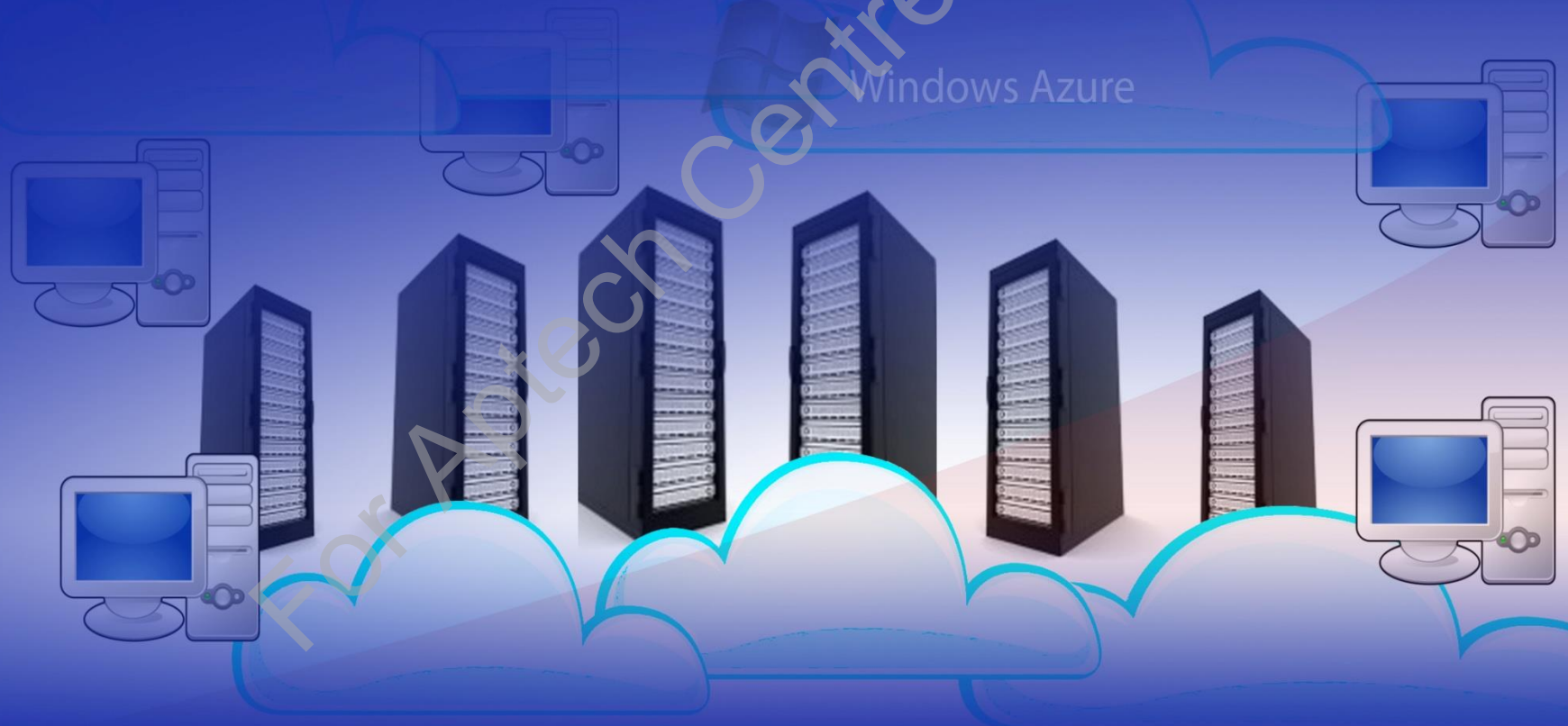
Enterprise Application Development Using Windows Azure and Web Services

Session 10

Service Bus in Windows Azure



Learning Objectives

- Explain Service Bus in Windows Azure
- Describe queues, topics, and relays
- State the use of AMQP in Service Bus
- Describe the use of Service Bus relays in cloud hybrid applications

Service Bus 1-3

- **Service Bus:**
  - Is a core feature of Windows Azure that acts as a messaging channel.
  - Sits between various components of the cloud app or between the cloud and on-premises applications and enables them to communicate through messages.
  - Is based on the multi-tenant concept.
- According to this concept, several users can use the same service.
- Application developers will create a namespace within which one or more communication mechanisms will be defined.

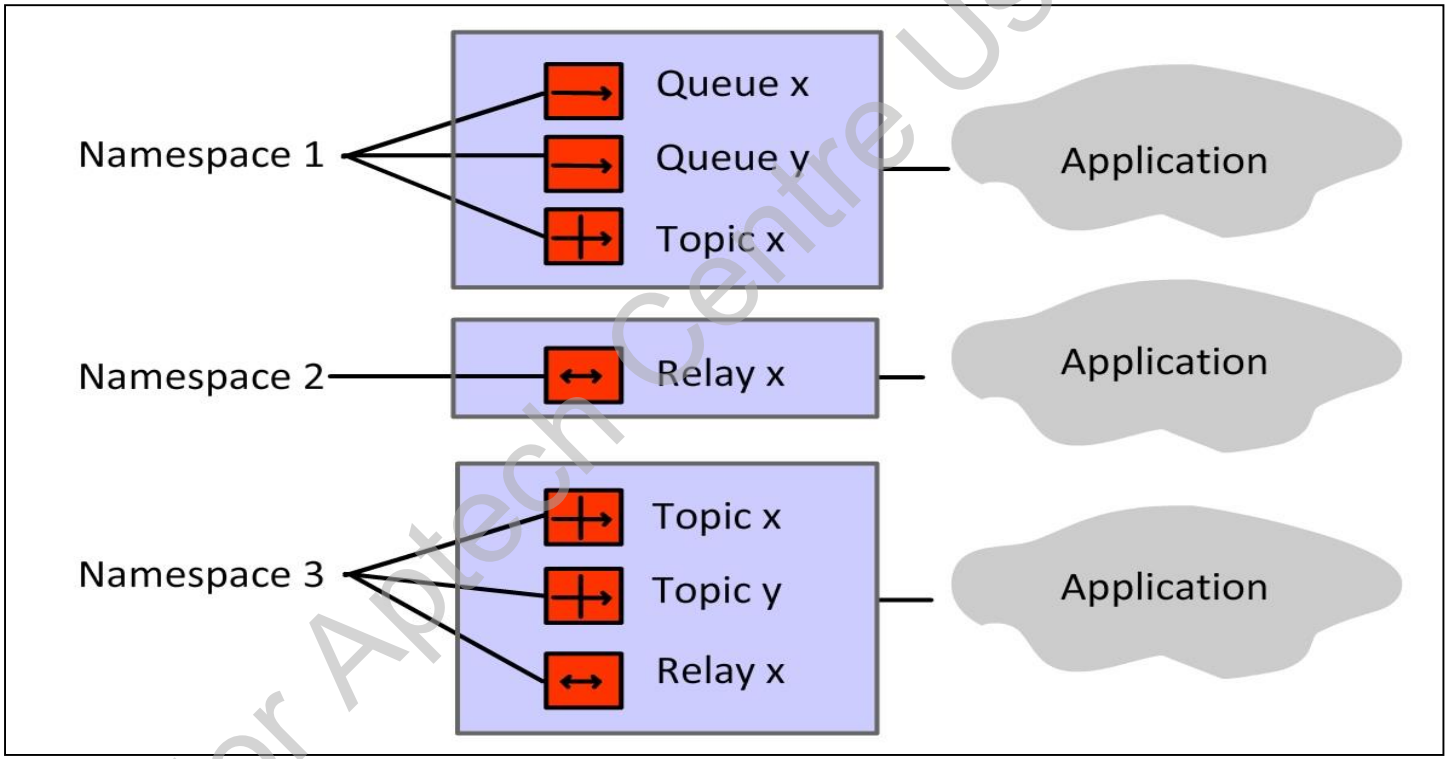
Service Bus 2-3

- An application developer can use from any one of the three communication mechanisms offered:

Queues	Topics	Relays
• Allows unidirectional communication, that is, every queue needs to store messages until they are received.	• Offers unidirectional communication. • Uses subscription and based on these criteria, it shows the message to the subscription.	• Offers bi-directional communication. • Passes messages to the application, that is, the destination application.

Service Bus 3-3

- Following figure depicts an overview of a Service Bus:



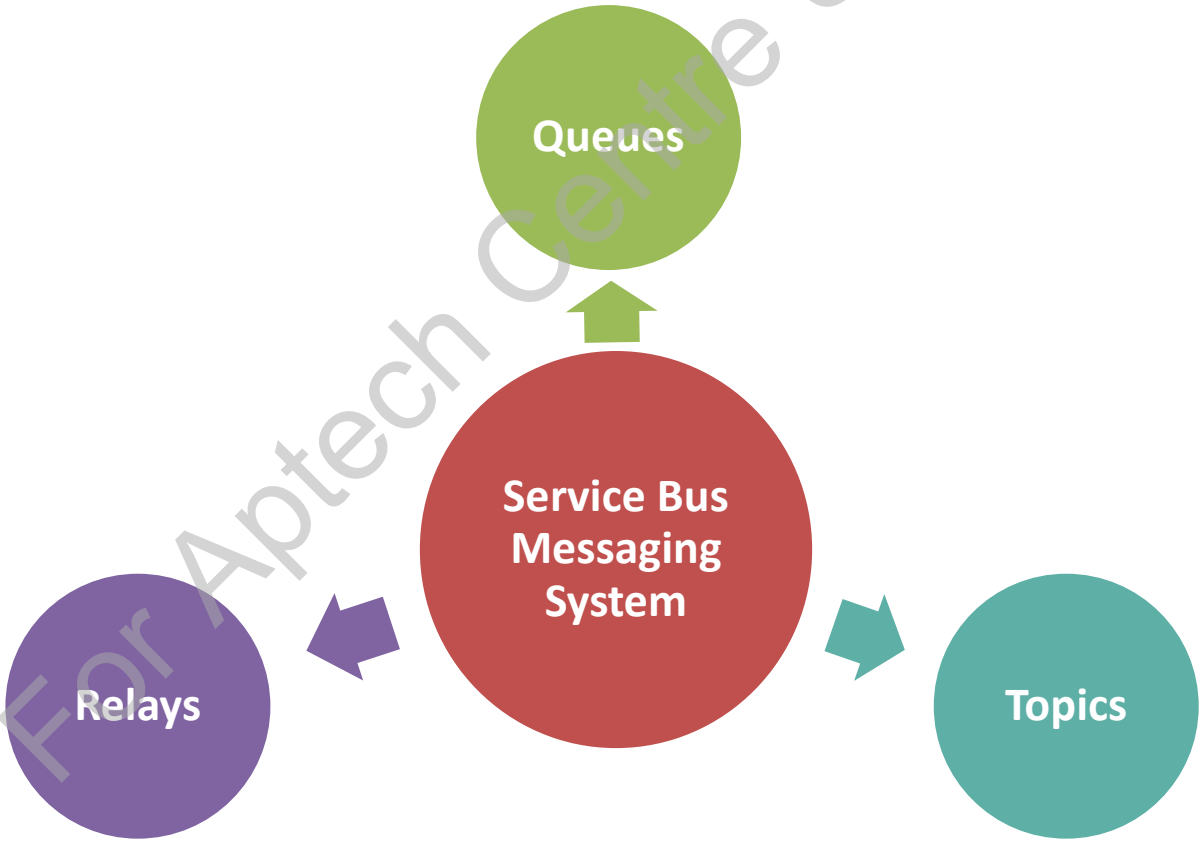
Advantages of Service Bus

- Azure Service Bus helps in:

- Managing the delivery of messages in the cloud
- Connecting the cloud to on-premises applications
- Pushing notifications easily to mobile apps or devices
- Controlling the access to services at a basic level
- Exposing the application functionality and data from the existing enterprise solutions and taking advantage of it from the cloud

Understanding the Service Bus Communication Mechanisms

- Integral aspects to the Service Bus messaging system are:



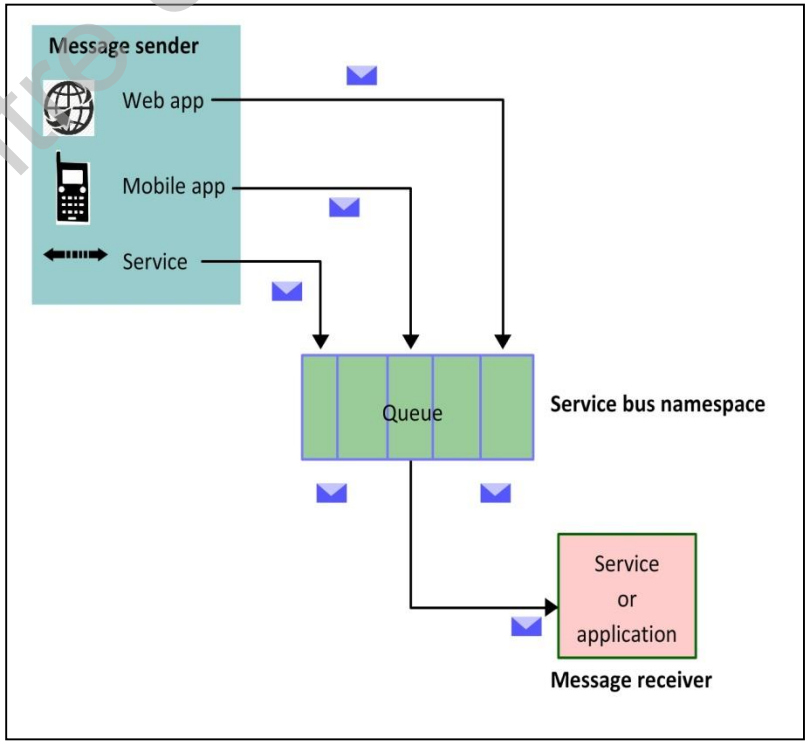
Queues 1-3

Queues

- Service Bus Queues works on the brokered messaging communication model and act as a mediator.
- A distributed application exchanges messages through queues and does not communicate directly.
- Queues follow First In First Out (FIFO) message delivery system.

Queues 2-3

- **The sender:**
  - Sends the message to the queue; the receiver receives the message from the queue to processes it further.
  - Can send further messages without waiting for a reply.
- The messages are received and processed by only one message receiver in the same sequence that was followed for sending to the queue.

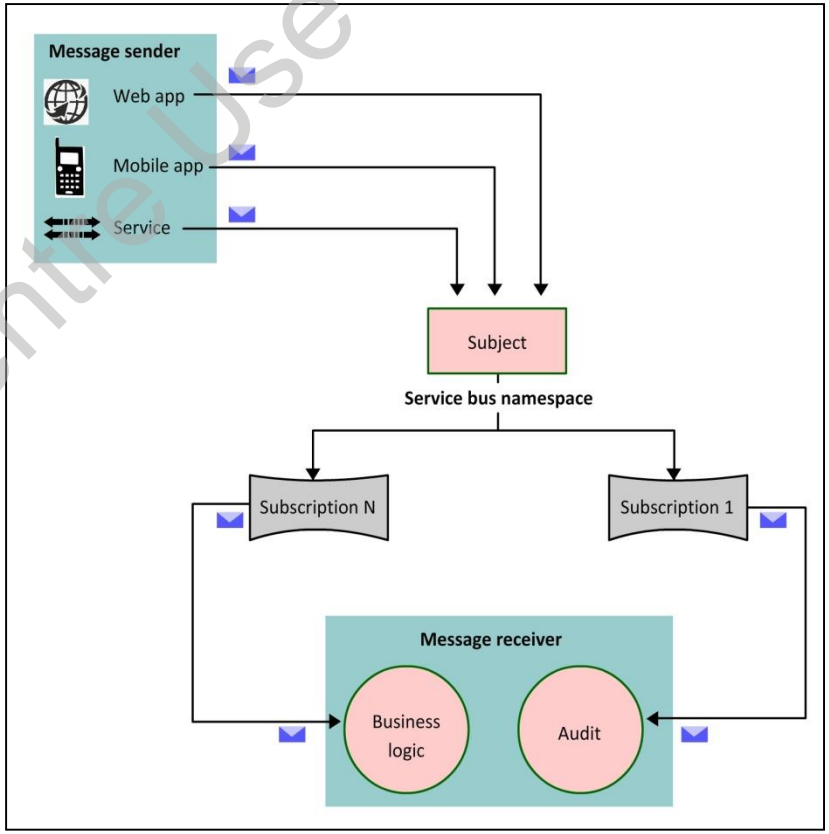


Queues 3-3

- Service Bus queues can be used for communication:
  - In a multi-tier Azure application that involves Web and worker roles
  - In a hybrid solution that involves on-premises apps and Azure hosted apps
  - In different organizations of a distributed application running on-premises
- Queues help to scale out the applications to improve architecture and resiliency.

Topics 1-2

- Service Bus topics works on a publish/subscribe messaging communication model and acts as a mediator.
- A distributed application exchanges messages through a topic and does not communicate directly.

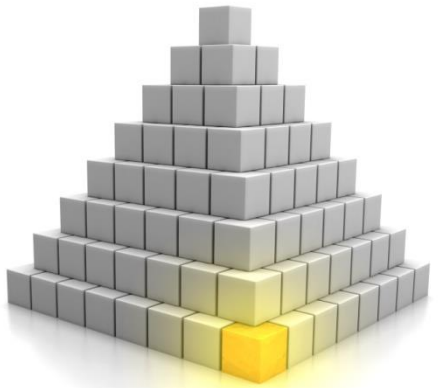


Topics 2-2

- Topics/subscriptions follow one-to-many form of communication using publish/subscribe pattern.
- Multiple subscriptions to a topic can be registered.
- Each message sent to a topic is made available to each subscription to process.
- Filter rules can be registered for a topic for each subscription to filter the topic of messages received by a topic subscription.
- Service Bus topics and subscriptions help scale large number of messages to a large number of applications and users.

Relays 1-6

- Enterprise applications comprise several features in the form of services, components, and so on.
- Consolidating all these diverse components together into the single system is not easy even when all of them reside locally.
- If some of the components reside on a cloud, then it becomes even more tough.



Components

Relays 2-6

- Consider a Windows Azure based enterprise application:
  - Having Web and worker roles
  - Storing its data in SQL Database
  - Interacting with third-party provider services for authentication or other tasks

- The application may also make use of some local components that cannot be migrated to the cloud.

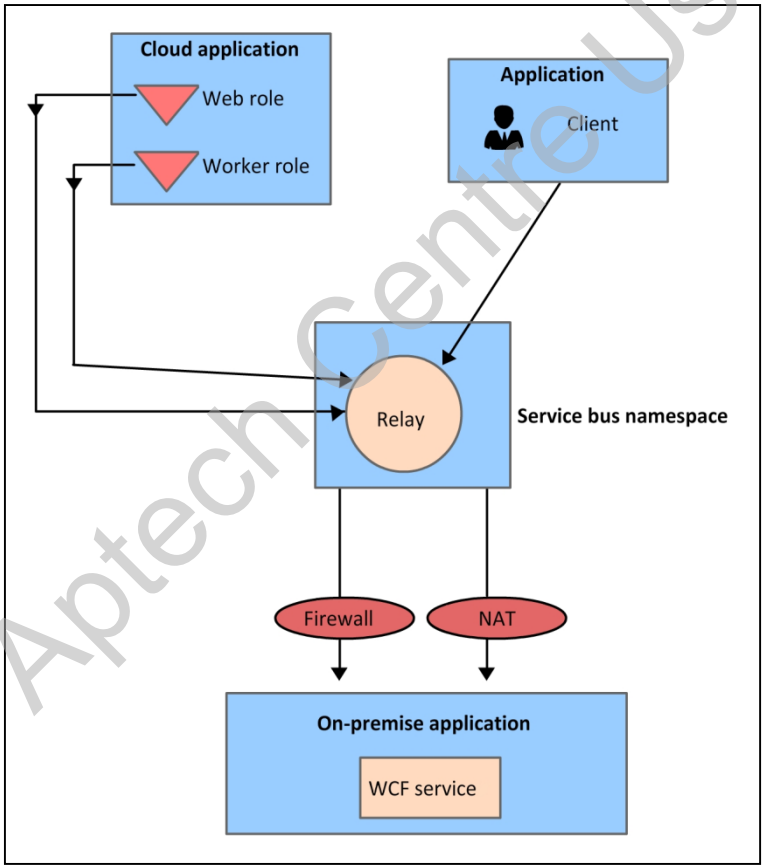
- Such applications are called cloud hybrid applications.

Relays 3-6

- The Service Bus Relay:
  - Helps to build cloud hybrid application that run in an Azure datacenter and on-premises environment.
- In cloud hybrid applications:
  - The service bus relay is quite useful as it takes existing WCF Web services and makes them accessible to cloud-based solutions without compromising on security or infrastructure.

Relays 4-6

- Following figure depicts a Service Bus relay:



Relays 5-6

- WCF services can be hosted in the existing enterprise environment using the Service Bus relay.
- Incoming sessions and requests to these WCF services can be delegated to the Service Bus running in Azure.
- The services are then exposed to the application code running in Azure or to mobile workers or extranet partner environments.

Relays 6-6

- Service Bus offers two types of messaging capabilities:

Relayed	• Allows request/response messaging, direct one-way messaging and peer-to-peer messaging
Brokered	• Allows Subscriptions, Topics, and Queues that are components of asynchronous messaging

Creating a Windows Azure Service Bus Queue 1-3

- Service Bus queues offer messaging capabilities, which help various applications to run in the cloud or on-premises to exchange messages across trust and network boundaries in a flexible manner.

Creating a Windows Azure Service Bus Queue 2-3

- Some of the common ways of creating a Windows Azure Service Bus Queue are:

.NET Code

- You can create queues from .NET code by downloading the NuGet package from Visual Studio.

Windows Azure Portal

- You can create queues from the Windows Azure Portal.
- For this, you need to login to the portal, choose the **Service Bus** option from the menu icons that are on the left pane.
- Choose the namespace where you want to create a queue and click **New**.

Creating a Windows Azure Service Bus Queue 3-3

Service Bus Explorer Tool

- Download and install the Service Bus Explorer. Then, right-click **Queues** and select the **Create Queue** option.

Windows Azure SDK for Visual Studio

- You can use the connection string to create a new Windows Azure Service Bus.
- After the connection is established, the queues can be directly created from Visual Studio.

AMQP Support

- AMQP is a consistent and well-organized wire-level protocol that helps to build easy messaging applications between different vendor products.

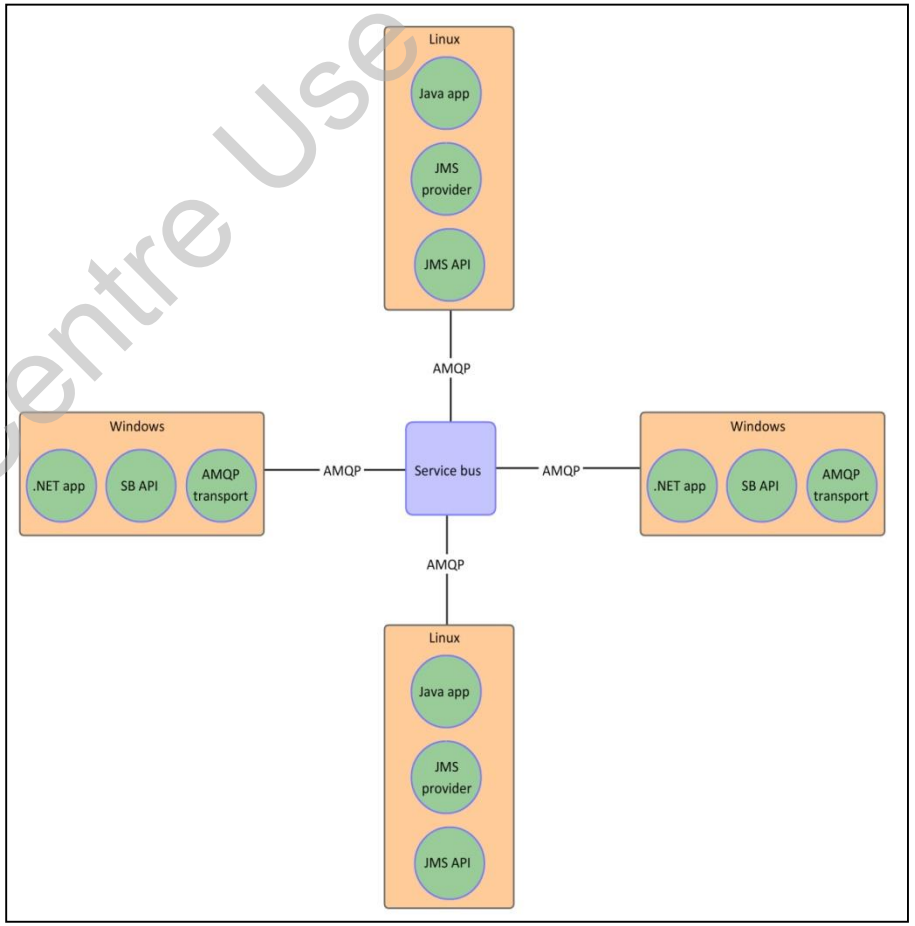
- Following are the features of AMQP:
  - Enables building cross-platform and hybrid applications using an open standard protocol
  - Constructs applications using components with different languages and frameworks
  - Runs the applications on different operating systems
  - Connects Service Bus and exchanging business messages easily and efficiently
  - Provides flexible protocol that supports communications at all levels
  - Exchanges reliable messages
  - Supports existing messaging brokers

Using AMQP in Service Bus 1-2

- AMQP 1.0 influences the queuing, publishing, or subscribing brokered messaging features of Service Bus.
- Applications can be built using different languages, operating systems, and frameworks.

Using AMQP in Service Bus 2-2

- The figure shows deployment in which messages are exchanged through Service Bus using AMQP 1.0 for the Java clients.
- These run on Linux and are written using the standard Java Message Service (JMS) API and .NET clients that run on Windows.



Notification Hub 1-4

- Azure Notification Hubs have a Push feature that helps the consumers and enterprise applications for mobile platforms to access the infrastructure with ease.

- Push Notification:

  - Is a feature that notifies users about an event that has occurred.
  - Is found exclusively in tablets and smartphones.
  - For example, in Windows Store applications, the notification appears in a window with a sound that indicates a new push.

Notification Hub 2-4

- Push Notifications offer following benefits:
  - Enable mobile devices to display new information keeping the energy intact
  - Increase app engagement and usage
  - Update information to employees regularly
  - Increase user awareness

Notification Hub 3-4

- Platform Notification Systems (PNS) are platform-specific infrastructures that are used to notify Push Notifications.

- For example:
  - A developer needs to contact the Windows Notification Service (WNS) to notify a Windows Store app. Here, PNS is implemented as WNS.

Notification Hub 4-4

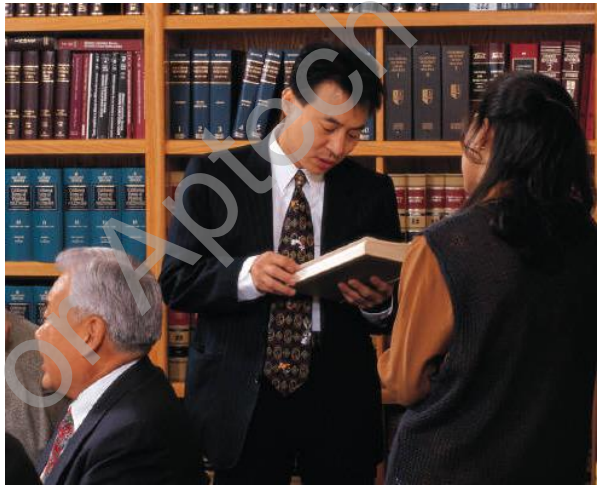
- The limitations of Push Notifications include:

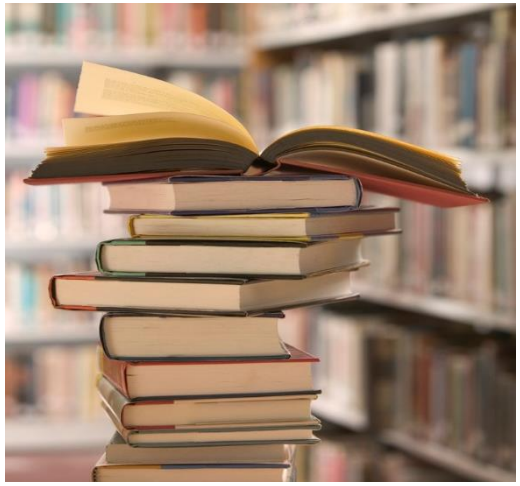
Dependency on different platforms	Constant updating and refreshing	Route notifications
• Used to notify devices on different platforms.	• Is required each time the app is launched leading to traffic.	• Increases the maintenance costs of an app.

Working with Service Bus 1-2

- The Service Bus is useful in integrating on-premises and cloud hybrid applications.

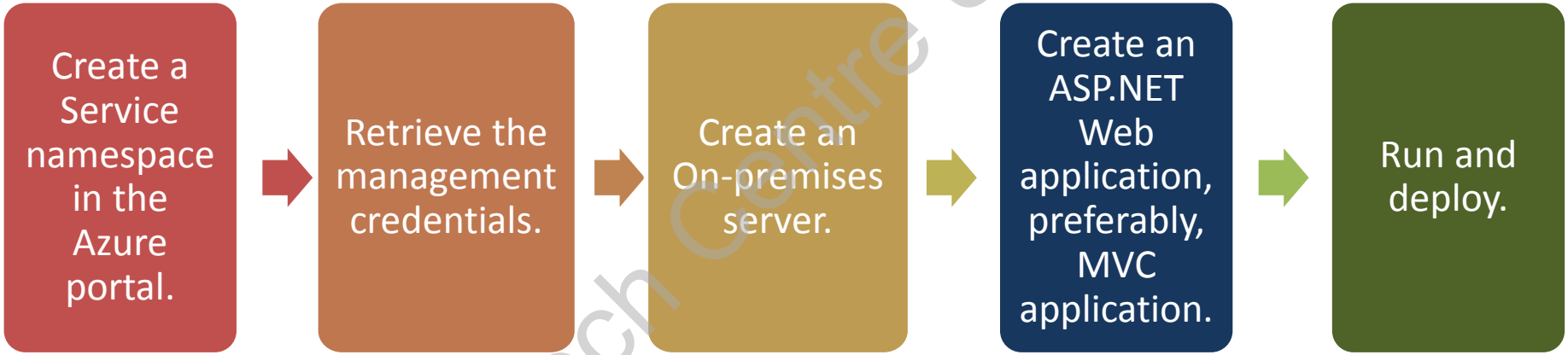
- Consider an example of a library system maintaining books and various details pertaining to the books.





Working with Service Bus 2-2

- Fundamental tasks to be performed for this application are:



Service Bus Namespace Setup Using the Management Portal

- Steps for Service Bus namespace setup using the Management Portal:

Step 1	Log in to the **Azure Management Portal** by typing the username and password**.**
Step 2	From the list of items on the left, select **Service Bus**.
Step 3	From the options mentioned at the bottom of page, select **CREATE**.
Step 4	Type the name in the **Namespace Name** field to create a new namespace.
Step 5	Select the region. Assume that you have selected East Asia for the current example.
Step 6	Click the check mark. This helps to create and enable the service namespace.

Retrieve the Management Credentials

- Following are the steps to retrieve the management credentials:

Step 1	• Select the name of the service namespace in the main window.
Step 2	• Select **Connection Information** from the options mentioned at the bottom of page.
Step 3	• Find the **Default Issuer** and **Default Key values** in the **Access connection information** pane.
Step 4	• Copy the key to the clipboard, which will be used for creating an on-premises server.

Create On-Premises Server 1-5

- Steps to create an on-premises server using Visual Studio 2013:

Step 1	• Go to Microsoft Visual Studio 2013. Right-click and select **Run as administrator**.
Step 2	• Click **File** → **New**, and then click **Project**.
Step 3	• Under **Installed Templates and Visual C#,** click **Console Application**. Type **BooksServer** in the **Name** box.
Step 4	• Click **OK.**
Step 5	• In the **Solution Explorer,** right-click **BooksServer,** and then click **Properties**.
Step 6	• Go to **Application** tab. Use the drop-down menu to select **.NET Framework 4** or **.NET Framework 4.5** in the **Target framework.** Click **Yes** to reload the project.

Create On-Premises Server 2-5

Step 7	• Navigate to the **Solution Explorer,** right-click the project name to open the shortcut menu for the project.
Step 8	• Click **Install NuGet** if NuGet is not already present by default.
Step 9	• Click **Manage NuGet Packages** and then click **Online** in the NuGet dialog box.
Step 10	• Click **SearchResults**, enter **WindowsAzure,** and select the **Windows Azure Service Bus** item. Click **Install** and close the dialog box. The client assemblies have been referenced.
Step 11	• To add a new class, go to **Solution Explorer**, right-click the **BooksServer** project, click **Add**, and click **Class**.
Step 12	• Go to **Name** box, enter the name **BooksContract.cs**, and click **Add**.

Create On-Premises Server 3-5

Step 13

- Go to **BooksContract.cs** to replace the namespace definition with the help of following code.

```
namespace BooksServer {
[DataContract]
public class BookData{
     [DataMember]
     public string BookCode { get; set; }
     [DataMember]
     public string Title { get; set; }
     [DataMember]
     public string Author { get; set; }
     }
     [ServiceContract]
     interface IBooks{
     [OperationContract]
     IList<BookData> GetBooks();
     }
     interface IBooksChannel : IBooks, IClientChannel{}
}
```

Create On-Premises Server 4-5

Step 14

- In the file, **Program.cs**, edit the namespace definition and add the following code to add the profile service and host for it.

```
namespace BooksServer{
  class BooksService : IBooks {
    BookData[] products = new []{
       new BookData{ BookCode = "1", Title = "Under the Rock", Author =
       "Tim Simmons"}, new BookData{ BookCode = "2", Title = "Paper
       Scissors", Author = "Yuan Lee"}, new BookData{ BookCode = "3", Title
       = "Scientific Dreams", Author = "William Kingsley"},new BookData{
       BookCode = "4", Title= "Wellness Demystified", Author = "Heather
       Robin"}, };
       public IList<BookData> GetBooks() {
            Console.WriteLine("GetBooks has been called.");
            return products;
        }
  }
  class Program{
  static void Main(string[] args){
  var shost = new ServiceHost(typeof(BooksService));
  shost.Open();
  Console.WriteLine("Press ENTER to close");
  Console.ReadLine();
  shost.Close();} } }
```

Create On-Premises Server 5-5

Step 15

- In the **Solution Explorer**, double-click the **App.config** file to open it in the **Visual Studio** editor.

Step 16

- Replace the contents under the element **<system.ServiceModel>** with the code. Replace `<yourIssuerSecretKey>` with the key that was retrieved from the Azure Management Portal.

Step 17

- Then, replace the `appSettings` element with the following code.

```
<appSettings>
<!-- Service Bus specific app settings for messaging connections -->
<add key="Microsoft.ServiceBus.ConnectionString"
value="Endpoint=sb://[sampleservicebusdemo].servicebus.windows.
net;Shared AccessKeyName=RootManageSharedAccessKey;SharedAccessKey=[
yourIssuerSecretKey]" />
</appSettings>
```

Step 18

- Build the application.

Create an Application that Uses Server 1-6

- Steps to create an application that will make use of this server are:

Step 1

- Create an ASP.NET MVC application named **BooksPortal** that will consume this service.

Step 2

- Open the `_Layout.cshtml` present under **Shared** folder of **Views** in the Solution Explorer. Delete the following lines from the `_Layout.cshtml`.

```
 <ul>
<li>@Html.ActionLink("Home", "Index", "Home")</li>
<li>@Html.ActionLink("About", "About", "Home")</li>
<li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>
```

Create an Application that Uses Server 2-6

Step 3

- Add a new Model class named `Book.cs`. Add the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
namespace BooksPortal.Models
{
     public class Book
     {
          public string BookCode { get; set; }
          public string Title { get; set; }
          public string Author { get; set; }
     }
}
```

Create an Application that Uses Server 3-6

Step 4

- Open the `HomeController.cs` file and add the code.

```
using BooksPortal.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace BooksPortal.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult About()
        {
            ViewBag.Message = "Books page.";
            return View();
    }
```

Create an Application that Uses Server 4-6

```
    public ActionResult Contact()
    {
        ViewBag.Message = "Your contact page.";
        return View();
    }
    public ActionResult Index(string Identifier, string BookTitle)
    {
        var books = new List<Book> { new Book { BookCode =
        Identifier, Title = BookTitle } };
        return View(books);
    }
  }
}
```

Step 5

- Replace the default text 'My ASP.NET MVC Application' with 'Delaware Books' wherever applicable.

Create an Application that Uses Server 5-6

Step 6

- Open the `Index.cshtml` file and replace its contents with the code.

```
@model IEnumerable<BooksPortal.Models.Book>
@{ ViewBag.Title = "Index";}
<h2>Prod Inventory</h2>
<table>
<tr> <th>
@Html.DisplayNameFor(model => model.Title)
</th>
<th></th>
<th>
@Html.DisplayNameFor(model => model.Author)
</th> </tr>
@foreach (var item in Model) {
<tr> <td>
@Html.DisplayFor(modelItem => item.Title)
</td> <td>
@Html.DisplayFor(modelItem => item.Author)
</td> </tr>
}
</table>
```

Create an Application that Uses Server 6-6

Step 7

- Click **Build→Build Solution**.

Step 8

- Execute the application locally to view the output.

Linking On-Premises Server with the Application 1-7

- To link the on-premises server, **BooksServer**, with the ASP.NET MVC application using the following steps:

Step 1	Open the **BooksPortal** project and in the Solution Explorer, right-click and select **ManageNuGetPackages**.
Step 2	Search for `WindowsAzure.ServiceBus`, select the **Windows Azure Service Bus** item, and click **Install**.
Step 3	Right-click **BooksPortal** in the Solution Explorer and click **Add** → **Existing Item**.
Step 4	Browse to the `BooksContract.cs` file from the **BooksServer** console project and add it as a link by clicking the down arrow next to **Add**.

Linking On-Premises Server with the Application 2-7

Step 5

In the `HomeController.cs` file, add the following code. Replace your `IssuerSecret` with the actual key. This enables the client to call the on-premises service and display the output.

```
using System.Linq;
using System.ServiceModel;
using System.Web.Mvc;
using Microsoft.ServiceBus;
using Models;
using BooksServer;

namespace BooksPortal.Controllers {
public class HomeController : Controller {
// Declare the channel factory
static ChannelFactory<IBooksChannel> channelFactory;
static HomeController() {
// Create shared secret token credentials for authentication
channelFactory = new ChannelFactory<IBooksChannel>(new
NetTcpRelayBinding(), "sb://
sampleservicebusdemo.servicebus.windows.net/products");
    channelFactory.Endpoint.Behaviors.Add(new
    TransportClientEndpointBehavior {
```

Linking On-Premises Server with the Application 3-7

```
            TokenProvider = TokenProvider.CreateSharedSecretTokenProvider (
            "owner", "yourIssuerSecret") });
        }
        public ActionResult Index(){
          using (IBooksChannel channel = channelFactory.CreateChannel()){
                    // Return a view of the products inventory
                    return this.View(from bks in channel.GetBooks()
                    select new Book { BookCode = bks.BookCode,
                    Title = bks.Title,Author = bks.Author});
          }
        }
    }
}
```

Step 6

Right-click **BooksPortal** solution in the Solution Explorer and then, click **Add → Existing Project**.

Step 7

Browse to the **BooksServer** project and add the **BooksServer.csproj** solution file.

Linking On-Premises Server with the Application 4-7

Step 8	Open the **Properties** window for the **BooksPortal** solution, click **Startup Project** on the left, and then select **Multiple startup projects**.
Step 9	Ensure that **BooksServer** is the first in the list.

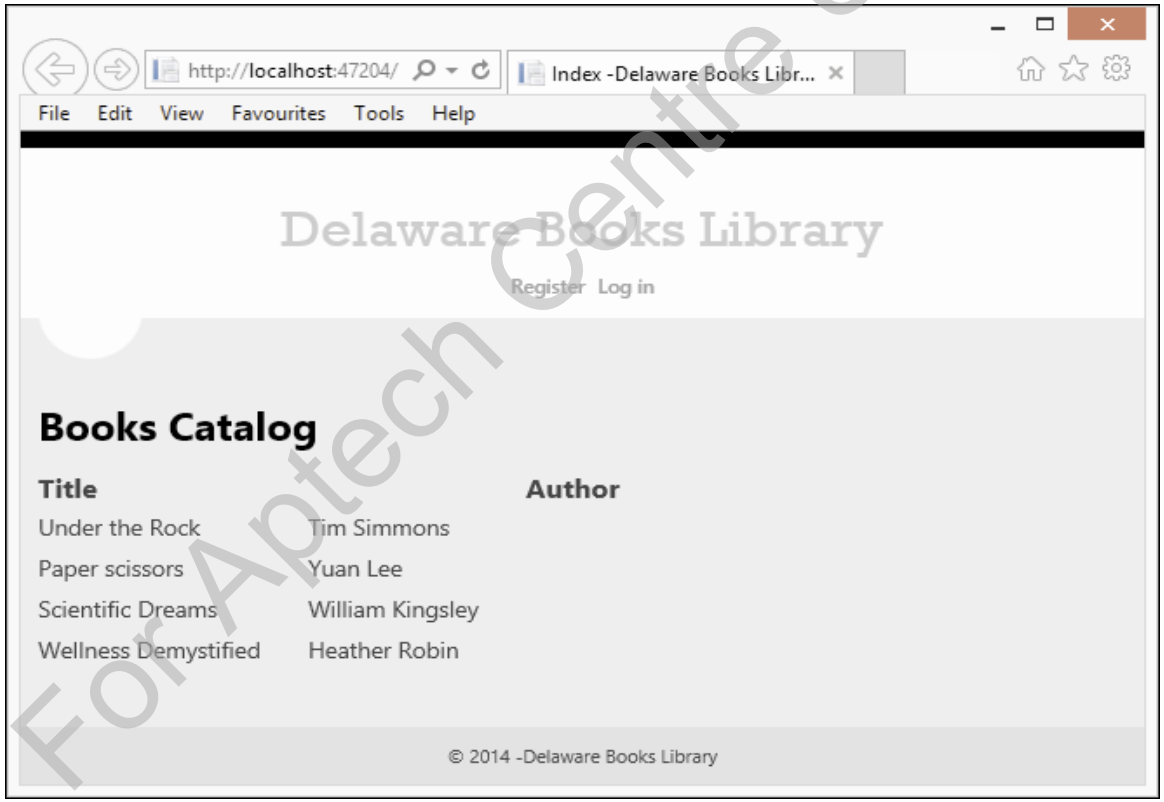
- Press **F5** to build and run the application.
- The **BooksServer** console application, which acts as on-premises server should start first.
- Then the **BooksPortal** application should start in a browser window.

Linking On-Premises Server with the Application 5-7

- This time, you will see that the books catalog lists data retrieved from the **BooksServer** present on-premises.
- Following figure displays the **BooksServer** running in the command window:

Linking On-Premises Server with the Application 6-7

- Following figure displays the output of the **BooksPortal** Web application:



Linking On-Premises Server with the Application 7-7

- If you check the command window running the server, you will see a message saying `GetBooks` has been called, as shown in the following figure:

Publishing to Windows Azure 1-2

- You have created an on-premises server and an MVC application and linked them to each other.
- The real use of the Service Bus comes into picture when you publish your application to Windows Azure and run the application.
- Following are the steps to do this:

Step 1

- Right-click the **BooksPortal** project in **Solution Explorer** and click **Publish**.
- Provide the required credentials for your subscription and follow all the steps in the dialog box.

Step 2

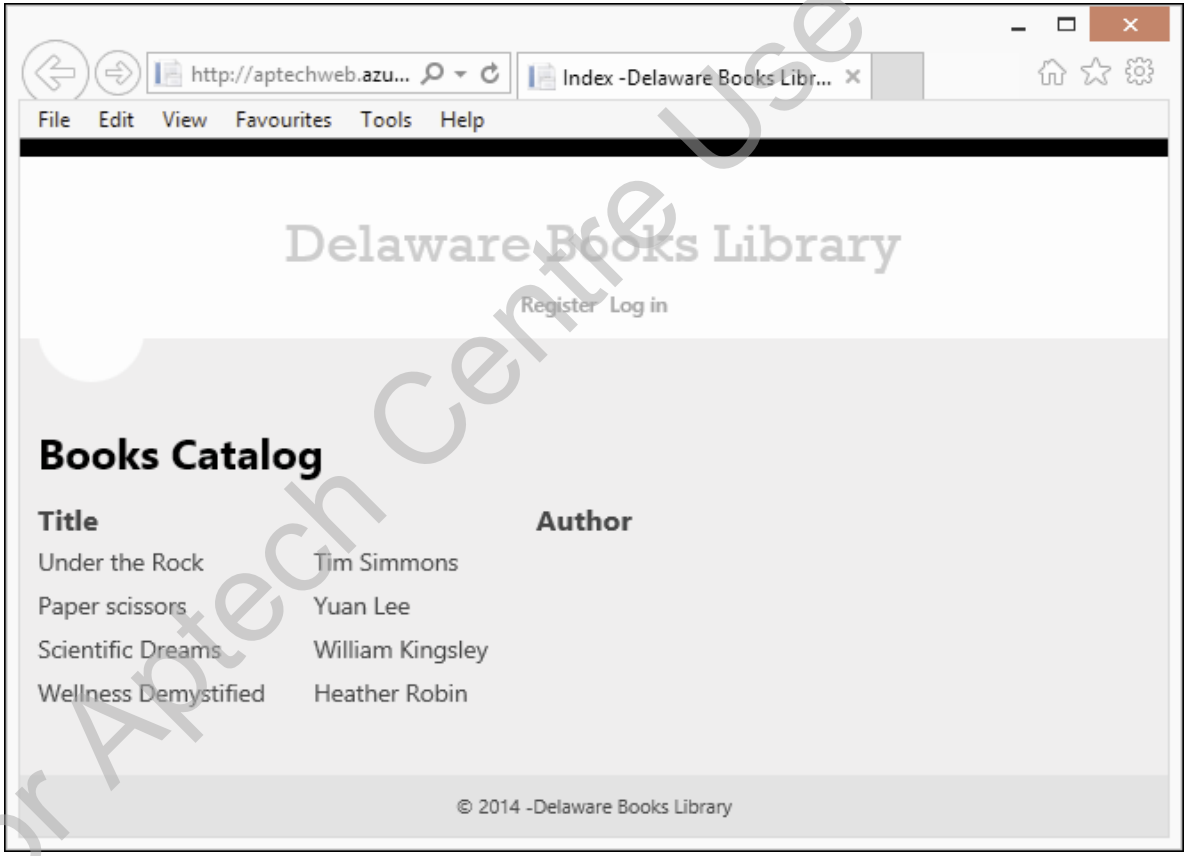
- Run the on-premises server.

Step 3

- Open the Windows Management Portal, click the **Web Site**, which is associated with the **BooksPortal** application, and then click the **BROWSE** option present at the bottom of the portal.

Publishing to Windows Azure 2-2

- The output in the Management Portal is as shown in the following figure:



- The role of the Service Bus in this example is to leverage the existing on-premises server and enable it to be used across the cloud-based Azure infrastructure.

Summary 1-2

- Service Bus is a Windows Azure Cloud Computing initiative that provides a solution for security and scalability issues.
- Queues, topics, and relays are integral to the Service Bus messaging system.
- AMQP is a consistent and well-organized wire-level protocol that helps to build easy messaging applications between different vendor products.
- Azure Notification Hubs have a Push feature that helps the consumers and enterprise applications for mobile platforms to access the infrastructure with ease.

Summary 2-2

- Push Notifications is a feature that notifies users about an event that has occurred.
- Service Bus Queues work on the brokered messaging communication model and acts as a mediator.
- Service Bus Topics work on a publish/subscribe messaging communication model and acts as a mediator.
- Service Bus Relays help to build cloud hybrid applications that run in an Azure datacenter and on-premises environment.