

**Quick Start Tutorial**

**Paweł Adamus**

Spis treści:

[1. Co potrzebujemy: 1](#_Toc41094136)

[2. Instalacja: 2](#_Toc41094137)

[3. Uruchomienie Azure Fabric Services 3](#_Toc41094138)

[4. Stawiamy pierwszą apkę! 4](#_Toc41094139)

[5. Debugujemy i uruchamiamy pierwszy serwis 9](#_Toc41094140)

[6. Wstęp do Multi-Node Stateless Service 13](#_Toc41094141)

[7. Stateless API 19](#_Toc41094142)

[8. Stateful Service Partitioning 24](#_Toc41094143)

[9. Stateful Service Named Partitioning 32](#_Toc41094144)

[10. Ustawianie komunikacji między serwisami 34](#_Toc41094145)

[11. Komunikacja między serwisami 37](#_Toc41094146)

[12. Komunikacja z Partitioned Services 40](#_Toc41094147)

[13. Reliable Collections: Reliable Dictionary 41](#_Toc41094148)

[14. Reliable Collections: Reliable Queues 45](#_Toc41094149)

[15. Reliable Actors: Actor State & Communication 48](#_Toc41094150)

[16. Reliable Actors: Actor Life Cycle 56](#_Toc41094151)

[17. Reliable Actors: Actor Timers & Reminders 58](#_Toc41094152)

[18. Data Packages, konfiguracja I zmienne środowiskowe 63](#_Toc41094153)

# Co potrzebujemy:

* Windows 10 lub Server 2016+
* .Net Core 3.0 SDK
* Visual Studio 2019 Community z zainstalowanym:
  + ASP.NET and web development
  + Azure development
  + Node.js development
  + .NET Core cross-platform development
* Service Fabric SDK

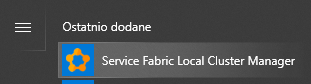
# Instalacja:

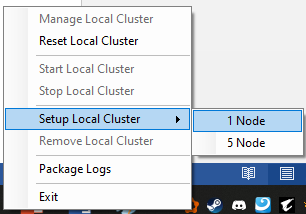
Zakładam, że każdy wie jak zainstalować VS Studio więc zamieszczam linki do:

.NET Core 3.0+ : <https://dotnet.microsoft.com/download/dotnet-core/3.1>

Service Fabric SDK: <https://docs.microsoft.com/pl-pl/azure/service-fabric/service-fabric-get-started>

# Uruchomienie Azure Fabric Services

* 1. Uruchamiamy zainstalowany Service Fabric Local Cluster Manager:  
      
  2. Następnie klikamy prawym ikonę w Tray’u:  i wybieramy:



W komunikacie klikamy „Tak” i nasz lokalny Cluster powinien wystartować.

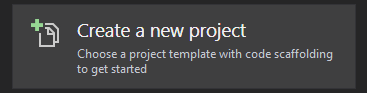
* 1. Następnie w oknie przeglądarki internetowej wpisujemy: **localhost:19080**



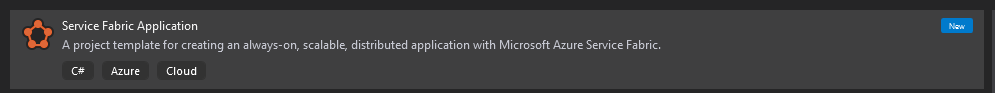
Na lewo możemy rozwinąć listę co przechowujemy w Klastrze.

# Stawiamy pierwszą apkę!

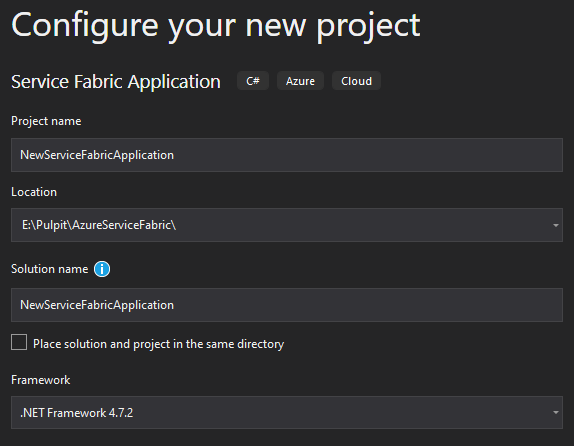
* 1. Uruchamiamy Visual Studio Code 2019 i wybieramy nowy projekt:



Następnie, wybieramy z listy: **Service Fabric Application**:

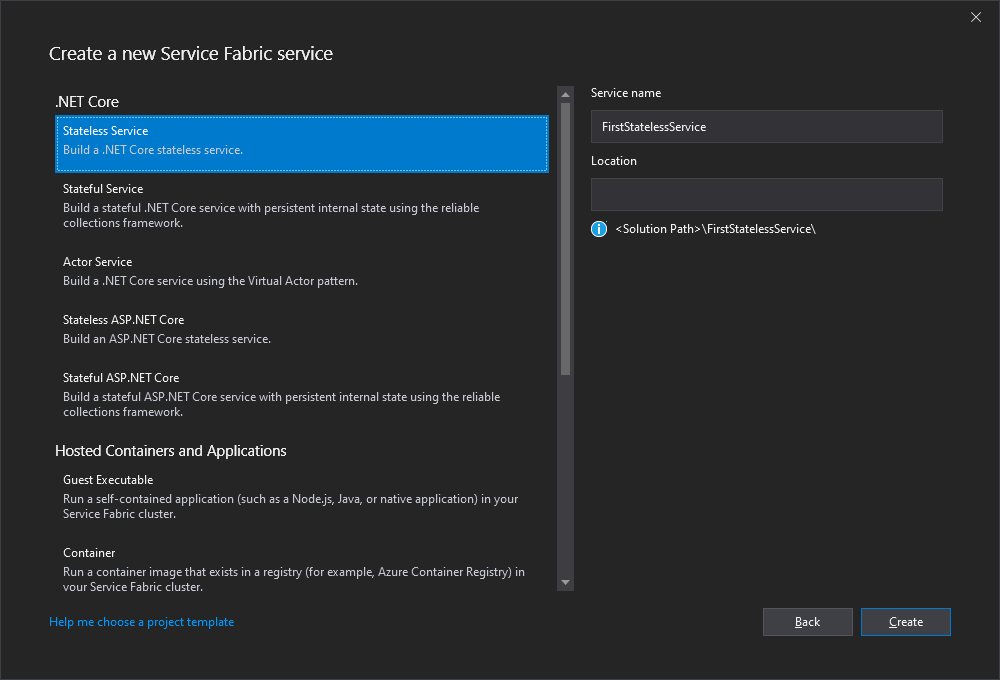


Następnie nazywamy swój projekt i wybieramy framework, moje ustawienia:



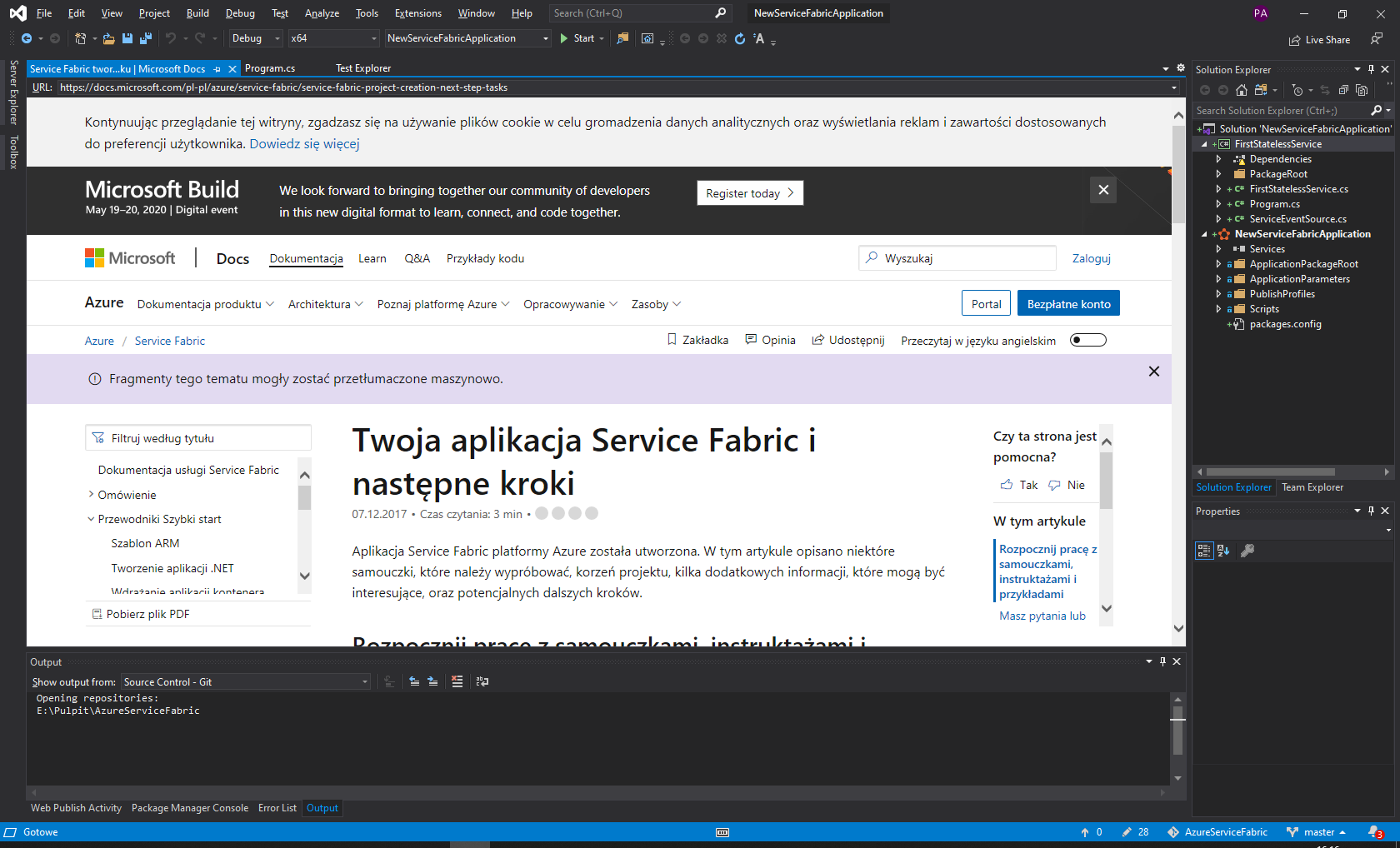
Halo halo! Ale po co był instalowany .NET Core 3.0 w takim razie?? Ano spokojnie, aplikację sobie postawimy na zwykłym .NET Frameworku, a w jej środku serwisy już będą hulały z .NET Core-em 3.0. Wszystko jest okej. ;)

Następnie wybieramy jaki typ serwisu będzie w naszej aplikacji:

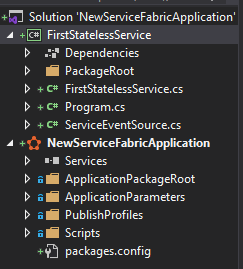


Na razie wybieramy **Stateless Service** i nazywamy go wg uznania i klikamy Create.

* 1. Jeśli nic nie schrzaniliśmy, to powinno się nam ukazać naszym oczom to:

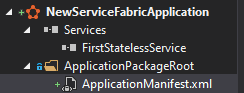


Czyli utworzona apka z serwisem i wyświetloną dokumentacją.



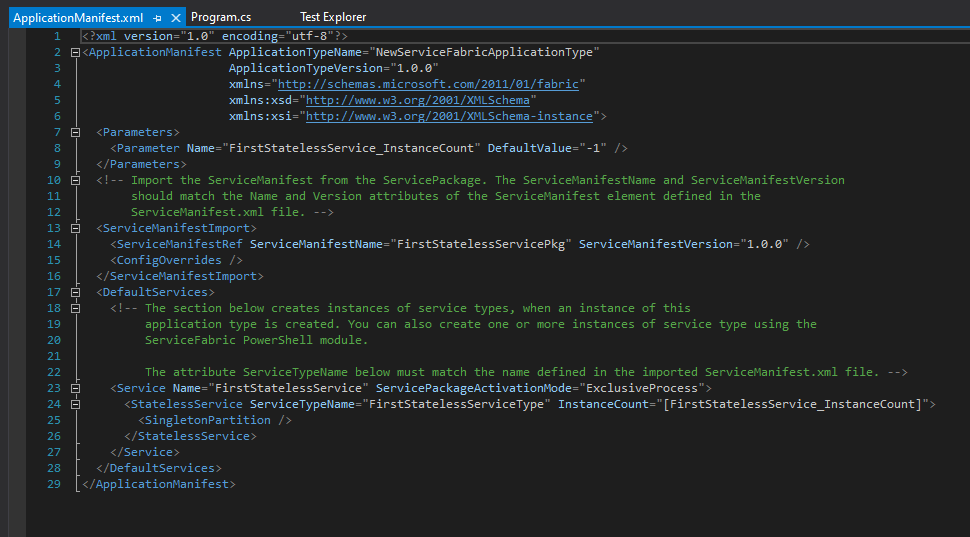
Jak widać solucja 2 projekty:

* FirstStatelessService
* NewServiceFabricApplication



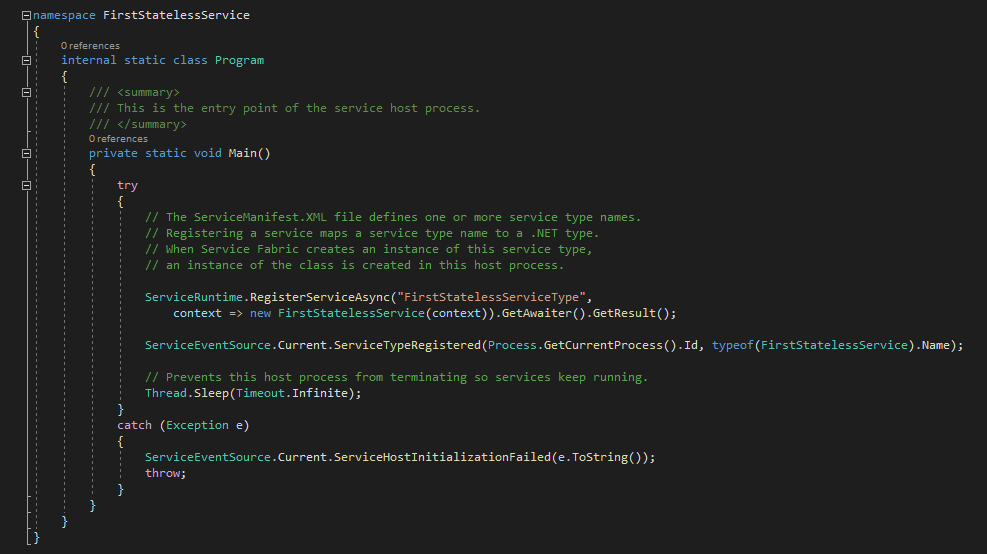
Jak widać do Projektu aplikacji mamy podpięty nasz pierwszy serwis.

* 1. Zajmijmy się jeszcze na chwilę plikiem .xml z ApplicationPackageRoot:

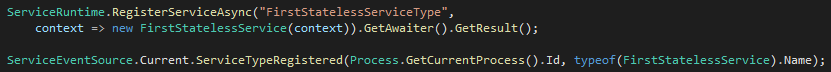


Co tutaj jest ciekawego? Ano tu mamy zawarte wszystkie szczegóły i referencje naszych podpiętych serwisów do Aplikacji.

* 1. Dobrze byłoby się też przyjrzeć Program.cs w naszym serwisie:



Co tu jest ciekawego:

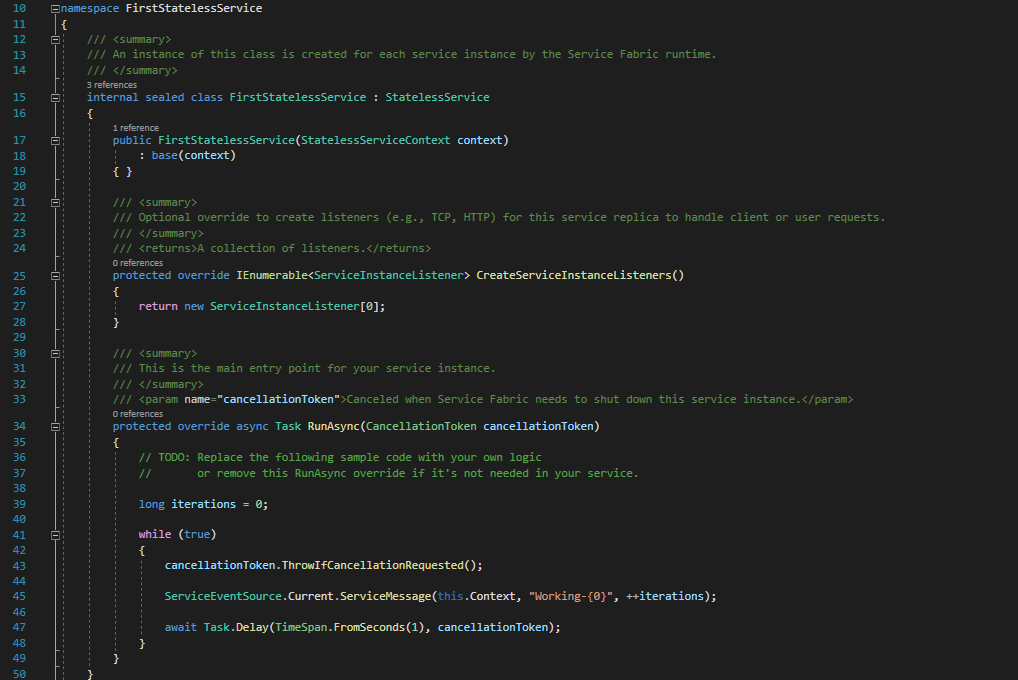


Odpowiada za rejestrowanie serwisu w Service Fabric.

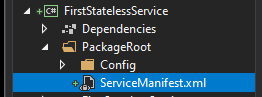


„Śpimy na wątku” bez timeouta – czyli nasz serwer będzie ciągle aktywny. ☺

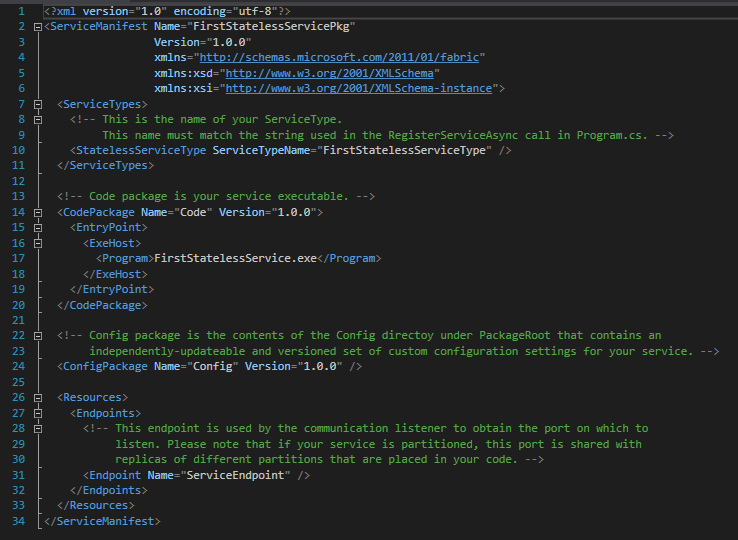
* 1. No dobra, a co zawiera nasz .cs z Stateless Service’a?



* Dziedziczy z StatelessService (w końcu taki typ serwisu wybraliśmy)
* Domyślny konstruktor
* Metody
  + CreateServiceInstanceListeners() – zezwala na komunikację pomiędzy różnymi Service Fabric-ami odpalonymi na klastrze
  + RunAsync() – odpala wszystkie taski które chcemy żeby były odpalone (mogą być też taski wykonywane w tle, start-up’y) – więc je tam wrzucamy
    - Pętla while odpowiada za to, że serwis będzie działał nieprzerwanie, chyba że trafi token który przerwie działanie serwisu. W środku pętli też jest ServiceEventSource, który robi log naszych komunikatów
  1. Tak samo nasz serwis posiada plik manifest .xml z całą konfiguracją.



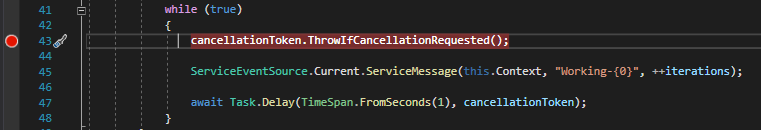
Zawarte jest w nim między innymi:



* Typ serwisu
* Wersja kodu
  + Nazwa pliku wykonywalnego serwisu który będzie uruchomiony w klastrze
* Config

# Debugujemy i uruchamiamy pierwszy serwis

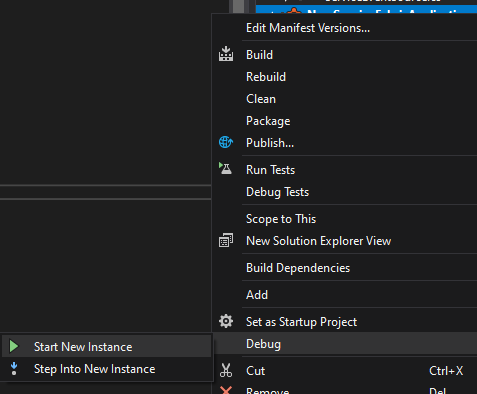
* 1. Wchodzimy do FirstStatelessService.cs i ustawiamy pułapkę:



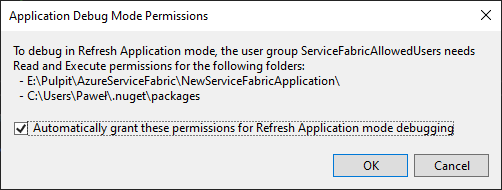
Klikamy prawym przyciskiem Projekt:



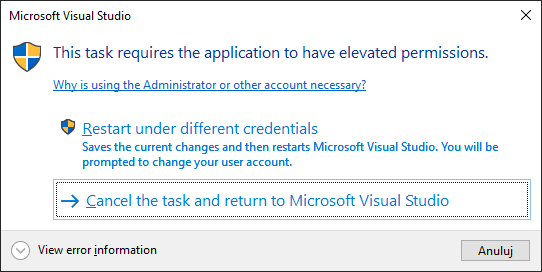
I wybieramy Debug -> Start New Instance:



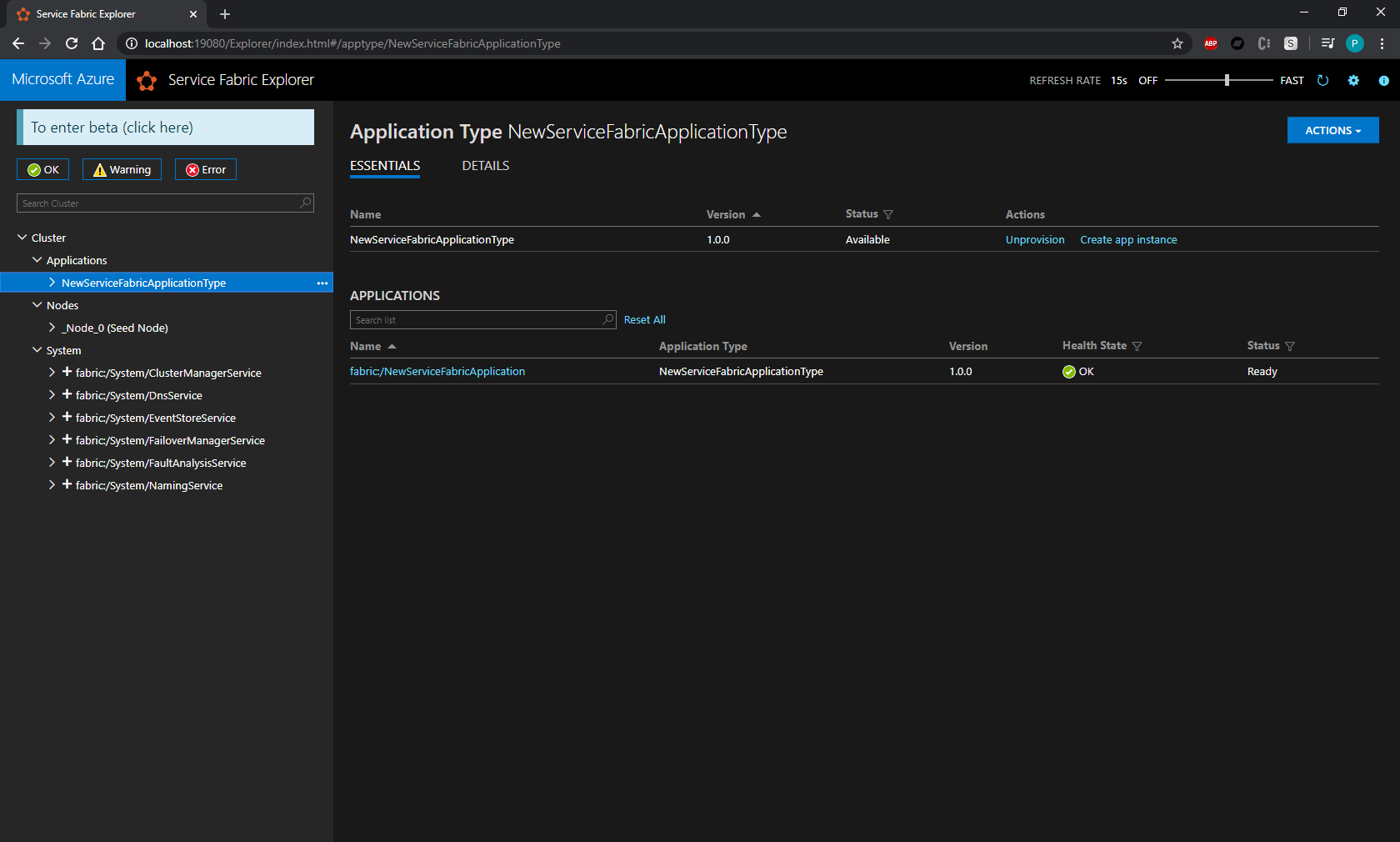
Może pojawić się okno:



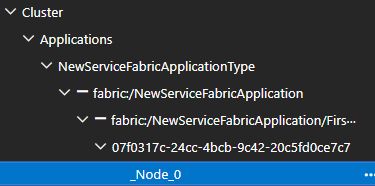
Dajemy Ok i jeśli pojawi się UAC dajemy OK ponownie



Wybieramy Restart under different credentials lub uruchamiamy VS z uprawnieniami Administratora.



W przeglądarce powinna pojawić się nasza Apka w klastrze.

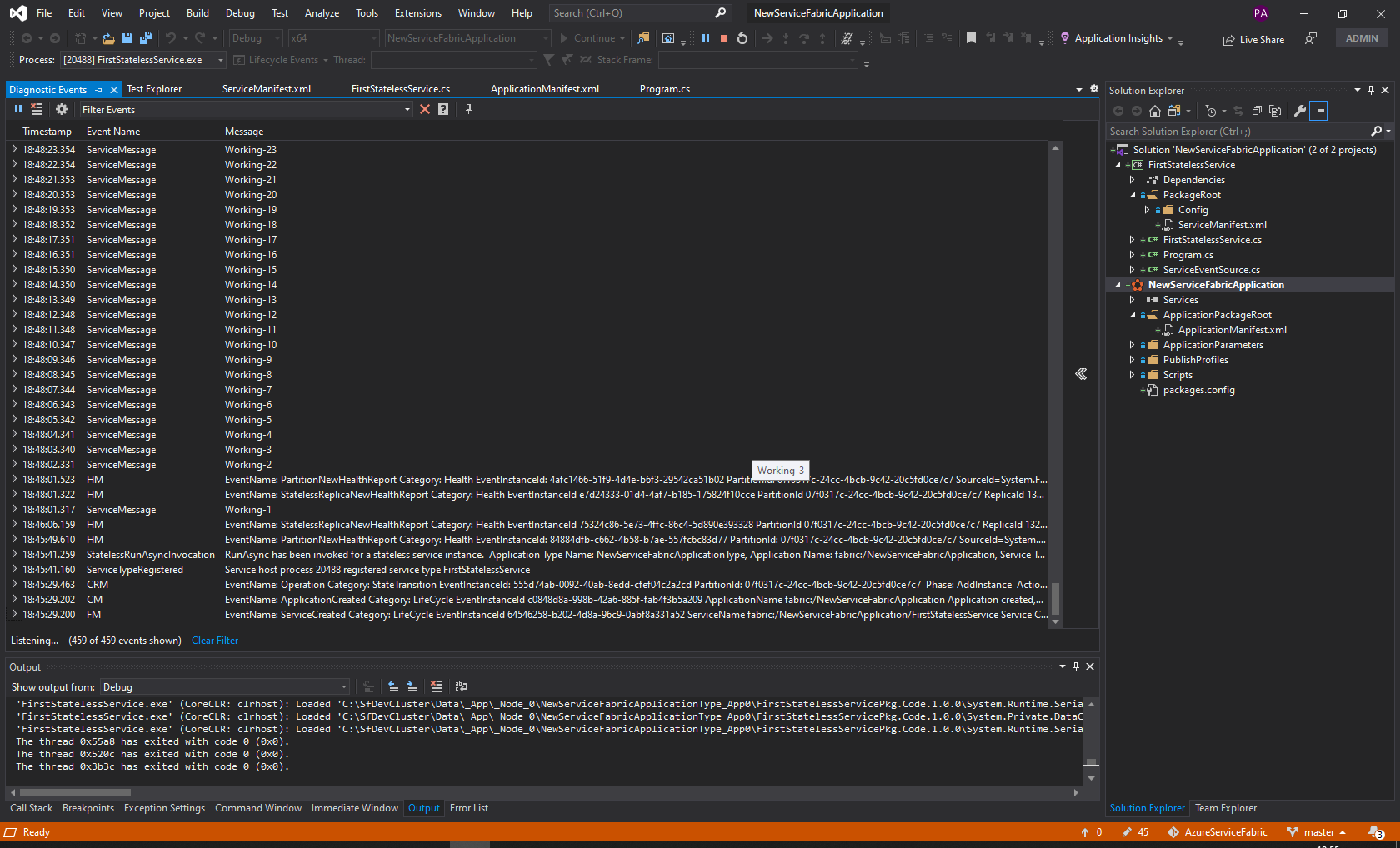


Struktura naszych serwisów w aplikacji w klastrze Azure Service Fabric:

Klaster/Aplikacje/TypAplikacji/Aplikacja/Partycja/Węzeł

Notka: Stateless Services mają zawsze tylko jedną partycję. Jak widać, na razie mamy tylko jedna replikę na na naszym węźle 0.

Możemy zwolnić teraz break-pointa, i przejść do Diagnostic Window w Visual Studio:



Jak widać asynchronicznie jest zapisywany log co sekundę ( wspomniana metoda wcześniej RunAsync() ).

* 1. Następnie przechodzimy do okna przeglądarki:

# Wstęp do Multi-Node Stateless Service

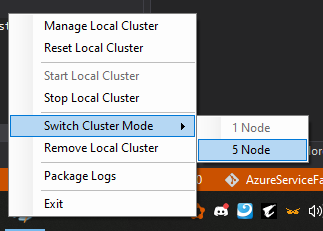
* 1. Uruchomimy teraz kika identycznych replik naszej instancji.

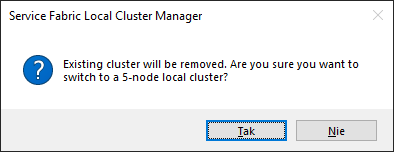
Zanim to jednak zrobimy, należy usunąć aplikacje z naszego single node klastra. Zobaczmy czy sobie poradzicie bez zrzutów. Jeśli ci się to uda – gratulacje, jeśli nie, przejdź do pkt. 6.2 trochę errorów dostaniesz i ciekawego info. ☺

Podpowiedź: robimy to w przeglądarce gdzie mamy odpalony Azure Fabric Server.

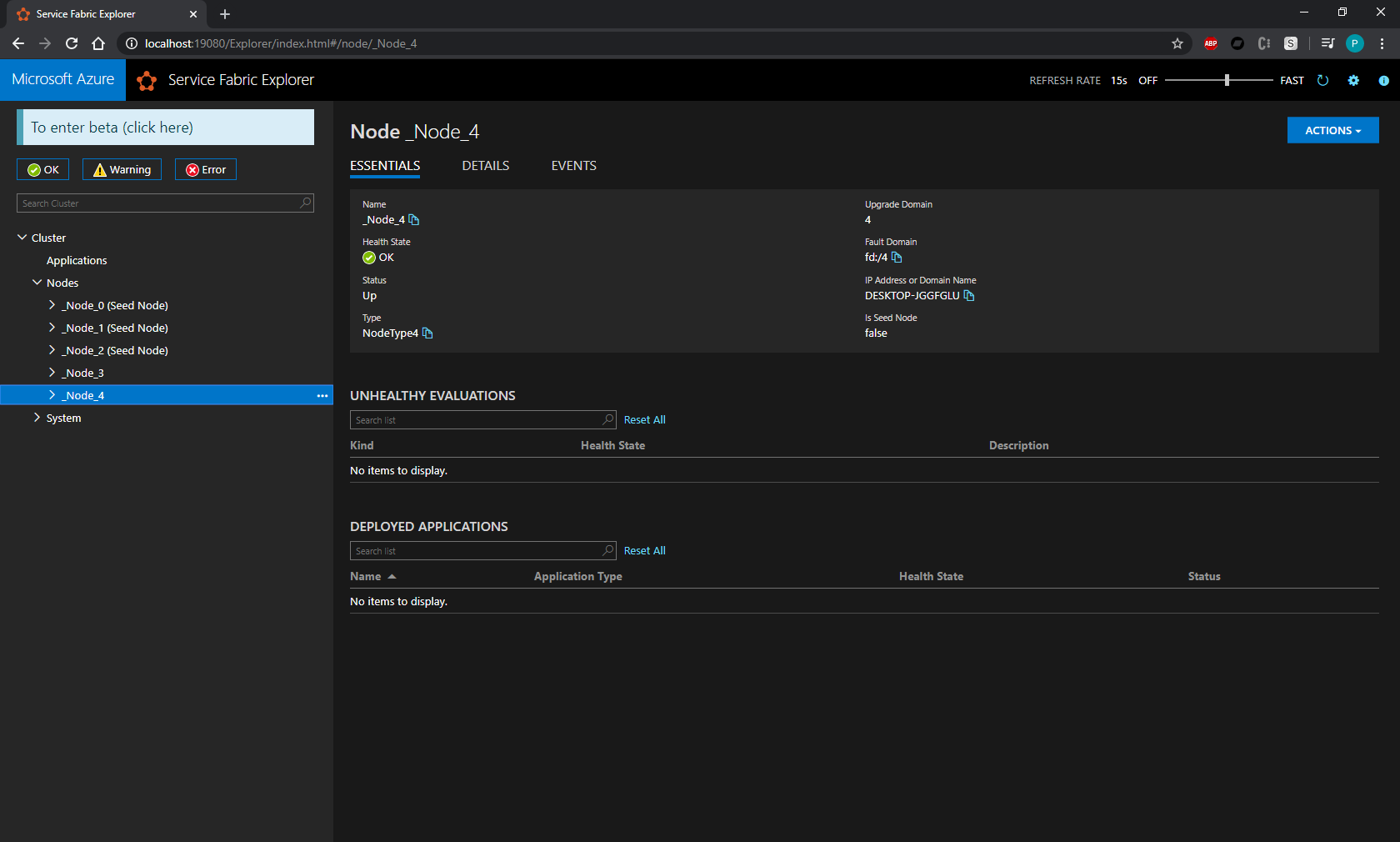
Podpowiedź 2: Należy najpierw usunąć fabric przed usuwaniem aplikacji.

* 1. Zmieniamy Cluster Mode z jednego node’a na 5:





Potwierdzamy



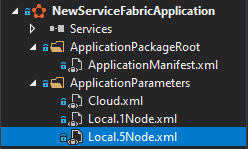
Możemy teraz operować na klastrze z 5 węzłami.

Gdy jedna z replik dostanie faila, pozostałe będą nadal działały i będą mogły przejąć taski z wykrzaczonego node’a. Jako że nasz serwis jest Stateless – nie musimy na razie się martwić o przekazywanie stanów pomiędzy naszymi replikami.

* 1. Wracamy do Visual Studio i zaczniemy modyfikować FirstStatelessService.cs. Zaczniemy od zmian w metodzie RunAsync() żeby log był trochę ciekawszy:

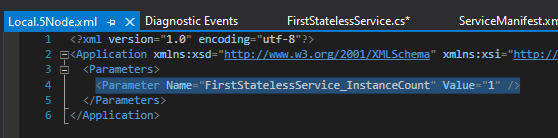
|  |
| --- |
| ServiceEventSource.Current.ServiceMessage(this.Context, $"Working-{iterations}, {this.Context.InstanceId}", ++iterations); |

Kiedy uruchomimy serwis, będzie on pobierał ustawienia z pliku z parametrami aplikacji:



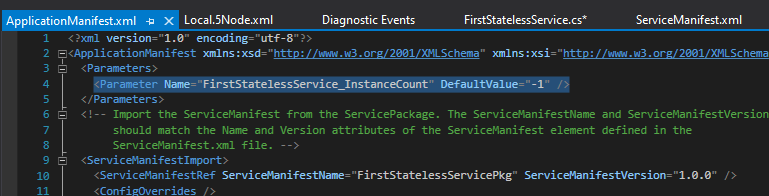
I jest to bardzo ważne – te konfigi definiują jak nasze aplikacje i serwisy będą uruchamiane na danym srodowisku (1 czy 5 węzłowym).

* 1. Zmieńmy ustawienia dla Local.5Node.xml:



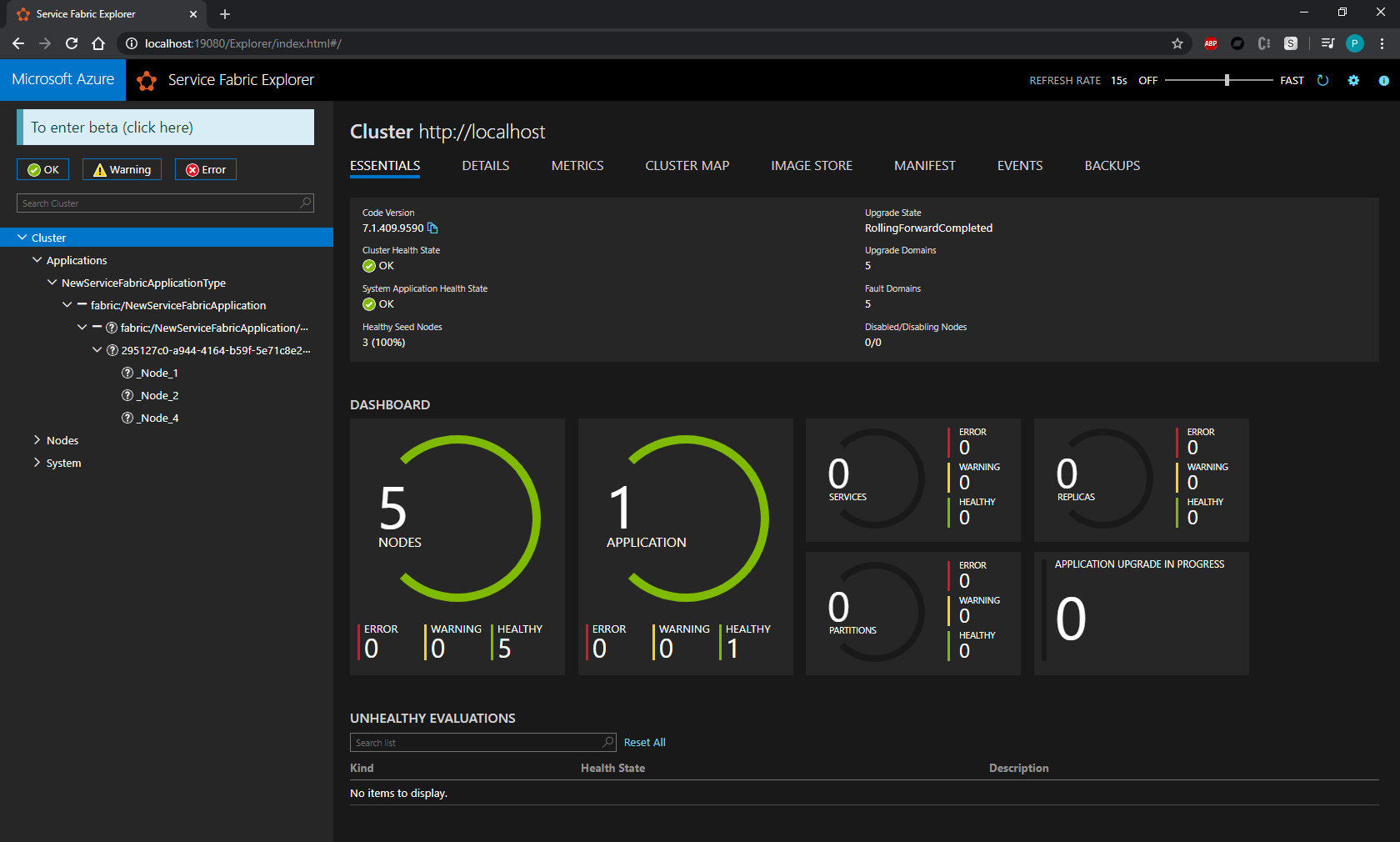
Jak widać domyślnie jest tworzona tylko jedna instancja. Zmieńmy wartość na 3.

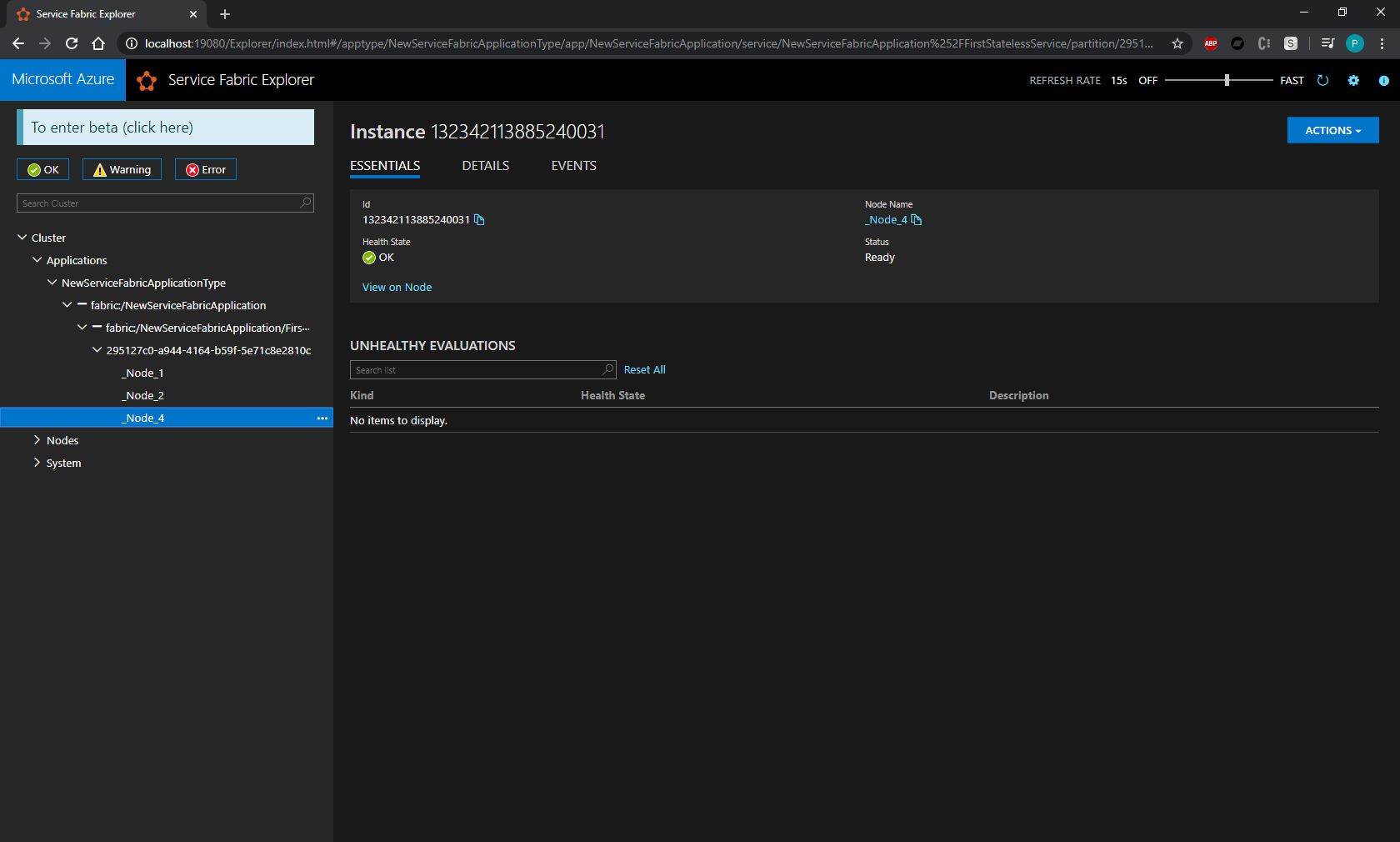
* 1. Te ustawienia są importowane do pliku manifest naszej aplikacji (ApplicationManifest.xml):



Można zauważyć, że wartość domyślna wynosi -1. Oznacza to, że serwis będzie wrzucony do każdego Node’a w klastrze. Można to zmodyfikować, ale pozostawię to dla zainteresowanych do poszukania w internetach. Domyślnie nasz serwis będzie zreplikowany do 3 węzłów.

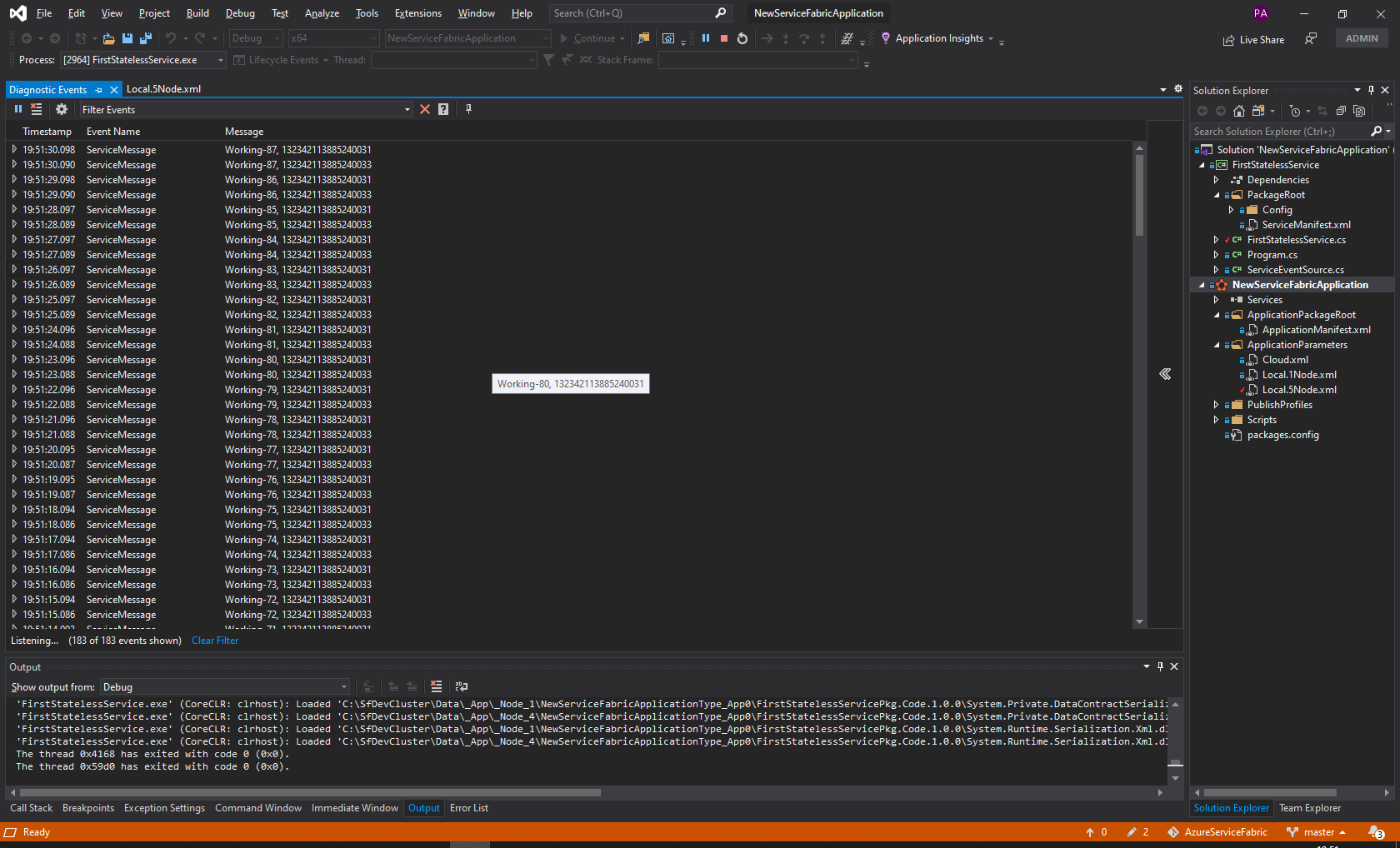
* 1. Sprawdźmy jak zdeployuje się nasza aplikacja. Klikamy prawym naszą aplikację -> Debug i Start New Instance.





Jak widać na partycji mamy teraz 3 node’y naszego serwisu.

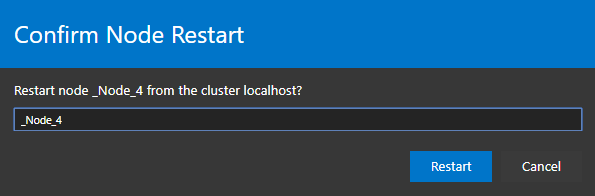
* 1. Sprawdźmy teraz okno Diagnostic Events w Visual Studio:

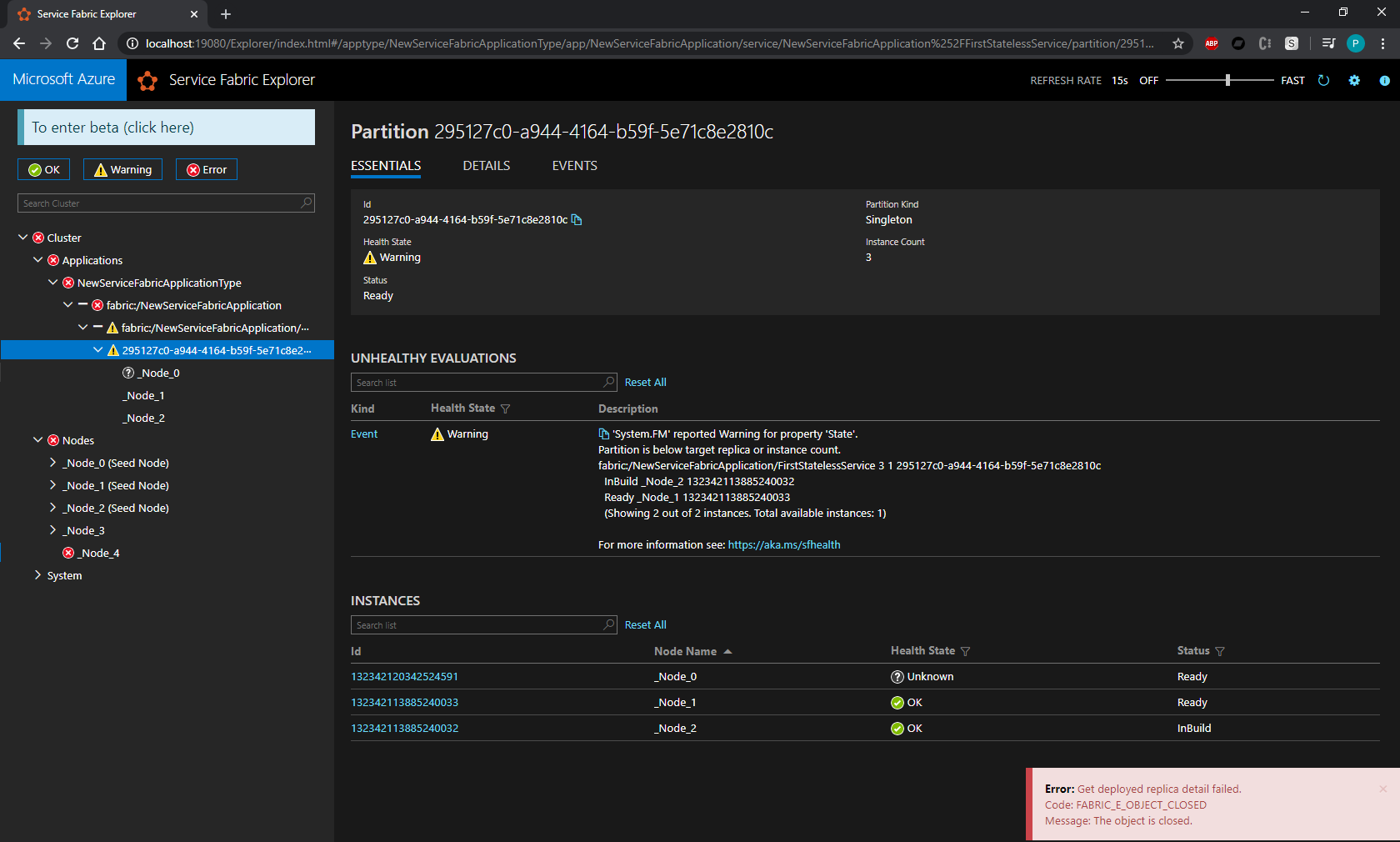


Jak widać w naszym zmodyfikowanym logu możemy teraz rozpoznać po ID który node robił wpis do loga. Widać także, że wszystkie 3 node’y pracują jednocześnie.

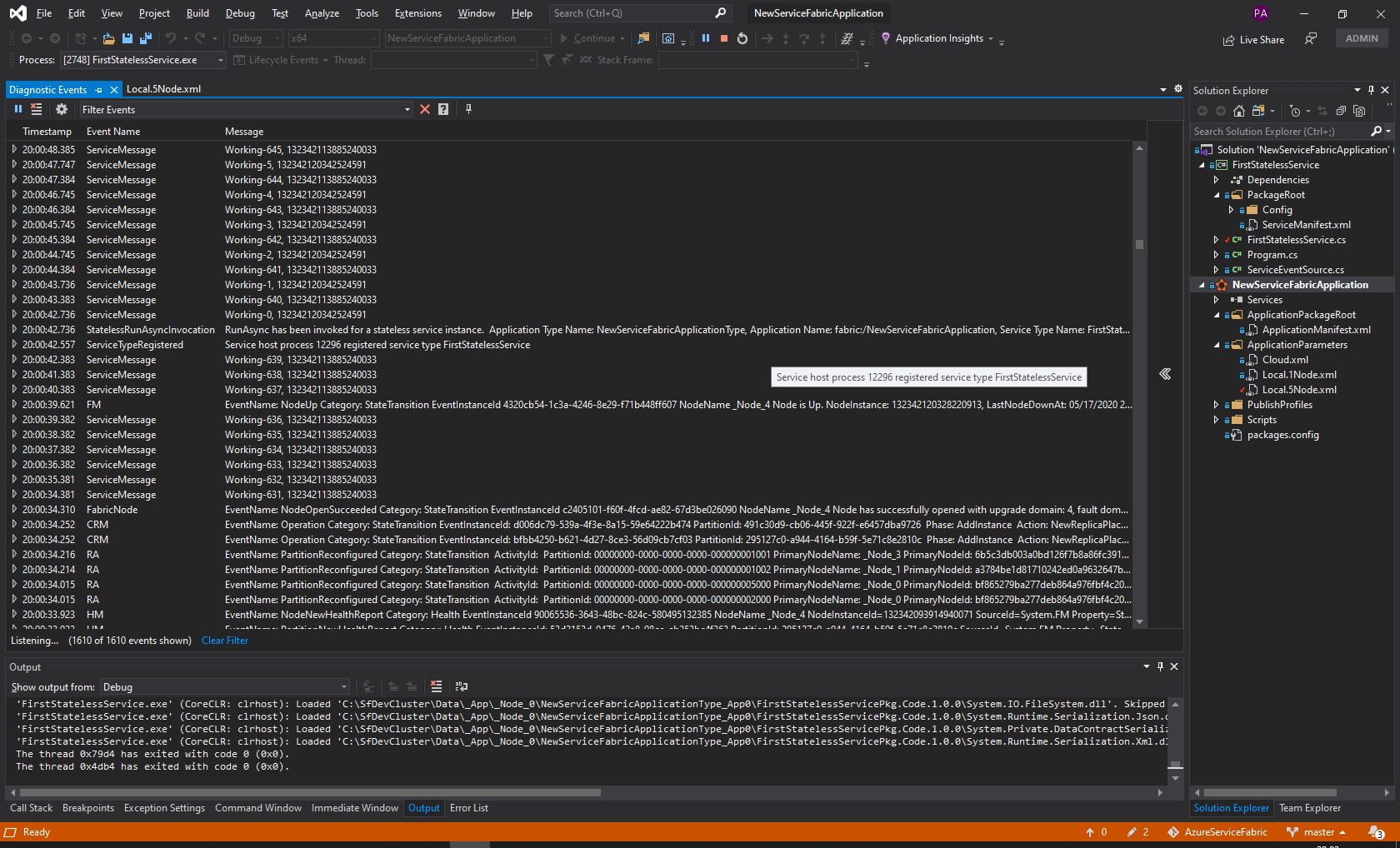
* 1. Zasymulujemy teraz ‘awarię’. Sprawdźmy co się stanie gdy w Service Fabric Explorerze w przeglądarce zrestartujmy np. Node 4:







Powinien pojawić się error message. Jak widać 2 pozostałe węzły nadal pracują.



Gdy wrócimy do naszego logu, widzimy że jeden wpis ma dużo mniejszy runtime (working). Jest to zrestartowany serwis na Node 4.

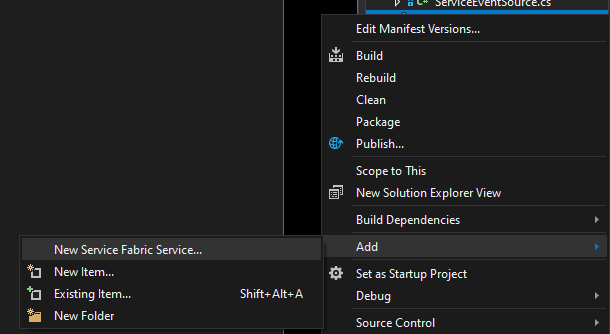
Krótkie wyjaśnienie:

Jest to Stateless Service – dlatego nie trackuje on restartów ani wartości które były w zmiennej „iterations”. Aby trackować takie informacje, zajmiemy się później State Services.

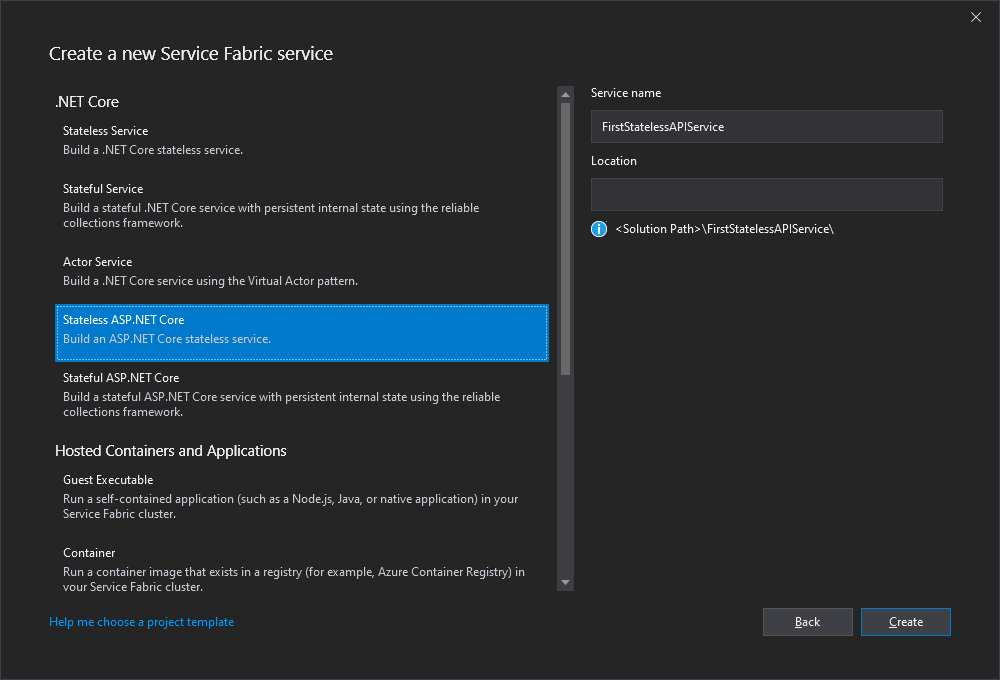
Sprawdź co stanie się gdy np. zrestartujesz Node 0.

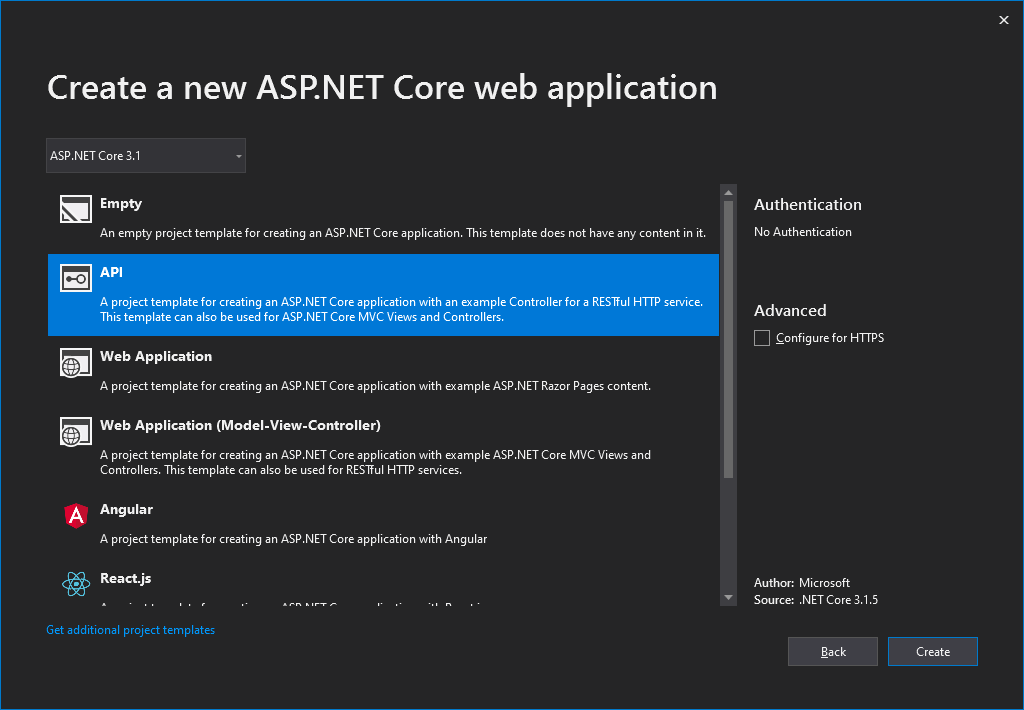
# Stateless API

* 1. Dodamy teraz kolejny typ serwisu – Stateless API. Aby to zrobić klikamy prawym naszą aplikację i wybieramy:

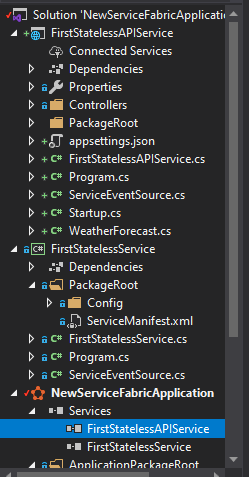


Następnie:





* 1. Jak widać nasz nowy Stateless API service został dodany do solucji:

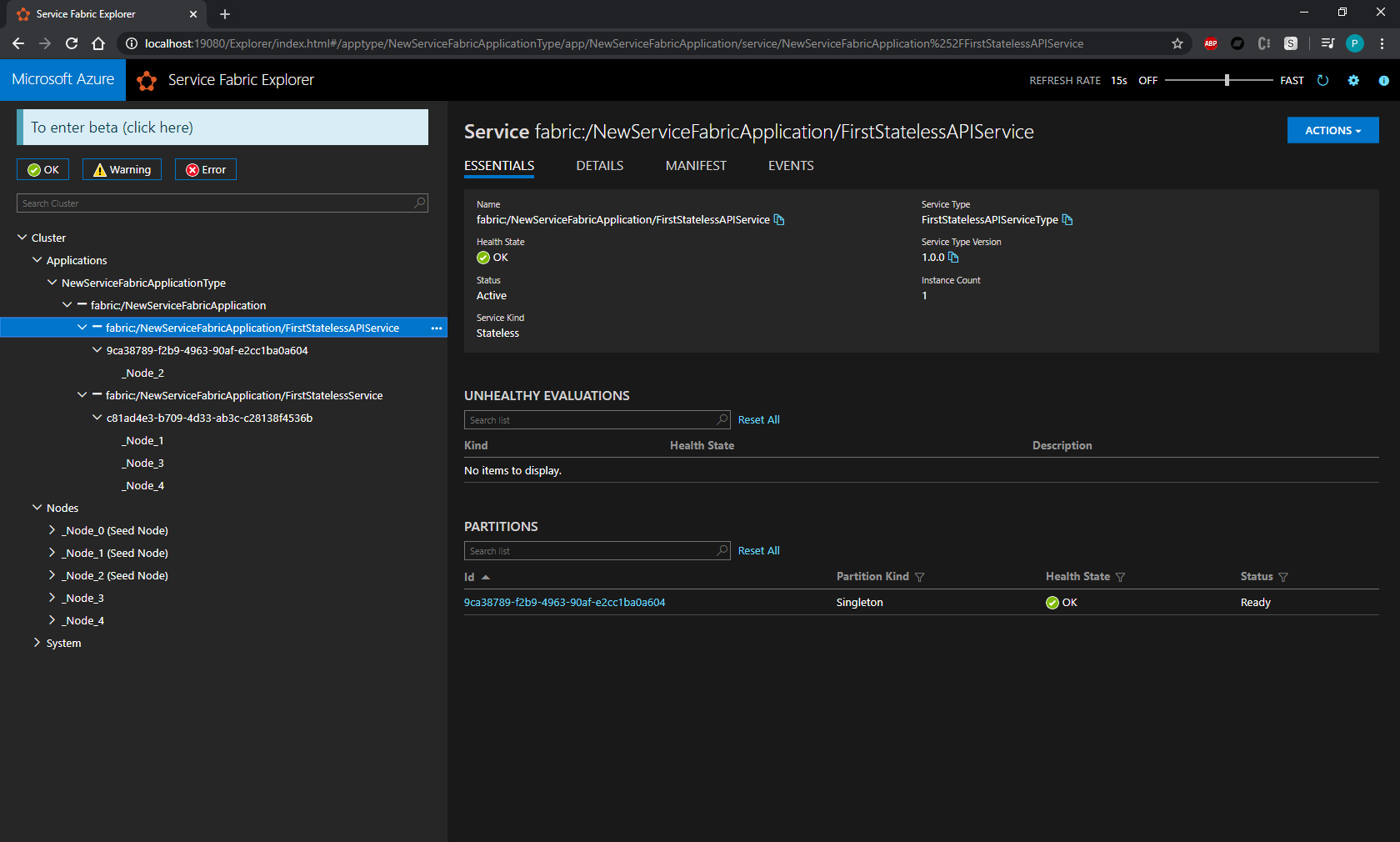


Plik manifest .xml naszej aplikacji również dodał konfiguracje naszego nowego serwisu. Sprawdźcie co zostało tam dodane.

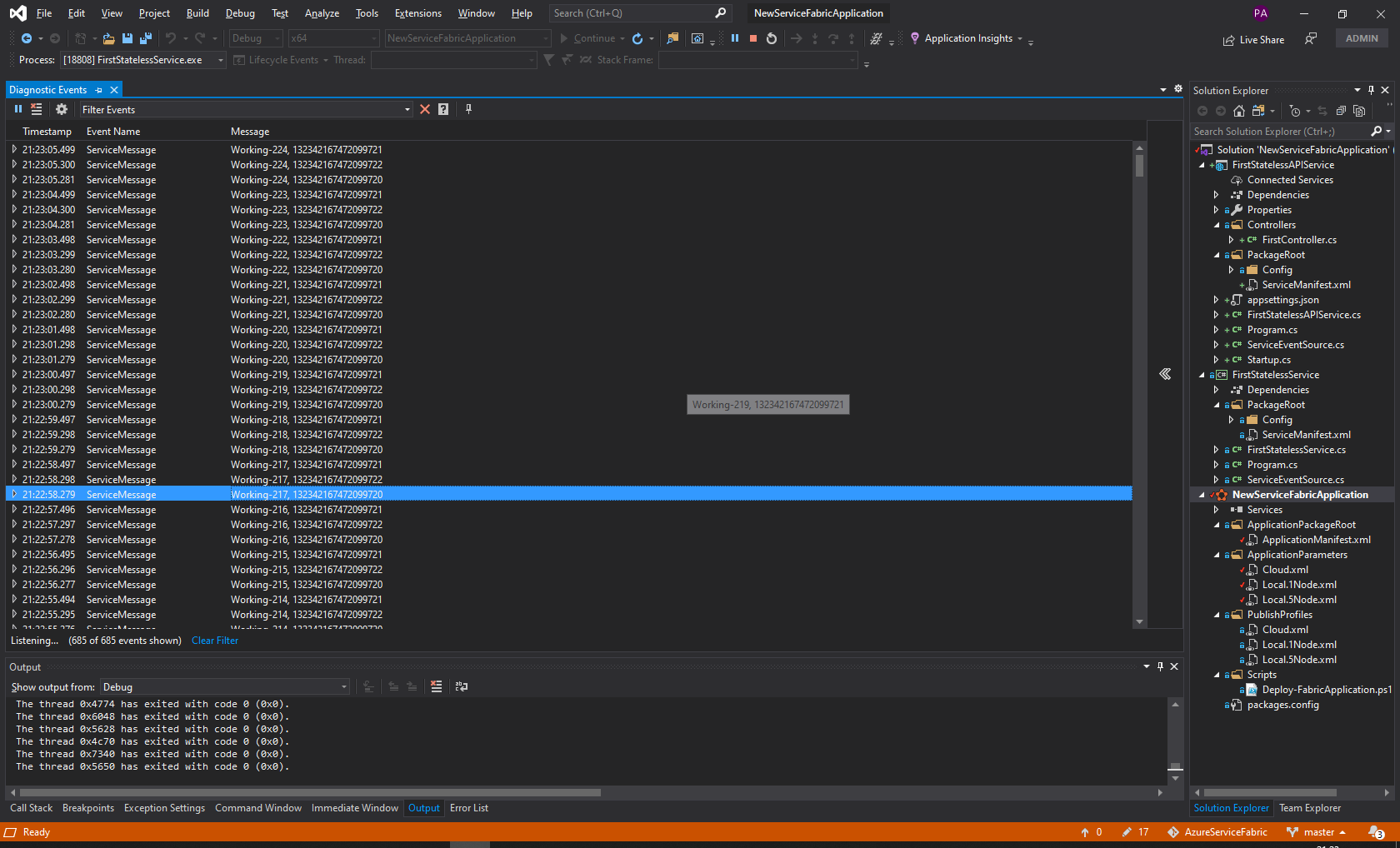
* 1. Nasz serwis utworzył też defaultowe śmieci – usuwamy WeatherForecast.cs.
  2. Można też zobaczyć .NET Core’owy Controller, który jest związany również z demo-danymi – zmieniamy jego nazwę na: FirstController.cs
  3. Następnie czyścimy zawartość i wklejamy:

|  |
| --- |
| namespace FirstStatelessAPIService.Controllers  {  [ApiController]  [Route("[controller]")]  public class FirstController : ControllerBase  {  [HttpGet]  [Route("get")]  public async Task Get() // method when that API endpoint will be called  {  await Task.Delay(TimeSpan.FromSeconds(1));  }  }  } |

* 1. Rebuildujemy apke, debugujemy jak poprzednio. Sprawdzamy deploy w przeglądarce:

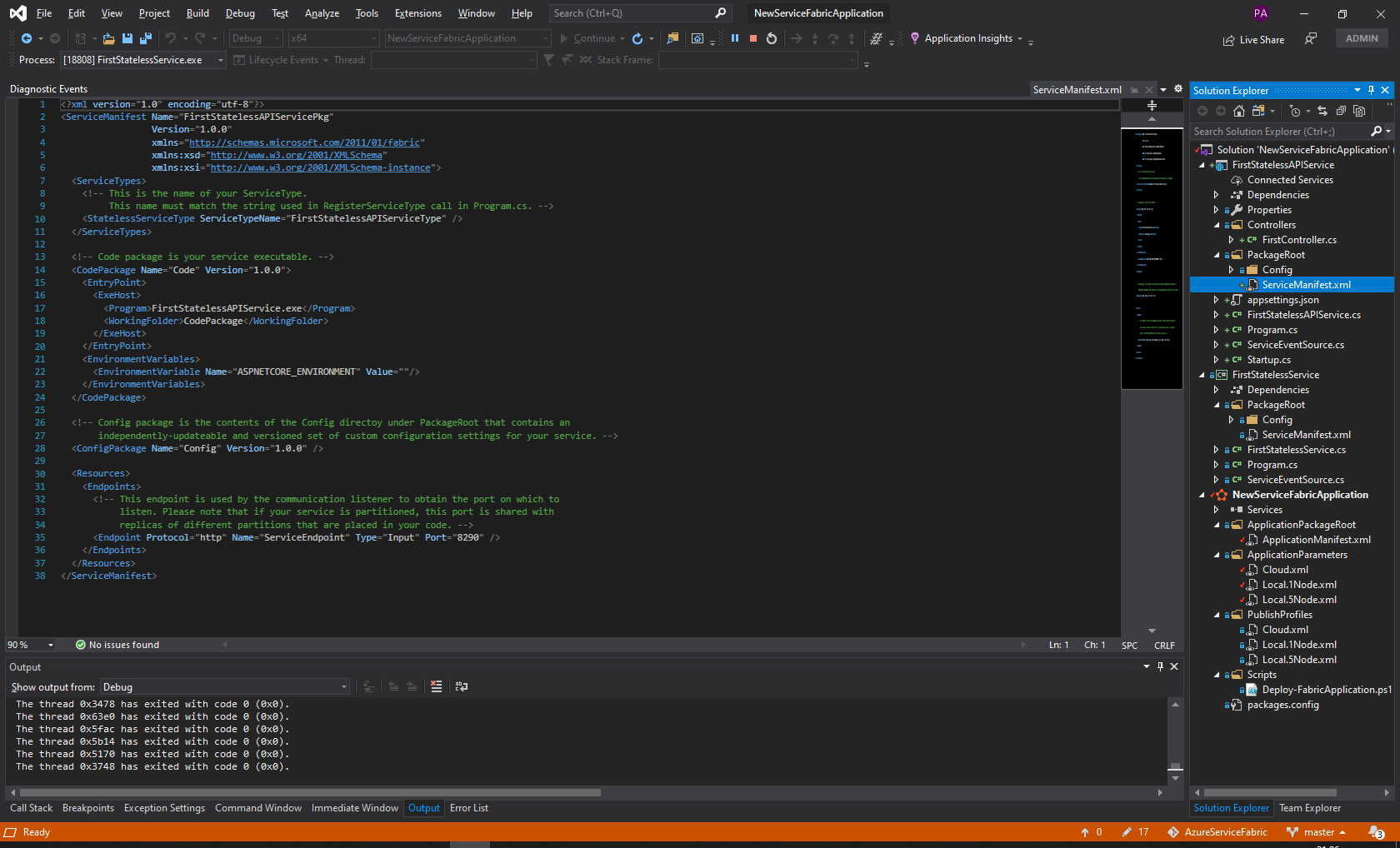


Jak widać nasze API jest na jednym node



Nasz serwis stateless API pojawia się także w logu.

Uwaga. W chmurze nasz serwis stateless API z konfiguracją w manifest -1 aplikacji, odpaliłoby się na każdym node! Niestety tutaj lokalnie jest restrykcja.

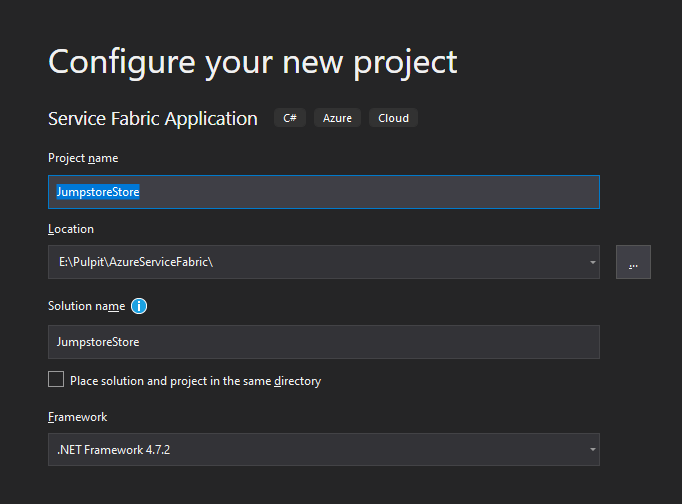
* 1. Przyjrzyjmy się jeszcze plikowi ServiceManifest.xml  
     

Jest tam określone na jakim porcie jest nasze API.

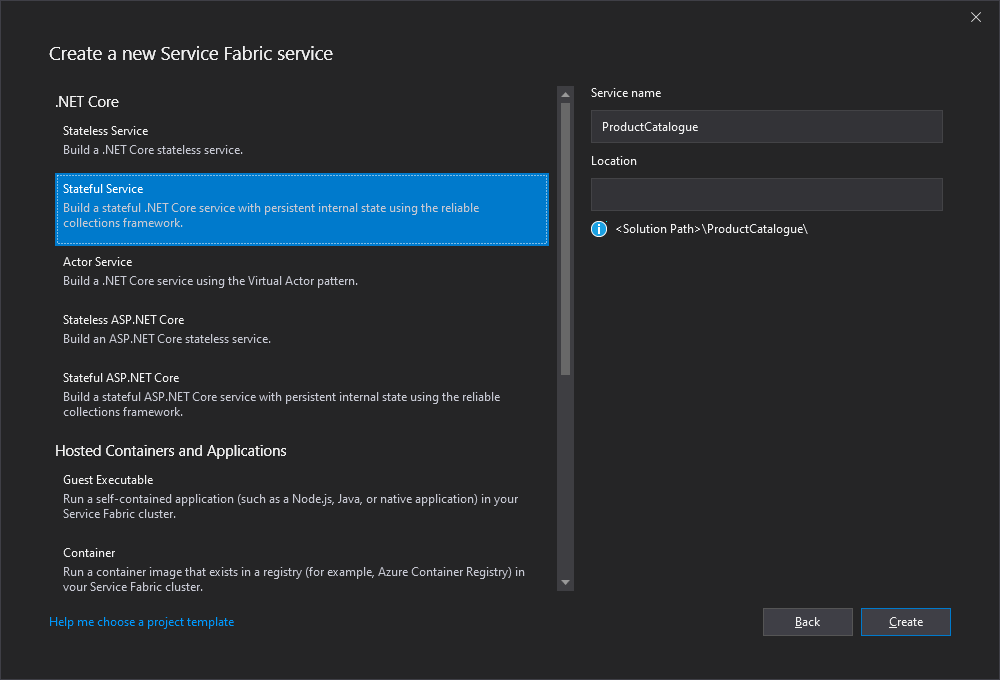
Spróbujcie sprawdzić <http://localhost:8290/first/get> odpowiadający waszemu endpointowi.

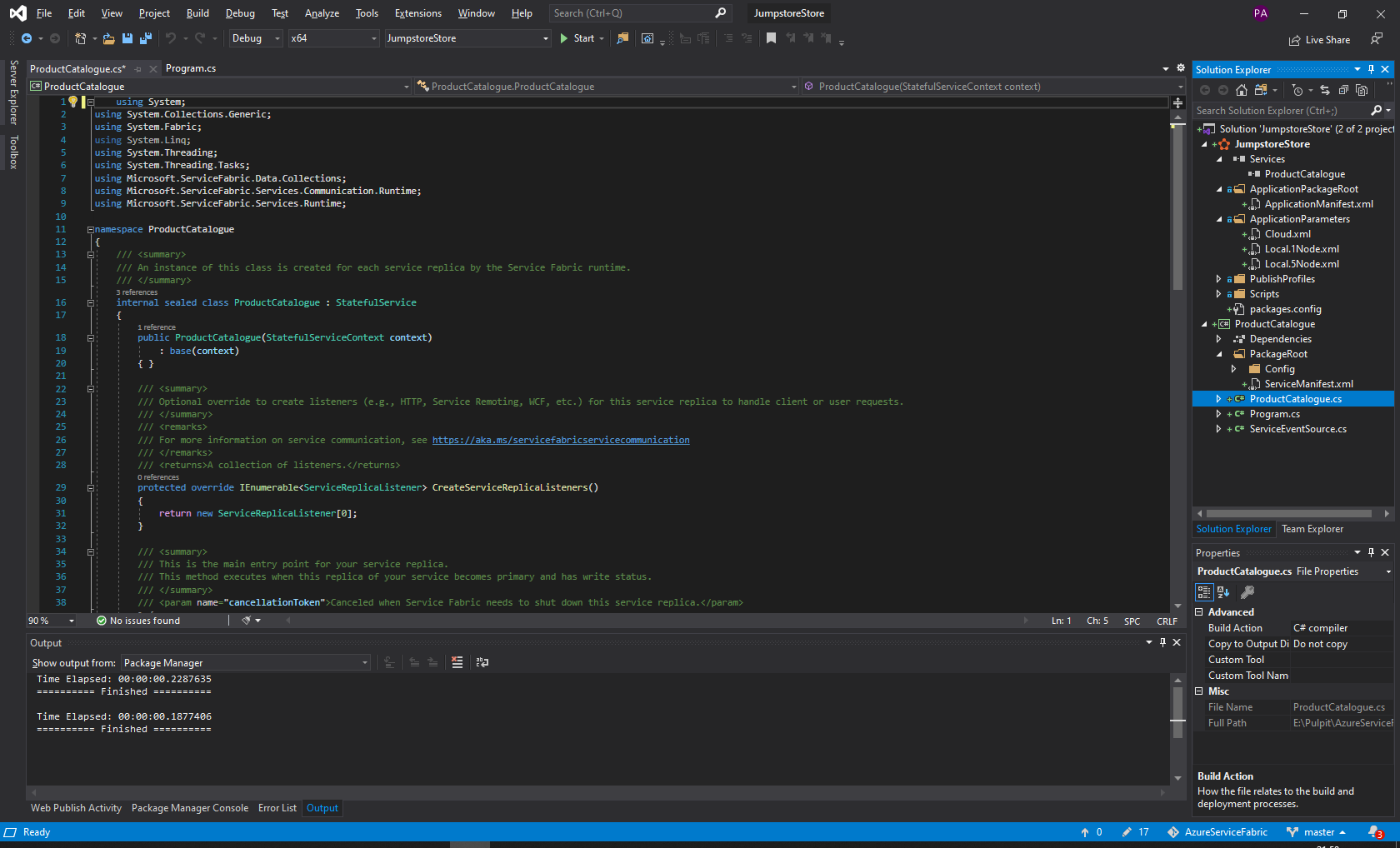
# Stateful Service Partitioning

* 1. Tworzymy całkiem nowy projekt. Wybieramy Service Fabric Application.



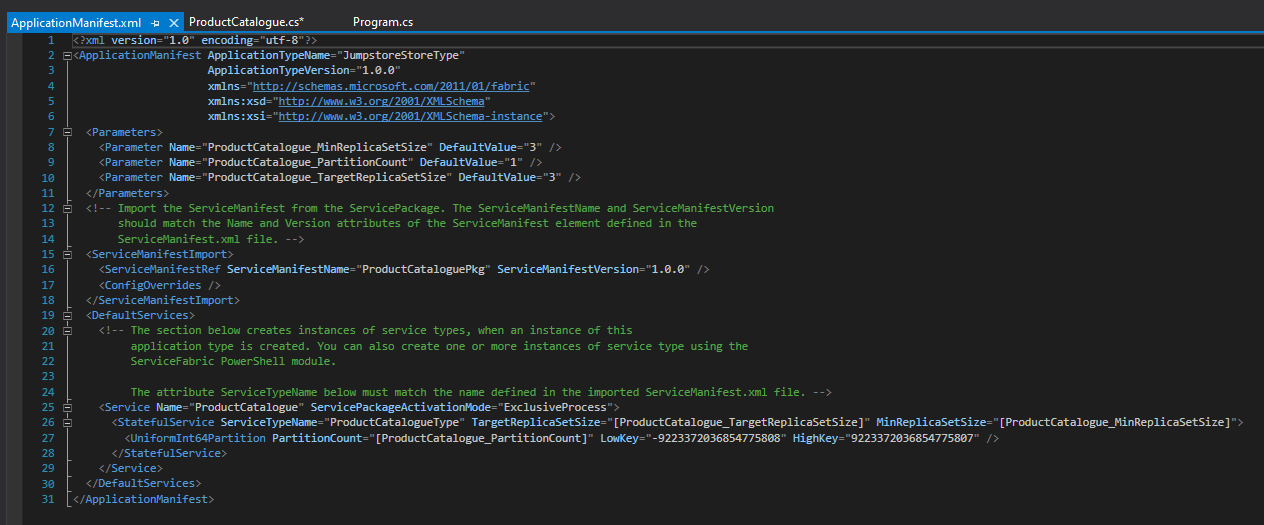
* 1. Wybieramy nasz pierwszy serwis:





Jak widać projekt jest bardzo podobny do poprzedniego, z tym że jest to teraz Stateful Service.

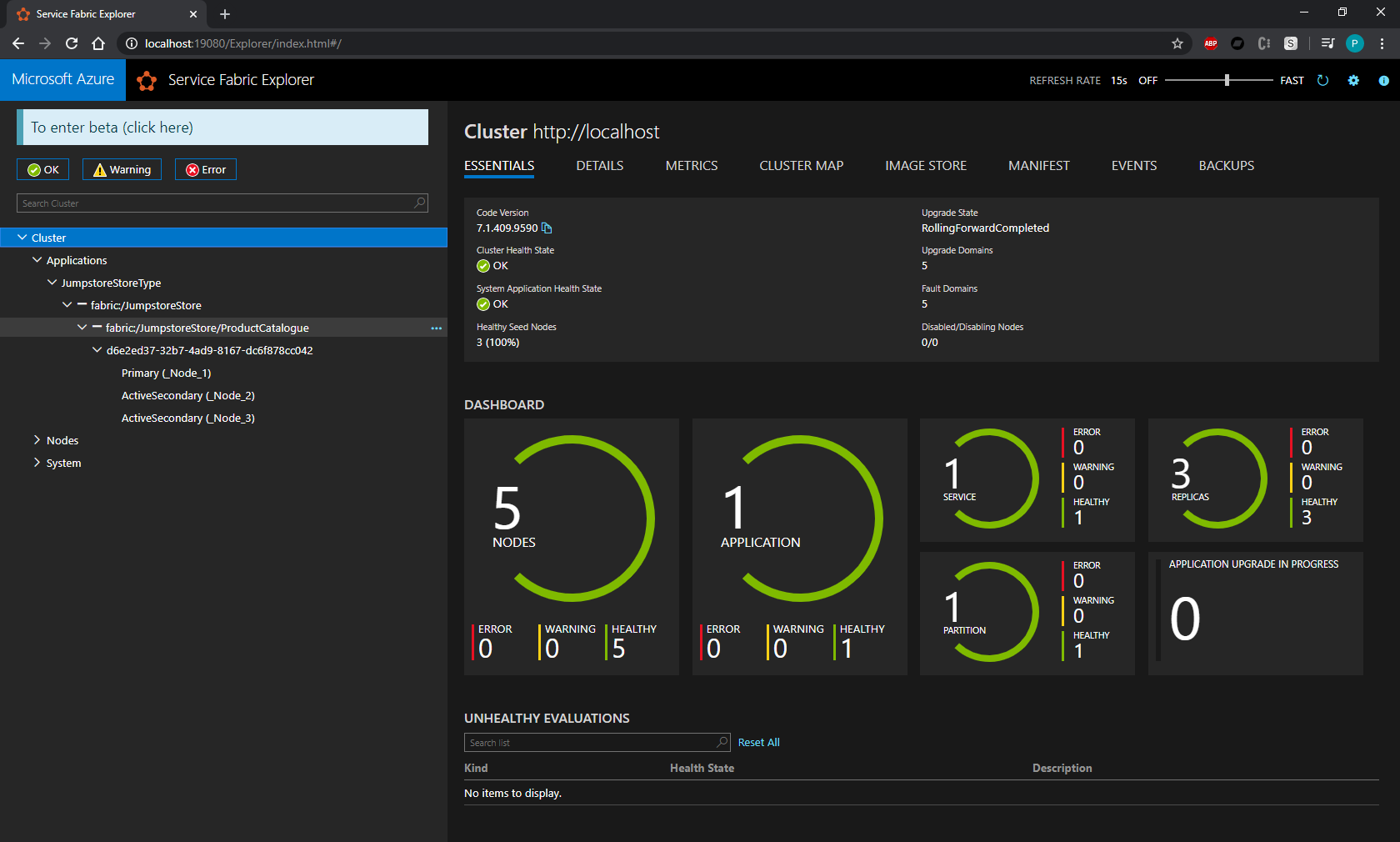
* 1. Sprawdźmy co zwiera plik manifest .xml naszej aplikacji:

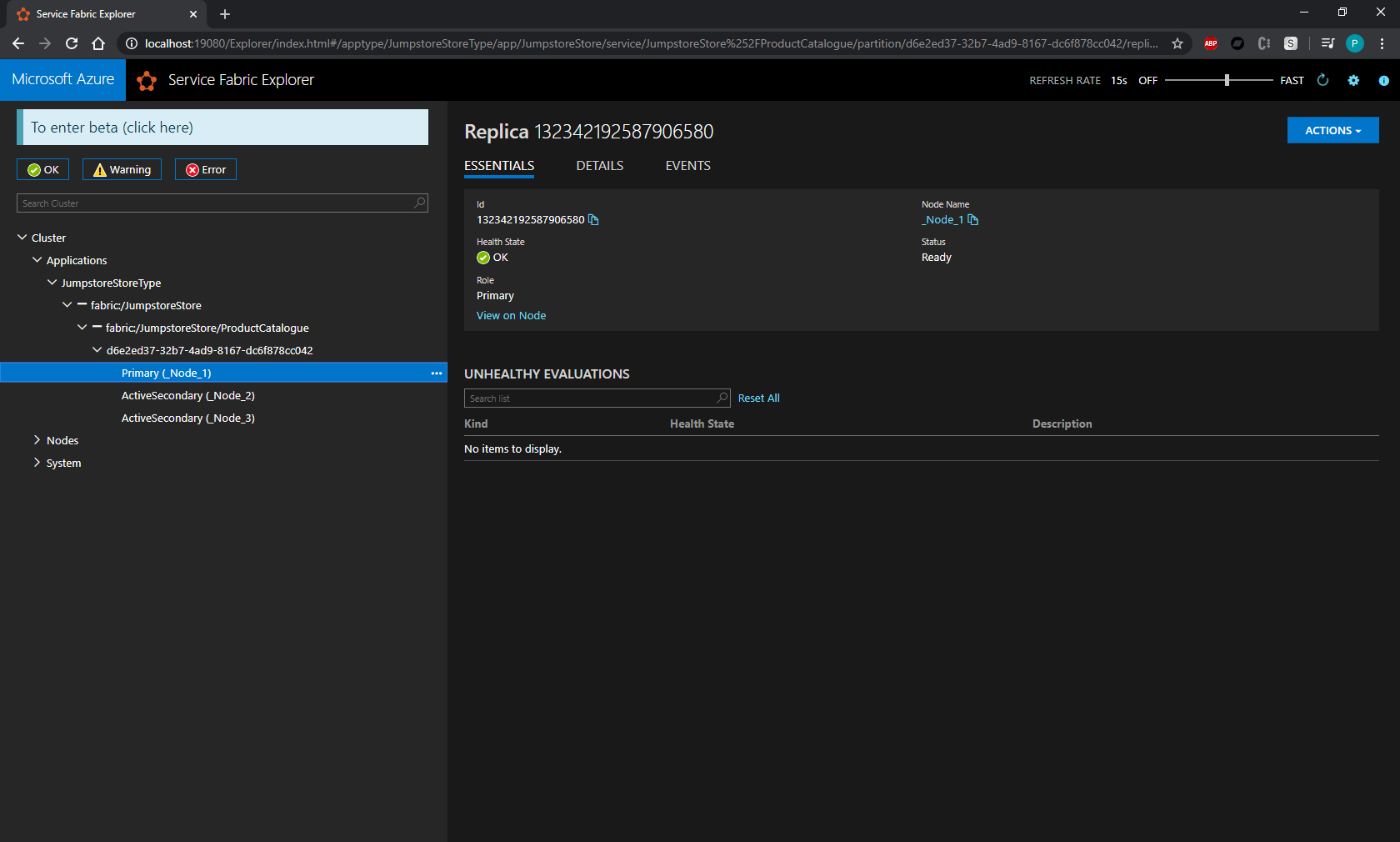


Widać ile powinno być minimum replik, partycji i docelowych replik dla naszego serwisu.

Uprośćmy trochę zmiennie usuwając ` ProductCatalogue\_` w pliku manifest i konfiguracjach lokalnych node’ów.

* 1. Następnie robimy deploy aplikacji i sprawdzamy w przeglądarce.





Jak widać, nasza aplikacja po deploy’u jest na jednej partycji z 3 replikami: primary i dwoma secondary.

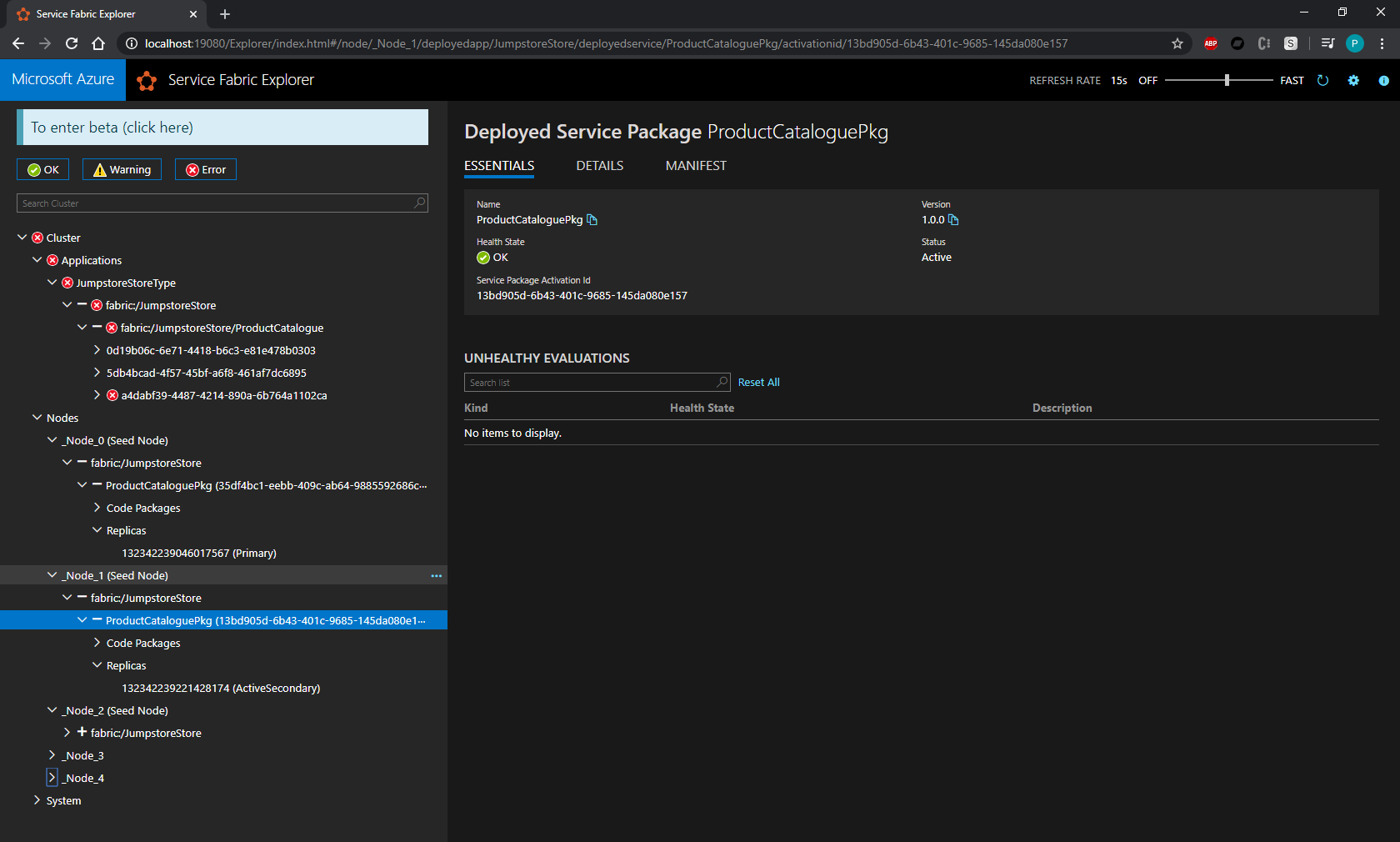
* 1. Zobaczmy jak zachowa się nasza aplikacja ze zmianami.

ApplicationManifest.xml:

|  |
| --- |
| <?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="JumpstoreStoreType" ApplicationTypeVersion="1.0.0" xmlns="http://schemas.microsoft.com/2011/01/fabric">  <Parameters>  <Parameter Name="MinReplicaSetSize" DefaultValue="3" />  <Parameter Name="PartitionCount" DefaultValue="3" />  <Parameter Name="TargetReplicaSetSize" DefaultValue="3" />  <Parameter Name="PartitionLowKey" DefaultValue="0" />  <Parameter Name="PartitionHighKey" DefaultValue="2" />  </Parameters>  <!-- Import the ServiceManifest from the ServicePackage. The ServiceManifestName and ServiceManifestVersion  should match the Name and Version attributes of the ServiceManifest element defined in the  ServiceManifest.xml file. -->  <ServiceManifestImport>  <ServiceManifestRef ServiceManifestName="ProductCataloguePkg" ServiceManifestVersion="1.0.0" />  <ConfigOverrides />  </ServiceManifestImport>  <DefaultServices>  <!-- The section below creates instances of service types, when an instance of this  application type is created. You can also create one or more instances of service type using the  ServiceFabric PowerShell module.    The attribute ServiceTypeName below must match the name defined in the imported ServiceManifest.xml file. -->  <Service Name="ProductCatalogue" ServicePackageActivationMode="ExclusiveProcess">  <StatefulService ServiceTypeName="ProductCatalogueType" TargetReplicaSetSize="[TargetReplicaSetSize]" MinReplicaSetSize="[MinReplicaSetSize]">  <UniformInt64Partition PartitionCount="[PartitionCount]" LowKey="[PartitionLowKey]" HighKey="[PartitionHighKey]" />  </StatefulService>  </Service>  </DefaultServices>  </ApplicationManifest> |

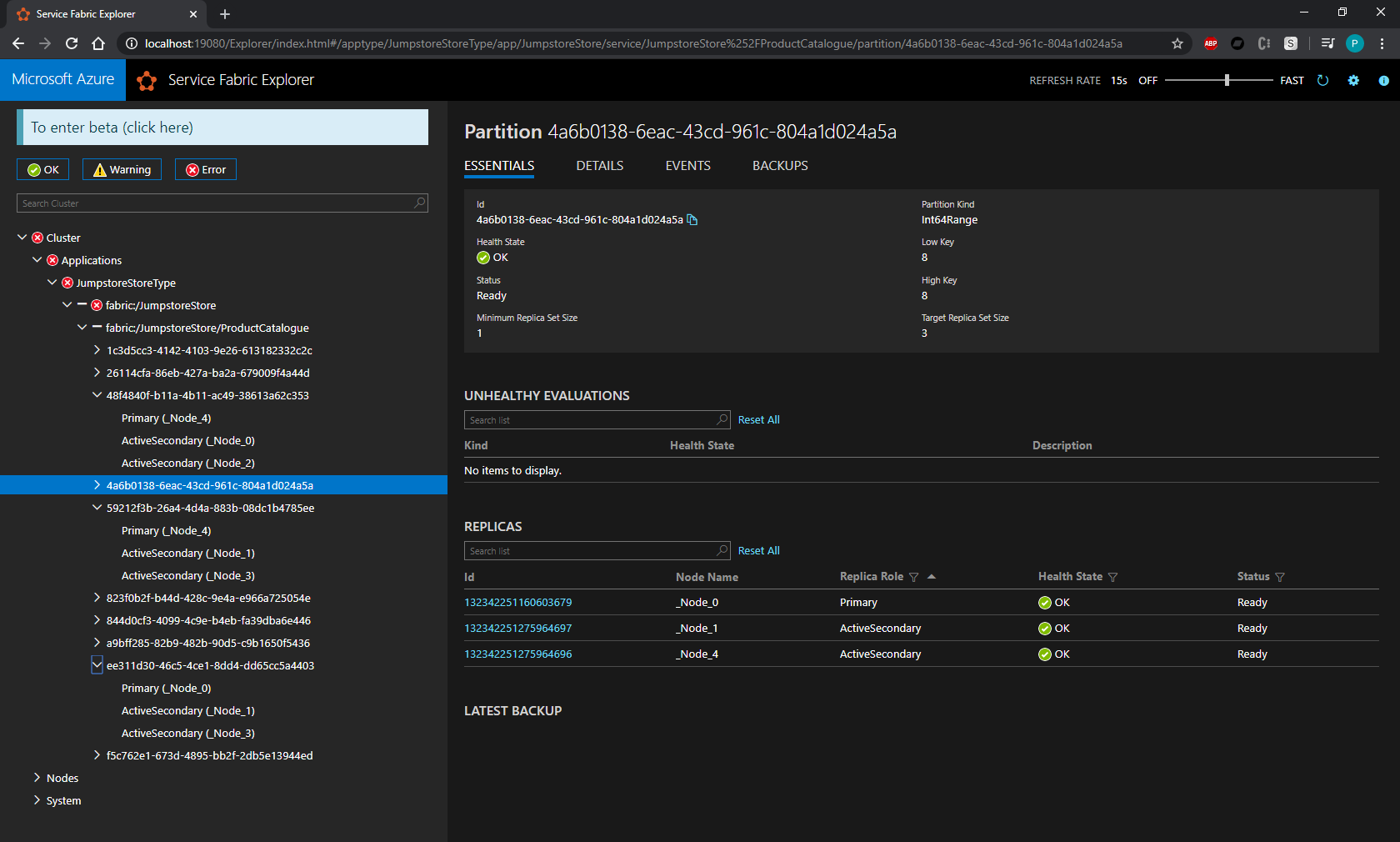
Local.5node.xml:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <Application xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Name="fabric:/JumpstoreStore" xmlns="http://schemas.microsoft.com/2011/01/fabric">  <Parameters>  <Parameter Name="PartitionCount" Value="3" />  <Parameter Name="MinReplicaSetSize" Value="3" />  <Parameter Name="TargetReplicaSetSize" Value="3" />  <Parameter Name="PartitionLowKey" Value="0" />  <Parameter Name="PartitionHighKey" Value="2" />  </Parameters>  </Application> |

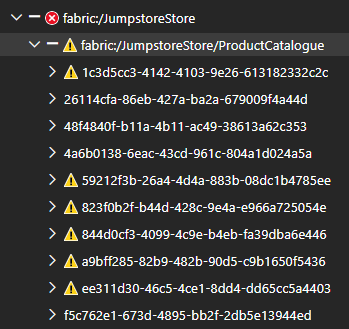


Jak widać, nasza aplikacja jest teraz zwielokrotniona na 3 partycjach.

Spróbujmy teraz ustawić 10 node’ów:

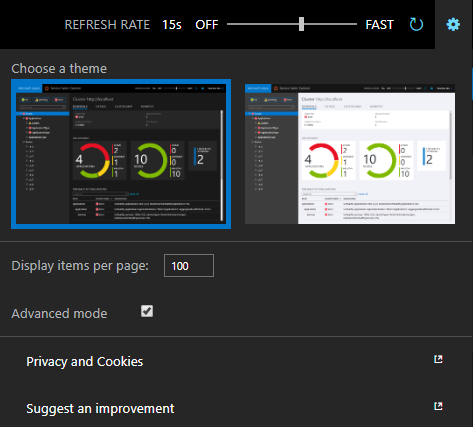


* 1. Sprawdź jak zachowa się aplikacja gdy zrestartujemy jeden z node’ów.



Jeśli zrestartujemy jeden z węzłów który przechowywał primary, to z tej partycji pierwszy secondary powinien przejąć jego taska (z innego noda, np. primary był node 0, secondary node 2, to na tej partycji teraz primary będzie node 2).

* 1. Włączmy Advanced Mode:

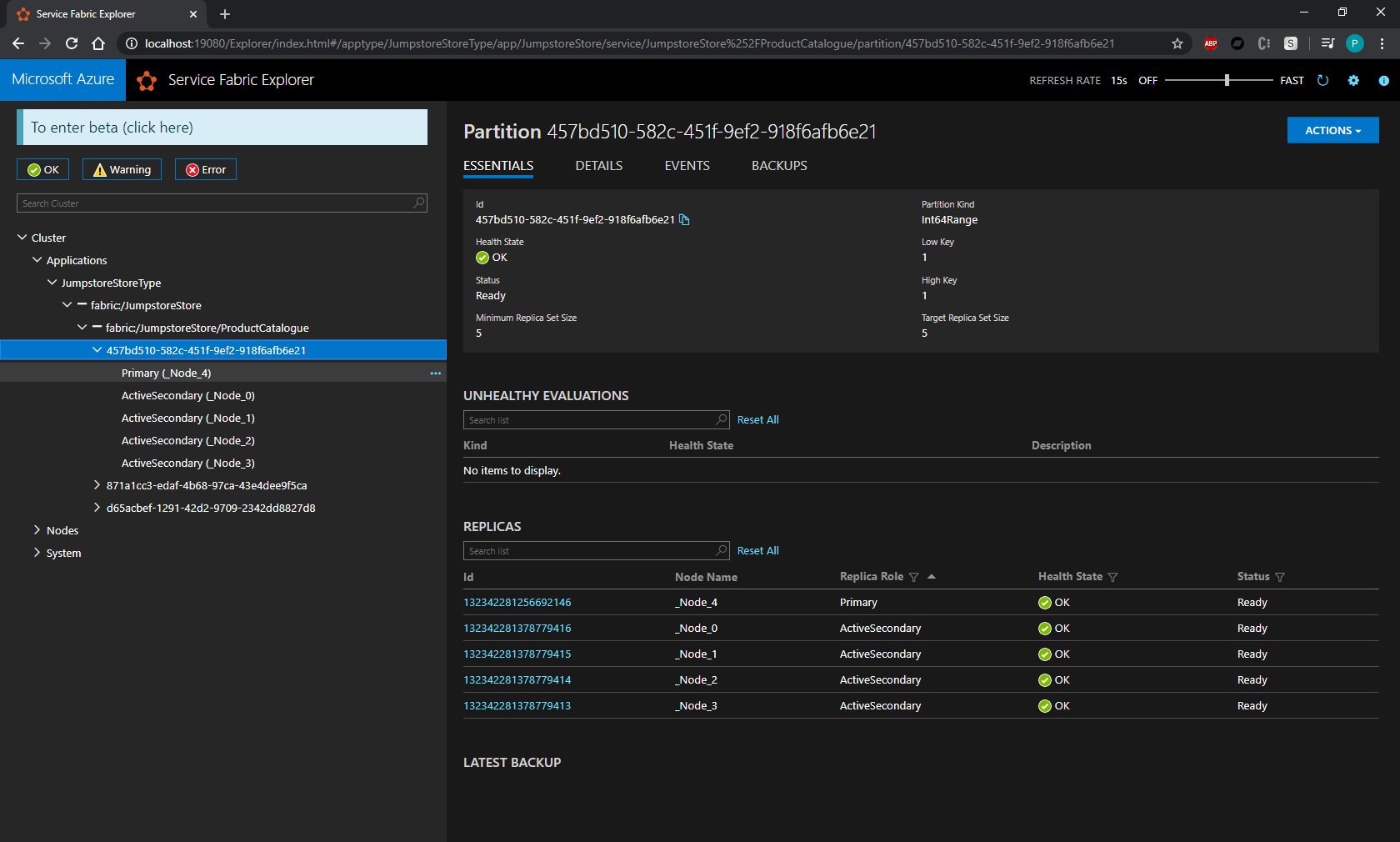


Przetestuj na jednym z node’ów Deactivate(remove Data). Co się stało?

Dane będą streamowane z innych node’ów aby je odbudować.

# Stateful Service Named Partitioning

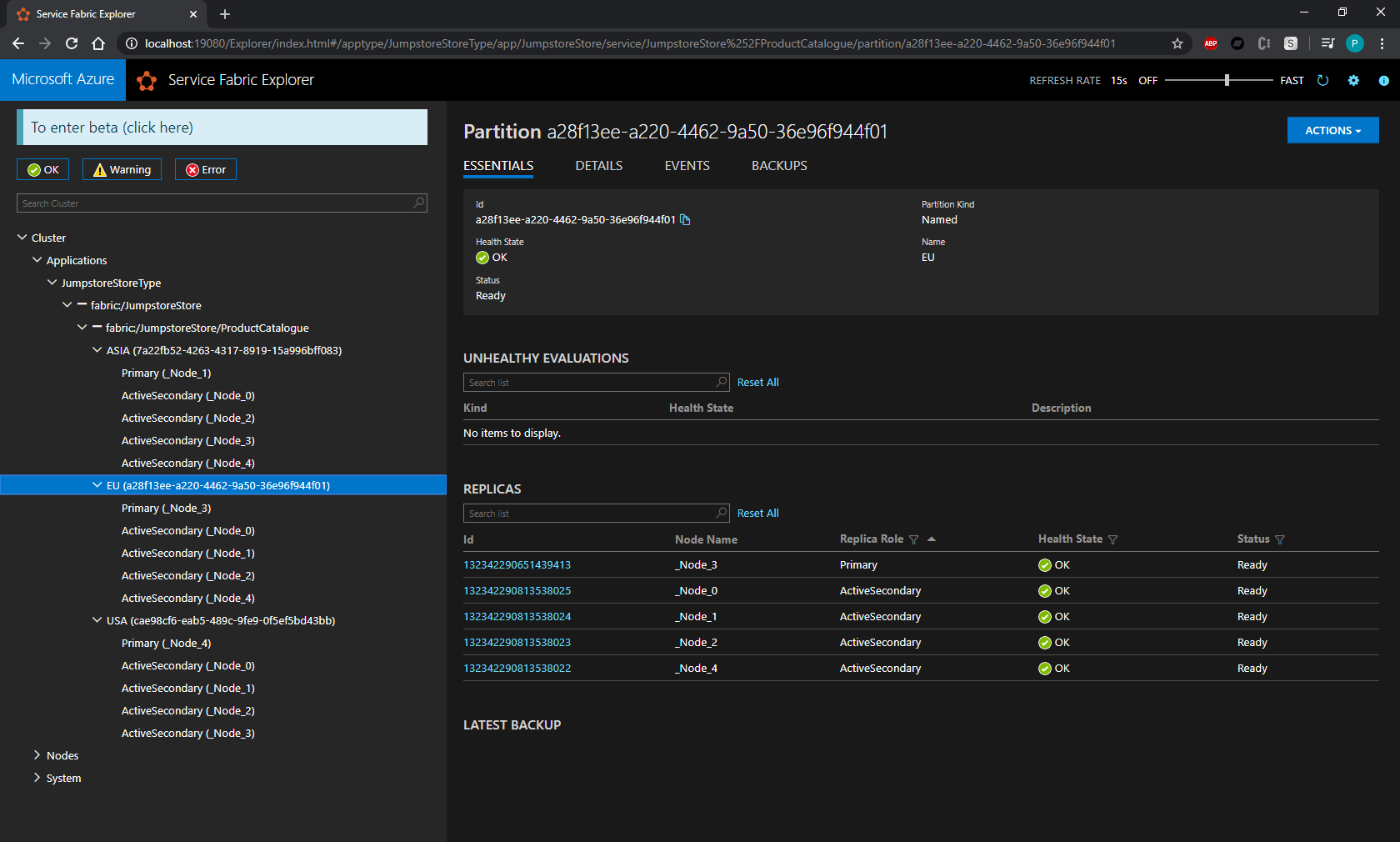
* 1. Modyfikujemy na razie naszą aplikację aby miała 3 partycje i 5 replik:



* 1. Modyfikujemy ApplicationManifest.xml i zastępujemy UniformInt64Partition:

|  |
| --- |
| <NamedPartition>  <Partition Name="USA"/>  <Partition Name="EU"/>  <Partition Name="ASIA"/>  </NamedPartition> |

Następnie deplojujemy apkę. Jakie nastąpiły zmiany?

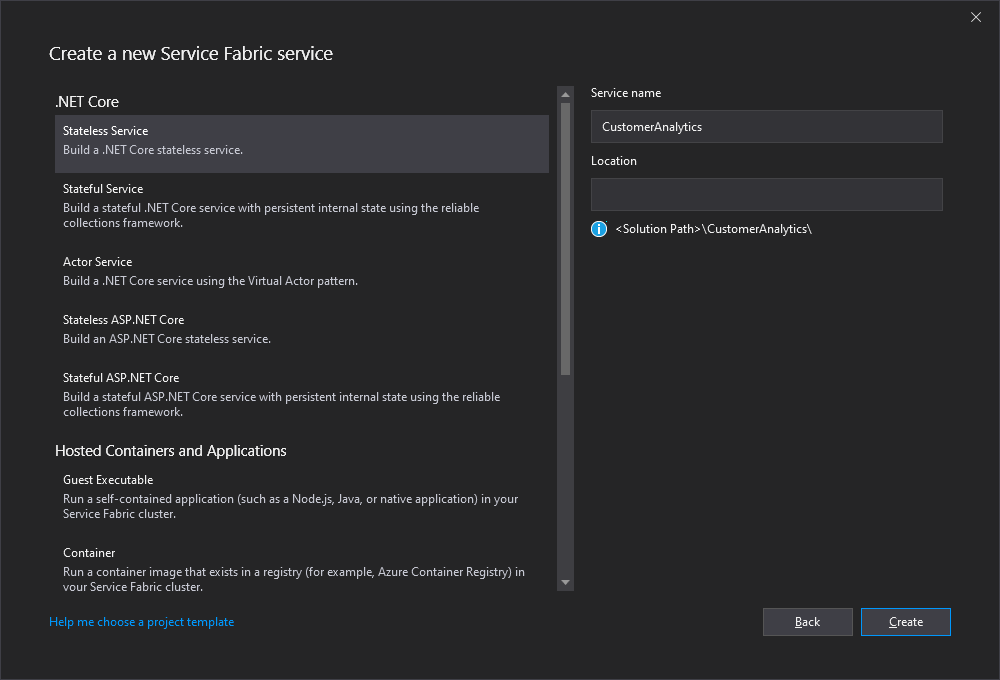


Jak widać nasze partycję teraz mają dokładnie takie nazwy jakie im nadaliśmy.

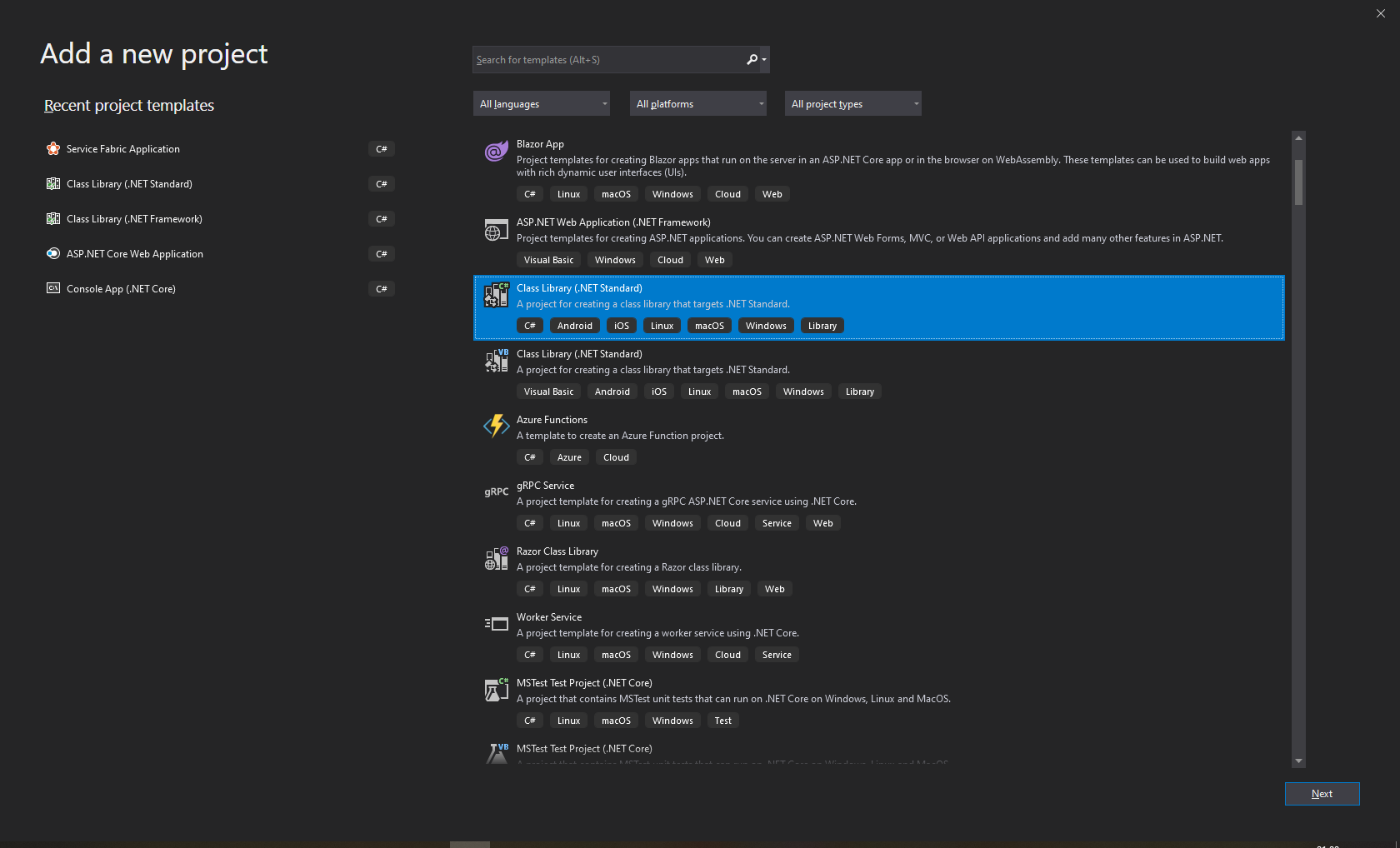
# Ustawianie komunikacji między serwisami

Service Fabric pozwala nam na komunikację pomiędzy dowolnymi serwisami w naszym klastrze. Możemy w nim mieć kilka aplikacji, a nawet serwisy z innych aplikacji które mogą komunikować się między sobą.

* 1. Dodamy do naszej aplikacji nowy Stateless Service. Add -> New Service Fabric Service:



Następnie, tworzymy nowy **Class Library (.NET Standard)** w solucji (ponieważ chcemy zdefiniować interfejsy, które są dosłownie kontraktami do komunikowania się pomiędzy serwisami):

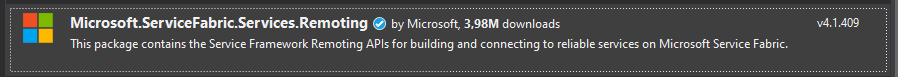


I nazywamy ją np. Communication

* 1. Zmieniamy nazwę Class1.cs na IStatelessInterface.cs i zawartość na:

|  |
| --- |
| using System;  using System.Threading.Tasks;  using Microsoft.ServiceFabric.Services.Remoting;  namespace Communication  {  public interface IStatelessInterface : IService  {  Task<string> GetServiceDetails();  }  } |

Musimy jeszcze pobrać NuGet: **microsoft.servicefabric.services.remoting:**



Następnie analogicznie tworzymy nową klasę dla IStatefulInterface.cs:

|  |
| --- |
| using Microsoft.ServiceFabric.Services.Remoting;  using System;  using System.Threading.Tasks;  namespace Communication  {  public interface IStatefulInterface: IService  {  Task<string> GetServiceDetails();  }  } |

* 1. Po zdefiniowaniu naszych interfejsów-kontraktów, należy je teraz zaimplementować w serwisach. Service Fabric zapewnia out-of-the box communication stack który potrzebujemy (bazowany na PC Proxy).
  2. Klikamy prawym na projekt CustomerAnalytics -> Add ->Reference i dodajemy referencję do Communication. Dodajemy i implementujemy interfejs IStatelessInterface w CustomerAnalytics.cs:

|  |
| --- |
| internal sealed class CustomerAnalytics : StatelessService, IStatelessInterface  {  public CustomerAnalytics(StatelessServiceContext context)  : base(context)  { }  public async Task<string> GetServiceDetails()  {  return this.Context.ServiceName.ToString();  } |

Musimy teraz zapewnić communication stack. Pobieramy ponownie tą paczkę NuGetową tym razem dla CustomerAnalytics:

Następnie Modyfikujemy metodę CreateServiceInstanceListeners():

|  |
| --- |
| protected override IEnumerable<ServiceInstanceListener> CreateServiceInstanceListeners()  {  return this.CreateServiceRemotingInstanceListeners();  } |

Teraz nasz Stateless Service może być wywołany przez inne serwisy w naszym klastrze.

* 1. Powtarzamy teraz czynności dla ProductCatalogue. Dodajemy referencję i instalujemy paczki NuGet. Dodajemy interfejs i modyfikujemy ProductCatalogue.cs:

|  |
| --- |
| internal sealed class ProductCatalogue : StatefulService, IStatefulInterface  {  public ProductCatalogue(StatefulServiceContext context)  : base(context)  { }  public async Task<string> GetServiceDetails()  {  var serviceName = this.Context.ServiceName.ToString();  var partition = this.Context.PartitionId.ToString();  return $"{serviceName} --- {partition}";  } |

Następnie Modyfikujemy metodę CreateServiceReplicaListeners():

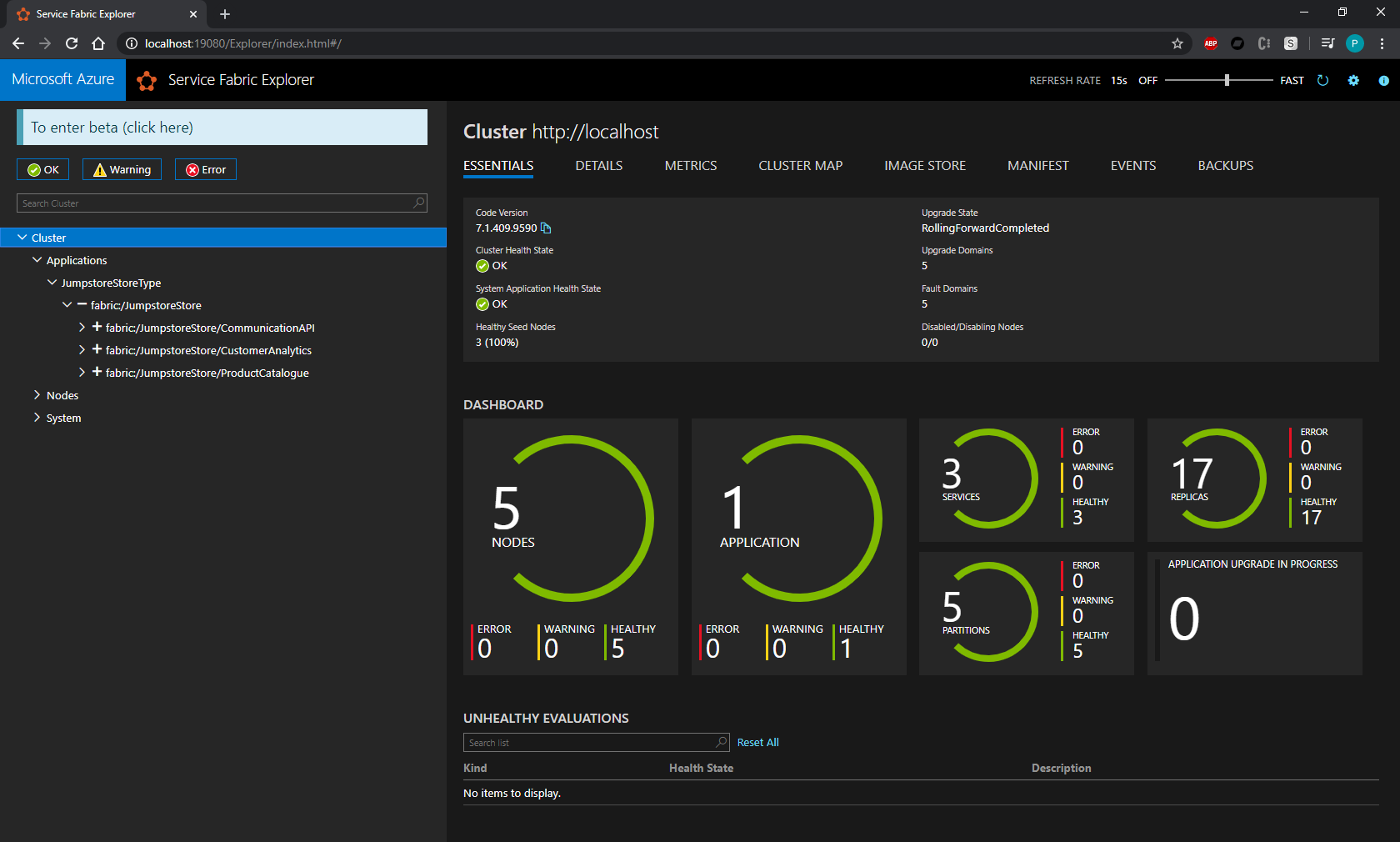
|  |
| --- |
| protected override IEnumerable<ServiceReplicaListener> CreateServiceReplicaListeners()  {  return this.CreateServiceRemotingReplicaListeners();  } |

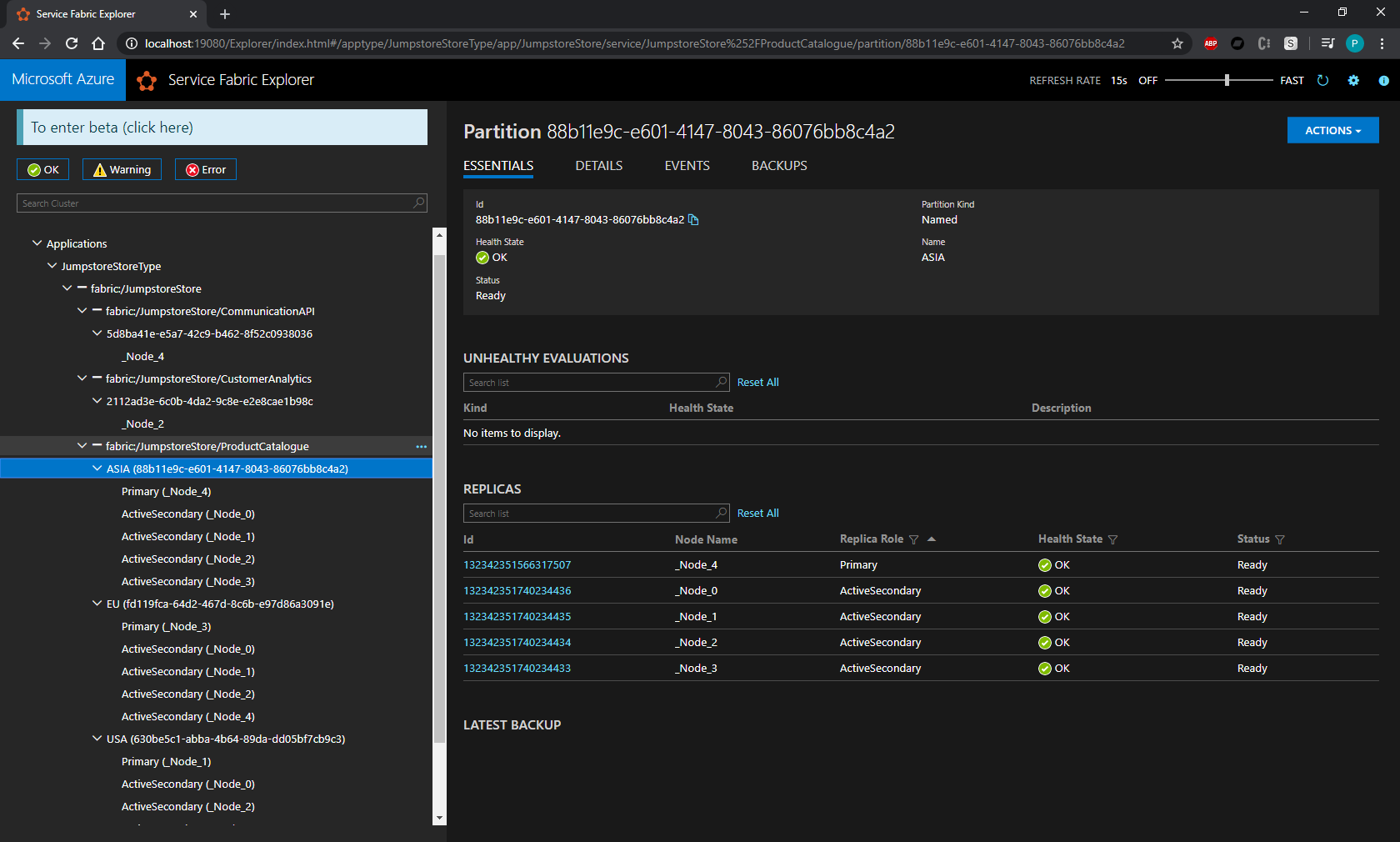
# Komunikacja między serwisami

* 1. Potrzebujemy teraz dodać API. Robimy podobnie jak wcześniej, klikamy prawym na naszą aplikację i wybieramy New Service Fabric Service -> Stateless ASP.NET Core -> API. Nazywamy go: CommunicationAPI .  
       
     Dodajemy paczkę NuGet: microsoft.servicefabric.services.remoting i dodajemy referencję do Communication.
  2. Usuwamy demo z APi: WeatherForecast.cs I zmieniamy nazwę kontrolera na: CommunicationController.cs

|  |
| --- |
| [ApiController]  [Route("[controller]")]  public class CommunicationController : ControllerBase  {  [HttpGet]  [Route("stateless")]  public async Task<string> StatelessGet()  {  var statelessProxy = ServiceProxy.Create<IStatelessInterface>(  new Uri("fabric:/JumpstoreStore/CustomerAnalytics"));  var serviceName = await statelessProxy.GetServiceDetails();  return serviceName;  }  } |

Deploy-ujemy apkę:





* 1. Sprawdź co wyświetli <http://localhost:8180/communication/stateless> (port dla API, może być inny)
  2. Dopiszmy metodę która wyświetli Stateful serwisy w naszym API kontrolerze:

|  |
| --- |
| [HttpGet]  [Route("stateful")]  public async Task<string> StatefulGet([FromQuery] string region)  {    var statefulsProxy = ServiceProxy.Create<IStatefulInterface>(  new Uri("fabric:/JumpstoreStore/ProductCatalogue"),  new Microsoft.ServiceFabric.Services.Client.ServicePartitionKey(region.ToUpperInvariant()));  var serviceName = await statefulsProxy.GetServiceDetails();  return serviceName;  } |

* 1. Sprawdź co wyświetli <http://localhost:8180/communication/stateful?region=EU> (port dla API, może być inny). Zmień ?region na =USA i =ASIA.

# Komunikacja z Partitioned Services

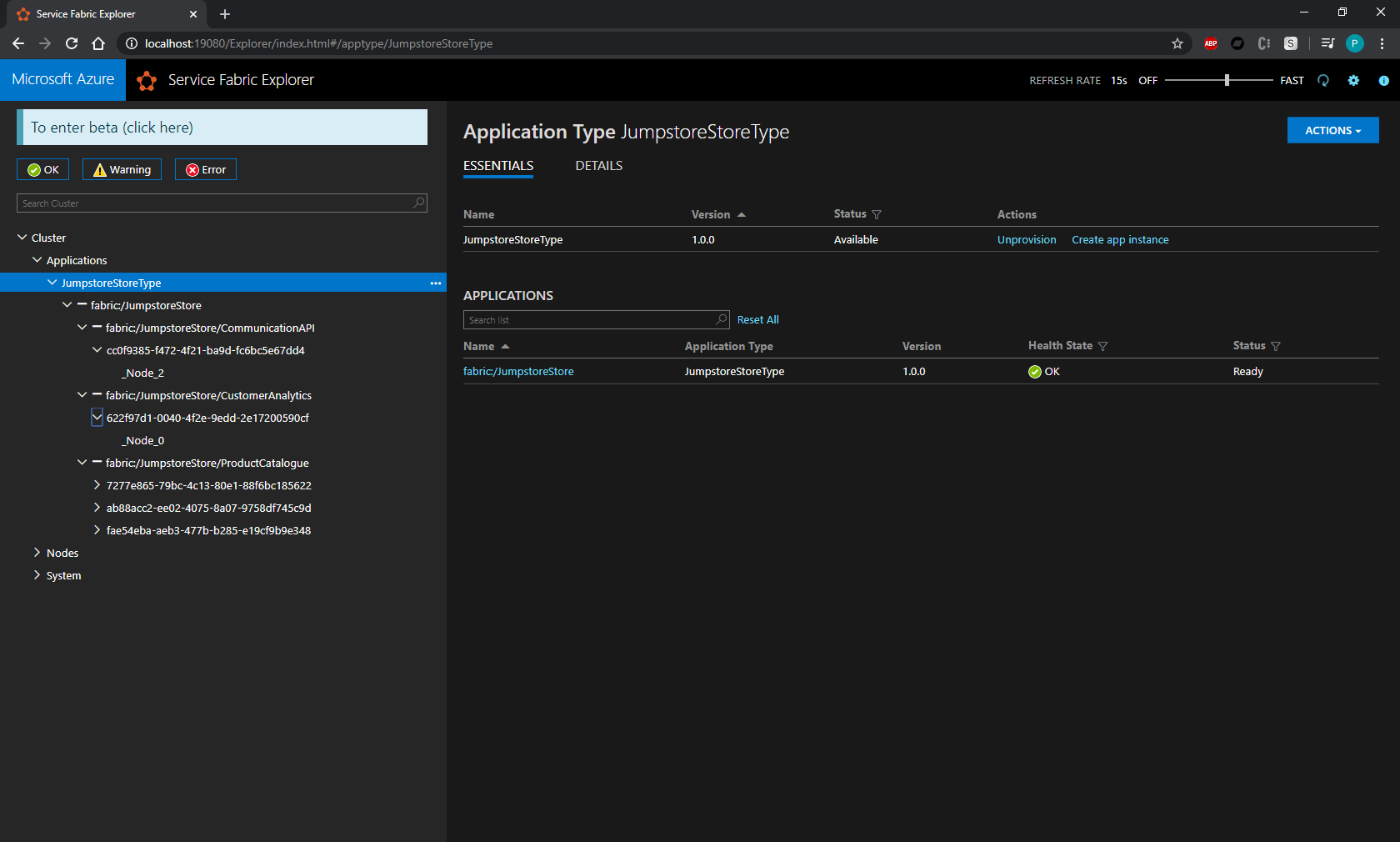
* 1. Modyfikujemy ponownie AppManifest.xml i zmieniamy sekcje <Partitions> na:

|  |
| --- |
| <UniformInt64Partition PartitionCount="3" HighKey="2" LowKey="0"/> |

CommunicationController.cs:

|  |
| --- |
| [HttpGet]  [Route("stateful")]  public async Task<string> StatefulGet([FromQuery] int productId)  {  var partitionId = productId % 3; // it will cause productId to be split betweeen partitions  var statefulProxy = ServiceProxy.Create<IStatefulInterface>(  new Uri("fabric:/JumpstoreStore/ProductCatalogue"),  new Microsoft.ServiceFabric.Services.Client.ServicePartitionKey(partitionId));  var serviceName = await statefulProxy.GetServiceDetails();  return serviceName;  } |

Robimy deploy.



Jak widać nasze partycje znów są bez nazw i mają tylko swoje ID.

Wpisujemy w przeglądarce: <http://localhost:8180/communication/stateful?productid=1>

Co wyświetli się dla productid=2 i productid =3 ?

Wyjaśnienie co uzyskaliśmy:

Rozdzieliliśmy produkty na różne partycje, dzięki czemu każda z partycji będzie miała mniejszy load. Pamiętajmy, że możemy to skalować, więc możemy np. zwiększyć ilość partycji i nadal będziemy mieli taką samą wydajność i możliwość obsłużenia większej bazy klientów z zachowaną stabilnością i wydajnością w przypadku awarii.

# Reliable Collections: Reliable Dictionary

Zrobimy teraz reliable store – w skrócie, dane będą przechowywane nie tylko w Primary replica ale także w Secondary w naszym klastrze. Dzięki temu np. gdy ulegnie awarii jeden z Primary, dane będą nadal dostępne z Secondary replica – dzięki czemu dane będą serwowane w sposób ciągły. Będziemy też przechowywali obiekty danych odpowiadających kluczowi partycji na wybranej partycji. Ustalimy też że każda partycja będzie przechowywała dane danego typu, aby rozłożyć obłożenie na naszych partycjach w miarę równomiernie.

* 1. Zacznijmy od pisania kodu. Klikamy prawym na nasz projekt Communication -> Add -> New Class. Nową klasę nazywamy Product
  2. Dodajmy teraz w niej parę właściwości, aby określić co chcemy przechowywać jako dane serializable w Reliable State.

|  |
| --- |
| namespace Communication  {  [DataContract]  public class Product  {  [DataMember]  public int Id { get; set; }  [DataMember]  public string Name { get; set; }  [DataMember]  public string Type { get; set; }  }  } |

* 1. Następnie przechodzimy do projektu: ProductCatalogue. W ProductCatalog.cs dodamy teraz reliable dictionary które pozwoli nam na przeprowadzenie transakcji za pomocą State Managera z Azure Fabric Service:

|  |
| --- |
| public async Task AddProduct(Product product)  {  var stateManager = this.StateManager;  var productDict = await stateManager.GetOrAddAsync<IReliableDictionary<int, Product>>("productdict");  using (var transaction = StateManager.CreateTransaction())  {  await productDict.AddOrUpdateAsync(transaction, product.Id, product, (key, value) => value);  await transaction.CommitAsync();  }  } |

Metoda ta, pozwoli nam na załadowanie danego Reliable Dictionary do State Managera, następnie utworzymy transakcję dodawania/update tam produktu i zakończymy transakcję.

* 1. Drugą metodą będzie wstawiona nad powyższą:

|  |
| --- |
| public async Task<Product> GetProductById(int id)  {  var stateManager = this.StateManager;  var productDict = await stateManager.GetOrAddAsync<IReliableDictionary<int, Product>>("productdict");    using (var transaction = stateManager.CreateTransaction())  {  var product = await productDict.TryGetValueAsync(transaction, id);  return product.Value;  }  throw new Exception();  } |

Metoda podobnie jak powyższa, ładuje dany Reliable Dictionary do State Managera, następnie pobieramy wartość produktu z RD, i ją zwracamy.

Notka: Nie jest to może najlepszy kod (na proda bym nie wrzucił ☺) – ale pokazuje jak to działa, więc zainteresowani mogą sobie poszukać więcej informacji i przykładów. Na obecną chwilę będzie wystarczający.

* 1. Następnie musimy poprawić nasz IStatefulInterface.cs, aby nasze nowe metody były dostępne dla API:

|  |
| --- |
| Task<Product> GetProductById(int id);  Task AddProduct(Product product); |

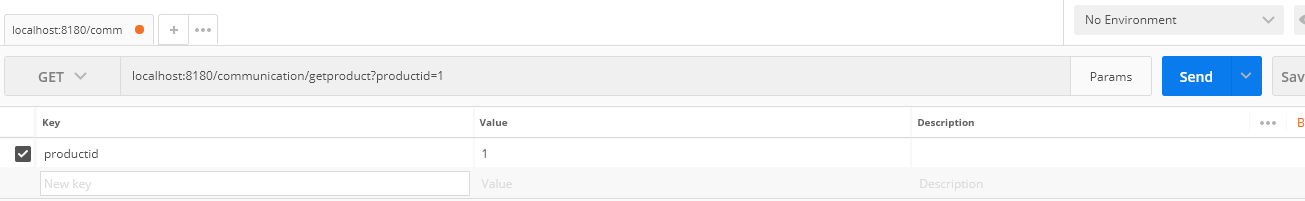
* 1. Przejdźmy teraz do naszego kontrolera API – CommunicationController.cs. Dodajmy tam metody które pozwolą nam na proxy do naszego katalogu produktów i pobierze produkty z różnych partycji i reliable State Managera na tych partycjach:

|  |
| --- |
| [HttpPost]  [Route("addproduct")]  public async Task AddProduct([FromQuery] Product product)  {  var partitionId = product.Id % 3; var statefulProxy = ServiceProxy.Create<IStatefulInterface>(  new Uri("fabric:/JumpstoreStore/ProductCatalogue"),  new Microsoft.ServiceFabric.Services.Client.ServicePartitionKey(partitionId));  await statefulProxy.AddProduct(product);  }  [HttpGet]  [Route("getproduct")]  public async Task<Product> GetProduct([FromQuery] int productId)  {  var partitionId = productId % 3;  var statefulProxy = ServiceProxy.Create<IStatefulInterface>(  new Uri("fabric:/JumpstoreStore/ProductCatalogue"),  new Microsoft.ServiceFabric.Services.Client.ServicePartitionKey(partitionId));  var product = await statefulProxy.GetProductById(partitionId  return product;  } |

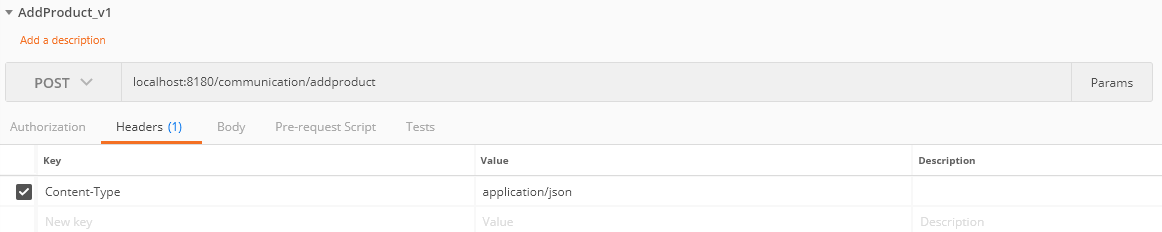
Utworzyliśmy metodę które będzie dodawała produkt i metodę która pobiera produkt poprzez jego ID. Dodatkowo produkty rozdzielamy na 3 partycje.

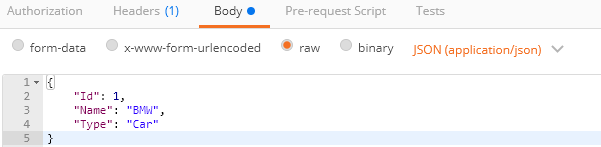
* 1. Robimy deploy, uruchamiamy Postmana.

Tworzymy nowe zapytanie GET:



I nowe zapytanie POST:





Wysyłamy najpierw zapytanie POST. Pamiętaj aby ustawić poprawny port jaki ma w configu API! GET powinien zwrócić dodany wpis.

# Reliable Collections: Reliable Queues

Teraz zapoznamy się z Reliable Queues. Kolejka operuje jako FIFO (First in first out) – czyli kiedy wyciągamy obiekt z kolejki, to będzie to obiekt który był najdłużej w kolejce. Tak samo, jak będziemy dodawali obiekty, np. nasze produkty – będą one wtedy usuwane jako ostatnie.

* 1. Dodajmy trochę kodu do ProductCatalogue.cs:

|  |
| --- |
| public async Task<Product> GetFromQueue()  {  var stateManager = this.StateManager;  var productQueue = await stateManager.GetOrAddAsync<IReliableQueue<Product>>("productqueue");  using (var transaction = stateManager.CreateTransaction())  {  var product = await productQueue.TryDequeueAsync(transaction);  await transaction.CommitAsync();  return product.Value;  }  throw new ArgumentException();  }  public async Task AddToQueue(Product product)  {  var stateManager = this.StateManager;  var productQueue = await stateManager.GetOrAddAsync<IReliableQueue<Product>>("productqueue");  using (var transaction = stateManager.CreateTransaction())  {  await productQueue.EnqueueAsync(transaction, product);  await transaction.CommitAsync();  }  } |

Tworzymy metody które będą pobierały i dodawały produkty z/do kolejki. Jak poprzednio, odbywa się to poprzez transakcję z tą różnicą że jest to teraz kolejka a nie dictionary.

* 1. Oczywiście dodajemy sygnatury w IStatefulInterface.cs aby można było je wywoływać:

|  |
| --- |
| Task<Product> GetFromQueue();  Task AddToQueue(Product product); |

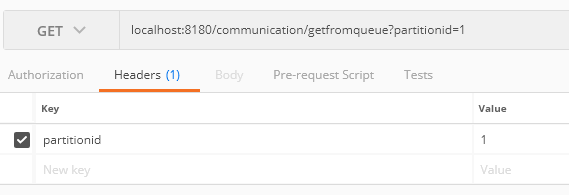
* 1. Następnie dodajemy w kontrolerze API CommunicationController.cs:

|  |
| --- |
| [HttpPost] // we want to post products  [Route("addtoqueue")]  public async Task AddToQueue([FromQuery] int partitionId, [FromBody] Product product)  {  var statefulProxy = ServiceProxy.Create<IStatefulInterface>(  new Uri("fabric:/JumpstoreStore/ProductCatalogue"),  new Microsoft.ServiceFabric.Services.Client.ServicePartitionKey(partitionId));  await statefulProxy.AddToQueue(product);  }  [HttpGet] // we want to get products  [Route("getfromqueue")]  public async Task<Product> GetFromQueue([FromQuery] int partitionId)  {  var statefulProxy = ServiceProxy.Create<IStatefulInterface>(  new Uri("fabric:/JumpstoreStore/ProductCatalogue"),  new Microsoft.ServiceFabric.Services.Client.ServicePartitionKey(partitionId));  var product = await statefulProxy.GetFromQueue();  return product; // return that product  } |

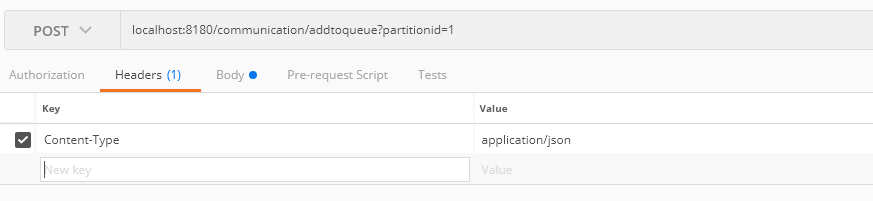
Metody te pozwalają na dodawanie i pobieranie obiektów do/z kolejki.

* 1. Testujemy w Postmanie:

GET:



POST:





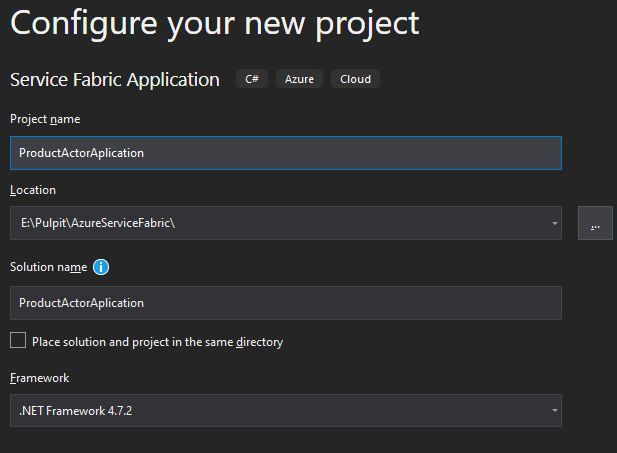
Sprawdź jak zachowuje się kolejka, spróbuj porozsyłać dane na różne partycje i odpytuj GETem jakie dane się wyświetlą.

Jak widać kolejka będzie zwracała najstarszy obiekt i dane są widoczne tylko na poszczególnych partycjach.

# Reliable Actors: Actor State & Communication

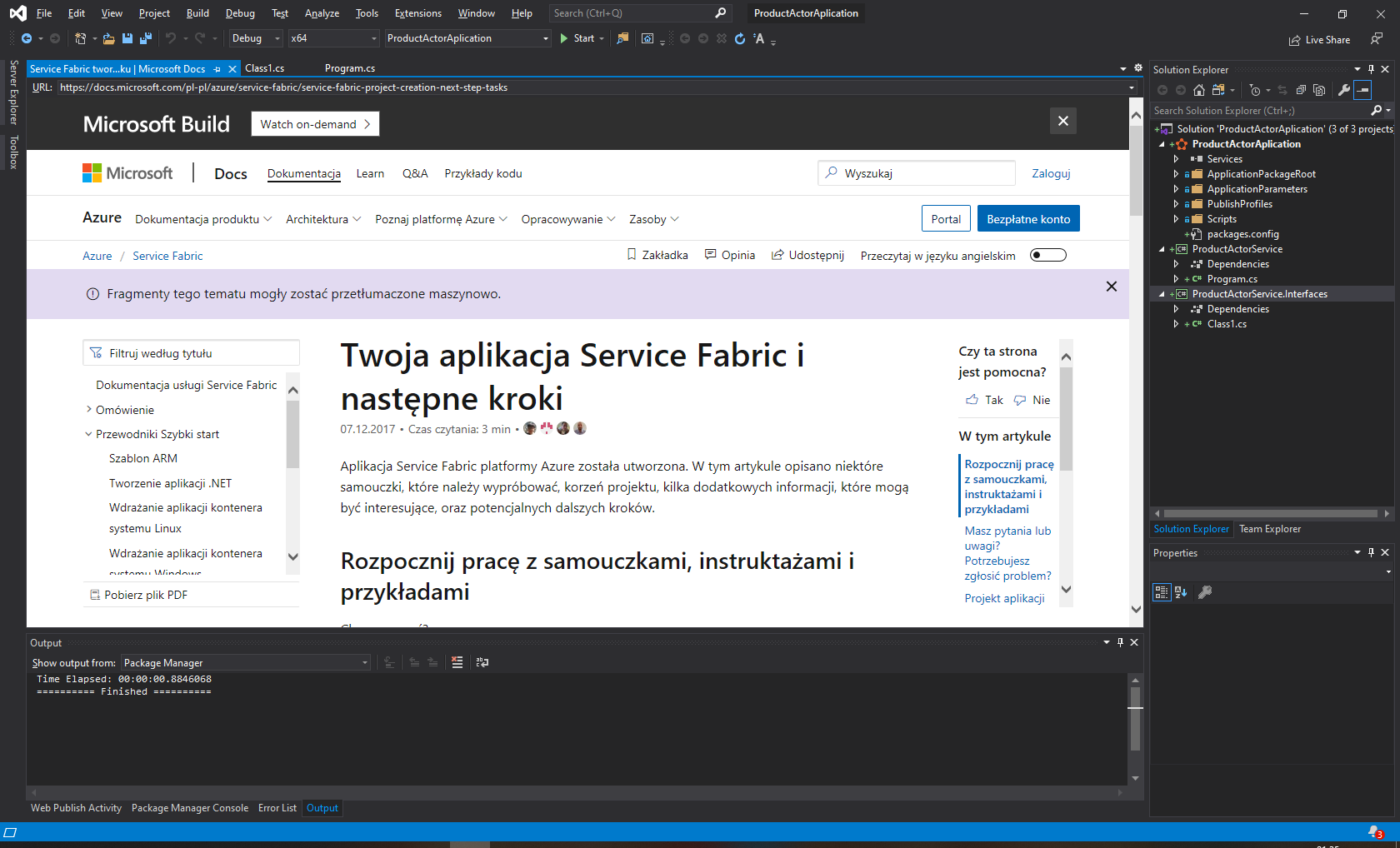
Aktor – to izolowana, niezależna jednostka komputera (i jego stanu), która ma jeden wątek wykonywania. Jest to jeden z wzorców projektowych. Są oni użyteczni, gdy chcemy pracować z jednowątkowymi obiektami, które nie wymagają zbytnio interakcji z zewnętrznymi komponentami. Nie używamy ich np. gdy odpytujemy query stanu z dużego zestawu aktorów – w Service Fabric każdy reliable service aktor którego napiszemy jest aktualnie partycjonowanym reliable stateful serwisem. Nie chce zbytnio się rozpisywać, zobaczymy w kodzie. ☺

* 1. Tworzymy nową solucję: ProductActorAplication, wybieramy New Service Fabric Service.



Następnie wybieramy:





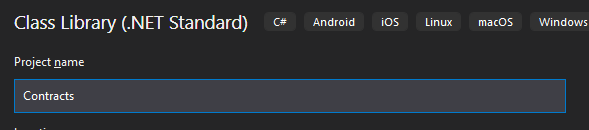
Powinien pojawić się nowy projekt.

* 1. Nasza aplikacja-Aktor jest bardzo podobna do wcześniejszej. Mamy też pliki konfiguracyjne manifest .xml, ale mamy też od razu przygotowany serwis ProjectActorService w .NET Core 3 i interfejs out-of-the-box po utworzeniu.

Przyjrzyjmy się co zawiera ProductActorService.cs:

* Dziedziczy po klasie Actor – to odróżnia go jako Actor Service
* Implementuje interfejs IActorService – zmienimy go dla naszych potrzeb w tym tutorialu

Dodajemy nowy projekt w solucji -> Class Library (.NET Standard)



Tutaj będziemy storowali wszystkie kontrakty dla naszych produktów. Zmieniamy nazwę klasy na Product.cs

Dodajemy trochę serializowalnego kodu:

|  |
| --- |
| namespace Contracts  {  [DataContract]  public class Product  {  [DataMember]  public int Id {get; set;}  [DataMember]  public string Name { get; set; }  [DataMember]  public string Type { get; set; }  }  } |

* 1. Dodajemy refenencję Contracts w ProductActorService.Interfaces. Edytujemy IProductActorService.cs:

|  |
| --- |
| public interface IProductActorService : IActor  {  Task<Product> GetProductAsync(CancellationToken cancellationToken);  Task AddProductAsync(Product product, CancellationToken cancellationToken);  } |

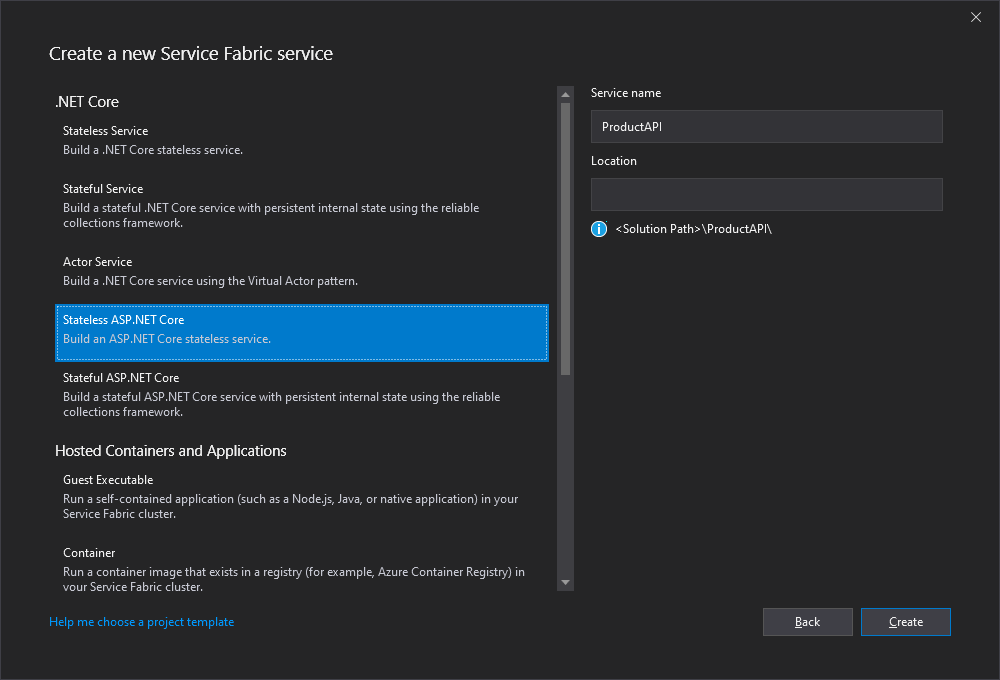
Dlaczego nie przekazujemy żadnego ID? Ponieważ kiedy podłączamy się do aktora, aktor ma swoje własne ID – będziemy używali jego.

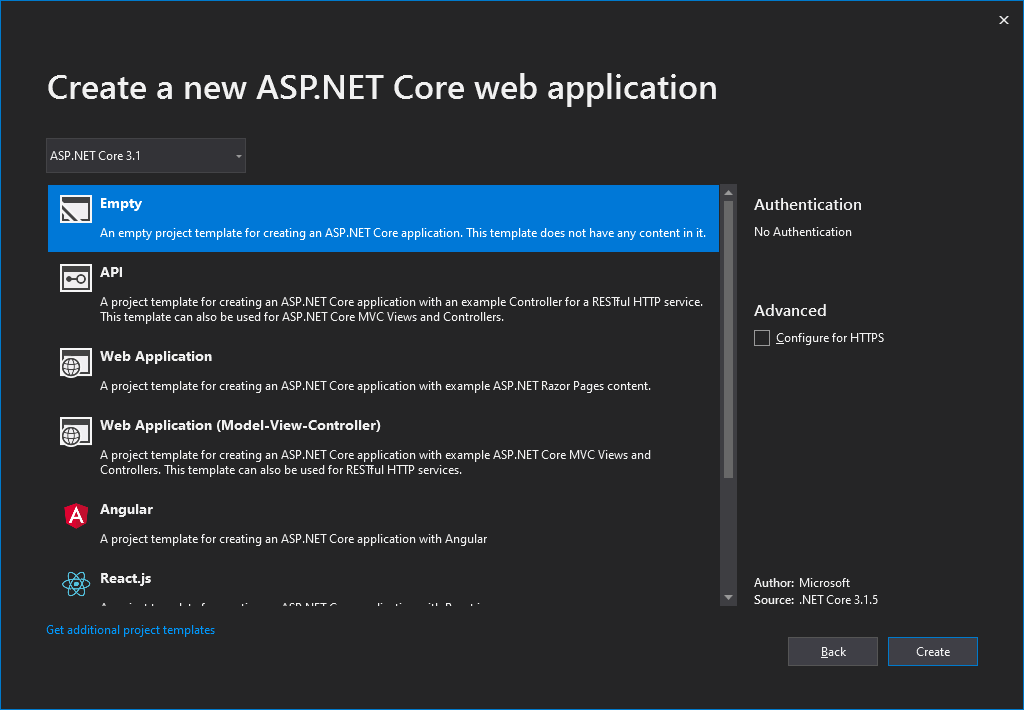
* 1. Wracamy do: ProductActorService.cs i podmieniamy:

|  |
| --- |
| namespace ProductActorService  {  [StatePersistence(StatePersistence.Persisted)]  internal class ProductActorService : Actor, IProductActorService  {  private string ProductStateName = "ProductState";  public ProductActorService(ActorService actorService, ActorId actorId)  : base(actorService, actorId)  {  }  public async Task AddProductAsync(Product product, CancellationToken cancellationToken)  {  await this.StateManager.AddOrUpdateStateAsync(ProductStateName, product, updateValueFactory: (key, value) => product, cancellationToken);  await this.StateManager.SaveStateAsync(cancellationToken);  }  public async Task<Product> GetProductAsync(CancellationToken cancellationToken)  {  var product = await this.StateManager.GetStateAsync<Product>(ProductStateName, cancellationToken);  return product;  }  protected override Task OnActivateAsync()  {  ActorEventSource.Current.ActorMessage(actor: this, message: "Actor activated.");  return this.StateManager.TryAddStateAsync("count", value: 0);  }  }  } |

Metody te pozwolą nam na dodawanie i pobieranie produktów asynchronicznie.

* 1. Następnie dodajemy do naszej apki-Aktora API, klikamy prawym ProductActorApplication -> New Service Fabric Service -> Stateless ASP.NET Core:





Następnie dodajemy folder: Controllers, a w nim nową klasę: ProductControler.cs:

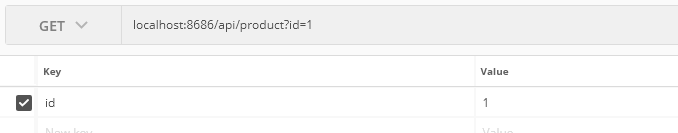
|  |
| --- |
| namespace ProductAPI.Controllers  {  [ApiController]  [Route("api/[controller]")]  public class ProductController : Controller  {  [HttpGet]  public async Task<Product> GetProductById([FromQuery] int id)  {  var actorId = new ActorId(id);  var proxy = ActorProxy.Create<IProductActorService>(actorId, new Uri("fabric:/ProductActorAplication/ProductActorServiceActorService "));  var product = await proxy.GetProductAsync(new CancellationToken());  return product;  }  [HttpPost]  public async Task AddProduct([FromBody] Product product)  {  var actorId = new ActorId(product.Id);  var proxy = ActorProxy.Create<IProductActorService>(actorId, new Uri("fabric:/ProductActorAplication/ProductActorServiceActorService "));  await proxy.AddProductAsync(product, new CancellationToken());  }  }  } |

* 1. Przechodzimy do Startup.cs:

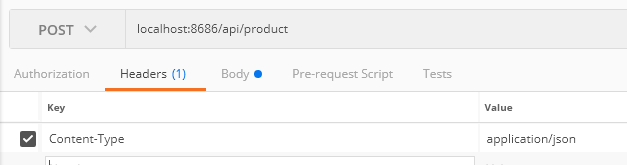
|  |
| --- |
| namespace ProductAPI  {  public class Startup  {  // This method gets called by the runtime. Use this method to add services to the container.  // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?LinkID=398940  public void ConfigureServices(IServiceCollection services)  {  services.AddControllers();  }  // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  {  if (env.IsDevelopment())  {  app.UseDeveloperExceptionPage();  }  app.UseRouting();  app.UseEndpoints(endpoints => { endpoints.MapControllers();  });  }  }  } |

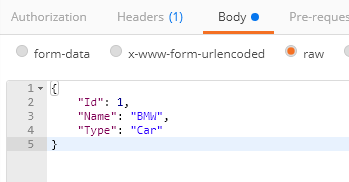
* 1. Robimy testy w postmanie czy działa:

GET:



POST:





Przetestuj warianty. Co się stanie dla id=2 ?

Dlaczego pojawił się wyjątek?

Jeśli nie dodaliśmy produktu z id 2, to nie mieliśmy jeszcze utworzonego aktora czekającego na ID 2. Więc aktorzy tak naprawdę są tworzeni dopiero kiedy są wywoływani pierwszy raz.

Aktorzy mogą być także wywoływani z stateful i stateless serwisów a także przez innych aktorów. Możesz poczytać więcej o aktorach jeśli jesteś zainteresowany tematem. ☺

# Reliable Actors: Actor Life Cycle

Kiedy aktor jest deaktywowany, Garbage Collector czyści obiekty aktora ale –nie- usuwa stanu aktora w State Managerze. Więc jeśli jest ponownie aktywowany, to jest także tworzony nowy obiekt aktora. Dlaczego możemy chcieć usunąć aktora? Dlatego, że jego dane są persisted na dysku – z czasem nasza liczba aktorów w systemie może rosnąć i rosnąć… razem z gromadzonymi danymi aktorów – dlatego możemy chcieć pozbyć się starych aktorów i ich stanów (a jak wspomniałem Garbage Collector się tym nie zajmie).

* 1. Dodajmy 3 nowe metody w ProductActorService.cs:

|  |
| --- |
| protected override Task OnPostActorMethodAsync(ActorMethodContext actorMethodContext)  {  ActorEventSource.Current.ActorMessage(actor: this, message: $"{actorMethodContext.MethodName} has finished.");  return base.OnPostActorMethodAsync(actorMethodContext);  }  protected override Task OnPreActorMethodAsync(ActorMethodContext actorMethodContext)  {  ActorEventSource.Current.ActorMessage(actor: this, message: $"{actorMethodContext.MethodName} will start soon.");  return base.OnPreActorMethodAsync(actorMethodContext);  }  protected override Task OnDeactivateAsync()  {  ActorEventSource.Current.ActorMessage(actor: this, message: "Actor deactivated.");  return base.OnDeactivateAsync();  } |

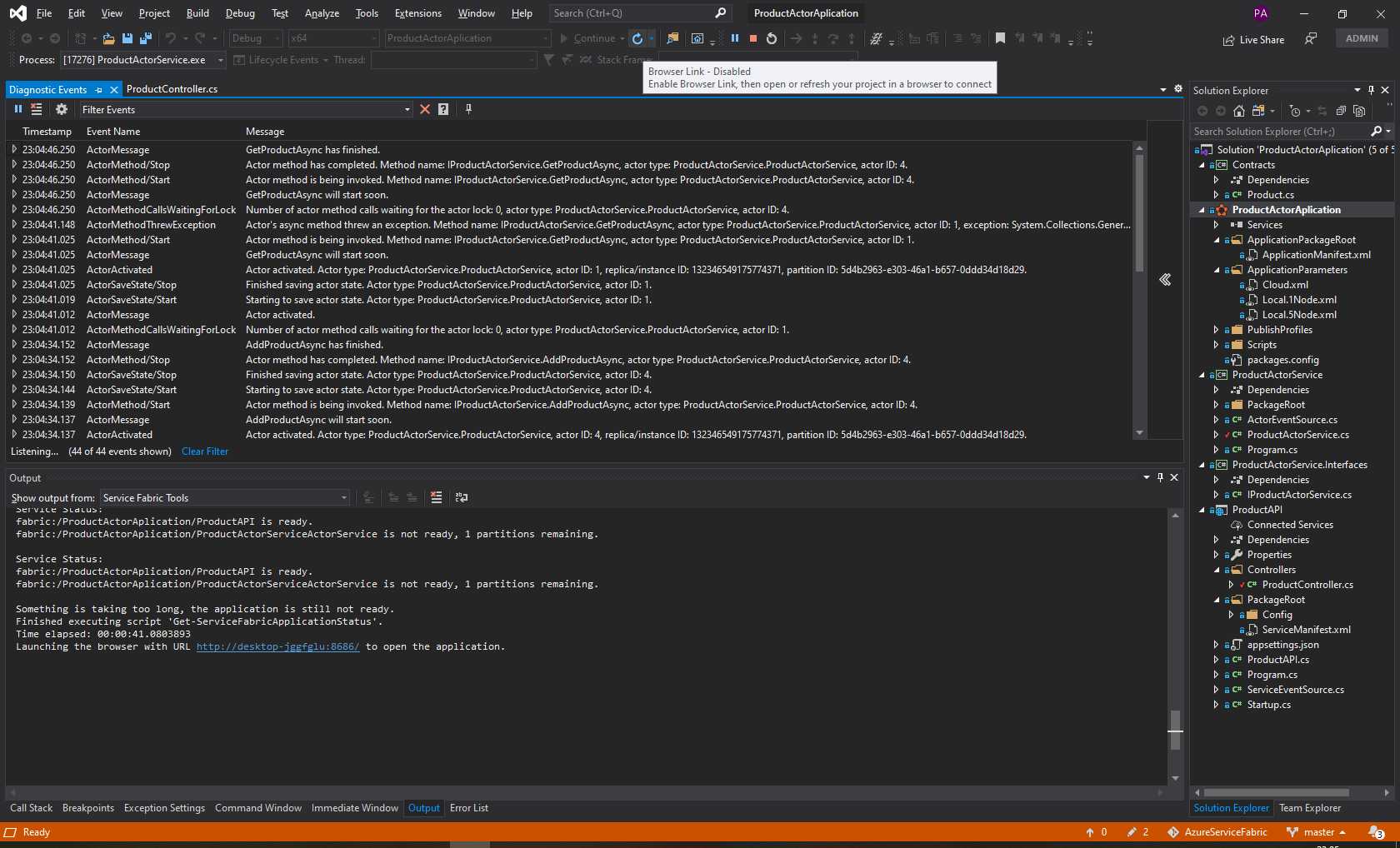
Dodaliśmy metody odpowiadające za informowanie że: aktor skończył, aktor zaraz zacznie swoje działanie i deaktywację aktora.

* 1. Następnie przechodzimy do API w ProductController.cs i dopisujemy:

|  |
| --- |
| [HttpDelete]  public async Task DeleteActorById([FromQuery] int id)  {  var actorId = new ActorId(id);  var actorServiceProxy = ActorServiceProxy.Create(new Uri("fabric:/ProductActorAplication/ProductActorServiceActorService"), actorId);  await actorServiceProxy.DeleteActorAsync(actorId, new CancellationToken());  } |

Metoda ta pozwoli nam na usuwanie aktora po id.

* 1. Robimy deploy aplikacji. Odpalamy Postmana, i sprawdzamy naszego POSTa i GETa z poprzedniego ćwiczenia: z drobną zmianą, w POST wpiszmy np. id=4. Sprawdzamy Diagnostic Events w VS:



Prześledź jak zostały wykonane nasze metody.

Co wyświetliło się w logach? Co jeśli np. GETem chcieliśmy wyświetlić produkt z id którego nie utworzyliśmy?

# Reliable Actors: Actor Timers & Reminders

Kolejną fajną zaletą używania aktorów są Timery i Remindery. Pozwalają one nam na wykonanie pracy aktora po pewnym interwale albo pełnym cyklu pracy po pewnym interwale.

* 1. Dodajmy trochę kodu w ProductActorService.cs pod „ProductState”:

|  |
| --- |
| private IActorTimer \_actorTimer; |

I w OnActivateAsync podmieniamy:

|  |
| --- |
| \_actorTimer = RegisterTimer(DoWork, null, TimeSpan.FromSeconds(10), TimeSpan.FromSeconds(15));  ActorEventSource.Current.ActorMessage(actor: this, message: "Actor activated.");  return this.StateManager.TryAddStateAsync("count", value: 0); |

Gdy chcemy zacząć pracę możemy chcieć też wykonać jakieś zadania w tle na aktorze gdy zostanie aktywowany, właśnie tam też chcemy rejestrować nasz Timer. Robimy to w OnActivateAsync().

* 1. Pod metodą dopisujemy kolejną która będzie callbackiem dla naszego Timera:

|  |
| --- |
| private Task DoWork(object work) {  ActorEventSource.Current.ActorMessage(actor: this, message: $"Actor is doing work");  return Task.CompletedTask;  } |

Zaletą Timerów jest to że respektują turn-based concurency – które jest nieodłączne od aktorów, co oznacza że aktorzy, metody, timery i remindery etc. nie będą przetworzone zanim ten callback się zakończy.

Np. jeślibyśmy dopisali do callback’a DoWork():

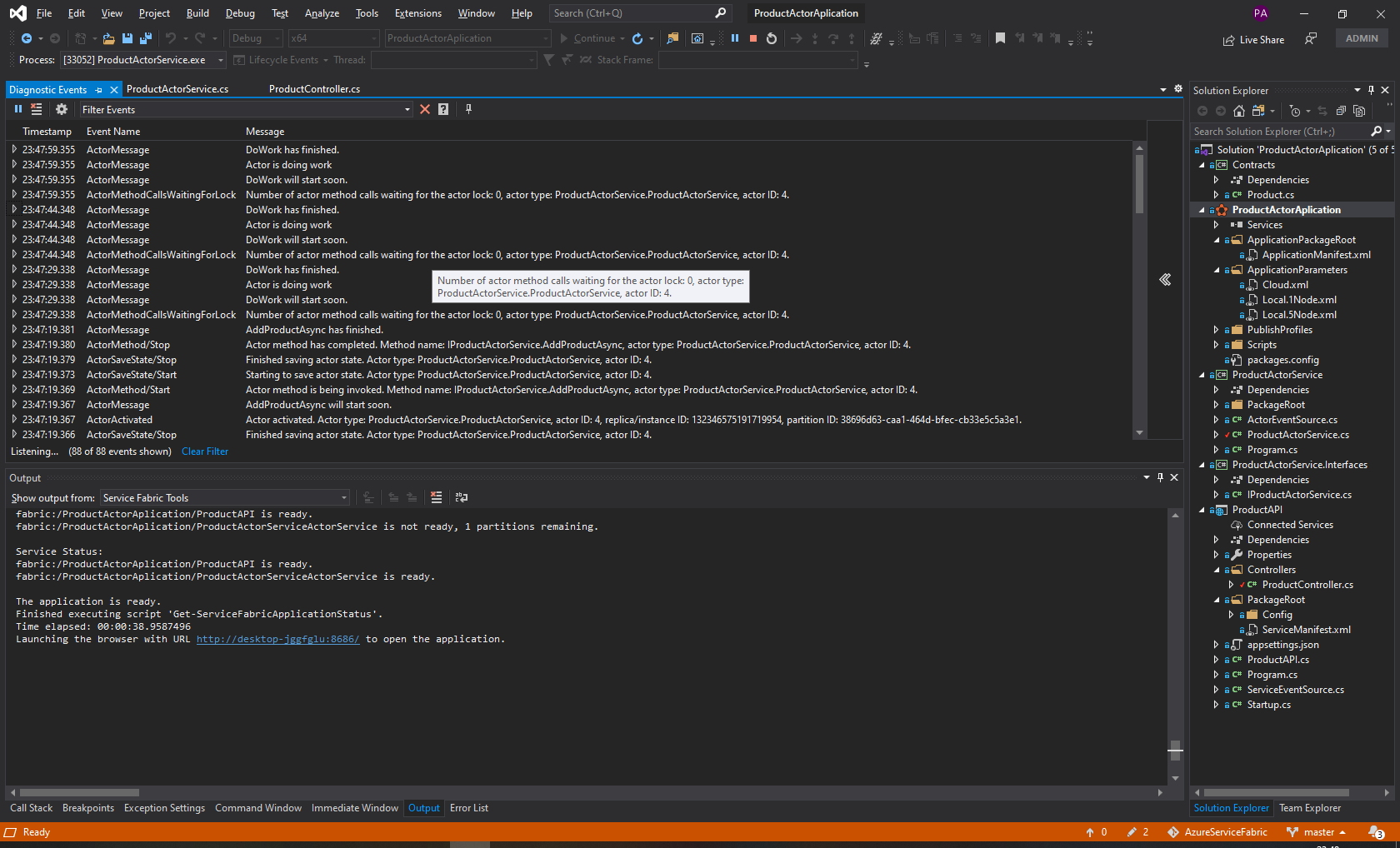
|  |
| --- |
| Task.Delay(TimeSpan.FromSeconds(100); |

To zablokowalibyśmy działanie aktora na 100 sekund – bo może on robić tylko jedną czynność w danym czasie – nie przetwarzałby requestów od klientów podczas tego czasu. Dlatego pamiętaj o tym podczas używania takich rzeczy i staraj się aby taski były wykonywane szybko!

* 1. Kolejną rzeczą jaką chcemy zrobić, jest wyczyszczenie Timer’a aktora kiedy jest dezaktywowany. Więc chcemy sprawdzić i wyrejestrować Timer. W OnDeactivateAsync() dopisujemy warunek:

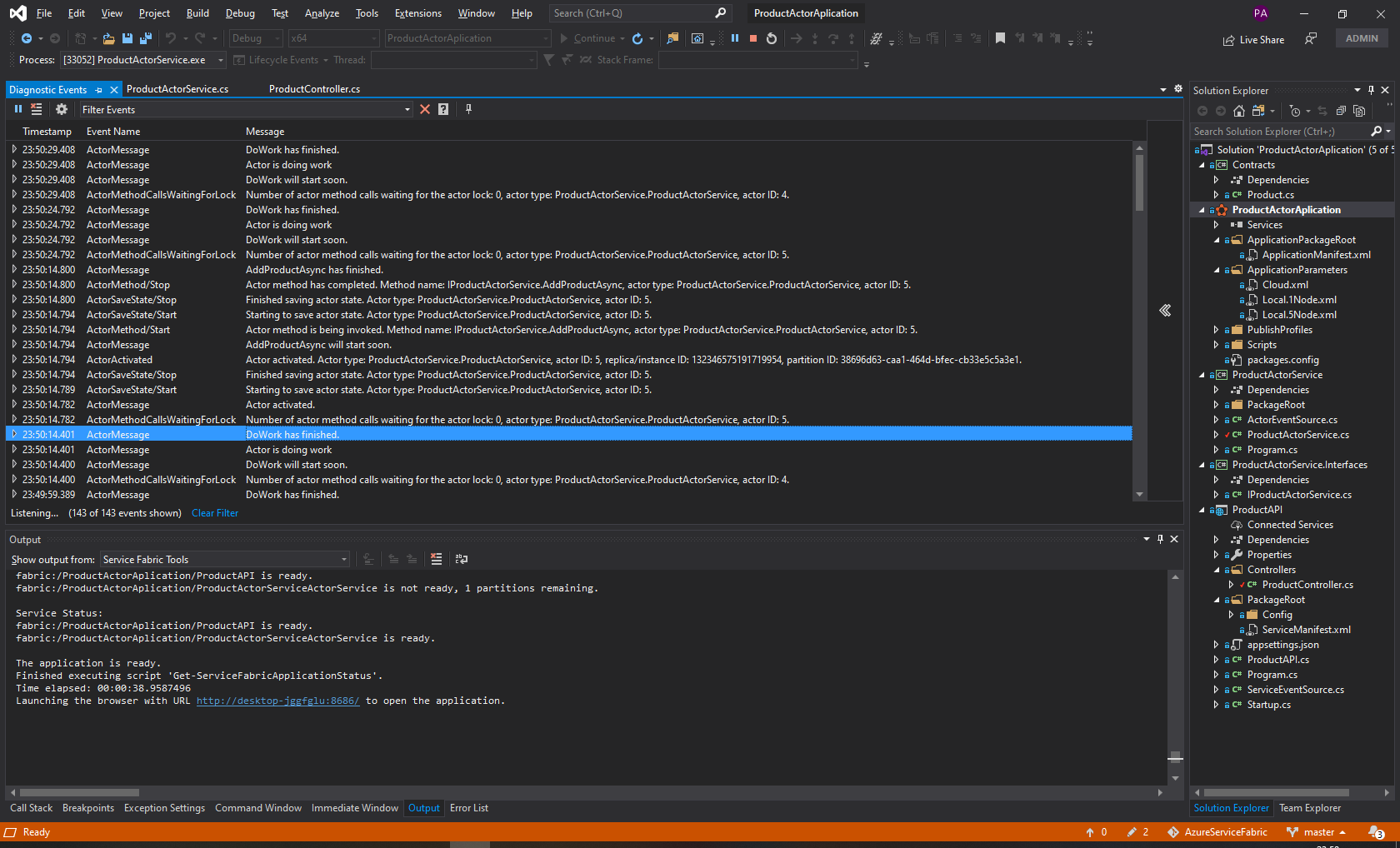
|  |
| --- |
| if (\_actorTimer != null)  {  UnregisterTimer(\_actorTimer);  } |

* 1. Robimy deploy naszej apki. Odpalamy ponownie Postmana – używamy naszego POSTa. Sprawdzamy Diagnostic Events:



Jak widać, nasz aktor teraz odpalił się i pierwszy raz wykonał swoje taski po 10 sekundach, następnie co 15 sekund.

W naszym POST np. teraz odpalmy z id=5 nie stopując obecnego aktora:



Jak widać teraz są odpalone 2 instancje aktora z własnymi timerami.

Zastopuj na razie deploy. Sprawdź co się stanie w wspomnianym wcześniej delay w callbacku ze 100 sekundami. Sprawdź zachowanie ponownie zdeployowanej apki. Spróbuj zrobić w trakcie ponownego POSTa.

Wszystkie timery są zatrzymane kiedy aktor jest zatrzymany: nie ma żadnego calla z invoke po tym. W prawdziwym systemie, aktor może się sam aktywować, dezaktywować po danym okresie czasu, jego stan będzie persisted, ale oznacza to że dowolny z timerów będzie zatrzymany. Aby zapewnić że timery będą nadal wywoływane po dezaktywacji, musimy użyć Actor Remindera.

Actor Reminder jest mechanizmem do wywoływania persistent callbacków na aktorze w określonym czasie. Są podobne funkcjonalnie do timerów ale są wywoływani w określonych warunkach – nawet gdy aktor był dezaktywowany(dopóki go całkiem nie usunęliśmy).

* 1. Zaimplementujmy Remindera. Dodajemy implementację IRemindable w ProductActorService.cs I wyczyśćmy chwilowo timer kod powinien wyglądać tak:

|  |
| --- |
| namespace ProductActorService  {  [StatePersistence(StatePersistence.Persisted)]  internal class ProductActorService : Actor, IProductActorService, IRemindable  {  private string ProductStateName = "ProductState";  private IActorTimer \_actorTimer;  public ProductActorService(ActorService actorService, ActorId actorId)  : base(actorService, actorId)  {  }  public async Task AddProductAsync(Product product, CancellationToken cancellationToken)  {  await this.StateManager.AddOrUpdateStateAsync(ProductStateName, product, updateValueFactory: (key, value) => product, cancellationToken);  await this.StateManager.SaveStateAsync(cancellationToken);  }  public async Task<Product> GetProductAsync(CancellationToken cancellationToken)  {  var product = await this.StateManager.GetStateAsync<Product>(ProductStateName, cancellationToken);  return product;  }  protected override Task OnPostActorMethodAsync(ActorMethodContext actorMethodContext)  {  ActorEventSource.Current.ActorMessage(actor: this, message: $"{actorMethodContext.MethodName} has finished.");  return base.OnPostActorMethodAsync(actorMethodContext);  }  protected override Task OnPreActorMethodAsync(ActorMethodContext actorMethodContext)  {  ActorEventSource.Current.ActorMessage(actor: this, message: $"{actorMethodContext.MethodName} will start soon.");  return base.OnPreActorMethodAsync(actorMethodContext);  }  protected override Task OnDeactivateAsync()  {  if (\_actorTimer != null)  {  UnregisterTimer(\_actorTimer); }  ActorEventSource.Current.ActorMessage(actor: this, message: "Actor deactivated.");  return base.OnDeactivateAsync();  }  protected override Task OnActivateAsync()  {  this.RegisterReminderAsync("TaskReminder", null, TimeSpan.FromSeconds(10), TimeSpan.FromSeconds(15));  ActorEventSource.Current.ActorMessage(actor: this, message: "Actor activated.");  return this.StateManager.TryAddStateAsync("count", value: 0);  }  public async Task ReceiveReminderAsync(string reminderName, byte[] state, TimeSpan duetime, TimeSpan period)  {  if (reminderName == "TaskReminder")  {  ActorEventSource.Current.ActorMessage(actor: this, message: $"Reminder is doing work");  }  }  }  } |

Jak widać implementuje się to podobnie jak timer, z tym że musimy zaimplementować interfejs IReminderable i musimy zaimplementować metodę ReceiveReminderAsync(). Oczywiście, remindery są persisted nawet wtedy, gdy deaktywujemy aktora.

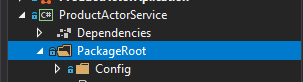
* 1. Robimy deploy I testujemy Postmanem jak poprzednio. Sprawdź zachowanie w Diagnostic Events. ☺

# Data Packages, konfiguracja I zmienne środowiskowe

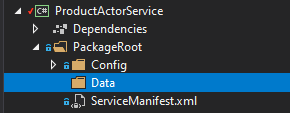
Przyjrzymy się teraz Data Packages, Config Packages i zmiennym środowiskowym – jak możemy je użyć w naszym Service Fabric.

Data Packages – pozwala nam na spakowanie pliku jako części naszego deploya Service Fabric: może być to plik JSON, XML, CSV, cokolwiek. Jak pamiętamy SF mają 3 główne komponenty: kod, konfigurację i dane.

* 1. Aby dodać plik z danymi do naszego serwisu, otwieramy w naszym serwisie PackageRoot:



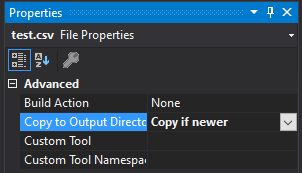
Tworzymy nowy folder



W środku tworzymy test.csv z danymi:

|  |
| --- |
| 1, Pawel, 35  2, Janek, 37  3, Michal, 32  4, Anna, 24 |

W properties pliku ustawiamy:



* 1. Następnie przechodzimy do naszego ProductActorService ServiceManifest.xml i dodajemy wpis o naszych danych pod <ConfigPackage />:

|  |
| --- |
| <DataPackage Name="Data" Version="1.0.0" /> |

Pozwoli to nam na dostęp do danych w naszym Actor Service.

Wracamy do ProductActorService.cs i dopisujemy na początku w OnActivateAsync():

|  |
| --- |
| var dataPackage = this.ActorService.Context.CodePackageActivationContext.GetDataPackageObject("Data");  var dataPath = Path.Combine(dataPackage.Path, "test.csv");  var contents = File.ReadAllText(dataPath); |

* 1. Zajmiemy się teraz konfiguracją. Zmieńmy konfigurację dla Local.5Node.xml, dopiszmy na końcu parametr:

|  |
| --- |
| <Parameter Name="DbConfig" Value="FiveNode" /> |

Robimy to samo dla 1Node, ale ustawiamy Value na OneNode. Pozwoli to nam na odwoływanie się do DbConfig niezależnie od wybranej konfiguracji działania naszego Azure Fabric Service.

Następnym krokiem będzie podmiana pliku ApplicationManifest.xml

|  |
| --- |
| <?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="ProductActorAplicationType" ApplicationTypeVersion="1.0.0" xmlns="http://schemas.microsoft.com/2011/01/fabric">  <Parameters>  <Parameter Name="ProductAPI\_ASPNETCORE\_ENVIRONMENT" DefaultValue="" />  <Parameter Name="ProductAPI\_InstanceCount" DefaultValue="-1" />  <Parameter Name="ProductActorServiceActorService\_PartitionCount" DefaultValue="10" />  <Parameter Name="ProductActorServiceActorService\_MinReplicaSetSize" DefaultValue="3" />  <Parameter Name="ProductActorServiceActorService\_TargetReplicaSetSize" DefaultValue="3" />  <Parameter Name="DbConfig" DefaultValue="" />  </Parameters>    <!-- Import the ServiceManifest from the ServicePackage. The ServiceManifestName and ServiceManifestVersion  should match the Name and Version attributes of the ServiceManifest element defined in the  ServiceManifest.xml file. -->    <ServiceManifestImport>  <ServiceManifestRef ServiceManifestName="ProductAPIPkg" ServiceManifestVersion="1.0.0" />  <ConfigOverrides />  <EnvironmentOverrides CodePackageRef="code">  <EnvironmentVariable Name="ASPNETCORE\_ENVIRONMENT" Value="[ProductAPI\_ASPNETCORE\_ENVIRONMENT]" />  </EnvironmentOverrides>  </ServiceManifestImport>    <ServiceManifestImport>  <ServiceManifestRef ServiceManifestName="ProductActorServicePkg" ServiceManifestVersion="1.0.0" />  <ConfigOverrides>  <ConfigOverride Name="Config">  <Settings>  <Section Name="Database">  <Parameter Name="DbConfig" Value="[DbConfig]"/>  </Section>  </Settings>  </ConfigOverride>  </ConfigOverrides>  </ServiceManifestImport>  <DefaultServices>    <Service Name="ProductAPI" ServicePackageActivationMode="ExclusiveProcess">  <StatelessService ServiceTypeName="ProductAPIType" InstanceCount="[ProductAPI\_InstanceCount]">  <SingletonPartition />  </StatelessService>  </Service>    <Service Name="ProductActorServiceActorService" GeneratedIdRef="8245c2b1-7c16-424d-b4bb-8a34e320b80b|Persisted">  <StatefulService ServiceTypeName="ProductActorServiceActorServiceType" TargetReplicaSetSize="[ProductActorServiceActorService\_TargetReplicaSetSize]" MinReplicaSetSize="[ProductActorServiceActorService\_MinReplicaSetSize]">  <UniformInt64Partition PartitionCount="[ProductActorServiceActorService\_PartitionCount]" LowKey="-9223372036854775808" HighKey="9223372036854775807" />  </StatefulService>  </Service>    <!-- The section below creates instances of service types, when an instance of this  application type is created. You can also create one or more instances of service type using the  ServiceFabric PowerShell module.    The attribute ServiceTypeName below must match the name defined in the imported ServiceManifest.xml file. -->    </DefaultServices>  </ApplicationManifest> |

Dodaliśmy tam konfigurację naszej bazy danych.

Następnie otwieramy Settings.xml w ProductActorService i podmieniamy:

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <Settings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.microsoft.com/2011/01/fabric">    <Section Name="ProductActorServiceActorServiceReplicatorConfig">  <Parameter Name="ReplicatorEndpoint" Value="ProductActorServiceActorServiceReplicatorEndpoint" />  <Parameter Name="BatchAcknowledgementInterval" Value="0.005" />  </Section>    <Section Name="ProductActorServiceActorServiceReplicatorSecurityConfig">  <Parameter Name="CredentialType" Value="None" />  </Section>  <Section Name="Database">  <Parameter Name="DbConfig" Value=""/>  </Section>  <!-- The content will be generated during build -->  </Settings> |

Nasz aktor ma teraz dostęp do bazy danych.

* 1. Dopiszmy teraz kod w Product Actor Service.cs i zakomentujmy nasz dataPath:

|  |
| --- |
| var dbConfig = this.ActorService.Context  .CodePackageActivationContext  .GetConfigurationPackageObject("Config")  .Settings  .Sections["Database"]  .Parameters["DbConfig"]  .Value;    /\*  var dataPackage = this.ActorService.Context.CodePackageActivationContext.GetDataPackageObject("Data");  var dataPath = Path.Combine(dataPackage.Path, "test.csv");  var contents = File.ReadAllText(dataPath);  \*/ |

Ustawmy pułapkę na dbConfigu. Sprawdź deploy dla 1Node i 5Node. Jaki Value został zwrócony?

* 1. W ApplicationManifest.xml podmieńmy ProductActorServicePkg:

|  |
| --- |
| <ServiceManifestImport>  <ServiceManifestRef ServiceManifestName="ProductActorServicePkg" ServiceManifestVersion="1.0.0" />  <ConfigOverrides>  <ConfigOverride Name="Config">  <Settings>  <Section Name="Database">  <Parameter Name="DbConfig" Value="[DbConfig]"/>  </Section>  </Settings>  </ConfigOverride>  </ConfigOverrides>  </ServiceManifestImport>  <EnvironmentOverrides CodePackageRef="code">  <EnvironmentVariable Name="DbPort" Value="8000" />  </EnvironmentOverrides>  <DefaultServices> |

Następnie w ServiceManifest.xml w ProductActorService dopisujemy pod</EntryPoint>:

|  |
| --- |
| <EnvironmentVariables>  <EnvironmentVariable Name="DbPort" Value=""/>  </EnvironmentVariables> |

* 1. W kodzie ProductActorService.cs komentujemy nasz dbConfig, dopisujemy nad nim:

|  |
| --- |
| var dbPort = Environment.GetEnvironmentVariable("DbPort"); |

Tak o to zamockowaliśmy dane do naszej bazy.

To tyle. Mam nadzieje że tutorial się spodobał. ☺