attempts to invoke a service exceeds a threshold value, the circuit trips for that service and any further calls to the service are short-circuited. The circuit breaker may keep the count of failed requests made to the service in a shared cache so that the count may be shared across all instances of the application. This setup ensures that application does not waste resources in waiting for response from a failing service. A fallback logic maybe implemented that gets invoked when the circuit breaker trips while the connection is still open which may return a value to the client application:



Circuit Breaker (Solution)

Considerations

The following considerations should be taken into account while implementing this pattern:

- **Performance**: Using circuit breaker has a performance implication depending on the type of cache used to store the failed request count.
- **Reacting to failures**: Clients using the circuit breaker pattern should implement fallback logic in case of failures. The client may either fail the operation or carry out a workaround such as saving the request in a queue that would be processed later.

- **Concurrency**: A large number of concurrent instances of an application may access the same circuit breaker object. The implementation of the circuit breaker should not block concurrent requests or add excessive overhead to each call to an operation.
- **Types of exceptions**: The type of exceptions that trip the circuit breaker should be carefully considered. A logical exception should not cause the circuit breaker to trip and short-circuit calls made to the underlying service.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Retry pattern**: Retry pattern works together with the Circuit Breaker pattern. The retry pattern is best used for transient fault handling which should be anticipated for cloud deployment of Microservices.
- **Health Endpoint Monitoring pattern**: Employing agents (external processes) to verify that applications and services are performing correctly.

Use cases

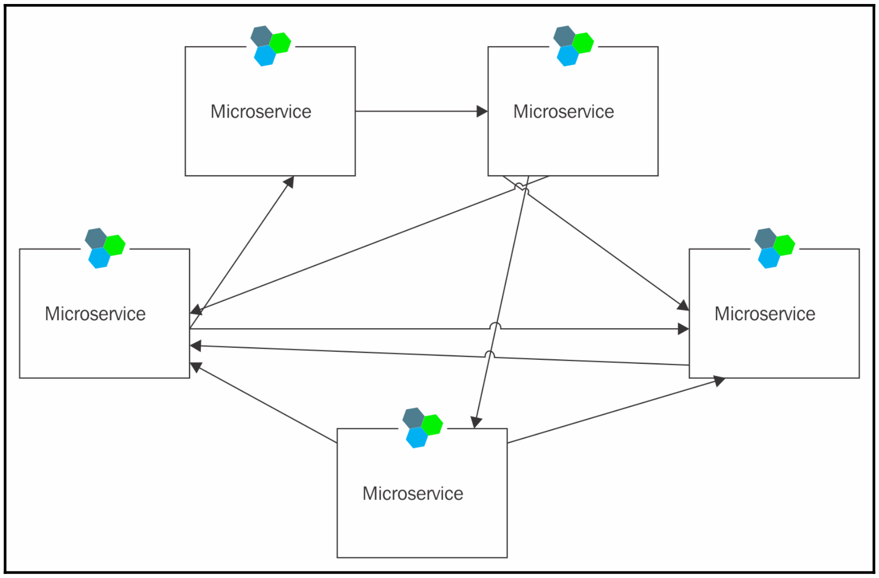
Following are few use cases where this pattern will be a right fit:

- If an operation to invoke a remote service or access a shared resource is highly likely to fail then the circuit breaker actively prevents it.
- Circuit breaker is not a substitute for handling exceptions in the business logic of your applications.

Message Broker for Communication

Problem

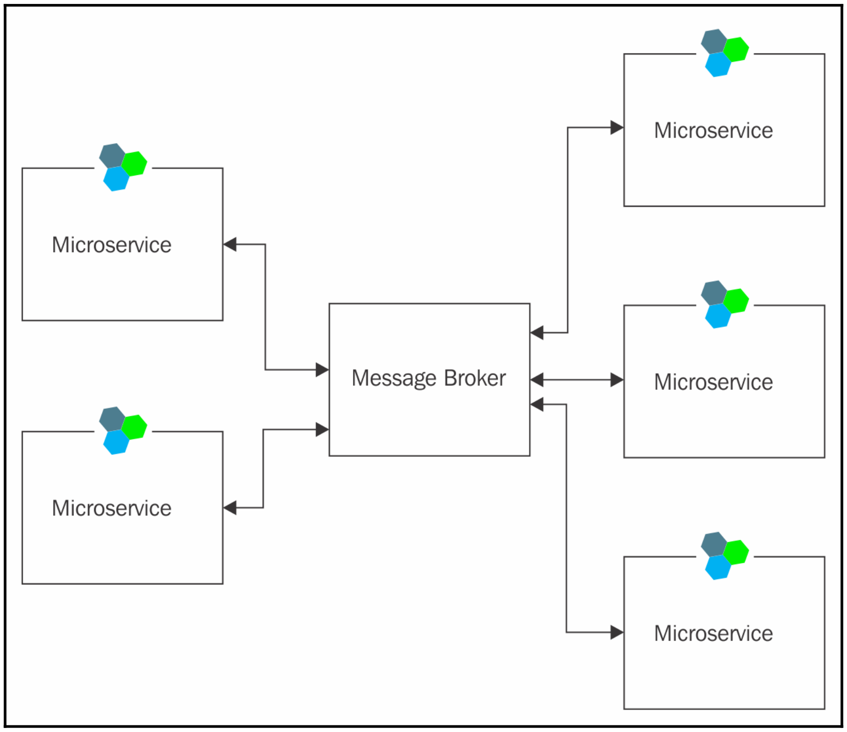
A Microservices based system is composed of a number of services. Integration of these services is a challenge because point-to-point integration between services requires many connections between them. Many connections usually translate into many interfaces. Change in interfaces of communication of service may lead to changes in all the services to which it is connected. Also, in several scenarios point-to-point communication is not possible because the various services in a Microservice based solution could be deployed to different security realms:



Message Broker for Communication (Problem)

Solution

The various services in a Microservice based application can communicate with each other via a message broker. A message broker is a physical component that handles the communication between services. Instead of communicating with each other, services communicate only with the message broker. An application sends a message to the message broker, providing the logical name of the receivers. The message broker looks up the services registered under the logical name and then passes the message to them:



Message Broker for communication(Solution)

Such a service in Azure may be realized by using Azure Service Bus Topics and Subscriptions. Each service may publish messages on the topic and specify a set of properties of the message that identify the receiver this message should reach to. The services may create a number of subscriptions on the topic to receive messages addressed to them.

Considerations

The following considerations should be taken into account while implementing this pattern:

- **Performance**: The message broker pattern adds another component to your system and therefore the involved services will experience additional latency in communication. Point-to-point communication has lower latency than communicating via a broker.
- **Increased agility**: This pattern helps isolate change to individual services. Due to loose coupling of services, this pattern facilitates changes in the application.
- **High scalability**: Due to loose coupling of services, each service can be individually scaled. This gives the application administrators fine-grained control of scaling the services.
- **Ease of development**: Since each functionality is isolated to its own service, developing a service or functionality will not affect the other services and functionalities.
- **Load balancing**: The services in a Microservice based application may get scaled at different rates. The messages broker queues requests waiting to be processed by the services which helps the services consume the messages at their own rates.
- **Increased complexity**: Communicating with message broker is more complex than point-to-point communication.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Pipes and Filters**: This pattern helps perform complex processing on a message while maintaining independence and flexibility
- **Message Router**: This pattern helps decouple individual processing steps so that messages can be passed to different filters based on a set of conditions
- **Publish/Subscribe pattern**: This pattern helps broadcast an event to all interested subscribers
- **Point-to-Point Channel**: This pattern helps the caller be sure that exactly one receiver will receive the message or perform the call

Use cases

Following are few use cases where this pattern will be a right fit:

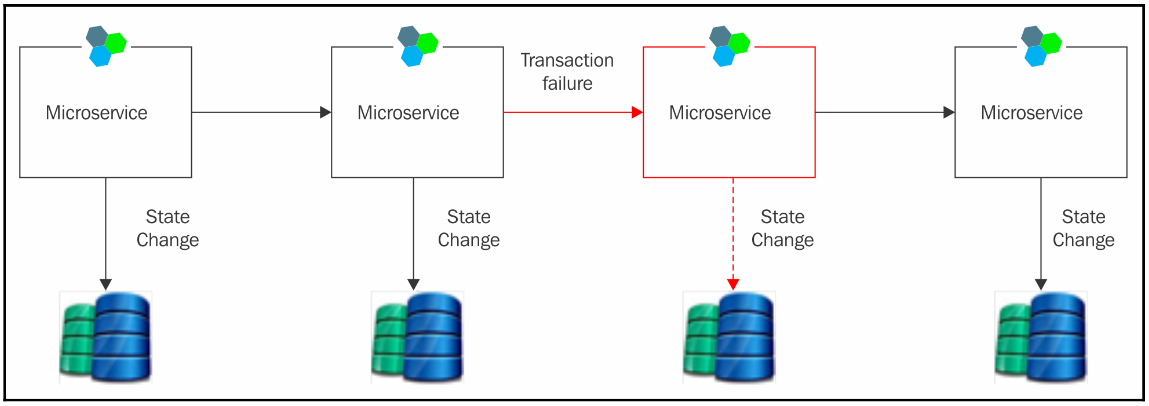
- **Reduced coupling**: The message broker decouples the senders and the receivers. The senders and receivers communicate only with the message broker.
- **Improved integrability**: The services that communicate with the message broker do not need to have the same interface. The message broker can also act as a bridge between services that are from different security realms.
- **Improved modifiability**: The message broker shields the services of the solution from changes in individual services.
- **Improved security**: Communication between applications involves only the sender, the broker, and the receivers. Other applications do not receive the messages that these three exchange. Unlike bus-based integration, applications communicate directly in a manner that protects the information without the use of encryption.
- **Improved testability**: The message broker provides a single point for mocking. Mocking facilitates the testing of individual applications as well as of the interaction between them.

Compensating Transaction

Problem

Transactions becomes complex in Microservices based system. This is because data owned by each Microservice is private to that Microservice and can only be accessed by the API exposed by the service. There is added complexity in Microservice based systems due to their polyglot persistence approach.

Although a partitioned and polyglot-persistent architecture has several benefits, this makes implementing transactions difficult:



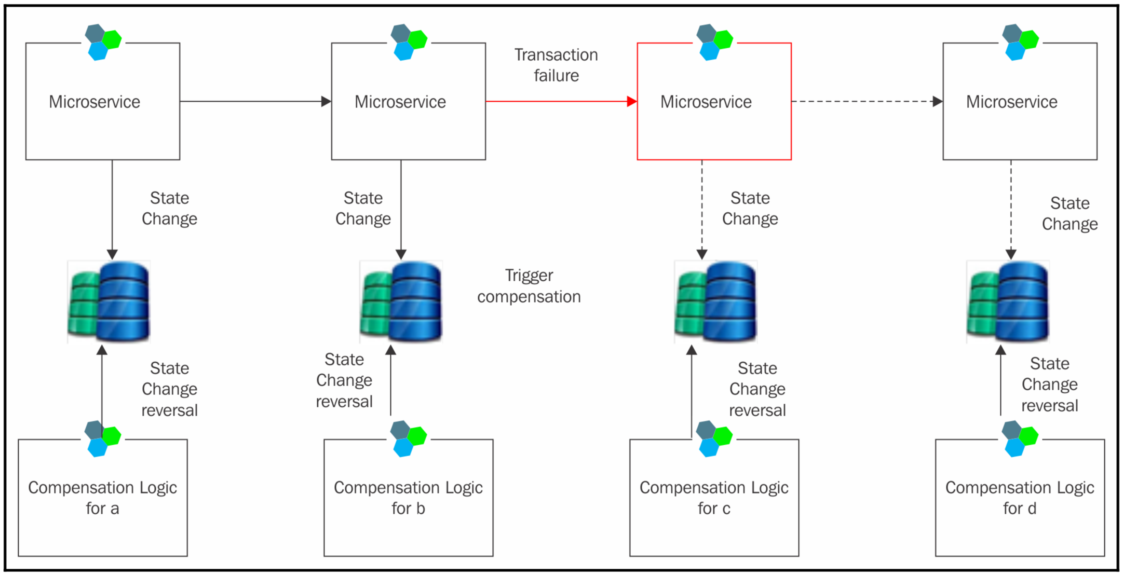
Compensating transaction (Problem)

Solution

For many Microservices based applications, the solution to achieve transactions is to use an event-driven architecture wherein state change is captured as events and published. Other services subscribe to these events and change their own data, which might lead to more events being published. Although Azure services such as Azure Service Bus Topics can be used to publish events, there are several other challenges with the approach such as how to atomically update state and publish events. Some of the approaches for maintaining atomicity are using the database as a message queue, transaction log mining, and event sourcing which are difficult to implement.

Eventual consistency of data may be achieved easily by the compensating transaction pattern. A common approach to realize this pattern is to use a workflow. A transactional operation should be designed as chain of separate work-steps that form a chain. If the operation fails at any point, the workflow rewinds back through the work-steps it has completed and reverses the work by using the compensator of each work-step.

The following diagram depicts work-steps and compensation steps in a multi-step workflow:



Compensating Transaction (Solution)

Using Azure Service Bus Queues auto forwarding property, such a workflow can be built easily. A working prototype of this pattern is available at Clemens Vasters' blog: `https://github.com/Azure-Samples/azure-servicebus-messaging-samples/tree/master/AtomicTransactions`.

Considerations

The following considerations should be taken into account while implementing this pattern:

- **Time-out scenarios**: An operation may not fail immediately but could block the operation. Time-outs must be considered while implementing this pattern.
- **Idempotency**: A work-step and its corresponding compensator should be idempotent which means that the compensator should not alter the state of work-step in a manner that it can be operated on again.
- **Data sufficiency**: A compensator should get sufficient data from the previous step to roll back the operation.
- **Retries**: Consider using retries in work-step before failing the operation.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Sagas**: Create workflows for Microservice based applications.
- **Retry pattern**: Use retry patter together with compensating transactions so that compensating logic doesn't get triggered on transient failures.

Use cases

Following are few use cases where this pattern will be a right fit:

- **Transactions**: In several scenarios such as travel bookings, the booking operations must be undone in case of failure. This pattern is useful for reversing operations in such scenarios.

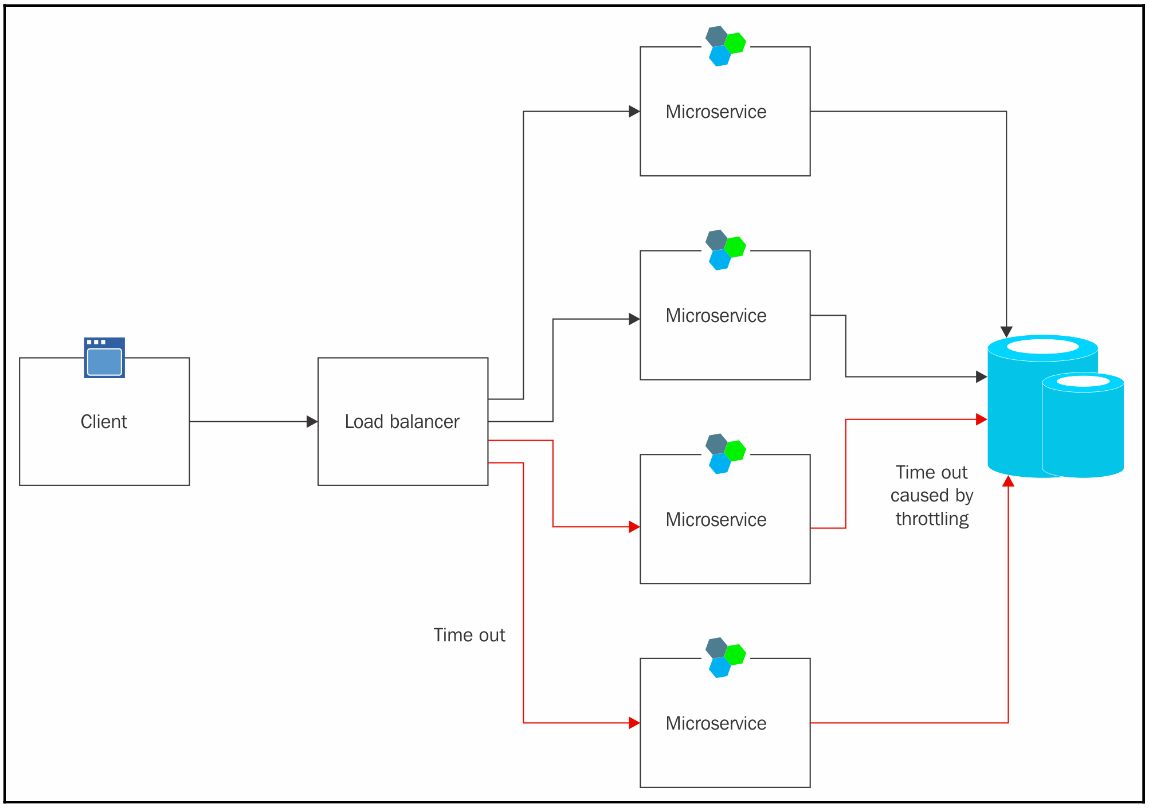
Rate Limiter

Problem

Individual services in Microservice architecture have different **Quality of Service** (**QoS**) guarantees. If the processing requirements of the system exceed the capacity of the resources that are available, it will suffer from poor performance and may even fail. The system may be obliged to meet an agreed level of service, and such failure could be unacceptable.

Using auto-scaling pattern helps provision more resources with an increase in demand. This pattern not only consistently meets user demand, but also optimizes running costs. However, auto-scaling is not the optimal solution in the following scenarios:

- The backend service or data store has throttling limits which affect the limit of scale the application can achieve.
- There might be resource deficit in the window of time when resource provisioning is still going on:

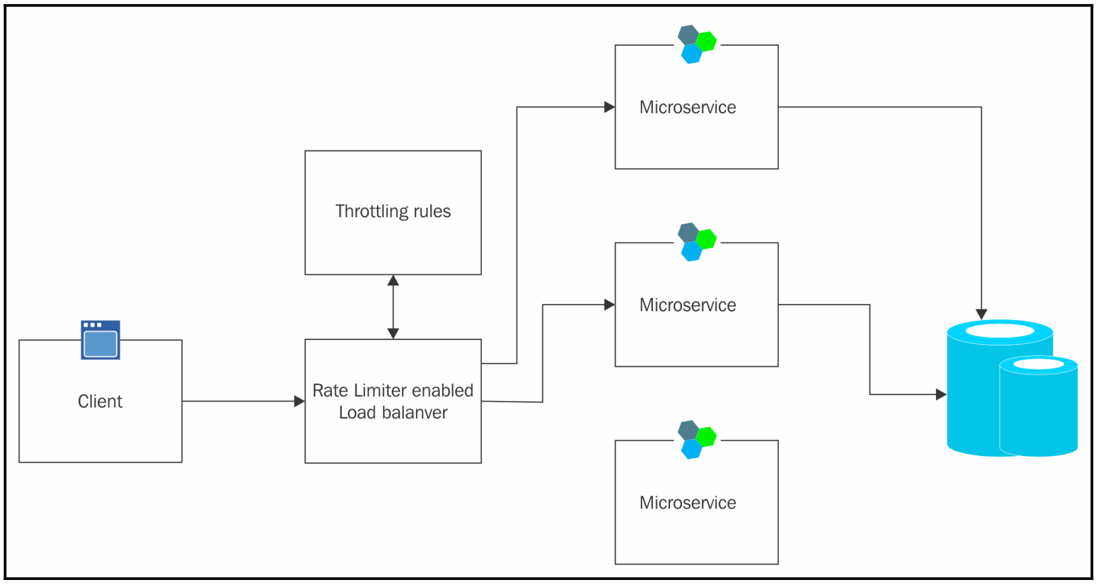


Rate Limiter (Problem)

Solution

A solution to overcome the above mentioned issues is to allow applications to use resources only up to a threshold limit and throttle the requests received after the limit is reached. The throttling limits may be applied to each user of the application or across users depending on the desired SLAs and QoS. Some of the throttling strategies that can be applied are:

- Rejecting requests from a client tagged with a particular key who has already accessed the APIs more than a particular number of times in a particular time duration
- Rejecting requests from a particular IP address after a particular number of requests have been made in a particular time duration
- Deferring operations performed by lower priority clients if the resources are nearing exhaustion



Rate Limiter (Solution)

Azure API Management has advanced throttling techniques that can be used to protect services and apply throttling limits on them.

Considerations

The following considerations should be applied while using this pattern:

- **Planning**: Decision to throttle requests affects the design of system. Throttling strategy and resources that need to be throttled should be decided on before implementation because it is difficult to add throttling after the system has been implemented.
- **Response**: A clear response indicating that user request has been throttled should be sent back to the client. This would help the client retry an operation after some time.
- **Auto-scale versus Throttle**: If there is a temporary spike in application load, then only throttling may be used instead of scaling the application.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Auto-scale**: Throttling can be used as an interim measure while a system auto-scales, or to remove the need for a system to auto-scale.
- **Queue-based Load Leveling pattern**: Queue-based load leveling is a commonly used mechanism for implementing throttling. A queue can act as a buffer that helps to even out the rate at which requests sent by an application are delivered to a service.
- **Priority Queue pattern**: A system can use priority queuing as part of its throttling strategy to maintain performance for critical or higher value applications, while reducing the performance of less important applications.

Use cases

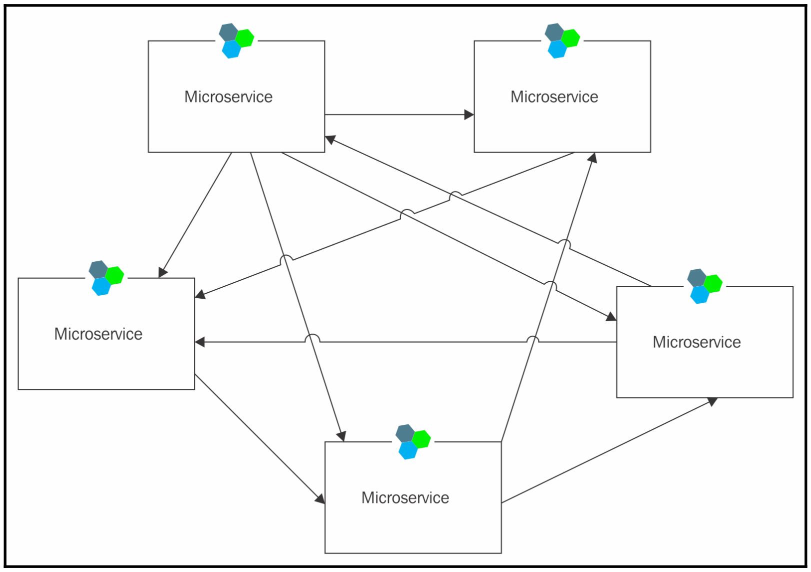
Following are few use cases where this pattern will be a right fit:

- To ensure that a system continues to meet service level agreements
- To prevent a single tenant from monopolizing the resources provided by an application
- To handle bursts in activity
- To help cost-optimize a system by limiting the maximum resource levels needed to keep it functioning

Sagas

Problem

In a distributed system, messages sent by the services might not be controlled. As the system grows and more services are added, the system starts becoming unmaintainable. In a large distributed system, it becomes hard to track which message goes where.



Sagas (Problem)

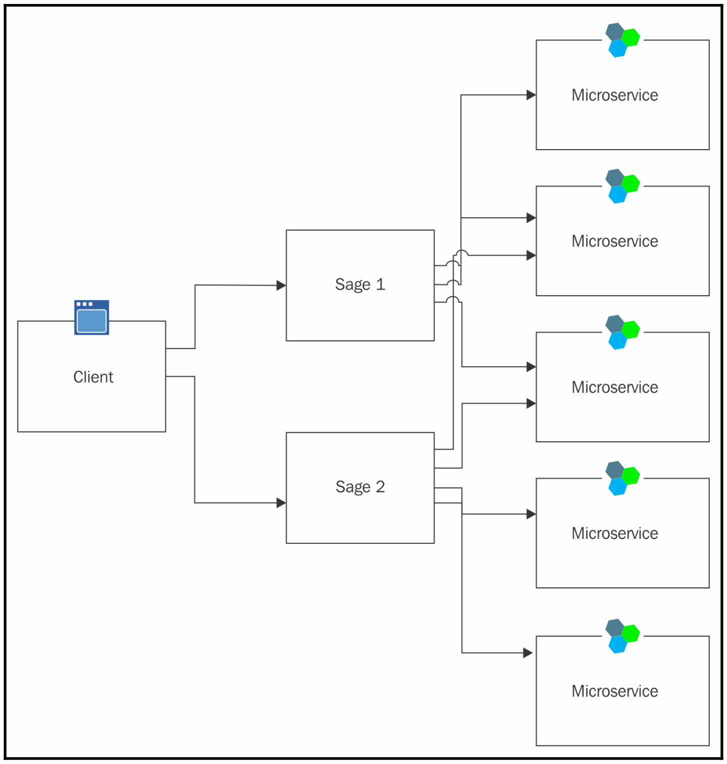
Another problem that Microservices-based system encounters is that there is no single source of truth that can be used by all the services. For instance, in an e-commerce application, when a customer places an order a message containing order details is sent to the procurement service. Once the order service responds, another message containing delivery details is sent to the delivery service. This leads to there being no single source of truth that contains the entire customer order data.

Solution

Using sagas, workflows can be implemented in a Microservice application. Sagas are state machines that model a workflow with steps to be executed. Sagas treat the various Microservices as implementations of workflow.

In a Microservice system, there is no central truth as the various Microservices each have a part of data stored in their database. Sagas can store the actual truth and each Microservice can get part of data it needs.

Saga maintains state in the form of object you define until saga finishes. A saga can coordinate message flow in the way you implement. You can instruct what message should be sent to a Microservice and what response should be expected from it and how the response should be used to send messages to other Microservices. A saga can persist its state in a durable storage which makes the sagas resistant to failures:



Sagas (Solution)

Although sagas can be implemented through several mechanisms, NHibernate provides an easy to use interface to build workflows and persist state in Azure storage.

Considerations

The following considerations should be applied while using this pattern:

- **Coordinate only**: Sagas are designed to coordinate the message flow. They make decisions through business logic. The actual work is delegated to the Microservices using the messages.
- **Message starting the Saga**: There might be more than one message that can start the saga.
- **Message order**: A delay in service response may result in messages arriving out of order. So, design the saga with fallacies of distributed computing in mind.

Related patterns

Following is the related cloud design patterns which can be used in conjunction with this pattern:

- **Compensating Transactions**: Sagas can be used to model compensating transactions so that messages are routed to compensators if work-step failed.

Use cases

Following are few use cases where this pattern will be a right fit:

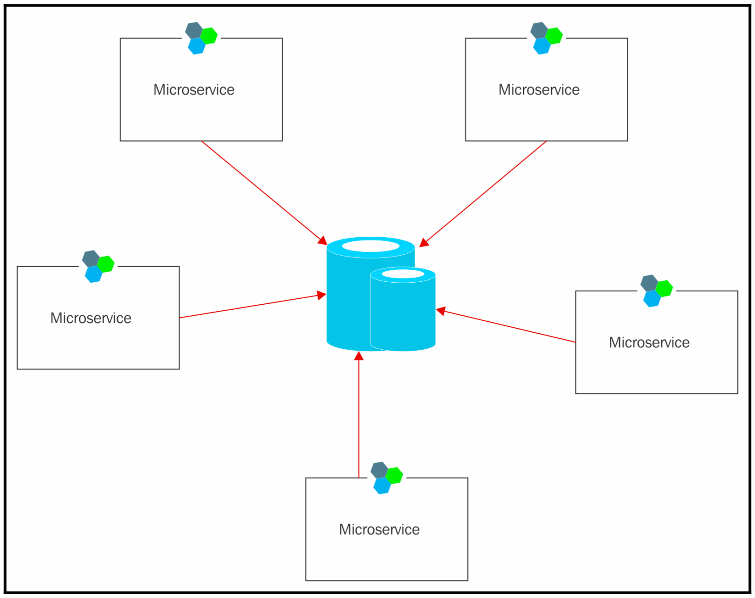
- **Process with multiple message roundtrips**: Any Microservice application that involves modeling a workflow involving processes that require interaction among several Microservices can benefit from this pattern.
- **Time related process requirements**: Designing Microservices that start processing once a certain period of time has lapsed, for example delay in approval.

Master Data Management

Problem

In a Microservice-based architecture, services are modeled as isolated units that manage independent activities. However, fully functional systems rely on the cooperation and integration of Microservices. Data sharing in a Microservice architecture has its own sets of problems such as:

- Handling big volumes of data
- Consistency guarantees while reducing data-access contention using simple locks
- Whether to share database for master data

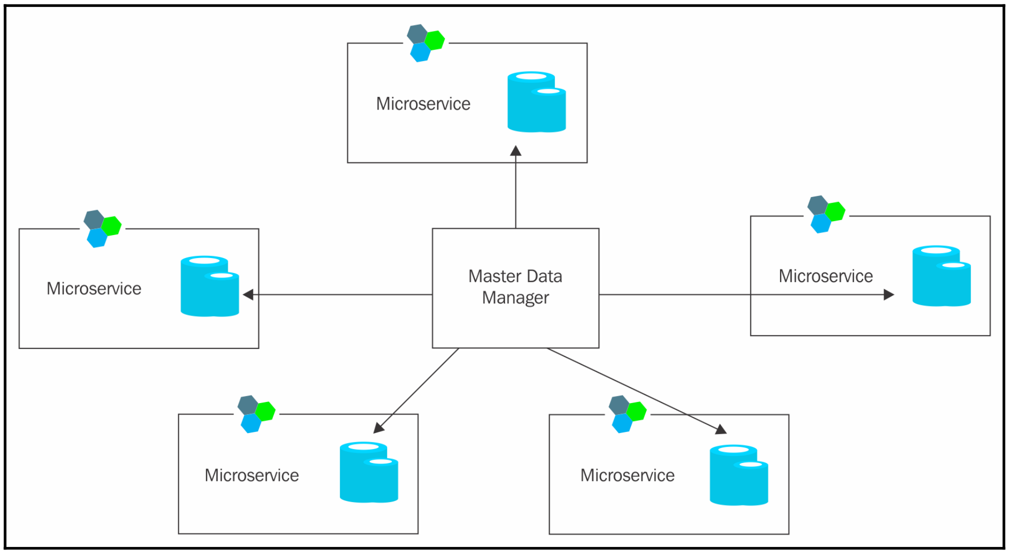


Master Data Management (Problem)

Solution

Large scale Microservice based applications such as Netflix use different databases for each Microservice. This approach helps the services stay independent of each other. A schema change in one of the service databases does not impact the rest of the services. This approach increases the complexity of data management as the systems may get out of sync or become inconsistent.

A custom or ready-made **Master Data Management** (**MDM**) tool should be used that operates in background to fix any inconsistencies. For example, an MDM tool can be used to find inconsistencies in customer ID across databases:



Master Data Management (Solution)

Considerations

The following considerations should be applied while using this pattern:

- **Eventual consistency**: Since MDM tools work in the background, only eventual consistency can be guaranteed by the system.
- **Conflict management**: In case inconsistency is found between records a conflict management strategy needs to be decided that would fix the inconsistent records.

- **Time constraint**: MDM Hub database needs to be loaded with data from various databases to detect inconsistencies.
- **Maintenance**: MDM implementation must incorporate tools, processes, and people to maintain the quality of the data. All data must have a data steward who is responsible for ensuring the quality of the master data.

Related patterns

Following is the related cloud design patterns which can be used in conjunction with this pattern:

- **ETL**: ETL is a type of data movement with possibly a complex acquisition from heterogeneous sources, and/or a complex manipulation with aggregation and cleansing, but always a simple write by overwriting any changes on the target.
- **Data Replication**: This pattern presents a special type of data movement (replication) with a simple acquisition and manipulation of the data, but possibly a complex write. The complexity of the write generally arises from the need to update both source and target and to eventually exchange the changes to the counterpart.

Use cases

Following are few use cases where this pattern will be a right fit:

- **Manage master data**: Inconsistency in shared data such as customer address may have a high business impact.
- **Merge other applications to an existing system**: In case of application mergers, data from other system needs to be merged into existing system. There might be cases where the same data is tagged with different identifiers in the individual application databases. An MDM implementation can help resolve these differences.

CQRS – Command Query Responsibility Segregation

Problem

State data in traditional applications is represented through a set of entities. The entities are stored in a single data repository against which two types of operations can take place.

- **Commands**: Operations that modify state
- **Queries**: Operations that read state

An operation cannot both update state and return data. This distinction of operations helps simplify understanding the system. The segregation of operations into commands and queries is called the **Command Query Separation** (**CQS**) pattern. The CQS pattern requires the commands to have void return type and the queries to be idempotent.

If a relational database such as SQL Server is used for storing state, the entities may represent a subset of rows in one or more tables in the database.

A common problem that arises in these systems is that both the commands and queries are applied to the same set of entities. For example, to update the contact details of a customer in a traditional e-commerce application, the customer entity is retrieved from the database and presented on the screen, which is a query operation. The entity is then modified and saved in the database through the data access layer (DAL), which is a command operation.

Although, this model works well for applications that have a limited set of business logic, this approach has certain disadvantages:

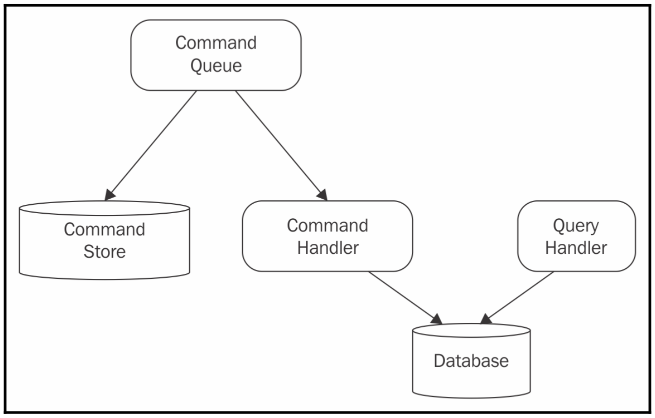
- There is a mismatch between the read and write representations of the data. While reading data, applications typically retrieve larger amount of data compared to writing that should affect one aggregate only.
- Reading data is a more frequent operation than writing. Since traditional applications use a single data repository, you can't scale read and write operations independently.

- Security and permissions are more cumbersome to manage in traditional applications because each entity is subject to both read and write operations, which might inadvertently expose data in the wrong context.
- Coupled representation of read and write models in traditional applications impede agility since changes in functionality result in high friction and incurs cost and time.

Solution

The CQRS pattern is like the CQS pattern with one major distinction – CQS pattern defines commands and queries as different methods within a type, whereas, the CQRS pattern defines commands and queries on different objects.

What this distinction means is that you can implement CQS in your applications by separating commands and queries as separate methods in a class, while retaining the same processing model to process both the models. CQRS, on the other hand completely separates the processing of commands and queries. Thus, the two patterns vary in scopes. The scope of the CQRS pattern is a bounded context, while the scope of CQS is a single class:



Structure of CQRS system

The preceding diagram presents the structure of a CQRS system. Users can submit commands to system which are stored in the command store and receive immediate feedback from the system. A user can check the status of progress of his commands through the user interface. A process will asynchronously move commands from the command store to the command queue, from where they will be picked by appropriate command handlers that will process the record and update the current state of data in the database.

Although not necessary, a CQRS system typically uses two separate physical databases for reads and writes so that they can be scaled independently. The read and write databases need not even have the same structure, for example, your application may write to a relational database such as SQL server whereas the reads operations are served through a document database such as MongoDB or Azure Cosmos DB. There are several approaches that can help synchronize the data in the read database and the write database. In case, the two databases have different structure, in case of failure of the read database, the state may be restored by replaying commands from the command store.

Responsibility of data validations is an important part of the CQRS pattern. The user interface should only be responsible for basic validations such as numeric values in phone number field, the business validations should be part of the domain logic and should be handled by the command handler.

Microservices in CQRS

The CQRS pattern can be realized through Microservices:

- The command store and command queue can be built through a Reliable Stateful Microservice. The service can forward the commands to appropriate command handlers. The command store can also be modelled as a Azure SQL database or as a Cosmos DB, whereas the command queue can be modelled as an Azure storage queue, a service bus queue or as a topic with a stateless reliable service acting as a broker between the two storages.
- The command handlers can be a separate Microservice. Each Microservices can handle the commands with its own logic.
- The query handler can be modelled as a separate Microservices. The command handler induces changes in the storage that the query handler queries on. These changes can be written to the read database by replicating the data written by the commands to the write database or by raising events using the Event Sourcing pattern.

Advantages

Following are a few of the advantages of this pattern:

- Splitting read operations and write operations to their own, Microservices help reduce its size. This in turn reduces the complexity of systems and helps improve scalability as you can scale the reads and writes independently.
- The read and write models can vary without affecting each other. This can help improve performance of read operations as you can denormalize the data or perform other operations to make read operations faster.
- You can run multiple versions of command handlers in parallel. This helps support deployment of new versions of Microservices.

Considerations

The following considerations should be applied while using this pattern:

- Transactions comprising read and write operations are hard to implement. Usually such operations are contained to an individual Microservice or require the use of other transaction mechanisms such as **2 Phase Commits** (**2PC**) or compensating transactions.
- CQRS is asynchronous by nature, therefore it is hard to ensure consistency across the various Microservices at any point of time.
- CQRS pattern might lead to higher development and infrastructure costs as multiple services need to be developed and deployed.

Related patterns

Following is the related cloud design patterns which can be used in conjunction with this pattern:

- **Event Sourcing**: This pattern describes in more detail how Event Sourcing can be used with the CQRS pattern to simplify tasks in complex domains; improve performance, scalability, and responsiveness; provide consistency for transactional data; and maintain full audit trails and history that may enable compensating actions.
- **Compensating Transaction**: This pattern can be used to reverse state changes caused by a faulty transaction.

Use cases

Following are few use cases where this pattern will be a right fit:

- **Collaborative data and complex business rules**: CQRS allows you to define commands with a sufficient granularity to minimize merge conflicts at the domain level. You can alter commands or queries independently of each other because their models are separate.
- **High performance reads and writes that can be tuned**: Scenarios where performance of data reads must be fine-tuned separately from performance of data writes, especially when the read/write ratio is very high, and when horizontal scaling is required.
- **Simplify business logic**: CQRS can be very helpful when you have difficult business logic. CQRS forces you to not mix domain logic and infrastructural operations.
- **Integration with other systems**: Especially in combination with Event Sourcing, where the temporal failure of one subsystem should not affect the availability of the others.

Event Sourcing

Problem

Traditional applications typically maintain the state of data by continuously updating the data as the user interacts with the application and keeps modifying the data. Continuous updates often involve transactions that lock the data under operation. Some of the problems with traditional applications are:

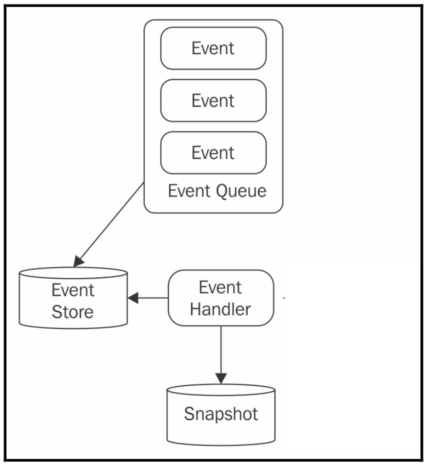
- Performing CRUD (Create, Read, Update, and Delete) operations directly against a data store impacts performance and responsiveness due to high processing overhead involved.
- In a collaborative domain, parallel updates on a single item of data may lead to conflicts.
- Traditional applications, in general, do not preserve history of operations performed on data and therefore there are no audit logs available unless maintained separately.

Solution

Event Sourcing models every change in the state of an application as an event object. The events are recorded in an append-only store. Application generates a series of events that capture the change that it has applied on data and these events are durably stored in sequence they were applied. The state itself is not saved in events, but it can be reconstructed by replaying the events.

The events are persisted in an event store that acts as the source of truth or system of record (the authoritative data source for a given data element or piece of information) about the current state of the data. The event store typically publishes these events so that consumers can be notified and can handle them if needed. Consumers could, for example, initiate tasks that apply the operations in the events to other systems, or perform any other associated action that is required to complete the operation. Notice that the application code that generates the events is decoupled from the systems that subscribe to the events.

Event Sourcing is like CQRS in approach. However, unlike commands in CQRS which define what is to be changed in an object, events contain data about a change that has been applied on state. Generally, CQRS and Event Sourcing work together. A command can change the data and the change can generate events to which other components of system can react. A practical example of this pattern is bank account statements, in which you can see the modification of account balance (state) by various transactions (events). Another practical example of application of this pattern is version control systems such as Git:



Event Sourcing (Problem)

The preceding diagram presents a high-level architecture of Event Sourcing pattern. The Event Queue can be implemented with messaging oriented middleware, which is a software or hardware infrastructure supporting the sending and receiving of messages between distributed systems. The Event Queue can receive events and send it to the Event Store.

The Event Store is an append-only store that receives all events from the Event Queue and appends them to the existing list of events. The event store can be used to reconstruct the state in case of failure or when you want to replay the events leading up to a desired state.

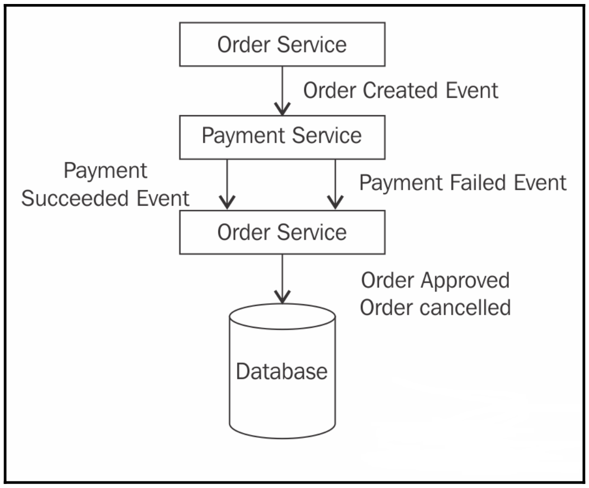
The Event Store can publish the saved events to which one or more Event Handlers can subscribe. The Event Handlers contain business logic which reacts to events. They can invoke commands that can make changes to the state. For example, a seat booking event handler can reserve seats in a scheduled flight and decrements the number of available seats.

Replaying events from beginning every time can lead to slow performance. Therefore, application can maintain Snapshots which contain the current state. Periodically the data in snapshot can be updated to stay in line with the new events. However, snapshot is an optional feature and may not be implemented if the application does not generate a lot of events.

To apply Event Sourcing pattern, look out for scenarios that involve auditing, reconstruction of historical state, or following a series of events such as bookings to an account.

Event Sourcing works well in concert with CQRS pattern. To implement this pattern through Microservices, it is recommended that your Microservices are built using the **Event Driven Architecture** (**EDA**).

EDA models the communication between Microservices through events. In this architecture, each service is an event receiver and an event publisher. A Microservice that is interested in the processing outcome of another service can register to receive events from that service. This allows for loose coupling between services and easy modification of the workflow without affecting multiple Microservices. For example, for an e-commerce, this architecture would model the workflow as follows:

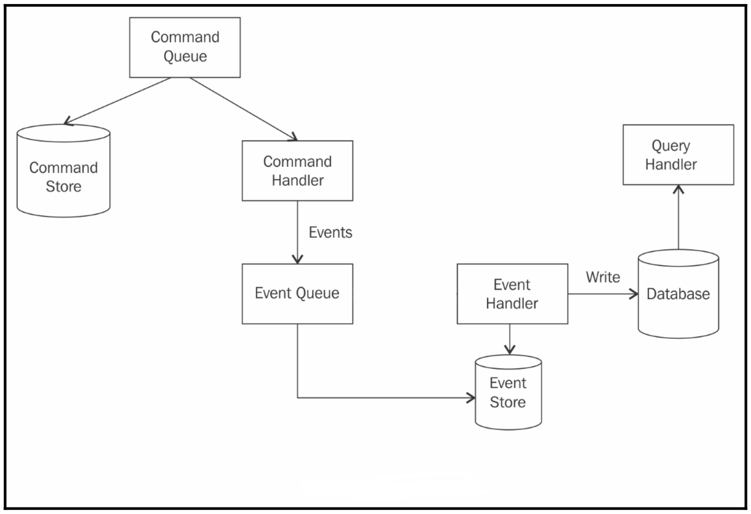


Event-driven architecture

- The Order Service creates an Order in a pending state and publishes an OrderCreated event.

- The Payment Service receives the event and attempts to charge payment instrument for the order received. It then publishes either a PaymentSucceeded event or a PaymentFailed event.
- The Order Service receives the event from the Payment Service and changes the state of the order to either approved or cancelled.

Following is a diagram of how CQRS and Event Sourcing patterns can be combined:



Event Sourcing with CQRS

Considerations

The following considerations should be taken into account while implementing this pattern:

- A common problem faced with this pattern is concurrency. If more than one event arrives concurrently and are processed by different event handlers, then optimistic locking should be used to ensure that the state is consistent. To implement optimistic locking, the following flow should be followed for handling a request. To understand the mechanism in detail let's consider a flight seat reservation scenario:
    1. The user interface raises a command to reserve two seats. A command handler handles the command raises the BookSeats event which gets loaded in the Event Queue and saved in Event Store.

2. The seat availability aggregate is populated by either querying all events in the Event Store. This aggregate is given a version identifier say X.
3. The aggregate performs its domain logic and raises the SeatsReserved event.
4. Before saving the SeatsReserved event, the event store verifies the version of the aggregate that raised the event. In case of concurrency, the version of the aggregate in store would be higher than the version of current aggregate.

Thus, you can avoid double booking by checking for versions before committing changes.

- Events cannot be changed. To correct the state appropriate events must be raised that compensate for the change in state.
- Event Sourcing pattern is based on domain driven design principles and therefore should follow terminology of the Ubiquitous Language. The names of events should be sensible in business context.
- Since events are basically messages, you need to keep standard messaging issues in consideration:
    - **Duplicate Messages**: Your command handlers should be idempotent to handle duplicate messages so that they don't affect the consistency of data.
    - **Out of Order Messages**: Choose FIFO message stores such as Service Bus Queues or Service Fabric Reliable Queue Collection to store events if you require messages to get stored in sequence.
    - **Poison Messages**: Messages that repeatedly cause failure should be marked as failed and sent to a dead-letter queue for further investigation.

Related patterns

Following is the related cloud design patterns which can be used in conjunction with this pattern:

- **CQRS**: Event Sourcing generally works in conjunction with CQRS pattern.
- **Message Broker for Communication**: Event Sourcing involves heavy communication between event store and event handlers which requires the use of a distributed messaging infrastructure.

Use cases

Following are few use cases where this pattern will be a right fit:

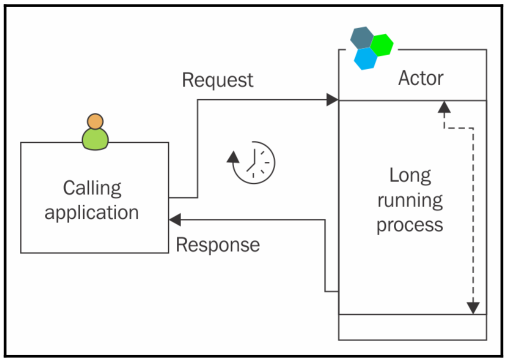
- **Event Driven Applications (EDA)**: Applications that use events to communicate with each other are a good fit for Event Sourcing.
- **Auditing**: Event Sourcing supports the feature of auditing without requiring any changes to be made in the application to support it.
- **Derive business value from event history**: Certain applications such as banking applications need to preserve history of events to determine the existing state. Preserving events can help answer historical questions from the business about the system.

Remindable Actors

Problem

An Actor based application may need to send several messages to an Actor instance. For example, an e-commerce application may require an Actor instance to update the user shopping cart. The Actor instance might be busy completing the process of adding a previous item to the cart. Since, the Actor operations are single threaded, the application will need to wait for the Actor to finish its previous operation to proceed with the next one.

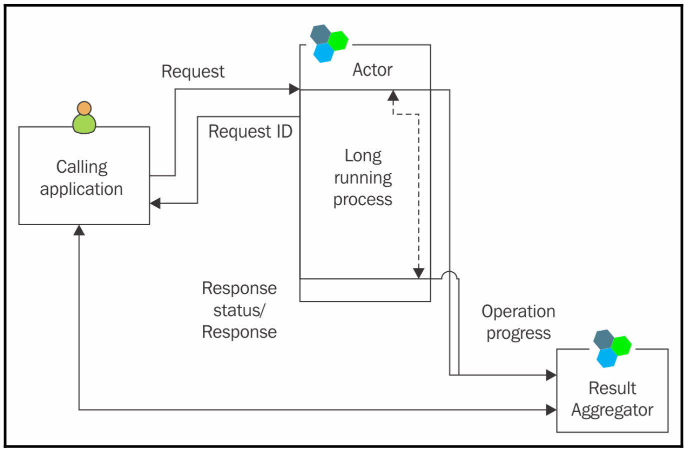
The wait to complete operations degrades performance of application and affects user experience:



Remindable Actor (Problem)

Solution

All Actors in the system should message themselves when they receive a request. In the above example, the shopping cart Actor can accept the request to add more items to the shopping cart and send a message to itself with the details of item to add to the shopping cart. Once the Actor instance is done processing the previous request, it can consume a message from the queue and add the item to the shopping cart:



Remindable Actor (Solution)

Considerations

The following considerations should be taken into account while implementing this pattern:

- The Actors should be fault tolerant and be able to remove poison messages from the queue.
- The Actor operations should be idempotent.
- Reminders are not an alternative to scaling and partitioning. A single Actor instance with a lot of pending messages to process can't benefit from additional compute.

Related patterns

Following is the related cloud design patterns which can be used in conjunction with this pattern:

- **Compensating Transaction Pattern**: If an Actor is unable to complete operation, the supervisor Actor may message other subordinates to undo any work that they have previously performed.
- **Circuit Breaker Pattern**: An Actor can use this pattern to handle faults that may take a variable amount of time to rectify when connecting to a remote service or resource.

Use cases

Following are few use cases where this pattern will be a right fit:

- For releasing the application to invoke other Actors or perform other operations while an Actor instance is busy completing the previous operations.
- To improve efficiency of distributed systems that communicates via messaging.

Managed Actors

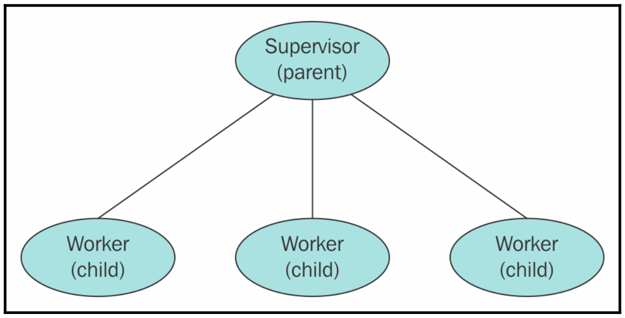
Problem

A task in an application composed of Microservices can comprise of several steps. The individual steps may be independent of each other but they are orchestrated by application logic that implements the task.

Application should ensure that all tasks related to an operation run to completion. This may involve resolving intermittent failures such as interrupted communication, temporary unavailability of remote resource and so on. In case of permanent failures; the application should restore the system to a consistent state and ensure integrity of end-to-end operation.

Solution

In the Actor programming model, an Actor can spawn other Actors. In such a scenario, the parent Actors are known as the supervisor and the Actors that it spawns are called subordinates. The subordinates carry out individual tasks of an operation that the supervisor is assigned to perform. The hierarchy may extend to deeper levels, that is, a subordinate may further delegate tasks to its subordinates and itself act as supervisor for those subordinates:



Managed Actors (Solution)

The role of supervisor is to aggregate the responses of the subordinates and report it to its parent, which may be the application itself or another supervisor.

In case of failures of one of the subordinates, the subordinate suspends its operations and reports the failure to its supervisor. The supervisor is responsible for managing failures. Based on the type of failure message, the supervisor can try to spawn the Actor again or message other Actors to rollback their operations.

The obvious benefit of this pattern is performance. Since the subordinates run concurrently, the over-all time it takes for the slowest subordinate to respond drives the performance of the system.

Another benefit of the pattern is the low cost of adding tasks to an operation. For example, to add a new task to an operation, you can add an independent subordinate to the hierarchy. The addition of new subordinate will not add latency to the system as the overall response time is still the response time of the slowest subordinate. You can find this pattern used widely in Netflix and LinkedIn.

Combining this pattern with Remindable Actors pattern can add resiliency and fault tolerance to your application.

The supervisor after sending a message to a subordinate can send a scheduled timeout reminder message to itself. If the subordinate responds before the message appears, then the message is discarded. If the response is not received or received after the timeout message appears, the missing information is either ignored or substituted with a default value. Thus, a failing service or a slow responding service will not break the application.

Considerations

The following considerations should be taken into account while implementing this pattern:

- The implementation of the pattern is complex and requires testing multiple failure scenarios.
- In case of failures, the supervisor may execute the Actors more than once. The logic of each of the subordinates should be idempotent.
- To recover from failure, the supervisor may ask the subordinates to reverse their operations. This may be a complex objective to achieve.

Related patterns

Following are the related cloud design patterns which can be used in conjunction with this pattern:

- **Compensating Transaction pattern**: If a subordinate is unable to complete operation, the supervisor Actor may message other subordinates to undo any work that they have previously performed.
- **Remindable Actors pattern**: A supervisor can request subordinate instances to queue tasks to be performed by them by messaging the task to themselves. This frees the supervisor to allocate tasks to other subordinates. This pattern also helps supervisor monitor latency of subordinates as previously discussed.
- **Circuit Breaker pattern**: A subordinate can use this pattern to handle faults that may take a variable amount of time to rectify when connecting to a remote service or resource.

Use cases

Following are few use cases where this pattern will be a right fit:

- For tasks that can operate in parallel without requiring input from previous tasks
- To improve efficiency of distributed systems that communicate via messaging

Summary

In this chapter we went through several patterns addressing problems related to challenges such as Optimization, Operation, and Implementation of enterprise grade Microservices.

In the next chapter, we will discuss how we can secure and manage Microservices deployed on Service Fabric using tools such as PowerShell.

9
Securing and Managing Your Microservices

Security is an integral part of your Microservices architecture. Due to many services at play in a Microservices application, the exploitable surface area of the application is higher than traditional applications. It is necessary that organizations developing Microservices adopt the Microsoft **Security Development Lifecycle** (**SDL**).

Using the SDL process, developers can reduce the number of vulnerabilities in software while shipping it using agile methods. At its core, SDL defines tasks which can be mapped to the agile development process. Since SDL tasks do not realize functional objectives, they don't require a lot of documentation.

To implement SDL in conjunction with agile methodology, it is recommended that SDL tasks be divided into three categories:

1. **Every sprint requirements**: The SDL tasks in this category are important to implement therefore they need to be completed in every sprint. If these tasks are not completed, then the sprint is not deemed complete and the product cannot ship in that sprint. A few examples of such tasks include:
    - Run analysis tools daily or per build
    - Threat model all new features
    - Ensure that each project member has completed at least one security training course in the past year

2. **Bucket requirements**: The SDL tasks in this category are not required to be completed in every sprint. The tasks in this category are categorized into multiple buckets, and tasks from each bucket may be scheduled and completed across sprints. An example of such a classification is as follows:
    - **Verification tasks**: This category includes mostly fuzzers and other analytics tools and may include tasks such as BinScope analysis, ActiveX fuzzing, and so on
    - **Design review tasks**: This category includes tasks such as privacy reviews, cryptography design reviews, and so on
    - **Response planning**: This category includes tasks such as defining the security bug bar, creating privacy support documents, and so on

   Note that the number of tasks in the various categories may vary across projects and therefore need to be uniquely tailored for each project that is undertaken.

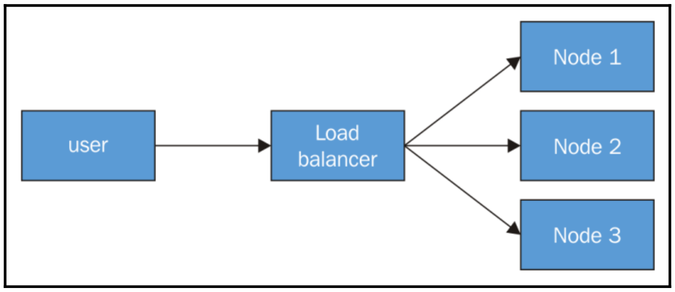
3. **One-time requirements**: These requirements need to be met only once in the lifetime of the project. These requirements are generally easy and quick to complete and are generally carried out at the beginning of the project. Even though these requirements are short and easy to accomplish, it may not be feasible to complete the requirements within an agile sprint as the team needs to deliver on functional requirements as well. Therefore, a grace period is assigned to each task in this category within which each task must be completed, which may vary depending on the complexity and size of the requirement. A few tasks that this category includes are adding or updating privacy scenarios in the test plan, creating or updating the network down plan, defining or updating the security bug bar, and so on.

You can read more about SDL by visiting the following link: `https://www.microsoft.com/en-us/SDL`.
SDL-Agile, which is an extension of SDL, is documented at: `https://www.microsoft.com/en-us/SDL/Discover/sdlagile.aspx`.

Securing the communication channels

Your Service Fabric application is deployed on the Service Fabric cluster, which is a set of federated nodes that talk to each other over the network.

The generic structure of your Service Fabric cluster looks like the following:



Generic structure of Service Fabric cluster

As you can see, there are two types of communication channels that are used by the Service Fabric:

- Channel to enable communication between cluster nodes (node-node)
- Channel to enable communication between clients and nodes (client-node)

Although you can choose not to secure the communication channels, by using open channels anonymous users can connect to the cluster and perform management operations on it and alter the cluster. Security cannot be an afterthought of implementation as an unsecured Service Fabric cluster cannot be secured later. Therefore, you should consider security of your cluster prudently.

Inter-node communication

Inter-node security ensures that only the authorized nodes can join a cluster and host applications. The nodes in a cluster running on Windows Server can be secured using either certificate security or Windows security. Let's discuss how we can secure a cluster using certificates so that only authorized nodes can join the cluster and only authorized cluster management commands can execute against the cluster.

While deploying the earlier examples, you must have noticed that we used to omit cluster security settings to simplify development. However, at this step you can specify the X.509 certificate that Service Fabric should use to secure communication between nodes. You can specify the certificate to use through any medium you wish to create your Service Fabric cluster, that is the Azure Portal, ARM template, or standalone JSON template. Also, you can specify up to two certificates in the configurations, one of which is the primary certificate used to secure the cluster, and the second certificate is an optional secondary certificate that can be used for certificate rollovers.

For the walkthrough, we will use a self-signed certificate; however, use of a certificate from a trusted **certificate authority** (**CA**) is advised for production workloads:

1. Create a new self-signed certificate. You can use PowerShell, tools such as OpenSSL, or any other tool that you personally prefer for the purpose. I will use PowerShell to generate a new self-signed certificate and save it on my desktop:

```
$certificatePassword = ConvertTo-SecureString
 -String [plain text password] -AsPlainText -Force
  New-SelfSignedCertificate -CertStoreLocation
Cert:\CurrentUser\My -DnsName [your cluster DNS name]
-Provider 'Microsoft Enhanced Cryptographic Provider v1.0' |
Export-PfxCertificate -FilePath
([Environment]::GetFolderPath('Desktop')+
 '/ClusterCertificate.pfx') -Password $certificatePassword
```

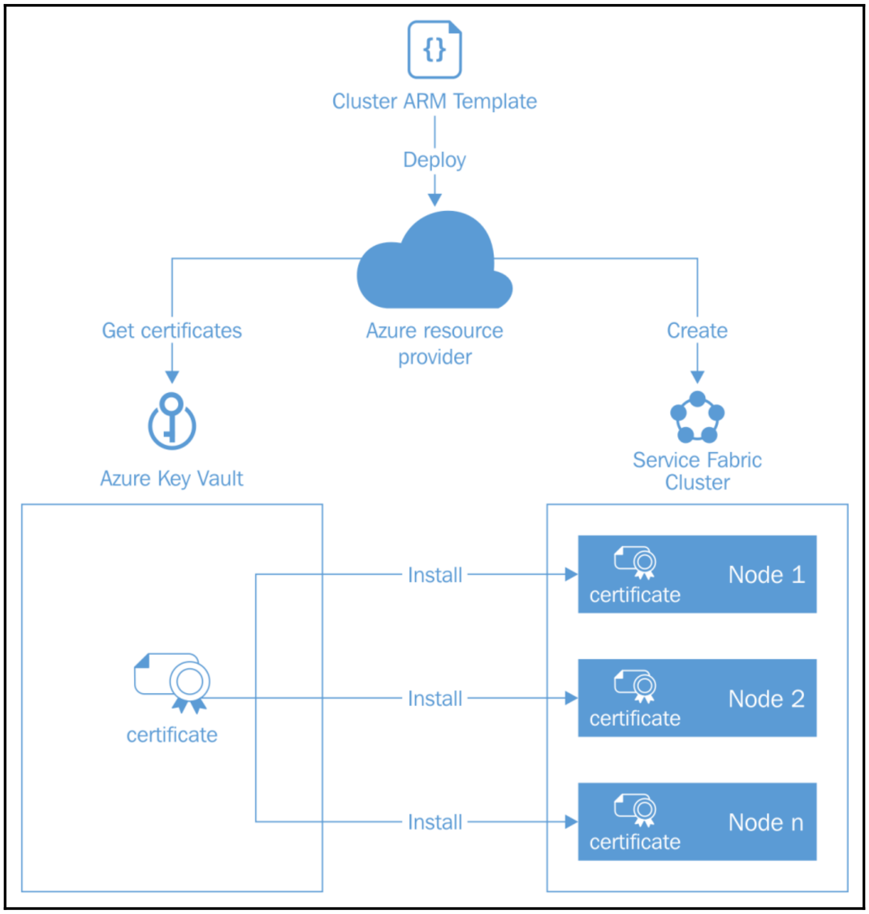
2. In the next step, we will sign in to our Azure subscription to perform the remaining operations:

```
Login-AzureRmAccount
Get-AzureRmSubscription
Set-AzureRmContext -SubscriptionId [azure subscription id]
```

3. Now let's create a resource group for our Service Fabric cluster and other resources that we will use:

```
New-AzureRmResourceGroup -Name [resource group name]
  -Location [resource group location]
```

4. We will use a cloud-based certificate store to store our certificate and provide it to our Service Fabric cluster. **Azure Key Vault** is a service that manages keys and passwords in the cloud. Key Vault helps decouple sensitive information such as passwords and certificates from applications. You may use an existing Key Vault but ideally, your Key Vault should reside in the same region and resource group for easy manageability and performance. The certificate added to the Key Vault will be installed on all the nodes in the cluster later through a configuration, as illustrated by the following diagram. This action can be accomplished through an ARM template, or by using a combination of PowerShell and Management Portal as we are doing currently:

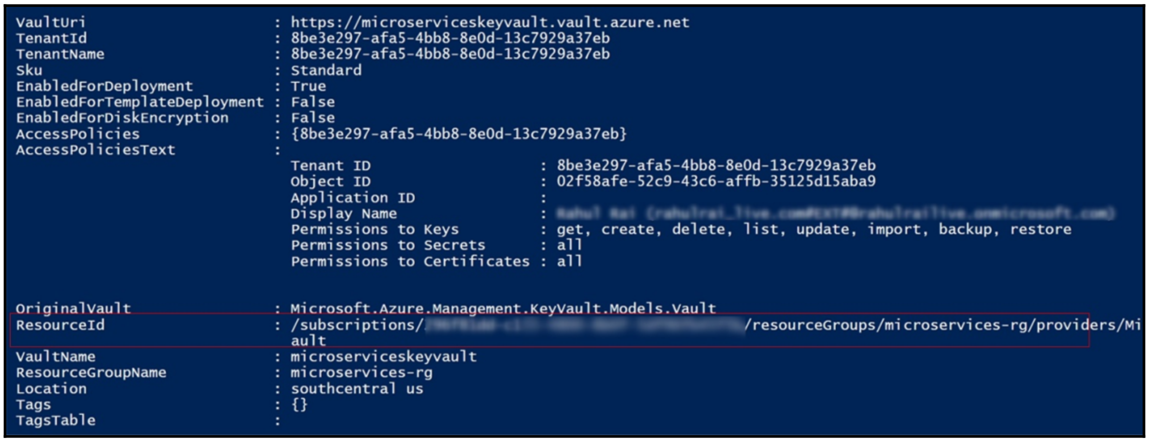


Applying security certificate on Service Fabric cluster from Key Vault

The following command will create a new Key Vault instance for you:

```
New-AzureRmKeyVault -VaultName [name of key vault]
 -ResourceGroupName [name of resource group]
 -Location [key vault location] -EnabledForDeployment
```

5. The execution of the preceding command will generate a summary of the generated Key Vault instance. You can copy the `ResourceId` of the generated instance to apply it on Service Fabric cluster configuration later. It is of the following format `/subscriptions/[subscription id]/resourceGroups/[resource group name]/providers/Microsoft.KeyVault/vaults/[key vault name]`:



```
VaultUri                     : https://microserviceskeyvault.vault.azure.net
TenantId                     : 8be3e297-afa5-4bb8-8e0d-13c7929a37eb
TenantName                   : 8be3e297-afa5-4bb8-8e0d-13c7929a37eb
Sku                          : Standard
EnabledForDeployment         : True
EnabledForTemplateDeployment : False
EnabledForDiskEncryption     : False
AccessPolicies               : {8be3e297-afa5-4bb8-8e0d-13c7929a37eb}
AccessPoliciesText           :
                               Tenant ID                  : 8be3e297-afa5-4bb8-8e0d-13c7929a37eb
                               Object ID                  : 02f58afe-52c9-43c6-affb-35125d15aba9
                               Application ID             :
                               Display Name               : [redacted]
                               Permissions to Keys        : get, create, delete, list, update, import, backup, restore
                               Permissions to Secrets     : all
                               Permissions to Certificates : all


OriginalVault                : Microsoft.Azure.Management.KeyVault.Models.Vault
ResourceId                   : /subscriptions/[redacted]/resourceGroups/microservices-rg/providers/Mi
                               ault
VaultName                    : microserviceskeyvault
ResourceGroupName            : microservices-rg
Location                     : southcentral us
Tags                         : {}
TagsTable                    :
```

Resource ID of KeyVault instance

6. We will now upload our certificate that we previously stored on the desktop to our Key Vault instance:

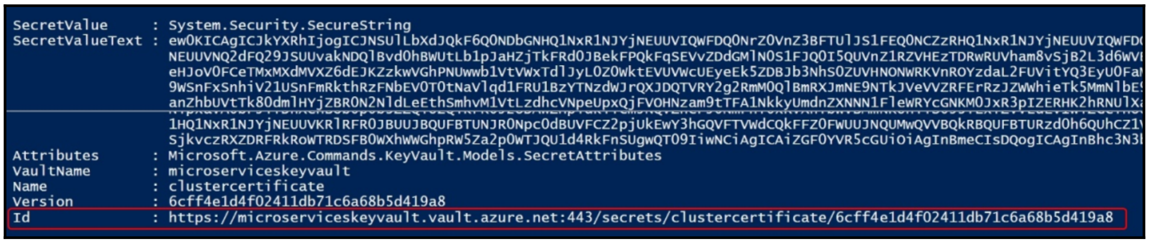
```
$cer = Add-AzureKeyVaultKey -VaultName [name of key vault]
  -Name [key name] -KeyFilePath
([Environment]::GetFolderPath('Desktop')+
'/ClusterCertificate.pfx') -KeyFilePassword $certificatePassword
```

7. You will now need to create a secret based on the certificate. For this purpose, we will create a JSON payload using the contents of the `.pfx` file:

```
$bytes = [System.IO.File]::ReadAllBytes(([Environment]:
         :GetFolderPath('Desktop')+'/ClusterCertificate.pfx'))
$base64 = [System.Convert]::ToBase64String($bytes)
$jsonBlob = @{
   data = $base64
   dataType = 'pfx'
   password = $password
   } | ConvertTo-Json
$contentbytes = [System.Text.Encoding]::UTF8.GetBytes($jsonBlob)
$content = [System.Convert]::ToBase64String($contentbytes)
```

```
$secretValue = ConvertTo-SecureString -String $content
  -AsPlainText -Force
 Set-AzureKeyVaultSecret -VaultName [name of key vault]
 -Name [key name] -SecretValue $secretValue
```

8. The `Set-AzureKeyVaultSecret` command returns a summary of the generated secret. From the summary, copy the ID of the secret:



```
SecretValue     : System.Security.SecureString
SecretValueText : ewOKICAgICJkYXRhIjogICJNSUlLbXdJQkF6Q0NDbGNHQ1NxR1NJYjNEUUVIQWFDQ0NrZ0VnZ3BFTUlJS1FEQ0NCZzRHQ1NxR1NJYjNEUUVIQWFD
                  NEUUVNQ2dFQ29JSUUvakNDQlBvd0hBWUtLb1pJaHZjTkFRd0JBekFPQkFqSEVvZDdGMlN0S1FJQ0I5QUVnZ1RZVHEzTDRwRUVham8vSjB2L3d6WV
                  eHJoV0FCeTMxMXdMVXZ6dEJKZzkwVGhPNUwwb1VtVWxTdlJyL0Z0WktEVUVWcUEyeEk5ZDBJb3NhS0ZUVHNONWRKVnROYzdaL2FUVitYQ3EyU0Fa
                  9WSnFxSnhiV21USnFmRkthRzFNbEV0T0tNaVlqd1FRU1BzYTNzdWJrQXJDQTVRY2g2RmM0QlBmRXJmNE9NTkJVeVVZRFErRzJZWWhieTk5MmNlbE
                  anZhbUVtTk80dmlHYjZBR0N2NldLeEthSmhvM1VtLzdhcVNpeUpxQjFVOHNzam9tTFA1NkkyUmdnZXNNN1FleWRYcGNKM0JxR3pIZERHK2hRNUlX
                  [illegible]
                  1HQ1NxR1NJYjNEUUVKRlRFR0JBUUJBQUFBTUNJR0Npc0dBUVFCZ2pjUkEwY3hGQVFTVWdCQkFFZ0FWUUJNQUMwQVVBQkRBQUFBTURzd0h6QUhCZ1
                  SjkvczRXZDRFRkRoWTRDSFB0WXhWWGhpRW5Za2p0WTJQU1d4RkFnSUgwQT09IiwNCiAgICAiZGF0YVR5cGUiOiAgInBmeCIsDQogICAgInBhc3N3
Attributes      : Microsoft.Azure.Commands.KeyVault.Models.SecretAttributes
VaultName       : microserviceskeyvault
Name            : clustercertificate
Version         : 6cff4e1d4f02411db71c6a68b5d419a8
Id              : https://microserviceskeyvault.vault.azure.net:443/secrets/clustercertificate/6cff4e1d4f02411db71c6a68b5d419a8
```

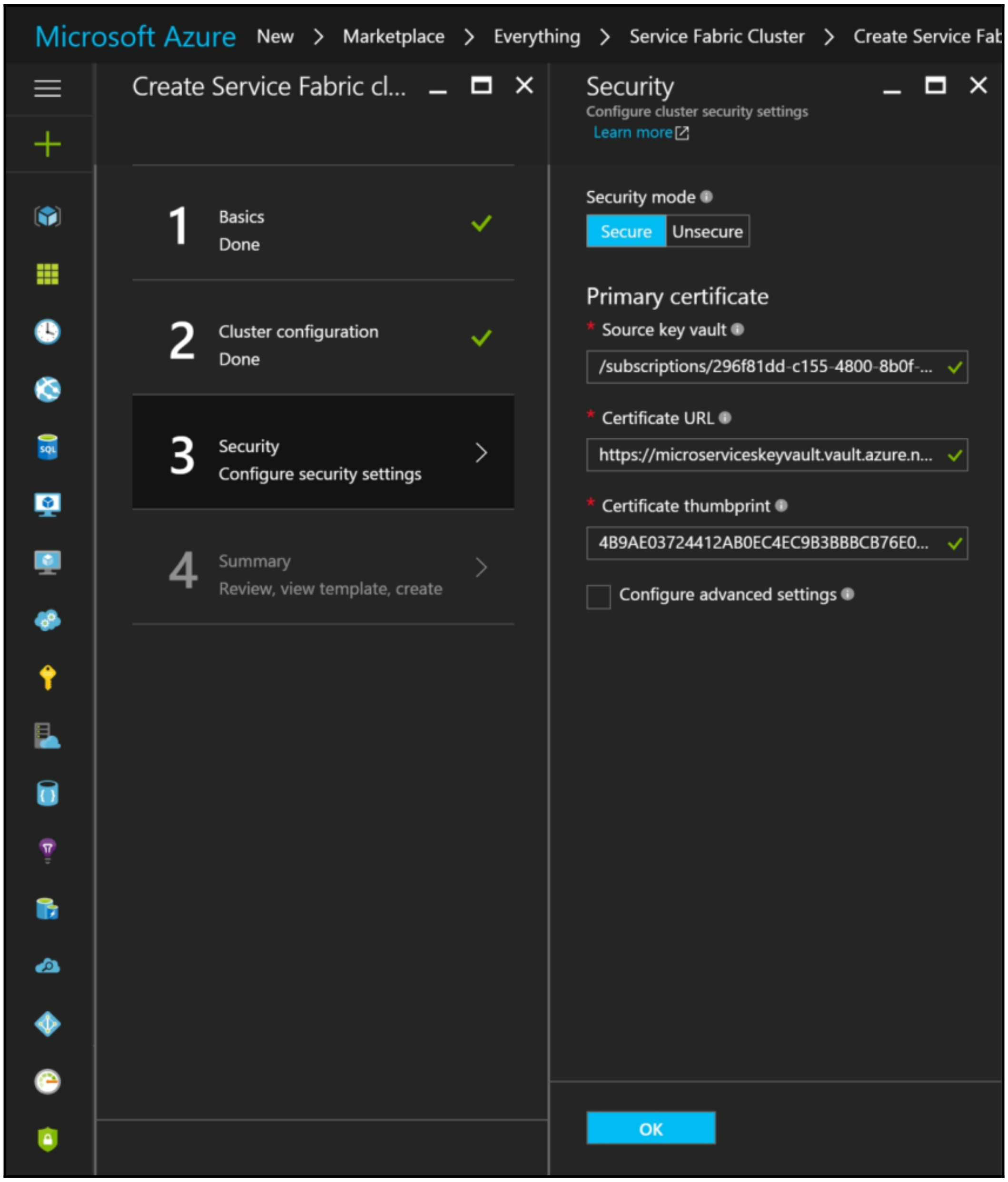
Id of Secret in KeyVault

9. You will also need the thumbprint of the certificate that you used. You can copy it from the certificate properties or use the following PowerShell command for the purpose:

```
$clusterCertificate = new-object
 System.Security.Cryptography.X509Certificates.
 X509Certificate2 ([Environment]::GetFolderPath('Desktop')+
 '/ClusterCertificate.pfx'),
$certificatePassword
$clusterCertificate.Thumbprint
```

10. Now, let's move to the Management Portal and create a new Service Fabric cluster. Populate the basic configuration options as before until you reach the security configuration step.

In the **Security** section, set the **Security mode** to **Secure**:


Microsoft Azure
New > Marketplace > Everything > Service Fabric Cluster > Create Service Fab
Create Service Fabric cl...
1 Basics Done
2 Cluster configuration Done
3 Security Configure security settings
4 Summary Review, view template, create
Learn more
OK

Security
Configure cluster security settings

Security mode
[✓ Secure] Unsecure

Primary certificate

* Source key vault: /subscriptions/296f81dd-c155-4800-8b0f-...

* Certificate URL: https://microserviceskeyvault.vault.azure.n...

* Certificate thumbprint: 4B9AE03724412AB0EC4EC9B3BBBCB76E0...

☐ Configure advanced settings

Service Fabric cluster security blade

11. You will not be asked to provide certificate details. Let's start populating the details one by one:
    - In the **Primary certificate** details section, set the value of **Source key vault** to the text that you generated in *step 5*
    - In the **Certificate URL** field, set the value to the secret ID that you copied in *step 8*
    - In the **Certificate thumbprint** field, set the value to the certificate thumbprint that you generated or copied from the PowerShell command output in *step 9*

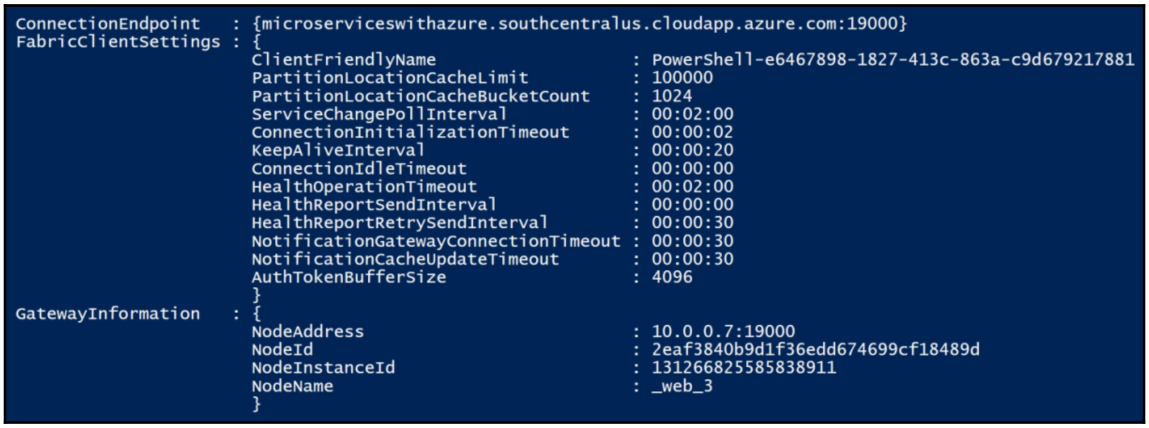
12. By now, you have successfully applied the security settings on the cluster. Proceed to complete the remaining settings and allow the cluster to provision.

After you have applied a certificate on a Service Fabric cluster, it can only be accessed over HTTPS. You will only be able to navigate to your application on the cluster over HTTPS, which might raise untrusted certificate warnings as we have used a self-signed certificate for this example. You can add the certificate to the **Trusted Root Certificate Authority** of the local computer store to make the warnings go away.

To connect to your Service Fabric cluster, you would need to use the following command that adds the certificate to your request:

```
Connect-ServiceFabricCluster -ConnectionEndpoint ([your cluster DNS name] +
':19000')  -X509Credential -ServerCertThumbprint
4b9ae03724412ab0ec4ec9b3bbbcb76e0d5374a9 -FindType FindByThumbprint -
FindValue 4b9ae03724412ab0ec4ec9b3bbbcb76e0d5374a9 -StoreLocation
CurrentUser -StoreName My
```

This command will present you with a summary of the cluster as follows:



```
ConnectionEndpoint   : {microserviceswithazure.southcentralus.cloudapp.azure.com:19000}
FabricClientSettings : {
                       ClientFriendlyName                     : PowerShell-e6467898-1827-413c-863a-c9d679217881
                       PartitionLocationCacheLimit            : 100000
                       PartitionLocationCacheBucketCount      : 1024
                       ServiceChangePollInterval              : 00:02:00
                       ConnectionInitializationTimeout        : 00:00:02
                       KeepAliveInterval                      : 00:00:20
                       ConnectionIdleTimeout                  : 00:00:00
                       HealthOperationTimeout                 : 00:02:00
                       HealthReportSendInterval               : 00:00:00
                       HealthReportRetrySendInterval          : 00:00:30
                       NotificationGatewayConnectionTimeout   : 00:00:30
                       NotificationCacheUpdateTimeout         : 00:00:30
                       AuthTokenBufferSize                    : 4096
                       }
GatewayInformation   : {
                       NodeAddress                            : 10.0.0.7:19000
                       NodeId                                 : 2eaf3840b9d1f36edd674699cf18489d
                       NodeInstanceId                         : 131266825585838911
                       NodeName                               : _web_3
                       }
```

Summary of Service Fabric cluster

As you can see, now only authenticated clients can connect to your cluster; but additionally, you may want only authorized clients to make structural changes to your cluster, such as stopping nodes.

Let's discuss how we can authorize clients to perform operations on our cluster.

You can secure the inter node communication channel on a standalone Windows cluster using Windows security. You can read more about it at: `https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-windows-cluster-windows-security`.

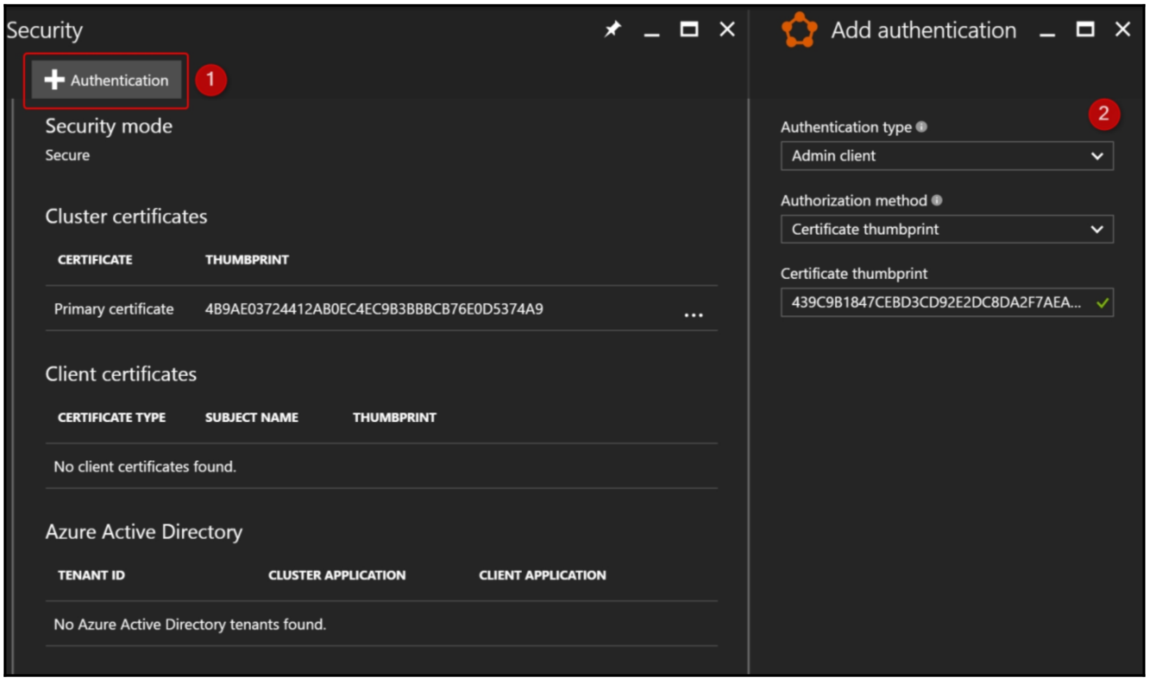
Client to node security

Previously, we talked about securing the communication between nodes of a Service Fabric cluster. To truly secure all the communication channels of Service Fabric, we need to secure.

Certificate security

Using X.509 certificates, you can configure your Service Fabric cluster to allow only authorized clients to execute management commands. You can set up certificates for two types of clients – the **admin client** which can perform administrative operations on your cluster, and the **read only client** which can perform only read operations on your cluster.

To specify the client certificate to use, you can use either the certificate thumbprint or the subject name of the certificate, which also requires the issuer thumbnail. To configure the client certificate, log on to the Management Portal and select your Service Fabric instance. Select the **Security** option and click on the **Authentication** button in the **Security** blade. Next, enter the details of the client certificate and allow the cluster updates to propagate:



Security mode
Secure

Cluster certificates

CERTIFICATE	THUMBPRINT
Primary certificate	4B9AE03724412AB0EC4EC9B3BBBCB76E0D5374A9

Client certificates

CERTIFICATE TYPE	SUBJECT NAME	THUMBPRINT
No client certificates found.		

Azure Active Directory

TENANT ID	CLUSTER APPLICATION	CLIENT APPLICATION
No Azure Active Directory tenants found.		

Authentication type: Admin client
Authorization method: Certificate thumbprint
Certificate thumbprint: 439C9B1847CEBD3CD92E2DC8DA2F7AEA...

Cluster authentication steps

Once the cluster finishes updating, you can connect to the cluster without using the cluster certificate:

```
Connect-ServiceFabricCluster
 -ConnectionEndpoint ([your cluster dns name] + ':19000') `
      -KeepAliveIntervalInSec 10 `
      -X509Credential -ServerCertThumbprint [cluster
             certificate thumbprint] `
      -FindType FindByThumbprint -FindValue [client
             certificate] `
      -StoreLocation CurrentUser -StoreName My
```

Note that since in the preceding step we provisioned the client certificate for an Admin client, the clients with this certificate will have full management capabilities of the cluster.

Azure Active Directory security

Clusters running on Azure can also secure access to the management endpoints using **Azure Active Directory** (**AAD**). For this purpose, you would need to create two AAD applications, one web application to enable authorized access to the web-based Service Fabric Explorer and one native application to enable authorized access to management functionalities through Visual Studio.

After provisioning these two applications, you can revisit the **Security** options of your cluster and add the tenant and application details in the **Authentication** blade.

Publishing an application to a secured cluster

To publish an application from Visual Studio, you only need to have the cluster security certificate added to `Cert:\Current User\My store` or `Cert:\Local Machine\My store`. Visual Studio automatically adds the following configuration to the published profile of your application:

```
<?xml version="1.0" encoding="utf-8"?>
<PublishProfile xmlns =" http:// schemas.microsoft.com/ 2015/ 05/
fabrictools" >
  <ClusterConnectionParameters
    ConnectionEndpoint ="[your cluster dns name]:19000"
    X509Credential =" true"
    ServerCertThumbprint =" [cluster certificate thumbprint]"
    FindType =" FindByThumbprint"
    FindValue =" [cluster certificate thumbprint]"
    StoreLocation =" CurrentUser"
    StoreName =" My" />
  <ApplicationParameterFile Path ="..\ ApplicationParameters\ Cloud.xml" />
</PublishProfile >
```

Managing Service Fabric clusters with Windows PowerShell

Microsoft Service Fabric is a distributed systems platform engineered for hosting hyperscale Microservices. The addressed challenges around managing Microservices are packaging, deployment, scaling, upgrading, and so on. Apart from the rich graphical user interface exposed through Azure Portal and the management APIs, Service Fabric clusters can also be managed using Windows PowerShell cmdlets. This is the preferred mechanism for automating the management process.

Prerequisites

Service Fabric supports automation of most of its application lifecycle management tasks. These tasks include deploying, upgrading, removing, and testing Azure Service Fabric applications. Following are the steps to prepare your machine to execute Windows PowerShell cmdlets to manage a Service Fabric cluster:

1. Install the Service Fabric SDK, runtime, and tools which include the PowerShell modules. The SDK currently supports Windows 7, Windows 8/8.1, Windows Server 2012 R2, and Windows 10 operating systems. Microsoft Web Platform Installer can be used to download and install Service Fabric SDK.
2. After the installation is complete, modify the execution policy to enable PowerShell to deploy to a local cluster. This can be done by running the following command from PowerShell in administrator mode:

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

3. Start a new PowerShell window as an administrator. Then, execute the cluster setup script. This script can be found in the following folder inside the SDK:

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

4. Use the following PowerShell cmdlet to connect to the local Service Fabric cluster:

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

This completes the setup for PowerShell to manage your local Service Fabric cluster. Now we can explore deploying, upgrading, testing, and removing applications to this cluster.

Deploying Service Fabric applications

After an application is built, it needs to be packaged for it to be deployed on a Service Fabric cluster. This task can be easily achieved using Visual Studio. The packaged application should be uploaded and registered before the Service Fabric cluster manager can instantiate the application. Following are the steps involved:

1. **Uploading an application**: This task is required as it places the application packages in a shared location from which the Service Fabric components can access it. An application package typically contains the following:
    - Application manifests
    - Service manifests
    - Code
    - Configuration
    - Data packages

   The following cmdlet can be used to upload an application package for deployment:

   ```
   Copy-ServiceFabricApplicationPackage <Source package folder>
   -ImageStoreConnectionString file:<Local image store path>
   -ApplicationPackagePathInImageStore <Application package name>
   ```

2. **Registering the application package**: Package registration makes the application type and version declared in the manifest available for use. Service Fabric reads the registered package, verifies it, and moves the package to an internal store for further processing. Application packages can be registered using the following cmdlet:

   ```
   Register-ServiceFabricApplicationType <Application name>
   ```

   All registered packages in the cluster can be listed using the following cmdlet:

   ```
   Get-ServiceFabricApplicationType
   ```

3. **Instantiating the application**: The `New-ServiceFabricApplication` cmdlet can be used to instantiate a specific version of a registered application type. The application name should be declared at the deploy time using the following scheme:

   ```
   Fabric:<Application name>
   ```

   - The following usage of the preceding cmdlet can be used to create a new application instance:

     ```
     New-ServiceFabricApplication fabric:/<Application name>
     <Application type> <Application version>
     ```

   - The following cmdlet can be used to list all the application instances which are successfully created in a cluster:

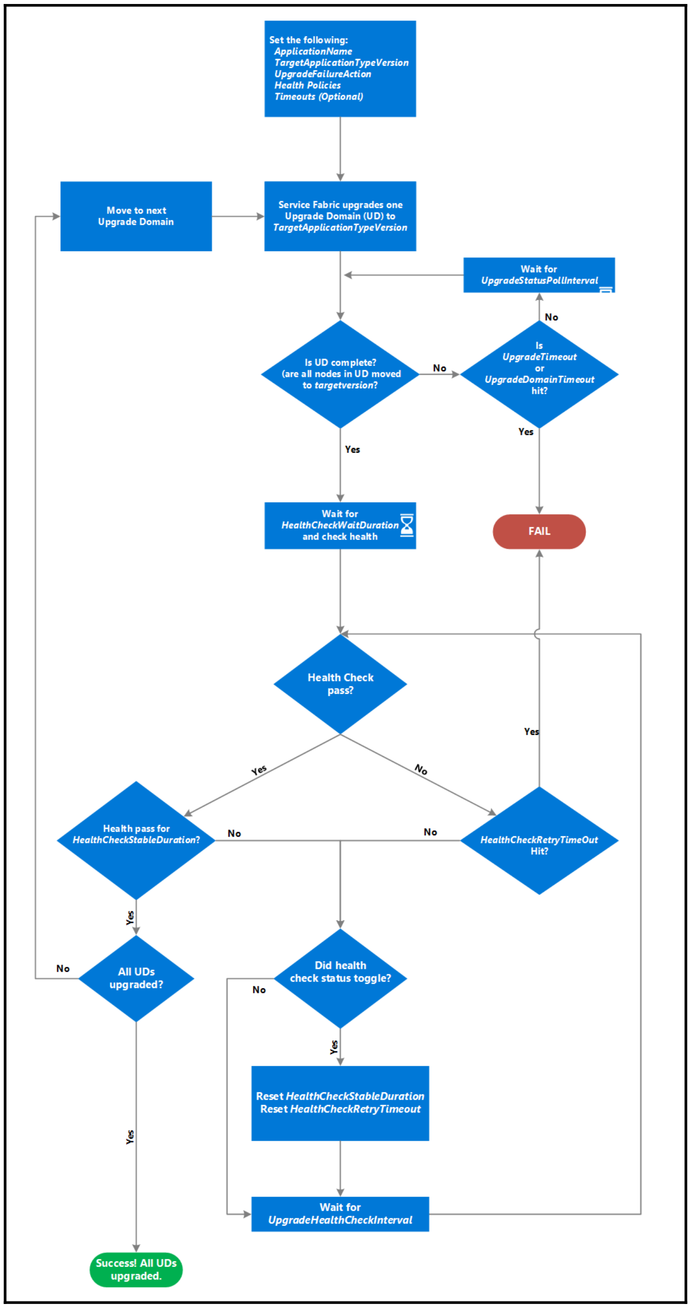
     ```
     Get-ServiceFabricApplication
     ```

   - The preceding command can be used along with the `Get-ServiceFabricService` cmdlet to list the services running under each application instance. Following is the usage:

     ```
      Get-ServiceFabricApplication | Get-ServiceFabricService
     ```

Upgrading Service Fabric applications

To perform the upgrade, Service Fabric performs a comparison of the old and the new application manifest. The application is upgraded only if there is a change in version number. The following flowchart explains the application upgrade process:


Set the following:
ApplicationName
TargetApplicationTypeVersion
UpgradeFailureAction
Health Policies
Timeouts (Optional)
Move to next Upgrade Domain
Service Fabric upgrades one Upgrade Domain (UD) to TargetApplicationTypeVersion
Wait for UpgradeStatusPollInterval
No
Is UD complete? (are all nodes in UD moved to targetversion?
No
Is UpgradeTimeout or UpgradeDomainTimeout hit?
Yes
Yes
Wait for HealthCheckWaitDuration and check health
FAIL
Health Check pass?
Yes
Yes
No
Health pass for HealthCheckStableDuration?
No
No
HealthCheckRetryTimeOut Hit?
Yes
No
All UDs upgraded?
Did health check status toggle?
No
Yes
Reset HealthCheckStableDuration
Reset HealthCheckRetryTimeout
Yes
Wait for UpgradeHealthCheckInterval
Success! All UDs upgraded.

Application upgrade process

An already deployed Service Fabric can be upgraded to a newer version using PowerShell cmdlets. Following are the steps to be performed to upgrade a Service Fabric application:

1. **Package the application**: The first step for upgrading is to package the newer version of the application. This can be done using Visual Studio.
2. **Upload package**: Once packaged, the package needs to be uploaded into the Service Fabric image store. The following PowerShell cmdlet can be used to perform this operation:

   ```
   Copy-ServiceFabricApplicationPackage <Package folder path>
   -ImageStoreConnectionString file:<Local image store path>
   -ApplicationPackagePathInImageStore <Application package name>
   ```

   The parameter `ApplicationPackagePathInImageStore` informs the Service Fabric orchestrator about the location of the application package.

3. **Register the updated application**: The uploaded package now should be registered for Service Fabric to use it. This can be done using the `Register-ServiceFabricApplicationType` cmdlet:

   ```
   Register-ServiceFabricApplicationType <Application name>
   ```

4. **Perform the upgrade**: Upgrading the application without downtime and handling failures gracefully is a key feature of Service Fabric. This process operates on various configurations set by the user such as timeouts, health criteria, and so on Parameters like `HealthCheckStableDuration`, `UpgradeDomainTimeout`, and `UpgradeTimeout` need to be set for Service Fabric to perform the upgrade. The `UpgradeFailureAction` also should be set by the user so that Service Fabric knows what to do if an upgrade operation fails. Once these parameters are decided, the upgrade process can be kicked off by using the `Start-ServiceFabricApplicationUpgrade` cmdlet:

   ```
    Start-ServiceFabricApplicationUpgrade -ApplicationName
    fabric:/<Application name>-ApplicationTypeVersion <New app
    version>
   -HealthCheckStableDurationSec <Duration>
   -UpgradeDomainTimeoutSec <Timeout>
   -UpgradeTimeout <Timeout> -FailureAction <Action> -Monitored
   ```

   The application name in the preceding command must match the one specified for the already deployed application, as Service Fabric uses this name to identify the application.

5. **Check upgrade**: Service Fabric Explorer can be used to monitor the upgrade process for applications. The `Get-ServiceFabricApplicationUpgrade` cmdlet can also be used for this purpose:

   ```
   Get-ServiceFabricApplicationUpgrade fabric:/<Application
    Name>
   ```

Removing Service Fabric applications

Service Fabric supports removing an application instance of a deployed application, removing the application itself, and clearing the application package from the image store for permanent deletion.

Following are the steps to be performed to completely remove the application from a Service Fabric cluster:

1. **Removing application instances**: The following cmdlet can be used to remove application instances:

   ```
   Remove-ServiceFabricApplication fabric:/<Application name>
   ```

2. **Unregistering the application**: The `Unregister-ServiceFabricApplicationType` cmdlet can be used to unregister an application type. Following is the usage:

   ```
   Unregister-ServiceFabricApplicationType <Application name>
     <version number>
   ```

3. **Removing the application package**: The `Remove-ServiceFabricApplicationPackage` cmdlet can be used to remove an application package from the image store:

   ```
   Remove-ServiceFabricApplicationPackage
   -ImageStoreConnectionString file:<Image store path>
   -ApplicationPackagePathInImageStore <Application Name>
   ```

Summary

We started off this chapter by detailing how the security of applications starts with **Secure Development Lifecycle** (**SDL**). We then went on to discuss the channel and node security of a Service Fabric cluster and how you can configure those using PowerShell and the Azure Management Portal. We later walked through the steps of deploying an application on a secure cluster.

In the next part of the chapter, we walked through the process of managing a Service Fabric cluster. We worked with multiple PowerShell cmdlets that help you manage your Service Fabric cluster.

10
Diagnostics and Monitoring

In an enterprise-scale Microservice deployment, your efficiency in managing services depends upon your ability to respond to outages and incidents. This in turn is heavily dependent on your ability to detect and diagnose issues quickly. The answer to the problem is to have an efficient monitoring and diagnostics solution in place. The fact that your services are hosted on commodity hardware, which is prone to failures, adds significant weight to the importance of having an efficient monitoring solution.

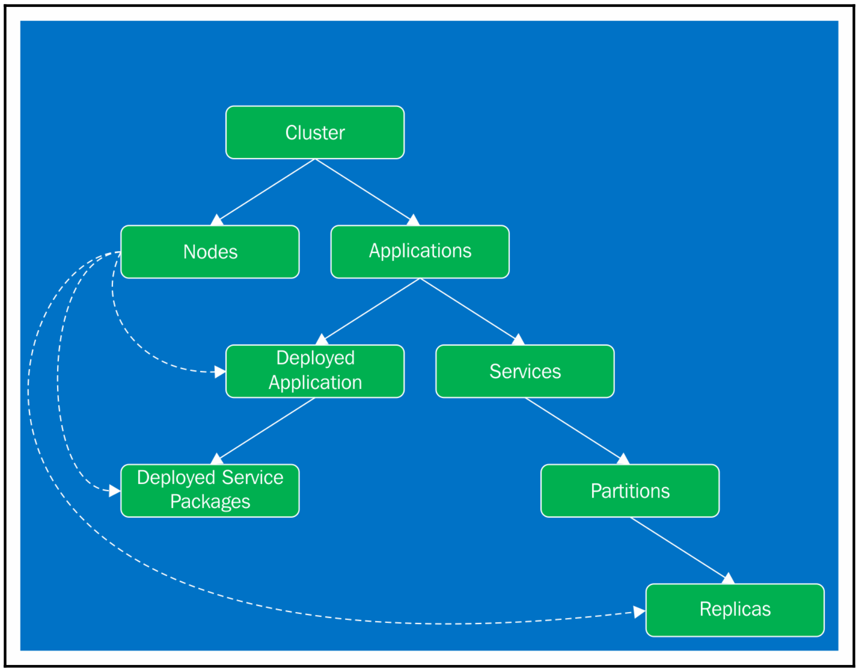
Service Fabric provides rich features to effectively monitor and manage the health of the Microservices deployed on its cluster. These features are capable of reporting near-real-time status of the cluster and the services running on it. Service Fabric employs a dedicated subsystem called the **health subsystem** to encapsulate all the monitoring and diagnostics features. To familiarize ourselves with this subsystem, let's explore the components within this subsystem, their roles, and responsibilities.

Health entities

Health entities, in the context of Service Fabric, can be considered as logical entities used by the runtime. These entities form a logical hierarchy based on the dependencies and interactions among them.

Service Fabric uses a *health store* to persist the hierarchy of its health entities and their health information. To ensure high availability, health store is implemented as a stateful service within the Service Fabric cluster and is instantiated with when the cluster starts.

The health subsystem leverages the health entities and hierarchy to effectively report, debugg, and monitor. The following diagram illustrates the hierarchy of health entities:



Hierarchy of health entities

You may notice that the health entities match the Service Fabric deployment entities discussed earlier in this book. Let's dive deeper to understand each of these entities and the parent-child relationships illustrated in the preceding figure.

The root node, **Cluster** represents the health of Service Fabric cluster. This is an aggregation of conditions that affects the entire cluster and its components. The first child of the cluster, **Node** represents the health of a Service Fabric node. Health information such as resource usages and connections for a node is captured within this health entity. **Applications**, the second child of the **Cluster**, represents a Service Fabric application. The health of all running instances of services in the scope of an application is aggregated into this entity. The **Services** entity represents all partitions of an application service deployed on the cluster. This entity can be further drilled down to **Partitions** which represents service partitions and **Replicas** which represents the replicas of each partition. The **Deployed Application** and **Deployed Service Packages** are entities which capture the application and services in scope of a specific node.

This granularity in the hierarchy makes it easy to identify a fault and fix is easily. For example, the cluster will appear unhealthy if one of its application is unhealthy. We can then drill down into the **Applications** sub tree to detect the service which is unhealthy and then to the partition and so on.

Service Fabric may use internal or external watchdog services to monitor a specific component. At any given point, the state of the health entities will represent near-real-time state of the cluster. A property called health state associated with each entity is used to identify the health of that entity.

Health state

Following are the four possible states for the health of a Service Fabric entity:

- **OK**: The entity and its children are healthy as of the current reports.
- **Warning**: This state informs us that there may be some potential issue on this entity of any of its children. A warning state does not mean that the entity is unhealthy. Delay in communication or reporting can also cause this state. A warning state will usually recover or degrade down to an error within some time. It is normal to see this state on your cluster while you are updating the application, service, or the cluster itself.
- **Error**: The entity has an error reported on it. A fix for this will be required to ensure correct functioning of this entity.
- **Unknown**: This state usually shows up when the reported entity is absent in the health store. This may be because of clean up or delay in the setup.

Health policies

Health store uses policies to determine the state of an entity. These policies are called **health policies**. Service Fabric, by default sets a set of health policies for the cluster based on parent-child relationships in the entity hierarchy. This means that a parent is marked as unhealthy if any of its children are reported as unhealthy. Apart from the default policies, custom health policies can be set for a Service Fabric deployment in the cluster or application manifest.

Cluster health policy

The health state of the cluster and the nodes deployed within the cluster is evaluated based on the cluster health policies. Custom cluster health policies can be set in the cluster manifest file under the `FabricSettings` section. The following is an example:

```
<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>
```

Let's now understand what each of these parameters listed in the preceding policy is used for:

- `ConsiderWarningAsError`: This parameter specifies weather or not to treat warning reported by the health monitors as errors. This parameter is set to `false` by default.
- `MaxPercentUnhealthyApplications`: This parameter decides the threshold for marking the cluster entity as unhealthy based on the number of applications which can afford to be unhealthy at a given point of time. The default value for this entity is `0`.
- `MaxPercentUnhealthyNodes`: This is the threshold for marking the cluster as unhealthy based on the number of unhealthy nodes at that given point of time. This parameter helps us accommodate for nodes which are undergoing maintenance.
- `ApplicationTypeHealthPolicyMap`: This parameter helps qualify special applications with specific rules. In the preceding example the parameter is used to specify a tolerance threshold for applications of the type `ControlApplicationType` to `0`. This feature helps specify significance of some applications over other in terms of their expected availability.

Application health policy

Application health policy defines the rules for evaluating and aggregating the health status for the application entity and its children entities. The custom application health policy can be configured in the application manifest by using the following parameters:

- `ConsiderWarningAsError`: Similar to the parameter in the cluster health policy. This will specify whether a warning will be treated as an error during health reporting
- `MaxPercentUnhealthyDeployedApplications`: This parameter specifies the threshold for the application entity to be marked unhealthy based on the number of unhealthy deployed applications in the cluster
- `DefaultServiceTypeHealthPolicy`: This parameter specifies the default service type health policy
- `ServiceTypeHealthPolicyMap`: This parameter specifies the service health policies map for each service type

Service type health policy

Service type health policy defines the rules to aggregate the health status of services deployed in a cluster on to the service health entity. This policy can be configured in the application manifest file. The following is an example:

```
<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>
```

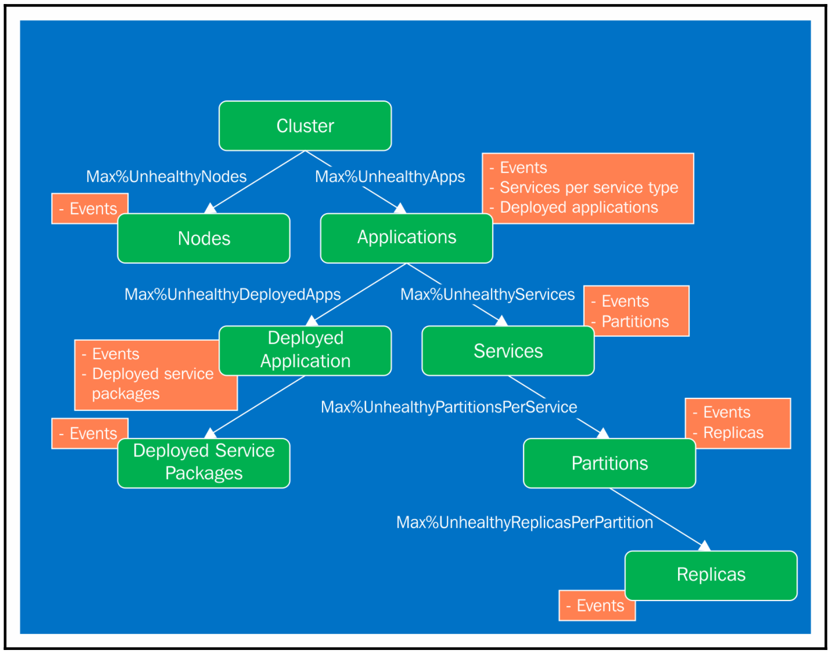
This policy uses the following parameters to specify the tolerance levels:

- `MaxPercentUnhealthyPartitionsPerService`: The threshold for unhealthy partitions before marking a service entity as unhealthy
- `MaxPercentUnhealthyReplicasPerPartition`: The threshold for unhealthy replicas before marking a partition entity as unhealthy
- `MaxPercentUnhealthyServices`: The threshold for unhealthy services before marking an application entity as unhealthy

The default value for all the three preceding listed parameters is zero.

Health evaluation

Health evaluation can be performed manually by a user or by automated services employed to report on the system health. Health policies are used to evaluate and report the health of an entity which is then aggregated at a parent level in the entity hierarchy. This process cascades up to the root element of the hierarchy which is the **Cluster**. The following diagram illustrates this process of cascading aggregations:



Cascading aggregations

After the health of child elements are evaluated, the health sub-system evaluates the health of a parent entity based on the maximum percentage of unhealthy children configured at a specific level.

Health reporting

The health sub-system used internal and external watchdogs to report the health of an entity. Watchdogs are background processes which are tasked to repeatedly perform a background tasks at pre-configured intervals. The watchdogs constantly monitor the health entities and generate health reports. A health report will have the following properties:

- `SourceId`: A unique identifier for this instance of the health report.
- `Entity identifier`: Used to identify the health entity on which the report is being generated. This can be any entity in the health hierarchy such as cluster, node, application, and likewise.
- `Property`: A string holding the metadata around which property of the health entity is being reported on.
- `Description`: A custom description.
- `HealthState`: One of the possible health states.
- `TimeToLive`: The time span specifying the validity of this health report.
- `RemoveWhenExpired`: A Boolean flag to specify if the report should be removed from the health store on expiry.
- `SequenceNumber`: An incrementing sequence number to identify the order in which the health reports were generated.

These reports are then transformed in to *health events* and stored in the health store after appending additional information such as UTC time stamps and *last modified* information.

PowerShell commands can be used to generate health reports and to view them. The following is an example:

```
PS C:\> Send-ServiceFabricApplicationHealthReport -ApplicationName
fabric:/<Application name> -SourceId <Report source> -HealthProperty
<Health property> -HealthState <Health state>

PS C:\> Get-ServiceFabricApplicationHealth fabric:/<Application name>
```

Now that we understand the health subsystem and the process how health is evaluation, let's explore the technologies which can be used to monitor a Service Fabric cluster.

Centralized logging

A Microservices application running on distributed platform such as Service Fabric is particularly hard to debug. You cannot debug the application by attaching a debugger to spot and fix the issue. Logging is a commonly used mechanism to track application behavior.

All the processes and applications running on distributed systems generate logs. The logs are usually written to files on local disk. However, since there are multiple hosts in a distributed system, managing and accessing the logs can become cumbersome. To solve this problem, a centralized logging solution is required so that multiple logs can be aggregated in a central location.

For your Service Fabric applications, you can enable collection of logs from each cluster node using Azure **Diagnostics** extension, which uploads logs to Azure Storage. Once you have aggregated the logs, you can use products such as Elastic Search, Azure Operations Management Suite, and so on, to derive useful information from it.

Collecting logs

There are two types of logs sources that you should stream to log storage:

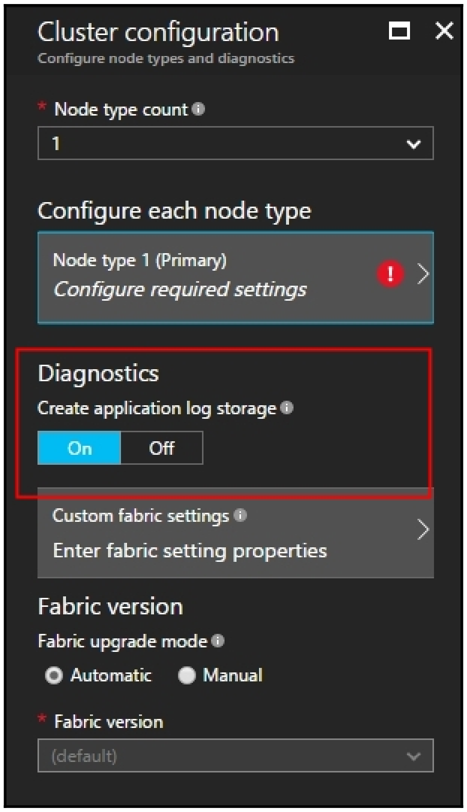
- **Service Fabric logs**: Service Fabric platform emits logs to **Event Tracing for Windows** (**ETW**) and `EventSource` channels. Service Fabric logs include events emitted from Service Fabric platform such as creation and application, state change of nodes, and so on. These logs also contain information emitted from the Reliable Actors and Reliable Services programming models.
- **Application Logs**: These logs are generated from the code of your service. You can use the `EventSource` helper class available in Visual Studio templates to write logs from your application.

Diagnostic extension

Distributed systems collect logs using agents deployed on the nodes in the cluster. In Service Fabric, the Azure **Diagnostics** extension provides the monitoring and diagnostics capabilities on a Windows based Azure Service Fabric nodes. The **Diagnostics** extension collects logs on each VM and uploads them to the storage account that you specify. You can use the Azure Management Portal or Azure Resource Manager to enable **Diagnostics** extension on cluster nodes.

Deploying the Diagnostics extension

To deploy the **Diagnostics** extension during cluster creation, as shown in the following image, use the **Diagnostics** settings panel. To enable Reliable Actors or Reliable Services event collection, ensure that **Diagnostics** setting is set to **On**. After you create the cluster, you can't change these settings by using the portal:


Cluster configuration
Configure node types and diagnostics

* Node type count: 1

Configure each node type
Node type 1 (Primary)
Configure required settings

Diagnostics
Create application log storage: [✓ On] Off

Custom fabric settings
Enter fabric setting properties

Fabric version
Fabric upgrade mode: [✓ Automatic] Manual

* Fabric version: (default)

Set the Diagnostics setting of the cluster

Resource Manager templates can be used to enable diagnostics on an existing cluster or to update the configurations of an existing cluster.

You can read more about the templates used to modify an existing cluster here: `https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-diagnostics-how-to-setup-wad`.

After setting up diagnostics on your Service Fabric cluster, you can connect the log data storage to services such as **Operations Management Suit** (**OMS**), Elastic Search, and so on. You can also use Application Insights SDK to send telemetry data to application insights so that you can monitor your application and respond to errors.

You can read more about how you can configure OMS to retrieve data from diagnostics storage and display it in the OMS portal here: `https://docs.microsoft.com/en-us/azure/log-analytics/log-analytics-service-fabric`.

Summary

In this chapter, we explored the health monitoring system of Service Fabric which includes the health entities and health states. We also looked at health evaluation and reporting techniques used by Service Fabric.

Towards the end of the chapter we looked at installing the **Diagnostics** extension on the Service Fabric cluster.

In the next chapter, we will discuss how we can integrate Service Fabric applications with Continuous Integration and Continuous Deployment workflows.

11
Continuous Integration and Continuous Deployment

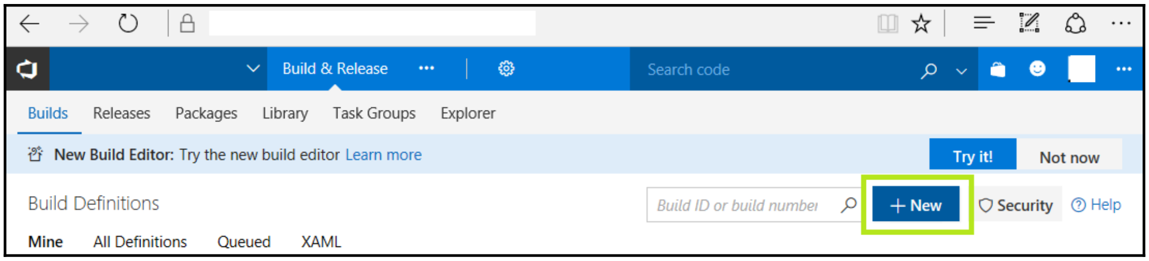
Continuous Integration (**CI**) and **Continuous Delivery** (**CD**) are two inevitable patterns for an agile software development team. These patterns become more relevant for an application development teams driven by Microservices due to the number of teams working on a project and their diverse release cycles. Continuous Integration can be defined as the process of merging developer code to a source depot after successfully building the project. Continuous Delivery then takes care of running the unit tests and deploying the application to a target environment.

Continuous Integration

Microsoft recommends the use of **Visual Studio Team Services** (**VSTS**) to continuously integrate your code into a source depot. The first step for setting up Continuous Integration is to set up a build definition. Following are the steps to be followed for creating a build definition from existing build templates:

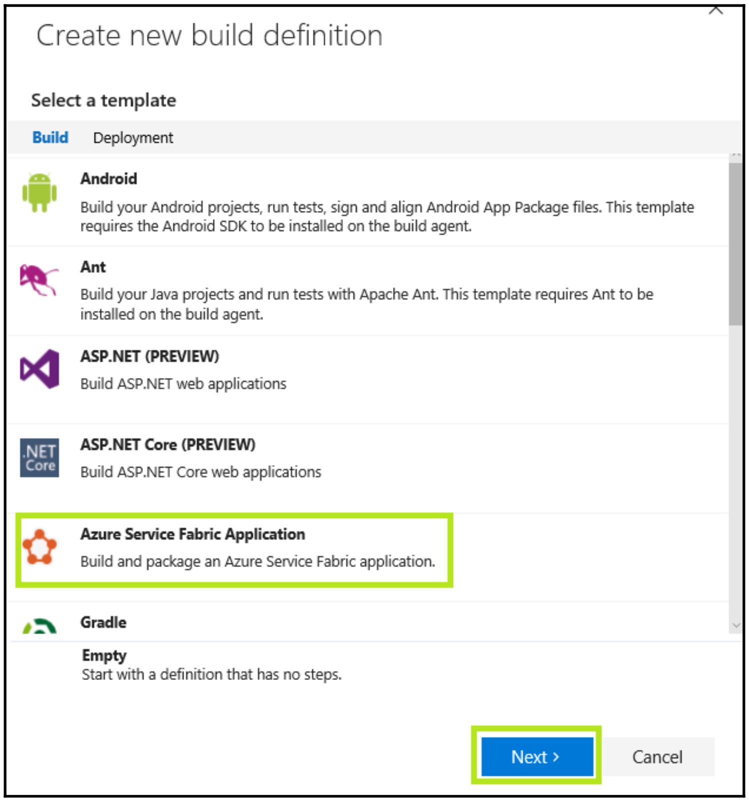
1. Browse to the VSTS portal and navigate to the **Builds** tab.

2. Click on the **New** button to create a new build definition:



Create new build definition

3. Select **Azure Service Fabric Application** within the **Build** template category and click **Next**:


Create new build definition

Select a template

Build Deployment

Android
Build your Android projects, run tests, sign and align Android App Package files. This template requires the Android SDK to be installed on the build agent.

Ant
Build your Java projects and run tests with Apache Ant. This template requires Ant to be installed on the build agent.

ASP.NET (PREVIEW)
Build ASP.NET web applications

ASP.NET Core (PREVIEW)
Build ASP.NET Core web applications

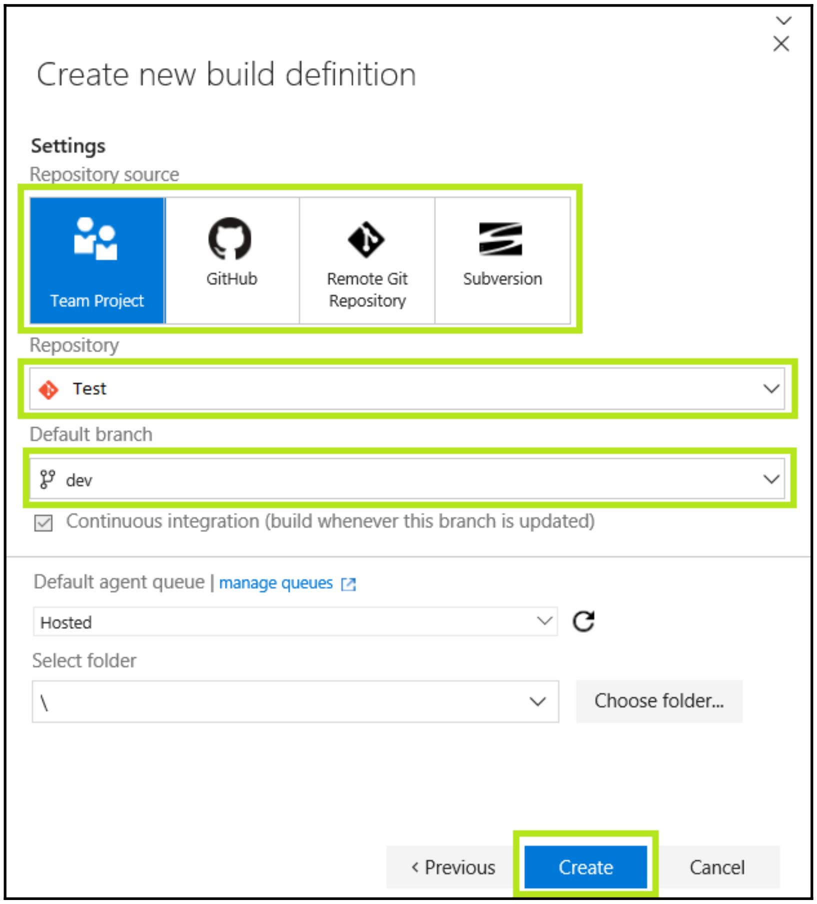
Azure Service Fabric Application
Build and package an Azure Service Fabric application.

Gradle

Empty
Start with a definition that has no steps.

Select Service Fabric build definition template

4. Select the source control repository for the Service Fabric application and click **Create**:


Create new build definition

Settings

Repository source

Repository: Test

Default branch: dev

☑ Continuous integration (build whenever this branch is updated)

Default agent queue | manage queues: Hosted

Select folder: \

Create new build definition

5. Save the build definition with a name.

The following build steps are added to the build template as a part of this template:

1. Restore all NuGet packages referred to in this solution.
2. Build the entire solution.
3. Generate the Service Fabric application package.

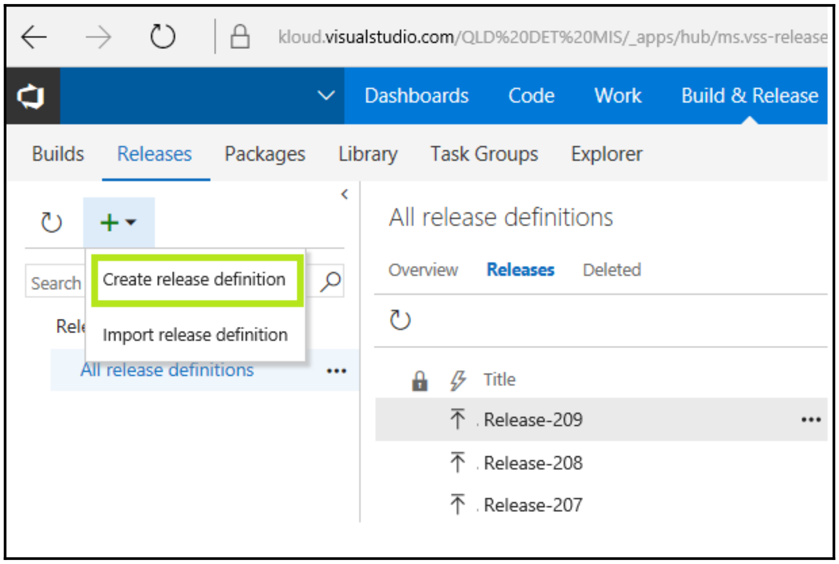
4. Update the Service Fabric application version.
5. Copy the publish profile and application parameters files to the build's artifacts to be consumed for deployment.
6. Publish the build's artifacts.

A build can now be queued to test the build definition.

Continuous Delivery

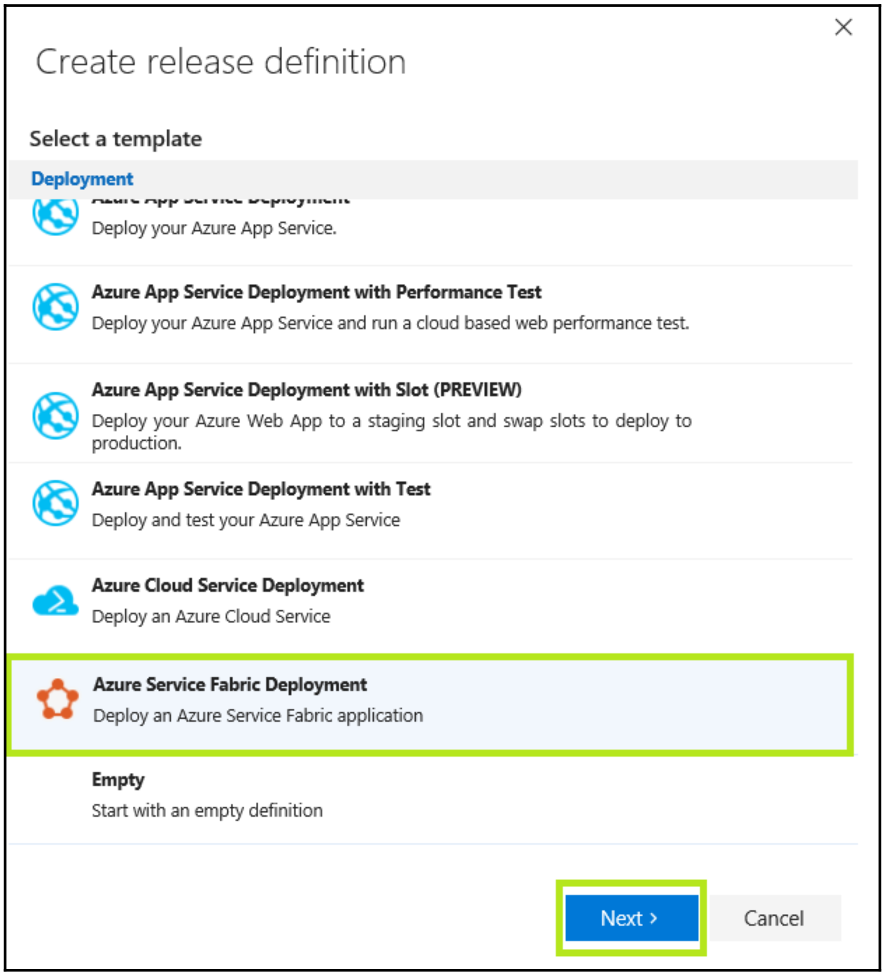
Once we have a successful build, we can move ahead to create a release definition to automate Continuous Delivery. A VSTS release definition can be used to define the tasks which should be executed sequentially to deploy a packages Service Fabric application to a cluster. Following are the steps to be followed to create a release definition:

1. Browse to the VSTS portal and navigate to the **Releases** tab.
2. Select the **Create release definition** menu item:



Create release definition

3. Select **Azure Service Fabric Deployment** within the **Deployment** template category and click **Next**:


Create release definition

Select a template

Deployment

Azure App Service Deployment
Deploy your Azure App Service.

Azure App Service Deployment with Performance Test
Deploy your Azure App Service and run a cloud based web performance test.

Azure App Service Deployment with Slot (PREVIEW)
Deploy your Azure Web App to a staging slot and swap slots to deploy to production.

Azure App Service Deployment with Test
Deploy and test your Azure App Service

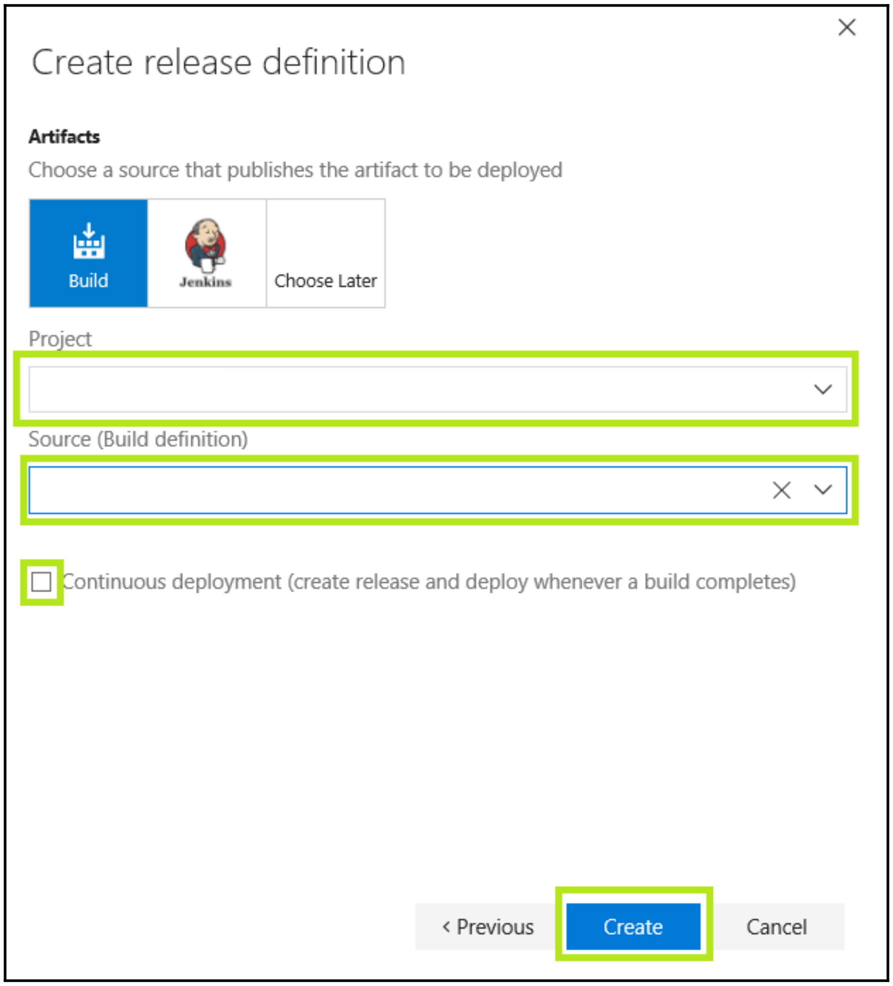
Azure Cloud Service Deployment
Deploy an Azure Cloud Service

Azure Service Fabric Deployment
Deploy an Azure Service Fabric application

Empty
Start with an empty definition

Select Azure Service Fabric template

4. Select the already created build definition from the drop-down list, click on the check box to enable Continuous Deployment and click on **Create**:



Create release definition

Artifacts

Choose a source that publishes the artifact to be deployed

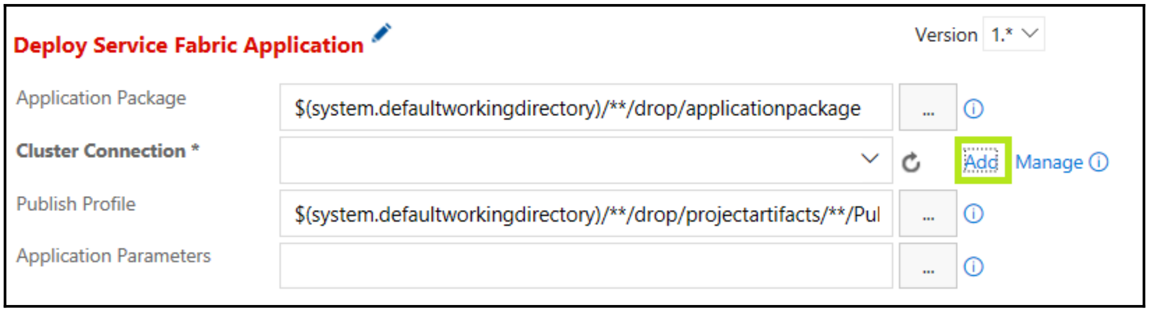
Project

Source (Build definition)

☐ Continuous deployment (create release and deploy whenever a build completes)

Select project and build definition

5. Add the **Cluster Connection** configuration:

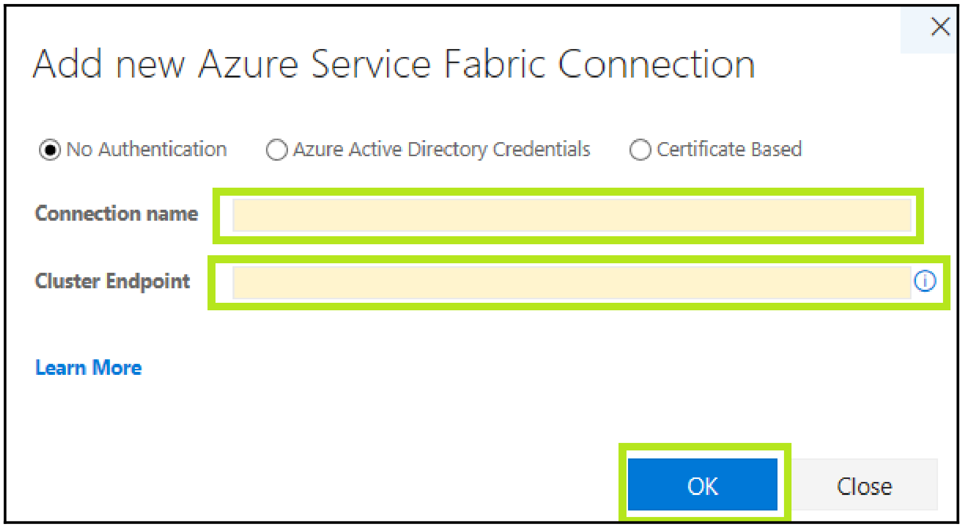

Deploy Service Fabric Application

Version 1.*

Application Package	$(system.defaultworkingdirectory)/**/drop/applicationpackage
Cluster Connection *	
Publish Profile	$(system.defaultworkingdirectory)/**/drop/projectartifacts/**/Pul
Application Parameters	

Setting application package path and publish profile

6. In the dialogue, give the cluster connection a name and add the **Cluster Endpoint** and click **OK**:


Add new Azure Service Fabric Connection

[✓ No Authentication] Azure Active Directory Credentials Certificate Based

Connection name	
Cluster Endpoint	

Add new Service Fabric connection

A release can now be triggered from the **Releases** tab manually to test the newly created release definition.

Now that we have explored a method of managing Continuous Deployment of application on Azure Service Fabric cluster, let's look into the options of running Microservices on other hosting platforms.

Deploying Service Fabric application on a standalone cluster

It is common for an enterprise application to be hybrid by nature in terms of its deployment strategy. Factors like strict enterprise policies around data storage and requirement of integrating with legacy systems usually forces an application to span across multiple data centers and hosting platforms.

A Service Fabric cluster can be created with machines running Windows Server on any hosting platform. These machines can be hosted on Azure, on a private on-premises data center, on **AWS** (**Amazon Web Services**), Google Cloud Platform, or any other similar platforms.

An installation package for Service Fabric can be downloaded from the Microsoft website which will contain the required files to setup a standalone cluster. It is advisable to enable internet access on the host machines to enable real-time downloading of files required for the Service Fabric runtime. Offline versions of these files are also available as an alternative.

The following link details the steps to create and configure a standalone cluster using Windows Server machines: `https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-creation-for-windows-server`.

Deploying the application

Once the cluster is created, PowerShell cmdlets can be used to deploy and manage applications on the standalone cluster. The first step would be connecting to the cluster. The following command can be used to connect to an existing Service Fabric cluster:

```
Connect-ServiceFabricCluster -ConnectionEndpoint
<*IPAddressofaMachine*>:<Client connection end point port>
```

Once connected, we should be able to launch the Service Fabric explorer remotely by browsing to the following URL:
`http://<IPAddressofaMachine>:19080/Explorer/index.html`.

The next step will be to upload the application package. The cmdlet for uploading the package is part of the Service Fabric SDK PowerShell module which should be imported before performing the upload.

The connection string for the local image store where the package has to be uploaded can be retrieved using the `Get- ImageStoreConnectionStringFromClusterManifest` cmdlet. `Copy-ServiceFabricApplicationPackage` cmdlet can then be used to copy the application package to the image store. Following is an example for the same:

```
PS C:\> Copy-ServiceFabricApplicationPackage -ApplicationPackagePath <path>
-ApplicationPackagePathInImageStore <application name> -
ImageStoreConnectionString (Get-
ImageStoreConnectionStringFromClusterManifest(Get-
ServiceFabricClusterManifest)) -TimeoutSec <timeout>
```

Once the package is copied to the image store the application can be registered using the following PowerShell cmdlet:

```
PS C:\> Register-ServiceFabricApplicationType <Application name>
```

Registry of an application can be verified using the `Get-ServiceFabricApplicationType` cmdlet:

```
PS C:\> Get-ServiceFabricApplicationType
```

After the application is registered, the next step is to create an instance of this application on the Service Fabric cluster. This can be achieved using the following cmdlet:

```
PS C:\> New-ServiceFabricApplication fabric:/<application name>
<application type> <version>
```

A deployed application can be verified using the `Get-ServiceFabricApplication` and the `Get-ServiceFabricService` cmdlets:

```
PS C:\> Get-ServiceFabricApplication
PS C:\> Get-ServiceFabricApplication | Get-ServiceFabricService
```

A Service Fabric cluster can also be deployed on Linux machines.

Details about setting up a Linux cluster can be found at the following link: `https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-linux-overview`.

Summary

In this chapter, we discussed how Visual Studio Team Services can be used to continuously integrate code to a centralized source repository and to continuously deploy Microservices on Azure Service Fabric cluster.

In the next chapter, we will walk you through the concept of serverless computing using Azure Functions.

12
Serverless Microservices

Until recently, every code development was accompanied with overheads of maintaining orchestrations, deployments, and so on. With the evolution in IT, developers desire to eliminate waste and focus on specific business objectives.

In a serverless environment, developers only stay concerned with solutions and the monitoring of usage. The business saves on costs by paying for computation cycles consumed and not for the idle time of system. A serverless system lowers the total cost of maintaining your apps, enabling you to build more logic faster. In a serverless computing model, the cloud provider manages starting and stopping of the container of the service as necessary to serve requests and the business need not pay for the virtual machines on which the services execute.

The growing requirement of developing Microservices that are much smaller and highly focused has given rise to a new breed of services called Nanoservices. Nanoservices can either be triggered by specific events, or be configured to run behind an API management platform to expose it as a REST API endpoint.

Nanoservices share certain commonalities with Microservices which also form the definition boundary of Nanoservices:

- Nanoservices generally run in a lesser isolation than Microservices, which means that they can be more densely packaged than Microservices. However, Nanoservices are independently deployable just like Microservices are.
- Nanoservices are more finely grained than Microservices. A Nanoservice may contain just a few lines of code, which perform a very specific task for example, send an email on successful registration of a user.
- The execution environment of Nanoservices is more efficient than that of Microservices. The Nanoservices execution environment is cognizant of activation and deactivation of Nanoservices and its resource consumption and can therefore dynamically allocate and deallocate resources.

The smaller size of Nanoservices affects the architecture of solution. Nanoservices decrease the deployment risk due to smaller size of deployment units and can help deliver even more understandable and replaceable services. On the other hand, the complexity of deployment increases significantly as a bounded context, which earlier used to contain one or a couple of Microservices will now contain many Nanoservices each of which implement a very narrow functionality.

Isolation is not a clear demarcation between Microservices and Nanoservices as Microservices can be configured to share the same virtual machine while Nanoservices can be configured to run on independent virtual machines. Therefore, Microservices can be said to be a coarser form of Nanoservices.

Just like isolation, minimum size is also not a clear demarcation between Microservices and Nanoservices. For instance, in order to implement **Command Query Responsibility Segregation** (**CQRS**) in a solution, a Microservice may only be responsible for writing a certain type of data and another Microservice may be responsible for reading the same type of data, leading to Microservices have a small scope just like Nanoservices.

Before committing to Nanoservices

Nanoservices live within the sizing constraints of Microservices. Smaller size of modules enables the service to maintain and change. Therefore, a system composed of Nanoservices can be easily extended.

In a typical domain driven system, each class and function can be modelled as a separate Nanoservice. This leads to an increase in infrastructure costs such as that of application servers and monitoring solutions. Since implementing a complete business solution using Nanoservices might involve deploying several hundreds to a couple of thousands of Nanoservices, the infrastructure costs on the desired cloud platform need to be as low as possible. In addition, Nanoservices should not be long running or resource intensive.

Nanoservices require a lot of communication among themselves. This may lead to degraded performance. On certain platforms, Nanoservices may share a process, which may cause resource starvation and take away technological freedom. This approach however, can reduce the overhead of inter-service communication.

Before you plan to commit to a platform for hosting Nanoservices, the following objectives should be considered:

- Cost of infrastructure for hosting Nanoservices should be low. The platform should support rapid deployment and monitoring of Nanoservices.
- The Nanoservices platform should support dense deployment of Nanoservices.
- Inter-service communication will degrade performance. A highly reliable and efficient means of communication would ensure that the performance remains within acceptable limits.
- Nanoservices should be independently deployable and the platform should ensure that failures are not propagated between Nanoservices.
- Nanoservices are limited in terms of choice of programming language, platforms, and frameworks. You should study technical feasibility of building Nanoservices on a platform before committing to it.

Building Nanoservices with Azure Functions

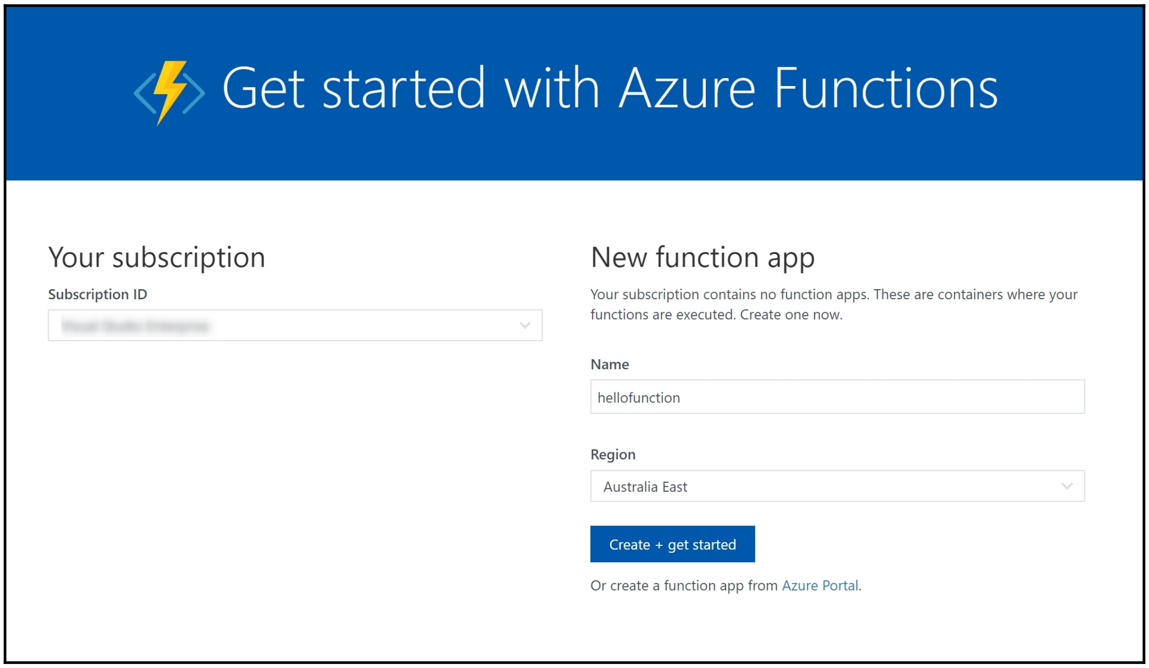
Azure Functions allow developers to write serverless applications, meaning that developers or operations do not have to worry about the infrastructure on which the application executes. In many scenarios, application or business needs require a small piece of logic to be reused by other services for some small task to be performed based on an event such as sending a notification to user when a message is sent to a queue.

Such tasks were previously handled by WebJobs or scripts, which are difficult to reuse and connect in a flow of logic. Azure Functions give developers the ability to wrap the logic in a Nanoservice that can connect and communicate with other services to carry out the logic flow.

Azure Functions is part of web and mobile suite of services in Azure. In addition to Visual Studio tooling, you can design and manage Azure Functions from a dedicated portal at: `https://functions.azure.com`.

Currently you can create functions in JavaScript, C#, Python, and PHP as well as with scripting languages such as Bash, Batch, and PowerShell. The functions can be triggered by virtually any event in an on cloud or on-premise system or a third-party service.

Let's head over to `https://functions.azure.com` to explore Azure Functions in greater detail:


Get started with Azure Functions

Your subscription

Subscription ID

New function app

Your subscription contains no function apps. These are containers where your functions are executed. Create one now.

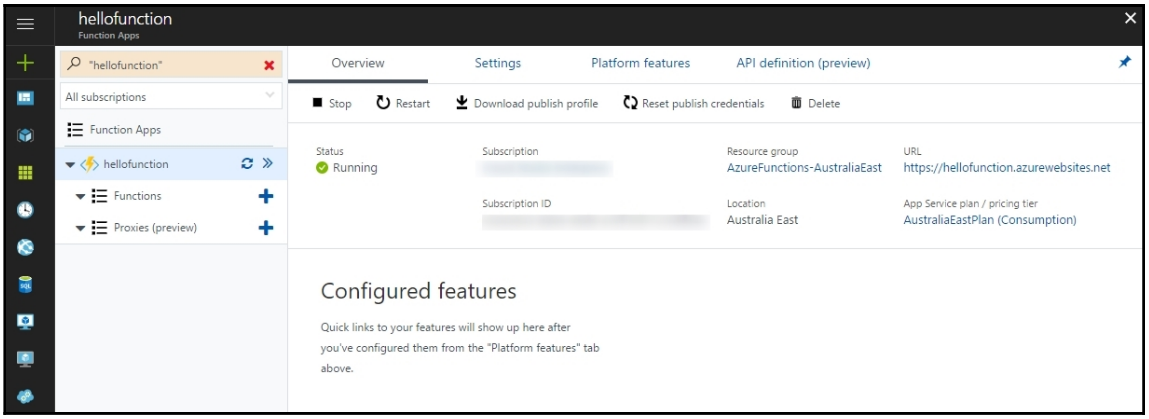
Name: hellofunction

Region: Australia East

Azure Functions landing page

In the function area, you will notice that you can select a subscription where you want to create your function and assign a **Name** and **Region** to your function. Assign a name to your function and click the **Create + get started** button.

Once your function gets provisioned, you will be redirected to the Management Portal, on which you will be able to compose and manage your function:



Status
Running

Subscription
[redacted]

Resource group
AzureFunctions-AustraliaEast

URL
https://hellofunction.azurewebsites.net

Subscription ID
[redacted]

Location
Australia East

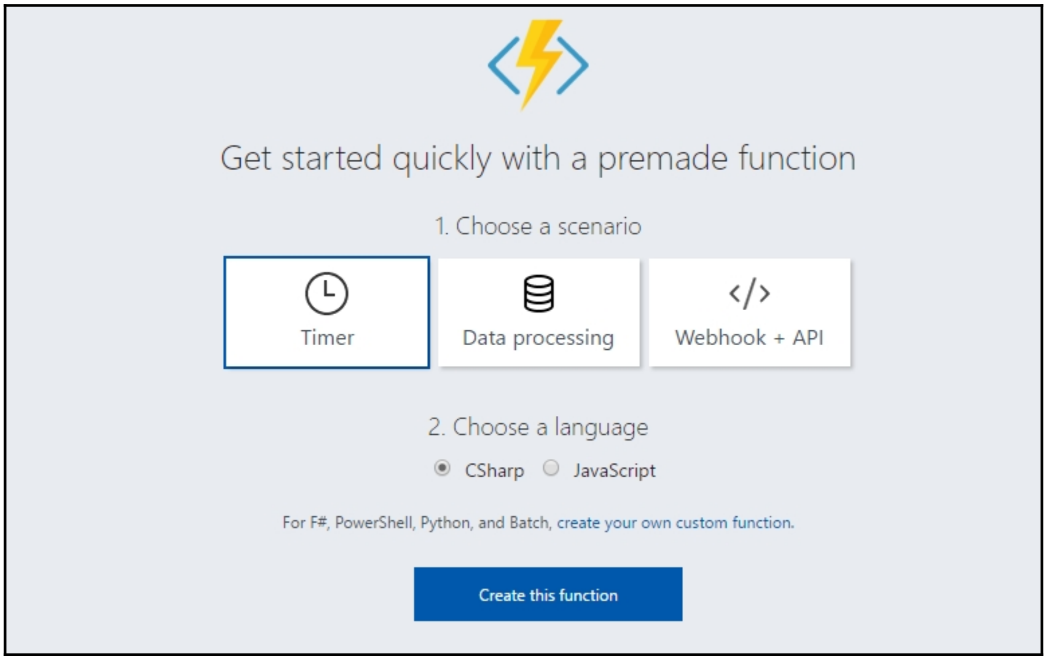
App Service plan / pricing tier
AustraliaEastPlan (Consumption)

Configured features

Quick links to your features will show up here after you've configured them from the "Platform features" tab above.

Azure Functions in Azure Management Portal

To add a function to your application, click on the + sign next to the **Functions** menu to bring up the quick start panel:



Get started quickly with a premade function

1. Choose a scenario

2. Choose a language

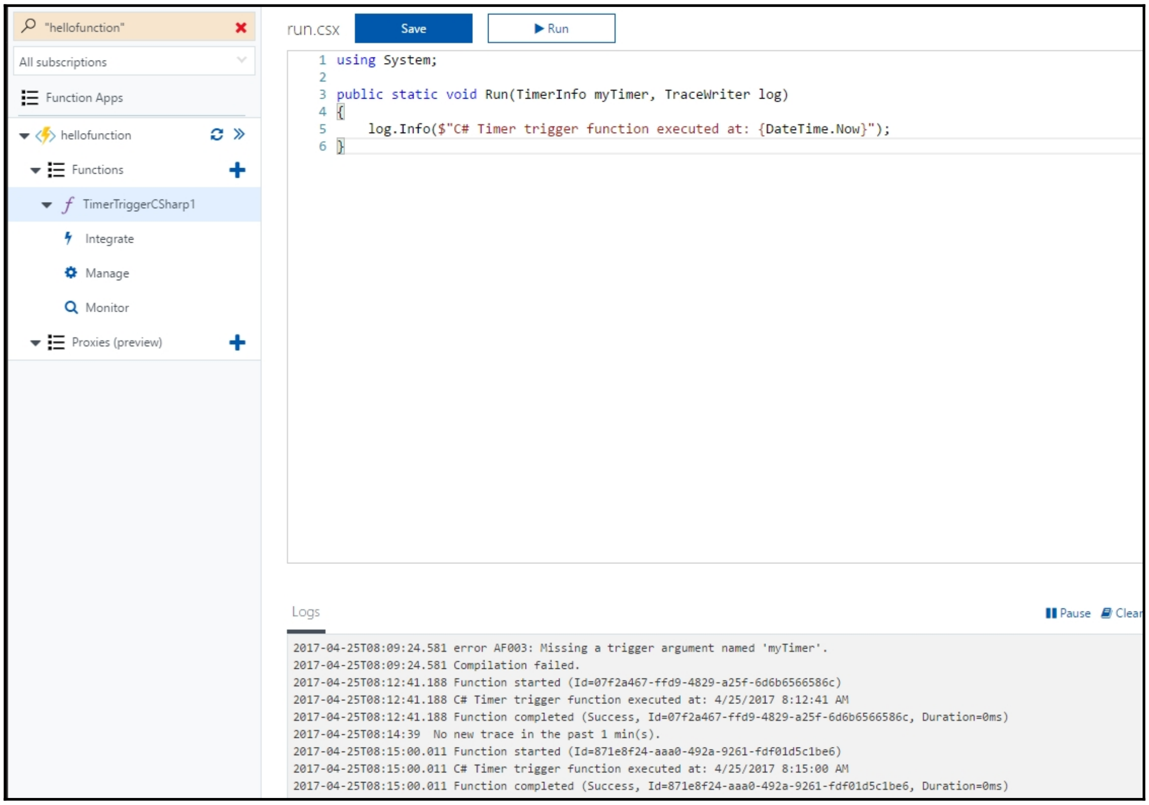
[✓ CSharp] JavaScript

For F#, PowerShell, Python, and Batch, create your own custom function.

Azure Functions quick start panel

Select the **Timer** scenario, which will configure your function to get executed based on a configurable schedule. Set the language preference to compose the function as C# and click on **Create this function** button.

The previous steps will create a function that write a message to log every five minutes. You can view the log output in the window below the editor console:

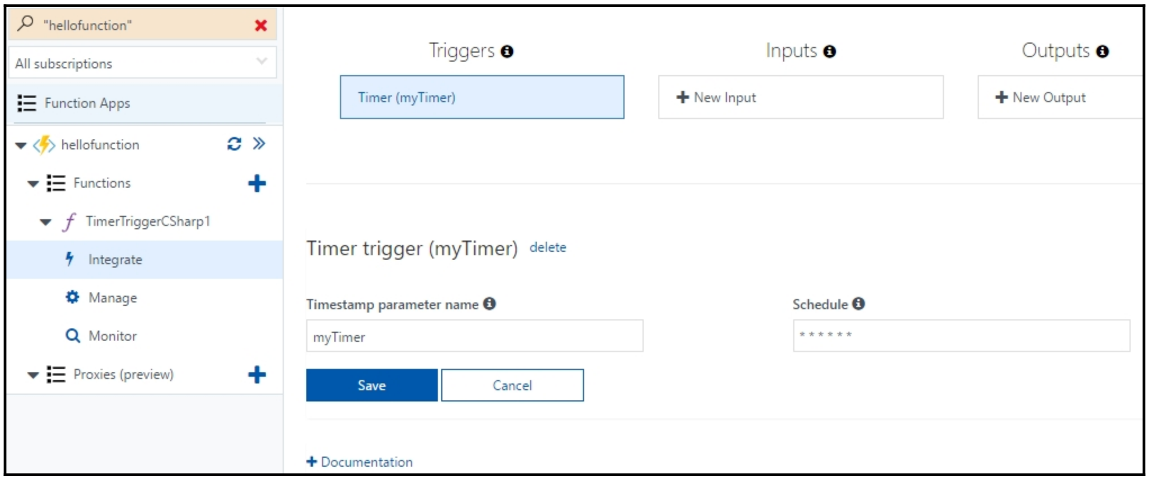


```
using System;

public static void Run(TimerInfo myTimer, TraceWriter log)
{
    log.Info($"C# Timer trigger function executed at: {DateTime.Now}");
}
```

```
2017-04-25T08:09:24.581 error AF003: Missing a trigger argument named 'myTimer'.
2017-04-25T08:09:24.581 Compilation failed.
2017-04-25T08:12:41.188 Function started (Id=07f2a467-ffd9-4829-a25f-6d6b6566586c)
2017-04-25T08:12:41.188 C# Timer trigger function executed at: 4/25/2017 8:12:41 AM
2017-04-25T08:12:41.188 Function completed (Success, Id=07f2a467-ffd9-4829-a25f-6d6b6566586c, Duration=0ms)
2017-04-25T08:14:39  No new trace in the past 1 min(s).
2017-04-25T08:15:00.011 Function started (Id=871e8f24-aaa0-492a-9261-fdf01d5c1be6)
2017-04-25T08:15:00.011 C# Timer trigger function executed at: 4/25/2017 8:15:00 AM
2017-04-25T08:15:00.011 Function completed (Success, Id=871e8f24-aaa0-492a-9261-fdf01d5c1be6, Duration=0ms)
```

Azure Function

Let's change the logging frequency of this function, so that we can see more messages in the console. Click on the **Integrate** option under the **Functions** menu:



Timer trigger (myTimer) delete

Timestamp parameter name: myTimer

Schedule: ******

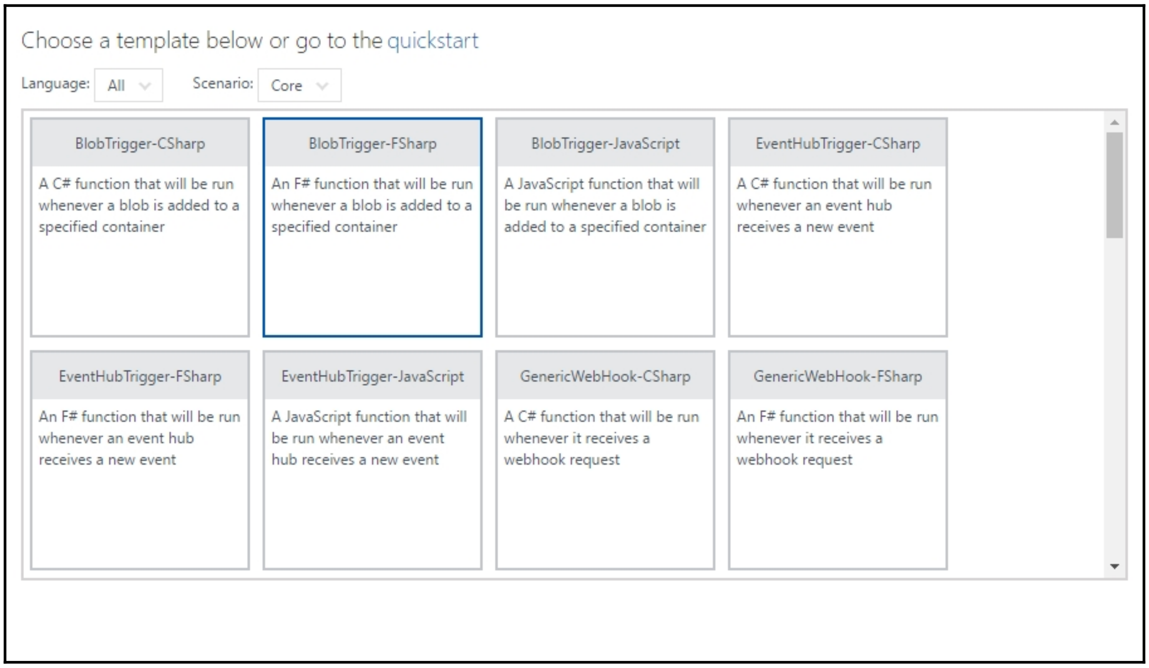
Change timer frequency

In the integration pane, change the **Schedule** CRON expression to six stars (`* * * * * *`), which means that function will execute every second, and click on **Save**.

Once the settings are saved, go back to the editor view where you will now find that new messages are logged every second. Click on **Manage** menu item and click on the **Delete** button to delete the function.

Function app templates

Although not necessary, function templates can help speed up development of functions. You can view the list of templates available to you by selecting **create your own custom function** in the function creation quick start panel:


Choose a template below or go to the quickstart

Language: All   Scenario: Core

BlobTrigger-CSharp
A C# function that will be run whenever a blob is added to a specified container

BlobTrigger-FSharp
An F# function that will be run whenever a blob is added to a specified container

BlobTrigger-JavaScript
A JavaScript function that will be run whenever a blob is added to a specified container

EventHubTrigger-CSharp
A C# function that will be run whenever an event hub receives a new event

EventHubTrigger-FSharp
An F# function that will be run whenever an event hub receives a new event

EventHubTrigger-JavaScript
A JavaScript function that will be run whenever an event hub receives a new event

GenericWebHook-CSharp
A C# function that will be run whenever it receives a webhook request

GenericWebHook-FSharp
An F# function that will be run whenever it receives a webhook request

Azure Function quick start templates

Even if you are creating custom functions from scratch, it is helpful to look at the parameters that get passed to functions and their values from the templates. Let us next study the categories of functions that we can create today.

Timer function apps

The timer function apps run at configurable intervals. The time at which the function should execute is defined through a CRON expression. The CRON expression is composed of six fields: `{second} {minute} {hour} {day} {month} {day of the week}`. These fields, separated by white space, can contain any of the allowed values with various combinations of the allowed characters for that field. For example, to trigger your function every 15 minutes, your scheduling CRON expression should be set to: `0 */15 * * * *`.

Timer function apps are generally used to send information to other systems. They don't generally return information and write the progress of the operation to logs. This category of functions is typically built to clean up or manage reconcile or manage data at regular intervals. Timer functions are also used for checking the health of other services by pinging them at regular intervals. Just like any other category of functions, these functions can be combined with other functions to develop a complex scenario.

Data processing function apps

Most of the systems built in organizations today are data processing systems. A typical data processing system can perform one or a combination of the following tasks:

- Conversion of data from one format to another
- Targeting of input data to appropriate storage
- Validation and clean-up of data
- Sorting of data
- Summarization of data
- Aggregation of data from multiple sources
- Statistical analysis of existing or new data
- Generating reports that list a summary or details of computed information

Data processing function apps can be used to build Nanoservices that can be aggregated to form data processing systems. Data processing function apps are always triggered by a data event. A data event is raised when state of data changes in a linked resource for example an item being added to a table, a queue, a container, and so on.

A data processing function has a set of in parameters which contain the data coming in for processing. Some of the scenarios where data processing functions are commonly used are:

- **Responding to CRUD operations**: Scenarios which require performing an action whenever state of data in a data store changes for example sending an email whenever a new use signs up.
- **Perform CRUD operations**: Scenarios in which data needs to be created or updated in another data store in response to data being added or updated in a data store.

- **Moving content between data stores**: Scenarios which require moving content between data stores for example moving a file to a temporary location for approval before moving it to a discoverable location. Such data transfer tasks can be carried out through a data processing Nanoservice.
- **Access data across services**: Scenarios which require pulling data from data stores such as blobs, queues, tables, and so on, and perform operations on them. The Nanoservice that accesses data from the stores can be integrated with other applications or Nanoservices that wish to reuse the workflow of the Nanoservice.

Webhook and API function apps

The webhook and API functions get triggered by events in external services such as GitHub, TFS, Office 365, OneDrive, and Microsoft PowerApps.

With the webhook and API functions, you can build notification Nanoservices that can perform custom operations whenever they receive a message on a configured webhook. For example, you can use webhooks with OneDrive that notifies your Nanoservice whenever a file gets uploaded to a folder.

Webhook and API functions accept a request and return a response. They mimic the web API or web service flows. These functions generally require some **CORS** (**Cross-Origin Resource Sharing**) settings to be managed. While developing the Nanoservices you can use an asterisk wildcard so they are wide open. However, you need to be aware that to invoke these Nanoservices from other services, you would need to set the cross-origin information in your function app settings.

These types of functions are generally used for exposing functionality to other apps and services. Other systems and clients can make web calls using HTTP protocols to them and expect a response. These Nanoservices are generally integrated with logic apps to form a workflow.

Summary

In this chapter, we looked at the definition of Nanoservices and how we can build Nanoservices using Azure Functions. We experimented with a simple Azure Function and customized a few of its attributes.

We looked at the various types of Azure Functions and what is the use of each one of them.

Index