Cloud Enable an Existing WPF LOB App

# Setup

1. Open Expenses solution in Expenses - 04 After Hybrid Connection
2. Republish the WCF service to rgreenvslive3.
3. Reset 2 charges so they are not approved

# Demo 1: Existing App

1. Open Expenses - 01 Existing App\Expenses.sln.
2. Show the app.
3. Run the app.

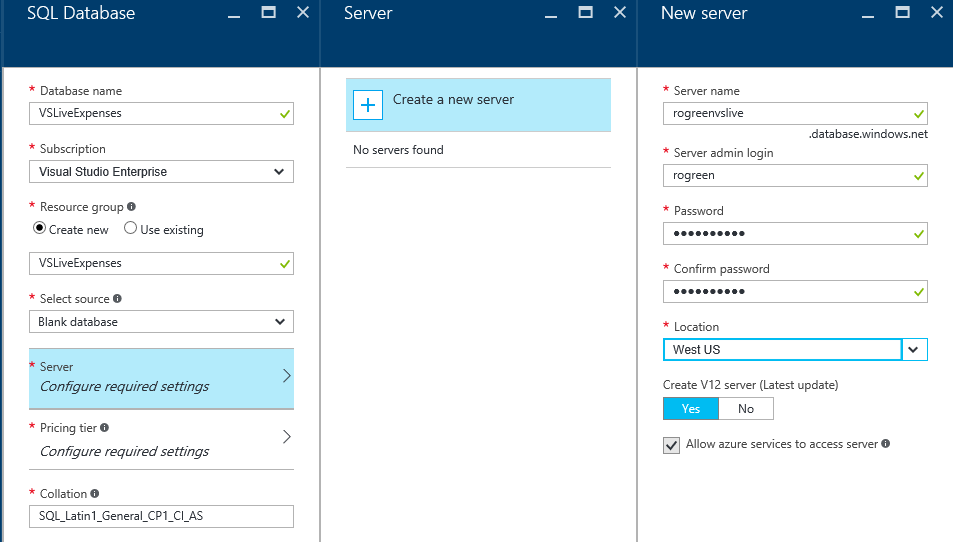
# Demo 2A: Move the Data to Azure SQL

## Review Existing Data

1. View data.
   1. Open SQL Server Object Explorer.
   2. Expand DefaultConnection
   3. Expand Tables
   4. Show Table Data for DbCharges
2. Show properties for DefaultConnection
3. Open SQL Server Management Studio
   1. Connect to (LocalDb)\MSSQLLocalDB
   2. Expand Expenses.Data
   3. Expand Tables

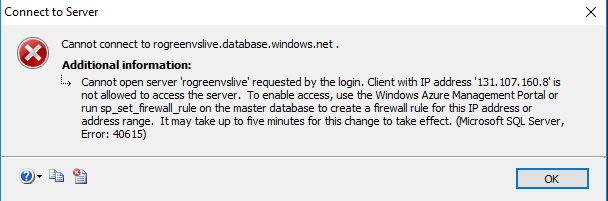
## Create the Azure SQL Database

1. Open the Azure portal.
2. Show how you created a new database.



Password is VSLive2016

1. Show VSLiveExpenses database in the portal.
2. Connect to rogreenvslive.database.windows.net in SQL Mgmt Studio.



1. Back in the portal, click Set server firewall.
2. Notice the choice to Add client IP.



1. That is not the right client ID.
2. Add the correct IP.
3. DO NOT FORGET TO SAVE YOUR CHANGES!!
4. Connect in SQL Mgmt Studio.
5. Show that once you have connected to the server, you could create a new database.
6. Show VSLiveExpenses database. No tables.

## Copy the Local Data to Azure SQL

1. Right-click Expenses.Data.
2. Select Tasks|Export Data.
3. Show the steps to use the Import and Export Wizard to export the data.
4. Note that you need to include \_\_MigrationHistory.
5. View the data in the Azure database.
6. Change the first merchant.

Update DbCharges

Set Merchant='Azure Northwind Inn'

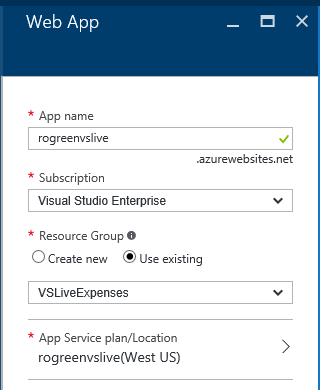
Where ChargeId = '1'

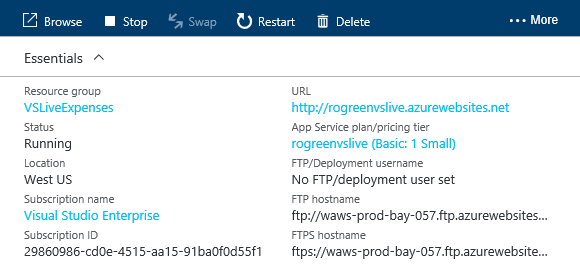
## Connect the App to the Database

1. In the Azure portal, select Show database connection strings for the database
2. Copy the ADO.NET connect string.
3. Open Expenses - 01 Existing App\Expenses.sln.
4. Open Web.config.
5. Comment out the local DefaultConnection.
6. Uncomment the Azure DefaultConnection.
7. Run the app.

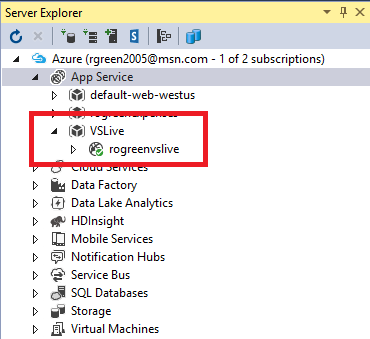
# Demo 2B: Move the WCF Service to Azure

1. Open the Azure portal.
2. Show how you created a new Web App.





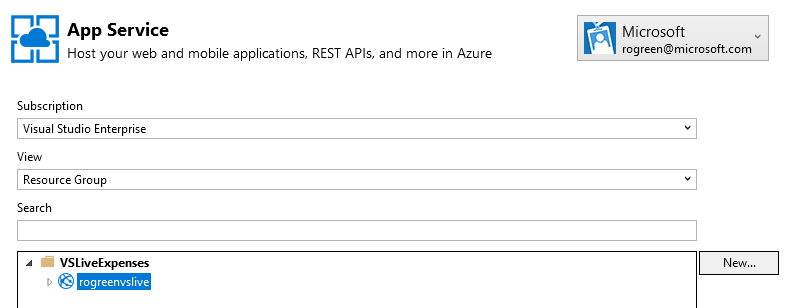
1. Click the URL or Browse to see the Web App.
2. In VS, connect to Azure in the Server Explorer.
3. Expand App Service and see the Web App.

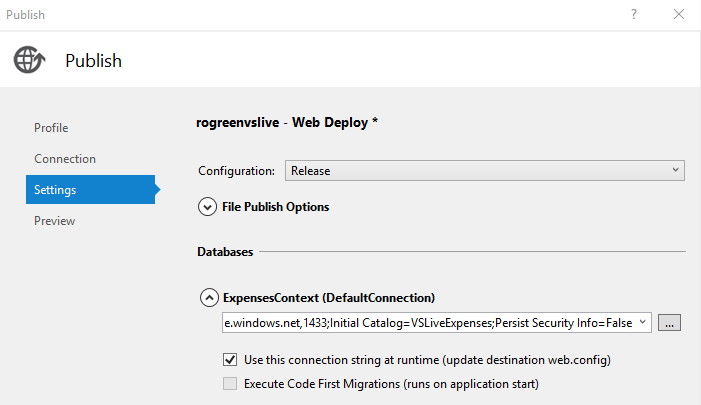


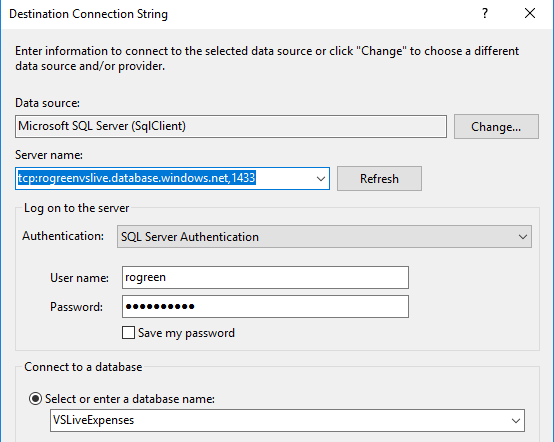
1. Right click on App Service and select Create New App Service to show that you could have created Web App from here.

## Publish WCF Service

1. Right click WCF Service and select Publish.
2. Use Publish Wizard to publish service.



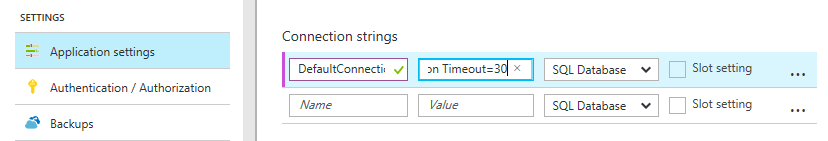




1. Navigate to <http://rogreenvslive.azurewebsites.net/expenseservice.svc> to confirm WCF service is there.
2. Copy URL.
3. In App.Config, comment out the localhost endpoint address.
4. Uncomment the Azure endpoint address and put it in the right place.
5. Set Expenses.WPF as startup project.
6. Run app.

## Hide the Password

1. In the Azure portal, go to Web App tile.
2. Click Application Settings.
3. Add a DefaultConnection connection string.



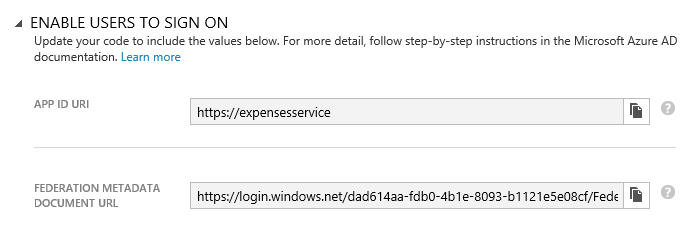
1. Copy the connection string from Web.config. Save.
2. In Web.config, comment out the connection strings.
3. Republish.
4. Run app.

# Demo 3A: Secure Service with Active Directory

1. Open the Azure portal.
2. Show how you created a new AD Tenant.
3. Show how you created the service application.

|  |  |
| --- | --- |
|  |  |

1. Click Enable Users To Sign On. You will need this info later.



## Secure the WCF Service

1. Added 3 NuGet packages

Install-Package Microsoft.IdentityModel.Clients.ActiveDirectory -Version 2.12.111071459

Install-Package Microsoft.Owin.Security.ActiveDirectory -Version 3.0.0

Install-Package Microsoft.Owin.Host.SystemWeb -Version 3.0.0

1. Add app settings keys to Web.config

<add key="aspnet:UseTaskFriendlySynchronizationContext" value="true" />

<add key="aad:Tenant" value="rogreenvslive.onmicrosoft.com"/>

<add key="aad:Audience" value="https://expensesservice"/>

<add key="aad:MetadataAddress"

value="https://login.windows.net/9eaa5bfd-bc6e-4da1-b152-d3029c110e7a/FederationMetadata/2007-06/FederationMetadata.xml"/>

1. The next step is to configure the usage of Azure Active Directory bearer authentication. The client will send bearer tokens to the service to authenticate each request. Right-click on Expenses.Wcf and select Add | Existing Item.
2. Open the Startup.cs file from the DemoFiles folder.
3. Open Startup.cs file in the code editor. This file will perform the necessary configuration to process the bearer tokens send with each request. Note that it has an assembly level attribute applied to it for OwinStartup.
4. We would like to be able to perform some claims based authentication in our service, so we can implement IAuthorizationPolicy to help with this. All WCF requests will be evaluated by this code once we get it configured in Web.config. Add the HttpContextAuthorizationPolicy.cs file from the DemoFiles folder to the project.
5. Right-click on the References node for the service project and select Add Reference. Add

System.IdentityModel

System.IdentityModel.Services

System.IdentityModel.Tokens.Jwt.dll

1. Open HttpContextAuthorizationPolicy.cs in the editor. Note that Evaluate method will look at the current HTTP context and add Principal and Identities properties to the current evaluation context.
2. Add the following to Web.config to configure the custom authorization policy that we just added to the project. This is a behavior that should go in the <system.ServiceModel><behaviors><serviceBehaviors><behavior> section.

<serviceAuthorization principalPermissionMode="Custom">

<authorizationPolicies>

<add policyType="Expenses.Wcf.HttpContextAuthorizationPolicy, Expenses.Wcf"/>

</authorizationPolicies>

</serviceAuthorization>

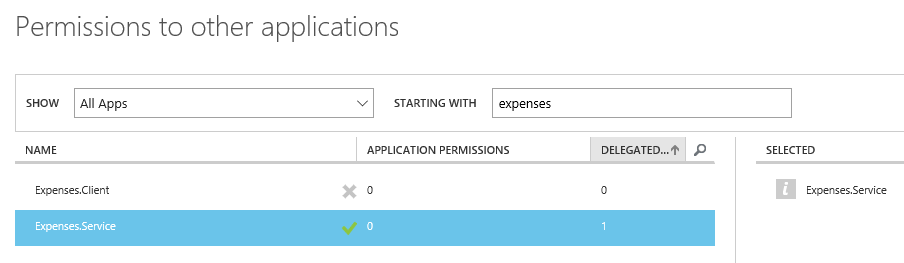
1. Now we just need to modify the service logic itself to check for authorization. Overwrite the ExpenseService.svc.cs file with the one found in the DemoFiles folder (easiest to do this using Explorer window).
2. Double-click on ExpenseService.svc.
3. The difference here is that we now have an AuthorizationCheck method in place that checks the claims of the current principal, specifically that the user has a scope value of “user\_impersonation”, or full access to the service. If the current principal does not have that claim, the proper unauthorized error code will be returned.
4. Save.
5. **Show that the app is published to rogreenvslive2.**
6. **Show that the WPF app calls the service on that Web App.**
7. Run the app. Unauthorized.

# Demo 3B: Call the Secured Service from the WPF App

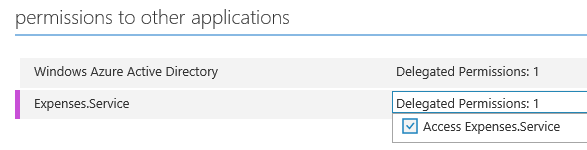
1. Open the Azure portal.
2. Show how you created the client application.

|  |  |
| --- | --- |
|  |  |

1. Click Configure
2. Click Add Application



1. Select Access Expenses.Service from Delegated Permissions. Save.



1. In the Package Manager Console window, ensure that the Expenses.WPF project is selected.
2. Use the following command to install the Active Directory Authentication Library (ADAL):

Install-Package Microsoft.IdentityModel.Clients.ActiveDirectory -Version 2.12.111071459

1. With the ADAL package added to the project, we can now use the library to acquire the necessary token from Azure Active Directory to be able to authenticate with the Expenses service. Start by creating a new folder in the WPF project named “AAD”.
2. Add the following files from the DemoFiles folder to this new AAD folder:
   * AADSignIn.cs
   * ExpensesClientMessageInspector.cs
   * FileCache.cs
3. Open AADSignIn.cs in the editor. This class was created to perform the sign in process using ADAL. It first creates an AuthenticationContext and then it calls the AcquireToken method with the appropriate settings to request a token from your tenant settings. Once the authentication result is determined, it is then stored for later used when making calls to the Expenses WCF service.
4. Right-click on the References node for the service project and select Add Reference. Add

System.Configuration

System.Security

1. Open FileCache.cs in the editor. This class inherits from the TokenCache class from the ADAL and is responsible for persisting tokens in a local file protected using the Data Protection API (DPAPI). This functionality is used by the AuthenticationContext that we just looked at in the AADSignIn class.
2. Open ExpensesClientMessageInspector.cs in the editor. This class implements a WCF message inspector (by implementing IClientMessageInspector) for the purposes of adding in the appropriate bearer authorization header to outgoing service requests. The ADAL is utilized once again here in order to create the appropriate authorization header with a call to CreateAuthorizationHeader.
3. There are a couple of more classes in the ExpensesClientMessageInspector.cs file that allow this message inspector to be specified in configuration.
4. Now we need to make a call to the SignIn method that we added when the WPF application starts up. Open App.xaml.cs in the editor and add the following line of code to the start of the OnStartup method.

AADSignIn.SignIn();

1. The final step is to add the expected configuration to App.config. Start by adding in the following application settings section with the expected keys:

<appSettings>

<add key="aad:Tenant" value="rogreenvslive.onmicrosoft.com" />

<add key="aad:ClientId" value="5bee5ad3-f9d1-4a27-82a2-03c2e7dcb838" />

<add key="aad:RedirectUri" value="https://expensesclient" />

<add key="aad:ResourceId" value="https://expensesservice" />

<add key="aad:AADInstance" value="https://login.windows.net/{0}" />

</appSettings>

1. Now we need to configure the WCF behavior extension to load our ExpensesClientMessageInspector so that outgoing requests will include the necessary bearer token. Copy the following configuration and add it to the <system.serviceModel> section.

<extensions>

<behaviorExtensions>

<add name="bearerTokenRequired" type="Expenses.WPF.ExpensesClientMessageInspectorExtensionElement, Expenses.WPF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"/>

</behaviorExtensions>

</extensions>

<behaviors>

<endpointBehaviors>

<behavior name="bearerTokenRequiredBehavior">

<bearerTokenRequired />

</behavior>

</endpointBehaviors>

</behaviors>

1. Finally add the following property to the client endpoint definition in the configuration file so that the endpoint uses the bearer token behavior:

behaviorConfiguration="bearerTokenRequiredBehavior"

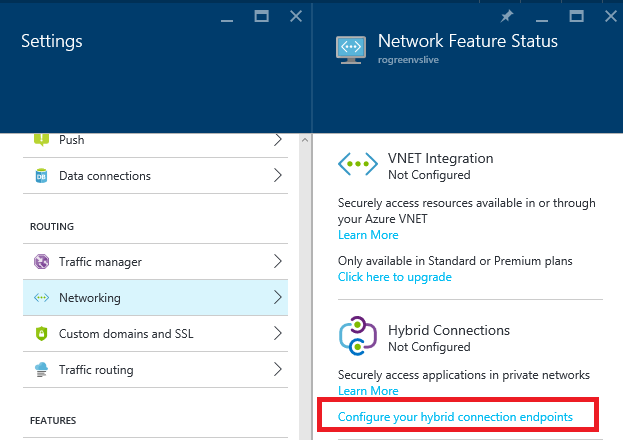
1. Run the app.
2. If you are not prompted to log in, make the following change in AADSignin.cs

AADSignIn.AADAuthResult = authContext.AcquireToken(serviceResourceId, clientId, redirectUri, PromptBehavior.Always);

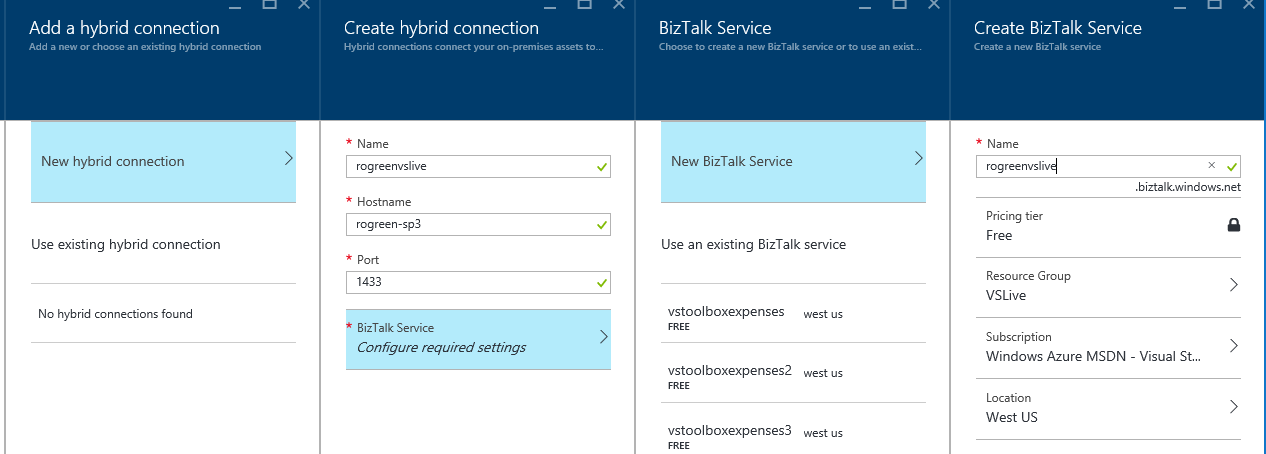
1. Run the app.
2. Sign in correctly.
3. Run the app again.
4. Sign in incorrectly. Unauthorized.

# Demo 4: Use Hybrid Connection to Connect to On-Prems Database

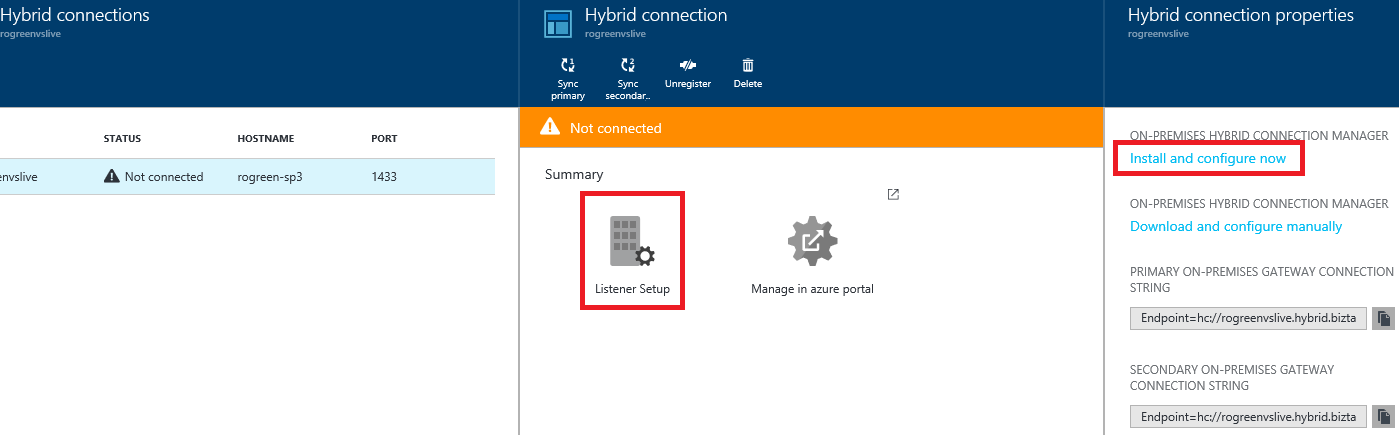
1. Show that the app is published to rogreenvslive3.
2. Show that the WPF app calls the service on that Web App.
3. In Azure Portal, go to the Web App.
4. Select Networking | Configure your hybrid connection endpoints.



1. Click Add



1. Click OK twice
2. Set up the listener



1. Wait for hybrid connection to show that it is connected. Refresh portal if necessary.
2. Click on Application Settings
3. Rename DefaultConnection to OldDefaultConnection
4. Add a new DefaultConnection and enter the following for connection string

Data Source=rogreen-0829;Initial Catalog=Expenses.Data;User ID=sa;Password=Passw0rd!

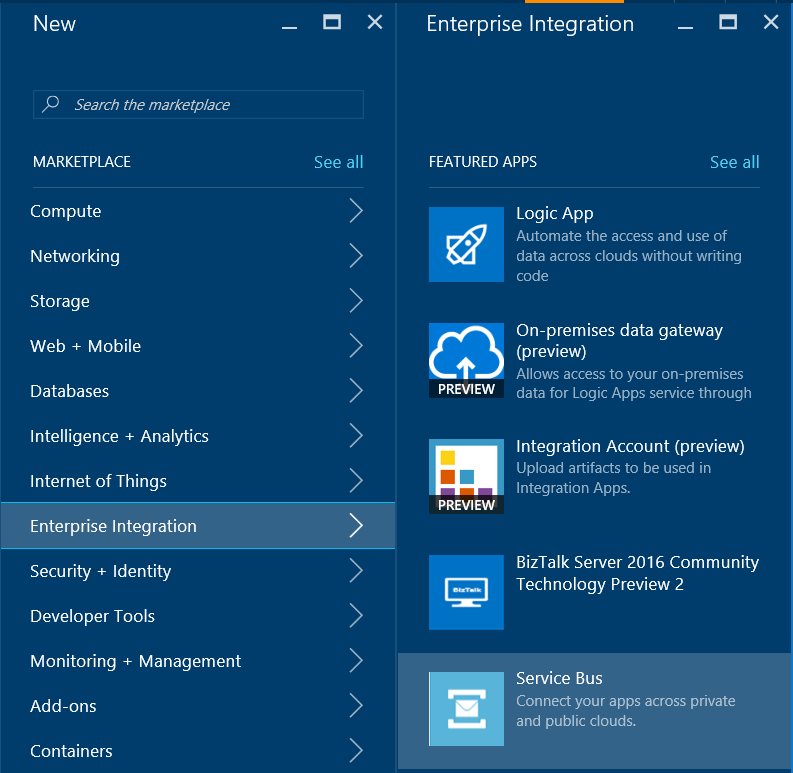
1. Save
2. Run app
3. If you get the arithmetic overflow error, make the fix in kb2643584.

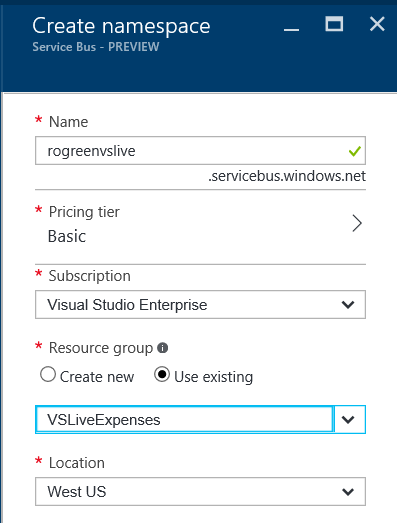
# Demo 5: Use Service Bus Relay to Connect to On Premises Service

1. Start another Visual Studio
2. Load InternalService solution
3. Show there is breakpoint in ProcessReport method
4. Select InternalService.svc and F5 to launch
5. Return to first VS and show WCF has service reference to InternalService.
6. In ExpenseService.svc.cs, find the UpdateExpenseReport method and show call to InternalService
7. Republish the app to rgreenvslive3.

## Create a Service Bus Namespace

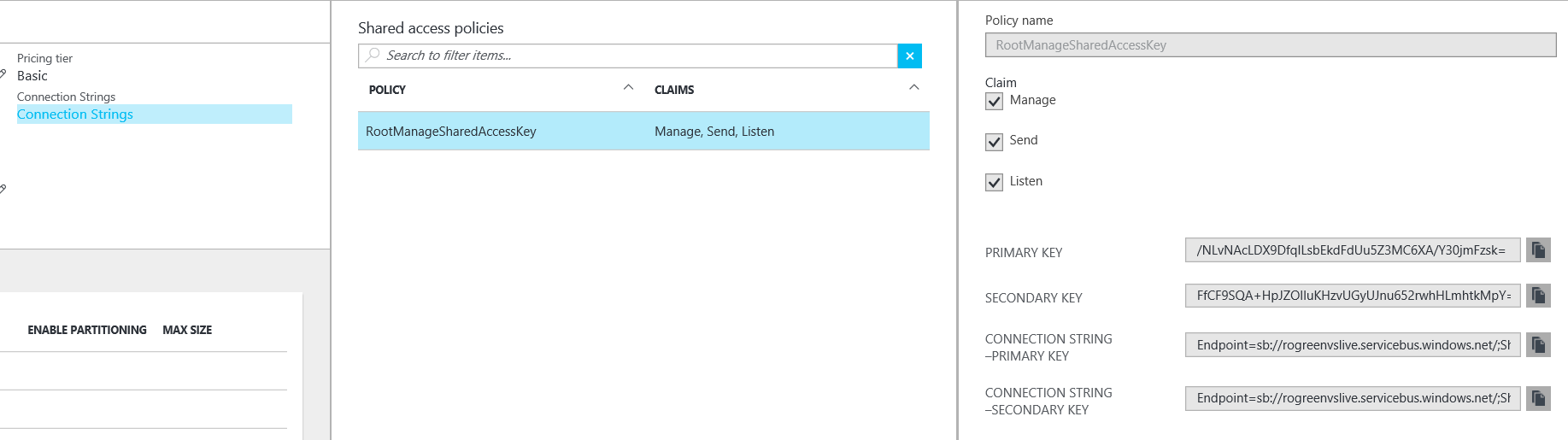
1. In the Azure Portal, show how you created a new Service Bus Relay





NOTE: Namespaces created via the portal will be setup to use Shared Access Signature (SAS) authentication. This provides an access key that is then used to generate a SAS token that clients can use to authenticate. This is recommended for applications scenarios where there is no need to manage the notion of an authorized “user”. In the event that the application scenario requires a richer, identity-based authentication option to perhaps federate with other standards-based identity providers (ADFS, Microsoft accounts, Google, Facebook), you can also create namespaces that use the Access Control Service (ACS).

1. Click Connection Strings. We will need this information shortly.



1. Return to InternalService project
2. Install the following

Install-Package WindowsAzure.ServiceBus -Version 2.5.3.0

1. Look at all the extensions added to Web.config
2. Add the following endpoint:

<endpoint contract="Expenses.InternalService.IInternalService"

binding="netTcpRelayBinding"

address="sb://rogreenvslive.servicebus.windows.net/internalservice"

behaviorConfiguration="sbTokenProvider"/>

1. Add the following into the Behaviors section

<endpointBehaviors>

<behavior name="sbTokenProvider">

<transportClientEndpointBehavior>

<tokenProvider>

<sharedAccessSignature keyName="RootManageSharedAccessKey" key="/NLvNAcLDX9DfqILsbEkdFdUu5Z3MC6XA/Y30jmFzsk=" />

</tokenProvider>

</transportClientEndpointBehavior>

</behavior>

</endpointBehaviors>

1. Select InternalService.svc and F5 to launch
2. In the Azure Portal, select Relays. InternalService is listed

## Call Internal Service from WCF Service

1. In WCF project add the following

Install-Package WindowsAzure.ServiceBus -Version 2.5.3.0

1. In Web.config comment out the client endpoint
2. Replace it with the following:

<endpoint contract="InternalService.IInternalService"

binding="netTcpRelayBinding"

address="sb://rogreenvslive.servicebus.windows.net/internalservice"

behaviorConfiguration="sbTokenProvider"/>

1. Add the sbTokenProvider behavior

<endpointBehaviors>

<behavior name="sbTokenProvider">

<transportClientEndpointBehavior>

<tokenProvider>

<sharedAccessSignature keyName="RootManageSharedAccessKey"

key="/NLvNAcLDX9DfqILsbEkdFdUu5Z3MC6XA/Y30jmFzsk=" />

</tokenProvider>

</transportClientEndpointBehavior>

</behavior>

</endpointBehaviors>

1. Republish
2. Run the app
3. Approve an expense report