Application Innovation through Cloud & Data. Real-Time personalized experiences at global scale

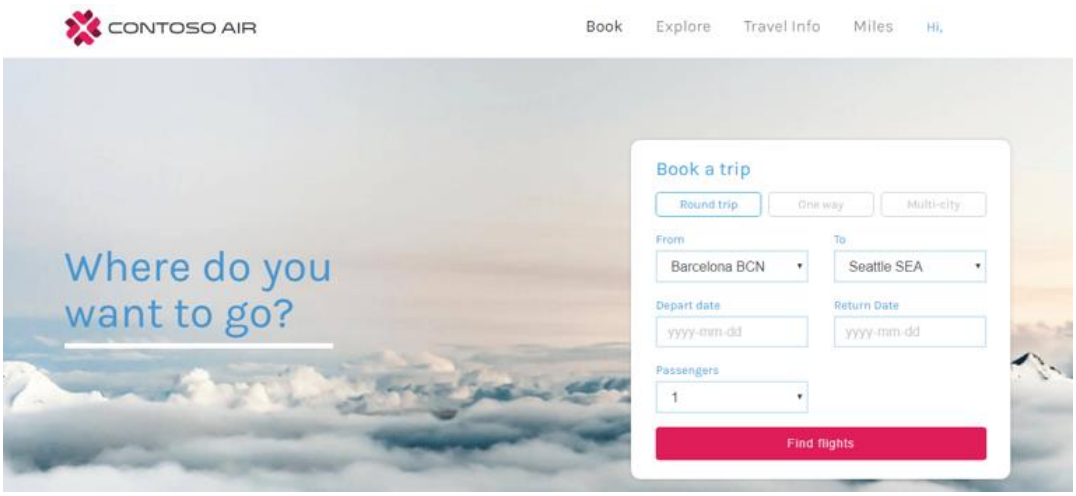
To remain competitive in today's customer-oriented marketplace, organizations must possess the ability to scale applications globally, and provide the intelligence and personalization their customers have become accustomed to in applications. Unfortunately, building globally distributed, intelligent database applications has traditionally been a tedious and time-consuming exercise for development teams.

With the introduction of **Azure Cosmos DB**, however, building intelligent and responsive planet-scale apps is now much easier. Built from the ground up with global distribution and horizontal scale at its core, Azure Cosmos DB offers turnkey global distribution across any number of Azure regions by transparently scaling and replicating data wherever users are. Plus, it offers the ability to elastically scale throughput and storage worldwide.

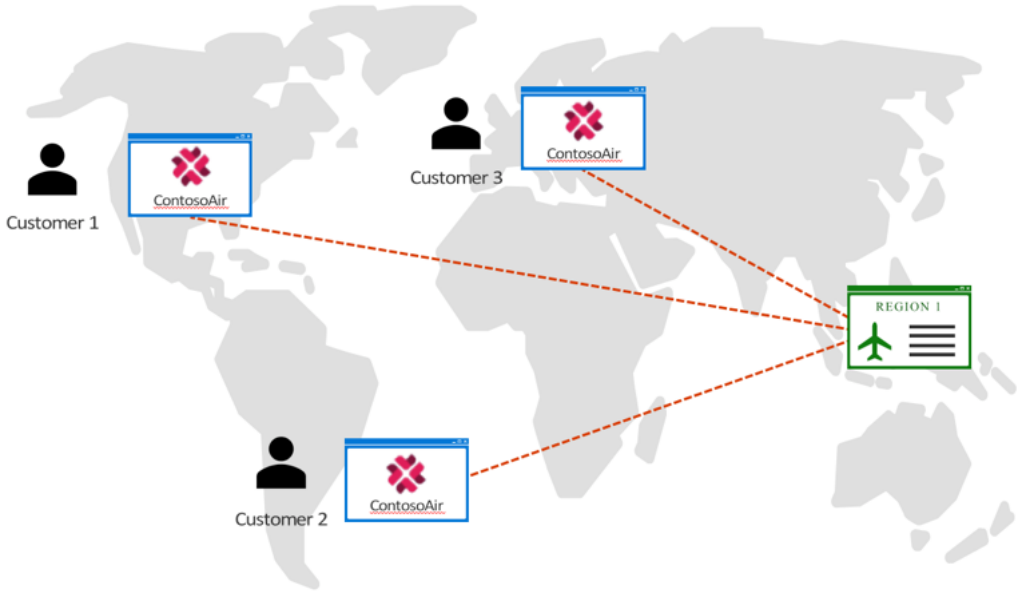
In the ContosoAir scenario below, we are going to take a developer-centric look at how Azure Cosmos DB, coupled with Microsoft Cognitive Services and Azure Functions, helped ContosoAir build a richer, real-time personalized experience for customers, and a more intelligent and responsive globally distributed serverless application.

The ContosoAir sample customer scenario

ContosoAir is a global airline, and a growing player in the international airline industry. They recently kicked off a project to take their ContosoAir application to the next level, with a stated goal of "delighting customers around the world with a more intelligent and responsive app."



Their starting point "ContosoAir application" consists of a .NET app written in C# which is deployed to an Azure App Service. They are using MongoDB as their back-end database. Within Azure, they have centralized all resources within a single region, and are using a single instance of the web app and database.



To ensure their customers are getting the best possible experience regardless of where they are in the world, the ContosoAir development team began seeking technological solutions that can elevate ContosoAir to an intelligent and responsive planet-scale application, offering rich personalization for their customers. They set out to achieve the following goals:

1. Performance around the world with a serverless architecture
2. Greater customer choice and real-time notifications

3. Intelligent predictions based on complex data
4. Improved customer awareness through event-driven scenarios
5. Responsive customer service through intuitive interactions

Performance around the world with a serverless architecture

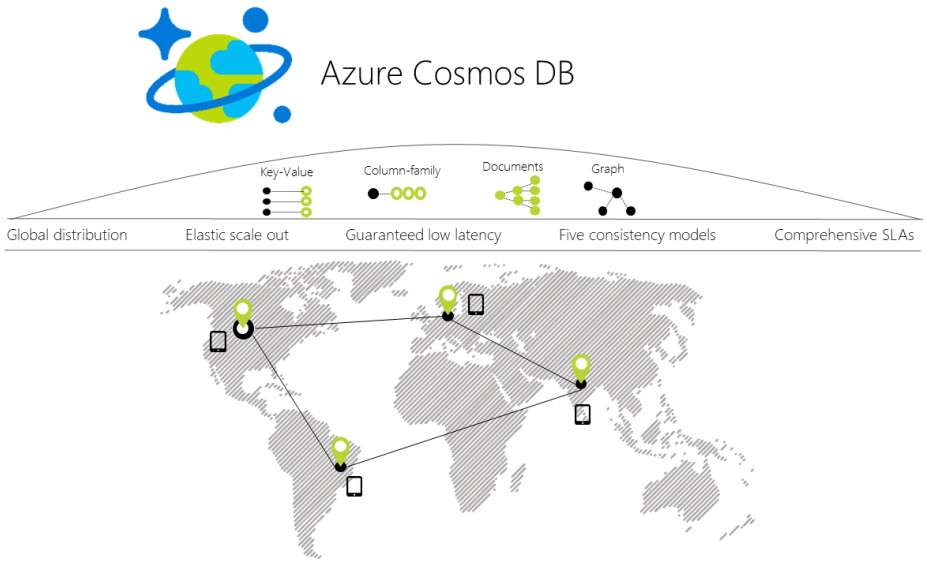
In the ultra-competitive international airline industry, ContosoAir is always looking for ways to differentiate themselves from the competition, and provide added value to their customers. As an airline, they are in the business of transporting people all over the world, and they want to guarantee their customers receive the same great service and consistency of experience from wherever they are, while also simplifying the administrative and DevOps workload for their development team.

Keeping a consistent experience across geos

For ContosoAir, application performance is a top priority. Achieving reliable performance with their centrally managed app, however, has proven difficult. Centrally managed apps accessing data stored in a centralized location don't always perform consistently, ultimately resulting in frustrated users when responsiveness and reliability become issues.

Global distribution improves availability and performance around the world

To achieve their goal of providing consistent and reliable plant-wide performance they need their data close to their customers, the ContosoAir development team decided to migrate their MongoDB database over to **Azure Cosmos DB**. Azure Cosmos DB is Microsoft's globally distributed multi-model database service, offering throughput, latency, availability, and consistency guarantees with comprehensive service level agreements (SLAs), something no other database service can offer. Cosmos DB offers the global scalability they desired, and provides a simple model for transitioning their existing MongoDB database and application code.



Since Azure Cosmos DB databases can be used as the data store for apps written for MongoDB, the ContosoAir developers will be able to continue using familiar tools and skills. Their existing drivers allow their MongoDB app to communicate with Cosmos DB, and use Cosmos DB databases instead of MongoDB databases.

For ContosoAir, switching from using MongoDB to Cosmos DB was achieved by changing the values assigned to a few variables. To learn more, see an Introduction to Azure Cosmos DB: API for MongoDB.

Let's look a the code used to initialize the MongoDB Client in the current ContosoAir data access layer code, contained in `Dal.cs`.

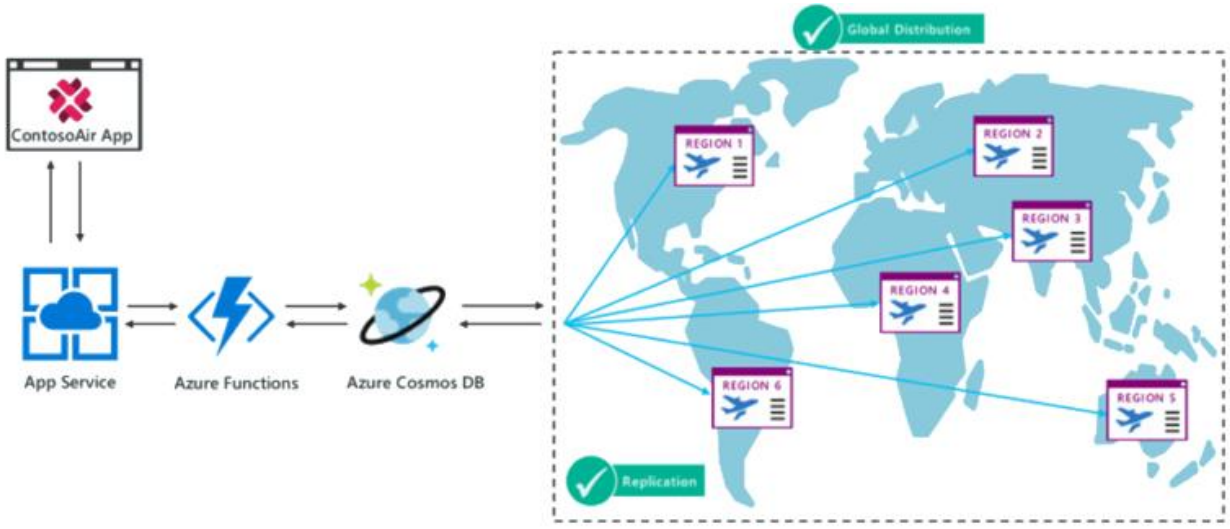
```
MongoClientSettings settings = new MongoClientSettings
{
    Server = new MongoServerAddress(host, 10255),
    UseSsl = true,
    SslSettings = new SslSettings
    {
        EnabledSslProtocols = SslProtocols.Tls12
    }
};

MongoIdentity identity = new MongoInternalIdentity(databaseName, userName);
MongoIdentityEvidence evidence = new PasswordEvidence(password);

settings.Credentials = new List<MongoCredential>()
{
    new MongoCredential("SCRAM-SHA-1", identity, evidence)
};

MongoClient client = new MongoClient(settings);
```

To switch this class over to connect to their new Cosmos DB instance, the team simply went back to the Azure portal to get their connection string information found on their Cosmos DB blade under Connection String, on the Read-write Keys tab.

1. Next, they copied the username value from the portal (using the copy button), and assigned it to the value of the `userName` variable being referenced in the `Dal.cs` file.
2. Then, they copied the host value from the portal, and made it the value of the `host` variable referenced in the `Dal.cs`file.
3. Finally, they copied the password value from the portal, and made it the value of the `password` variable being used in the `Dal.cs` file.

That's it! In just a few simple steps, the development team updated their app with all the info it needs to communicate with Azure Cosmos DB.

With their app now switched over to use Azure Cosmos DB, the ContosoAir development team selected 6 regions around the globe in which to locate their Cosmos DB and app instances. This places their data closer to their users throughout the world, ensuring the lowest possible latency for their customers.



Customers enjoy HA and performance anywhere with Azure Cosmos

For the ContosoAir team, ensuring their customers are able to access their application at any time, from anywhere, is critical. Azure Cosmos DB transparently replicates their data to all regions associated with their MongoDB account. Plus, Cosmos DB provides

transparent regional failover with multi-homing APIs, and the ability to elastically scale throughput and storage across the globe. Learn more in Distribute data globally.

With Azure Cosmos DB's turnkey global distribution, the ContosoAir developers did not have to build their own replication scaffolding. Using Azure Cosmos DB's multi-homing APIs, the app always knows where the nearest region is, and sends requests to the nearest data center. All of this was possible with no config changes. They set the write-region for their database, and assigned read-regions, and the rest was handled for them. There is no need to redeploy apps during regional failover.

To take advantage of global distribution, client applications can specify the ordered preference list of regions to be used to perform document operations. This can be done by setting the connection policy. Based on the Azure Cosmos DB account configuration, current regional availability and the preference list specified, the most optimal endpoint will be chosen to perform write and read operations. Let's look at an example of how ContosoAir configured their application deployed to the West US region to take advantage of multi-region availability, and be resilient in the face of regional failovers.

Before writing code to interact with the Azure Cosmos DB DocumentDB, the team installed the Azure Cosmos DB DocumentDB API Client Library using the Nuget package, `Microsoft.Azure.DocumentDB`.

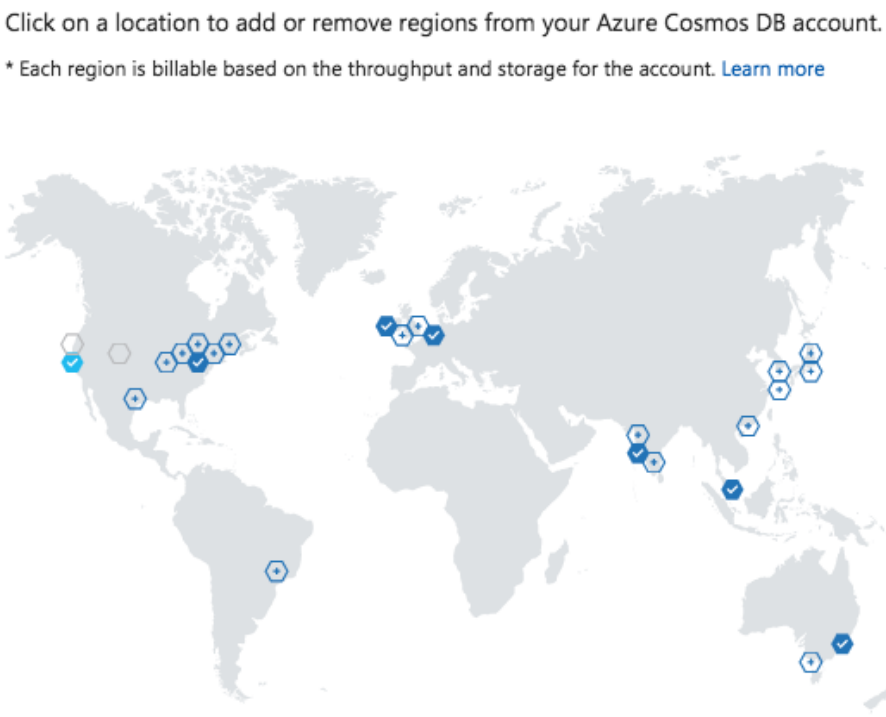
```
// Getting endpoints from application settings or other configuration location
Uri accountEndPoint = new Uri(Properties.Settings.Default.GlobalDatabaseUri);
string accountKey = Properties.Settings.Default.GlobalDatabaseKey;

ConnectionPolicy westUsConnectionPolicy = new ConnectionPolicy
{
    ConnectionMode = ConnectionMode.Direct,
    ConnectionProtocol = Protocol.Tcp
};

westUsConnectionPolicy.PreferredLocations.Add(LocationNames.WestUS); // first
preference
westUsConnectionPolicy.PreferredLocations.Add(LocationNames.EastUS); // second
preference
westUsConnectionPolicy.PreferredLocations.Add(LocationNames.NorthEurope); // third
preference

DocumentClient usClient = new DocumentClient(
    accountEndPoint,
    accountKey,
    westUsConnectionPolicy);
```

Each deployed app instance will contain a unique `ConnectionPolicy`, enabling each regional deployoment to define its failover regions. In the example above, the application is deployed to the West US region. The application deployed to the East US region, for example, would have the order of preferred regions modified. That is, East US

region is specified first for low-latency reads. Then, the West US region is specified as the second preferred region for high availability during region failures. For more details, read Automatic regional failover for business continuity in Azure Cosmos DB.
In the rare event of an Azure regional outage or data center outage, Cosmos DB automatically triggers failovers of all Cosmos DB accounts with a presence in the affected region, using the code above to determine which database the application should use.

Regions can easily scale with customer demand through global database replication

Azure Cosmos DB is a globally distributed, multi-model database service designed to help you achieve fast, predictable performance. It scales seamlessly along with your application as it grows. New regions can be added with the click of a button in the Azure portal, enabling ContosoAir to quickly add additional regions to respond to customer demand.



Azure Functions help streamline development in a serverless architecture

On of the goals the team set for the updated app was to leverage the power and flexibility provided by using a serverless architecture. Serverless computing is all about the ability to focus on individual pieces of logic that are repeatable and stateless. These

pieces require no infrastructure management and they consume resources only for the seconds, or milliseconds, they run for. At the core of the serverless computing movement are functions, which are made available in the Azure ecosystem by **Azure Functions**.

Leveraging the native integration between Azure Cosmos DB and Azure Functions, the ContosoAir developers have come up with several ways in which they can streamline development initiatives using a serverless architecture, directly from their Cosmos DB account, as we will see in the following sections.

Greater customer choice and real-time notifications

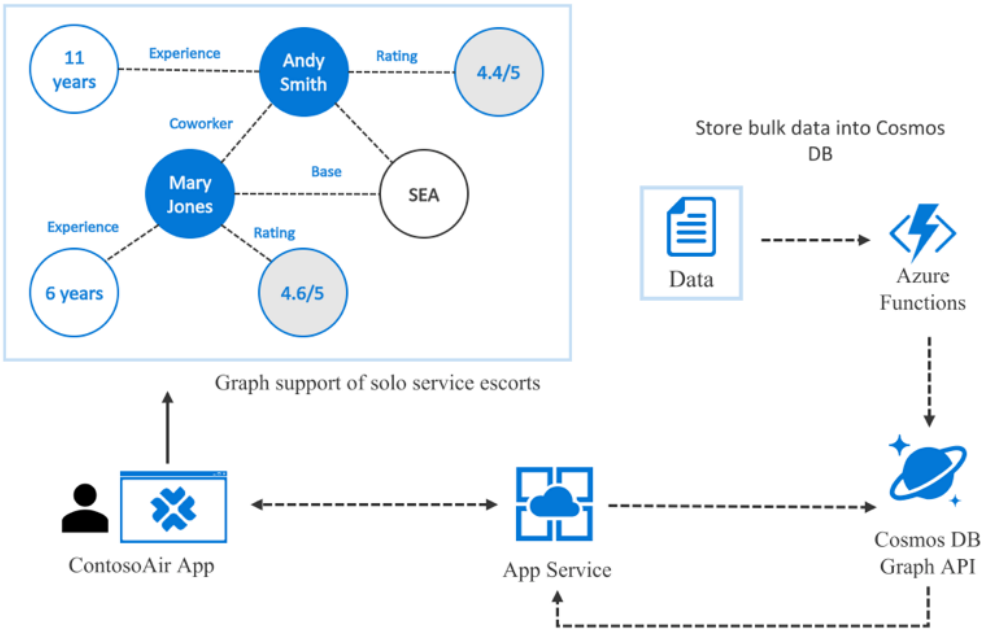
Entrusting your children into the care of strangers is never an easy thing, so it's no surprise that sending an unaccompanied minor on a flight can be a stressful and complex experience for a guardian. ContosoAir wanted to to lessen the anxiety around this by providing a better experience, enabling customers to select airline escorts and providing real-time status updates to guardians.

Opportunities of integrating different sources of data

Currently, the ContosoAir app does not integrate well with all their back-end systems, for instance, on the use case of 'minors travelling alone' they want to increase the transparency of the process, they want parents having total control and transparency about who is escorting their children through the journey.

Graph support shows codeshare flights for improved customer options

To enable the presentation of escort data, the development team decided to leverage the Graph API available in Azure Cosmos DB. Azure Cosmos DB provides the Graph API for applications that need to model, query, and traverse large graphs efficiently using the Gremlin standard. To learn more click here.



To accomplish this goal, the team created an Azure Function in Visual Studio, which contains an HTTP trigger, using an Azure Comsos DB input binding, to retrieve escort data from the escort graph database.

```
public static class EscortSearch
{
    static string endpoint = ConfigurationManager.AppSettings["Endpoint"];
    static string authKey = ConfigurationManager.AppSettings["AuthKey"];

    [FunctionName("EscortSearch")]
    public static async Task<HttpResponseMessage> Run(
        [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route =
null)]HttpRequestMessage req,
        TraceWriter log)
    {
        log.Info("C# HTTP trigger function processed a request for escort info.");

        // the escort objects are free-form in structure
        List<dynamic> results = new List<dynamic>();

        // open the client's connection
        using (DocumentClient client = new DocumentClient(
            new Uri(endpoint),
            authKey,
            new ConnectionPolicy
            {
                ConnectionMode = ConnectionMode.Direct,
                ConnectionProtocol = Protocol.Tcp
            }))
        {
            // get a reference to the database the console app created
            Database database = await client.CreateDatabaseIfNotExistsAsync(
```

```
                new Database
                {
                    Id = "graphdb"
                });

            // get an instance of the database's graph
            DocumentCollection graph = await
client.CreateDocumentCollectionIfNotExistsAsync(
                UriFactory.CreateDatabaseUri("graphdb"),
                new DocumentCollection { Id = "graphcollz" },
                new RequestOptions { OfferThroughput = 1000 });

            // build a gremlin query based on the existence of a airport parameter
            string airport = req.GetQueryNameValuePairs()
                .FirstOrDefault(q => string.Compare(q.Key, "airport", true) == 0)
                .Value;

            IDocumentQuery<dynamic> query = (!String.IsNullOrEmpty(airport))
                ? client.CreateGremlinQuery<dynamic>(graph,
string.Format("g.V('{0}')", flight))
                : client.CreateGremlinQuery<dynamic>(graph, "g.V()");

            // iterate over all the results and add them to the list
            while (query.HasMoreResults)
                foreach (dynamic result in await query.ExecuteNextAsync())
                    results.Add(result);
        }

        // return the list with an OK response
        return req.CreateResponse<List<dynamic>>(HttpStatusCode.OK, results);
    }
}
```

From the above code, a list of available escorts is returned within the app, displaying years of experience, customer ratings, and other information to help parents select the airline employee they feel is the best fit for escorting their unaccompanied child.

Consistent and frequent traveler status updates

Communication of flight status updates, from gate changes to the arrival of an unaccompanied minor at their destination, is an important feature of the ContosoAir app. With their notifications being sent from a centralized database and app, they have seen slow and inconsistent notifications, which can both inconvenience and frustrate customers.

Real-time notifications improve customer awareness and satisfaction

To better meet their goal of providing real-time notifications to their customers, the ContosoAir development team has opted to use **Notification Hub**. Using output binding, functions can send native or template push notifications to target platforms. In the code below, the ContosoAir team is sending an Apple Push Notification Service (APNS) native notification with C# queue triggers to inform guardians of the arrival of their unaccompanied minor at their destination.

```
#r "Microsoft.Azure.NotificationHubs"
#r "Newtonsoft.Json"

using System;
using Microsoft.Azure.NotificationHubs;
using Newtonsoft.Json;

public static async Task Run(string myQueueItem, IAsyncCollector<Notification>
notification, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");
    log.Info($"Sending APNS notification of an unaccompanied minor arrival");

    // The JSON format for a native APNS notification is ...
    // { "aps": { "alert": "notification message" }}
    dynamic passenger = JsonConvert.DeserializeObject(myQueueItem);
    string apnsNotificationPayload = "{\"aps\": {\"alert\": \"Your unaccompanied
minor, " + passenger.name + ", has arrived at their destination, and was greeted by
their escort, " + passenger.destinationEscort + "\" }}";
    log.Info($"{apnsNotificationPayload}");
    await notification.AddAsync(new AppleNotification(apnsNotificationPayload));
}
```

By combining the Graph API with Azure Notification Hub, the team has been able to greatly improve the capabilities the app offers for guardians of unaccompanied minors. They can now select their child's escorts, and will receive real-time notifications throughout the flight, keeping them informed as their child travels.

Intelligent predictions based on complex data

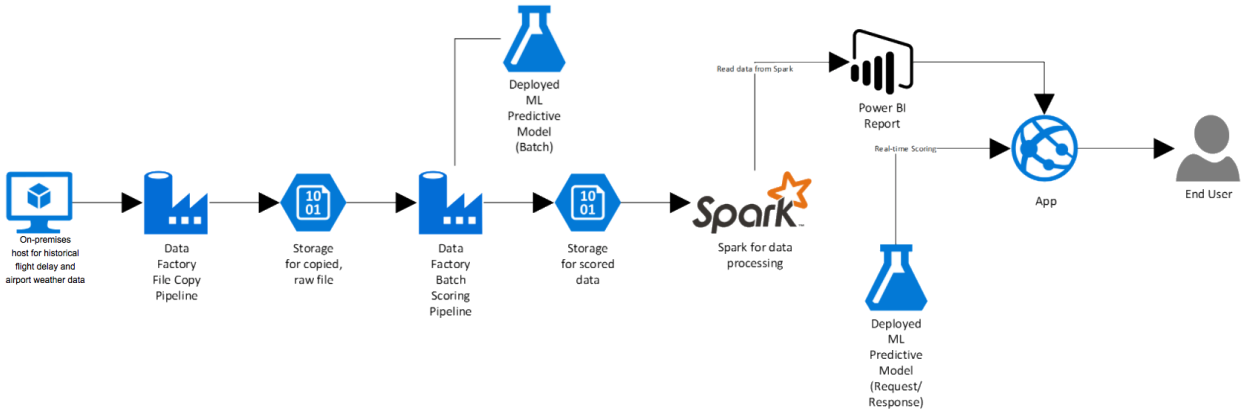
In their quest to differentiate themselves from competitors, ContosoAir has become interested in applying data science to the problem of discovering if weather forecasts combined with their historical flight delay data could be used to provide a meaningful input into the customer's decision-making process at the time of booking by communicating potential flight delays in real-time. ContosoAir has over 30 years of historical flight delay data stored as comma-separated value (CSV) files on an on-premises file server.

Predict flight delays from diverse data sources

Flight delays and cancellations result from numerous factors, and are difficult to predict. ContosoAir, like many organizations, does not have experienced data scientists on staff, and as such, they need cloud-based solutions which can make analyzing complex data sets from diverse data sources more accessible for developers.

Predictive analytics provide customers a complete view of flight itineraries

While there are many factors that they use to tailor guidance to customers (such as cost and travel duration), ContosoAir believes an innovative solution might come in the form of giving the customer an assessment of the risk of encountering flight delays. This is information they would like to provide at the time of booking to their customers, so they can make more informed decisions about which flights and connections to choose.



Below, we'll take a closer look at their solution, and the various Azure components they used.

Azure Data Factory and Integration Runtime to copy data

To handle moving their on-premises historical flight data into the Azure storage, ContosoAir wanted a solution which could be set up as a scheduled, automated process, capable of being run on a monthly basis. For this, ContosoAir selected **Azure Data Factory (ADF)**. ADF is a cloud-based data integration service that allows you to create data-driven workflows in the cloud that orchestrate and automate data movement and data transformation. For more information about ADF, see an Introduction to Azure Data Factory.

Using ADF, the development team created the pipeline that copies their historical flight delay data from their on-premises server into a container in Azure storage. As input for the copy activity, the team created a FileShare dataset. This dataset references an Azure Data Factory **Integration Runtime** installation on their on-premises file server. Integration Runtime is the compute infrastructure used by ADF to provide data movement across different network environments, and acts as the gateway for moving the data from on-premises to Azure storage.

```
{
    "name": "OnPremInputDataset",
    "properties": {
        "published": false,
        "type": "FileShare",
        "linkedServiceName": "OnPremFileServer",
        "typeProperties": {
            "folderPath": "FlightsAndWeather/",
            "format": {
                "type": "TextFormat",
                "columnDelimiter": ",",
                "firstRowAsHeader": true
            }
        },
        "availability": {
            "frequency": "Month",
            "interval": 1
        },
        "external": true,
        "policy": {}
    }
}
```

Next, the team created an output dataset, which defines the location and folder structure for where the copied files will be stored in the Azure storage account. In this case, they chose to write the data to the default storage container for their Apache Spark cluster, since they will be using Spark to handle some of the data processing.

```
{
    "name": "BlobStorageOutputDataset",
    "properties": {
        "published": false,
        "type": "AzureBlob",
        "linkedServiceName": "BlobStorageOutput",
        "typeProperties": {
            "fileName": "FlightsAndWeather.csv",
            "folderPath": "sparkcontainer/FlightsAndWeather/{Year}/{Month}/",
            "format": {
                "type": "TextFormat",
                "columnDelimiter": ",",
                "firstRowAsHeader": true
            },
            "partitionedBy": [
```

```
                {
                    "name": "Year",
                    "value": {
                        "type": "DateTime",
                        "date": "SliceStart",
                        "format": "yyyy"
                    }
                },
                {
                    "name": "Month",
                    "value": {
                        "type": "DateTime",
                        "date": "SliceStart",
                        "format": "MM"
                    }
                }
            ]
        },
        "availability": {
            "frequency": "Month",
            "interval": 1
        },
        "external": false,
        "policy": {}
    }
}
```

With the datasets in place, the team next created the pipeline, which is configured to run once a month, and copy files from the previous month to blob storage from their on-premises file server. For any previous months, which would include all the historical data, the pipeline will run with a concurrency of 10, allowing 10 months to be processed simultaneously.

```
{
    "name": "CopyOnPrem2AzurePipeline",
    "properties": {
        "description": "This pipeline copies timesliced CSV files from an on-premises
location to Azure Blob Storage as a continuous job.",
        "activities": [
            {
                "type": "Copy",
                "typeProperties": {
                    "source": {
                        "type": "FileSystemSource",
                        "recursive": true
                    },
                    "sink": {
                        "type": "BlobSink",
                        "copyBehavior": "MergeFiles",
                        "writeBatchSize": 0,
                        "writeBatchTimeout": "00:00:00"
                    }
                },
```

```
                "inputs": [
                    {
                        "name": "OnPremInputDataset"
                    }
                ],
                "outputs": [
                    {
                        "name": "BlobStorageOutputDataset"
                    }
                ],
                "policy": {
                    "timeout": "1.00:00:00",
                    "concurrency": 10,
                    "style": "StartOfInterval",
                    "retry": 3,
                    "longRetry": 0,
                    "longRetryInterval": "00:00:00"
                },
                "scheduler": {
                    "frequency": "Month",
                    "interval": 1
                },
                "name": "Copy on-premises files to Azure blob storage"
            }
        ],
        "start": "1987-01-01T00:00:00Z",
        "end": "2099-12-31T23:59:00Z",
        "isPaused": false,
        "pipelineMode": "Scheduled"
    }
}
```

With an ADF copy pipeline in place, and their data copied into Azure storage, the team is now ready to build an Azure Machine Leaning experiment for predicting the likelihood of delays based on the weather forecast.

Azure Machine Learning Experiment

Having heard about using **Azure Machine Learning (ML)** to gain insights into data, the ContosoAir team came up with the idea of using predictive analytics to help customers best select their travels based on the likelihood of a delay.

After creating a new Machine Learning Studio workspace in Azure, the team created a Machine learning model using Azure ML Studio.

To improve the quality of their predictions, the team needed to perform some cleanup on the data, also known as munging. By adding an Execute Python Script module their

ML model, they were able to handle removing rows containing missing data, as well as paring down the columns in the dataset, and transforming column values to ensure the data is consistent, and in a format that is easily consumable by the model.

```
import pandas as pd
import math

def azureml_main(dataframe1 = None, dataframe2 = None):

    # Round Time down to the nex hour
    # Add the rounded value to a new column named "Hour"
    dataframe1["Hour"] = dataframe1["Time"].apply(roundDown)

    # Replace any missing HourlyPrecip and WindSpeed values with 0.0
    dataframe1["HourlyPrecip"] = dataframe1["HourlyPrecip"].fillna('0.0')
    dataframe1["WindSpeed"] = dataframe1["WindSpeed"].fillna('0.0')

    # Replace any WindSpeed values of "M" (missing) with 0.005
    dataframe1["WindSpeed"] = dataframe1['WindSpeed'].replace(['M'], '0.005')

    # Replace any SeaLevelPressure values of "M" (missing) with 29.92 (the average
pressure)
    dataframe1["SeaLevelPressure"] = dataframe1['SeaLevelPressure'].replace(['M'],
'29.92')

    # Replace any HourlyPrecip values of "T" (trace) with 0.005
    dataframe1["HourlyPrecip"] = dataframe1['HourlyPrecip'].replace(['T'], '0.005')

    # Convert our WindSpeed, SeaLevelPressure, and HourlyPrecip columns to numeric
    dataframe1[['WindSpeed','SeaLevelPressure', 'HourlyPrecip']] =
dataframe1[['WindSpeed','SeaLevelPressure', 'HourlyPrecip']].apply(pd.to_numeric)

    # Pare down the variables in the Weather dataset to just the columns being used
by the model
    df_result = dataframe1[['AirportCode', 'Month', 'Day', 'Hour', 'WindSpeed',
'SeaLevelPressure', 'HourlyPrecip']]

    # Return value must be of a sequence of pandas.DataFrame
    return df_result

def roundDown(x):
    z = int(math.floor(x/100.0))
    return z
```

Using the historical flight and weather data, the team trained their model, and published it as a Predictive Web Service through Azure ML Studio. This provides an endpoint for the trained model to be operationalized, allowing the ContosoAir team to use another ADF pipeline to add prediction scores to their historical flight data.

Azure Data Factory (ADF) Pipeline to score data using the ML model

To apply their model predictions to the historical flight and weather data, the team created another ADF Pipeline, which adds prediction scores to the data using an Azure ML Linked Service. The Azure ML Linked Service enables ADF to connect to the Predictive Web Service's batch URL endpoint, so the historical data can be scored against the ML model. This pipeline, like the copy pipeline created previously, will process all the historical data, as well as be run on a once a month schedule, as new data arrives from USDOT.

The pipeline uses the output from the previous pipeline as input, and is similar to the one created previously, so we will only look at the output dataset, which writes the scored flight and weather data to blob storage.

```
{
    "name": "ScoredBlobOutput",
    "properties": {
        "published": false,
        "type": "AzureBlob",
        "linkedServiceName": "BlobStorageOutput",
        "typeProperties": {
            "fileName": "Scored_FlightsAndWeather{Year}{Month}.csv",
            "folderPath": "sparkcontainer/ScoredFlightsAndWeather",
            "format": {
                "type": "TextFormat"
            },
            "partitionedBy": [
                {
                    "name": "Year",
                    "value": {
                        "type": "DateTime",
                        "date": "SliceStart",
                        "format": "yyyy"
                    }
                },
                {
                    "name": "Month",
                    "value": {
                        "type": "DateTime",
                        "date": "SliceStart",
                        "format": "MM"
                    }
                }
            ]
        },
        "availability": {
            "frequency": "Month",
            "interval": 1
        }
    }
}
```

With the pipeline in place to add scoring information to the historical flight data, and the resulting files written to storage, the ContosoAir development team is ready to move on to using **Apache Spark on HDInsight** to summarize the data.

Azure Cosmos DB and Machine Learning with Apache Spark uncovers risk of flight delays

Apache Spark on HDInsight is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

The team wants to process the data by using Apache Spark on **Azure HDInsight**. The Spark to Azure Cosmos DB connector enables Azure Cosmos DB to act as an input source or output sink for Apache Spark jobs, and in this case, the team will be using Cosmos DB as an output sink, writing the data from their ML model to Cosmos DB.

Before executing any code, the team had to include the Azure Cosmoos DB Spark JAR files. To get started, the team downloaded the Spark to Azure Cosmos DB connector (preview) from the azure-cosmosdb-spark repository on GitHub. Once downloaded, they uploaded the `azure-cosmosdb-spark` JAR files to their cluster attached storage. Then, using the Azure HDInsight Jupyter notebook service, they used the spark magic command to include the JAR files.

```
%%configure
{ "jars": ["wasb:///example/jars/azure-documentdb-
1.10.0.jar","wasb:///example/jars/azure-cosmosdb-spark-0.0.3.jar"],
  "conf": {
    "spark.jars.excludes": "org.scala-lang:scala-reflect"
   }
}
```

After configuring the Spark connector, the developers were ready to open a connection to their Cosmos DB.

```
// Import Spark to Cosmos DB Connector
import com.microsoft.azure.cosmosdb.spark.schema._
import com.microsoft.azure.cosmosdb.spark._
import com.microsoft.azure.cosmosdb.spark.config.Config

// Connect to DocumentDB Database
val config = Config(Map("Endpoint" -> "https://contosoair.documents.azure.com:443/",
"Masterkey" ->
"xWpfqUBioucC2YkWV6uHVhgZtsPIjIVmE4VDPyNYnw2QUazvCHm3rnn9AeSgglLOT3yfjCR5YbLeh5MCc3aK
Nw==",
"Database" -> "contosoAir",
"preferredRegions" -> "West US;East US;",
```

```
"Collection" -> "flightDelays",
"SamplingRatio" -> "1.0"))
```

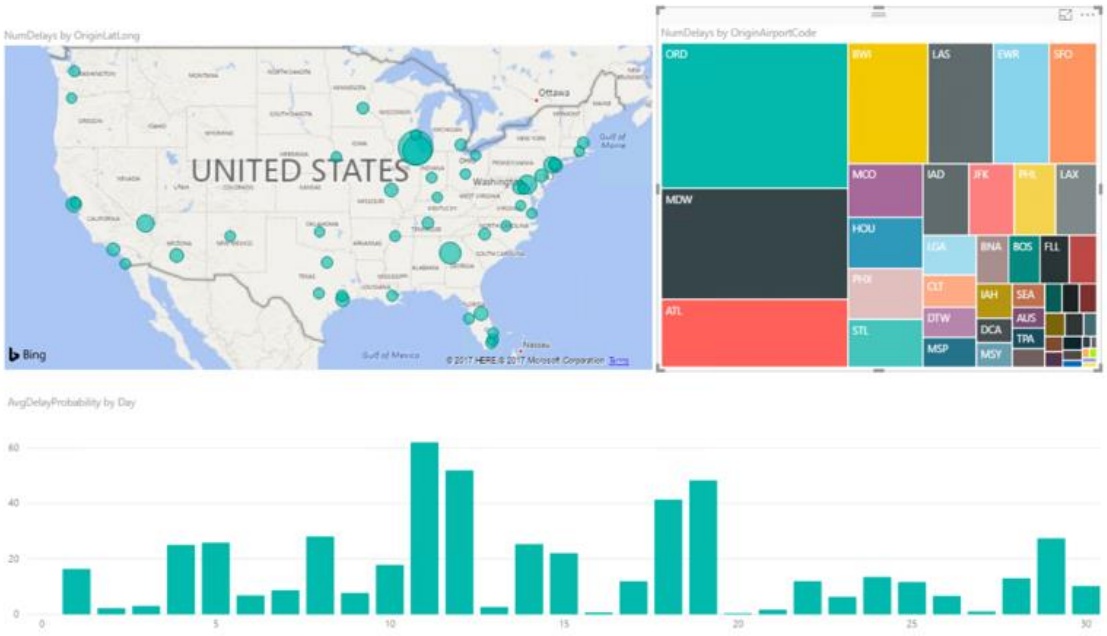
Using the connector as a sink, they next retrieved the scored flight and weather data from blob storage, and wrote the data from those files to Cosmos DB.

```
import spark.sqlContext.implicits._

val flightDelayTextLines = sc.textFile("/ScoredFlightsAndWeather/*.csv")

case class
AirportFlightDelays(OriginAirportCode:String,OriginLatLong:String,Month:Integer,Day:I
nteger,Hour:Integer,Carrier:String,DelayPredicted:Integer,DelayProbability:Double)

val flightDelayRowsWithoutHeader = flightDelayTextLines.map(s =>
s.split(",")).filter(line => line(0) != "OriginAirportCode")

val resultDataFrame = flightDelayRowsWithoutHeader.map(
    s => AirportFlightDelays(
        s(0), //Airport code
        s(13) + "," + s(14), //Lat,Long
        s(1).toInt, //Month
        s(2).toInt, //Day
        s(3).toInt, //Hour
        s(5), //Carrier
        s(11).toInt, //DelayPredicted
        s(12).toDouble //DelayProbability
        )
).toDF()

resultDataFrame.write.cosmosDB(config)
```

From here, they now have the full capabilities of both Spark and Cosmos DB, and can perform whatever data analysis and queries are needed to provide information to their customers, and for internal reporting needs. To learn more about using the Azure Cosmos DB Spark Connector, click here.

Embedded Power BI visualizations illustrate the risk of delays to customers in real-time

Power BI Embedded is intended to simplify how ISVs and developers use Power BI capabilities. Power BI Embedded simplifies Power BI capabilities by helping teams quickly add stunning visuals, reports, and dashboards into apps. By enabling easy-to-navigate data exploration, ContosoAir is able to provide intuitive displays of flight delay data for users during the booking process, ensuring they have the access to delay predictions when making the choice of which flights to take.



To learn more about Power BI Embedded, click here.

Improved customer awareness through event-driven scenarios

Airport gate assignments are occasionally changed with little notice, leading to customer confusion and frustration. ContosoAir needs to quickly learn and communicate unexpected information from diverse data sources to maintain customer satisfaction.

Last-minute, real-time notifications when gate assignments suddenly change

Difficult to quickly learn and communicate gate changes to the right customer. This difficulty in communication has led to inadequate awareness for their customers, which in turn leads to panicked customers and missed flights.

Customers learn gate changes and other key notifications instantly

As they did previously, the team decided to use Notification Hub and Azure Functions to ensure their customers receive gate change updates in real-time. In this case, the queue

item is a gate change message. Through this function, terminal gate information updates in Cosmos DB are quickly retrieved from Cosmos DB and customers receive real-time updates with events triggering Azure Functions code.

```
#r "Microsoft.Azure.NotificationHubs"
#r "Newtonsoft.Json"

using System;
using Microsoft.Azure.NotificationHubs;
using Newtonsoft.Json;

public static async Task Run(string myQueueItem, IAsyncCollector<Notification>
notification, TraceWriter log)
{
    log.Info($"C# Queue trigger function processed: {myQueueItem}");
    log.Info($"Sending APNS notification of a new user");

    dynamic departure = JsonConvert.DeserializeObject(myQueueItem);
    string apnsNotificationPayload = "{\"aps\": {\"alert\": \"Your flight's departure
gate has been changed to " + departure.gate + "\" }}";
    log.Info($"{apnsNotificationPayload}");
    await notification.AddAsync(new AppleNotification(apnsNotificationPayload));
}
```

To learn more, go to Azure Cosmos DB bindings for Functions.

Responsive customer service through intuitive interactions

Providing self-service functionality in apps can greatly improve the customer experience, and alleviate some of the heavy call volume experienced by many call centers. Through the use of automated services and intelligent feedback, ContosoAir seeks to provide these capabilities to their customers.

Broaden the customer service features directly available on app

The current ContosoAir app has very limited customer service functionality available directly on the app. For most things, the only option open to customers is to call a customer service representative, which can involve long wait times, and lead to frustrated customers.

Intuitive and speedy customer service thanks to intelligent bots

To provide faster service, the ContosoAir team is implementing a bot, through the Azure **Bot Framework**. The use of Bot service automation unburdens customer service resources, and can provide faster flight scheduling, reservation updates, and related services to their customers. The Bot Builder SDK for .NET provides an easy-to-use framework for developing bots using Visual Studio and Windows. The SDK leverages C# to provide a familiar way for .NET developers to create powerful bots.

To start, the team downloaded and installed the required Bot project templates (Bot Application, Bot Controller, and Bot Dialog) by copying the .zip files to the Visual Studio 2017 project templates directory.

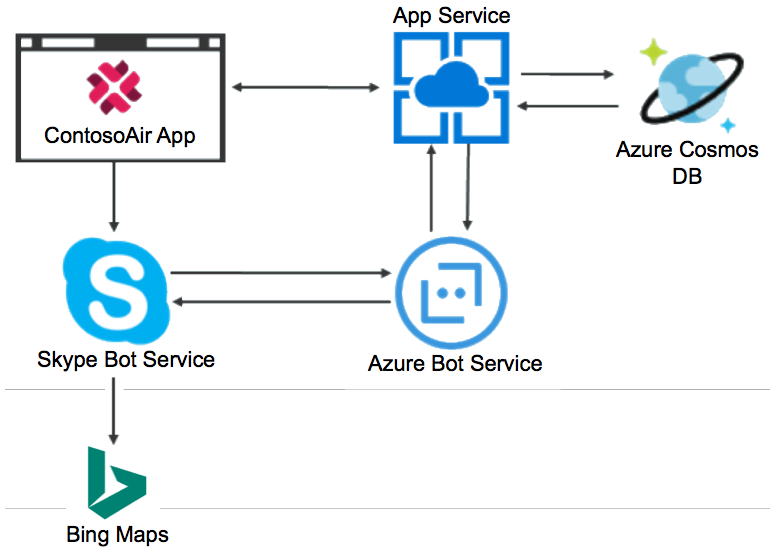
The next, created a new Bot Application project in Visual Studio. Within their project, the `MessageController` receives a message from a user containing a flight number, and invokes the root dialog, which processes the message and generates a response containing the current flight status.

```
[BotAuthentication]
public class MessagesController : ApiController
{
    /// <summary>
    /// POST: api/Messages
    /// Receive a message from a user and reply to it
    /// </summary>
    public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
    {
        if (activity.Type == ActivityTypes.Message)
        {
            await Conversation.SendAsync(activity, () => new Dialogs.RootDialog());
        }
        else
        {
            HandleSystemMessage(activity);
        }
        var response = Request.CreateResponse(HttpStatusCode.OK);
        return response;
    }
    ...
}
```

The `MessageReceivedAsync` method within `RootDialog.cs` sends a reply that contains the status for the flight number sent by the user.

```
[Serializable]
public class RootDialog : IDialog<object>
{
    public Task StartAsync(IDialogContext context)
    {
        context.Wait(MessageReceivedAsync);
        return Task.CompletedTask;
    }
```

```
    private async Task MessageReceivedAsync(IDialogContext context,
IAwaitable<object> result)
    {
        var activity = await result as Activity;

        // get flight status
        var flightStatus = GetFlightStatus(activity.Text);

        // return the flight status reply to the user
        await context.PostAsync($"Flight {activity.Text} has a current status of
{flightStatus}");

        context.Wait(MessageReceivedAsync);
    }
}
```



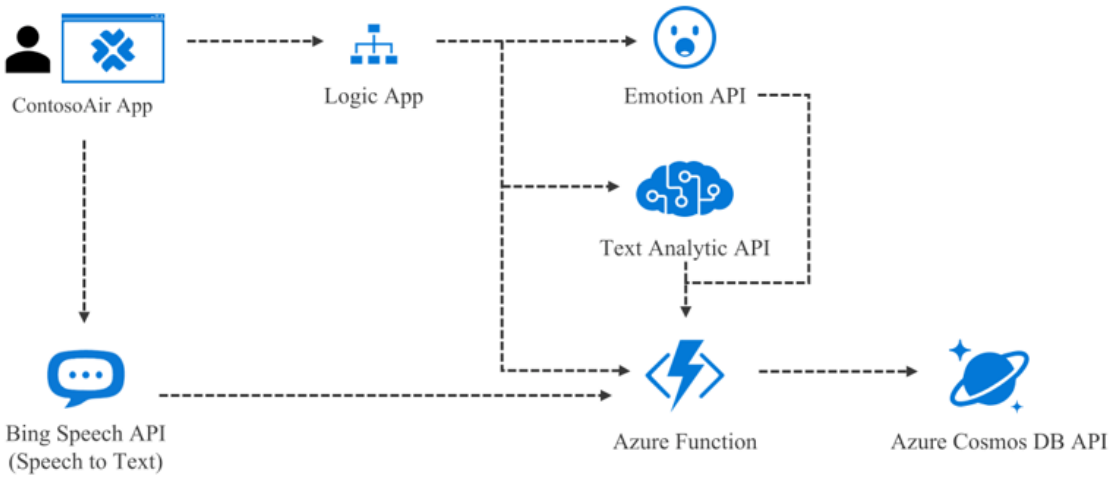
With the framework in place, the team can now build out much more functionality through the use of Bots, including allowing customers to select seats for their upcoming flights, and purchase tickets.

Understand and leverage customer sentiment

ContosoAir, like many organizations, suffers from an inability to transform complex customer feedback into actionable information. With their current system, the information they receive provides a delayed and incomplete understanding of customer sentiment. This leads to a hindrance in customer service when their customers need it most.

Transform sentiment into real-time insights with Cognitive Services

In order to gain better insight into customer sentiment, the development team at ContosoAir has decided to leverage the power of **Microsoft Cognitive Services**. Microsoft Cognitive Services offer a wide range of capabilities for analyzing customer sentiment, and transforming that information into actionable data. Specifically, the team has decided to implement text and speech analysis, so they can better capture data associated with spoken and written feedback.



Text Analytics API codifies written customer feedback into ratings

Leveraging the functionality provided by the **Text Analytics API**, the ContosoAir developers are looking to codify written customer feedback into numeric ratings, which can be used to provide trend analysis reports to management, marketing, and customer service. Text Analytics API is a cloud-based service that provides advanced natural language processing over raw text, and includes three main functions:

1. **Sentiment Analysis**: Find out what customers think of your brand or topic by analyzing raw text for clues about positive or negative sentiment.
2. **Key Phrase Extraction**: Automatically extract key phrases to quickly identify the main points.
3. **Language Detection**: For up to 120 languages, detect which language the input text is written in, and report a single language code for every document submitted on the request.

The team decided to start by using feedback messages received via their app. By sending these messages to the Text Analytics API, they get numeric scores representing the customer's sentiment. The ratings numbers can then incorporated into **Power BI** reports showing trends in customer sentiment. For more information on getting started with incorporating Text Analytics API into applications, click here.

To being using the Text Analytics API in their application, the team first installed the Cognitive Services Language SDK, using the NuGet package, `Microsoft.Azure.CognitiveServices.Language`.

NOTE: While it is possible to call the HTTP endpoints directly from C#, the `Microsoft.Azure.CognitiveServices.Language`SDK makes it much easier to call the service without having to worry about serializing and deserializing JSON.

After setting up Text Analytics API in the Azure portal, and retrieving their Subscription Key, the developers are ready to start coding. First, they created a class for storing the details of the scored results received from the Text Analytics API. Note the overridden `ToString()` method. This enables to object to be serialized into JSON, so the `ScoredFeedback` objects can easily be written to Cosmos DB DocumentDB.

```
using Newtonsoft.Json;
using System.Collections.Generic;

namespace ContosoAir
{
    public class ScoredFeedback
    {
        public ScoredFeedback()
        {
            KeyPhrases = new List<string>();
        }

        [JsonProperty(PropertyName = "id")]
        public string Id { get; set; }
        public string Language { get; set; }
        public string Text { get; set; }
        public List<string> KeyPhrases { get; set; }
        public double? Score { get; set; }

        public override string ToString()
        {
            return JsonConvert.SerializeObject(this);
        }
    }
}
```

The next class the team created was for passing data to the Text Analytics API. The Text Analytics API limits the number of calls per minute to 100, so the team decided to send feedback messages in batches, so the results returned are aggregated into a single JSON document. That JSON document is then parsed into individual `ScoredFeedback` objects. They also decided to handle identifying the language of messages within this class using Language Detection, as well as the extraction of key phrases, to be used for reporting and future data analysis.

```
using Microsoft.Azure.CognitiveServices.Language.TextAnalytics;
using Microsoft.Azure.CognitiveServices.Language.TextAnalytics.Models;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Threading.Tasks;

namespace ContosoAir
{
    public class TextAnalyzer
    {
        // TextAnalyticsAPI client.
        private readonly ITextAnalyticsAPI client;

        public TextAnalyzer()
        {
            client = new TextAnalyticsAPI
            {
                AzureRegion = AzureRegions.Eastus2,  // Region in which the Text
Analytics API was created.
                SubscriptionKey = "69098074b8bf4bbd9c9656466c34aacd" // Text
Analytics API Subscription Key
            };
        }

        public async Task<List<ScoredFeedback>>
AnalyzeCustomerFeedback(List<CustomerMessage> messages)
        {
            var languageBatchResult = await AnalyzeLanguage(messages);

            // Build MultiLanguageInput items to add to our batch sentiment analysis
input
            var multiLanguageInput = messages
                .Select(m => new MultiLanguageInput(
                    languageBatchResult.Documents.FirstOrDefault(d => d.Id ==
m.Id).DetectedLanguages[0].Iso6391Name, m.Id,
                    m.Message))
                    .ToList();

            var batchInput = new MultiLanguageBatchInput(multiLanguageInput);
            var sentimentResult = await AssignSentimentScore(batchInput);
            var keyPhrasesResult = await ExtractKeyPhrases(batchInput);

            var scoredFeedback = new List<ScoredFeedback>();

            foreach (var message in messages)
            {
                scoredFeedback.Add(new ScoredFeedback
                {
                    Id = message.Id,
                    Language = languageBatchResult.Documents.FirstOrDefault(d => d.Id
== message.Id).DetectedLanguages[0].Iso6391Name,
                    Text = message.Message,
                    KeyPhrases = keyPhrasesResult.Documents.FirstOrDefault(d => d.Id
== message.Id).KeyPhrases.ToList(),
                    Score = sentimentResult.Documents.FirstOrDefault(d => d.Id ==
message.Id).Score
                });
            }
```

```
            return scoredFeedback;
        }

        private async Task<SentimentBatchResult>
AssignSentimentScore(MultiLanguageBatchInput input)
        {
            return await client.SentimentAsync(input);
        }

        private async Task<LanguageBatchResult> AnalyzeLanguage(List<CustomerMessage>
messages)
        {
            var languageInput = messages.Select(m => new Input(m.Id,
m.Message)).ToList();
            return await client.DetectLanguageAsync(new BatchInput(languageInput));
        }

        private async Task<KeyPhraseBatchResult>
ExtractKeyPhrases(MultiLanguageBatchInput input)
        {
            return await client.KeyPhrasesAsync(input);
        }
    }
}
```

With the code in place to handle analyzing the customer feedback messages, the team wanted to next insert the resulting `ScoredFeedback` objects as documents into Cosmos DB. For this, they created a `CosmosWriter` class with the `CreatedScoredFeedbackDocuments` method, which handles updating and creating documents in a specified database and collection.

```
using Microsoft.Azure.Documents;
using Microsoft.Azure.Documents.Client;
using System;
using System.Collections.Generic;
using System.Net;
using System.Threading.Tasks;

namespace ContosoAir
{
    public static class CosmosWriter
    {
        private const string EndpointUrl =
"https://contosoair.documents.azure.com:443/";
        private const string PrimaryKey =
"61KXcKeLDqTSj9Y3lf7Zsve9OJZiudWHEE2XsXQqKHUtBurGwuWaxtctnoIy5ih88qRV7ru3UMOX5OWSwPyM
ag==";
        private static DocumentClient client;

        public static async Task CreateScoredFeedbackDocuments(string databaseName,
string collectionName, List<ScoredFeedback> scoredFeedback)
        {
            client = new DocumentClient(new Uri(EndpointUrl), PrimaryKey);
```

```
            await client.CreateDatabaseIfNotExistsAsync(new Database { Id =
databaseName });
            await client.CreateDocumentCollectionIfNotExistsAsync(UriFactory
                .CreateDatabaseUri(databaseName), new DocumentCollection { Id =
collectionName });

            foreach (var feedback in scoredFeedback)
            {
                // Replace the document, if exists. Otherwise, create the document.
                try
                {
                    await client.ReplaceDocumentAsync(UriFactory
                        .CreateDocumentUri(databaseName, collectionName,
feedback.Id), feedback);
                }
                catch (DocumentClientException de)
                {
                    if (de.StatusCode == HttpStatusCode.NotFound)
                    {
                        await
client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri(databaseName,
collectionName), feedback);
                    }
                    else
                    {
                        throw;
                    }
                }
            }
        }
    }
}
```

The last thing the team needed to do was instantiate a `TextAnalyzer`, and start passing in customer feedback messages.

```
var textAnalyzer = new TextAnalyzer();
var scoredFeedback = textAnalyzer.AnalyzeCustomerFeedback(messages).Result;

CosmosWriter.CreateScoredFeedbackDocuments(databaseName, collectionName,
scoredFeedback).Wait();
```

Below is an example of the scoredFeedback JSON documents inserted into Cosmos DB DocumentDB.


```
[
  {
    "id": "0",
    "Language": "en",
    "Text": "I had the best flight of my life.",
    "Score": 0.9907612204551697,
    "KeyPhrases": [
      "best flight",
      "life"
    ],
```

```
    "_rid": "drgDALTWhQEBAAAAAAAAAA==",
    "_self": "dbs/drgDAA==/colls/drgDALTWhQE=/docs/drgDALTWhQEBAAAAAAAAAA==/",
    "_etag": "\"23004a35-0000-0000-0000-59f8ec0a0000\"",
    "_attachments": "attachments/",
    "_ts": 1509485578
  },
  {
    "id": "1",
    "Language": "en",
    "Text": "This airline is horrible. My flight was cancelled, and I wasn't
notified.",
    "Score": 0.1395307183265686,
    "KeyPhrases": [
      "airline",
      "flight"
    ],
    "_rid": "drgDALTWhQECAAAAAAAAAA==",
    "_self": "dbs/drgDAA==/colls/drgDALTWhQE=/docs/drgDALTWhQECAAAAAAAAAA==/",
    "_etag": "\"23008635-0000-0000-0000-59f8ec0b0000\"",
    "_attachments": "attachments/",
    "_ts": 1509485579
  },
  {
    "id": "2",
    "Language": "fr",
    "Text": "Vous m'avez rapidement aidé à trouver un nouveau vol lorsque le mien a
été retardé.",
    "Score": 0.515317440032959,
    "KeyPhrases": [
      "nouveau vol"
    ],
    "_rid": "drgDALTWhQEDAAAAAAAAAA==",
    "_self": "dbs/drgDAA==/colls/drgDALTWhQE=/docs/drgDALTWhQEDAAAAAAAAAA==/",
    "_etag": "\"2300c935-0000-0000-0000-59f8ec0b0000\"",
    "_attachments": "attachments/",
    "_ts": 1509485579
  },
  {
    "id": "3",
    "Language": "it",
    "Text": "La notifica di un cambio di cancelli è stata lenta per arrivare,
causando a perdere il mio volo.",
    "Score": 0.7403281927108765,
    "KeyPhrases": [
      "cambio di cancelli",
      "notifica",
      "volo"
    ],
    "_rid": "drgDALTWhQEEAAAAAAAAAA==",
    "_self": "dbs/drgDAA==/colls/drgDALTWhQE=/docs/drgDALTWhQEEAAAAAAAAAA==/",
    "_etag": "\"23000136-0000-0000-0000-59f8ec0b0000\"",
    "_attachments": "attachments/",
    "_ts": 1509485579
  }
]
```

By leveraging the batch classes of the Text Analytics API, the team was able to execute analysis on a scheduled basis, reducing the number of calls being made to the API. With sentiment analysis functionality in place, the team is ready to move on to analyzing speech received from recorded customer services calls.

Bing Speech API converts customer voice data into written feedback

To capture data from recorded customer service calls, the team is going to use another service in the Cognitive Services suite, the **Bing Speech API**. Using the Microsoft Speech client libraries, they are able to transcribe audio streams into text via their C# app. To get started with the Bing Speech API, click here.

To being using the Bing Speech API in their application, the team first installed the Bing Speech Client Library using the Nuget package, `Microsoft.Bing.Speech`.
The developers created two classes to store speech analysis results, `SpeechAnalysisResult` and `RecognizedPhrase`, and also referenced an existing class containing information about customer service calls, `CustomServiceCall`.
The `SpeechAnalysisResult` class contains an overridden `ToString()` method, like the `ScoredFeedback` class above, to facilitate JSON serialization when inserting the objects into Cosmos DB.

```
using Newtonsoft.Json;
using System.Collections.Generic;

namespace ContosoAir
{
    public class SpeechAnalysisResult
    {
        public SpeechAnalysisResult()
        {
            RecognizedPhrases = new List<RecognizedPhrase>();
        }

        [JsonProperty(PropertyName = "id")]
        public string Id { get; set; }
        public CustomerServiceCall Call { get; set; }
        public List<RecognizedPhrase> RecognizedPhrases { get; set; }

        public override string ToString()
        {
            return JsonConvert.SerializeObject(this);
        }
    }
}
```

In this case, the `RecognizedPhrase` class is referenced by the `SpeechAnalysisResult` class, so it did not require an overridden `ToString()` method. The JSON serialization will be handled by the `SpeechAnalysisResult` class.

```
namespace ContosoAir
{
    public class RecognizedPhrase
    {
        public int PhraseNumber { get; set; }
        public string Phrase { get; set; }
        public string Confidence { get; set; }
        public ulong MediaTime { get; set; }
    }
}
```

For reference, below is a snippet of the `CustomerServiceCall` class, showing the fields that are being captured as part of this process.

```
namespace ContosoAir
{
    public class CustomerServiceCall
    {
        public int Id { get; set; }
        public int RepId { get; set; }
        public int CustomerId { get; set; }
        public string Locale { get; set; }  // e.g., en-US, en-GB, fr-FR, fr-CA
        ...
    }
}
```

The next class the developers created is `SpeechAnalyzer`, which wraps the Bing Speech API calls, and returns the results as a `SpeechAnalysisResult`. As phrases (or sentences) are identified by the Bing Speech API, they are returned via the `SubscribeToRecognitionResult` method of the `SpeechClient`, and added to a `SpeechAnalysisResult` object, which is ultimately returned by the `AnalyzeAudioFile` method. To handle authentication against the Bing Speech API, the team decided to use subscription key authentication, and they made use of the `CognitiveServicesAuthorizationProvider` class, downloaded from here.

```
using Microsoft.Bing.Speech;
using System;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

namespace ContosoAir
{
    public class SpeechAnalyzer
    {
        private Preferences preferences;
        private Uri serviceUrl = new
Uri(@"wss://speech.platform.bing.com/api/service/recognition/continuous");
        private string subscriptionKey = "66d632724586479b859517bd397f90a9";

        // Cancellation token used to stop sending the audio.
        private readonly CancellationTokenSource cts = new CancellationTokenSource();

        private SpeechAnalysisResult _speechAnalysisResult;
```

```
        private int _counter = 0;

        public async Task<SpeechAnalysisResult> AnalyzeAudioFile(string audioFile,
CustomerServiceCall call)
        {
            _speechAnalysisResult = new SpeechAnalysisResult
            {
                Id = call.Id.ToString(),
                Call = call
            };

            using (var speechClient = new SpeechClient(preferences))
            {
                preferences = new Preferences(call.Locale, serviceUrl, new
CognitiveServicesAuthorizationProvider(subscriptionKey));

                speechClient.SubscribeToRecognitionResult(OnRecognitionResult);

                using (var audio = new FileStream(audioFile, FileMode.Open,
FileAccess.Read))
                {
                    var deviceMetadata = new DeviceMetadata(DeviceType.Near,
DeviceFamily.Laptop, NetworkType.Wifi, OsName.Windows, "1607", "Dell", "T3600");
                    var applicationMetadata = new ApplicationMetadata("ContosoAir",
"2.0.0");
                    var requestMetadata = new RequestMetadata(Guid.NewGuid(),
deviceMetadata, applicationMetadata, "ContosoAirService");

                    await speechClient.RecognizeAsync(new SpeechInput(audio,
requestMetadata), cts.Token).ConfigureAwait(false);
                }
            }

            return _speechAnalysisResult;
        }

        private Task OnRecognitionResult(RecognitionResult result)
        {
            _counter++;

            foreach (var phrase in result?.Phrases)
            {
                _speechAnalysisResult.RecognizedPhrases.Add(new RecognizedPhrase
                {
                    PhraseNumber = _counter,
                    Phrase = phrase.DisplayText,
                    Confidence = phrase.Confidence.ToString(),
                    MediaTime = phrase.MediaTime
                });
            }

            return Task.FromResult(true);
        }
    }
}
```

To handle writing the speech analysis results into Cosmos DB, an additional method was added to the `CosmosWriter` class created above. The new method, `CreateSpeechAnalysisDocument` creates or updates documents in the specified database and collection.

```
public static async Task CreateSpeechAnalysisDocument(string databaseName, string collectionName, SpeechAnalysisResult speechAnalysisResult)
{
    client = new DocumentClient(new Uri(EndpointUrl), PrimaryKey);
    await client.CreateDatabaseIfNotExistsAsync(new Database { Id = databaseName });
    await client.CreateDocumentCollectionIfNotExistsAsync(UriFactory.CreateDatabaseUri(databaseName), new DocumentCollection { Id = collectionName });

    // Replace the document, if exists. Otherwise, create the document.
    try
    {
        await client.ReplaceDocumentAsync(UriFactory.CreateDocumentUri(databaseName, collectionName, speechAnalysisResult.Id), speechAnalysisResult);
    }
    catch (DocumentClientException de)
    {
        if (de.StatusCode == HttpStatusCode.NotFound)
        {
            await client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri(databaseName, collectionName), speechAnalysisResult);
        }
        else
        {
            throw;
        }
    }
}
```

The last thing the team needed to do was instantiate a `SpeechAnalyzer`, and start passing in customer service audio files.

```
var speechAnalyzer = new SpeechAnalyzer();
var speechAnalysisResult = speechAnalyzer.AnalyzeAudioFile(audioFile, customerServiceCall).Result;

CosmosWriter.CreateSpeechAnalysisDocument(databaseName, collectionName, speechAnalysisResult).Wait();
```

Using the Bing Speech API, the development team is able to convert speech from recorded Customer Service calls into text, which can then be analyzed using the Text Analytics API, and their audio data is now providing actionable data for management and Customer Service.

Emotion API generates ratings based on customer facial expressions

Installing cameras at check-in counters and service desks, enabling airport staff to receive input into detected emotions in real-time. Notifications are displayed on the worker's terminal, helping them to better recognize customer emotions.

TODO: QUESTION: Does this really fit in for this narrative? Camera's can be used to take photos of customers at service desks, but that is pretty limited, and would require real-time analysis of photos to be of any use to CS reps in the field. Otherwise, we would still be dealing with delayed analysis of information.

ContosoAir improved customer satisfaction and performance by building a better app

In the ContosoAir sample customer scenario, we have seen how **Azure CosmosDB** can be used with other Azure platform functionality, such as **Azure Functions** and **Cognitive Services** to build responsive, more intelligent global-scale application, featuring rich user personalization. Using those tools, the ContosoAir development team accomplished the following goals:

- Consistency and performance anywhere
- Making sense of data from disparate sources
- Real-time predictive analytics to forecast what's next
- Instant updates for customer awareness
- Automation and intelligence for intuitive services

Next Steps

ry an Azure Cosmos DB hands-on-lab (no Azure subscription required).

Activate your own free Azure account

Resources

- Azure Cosmos DB
- Azure Cosmos DB Emulator
- Set up Cosmos DB using Graph API
- Text Analytics API Overview
- Bing Speech API

- Emotion API
- Azure Notification Hubs
- An introduction to Azure Functions
- Spark Connector for Cosmos DB
- Bot Framework
- Cognitive Services
- Microsoft Cognitive Services Products
- Azure Machine Learning
- Azure Data Factory