***Azure Application Migration –***

***Leveraging Service Fabric while rebuilding Scalable and High Performing Applications***

**A White Paper**

# **Contents**

[Introduction 3](#_Toc494830909)

[Why Microservices? 3](#_Toc494830910)

[Why Service Fabric? 3](#_Toc494830911)

[Scope 3](#_Toc494830912)

[Business Requirements 4](#_Toc494830913)

[Challenges with Monolithic Architecture 4](#_Toc494830914)

[Migrating existing applications to Service Fabric 5](#_Toc494830915)

[Refactoring to Service Fabric and Microservices using .Net 6](#_Toc494830916)

[Rebuild to Service Fabric and Microservices using Java 6](#_Toc494830917)

[Summary and Conclusion 8](#_Toc494830918)

# Introduction

Microservices architecture is an approach to build a server application as a set of small services, each service running in its own process and communicating with each other via protocols such as HTTP and WebSockets.

Azure Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices. Service Fabric also addresses the significant challenges in developing and managing cloud applications.Service Fabric powers many Microsoft services today, including Azure SQL Database, Azure DocumentDB, Cortana, Microsoft Power BI, Microsoft Intune, Azure Event Hubs, Azure IoT Hub, Skype for Business, and many core Azure services. Service Fabric has flexibility to deploy the same application code on public, hosted, or private clouds using consistent platform services and the same application model across Azure, on-premises, and hosted datacenters*.*

## Why Microservices?

* Application is broken down into many small services, therefore each service can be built by a small focused team.
* If a service fails, other services continue to run. This helps in maintaining resiliency and fault isolation.
* Services are designed around business scenarios, making it easier to migrate services to new technologies, frameworks, or data stores.
* Services can be scaled independently.
* Services can be deployed independently, so you can take an incremental approach to upgrading an application.
* Individual services have less code, making the code easier to understand, develop and test.

## Why Service Fabric?

Service Fabric supports most of features of distributed systems therefore it is a good fit for a microservices architecture. It includes:

* Cluster management like node failover, health monitoring etc is handled by Service Fabric automatically
* Horizontal scaling wherein you can add nodes to a Service Fabric cluster
* Service Fabric provides a discovery service that can resolve the endpoint for a named service.
* Supports of Stateless and stateful services.
* Application lifecycle management where Services can be upgraded independently and without application downtime.
* Service orchestration across a cluster of machines.
* A single node can host multiple services thus allows higher density for optimizing resource consumption.

# Scope

Service Fabric is used by various Microsoft services, including Azure SQL Database, Cosmos DB, Azure Event Hubs, and others, making it a proven platform for building distributed cloud applications. The aim of Service Fabric is to solve the hard problems of building and running a service and utilize infrastructure resources efficiently, so that teams can solve business problems using a microservices approach. Service Fabric supports the following programming models for developing your service:

* Guest executables - Services can be packaged as guest executables which are arbitrary executables, written in any languages
* Reliable Services - Reliable services can be used to develop stateful as well as stateless service and supports transactions.
* Reliable Actors –Actor programming model is single threaded and is ideal for hyper scale scenarios (1000s or instances).

You can run any type of code, such as Node.js, Java, or C++ in Azure Service Fabric as a service. Service Fabric refers to these types of services as guest executables.

# Business Requirements

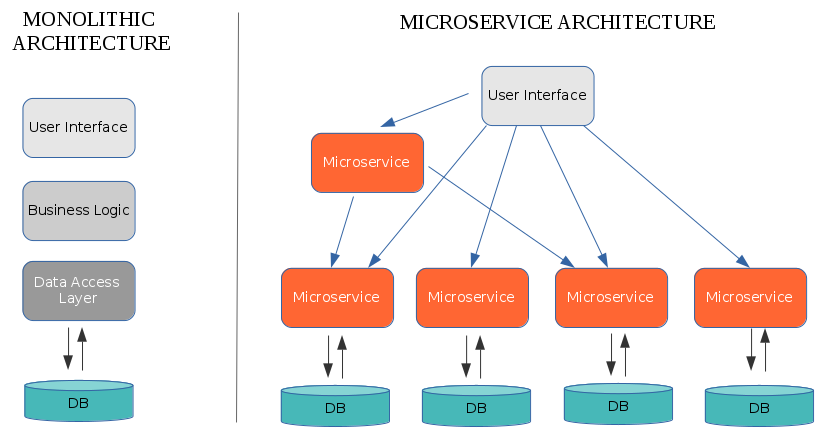
* Faster delivery or continuous deployments of features and capabilities to respond to customer demands in an agile way.
* Scalable service that's built and operates in new geographical regions as per demands
* Reduction in Cost with improved resource utilization.

# Challenges with Monolithic Architecture

* The application can be difficult to understand and modify as code is in large volume.
* Code is tightly coupled and have lot of dependencies. This makes Scalability difficult.
* In case of failure in single module, whole of application gets affected.
* Continuous deployment was difficult because to update one component you must redeploy the entire application.
* Huge impact on Developers productivity as large application takes longer time to start. Hence make application slow.

**Difference between Monolithic and Micros Services Architecture**

|  |  |
| --- | --- |
| **Monolithic** | **Microservice** |
| Its architecture represents a single logical program unit. All functions, libraries, and dependencies are located within one application block. | Its architecture consists of a series of small services that work independently and communicate with each other. Each service can be used in more than one application. |
| Here application scales horizontally behind a load balancer. | Here each service scales independently and on demand. |
| In case of changes to the system, entire application is compiled, tested, and deployed. | Each service can be changed independently. |
| Whole application is developed based on a single programming language. | Each service can be developed in a different programming language and integration can be done via API. |
| Big and complex source code hence difficult to manage. | Many independent modules of source code that are easier to maintain. |



# Migrating Applications to Service Fabric

Application can be refactored or rebuild based on the complexity, urgency, demand, architecture and knowledge of monolithic application: Some of the ways are highlighted:

**Incrementally Refactor the Monolithic Application**: Converting modules into small services and deployed gradually in conjunction with monolithic application.

**Refactor by splitting frontend and backend**: Split the presentation layer from the business logic and data access layers to convert into small services.

**Refactor by extracting services**: Start with a few modules that are easy to extract and extract those modules that will have the greatest benefits.

**Rebuild Services**: Use Monolithic application as a guide to functionalities and create new microservices both stateful and stateless depending on the functionalities.

## Refactoring to Service Fabric and Microservices using .Net

To refactor existing Cloud Services with Web and Worker Roles to Service Fabric stateless services, we can do as follows:

A Cloud Service project contains one or more Web or Worker Roles. Similarly, a Service Fabric Application project contains one or more services.

**To migrate a Worker Role to Service Fabric**: The biggest difference is that a Worker Role runs in a VM as background service so its lifecycle is tied to the VM, which includes events for when the VM starts and stops like OnStart() and OnStop(). A Service Fabric service has a lifecycle that is separate from the VM lifecycle, so it does not include events for when the host VM or machine starts and stop, as they are not related. A Worker Role instance will recycle if the Run () method exits. The RunAsync() method in a Service Fabric service however can run to completion and the service instance will stay up.

| **Service API Entry Point** | **Worker Role** | **Service Fabric service** |
| --- | --- | --- |
| Processing | Run() | RunAsync() |
| VM start | OnStart() | N/A |
| VM stop | OnStop() | N/A |
| Open listener for client requests | N/A | * CreateServiceInstanceListener() for stateless * CreateServiceReplicaListener() for stateful |

**To migrate a web application from a Web Role:** Create stateless service that can be self-hosted and does not depend on IIS or System. Example ASP.NET Core 1 does not depends on IIS.

| **Application** | **Supported** | **Migration path** |
| --- | --- | --- |
| ASP.NET Web Forms | No | Convert to ASP.NET Core 1 MVC |
| ASP.NET MVC | With Migration | Upgrade to ASP.NET Core 1 MVC |
| ASP.NET Web API | With Migration | Use self-hosted server or ASP.NET Core 1 |
| ASP.NET Core 1 | Yes | N/A |

## Rebuild to Service Fabric and Microservices using Java

For Java developers who are familiar with Eclipse and STS, Microsoft has provided the ‘[Windows Azure Toolkit for Eclipse with Java](http://msdn.microsoft.com/en-us/library/windowsazure/hh690946.aspx)’. This toolkit provides the following resources for aiding in Java development in Eclipse on Windows Azure:

* Windows Azure Plugin for Eclipse with Java
* Microsoft JDBC 4.0 Driver for SQL Server and Windows Azure SQL Database
* Package for Apache Qpid Client Libraries for JMS
* Package for Windows Azure Libraries for Java
* Windows Azure Access Control Services Filter
* Windows Azure Common Plugin

To build and work on the Service Fabric Java applications on Linux, you need to ensure that you have JDK 1.8 and Gradle installed. If not yet installed, you can run the following to install JDK 1.8 (openjdk-8-jdk) and Gradle:

1. sudo apt-get install openjdk-8-jdk-headless
2. sudo apt-get install gradle

Then Set up Service Fabric development environment. The Service Fabric SDK for Linux includes a Yeoman generator to provide the scaffolding for a Service Fabric application with a stateless service. Start by running the following Yeoman command:

$ yo azuresfjava

Following structure is created.

HelloWorldApplication/

├── build.gradle

├── HelloWorld

│ ├── build.gradle

│ └── src

│ └── statelessservice

│ ├── HelloWorldServiceHost.java

│ └── HelloWorldService.java

├── HelloWorldApplication

│ ├── ApplicationManifest.xml

│ └── HelloWorldPkg

│ ├── Code

│ │ ├── entryPoint.sh

│ │ └── \_readme.txt

│ ├── Config

│ │ └── \_readme.txt

│ ├── Data

│ │ └── \_readme.txt

│ └── ServiceManifest.xml

├── install.sh

├── settings.gradle

└── uninstall.sh

Service registration is performed in the process main entry point. In this example, the process main entry point is HelloWorldServiceHost.java:

public static void main(String[] args) throws Exception {

try {

ServiceRuntime.registerStatelessServiceAsync("HelloWorldType", (context) -> new HelloWorldService(), Duration.ofSeconds(10));

logger.log(Level.INFO, "Registered stateless service type HelloWorldType.");

Thread.sleep(Long.MAX\_VALUE);

}

catch (Exception ex) {

logger.log(Level.SEVERE, "Exception in registration:", ex);

throw ex;

}

}

To run the application, first build the application with gradle:

$ gradle

This produces a Service Fabric application package that can be deployed using Service Fabric CLI.

Run the install.sh script to deploy the application.

$ ./install.sh

# Summary and Conclusion

Service Fabric offers many out of the box features that provide a highly available, configurable, and scalable environment to deploy microservices. In addition to all of these features, the ability to run Service Fabric almost anywhere provides anyone with the ability to integrate it into their existing infrastructure. Managing the infrastructure for microservices via a PaaS solution like Service Fabric enables developers to construct a solution with minimal effort and minimum future maintenance

**Appendices/ References**

<https://dzone.com/articles/monolithic-vs-microservice-architecture>

<https://azure.microsoft.com/en-in/blog/migrating-a-java-spring-framework-application-to-windows-azure/>

<https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-migrate-old-javaapp-to-use-maven>

<https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/service-fabric/service-fabric-cloud-services-migration-differences.md>