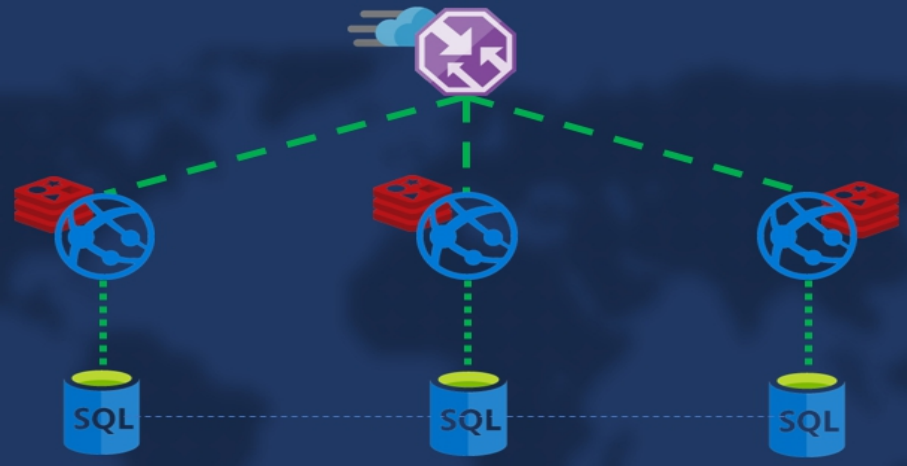
CHRISTOS SAKELLARIOS

Globally-distributed applications with Microsoft Azure



Build robust, highly available, planet-scale web applications using Microsoft Azure Services. Apply DevOps for business continuity to a data geo-replicated complex system

chsakell's blog

Globally-Distributed Applications with Microsoft Azure

For developers and architects

Christos Sakellarios

This book is for sale at
http://leanpub.com/globally-distributed-applications-with-microsoft-azure

This version was published on 2018-02-13

Contents

Preface

Why This Book

Microsoft Azure provides a huge amount of cloud services and is probably the best option for building applications in the cloud. While Microsoft provides documentation for its services, many times it isn't clear how those services can fit together in a single application. I decided to write this book because I believed it will help developers and architects to understand not only how to use Azure Services but also how to combine them to solve complex problems while optimizing performance in a globally-distributed cloud solution. A project has been built for this reason, a .NET Core - Angular application that makes use of the following Azure Services.

1. Azure Storage
2. Azure CDN
3. Azure Cosmos DB
4. Azure Search
5. Redis Cache
6. Service Bus
7. SQL Databases & SQL Active Geo-Replication
8. Azure Active Directory
9. Azure Traffic Manager
10. App Services & WebJobs

As you can see there's a lot of them being used. The purpose of the book is to help you create all these Azure Services, configure and deploy the Online.Store application in multiple regions over the globe. You will also understand how to **group** `Azure Resources`, something that will help you with the **DevOps** automation processes required for releasing software updates with zero downtime.

Versions Of This book

Anyone how buys this book from Leanpub gets new updated versions released instantly. Sometimes you get notified about the new release, sometimes you don't. It depends on the changes released, for example if just typos errors fixed you will not get notified but if a new section or some breaking changes released then you will. I will always try to improve the content of this book either by adding new content or making any type of corrections. Your comments and feedback are more than welcomed to help me improve this book.

About This Book

The book will get you deep into to Azure portal and Services, at least the most important ones that you always need to build scalable planet scale web applications. You will get familiar with them, understand why, when and how to use them properly.

Who Should Read This Book

This book is both for developers and architects.

Developers will profit by learning to code against `Azure Services`. For every Azure Service introduced such as `Redis Cache` or `Service Bus`, the source code contains a relevant library project with generic repositories.

Architects will find this book extremely useful as well. They will learn how to properly design and **group** Azure resources in order to ease and automate release processes while keeping business continuity. There are lots of `PowerShell` scripts written for `DevOps` automation and most of them can easily change and meet your application requirements.

How Is This Book Structured

There is a project associated with the book. The `Online.Store` project is a **cross-platform** application built with ASP.NET Core 2.0 and Angular and is designed as a globally-distributed web application. The settings though *(appsettings.json, secret keys)* are empty and expected to be configured by you.

The first 4 parts of the book intent to help you deploy this web application. You will use Azure Portal to create one by one the required Azure Services. After creating each `Azure Service`, you will fill the related configuration *(endpoints, connection strings, secret keys etc..)* in the respective property setting in the app. Each chapter begins with a small overview section where the main features of the `Azure Service` is described. Next there is an `Azure Portal` section where you actually create the `Azure Service` and configure the relevant part in the application. At the end of these parts, you should have the application deployed at least in two different regions over the world.

The last part of the book is all about **DevOps**. Everything that is repeated should be scripted, and should be scripted well. You will learn how to provision all the services you created in the previous parts by using `PowerShell` scripts. You will also see that resource grouping is a crusial matter in those types of applications and their right design will help you automate releases. You will combine `PowerShell` scripts and Appveyor for **Continuous Integration & Delivery**. At the end of this part, you should be able to run a full release cycle of your software, with or without database changes in a complex globally-distributed environment with zero downtime.

Source Code For The Book

The source code is available at the planet-scale-azure GitHub repository. `Fork` or `Clone` the repository by running the following command.

```
git clone https://github.com/chsakell/planet-scale-azure.git
```

Repository Fork

The `DevOps` part will require from you to fork the repository

System Requirements

In order to follow along with the book, create Azure Services and build the project you need the following:

1. An Azure free account. It's absolutely free, plus you get a $200 credit you can spend during the first 30 days. In a nutchell, to register a free account you need a Microsoft email and a credit cart. Don't worry your credit cart **will not be charged**. You can read more about the Azure free account here
2. Install Microsoft - NET Core
3. Install Node JS. Version **v6.11.2** and later should be good enough
4. Install Git

In case you are in a `Windows` OS it is highly advised to install Visual Studio 2017. The project however can be built and run outside of Visual Studio as well *(using the .NET Core CLI)*. `PowerShell` installation and requirements are described in the `DevOps` section.

Build the project

- If you use `Visual Studio 2017` just open and build the solution. The first time you open it, it will take some time to install the `npm packages` in the `Online.Store` web application project.
- You can also build and run the project using the `.NET Core CLI`
    1. **Install npm packages**: Open a terminal and `cd` to `Online.Store` project where the `packages.json` file exists. Run `npm install` to install the required npm packages
    2. **Restore packages**: `cd` to the root of the solution and run `dotnet restore`
    3. **Build project**: Run `dotnet build`

Conventions & App Settings

Naming conventions

The design and the Azure resources grouping of the `Online.Store` web application is based in a **naming convension** where `Azure resources` are named after the resource group they belong to, plus a dash *(-)* and a prefix based on the resource type.

Resource Group

A container that holds related resources for an Azure solution. A resource group usually includes resources that you want to manage as a group and share the same lifecycle

The `naming convesion` defines how you name your resource groups and the recources they contain. The resource groups in `Online.Store` application are devided in the following **logical** categories.

1. Primary resource group
2. Parent resource group
3. Child resource group

Note that these are only **logical** categories just to help you understand the dependencies between the resource groups. The `naming-convention` starts with a `primary name` and follows the listed rules.

- Primary's resource group name is equal to `<primary name>`. There will be **only one** primary resource group
- A Parent's resource group name is equal to `<primary name>-<region>` where `region` is the Azure region it belongs to *(lowercase with no spaces)*. There will be only one parent resource group **per region**
- A Child's resource group name is equal to `<primary name>-<region>-app`. There will be one child resource group **per region**

- All resources in a resource group have the same name with the resource group they belong to, and optionally a dash *(-)* and a prefix based on the resource type.

The following table lists the prefixes for the all the resource types used in `Online.Store` application.

Resource Type	Dash	Prefix
Traffic Manager Profile	NO	
Storage Account	NO	storage
CDN Profile	YES	cdn
CDN Endpoint	YES	endpoint
Cosmos DB account	YES	cosmosdb
SQL Server	YES	sql
Service Bus	YES	servicebus
Redis Cache	YES	rediscache
Search Service	YES	search
App Service Plan	YES	plan
App Service	NO	

Storage account name

When you create a `Storage account` you will omit the dash "-" symbol because isn't allowed by Azure

Thoughout this book the primary named used in the examples is **planetscalestore**. Any time you see the word `planetscalestore` or `<Primary-Resource-Group>` or `<PrimaryName>` make sure to replace it with your selected `primary name`. The `primary name` must be a **lowercase** case word without spaces or dashes. Any time you see `<Parent-Resource-Group>` replace it with the parent's resource group name for that `region`. The same applies for child resource groups so any time you see `<Child-Resource-Group>` replace it with the respective child's resource group name. The following table shows resource group examples for primary name `planetscalestore`.

Category	Name	Description
Primary	planetscalestore	Primary's resource group name
Parent	planetscalestore-westeurope	Paren't resource group name in `West Europe` region
Child	planetscalestore-westeurope-app	Child's resource group name in `West Europe`
Parent	planetscalestore-westcentralus	Paren't resource group name in `West Central US` region
Child	planetscalestore-westcentralus-app	Child's resource group name in `West Central US`

The following table shows examples for `Resource types` such as `App Service` or `Service Bus`.

Resource Type	Name	Description
Traffic Manager Profile	planetscalestore	Traffic Manager in `planetscalestore` primary resource group
App Service	planetscalestore-westeurope-app	App Service in `planetscalestore-westeurope-app` child resource group
Service Bus	planetscalestore-westeurope-servicebus	Service Bus in `planetscalestore-westeurope` parent resource group
Redis Cache	planetscalestore-westcentralus-rediscache	Redis Cache in `planetscalestore-westcentralus` parent resource group

Primary Name

You must select a **unique** primary name because this name will be used in the endpoints of the `Azure resources` you are going to create and those endpoints **must** be unique. For example the endpoint of your `Traffic Manager` will look like this:

```
http://<primary-name>.trafficmanager.net
```

which means that this URL must be globally available before creating it. A simple suggestion for selecting a `primary name` is use the website's name you wish to use. That name must be unique otherwise you wouldn't be able to use it. Prefixes for some `Azure Services` will help to avoid such types of conficts.

Primary Name rule

Your `primary name` word must be a word without spaces or dashes with maximum 17 characters

Primary Name - planetscalestore

Don't use the word `planetscalestore` for your primary name because apparently it will be taken. This word was used during the creation of the book just to give you some examples of how your resources should be named. Every time you see the word `planetscalestore` either in images or in code examples, you have to use your own unique `primary name` word

Application Settings & Secret Keys

There are two main files you will be configuring in order to run and deploy the `Online.Store` application, the **appsettings.json** and the **secrets.json** files. The first one can be found in the `Online.Store` web application project and contains the basic properties *(endpoints, names etc..)* pointing to your Azure Services. When prompt simply set the property value you asked to, in that file. The **secrets.json** file is a little bit trickier. This file's purpose is to contain the secret keys for your app outside of your solution and hence the version control. Any property existing in the **secrets.json** file will override the same property in the **appsettings.json** file *(if exists)*. Depending on your environment create the **secrets.json** file as follow.

Using Visual Studio 2017

Right click the `Online.Store` web application project *(not the solution)* and select **Manage User Secrets** from the context menu. This gesture creates the **secrets.json** file and adds a new `UserSecretsId` node within a `PropertyGroup` of the `Online.Store.csproj` file

```
<PropertyGroup>
   <TargetFramework>netcoreapp2.0</TargetFramework>
   <TypeScriptCompileBlocked>true</TypeScriptCompileBlocked>
   <TypeScriptToolsVersion>Latest</TypeScriptToolsVersion>
   <IsPackable>false</IsPackable>
   <UserSecretsId>User-Secret-ID</UserSecretsId>
</PropertyGroup>
```

Any time you are asked to add a property value in the **secrets.json** file simply follow the same process to open and edit that file

Using .NET Core CLI

Open a terminal and `cd` to the `Online.Store` web application project

When asked to add a key-value pair in the **secrets.json** file run the following command:

```
dotnet user-secrets set key value
```

The first time you run the command, the `.NET Core CLI` will create the **secrets.json** file for you. For example the following command will add the value `4B99B01C9B3FB1F9475F6F99A7ADQWER` for the property `SearchService:ApiKey`.

```
dotnet user-secrets set "SearchService:ApiKey" "4B99B01C9B3FB1F9475F6F99A7ADQWER"
```

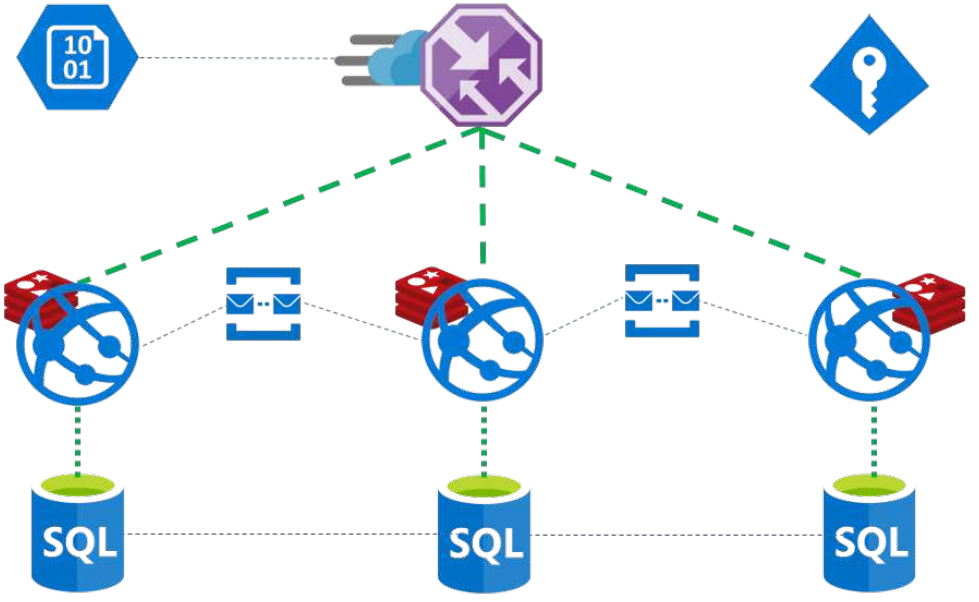
Depending on your OS the **secrets.json** file will be stored in the following locations.

- **Windows**: `%APPDATA%\microsoft\UserSecrets\<userSecretsId>\secrets.json`
- **Linux**: `~/.microsoft/usersecrets/<userSecretsId>/secrets.json`
- **Mac**: `~/.microsoft/usersecrets/<userSecretsId>/secrets.json`

The `userSecretsId` value comes from the value specified in *Online.Store.csproj* file and is equal to `649600d0-c65f-419d-a488-8590c3c9d0ea`. For your reference, there is a *secrets.template.json* file in the `Online.Store/App_Data/templates` folder that shows the properties you will be configuring in the **secrets.json** file throught this book. This file is just a template and nothing more. You can paste its contents in the real generated **secrets.json** file and start configuring that file instead.

Distributing Data Across Regions

In the following sections you are going to create the `Primary Resource Group` and add resources on it. The `primary resource group` contains the `Storage Account`, `Azure CDN` and `Azure Cosmos DB` services for `Online.Store` application and acts as the top level in the resource groups hierarchy. Before proceeding make sure you have created an Azure Free account and signed in to Azure Portal.



Distributing Data Globally

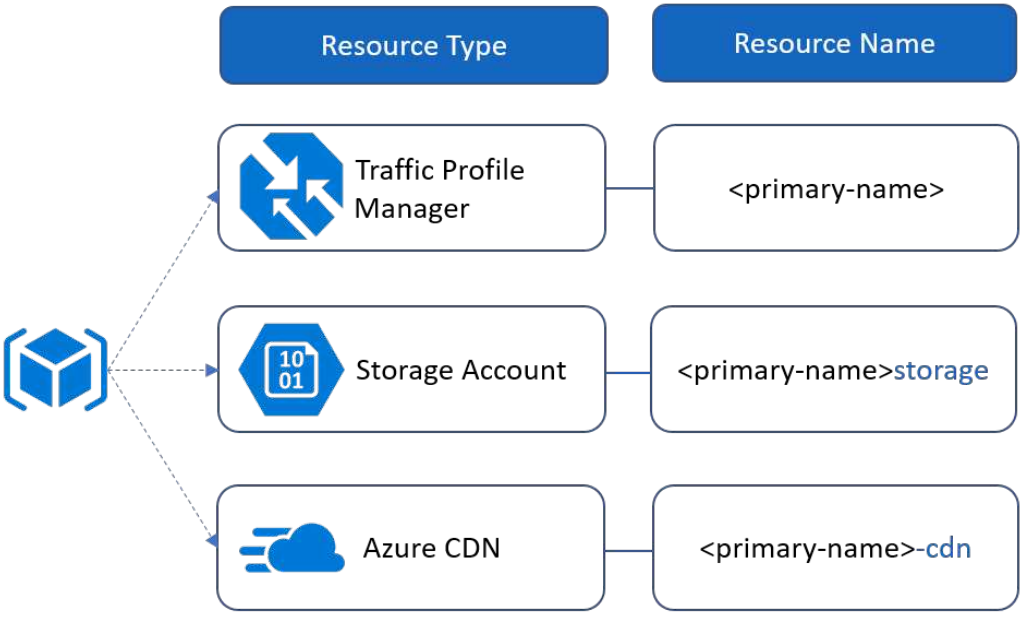
Architecture

A web application deployed in only one region, having all of its resources *(data, images, caches etc..)* has a big disadvantage, it doesn't ensure the lowest possible latency to all of the customers around the world. Assuming that all the resources are deployed in West Europe, the response will always be slower for users in United States or Asia. This is why we deploy our app globally, in multiple regions. But what does this mean about our data? If we deploy the app in multiple regions but still keep tha data on the same data center, then we don't solve the problem. The solution is to distribute the data globally as well but in such a way that there is a consinstensy for all users in different regions.

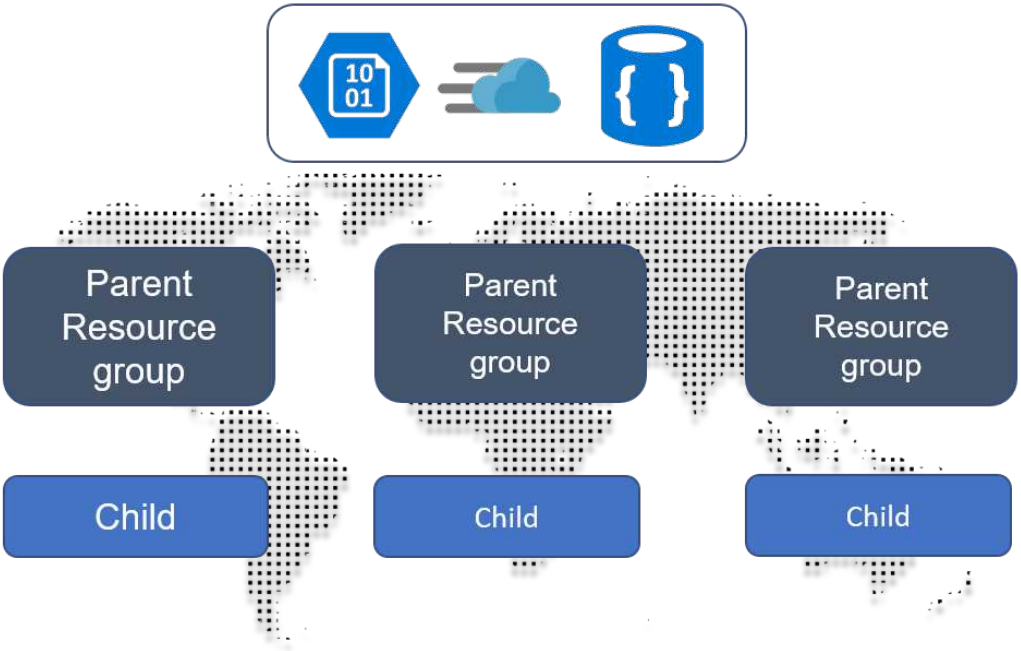
Let's examine the scenario of a simple e-commerce application where there are product items with images. Product related data don't change a lot and they are very good candidates to be stored as JSON objects or documents if you prefer, in Azure Cosmos DB. The great thing about Azure Cosmos DB is that you can replicate your data *(read only)* to other regions with one click. Replicating the data is the first part, reading them is the second one. Luckily, using the *Microsoft.Azure.DocumentDB.Core* package used for accessing Cosmos DB accounts, you can set preferred locations called *Connection Policies* for quering a Cosmos DB account. Assuming that the main read/write Cosmos DB database is in West Europe and it is replicated in West Central US, the web application deployed in West Central US will set as the first preferred location "West Central US" and as the second one "West Europe". The SDK will send the requests based on that Connection Policy configuration.

While product information is stored as JSON documents in Azure Cosmos DB, images shouldn't. JSON objects stores the image URI, a reference to the real image which should be stored in Azure Storage as a blob. Azure Storage is the recommended Microsoft-managed cloud service for storing data such as pictures, media files and many more. In our case we will store the product images. But how do we replicate images globaly? We use Azure Content Delivery Network which will cache the static images in many CDN POP locations around the world. The image URI will be the same for all deployed apps but they will always be served by the nearest POP location.

In the following sections you are going to create the **primary resource group**, the backbone of `Online.Store` application. For the moment, the primary resource group will contain the `Azure Storage`, `Azure CDN` and `Azure Cosmos DB` Azure services. Before proceeding make sure you have selected a **primary name** for your Azure resources. Due to the naming convenstion the resources in the primary resource group will be named as follow:



Primary Resource Group - Resources Names



Primary Resource Group

Primary Resource Group Location

Follow along with the examples and create the resource group in `West Europe`. Don't worry if you stay in a different region, when you finish the book will be able to select your own geographic locations for your resource groups

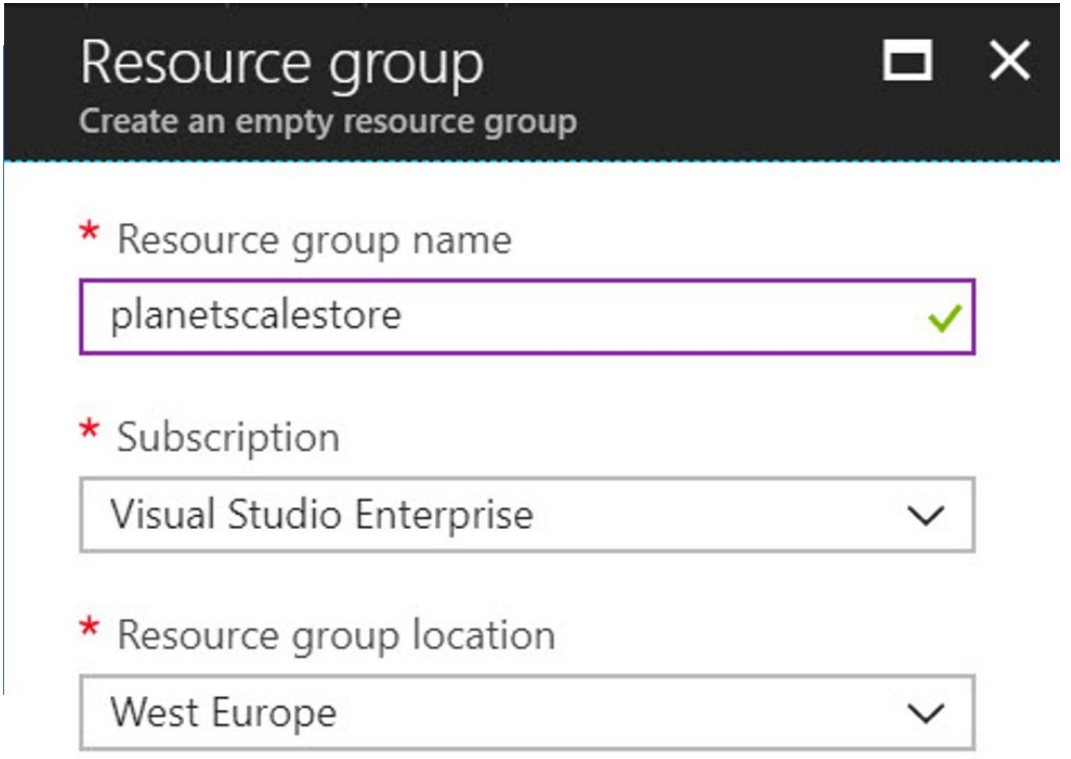
Create Primary Resource Group

In Azure Portal click the `Resource groups` menu item on the left panel and next click the `Add` button. Fill the form as follow:

1. **Resource group name**: <Primary-Name>
2. **Subscription**: <Select-Your-Subscription>
3. **Resource group location**: West Europe

Since this is the primary resource group, you give it the name you selected as the `primary`

name.



Create Primary Resource Group

Azure Storage

Overview

`Azure Storage` is a massively scalable, durable, highly available and secure cloud storage solution from Microsoft. The different types of storage that Microsoft offers are:

1. Blob storage
2. Table storage
3. Queue storage
4. Standard disk storage and Premium disk storage for high IOPS workloads
5. File storage for fully managed file share

`Azure storage` is considered as one of the most vital building blocks for building cloud based globally-distributed applications. It's a **pay-as-you-go** service that can be easily accessed through its exposed REST APIs using several multi-platform client libraries. The first 3 of the above types are intensively used in `Platform as a Service` ***(PaaS)*** solutions. Following are their basic features with examples.

Blob storage

You can think of `Blob storage` as a highly scalable, REST interface based object store in the cloud. In case your cloud application requires one of the following features, then the `unstructured` Blob Storage is the solution for you.

Feature	Examples
Data sharing	Share documents, pictures, video, music etc..
Big Data	Store raw data/logs and compute/map reduce over data
Backups	keep data and device backups

There are two types of blobs, `Block` blobs and `Page` blobs. The former is where you store your normal files such as pictures or documents and it is considered as one of the most cost

effective storage on top of azure. If you have a lots of files and you read them sequentially *(e.g. e-commerce apps)* then you put those files in block blobs. `Page` blobs on the other hand are optimized for sparse reads and random access files up to `8` TB in size.

Table storage

`Table storage` is the solution for storing **structured** NoSQL data in the cloud. Due to its schemaless nature, it's easy to adapt your data as your application models evolve. Accessing table storage data is fast enough and cost-effective *(typically lower cost than traditional SQL storage queries)*. Here are some of the `Table Storage` features.

- Store petabytes of semi-structured data and keep costs down
- Scale up without having to manually shard your dataset
- Highly available with `geo-redundant storage` being replicated three times within the same region and another three to a different one
- Store web application user data, address books, device information, and other metadata
- Easily accessible by developers through multi-platform client libraries

Azure Queues

Azure Queue storage provides a reliable communication mechanism for cloud services. When designing planet scale applications you want to decouple the components as much as possible so that they can scale independently. Those components can communicate each other using the asynchronous message queueing provided by Azure Queues. Consider the scenario of an e-commerce app that commits directly the orders in the database. If the database goes down then the web app goes down as well *(at least part of it)*. If the web app submits the orders in a queue though and other components asynchronously pick up those messages, when the database goes down the web app will still continue to function properly and continue to send messages to the queue. When the database goes up again, the *listening* components will pick up the messages and commit them to the database.

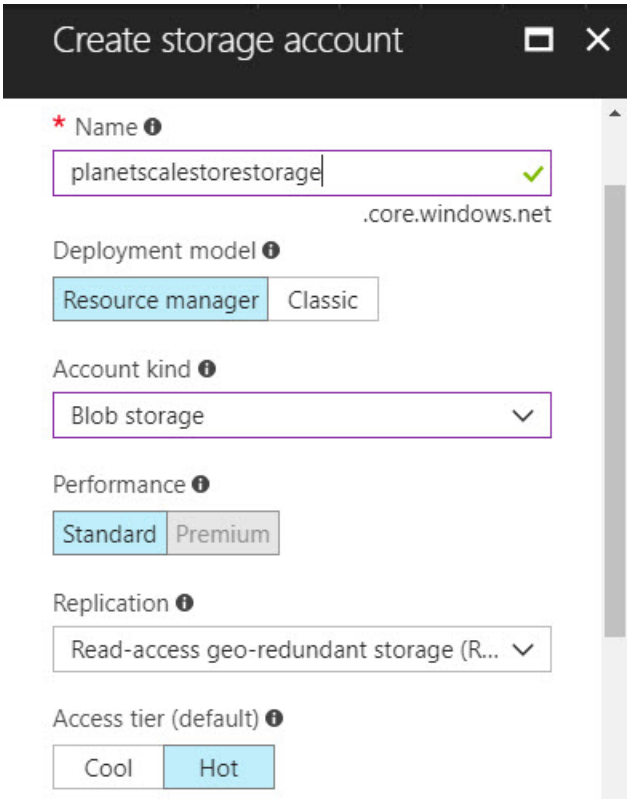
If you want to use all these services in Microsoft Azure, you need to create a **Storage account**. `Online.Store` application uses a single Azure Storage account of type `Blob` to store images for products and forum's topics. Blobs are stored in `containers` and hence there will be two containers, one for product images named `product-images` and another one named `forum-images`. Containers can be created either from the portal or from code *(Online.Store case)*

Azure Portal

Create a Storage Account

Click the `Storage Accounts` tab from the left panel and select `Add` from the new blade. Fill the form as follow and click the `Create` button:

1. **Name**: <Primary Name>storage
2. **Deployment Model**: Resource Manager
3. **Account kind**: Blob storage
4. **Performance**: Standard
5. **Replication**: Read-access geo-redundant storage
6. **Access tier**: Hot
7. **Secure transfer required**: Disabled
8. **Subscription**: <Select-Your-Subscription>
9. **Resource group**: Use existing <Primary Resource Group>
10. **Location**: West Europe

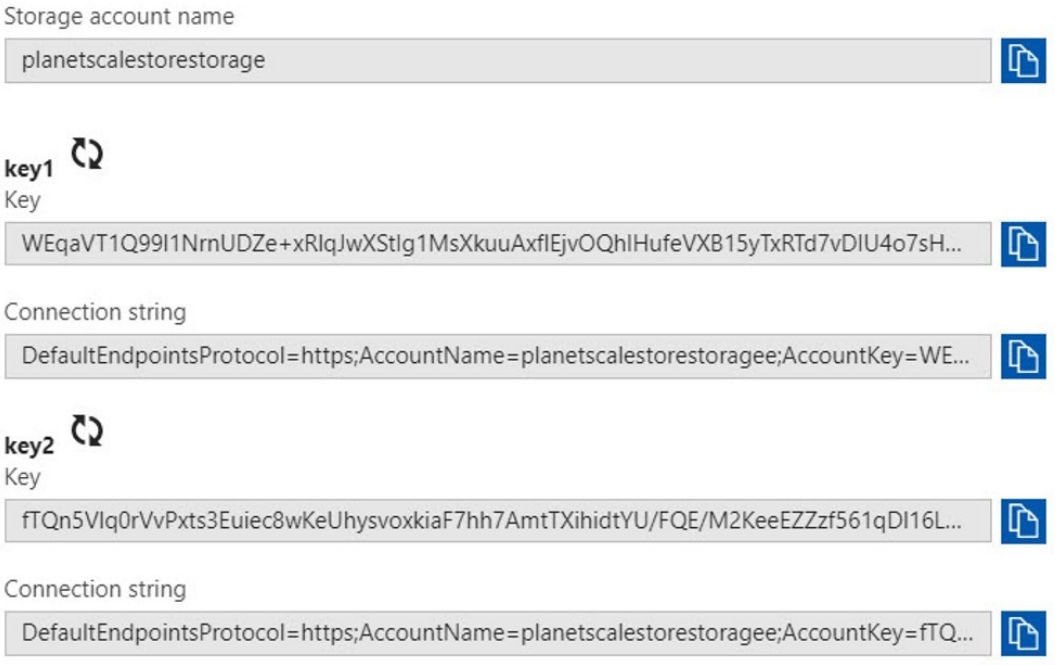


Create Azure Storage account

Make sure you omit the dash *(-)* in the name. This is required only for storage accounts. When your storage account is created go ahead and open that resource. You can to this either by:

- Select `Resource Groups` from the left menu and then your primary resource group. There you can always find all the resources belonging to the resource group and hense your new storage account
- Select `Storage accounts` from the left menu. You will find the new storage account named `<primary-name>storage`

When you open the resource, locate and select the `Access keys` tab. You need to get an key to access your account from code.



Storage account access keys

There are two properties to copy from the new blade. Copy the `Storage account name` value which should be equal to `<primary name>storage` and set it as the value for the *Storage:AccountName* in the **appsettings.json** file.

```
{
  ...
  "Storage:AccountName": "<your-storage-account-name>",
  ...
}
```

Next, copy the `Key1` Key value *(not the Connection String)* and set it as the value for the *Storage:AccountKey* property in the **secrets.json** file.

```
{
  ...
  "Storage:AccountKey": "<key1-key-value>",
  ...
}
```

Or just open a terminal, `cd` to `Online.Store` web application project and run:

```
dotnet user-secrets set "Storage:AccountKey" "<key1-key-value>"
```

Azure CDN

Overview

You build distributed applications because you want put the data that users trying to access, as close to their geographic locations as possible because `closer is always faster`. But what about static content that is deployed to a centralized location rather than the web application itself? You have seen that `Azure Storage` is a great solution to store images, media or other static files but does this mean that an `Azure Storage` account needs to be replicated to other regions around the world so that users can access static content faster? The answer is NO and this is where `Azure CDN` which stands for `Azure Content Delivery Network`, comes into the scene. Azure CDN is a cloud service that allows you to deliver static web content faster and more reliably to customers over the globe. It caches static content to a worldwide network of `Point-of-Presence` (POP) strategically chosen locations around the world in order to serve that content faster.

The way it works is simple. User requests static content such as images, from the `CDN` using a special endpoint like `<endpointname>.azureedge.net`. DNS will route the request to the POP location that is geographically closest to that user. If the `asset` not found in the POP edge server it will be fetched from the origin where is actually stored *(cloud service, App Service, storage account, etc..)*. The POP edge server will cache the `asset` for TTL (Time-To-Live) time and all feature requests to that asset will be served by the POP edge server itself. Here are the benefits you get by using Azure CDN.

- Ensures consistent user experience in global scale
- Reduces load times, saves bandwidth, and increases responsiveness
- Identify, absorbs and blocks security threats to absorb distributed `denial-of-service` (DDoS) attacks
- Handles sudden traffic spikes and heavy loads
- Monitor your traffic in real-time with granular analytics

Azure CDN comes in three different products, Azure CDN Standard from Akamai, Azure CDN Standard from Verizon, and Azure CDN Premium from Verizon. Each product offers different set of the features listed on the following table.

Performance / Optimizations	Security	Analytics and Reporting	Ease of Use
Dynamic Site Acceleration	HTTPS support with CDN endpoint	Azure diagnostic logs	Easy integration with Azure services
Large file optimization	Custom domain HTTPS	Core reports from Verizon	Management via REST API, .NET, Node.js, or PowerShell
Global Server Load balancing (GSLB)	Custom domain name support	Custom reports from Verizon	Customizable, rule-based content delivery engine
Fast purge	Geo-filtering	Advanced HTTP reports	URL redirect/rewrite
Asset pre-loading	Token authentication	Real-time stats	Mobile device rules
Cache/header settings	DDOS protection	Edge node performance	
IPv4/IPv6 dual-stack		Real-time alerts	
HTTP/2 support			

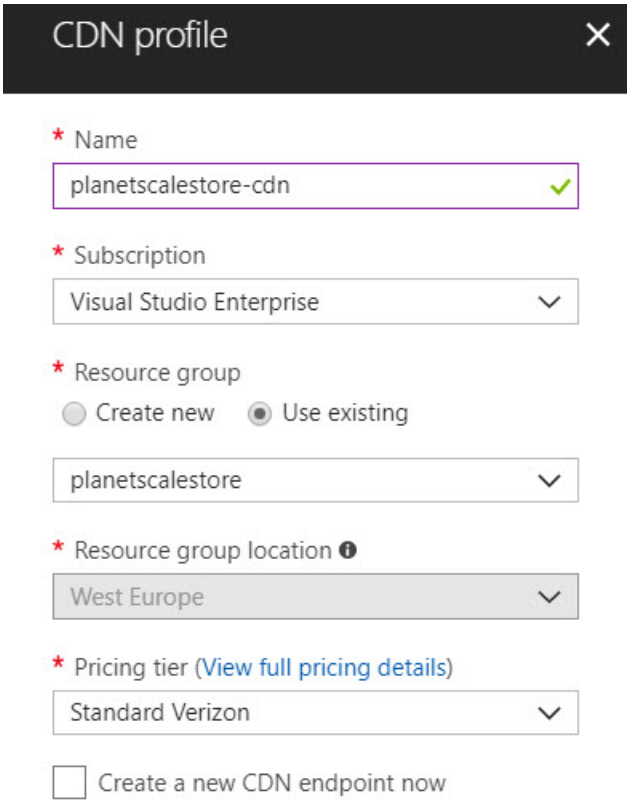
`Online.Store` application makes use of `Azure CDN` for serving static content stored in the Azure Storage account. Product images blobs are requested from the CDN endpoint and if not found they are served by the origin.

Azure Portal

Create a CDN Profile

In the left panel locate and click the `More Services` button at the bottom. Next search for and select the `CDN profiles` option. In the new blade click the `Add` button to create a new CDN profile. Fill the form as follow:

1. **Name**: <Primary Name>-cdn
2. **Subscription**: <Select-Your-Subscription>
3. **Resource group**: Use existing <Primary Resource Group>
4. **Pricing tier**: Standard Verizon
5. **Create a new CDN endpoint now**: Leave unchecked, you will create it later



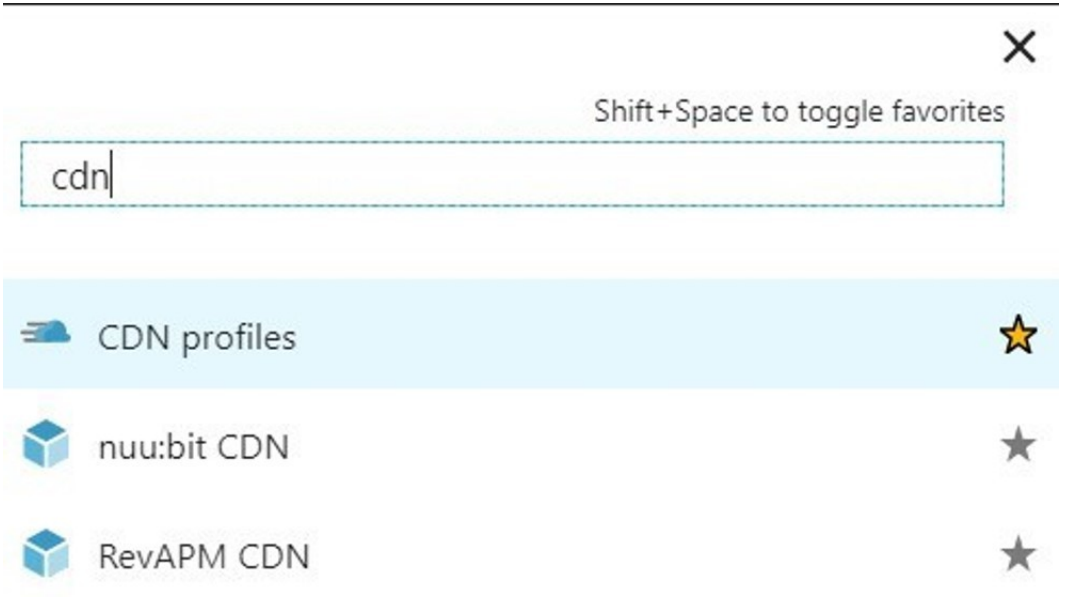
Create Azure CDN profile

Next locate the new CDN Profile resource. You can to this either by:

- In the left panel search for and select `CDN profiles` in the `More Services` blade as you did before. Notice that in the Search blade, you can click the **asterisk** button next to each resource type and it will always appear on the left panel.
- Locate the CDN profile resource after navigating to your Primary Resource Group.

Add resource type to favorites

If you want to list all resources of a specific type directly from the left panel rather than locating them from the resource group they belong to, all you have to do is click the **asterisk** button in the `More services` blade for the type of resource you wish to easily locate. The following image shows how to add `CDN Profiles` to your favorites resource types. After doing this, `CDN Profiles` option will always appear on the left panel.



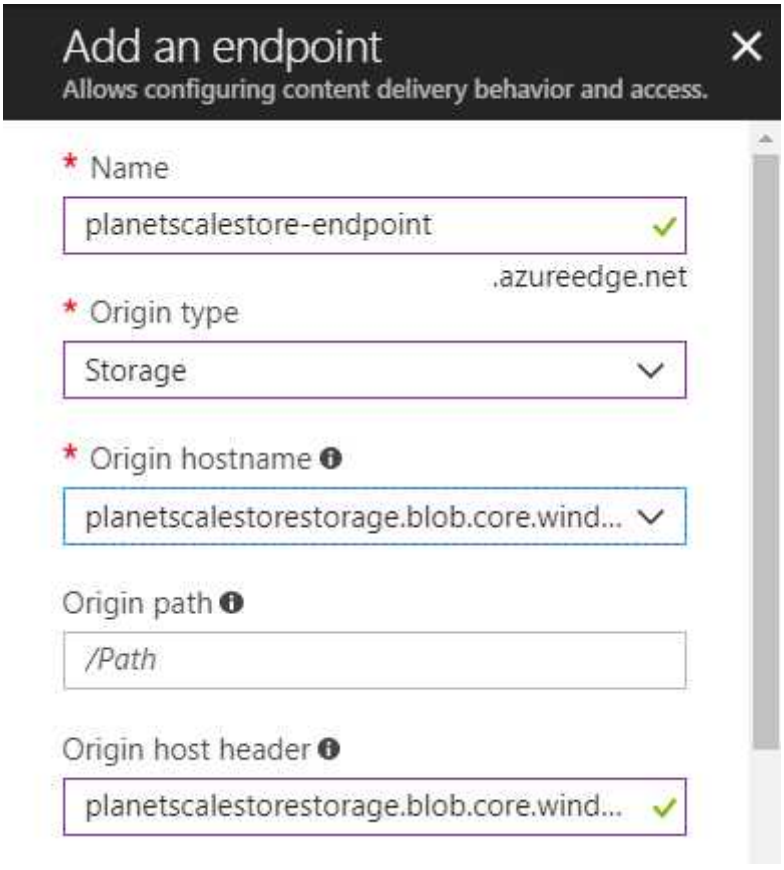
Mark a CDN Profiles as favorite

Add endpoint to CDN Profile

In the main blade of your CDN Profile select the `Endpoints` option

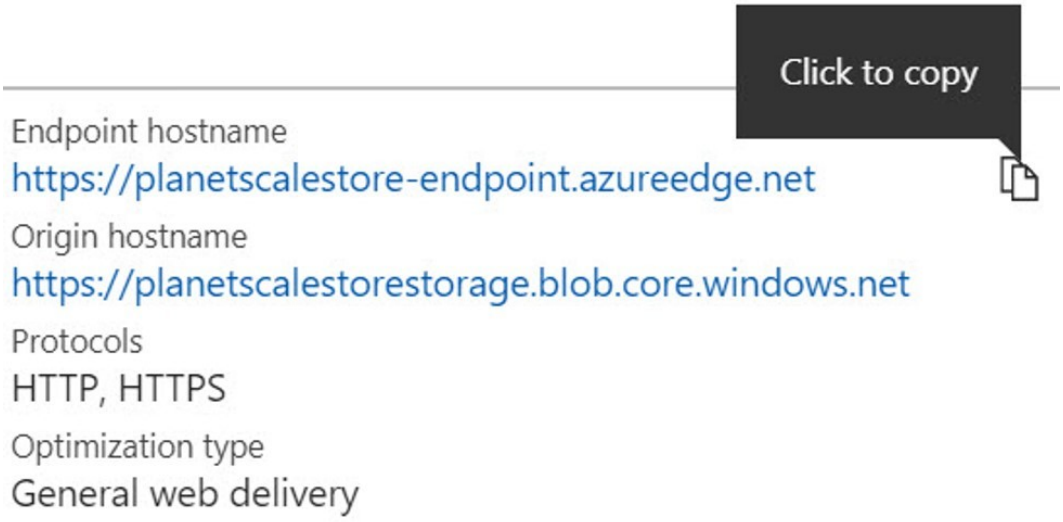
In the new blade click the `Endpoint` button with the plus sign to add a new Endpoint to your CDN profile. You will make this endpoint point to the Storage Account you created before. Fill the form as follow:

1. **Name**: <Primary Name>-endpoint
2. **Origin type**: Storage
3. **Origin hostname**: Select <Storage-Account-Name>.blob.core.windows.net from the dropdown list
4. **Origin path**: Leave empty
5. **Origin host header**: Should be filled automatically and have the same value as the `Origin hostname`
6. Protocol
   - **HTTP**: 80
   - **HTTPS**: 443
7. **Optimized for**: General web delivery



Add endpoint to CDN Profile

When the new endpoint is created, click on it and copy the `Endpoint hostname` value. This should be equal to *https://<primary-name>-endpoint.azureedge.net*.



CDN endpoint hostname

Set it as the value for the *CDN:Endpoint* property in the **appsettings.json** file.

```
{
  ...
  "CDN:Endpoint": "https://<primary-name>-endpoint.azureedge.net",
  ...
}
```

Azure Cosmos DB

Overview

`Azure Cosmos DB`, the next generation of *Azure Document DB*, is a globally distributed, multi-model database as a service which in single terms can bring your data to your users wherever they are. `Azure Cosmos DB` can automatically replicate your data to any number of regions of your choice, for fast and responsive access. Your web apps deployed in multiple regions can be configured to access data using region-based priorities *(preferred region list)*, for example the web app deployed in `West Central US` sends requests to the nearest data center first and if that region is not available then query the next available. Here are some of the features that `Azure Cosmos DB` offers.

- **Multi-region Replication** *(Turnkey global distribution)*: Distribute your data across regions with a single click
- **Manual and Automatic Failover**
  - If a read region has an outage it gets disconnected and all read calls are redirected to the next available region in the preferred region list
  - If a write region *(there can be only one write region)* fails, the region is automatically marked as offline and an alternative region is promoted as the write region
- "**Always on**" availability: 99.99% availability SLA for all single region database accounts, and all 99.999% read availability on all multi-region database accounts
- **Multi-model + multi-API**: One single database which gives you `KEY-VALUE`, `COLUMN-FAMILY`, `DOCUMENT (Mongo or SQL)` or `GRAPHS`. Access data using your favorite API including SQL, JavaScript, Gremlin, MongoDB, Apache Cassandra and Azure Table Storage

If your data changes a lot, not matter what their structure is (*key-values, documents or graphs*), if your data is being injested at really high speeds such as millions of transactions per second or if just there 's a lot of data *(e.g. petabytes)* then `Azure Cosmos DB` is what you go with.

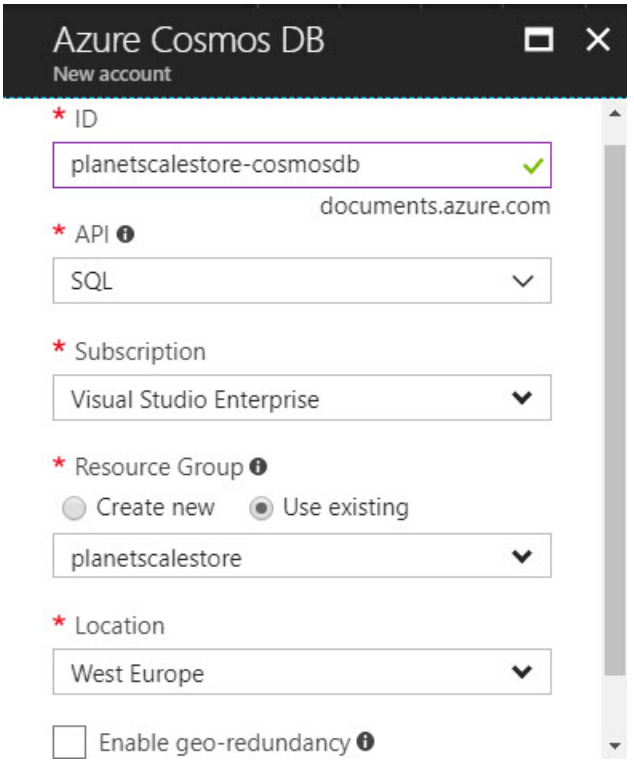
`Online.Store` application uses `Azure Cosmos DB` to save product and forum's data. Data are stored in database `collections` and therefore there will one database with two collections: `Items` for products and `Forum` for forums's topics and posts.

Azure Portal

Create an Azure Cosmos DB Account

At the Azure Portal select *Azure Cosmos DB* from the left panel and click the `Add` button. In case `Azure Cosmos DB` option doesn't appear on the left panel, you can always search for it by clicking `More services`. Clicking the `Add` button opens a blade which you need to fill as follow:

1. **ID**: <Primary-Name>-cosmosdb
2. **API**: SQL
3. **Subscription**: Your Subscription
4. **Resource Group**: Use existing <Primary-Resource-Group>
5. **Location**: West Europe
6. **Enable geo-redudancy**: Leave unchecked

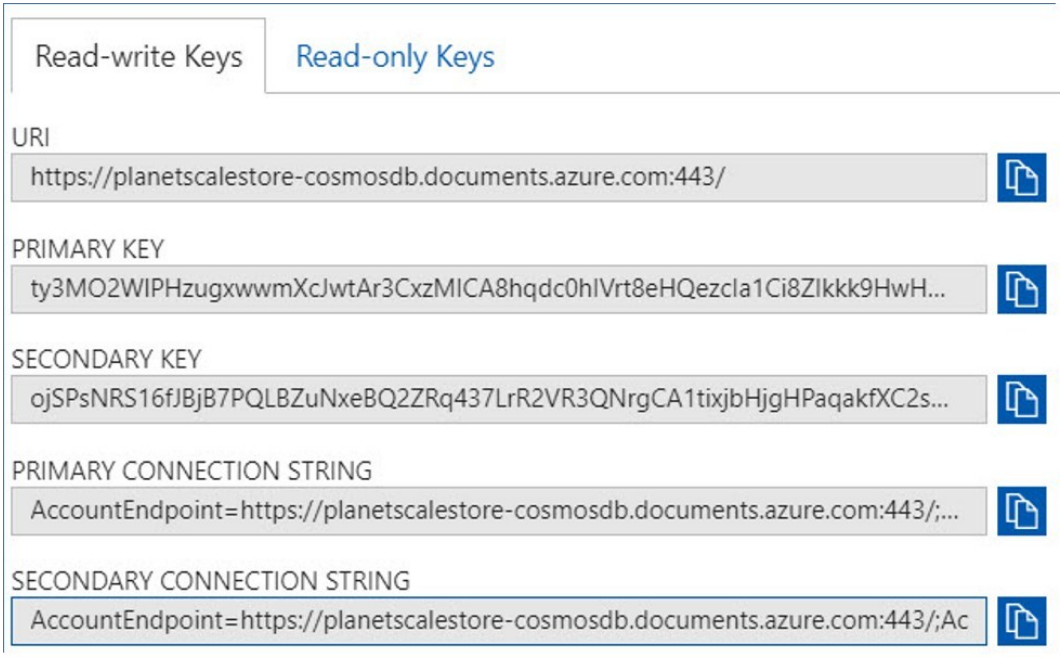


Create Azure Cosmos DB account

It usually takes a while for the deployment to finish but when it's done, go and check the new resource. You can find the resource created either by:

- Select `Resource Groups` from the left menu and then your primary resource group. There you can always find all the resources belonging to the resource group
- Select `Azure Cosmos DB` from the left menu. You will find the new Azure Cosmos DB account named `<primary-name>-cosmosdb`

Clicking the new resource will open a blade with many options. To access the account from the application you need to get the account's endpoint and an access key. Click **Keys** to open the associated blade.



Azure Cosmos DB account access keys

Copy the **Primary Key** value from the Keys and set it as the *DocumentDB:Key* value in the **secrets.json** file.

```
{
  "DocumentDB:Key": "<your-documentdb-primary-key>",
  "DocumentDB:ConnectionPolicies": "",
  "SearchService:ApiKey": "",
  ...
}
```

The equivalent `dotnet` command is the following:

```
dotnet user-secrets set "DocumentDB:Key" "<your-documentdb-primary-key>"
```

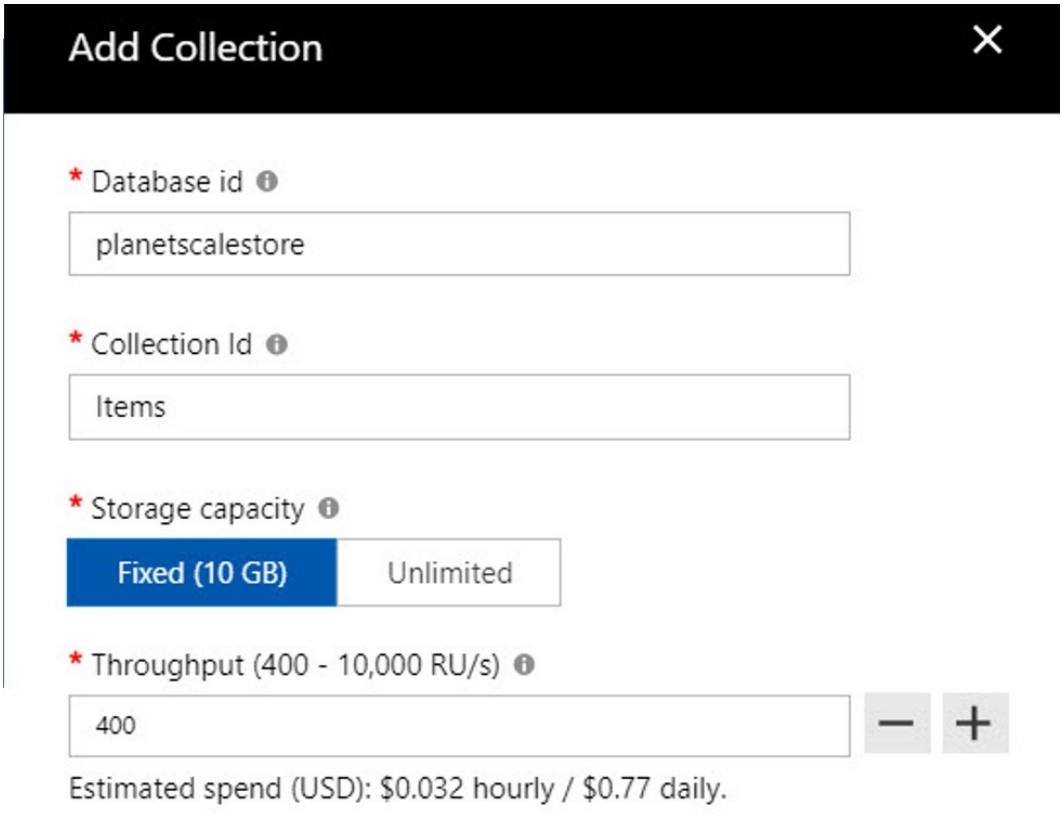
Next, set the *DocumentDB:Endpoint* property in the **appsettings.json** file equal to your `Azure Cosmos DB` account name.

```
{
  ...
  "DocumentDB:Endpoint": "<Primary Name>-cosmosdb",
  ...
}
```

Create an Azure Cosmos DB Collection

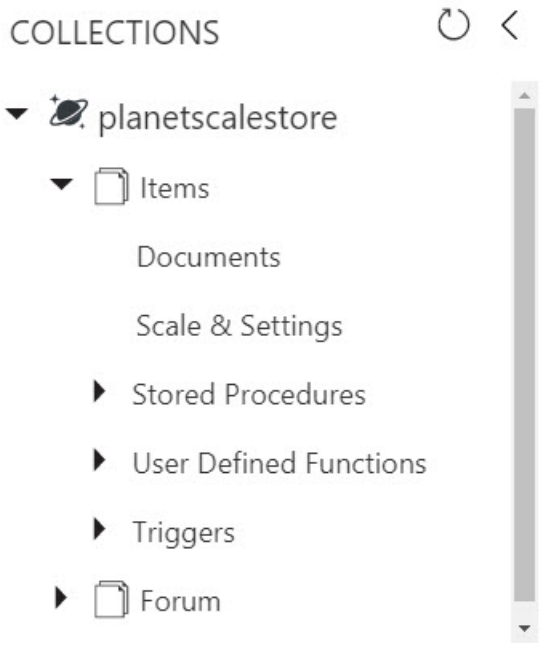
Close the *Keys* blade if it is still opened and click *Data Explorer*. Click `New Collection`, fill the form as follow and click OK:

1. **Database id**: <Primary-Name>
2. **Collection Id**: Items
3. **Storage capacity**: Leave the default
4. **Throughput**: 400



Azure Cosmos DB create Collection

Create another collection by clicking the `New Collection` button again and make sure to select the previously created Database id from the dropdown list. This time, name the collection *Forum*. By the end, you should have two collections under the database named by your `primary name`. The `Items` collection will hold the product documents while the `Forum` collection will hold topic related documents.



Azure Cosmos DB Collections

Now that you have created both the database and the collections, set the value for the `"DocumentDB:DatabaseId"` property in the **appsettings.json** file equal to your `primary name`.

```
{
  ...
  "DocumentDB:Endpoint": "<primary-name>-cosmosdb",
  "DocumentDB:DatabaseId": "<primary-name>",
  ...
}
```

Code Overview

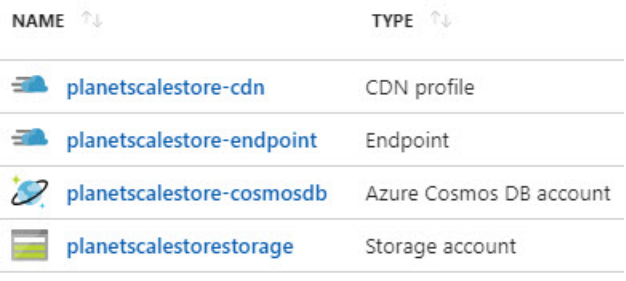
At this point you should have deployed the following Azure Resources:

- Azure Storage Account
- CDN Profile
- Azure Cosmos DB Account

These are all you need to fire up the `Online.Store` ASP.NET Core application for the first time but before doing this make sure that you have set the minimum required properties in the configuration files:

- **appsettings.json**
  1. `DocumentDB:Endpoint`: <primary-name>-cosmosdb
  2. `DocumentDB:DatabaseId`: <primary-name>
  3. `CDN:Endpoint`: https://<primary-name>-endpoint.azureedge.net
  4. `Storage:AccountName`: <primary-name>storage
- **secrets.json**
  1. `DocumentDB:Key`: <your-documentdb-primary-key>
  2. `Storage:AccountKey`: <key1-key-from-storage-accesskeys>

In the example images you have seen so far, **planetscalestore** has been used as the Primary Resource Group name. Here is how the resources looks like in that resource group created in `West Europe`.



Primary Resource Group Resources

In your own primary resource group you should have the same resources with a different name though. Build the solution and make sure you don't have any compile errors. The project is built 100% in .NET Core so it should compile in Windows, Linux and MAC OS *(tested in Windows 10 and Ubuntu 16.04)*. The web application project is the `Online.Store` and can be started in two ways:

- **Windows - Visual Studio 2017**: Right click the `Online.Store` web application project and set it as the default. Hit F5 to fire it up
- **.NET Core CLI**:
  1. Open a terminal and `cd` to the solution's folder
  2. Restore and build the packages

     ```
     dotnet restore
     dotnet build
     ```

  3. `cd` to the `Online.Store` project
  4. Run the project

     ```
     dotnet run
     ```

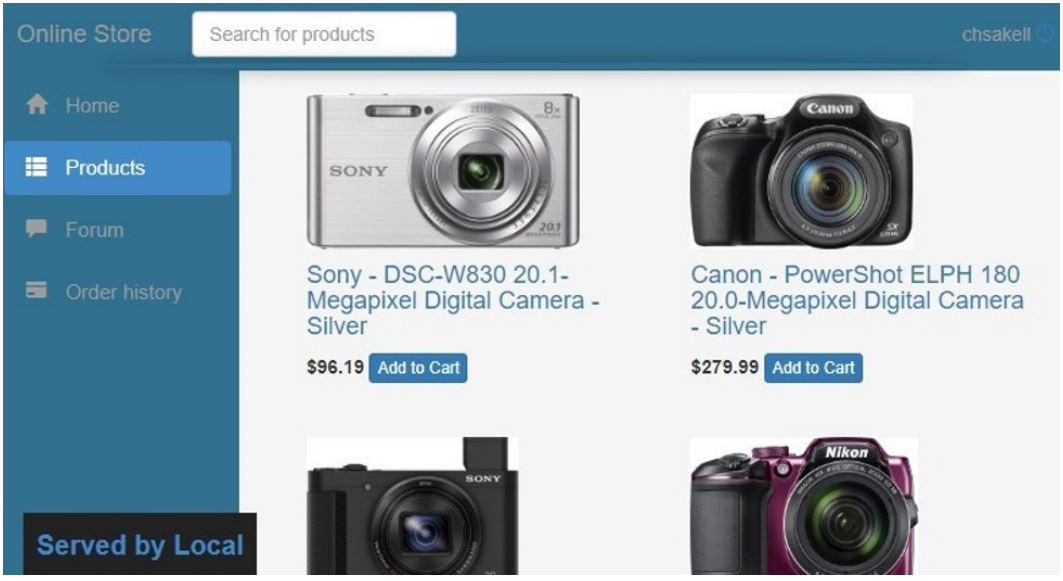
NPM packages

Before building for the first time, make sure you have **npm packages** installed in the `Online.Store` web project. In case you haven't, open a terminal, `cd` to that folder and run `npm install`

Open a browser and navigate to http://localhost:57278 *(dotnet cli)* or https://localhost:44309/ *(Visual Studio)*. If it's the first time you run the application it will take a while to initialize some mock products for you. During this process and only if you have properly created and configured the settings for the `Storage Account`, `CDN` and `Cosmos DB` Azure services, the app will do the following:

1. Create blob containers in your storage account to store images for products and forum topics - `StorageInitializer` class
2. Upload the products and topics found in the *wwwroot/products.json* and *wwwroot/-topics.json* files respectively at the `Items` and `Forum` collections you created in your Cosmos DB database. Don't worry if you haven't created the collections from the portal, the initializer will do it for you in that case - `DocumentDBInitializer` class

In the website, click `Products` from the left menu and confirm that you can see the products.



Product list

Now let's review some code starting with `Azure Storage`. Each Azure Service in the solution has its own library, so you will find a `.NET Core` library project named *Online.Store.Storage*. To access your storage account from code you need to install the WindowsAzure.Storage `NuGet` package. The *Online.Store.Storage* project contains a `IStorageRepository` interface with the following functions:

```
public interface IStorageRepository
{
    void Connect(string accountName, string accountKey);

    CloudBlobContainer GetBlobContainer(string containerName);

    Task CreateBlobContainerAsync(string containerName);
    Task UploadToContainerAsync(string containerName,
                                string filePath,
                                string blobName);

    Task<string> UploadToContainerAsync(string containerName,
                                        Stream fileStream,
                                        string blobName,
```

```
                                        string contentType);
}
```

`StorageRepository` class exists on the same project and implements the interface. It contains two basic properties for creating Azure Storage clients.

```
protected CloudStorageAccount cloudStorageAccount { get; set; }
protected CloudBlobClient cloudBlobClient { get; set; }
```

The implementation of the `Connect` method explains what they are used for:

```
public void Connect(string accountName, string accountKey)
{
  string connection =
  string.Format(@"DefaultEndpointsProtocol=https;AccountName={0};AccountKey={1}"
  ,accountName, accountKey);

  cloudStorageAccount = accountName == "devstoreaccount1" ?
      CloudStorageAccount.DevelopmentStorageAccount :
      CloudStorageAccount.Parse(connection);
  cloudBlobClient = cloudStorageAccount.CreateCloudBlobClient();
}
```

First you create an object of type `CloudStorageAccount` using the storage account name and an access key. Then you use this object to create clients for accessing the account. This method may be called only once during the repository initialization but must be the first call before trying to access a storage account. `Dependency injection` for repositories and services are configured in the `Startup` class in the `Online.Store` project.

```
services.AddScoped<IDocumentDBRepository, DocumentDBStoreRepository>();
services.AddScoped<IRedisCacheRepository, RedisCacheReposistory>();
services.AddScoped<IServiceBusRepository, ServiceBusRepository>();
services.AddScoped<IStorageRepository, StorageRepository>();
services.AddScoped<ISearchRepository, SearchStoreRepository>();
services.AddScoped<IStoreService, StoreService>();
services.AddScoped<IServiceBusService, ServiceBusService>();
services.AddScoped<IMediaService, MediaService>();
services.AddScoped<IAzureSearchService, AzureSearchService>();
```

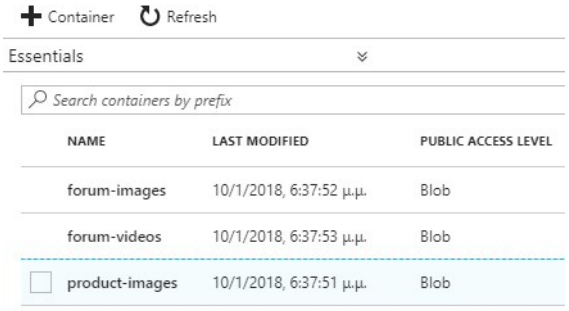
`Online.Store` web project doesn't have a direct reference to repositories but access them through the `Online.Store.Azure.Services` library where all the services exist. There you will find a `MediaService` class injecting in the constructor an instance of `IStorageRepository` which is used to connect to the configured storage account.

```
public MediaService(IConfiguration configuration,
                    IStorageRepository storageRepository)
{
    _storageRepository = storageRepository;

    _storageRepository.Connect(configuration["Storage:AccountName"],
                               configuration["Storage:AccountKey"]);
}
```

Back in the `StorageRepository` concrete class, here is how you reference or create a container.

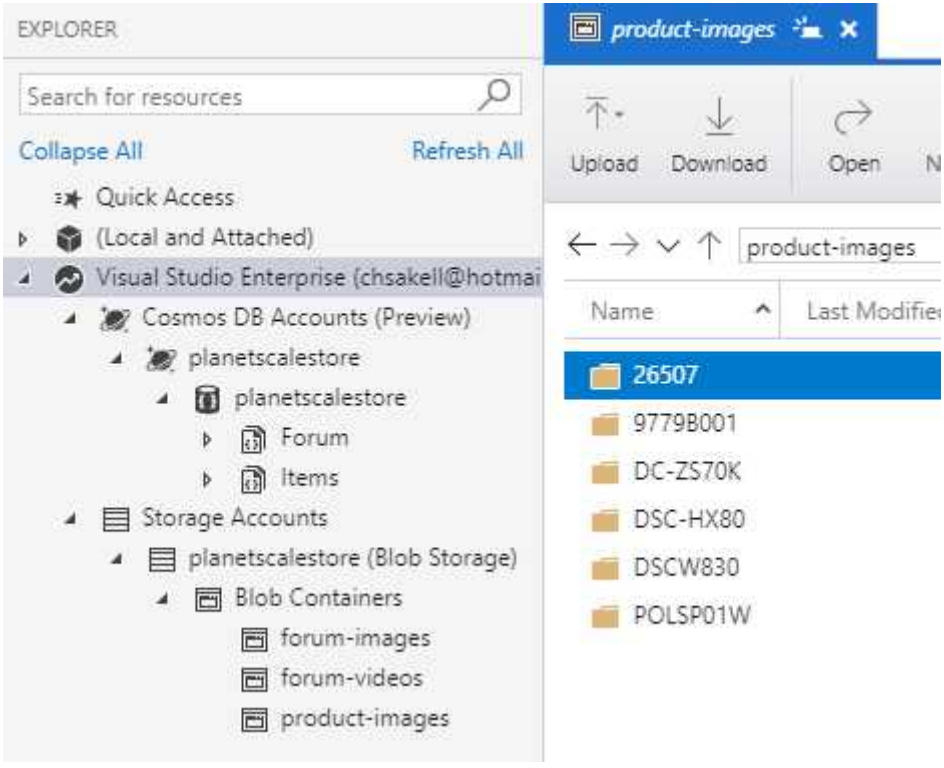
```
public async Task CreateBlobContainerAsync(string containerName)
{
    CloudBlobContainer cloudBlobContainer = GetBlobContainer(containerName);

    bool containerExists = await cloudBlobContainer.ExistsAsync();
    {
        if (!containerExists)
        {
            await cloudBlobContainer.CreateAsync();
            BlobContainerPermissions permissions =
              new BlobContainerPermissions();
            permissions.PublicAccess = BlobContainerPublicAccessType.Blob;
            await cloudBlobContainer.SetPermissionsAsync(permissions);
        }
    }
}

public CloudBlobContainer GetBlobContainer(string containerName)
{
    CloudBlobContainer cloudBlobContainer =
      cloudBlobClient.GetContainerReference(containerName);
    return cloudBlobContainer;
}
```

You get a reference to a container through an instance of `CloudBlobClient`. The `CreateBlobContainerAsync` method configures the container so that can be accessed from public, otherwise you wouldn't be able to view product images. But what is a container anyway? In Azure Portal go to your storage account resource and in the main blade select `Containers`. You should see 3 containers.



Product list

You can select a specific container and check the blobs inside it, sometimes though you might not be able to view them in the portal *(issues with folders)*. This is why you should use and install Azure Storage Explorer, a tool that allows you to access and manage both Storage and Cosmos DB accounts. After installing it, sign in with your Azure login and you will see all your storage accounts.



Product list

There are two different methods to upload files in `StorageRepository`. The first one is the `UploadToContainerAsync` which is used to upload a file directly from its path:

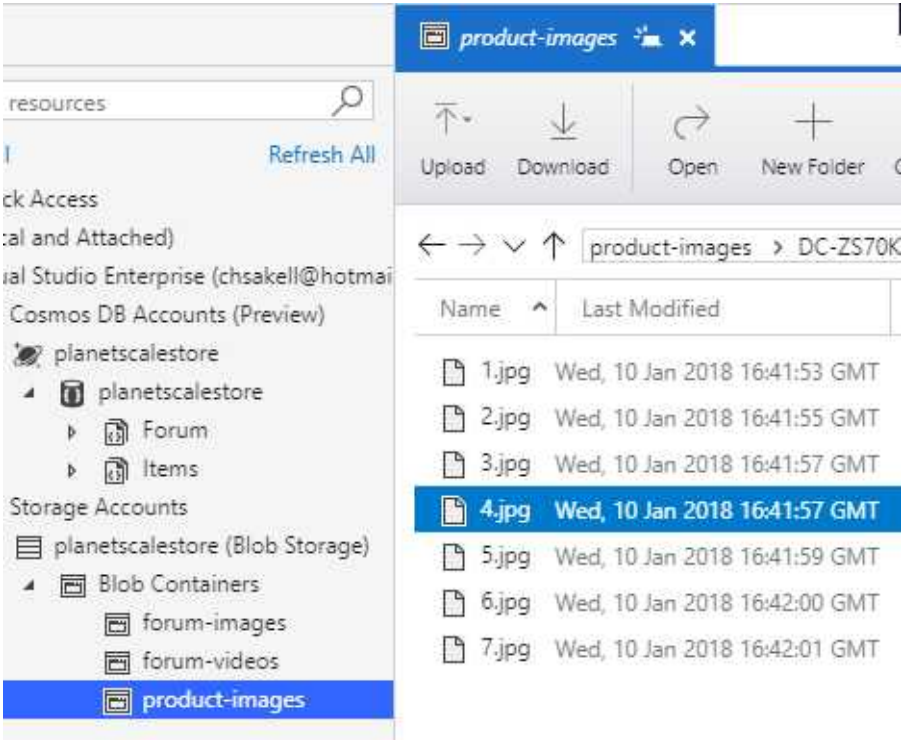
```
public async Task UploadToContainerAsync(string containerName,
                                         string filePath,
                                         string blobName)
{
    CloudBlobContainer cloudBlobContainer =
      GetBlobContainer(containerName);

    CloudBlockBlob cloudBlockBlob =
      cloudBlobContainer.GetBlockBlobReference(blobName);

    await cloudBlockBlob.UploadFromFileAsync(filePath);
}
```

After retrieving a reference to the container you want to upload a blob *(e.g. product-images)*,

you create an instance of `CloudBlockBlob` using the path where you want this blob to be uploaded. A Camera product in `Online.Store` app may have many images which are stored in a folder named as the product's **Model** property. So all URI images for product *DC-ZS70K* are in the form of:

```
https://<storage-account-name>.blob.core.windows.net/product-images/DC-ZS70K/
```

plus the file name.



Product images

The part where the mock product images are uploaded during the first initialization lies in `DocumentDBInitializer` class:

```
private static async Task UploadProductImagesAsync(Product product)
{
    string[] images = Directory.GetFiles(@"wwwroot/images/" + product.Model);

    await StorageInitializer._repository
      .UploadToContainerAsync("product-images", images[0],
                              product.Model + "/" +
                              Path.GetFileName(images[0]));
    product.Image =
    string.Format("https://{0}.blob.core.windows.net/{1}/{2}/{3}",
                  storageAccount, "product-images",
                  product.Model, Path.GetFileName(images[0]));

    ProductComponent mediaComponent =
      product.Components
      .Where(c => c.ComponentType == "Media").First();

    for(int i=1; i < images.Length; i++)
    {
        await StorageInitializer._repository
          .UploadToContainerAsync("product-images", images[i],
                                  product.Model + "/" + Path.GetFileName(images[i]));

        ProductMedia media = new ProductMedia();
        media.Id = Guid.NewGuid().ToString();
        media.CreatedDate = DateTime.Now;
        media.Name = Path.GetFileNameWithoutExtension(images[i]);
        media.Url =
          string.Format("https://{0}.blob.core.windows.net/{1}/{2}/{3}",
                        storageAccount, "product-images",
                        product.Model, Path.GetFileName(images[i]));
        media.Type = "image";
        mediaComponent.Medias.Add(media);
    }
}
```

The second method to upload files in storage container is through a `Stream`:

```
public async Task<string> UploadToContainerAsync(
                          string containerName, Stream fileStream,
                          string blobName, string contentType)
{
    CloudBlobContainer cloudBlobContainer = GetBlobContainer(containerName);
    CloudBlockBlob cloudBlob =
      cloudBlobContainer.GetBlockBlobReference(blobName);

    cloudBlob.Properties.ContentType = contentType;

    using (var stream = fileStream)
    {
        await cloudBlob.UploadFromStreamAsync(stream);
    }

    return cloudBlob.StorageUri.PrimaryUri.AbsoluteUri;
}
```

This method is used at the `MediaService` class for uploading topic images:

```
public async Task<string> UploadMediaAsync(Stream stream, string filename,
                                           string contentType)
{
    string container = contentType.ToLower().Contains("image")
      ? "forum-images" : "forum-videos";

    return await _storageRepository
                  .UploadToContainerAsync(container, stream,
                                          filename, contentType);
}
```

You cannot test this method at the moment because you haven't configured an **Authentication** mechanism for the application yet. Don't worry, you 'll get to it later on.

The repositories for accessing a `DocumentDB` account are inside the `Online.Store.DocumentDB` library project. In the `IDocumentDBRepository` interface you will find several methods for accessing collections in a DocumentDB account. Let's start with the basics and check how you connect to a speficic database collection. There is a `DocumentDBRepositoryBase` class that implements the interface but doesn't configures the connection to DocumentDB account. It has though all the required properties to do so:

```
public abstract class DocumentDBRepositoryBase : IDocumentDBRepository
{
    protected string Endpoint = string.Empty;
    protected string Key = string.Empty;
    protected string DatabaseId = string.Empty;
    protected string CollectionId = string.Empty;
    protected Database _database;
    protected DocumentClient _client;
    protected DocumentCollection _collection;
    // code omitted
}
```

For each account you wish to access you create a concrete class such as the `DocumentDBStoreRepository` that inherits from `DocumentDBRepositoryBase`. This pattern will help you separate configuration settings in case you want to access more than one DocumentDB accounts. The constructor of this class sets the basic configuration properties:

```
private List<string> ConnectionPolicies;

public DocumentDBStoreRepository(IConfiguration configuration)
{
    Endpoint = string.Format("https://{0}.documents.azure.com:443/",
                    configuration["DocumentDB:Endpoint"]);
    Key = configuration["DocumentDB:Key"];
    DatabaseId = configuration["DocumentDB:DatabaseId"];

    string policies = configuration["DocumentDB:ConnectionPolicies"];
    if(!string.IsNullOrEmpty(policies))
    {
        ConnectionPolicies = policies.Split(",").ToList();
    }
}
```

All these properties are required for creating `Azure Cosmos DB` clients. You haven't configured *DocumentDB:ConnectionPolicies* but it doesn't matter at the moment. One of the most important features of Azure Cosmos DB is its replication capabilites. You can replicate an account to many regions around the world with a simple click. The connection policies configures the priorites for accessing those replicas. For example if you have the main account in *West Europe* and you replicate it to *West Central US* then the web app that is deployed in Europe should have the `West Europe` instance as the first priority and `West Central US`

as the second. The opposite applies for the web app that is deployed to `West Central US`, meaning that first it should try to connect to `West Central US` instance and if failed try the default one in `West Europe`. If you don't set any connection policy on client initialization then the default instance will be used *(as happens till now)*. `DocumentDBStoreRepository` class provides the implementation for initializing clients that can access a specific collection.

```
public override async Task<DocumentCollection> InitAsync(string collectionId)
{
    ConnectionPolicy connectionPolicy = new ConnectionPolicy();

    //Setting read region selection preference
    if (ConnectionPolicies != null)
    {
        foreach (var policy in ConnectionPolicies)
        {
            connectionPolicy.PreferredLocations.Add(policy);
        }
    }

    // Check Microsoft.Azure.Documents.LocationNames

    // first preference
    //connectionPolicy.PreferredLocations.Add(LocationNames.WestCentralUS);
    // second preference
    //connectionPolicy.PreferredLocations.Add(LocationNames.EastUS);

    if (_client == null)
        _client = new DocumentClient(new Uri(Endpoint), Key, connectionPolicy);

    if (CollectionId != collectionId)
    {
        CollectionId = collectionId;
        _collection = await _client
          .ReadDocumentCollectionAsync(UriFactory
            .CreateDocumentCollectionUri(DatabaseId, CollectionId));
    }

    return _collection;
}
```

Every time you want to query a collection, all you have to do is call the `InitAsync` collection method passing the collection id *(e.g. Items)*. Using the same client you can switch and query

a different collection simply by calling this method.

Documents in Cosmos DB are saved as **JSON** objects something that allows you to use generic methods for CRUD operations. Before viewing the implentations for quering documents, take a look either in the portal or the `storage explorer` how a product or a topic document looks like inside a collection. In the portal open the Cosmos DB resource on your primary resource group and in the main blade click `Data Explorer`. Open the *Items* collection and click documents. Select one of the documents and check the JSON object.

```
{
"components": [
 {
    "id": "8c2a93fb-7bd2-40a4-a75b-3d9306986471",
    "componenttype": "Media",
    "title": "",
    "detail": "",
    "created": "2014-11-22T09:48:52.1054935Z",
    "updated": null,
    "medias": [
    {
        "id": "0133077d-8ab9-4a62-8d19-dba7f740f043",
        "name": "2",
        "type": "image",
        "url": "https://planetscalestore.blob.core.windows.net/product-images/DSCW830\
/2.jpg",
        "created": "2018-01-10T18:40:47.51567+02:00",
        "updated": null
    },
    // code omitted
```

This is a good opportunity to start checking the `Online.Store.Core` library project where all the domain classes exist. Take a look the components that `Product` class aparts from. `ProductInfo` has all the basic information such as the *Id*, the main *Image*, *Title* and *Model*. It has a list of `ProductComponent` items which in turn apart from some basic properties and a list of `ProductMedia` items.

The `GetItemsAsync` generic method implemented in the `DocumentDBRepositoryBase` class is used for quering documents in a collection.

```
public async Task<IEnumerable<T>> GetItemsAsync<T>() where T : class
{
    IDocumentQuery<T> query = _client.CreateDocumentQuery<T>(
        UriFactory.CreateDocumentCollectionUri(DatabaseId,
                                               CollectionId),
        new FeedOptions {
            MaxItemCount = -1,
            EnableCrossPartitionQuery = true
        })
        .AsDocumentQuery();

    List<T> results = new List<T>();
    while (query.HasMoreResults)
    {
        results.AddRange(await query.ExecuteNextAsync<T>());
    }

    return results;
}
```

In the `StoreService` you can find how you use this method to query products.

```
public async Task<IEnumerable<Product>> GetProducts()
{
    List<Product> products = new List<Product>();
    await _docDbRepository.InitAsync(_PRODUCT_COLLECTION_ID);

    return await _docDbRepository.GetItemsAsync<Product>();
}
```

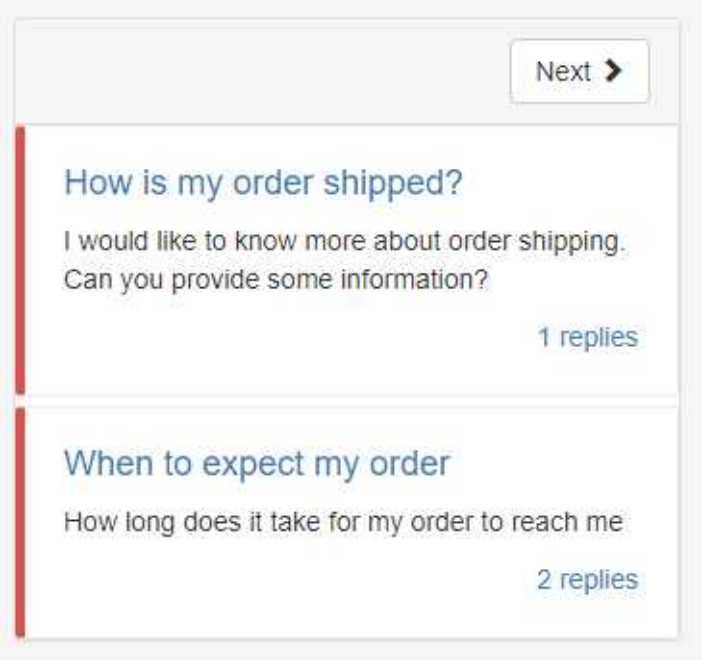
It's that simple, first you **target** the collection you wish to query and then you pass the type of documents you expect to retrieve. Most of the times you propably want to retrieve a paged result rather than all the items in a collection. Moreover, the JSON object might have much more information than you actually need to render in the page. For example the `forum/topics` route renders only the basic information for topics, such as title or number of replies. The `CreateDocumentQueryAsync` method implements this functionality:

```
public async Task<Dictionary<List<T>, string>>
    CreateDocumentQueryAsync<T>(int size, string continuationToken,
        string query = null) where T : class
{
    Dictionary<List<T>, string> result = new Dictionary<List<T>, string>();

    var options = new FeedOptions
    {
        MaxItemCount = size
    };

    if(!string.IsNullOrEmpty(continuationToken))
    {
        options.RequestContinuation = continuationToken;
    }

    IDocumentQuery<T> documentQuery;
    if (string.IsNullOrEmpty(query)) {
        documentQuery = _client
        .CreateDocumentQuery<T>(_collection.DocumentsLink, options)
        .AsDocumentQuery();
    }
    else  {
        documentQuery = _client
        .CreateDocumentQuery<T>(_collection.DocumentsLink, query, options)
        .AsDocumentQuery();
    }

    var response = await documentQuery.ExecuteNextAsync<T>();

    var list = new List<T>();
    list.AddRange(response);

    result.Add(list, response.ResponseContinuation);

    return result;
}
```

Here a **ResponseContinuation** token is used for subsequent requests. If a response has a **ResponseContinuation** token then passing that value on the next request will fetch the next paged result set. If not it means that you are on the last page. You can test this functionality by selecting the *Forum* menu item which retrieves Topics in the `Forum` collection. Each time

2 topics are only retrieved from the `StoreService`. Notice that only specific properties are requested while replies are ignored.

```
public async Task<PagedTopics> GetTopics(int size, string continuationToken)
{
    await _docDbRepository.InitAsync(_FORUM_COLLECTION_ID);

    string query = "select c.id, c.title, c.content,
                    c.mediaDescription, c.mediaUrl, c.mediaType,
                    c.userId, c.createdDate from c";

    var dic = await _docDbRepository
        .CreateDocumentQueryAsync<Topic>(size, continuationToken, query);

    return new PagedTopics
    {
        Topics = dic.Keys.First().OrderByDescending(t => t.CreatedDate).ToList(),
        ContinuationToken = dic.Values.First()
    };
}
```



Paged Topics

You can use the `GetItemAsync` method to get the data for a specific document. The method parses the entire document object:

```
public async Task<T> GetItemAsync<T>(string id) where T : class
{
    try
    {
        Document document = await _client
            .ReadDocumentAsync(UriFactory
                .CreateDocumentUri(DatabaseId, CollectionId, id));
        return (T)(dynamic)document;
    }
    catch (DocumentClientException e)
    {
        if (e.StatusCode == System.Net.HttpStatusCode.NotFound)
        {
            return null;
        }
        else
        {
```

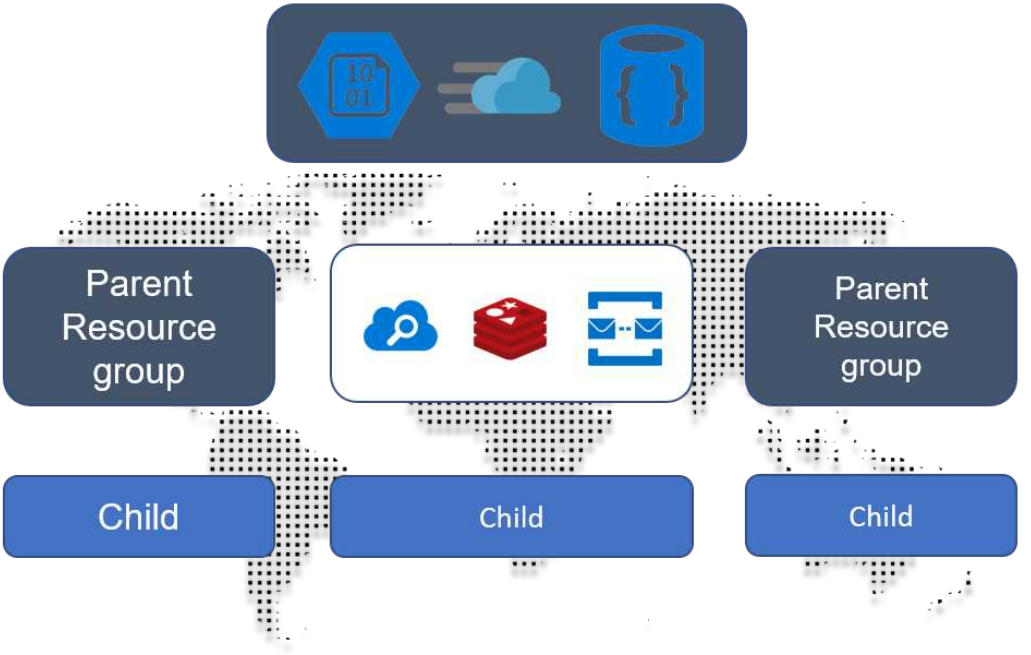
```
            throw;
        }
    }
}
```

This method is called in the `StoreService` when you click on a specific Camera product. It retrieves all available information with all the components *(Features, Specifications, Compatibility etc..)*.

```
public async Task<Product> GetProductDetails(string productId)
{
    await _docDbRepository.InitAsync(_PRODUCT_COLLECTION_ID);
    var product = await _docDbRepository.GetItemAsync<Product>(productId);
    return product;
}
```

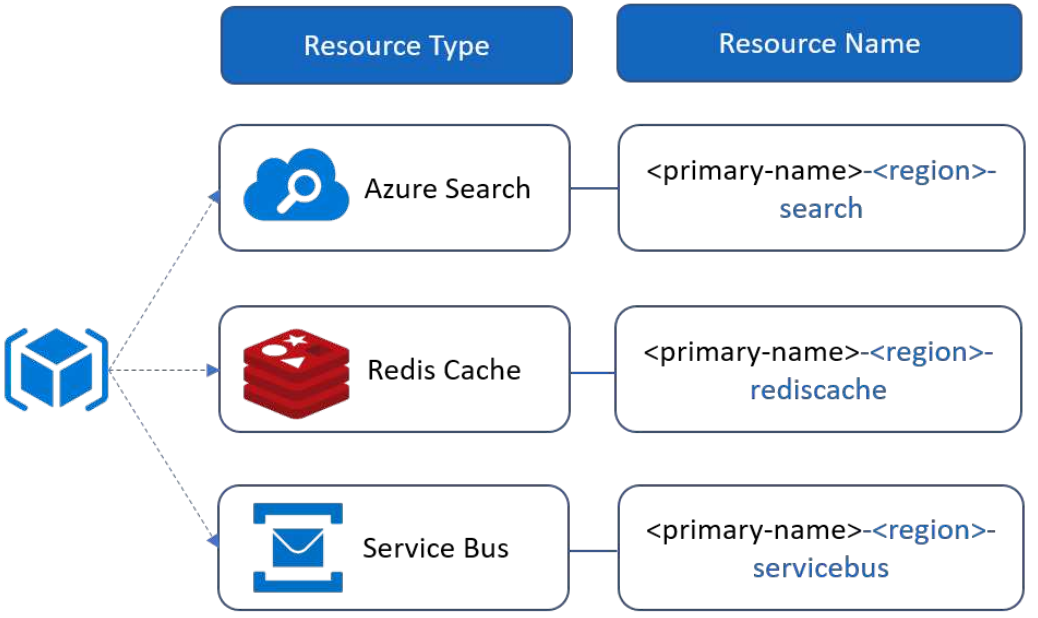
Performance Optimization

In the following sections you are going to create your first `Parent Resource Group` in `West Europe`. A `parent resource group` contains the `Azure Search`, `Redis Cache` and `Service Bus` services for a specific Azure region and acts as the second level in the resource groups hierarchy.



Parent Resource Groups

Before creating and start adding resources in the parent resource group, recall the naming convention you should follow.



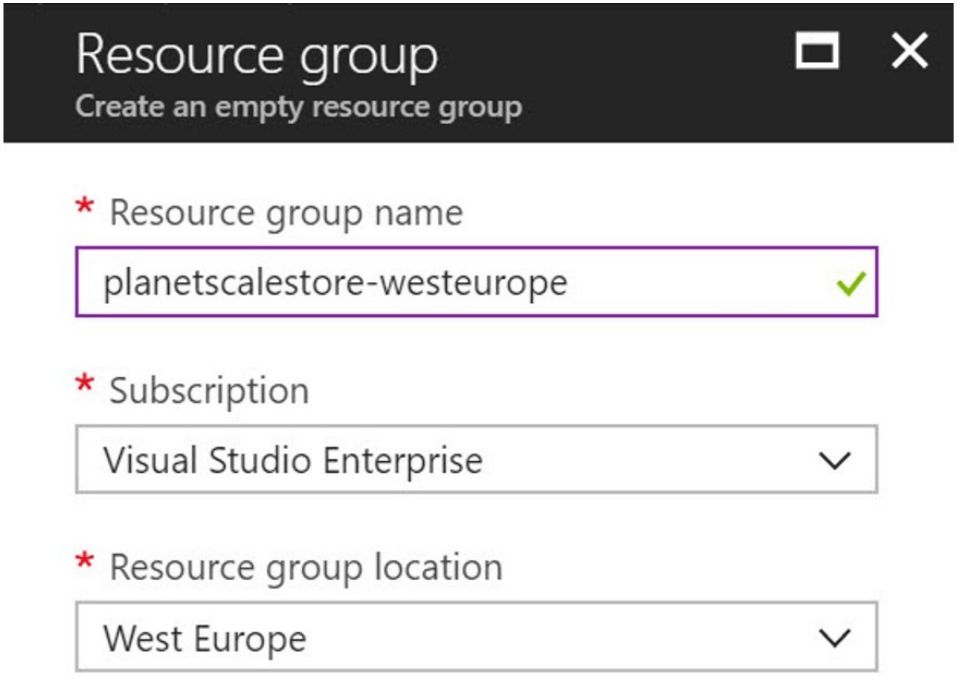
Parent Resource Group - Resources Names

Create Parent Resource Group

In Azure Portal click the `Resource groups` menu item on the left panel and next click the `Add` button. Fill the form as follow:

1. **Resource group name**: <Primary-Name>-westeurope
2. **Subscription**: <Select-Your-Subscription>
3. **Resource group location**: West Europe

Since this is a parent resource group, you name it by the name you selected for primary resource group plus '-' and the region with no spaces and lowercase.



Create Parent Resource Group

Azure Search

Overview

Searching is one the most commonly used features among all types of applications and many times the first customer interaction with the apps. It is unlikely to find an e-commerce app without an input search for finding product items. People use search engines such as Google or Bing every day and they expect to get a similar experience when using your applications. Microsoft's product for search solutions is `Azure Search` which is a `PaaS` fully managed search engine for web and mobile app development. `Azure Search` allows developers incorporate great search experiences into applications without managing infrastructure *(availability, durability, scaling etc..)* or needing to become experts.

`Azure Search` provides a REST API which hides the complexity of information retrieval and can be used by multiple client libraries and programming languages such as `C#`, `Node.js` or `Java`. The following bullet list describes most of the features `Azure Search` service offers.

- **Full text search and text analysis**: Full text search is a primary use case for most search-based apps. Queries can be formulated using a supported syntax. **Simple query syntax** provides logical operators, phrase search operators, suffix operators, precedence operators. **Lucene query syntax** includes all operations in simple syntax, with extensions for fuzzy search, proximity search, term boosting, and regular expressions
- **Data integration**: Azure Search indexes accept data from any source, provided it is submitted as a JSON data structure. Optionally, for supported data sources in Azure, you can use **indexers** to automatically crawl Azure SQL Database, Azure Cosmos DB, or Azure Blob storage to sync your search index's content with your primary data store. Azure Blob indexers can perform *document cracking* for indexing major file formats, including Microsoft Office, PDF, and HTML documents
- **Linguistic analysis**: Analyzers are components used for text processing during indexing and search operations. There are two types. **Custom lexical analyzers** are used for complex search queries using phonetic matching and regular expressions. **Language analyzers** from Lucene or Microsoft are used to intelligently handle language-specific linguistics including verb tenses, gender, irregular plural nouns (for example, 'mouse' vs. 'mice'), word de-compounding, word-breaking (for languages with no spaces), and more

- **Geo-search**: Azure Search processes, filters, and displays geographic locations. It enables users to explore data based on the proximity of a search result to a physical location
- **User experience features**: **Search suggestions** can be enabled for type-ahead queries in a search bar. Actual documents in your index are suggested as users enter partial search input. **Faceted navigation** is enabled through a single query parameter. Azure Search returns a faceted navigation structure you can use as the code behind a categories list, for self-directed filtering (for example, to filter catalog items by price-range or brand). **Filters** can be used to incorporate faceted navigation into your application's UI, enhance query formulation, and filter based on user- or developer-specified criteria. Create filters using the OData syntax. **Hit highlighting** applies text formatting to a matching keyword in search results. You can choose which fields return highlighted snippets. **Sorting** is offered for multiple fields via the index schema and then toggled at query-time with a single search parameter. **Paging** and throttling your search results is straightforward with the finely tuned control that Azure Search offers over your search results
- **Relevance**: **Simple scoring** is a key benefit of Azure Search. Scoring profiles are used to model relevance as a function of values in the documents themselves. For example, you might want newer products or discounted products to appear higher in the search results. You can also build scoring profiles using tags for personalized scoring based on customer search preferences you've tracked and stored separately
- **Monitoring and reporting**: **Search traffic analytics** are collected and analyzed to unlock insights from what users are typing into the search box. Metrics on queries per second, latency, and throttling are captured and reported in portal pages with no additional configuration required. You can also easily monitor index and document counts so that you can adjust capacity as needed
- **Tools for prototyping and inspection**: In the portal, you can use the **Import data wizard** to configure indexers, index designer to stand up an index, and **Search explorer** to test queries and refine scoring profiles. You can also open any index to view its schema
- **Infrastructure**: The **highly available platform** ensures an extremely reliable search service experience. When scaled properly, Azure Search offers a 99.9% SLA. **Fully managed and scalable** as an end-to-end solution, Azure Search requires absolutely no infrastructure management. Your service can be tailored to your needs by scaling in two dimensions to handle more document storage, higher query loads, or both.

There are 4 steps you need to follow in order to incorporate `Azure Search` in your application. First you provision an `Azure Search` service in Microsoft Azure. Next you define a **Search**

Index which is like a database that holds your data and accepts search queries. In this step you provide schema and structure information of your documents to be indexed. Also, you configure how you wish your documents to be indexed that is, which fields should be searchable, filterable, quearable and so on. In the third step you `push` or `pull` data into your search index. With the `push` model you run CRUD document operations using the REST API while using the `pull` model you retrieve data from external data sources. Pulling data is supported through `indexers` and is available for Azure Cosmos DB, Azure SQL Database, Azure Blob Storage, and SQL Server hosted in an Azure VM. Finally, you can start querying your search index using the Azure Search .NET SDK.

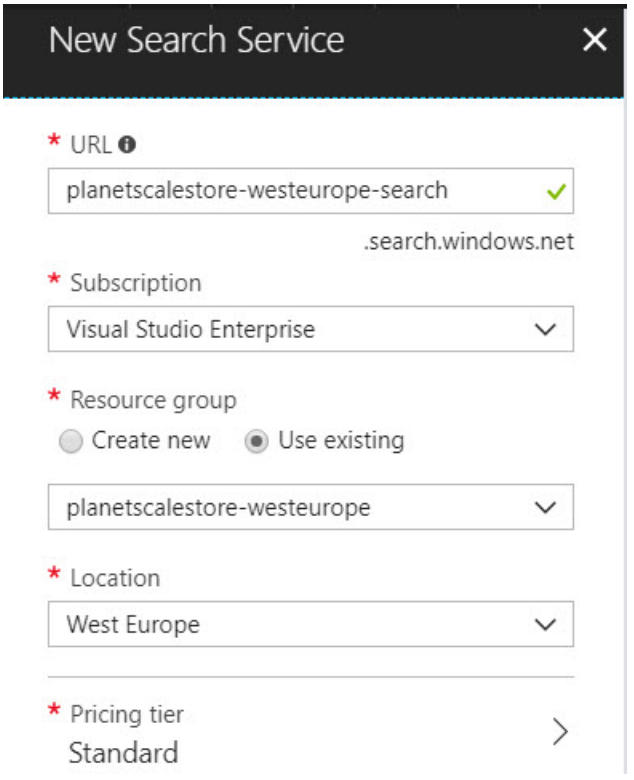
`Online.Store` application uses `Search Service` for searching products in the `Azure Cosmos DB` account. This means that it follows the `pull` model and creates an indexer *bound* to the Azure Cosmos DB `Items` collection that hosts the product JSON documents. The creation of `product-index` index and all query operations are made programmatically using the Azure Search .NET SDK.

Azure Portal

Create Azure Search Service

In Azure Portal click the `New` button on the left panel, search for and select `Azure Search`. Click `Create` and fill the form as follow:

1. **URL**: <Parent-Resource-Group>-search
2. **Subscription**: <Select-Your-Subscription>
3. **Resource group**: Use existing <Parent-Resource-Group>
4. **Location**: West Europe
5. **Pricing tier**: Standard

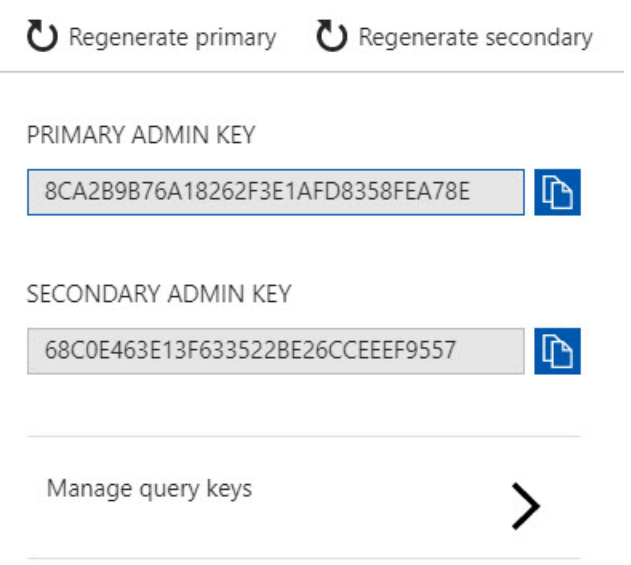


Create Azure Search Service

When the Search Service is created go ahead and open that resource. You can to this either by:

- Select `Resource Groups` from the left menu and then click your parent resource group. Click in the resource of type `Search Service`
- Click `More service` from the left menu, search for and click `Search Services`. The new blade will list all the `Search Services` you have deployed

In the main blade of your `Search Service` resource, click `Keys` and copy the `PRIMARY ADMIN KEY` value.



Search Services keys

Set it as the value for the property *SearchService:ApiKey* in the **secrets.json** file.

```
{
  ...
  "SearchService:ApiKey": "<search-service-primary-key>",
  ...
}
```

The equivalent `dotnet` command is the following:

```
dotnet user-secrets set "SearchService:ApiKey" "<search-service-primary-key>"
```

In **appsettings.json** file set the *SearchService:Name* property equal to your `Azure Search` service name.

```
{
  ...
  "SearchService:Name": "<parent-resource-group>-search",
  ...
}
```

Redis Cache

Overview

`Azure Redis Cache` is a fully managed service for `Redis` *(Redis as a Service)* which is a high-throughput and consistent low-latency data store. With `Redis` engine you can save key-value pairs where keys contain any data structures such as strings, hashes, lists, sets and sorted sets. `Redis` also provides great scalability capabilities such as **master-slave replication** that allows slave Redis servers to be exact copies of master servers. Due to its high throughput and super fast access to data, `Redis` is the preferred backbone component for building `publishing/subscribing` and `Real-Time` applications that need to scale-out. `Azure Redis` comes with the following tiers.

- **Basic**: Single node, multiple sizes, ideal for development/test and non-critical workloads. The Basic tier has no SLA.
- **Standard**: A replicated cache in a two-node primary/secondary configuration managed by Microsoft, with a high-availability SLA.
- **Premium**: All of the Standard tier features, including a high-availability SLA, as well as better performance over Basic and Standard-tier caches, bigger workloads, disaster recovery, enhanced security, and more.

Creating a Redis Cache is very easy and can be done either from Azure Portal or using `PowerShell` and `Azure CLI`. As a managed service, Microsoft can monitor and manage your cache's replication in order to increase its data availability across failures.

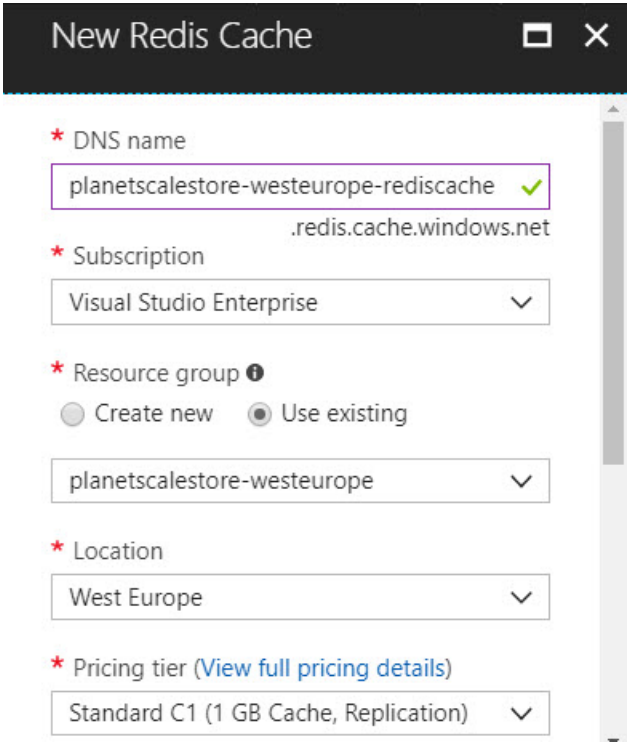
`Online.Store` application uses `Redis Cache` to save cart items before customers checkout. Carts are perfect candidates for `Redis Cache` since they are temporary data which always needs to be accessed as fast as possible. Also, it saves new `orders` temporary, so that are available for display before `WebJobs` and `SQL Geo-Replication` commit them in database.

Azure Portal

Create Redis Cache

In Azure Portal click the `New` button on the left panel, search for and select `Redis Cache`. Click `Create` and fill the form as follow:

1. **DNS name**: <Parent-Resource-Group>-rediscache
2. **Subscription**: <Select-Your-Subscription>
3. **Resource group**: Use existing <Parent-Resource-Group>
4. **Location**: West Europe
5. **Pricing tier**: Standard C1

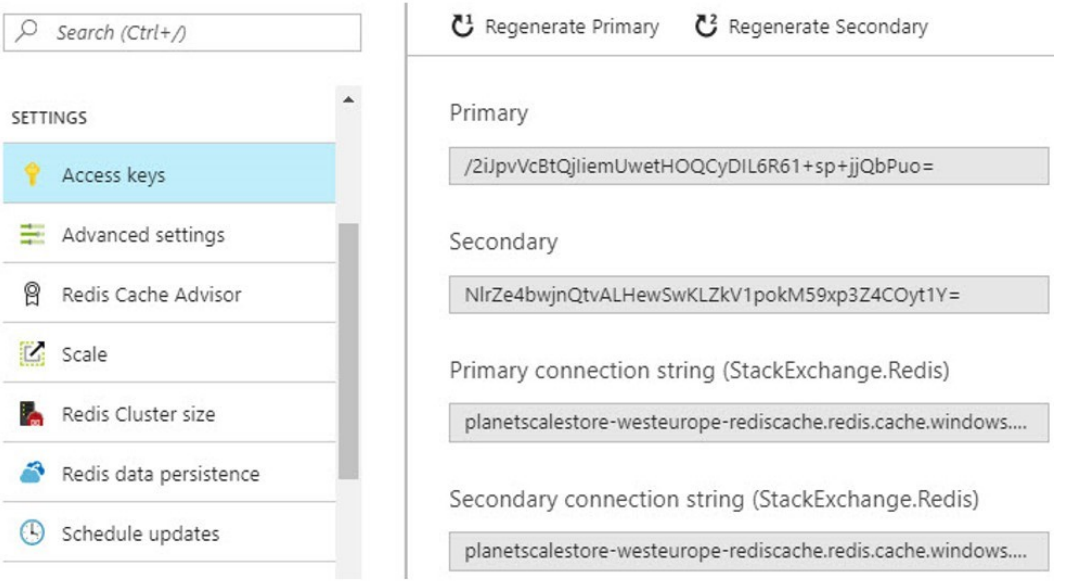


Create Redis Cache

When the Redis Cache is created go ahead and open that resource. You can to this either by:

- Select `Resource Groups` from the left menu and then click your parent resource group. Click in the resource of type `Redis Cache`.
- Click `More services` from the left menu, search for and click `Redis Caches`. The new blade will list all the `Redis Caches` you have deployed.

In the main blade of your `Redis Cache` resource, click `Access keys` and copy the `Primary` key value.



Redis Cache keys

Set it as the value for the property *RedisCache:Key* in the **secrets.json** file.

```
{
  ...
  "RedisCache:Key": "<redis-cache-primary-key>",
  ...
}
```

The equivalent `dotnet` command is the following:

```
dotnet user-secrets set "RedisCache:Key" "<redis-cache-primary-key>"
```

In **appsettings.json** file set the *RedisCache:Endpoint* property equal to your `Redis Cache` name.

```
{
  ...
  "RedisCache:Endpoint": "<parent-resource-group>-rediscache",
  ...
}
```

Service Bus

Overview

`Azure Service Bus` is a Microsoft's cloud `Messaging as a Service` *(MaaS)* solution which allows reliably sharing data between systems that are decoupled. You can think of `Service Bus` as an information delivery service similar to the postal service, where data is being exchanged between decoupled components even if senders and reciepients do not know each other or their current availability status. `Service Bus` offers the following communication options.

- **Queues**: `First in first out` message delivery to one or more reciepients. While many reciepients may get messages from the same queue, each message is only processed once *(one-direction communication)*
- **Topics & Subscriptions**: `Publish and subscribe` approach where a single message can be send to any consumer that is subscribed. Another difference between queues and topics is that in topics subscribers may define a `filter` for receiveing messages, for example a suscriber wishes to receive orders that the property `TotalCost` in the message is greater than 100$ *(content-based routing)*
- **Relays**: Provides bidirectional communication where data can go between the sender and the receiver

Queues, Topics and Relays provide reliable asynchronous communication with great features such as rich temporal control so that you can monitor how messages are flowing in time. The most significant features and events they provide are listed in the following table.

Scheduled delivery	Auto delete on idle
Poison message handling	OnMessage
ForwardTo	Duplicate detection
Defer	Filters
Sessions	Actions
Batching	Transactions

`Online.Store` application uses `Azure Service Bus` queues for decoupling the Web applica-

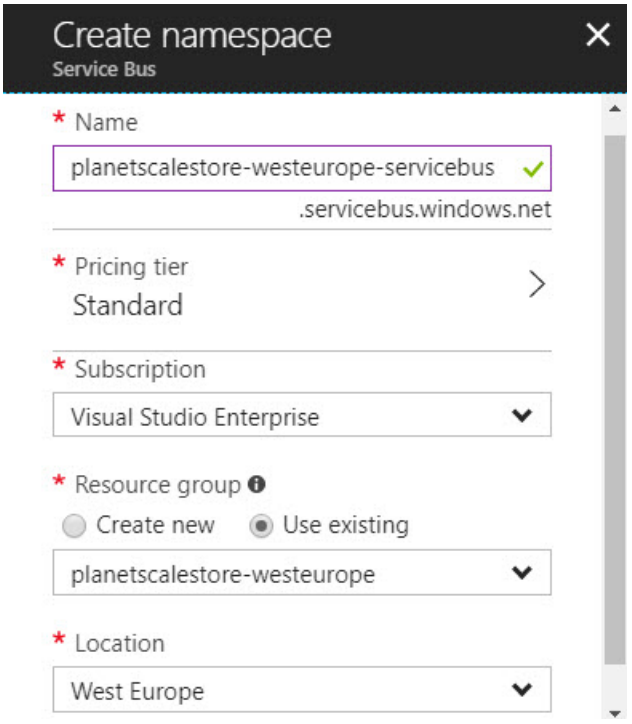
tion from the database `Write` operations. Orders are submitted in a `Service Bus Namespace` queue from the web tier *(sender)* and are received and processed by `WebJobs` asynchronously. Even if the `Read-Write` primary database is offline, the web tier doesn't have to know or even care about. It should continue to function properly and send messages to the queue. On the other side, when the database is back online, `WebJobs` will continue processing the orders as used to.

Azure Portal

Create Service Bus Service

In Azure Portal click the `New` button on the left panel, search for and select `Service Bus`. Click `Create` and fill the form as follow:

1. **Name**: <Parent-Resource-Group>-servicebus
2. **Pricing tier**: Standard
3. **Subscription**: <Select-Your-Subscription>
4. **Resource group**: Use existing <Parent-Resource-Group>
5. **Location**: West Europe



Create Service Bus Namespace

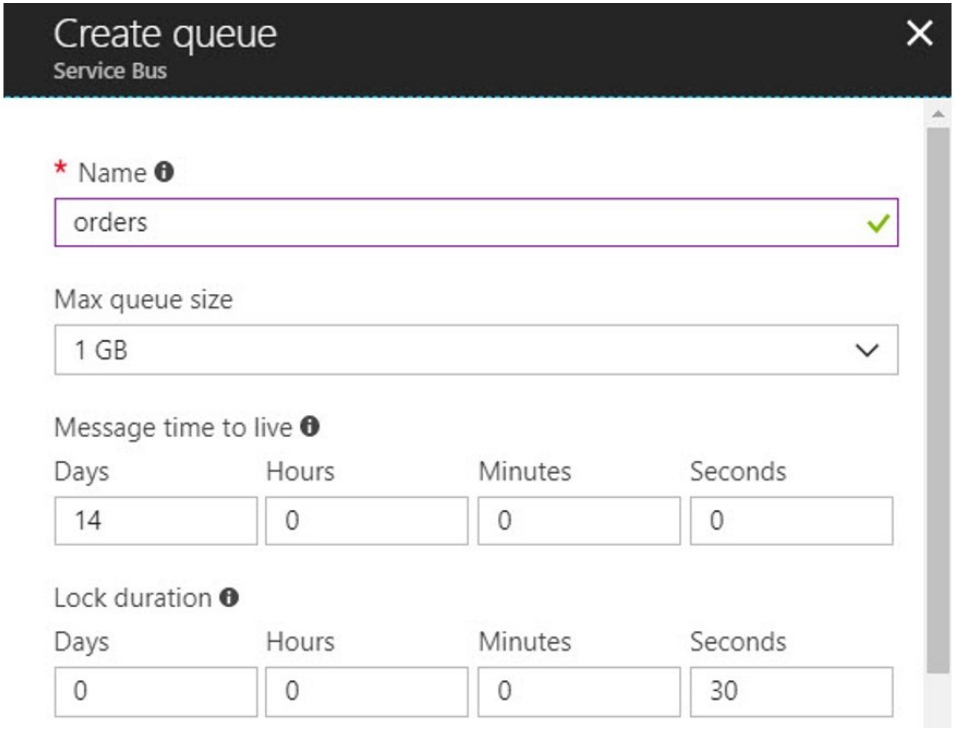
When the Service Bus namespace is created go ahead and open that resource. You can to this either by:

- Select `Resource Groups` from the left menu and then click your parent resource group. Click in the resource of type `Service Bus`
- Click `More services` from the left menu, search for and click `Service Bus`. The new blade will list all the `Service Bus` services you have deployed

In the main blade of your `Service Bus` resource, select `Queues` and then click the `Queue` button with the plus sign. Fill the form as follow to create a queue named *orders*.

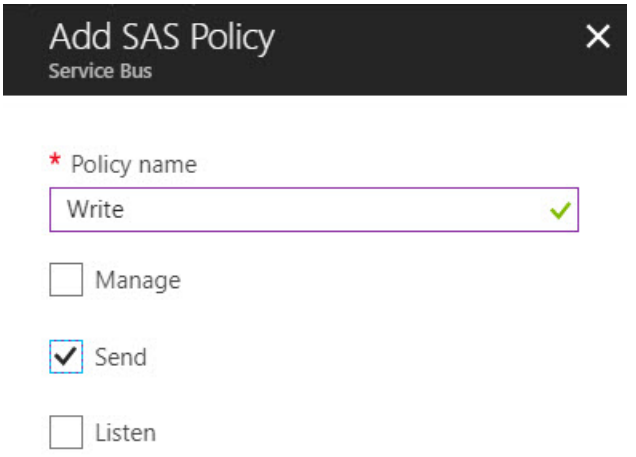
1. **Name**: orders
2. **Max queue size**: 1 GB
3. **Message time to live**: 14 Days 0 (Hours-Minutes-Seconds)
4. **Lock duration**: 30 Seconds

5. **Enable duplicate detection**: Unchecked
6. **Enable dead lettering on message expiration**: Unchecked
7. **Enable sessions**: Unchecked
8. **Enable partitioning**: Checked



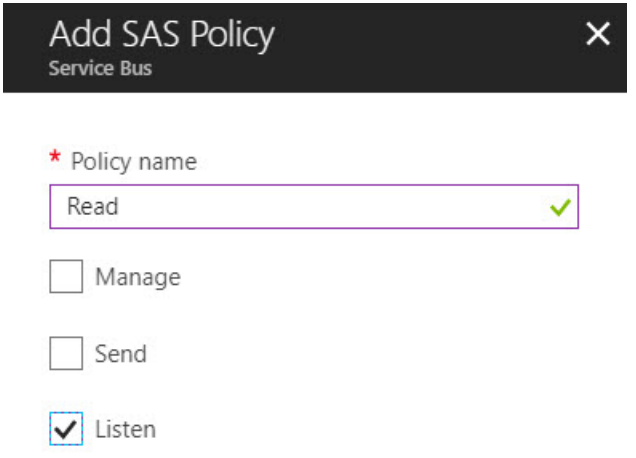
Create Service Bus Queue

When the *orders* queue is created click on it to set the *access policies*. In the *orders* queue blade select `Shared access policies` and click the `Add` button. Create a policy named `Write` and check the `Send` checkbox. This policy will be used from the web application to send messages to the *orders* queue.



Create Write Access Policy

Create another policy named `Read` but this time check the `Listen` checkbox. This policy will be used by **WebJobs** to read messages from the *orders* queue.



Create Read Access Policy

Next click on the **Write** access policy you created before and copy the **Primary Key** value. Set it as the value for the *ServiceBus:WriteAccessKey* property in the **secrets.json** file. Also make sure to add a new property named *ServiceBus:WriteAccessKeyName* with value **Write**.

```
{
  ...
  "ServiceBus:WriteAccessKeyName": "Write",
  "ServiceBus:WriteAccessKey": "<write-access-policy-key>"
  ...
}
```

The equivalent `dotnet` command is the following:

```
dotnet user-secrets set "ServiceBus:WriteAccessKey" "<write-access-policy-key>"
```

In the **appsettings.json** file set the following properties:

```
{
  ...
  "ServiceBus:Namespace": "<parent-resource-group>-servicebus",
  "ServiceBus:Queue": "orders",
  ...
}
```

Code Overview

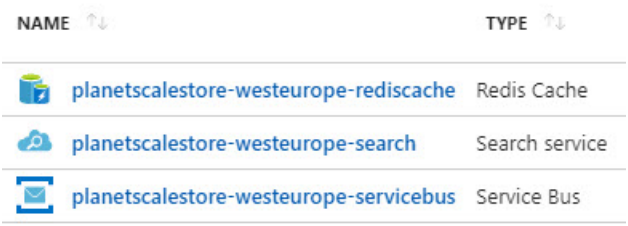
At this point you should have deployed the following Azure Resources:

- Azure Storage Account *(previous part)*
- CDN Profile *(previous part)*
- Azure Cosmos DB Account *(previous part)*
- Azure Search Service
- Redis Cache
- Service Bus

Before running again the `Online.Store` application verify that during this part, you have set the following properties in the configuration files:

- **appsettings.json**
  1. `SearchService:Name`: <parent-resource-group>-search
  2. `RedisCache:Endpoint`: <parent-resource-group>-rediscache
  3. `ServiceBus:Namespace`: <parent-resource-group>-servicebus
  4. `ServiceBus:Queue`: "orders"
- **secrets.json**
  1. `SearchService:ApiKey`: <search-service-primary-key>
  2. `RedisCache:Key`: <redis-cache-primary-key>
  3. `ServiceBus:WriteAccessKeyName`: "Write"
  4. `ServiceBus:WriteAccessKey`: <write-access-policy-key>

In the example images you have seen so far, `planetscalestore` has been used as the Primary Resource Group name. Here is how the resources looks like in the `parent resource group` created in `West Europe`.



Parent Resource Group Resources

Now that you have set the `Search Service` when you run `Online.Store` it will automatically **bind** the search engine with your **Cosmos DB** account. More specifically the app will create an index named *product-index* with a data-source pointing the `Items` collection where the product documents exist. The library project for `Search Service` is the `Online.Store.AzureSearch`. You can access and manage an Azure `Search Service` using the Microsoft.Azure.Search NuGet package. The `ISearchRepository` interface has 3 methods:

```
public interface ISearchRepository
{
    Task CreateOrUpdateDocumentDbDataSourceAsync(string dataSourceName,
                                                 string dataSourceColletion);

    Task CreateOrUpdateIndexAsync<T>(string indexName, string suggesterName,
                                                   List<string> suggesterFields);

    Task<List<T>> SearchAsync<T>(string searchIndex, string term,
                        List<string> returnFields) where T : class;
}
```

`SearchRepository` is an abstract class implementing the `ISearchRepository`. It also has the properties for concrete classes such as `SearchStoreRepository`, to initialize clients.

```
public SearchStoreRepository(IConfiguration configuration)
{
    _documentDbEndpoint = string.Format("https://{0}.documents.azure.com:443/",
                    configuration["DocumentDB:Endpoint"]);
    _documentDbAccountKey = configuration["DocumentDB:Key"];
    _documentDbDatabase = configuration["DocumentDB:DatabaseId"];

    string _azureSearchServiceName = configuration["SearchService:Name"];
    string _azureSearchServiceKey = configuration["SearchService:ApiKey"];

    _serviceClient = new SearchServiceClient(_azureSearchServiceName,
        new SearchCredentials(_azureSearchServiceKey));
}
```

There are 3 parts for binding a search service to a Cosmos DB collection:

1. Create the **datasource**
2. Create the **index**
3. Create the **indexer**

Open `SearchRepository` class to see the implementations for these one by one.

```
public async Task CreateOrUpdateDocumentDbDataSourceAsync(
            string dataSourceName, string dataSourceColletion)
{
    string connectionString =
        string.Format("AccountEndpoint={0};AccountKey={1};Database={2}",
        _documentDbEndpoint, _documentDbAccountKey, _documentDbDatabase);
    DataSourceCredentials credentials =
        new DataSourceCredentials(connectionString);
    DataContainer container = new DataContainer(dataSourceColletion);
    DataSource docDbSource =
        new DataSource(dataSourceName, DataSourceType.DocumentDb,
                    credentials, container);
    await _serviceClient.DataSources.CreateOrUpdateAsync(docDbSource);
}
```

`CreateOrUpdateDocumentDbDataSourceAsync` method needs the Cosmos DB connection string to build the datasource. `CreateOrUpdateIndexAsync` method creates the index which will be used for quering the Search Service.

```
public async Task CreateOrUpdateIndexAsync<T>(
                string indexName, string suggesterName,                    \
                List<string> suggesterFields)
{
    List<Suggester> suggesters = new List<Suggester>();
    suggesters.Add(new Suggester(suggesterName,
        SuggesterSearchMode.AnalyzingInfixMatching, suggesterFields));
    var definition = new Index()
    {
        Name = indexName,
        Fields = FieldBuilder.BuildForType<T>(),
        Suggesters = suggesters
    };

    await _serviceClient.Indexes.CreateOrUpdateAsync(definition);
}
```

This method is generic and can create indexes for any type of component in Cosmos DB. Mind though that type `<T>` should have certain attributes from `Microsoft.Azure.Search` namespace that informs the SDK how to build the index. Check `ProductInfo` in `Online.Store.Core` to view some of those attributes. The following table comes directly from the official documentation for creating indexes.

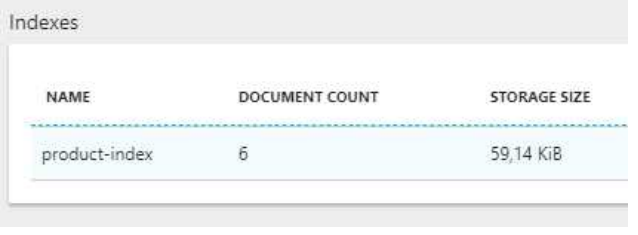
Attribute	Description
searchable	Full-text searchable, subject to lexical analysis such as word-breaking during indexing. If you set a searchable field to a value like "sunny day", internally it will be split into the individual tokens "sunny" and "day".
filterable	Referenced in **$filter** queries. Filterable fields of type `Edm.String` or `Collection(Edm.String)` do not undergo word-breaking, so comparisons are for exact matches only. For example, if you set such a field f to "sunny day", `$filter=f eq 'sunny'` will find no matches, but `$filter=f eq 'sunny day'` will.
sortable	By default the system sorts results by score, but you can configure sort based on fields in the documents. Fields of type `Collection(Edm.String)` cannot be **sortable**.
facetable	Typically used in a presentation of search results that includes a hit count by category (for example, hotels in a specific city). This option cannot be used with fields of type `Edm.GeographyPoint`. Fields of type `Edm.String` that are **filterable**, **sortable**, or **facetable** can be at most 32 kilobytes in length. For details, see Create Index (REST API).

Attribute	Description
key	Unique identifier for documents within the index. Exactly one field must be chosen as the key field and it must be of type `Edm.String`.
retrievable	Determines whether the field can be returned in a search result. This is useful when you want to use a field (such as *profit margin*) as a filter, sorting, or scoring mechanism, but do not want the field to be visible to the end user. This attribute must be `true` for `key` fields.

The last method is the one that creates the **indexer** and completes an index in Azure Service.

```
public async Task CreateOrUpdateIndexerAsync(string indexerName,
                                string dataSource, string index)
{
    await _serviceClient.Indexers.CreateOrUpdateAsync(
            new Indexer(indexerName, dataSource, index));
}
```

After running the project again, go back in the portal and verify that the index has been created successfully. Click the `Search Service` resource in your parent resource group and select `Overview`. You should see the `product-index` having indexed 6 product documents.



Product index

Click on the index and you should see the configuration have been set through code in `ProductInfo` class.

Fields

FIELD NAME	TYPE	ATTRIBUTES
id	Edm.String	Key, Filterable, Retrievable
title	Edm.String	Searchable, Filterable, Retrievable
model	Edm.String	Searchable, Filterable, Retrievable
sku	Edm.String	Retrievable
price	Edm.Double	Retrievable
image	Edm.String	Retrievable
description	Edm.String	Searchable, Retrievable
rating	Edm.Double	Sortable, Retrievable
rates	Edm.Int32	Retrievable
created	Edm.DateTimeOffset	Retrievable

Product index configuration

You should be able to search for products in the app. Try searching the term `sony`.

Search Service results

The library project for `Redis Cache` is the `Online.Store.RedisCahce`. It uses the Caching.Redis NuGet package to connect and access the Azure Redis Cache. The `IRedisCacheRepository` interface declares methods to save items of type *string* or *object*, a generic method to get an item and last but not least a method to remove an item from cache. Items in Cache are saved in key-value pairs.

```
public interface IRedisCacheRepository
{
    Task SetStringAsync(string key, string value);
    Task SetStringAsync(string key, string value, int expirationMinutes);

    Task SetItemAsync(string key, object item);
    Task SetItemAsync(string key, object item, int expirationMinutes);

    Task<T> GetItemAsync<T>(string key);
    Task RemoveAsync(string key);
}
```

`RedisCacheReposistory` is the concrete class that implements the `IRedisCacheRepository` interface. It uses an instance of `IDistributedCache` for accessing the cache. These are the methods available in `IdistributedCache` interface:

```
public interface IDistributedCache
{
    byte[] Get(string key);
    Task<byte[]> GetAsync(string key, CancellationToken token
        = default(CancellationToken));
    void Refresh(string key);
    Task RefreshAsync(string key, CancellationToken token
        = default(CancellationToken));
    void Remove(string key);
    Task RemoveAsync(string key, CancellationToken token
        = default(CancellationToken));
    void Set(string key, byte[] value,
            DistributedCacheEntryOptions options);
    Task SetAsync(string key, byte[] value,
        DistributedCacheEntryOptions options,
        CancellationToken token = default(CancellationToken));
}
```

`SetStringAsync` in `RedisCacheReposistory` saves a `string` value in cache.

```
public async Task SetStringAsync(string key, string value, int expirationMinutes)
{
    var options = new DistributedCacheEntryOptions
    {
        AbsoluteExpiration = DateTime.Now.AddMinutes(expirationMinutes)
    };

    await _distributedCache
            .SetAsync(key, Encoding.UTF8.GetBytes(value), options);
}
```

`SetItemAsync` method uses internaly the same method to save a serializable object.

```
public async Task SetItemAsync(string key, object item)
{
    string json = JsonConvert.SerializeObject(item);

    await SetStringAsync(key, json);
}
```

`GetItemAsync<T>` and `RemoveAsync` methods retrieves and removes an item from cache respectively.

```
public async Task<T> GetItemAsync<T>(string key)
{
    string json = await _distributedCache.GetStringAsync(key);

    if (json == null)
        return default(T);

    return JsonConvert.DeserializeObject<T>(json);
}

public async Task RemoveAsync(string key)
{
    await _distributedCache.RemoveAsync(key);
}
```

The configuration for `Redis Cache` exist in the `Online.Store` **Startup** class.

```
services.AddDistributedRedisCache(option =>
{
    option.Configuration = string.Format(RedisCacheConnStringFormat,
                Configuration["RedisCache:Endpoint"] +
                    ".redis.cache.windows.net:6380",
                Configuration["RedisCache:Key"]);

    option.InstanceName = "master-";
});
```

The instance name will prefix all keys stored in this cache. `Online.Store` application uses the cache to store `cart items`. Check the `AddProductToCart` method inside the `StoreService` class.

```
// code omitted
cart = await _cacheRepository.GetItemAsync<CartDTO>("cart-" + cardId);
// code omitted
await _cacheRepository.SetItemAsync("cart-" + cardId, cart);
```

The *cartId* is a **Guid** generated and saved in browser's cookie to track the cart. Run the web app and add products to cart. Back in the portal open your `Redis Cache` instance and in the main `Overview` blade click `>_Console`. Run the following command to list all cart keys stored in the cache.

```
SCAN 0 COUNT 1000 MATCH *cart*
```

Redis Cache - List cart keys

You can use the `FLUSHALL` command to clear all items in the cache.

The library project for `Service Bus` operations is the `Online.Store.ServiceBus`. It uses the Microsoft.Azure.ServiceBus NuGet package and has a simple `IServiceBusRepository` interface.

```
public interface IServiceBusRepository
{
    void InitQueueClient(string serviceBusAccessKeyName,
                string serviceBusAccessKey, string queue);

    Task SendAsync(object message);
}
```

`ServiceBusRepository` class is the concrete class that implements the interface. It uses a instance of `IQueueClient` to send messages to queues. Following is the implementation for `IServiceBusRepository` repository.

```
public void InitQueueClient(string serviceBusAccessKeyName,
                  string serviceBusAccessKey, string queue)
{
  var connectionString =
    $"Endpoint=sb://{_serviceBusNamespace}" +
    $".servicebus.windows.net/;SharedAccessKeyName={serviceBusAccessKeyName};"+
    $"SharedAccessKey={serviceBusAccessKey}";

    _queueClient = new QueueClient(connectionString, queue);
}

public async Task SendAsync(object body)
{
    var serializedBody = JsonConvert.SerializeObject(body);
    var message = new Message(Encoding.UTF8.GetBytes(serializedBody));
    await _queueClient.SendAsync(message);
}
```

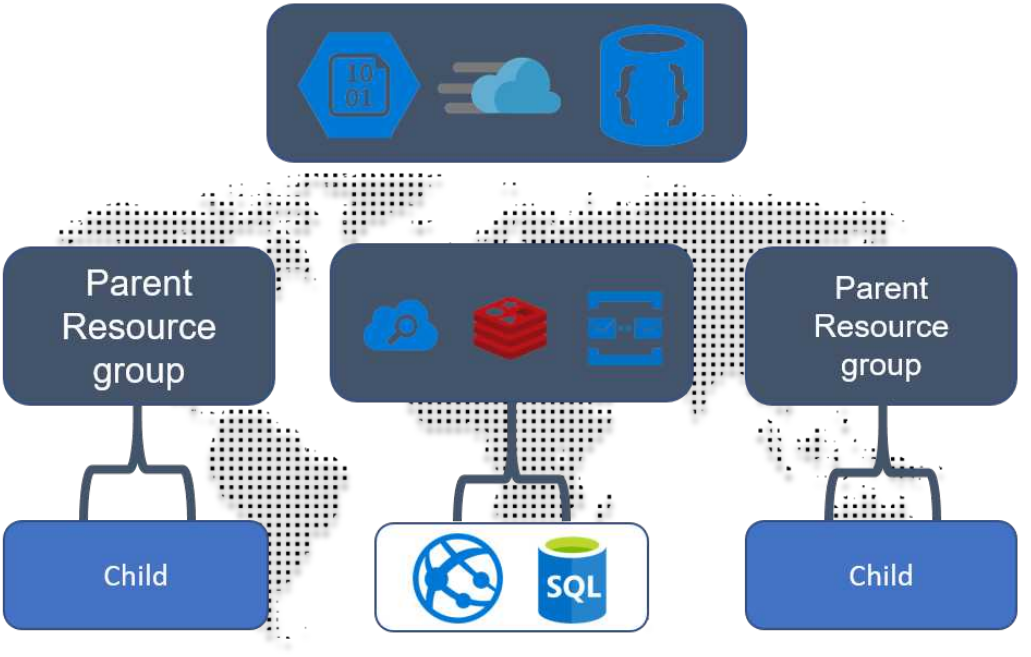
`Online.Store` application uses Service Bus to submit *orders* to the orders queue you created before. It hasn't have to wait till the order is actually saved in database, a **Web Job** will be responsible to complete this task asynchronously. Check the `ServiceBusService` class inside `Online.Store.Azure.Services` project.

```
public async Task SubmitOrderAsync(Order order)
{
    _serviceBusRepository
        .InitQueueClient(_configuration["ServiceBus:WriteAccessKeyName"],
                         _configuration["ServiceBus:WriteAccessKey"],
                         _configuration["ServiceBus:Queue"]);

    await _serviceBusRepository.SendAsync(order);
}
```

At this point you cannot test servive bus operations because you haven't setup **Authentication**, the required **Azure SQL Database** for saving orders and of course the **WebJob** that listens for and processes messages from the *orders* queue. The part that follows covers all these and when you are done you will have a full operational website. Next you will procceed with the **Global Scale** part where you will actually deploy the website at least at two Azure Regions.

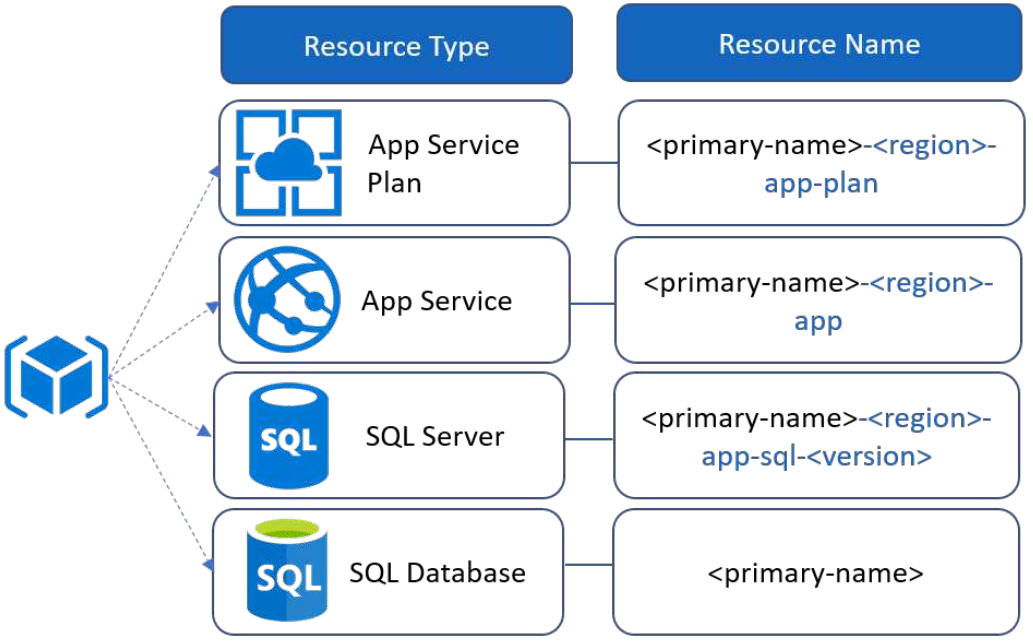
Web Application

In the following sections you are going to create your first `child resource group` in `West Europe`. A `child resource group` contains an `App Service`, an `SQL Server` and an `Azure SQL Database`. Child resource groups are the third level in the resource groups hierarchy and consumes the services contained in the parent resource group provisioned in the same region. You will also select the membership method *(either `ASP.NET Core Identity` or `Azure Active Directory B2C`)* for authenticating users in `Online.Store` application.



Child Resource Groups

Before creating and start adding resources in the child resource group, recall the naming convention you should follow.



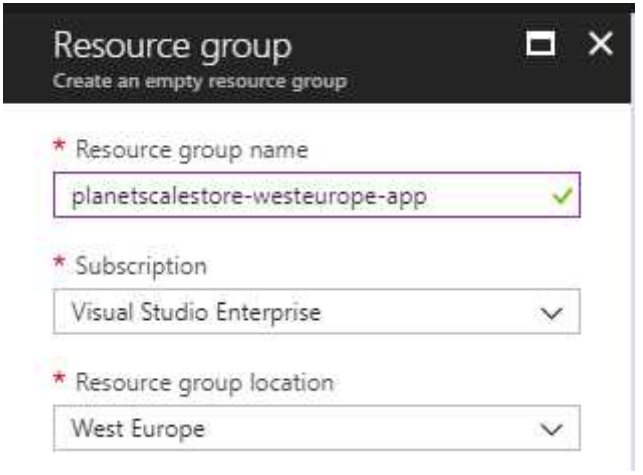
Child Resource Group - Resources Names

Create A Child Resource Group

In Azure Portal click the `Resource groups` menu item on the left panel and next click the `Add` button. Fill the form as follow:

1. **Resource group name**: <Parent-Resource-Group>-app
2. **Subscription**: <Select-Your-Subscription>
3. **Resource group location**: <same as parent's>

A **Child** resource group has the same name as its parent that *'belongs'* to, plus "**-app**". The `app` denotes that this is where the *app* actually exists. The `Resource group location` **must** be the same with the one that the `Parent Resource Group` belongs to.



Create Child Resource Group

Azure SQL Database

Overview

`Azure SQL Database` is a Microsoft's `Platform as a Service` solution. An `Azure SQL Database` is a relational cloud database as a service built for maximizing performance, reliability, scalability and data protection. As fully managed service provides features such as monitoring or alerting and failover options. In case you wonder what is the difference between the traditional `SQL Server` *(Infrastructure as a Service)* and `Azure SQL` the following table makes it clear.

SQL Server (IaaS)	SQL Databases (PaaS)
Manage upgrades and patching	Microsoft manages everything
Configure High Availability	High availability already configured and managed
Manage backups	Built-in backups
Full administrative rights	Administrative rights to the database only
Licensing costs	Eliminates machine costs and SQL licensing
Customizable	Focus on the application layer

`Azure SQL Databases` support `bulk data migrations` where you can transform unstructured and semi-structured flat files, or migrate an entire database schema directly into SQL Database to be used for analytics, reporting, or intelligent applications. You can also **replicate** data among several SQL Databases to isolate mission-critical workloads from longer running analytics queries. `Azure SQL Databases` currently offers the following four tiers.

- **Basic**: Recommended for small databases or for dev and test deployments
- **Standard**: Recommended for low to medium input/output databases with max storage per database `1 TB`
- **Premium**: Designed for IO-intensive transactional volume, high-availability and zero downtime. Microsoft's recommendation for mission-critical databases
- **Premium RS**: Used for intensive workloads where performance is more critical than availability

One of the greatest `Azure SQL Databases` features is the **Elastic Pools** where multiple databases may share a pool of resources and a single database can be configured to consume specific set of resources. This feature is ideal for multitentant scenarios where a single customer has its own database which can be configured to consume more or less resources *(e.g. A customer with lots of clients would require more resources in terms of memory and CPU)*

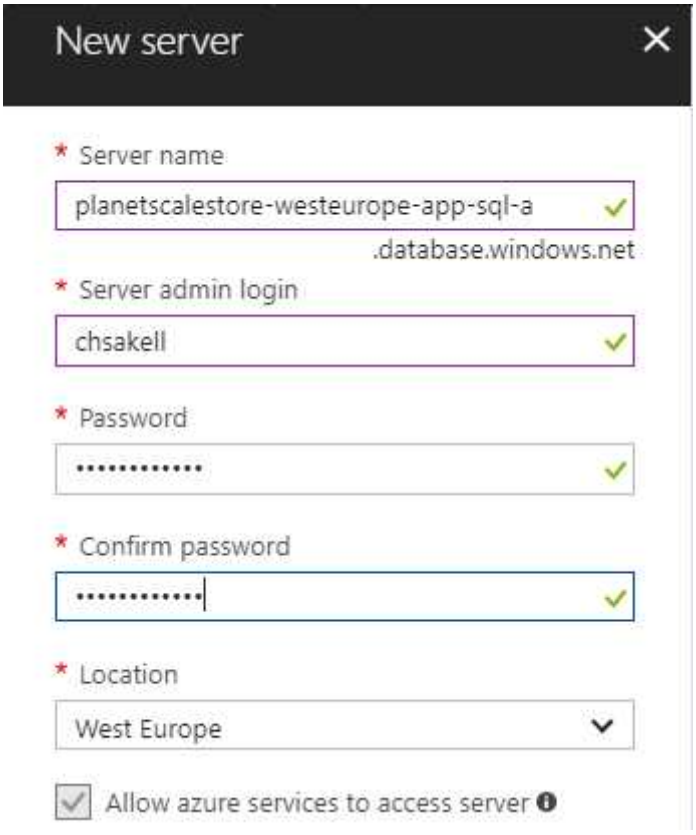
`Online.Store` application uses `Azure SQL Databases` to save relational data such as the orders submitted by customers. There is only one `Read-Write` database in one region *(primary database)* while other regions host `Read-Only` secondary databases being replicated from the primary.

Azure Portal

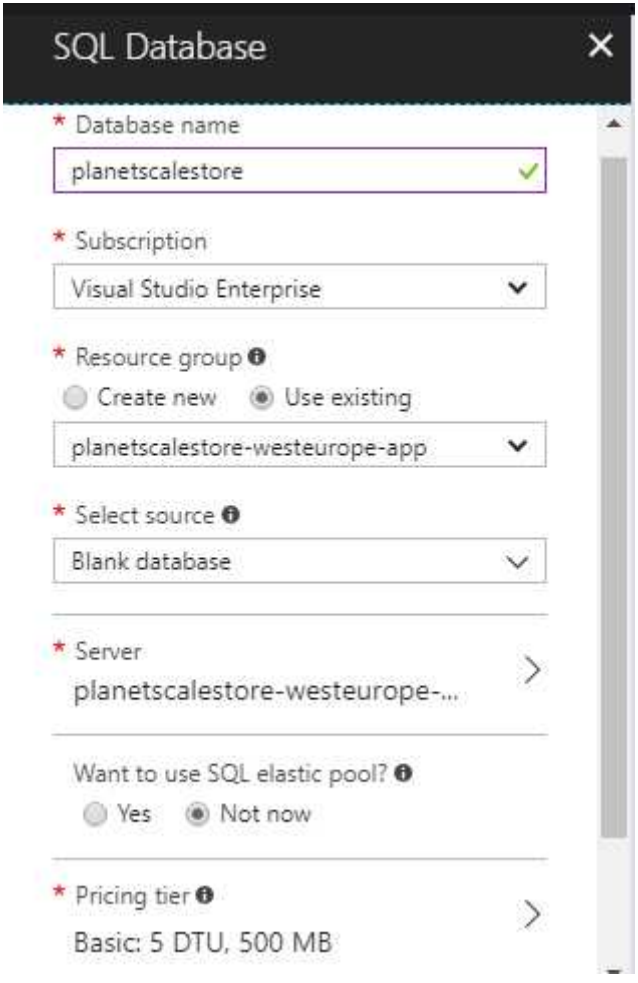
Create SQL Database

In Azure Portal select `SQL Databases` from the left panel, click the `Add` button and fill the form as follow:

1. **Database name**: <Primary-Name>
2. **Subscription**: <Select-Your-Subscription>
3. **Resource group**: Use existing <Child-Resource-Group>
4. **Select source**: Blank database
5. **Configure required settings**: Click to create the server as follow
    - **Server name**: <Child-Resource-Group>-sql-a
    - **Server admin login**: <your-admin-login>
    - **Password**: <your-admin-password>
    - **Confirm password**: <your-admin-password>
    - **Location**: <same-as-parent-location>
    - **Allow azure services to access server**: Checked
6. **Want to use SQL elastic pool?**: Not now
7. **Pricing Tier**: Basic: 5 DTU, 500 MB
8. **Collation**: SQL_Latin1_General_CP1_CI_AS



Create SQL Server



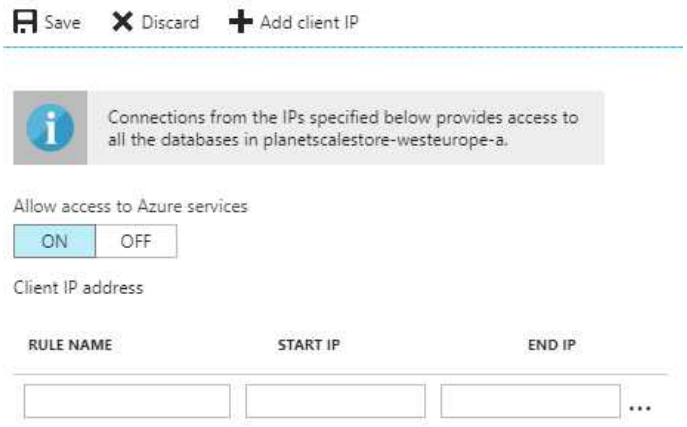
Create SQL Database

Notice that for the `SQL Server's` name other than the prefix *sql* you have also added an **-a** at the end. This means that at some point you will create another one with **-b** at the end. This is something required in the `release process` where only version *a* or only version *b* sql servers will be actually active. Wait till the resources are deployed and when done, open the `SQL Server` resource. You can to this either by:

- Select `Resource Groups` from the left menu and then click your **child** resource group. Find and click the resource of type `SQL Server`
- Click `More services` from the left menu, search for and click `SQL Servers`. The new

blade will list all the SQL Servers you have deployed

In the main blade of your SQL Server, select Firewall / Virtual Networks. Next click Add client IP and Save.



Add Firewall Rule

This will allow you to access the SQL Server from your local machine.

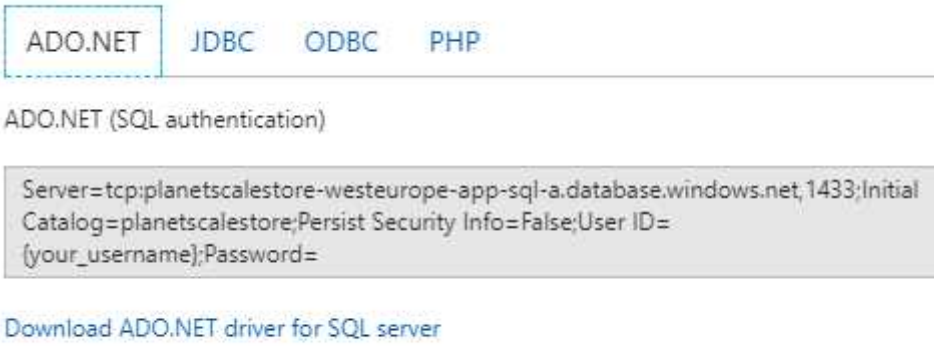
Allow access to Azure services

For all your SQL Servers in Azure, make sure to set Allow access to Azure services to **ON**

Back in the main blade of the SQL Server select SQL Databases and click the database you created before. You can open this resource also by:

- Select Resource Groups from the left menu and then click your **child** resource group. Find and click the resource of type SQL Database
- Click SQL Databases from the left menu. The new blade will list all the SQL Databases you have deployed

In the main blade of the SQL Database select Connection strings and copy the value from the ADO.NET tab.



SQL Database connection string

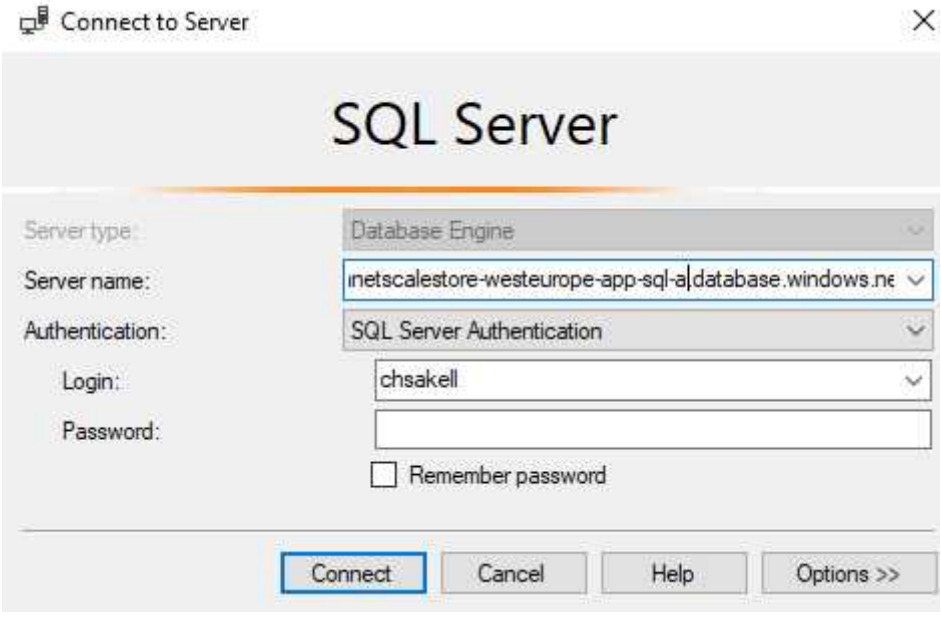
Set this as the value for the *ConnectionStrings:DefaultConnection* in the **secrets.json** file. Make sure to replace the placeholders **{your_username}** and **{your_password}** with those you have setup when created the SQL Server.

```
{
  ...
  "ConnectionStrings:DefaultConnection": "<connection-string>",
  ...
}
```

The next step is to update the **schema** in the database using the **migrations.sql** file which you can find in the `Online.Store` application at *App_Data/db/* folder. You can do this with several ways *(Visual Studio, SQL Server Management Studio, apply migrations)* but the most efficient is through `SQL Server Management Studio`.

Using SQL Server Management Studio

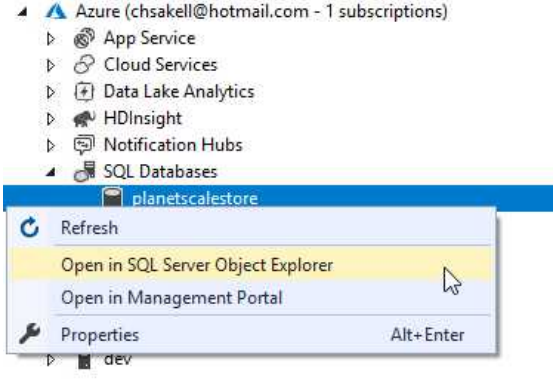
1. Open SQL Server Management Studio and fill the connection window as follow:
2. **Server type**: Database Engine
3. **Server name**: <child-resource-group>-sql-a.database.windows.net
4. **Authentication**: SQL Server Authentication
    - **Login**: <sql-server-login>
    - **Password**: <sql-server-password>
5. Find the database, right click and select `New Query`
6. Paste the contents from `migrations.sql` file and click the **Execute** button



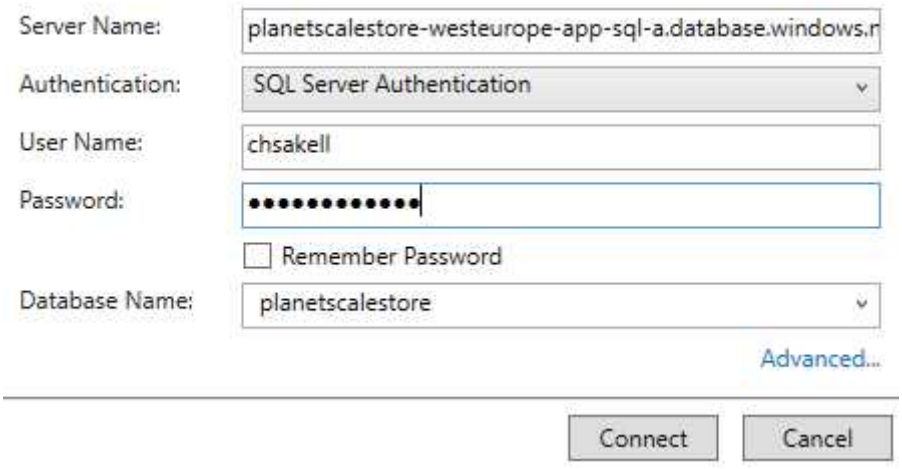
SQL Server Management Studio

Using Visual Studio

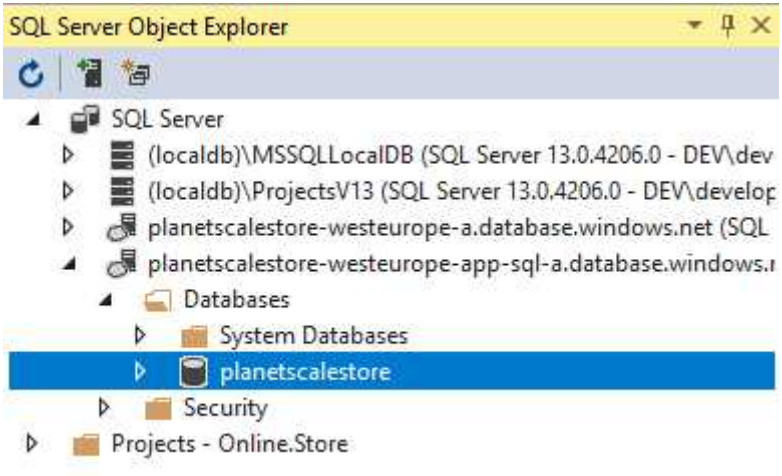
1. Login in Visual Studio with your `Azure Account`
2. Open the `Server Explorer` window. If not visible click *View* and select *Server Explorer*
3. Under `Sql Databases` find and right click the database. Select `Open in SQL Server Object Explorer`
4. If asked enter the credentials you used when created the SQL Server
5. Right click the database and select `New Query`
6. In the editor paste the contents from `migrations.sql` file and click the **Execute** button



Server Explorer



Object Explorer Connection



Object Explorer New Query

ASP.NET Core Identity

Overview

`ASP.NET Core Identity` is an open-source membership system which offers the ability to manage user accounts and implement authentication and authorization. Users can either create a new account and sign in with their custom credentials or use an external login providers. Following are some of the most significan features the framework provides.

- **User accounts**: Create and manage new accounts. Email confirmation, resetting passwords, lockout users and much more related built in features
- **Login and logout**
- **External authentication providers**: Support for `Google`, `Microsoft`, `Facebook`, `Twitter` and others
- **Two-factor authentication**
- **Fully customizable**: Customize the membership system to suit your needs
- **Token-based Authentication support**

`ASP.NET Core Identity` uses the *Claims-Based* security model where a single user may have multiple identities and each identity might allow access to different resources. It comes by default with **Cookie-Based Authentication** where all user's claims are serialized into an encrypted cookie. It uses a `policy-based` model to handle claims authorization where a `policy` is a set of requirements that has be met *(presence of a specific claim or value in the claim)*

`Online.Store` application can be configured to use either `ASP.NET Core Identity` or `Azure Active Directory` for authenticating users. In case you prefer the `Identity` scheme make sure to set the `UseIdentity` property equal to `true` in the **appsettings.json** file of the `Online.Store` web application project.

```
{
  ...
  "UseIdentity": true
}
```

If you choose to use Identity for authenticating users in your application you can safely skip the `Azure Active Directory` section.

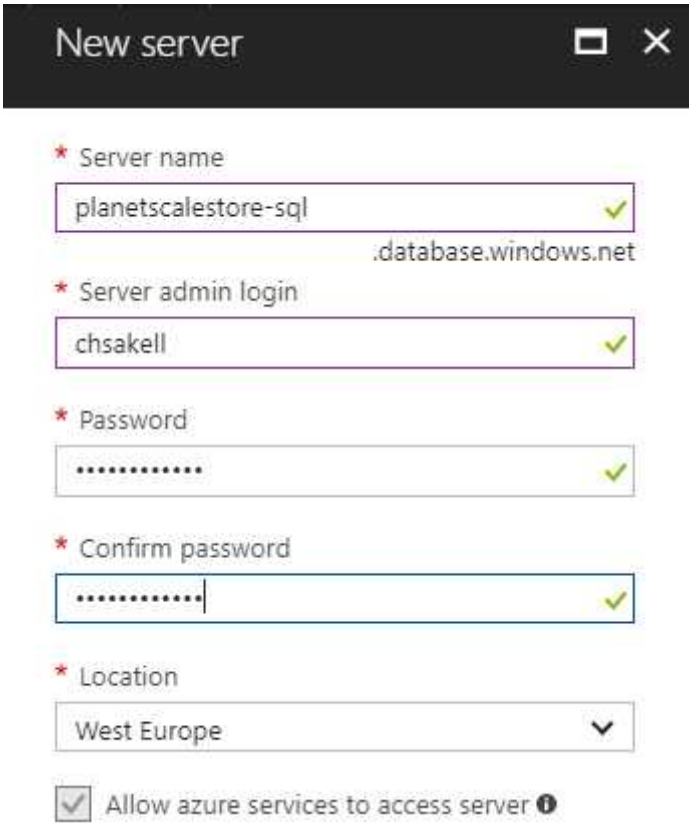
Azure Portal

You will follow the same process to create a database named **identitydb** but this time in your **primary resource group**. All `Online.Store` web app instances deployed around the world use the `identitydb` database for authenticating users, so this is why you add it to the primary resource group.

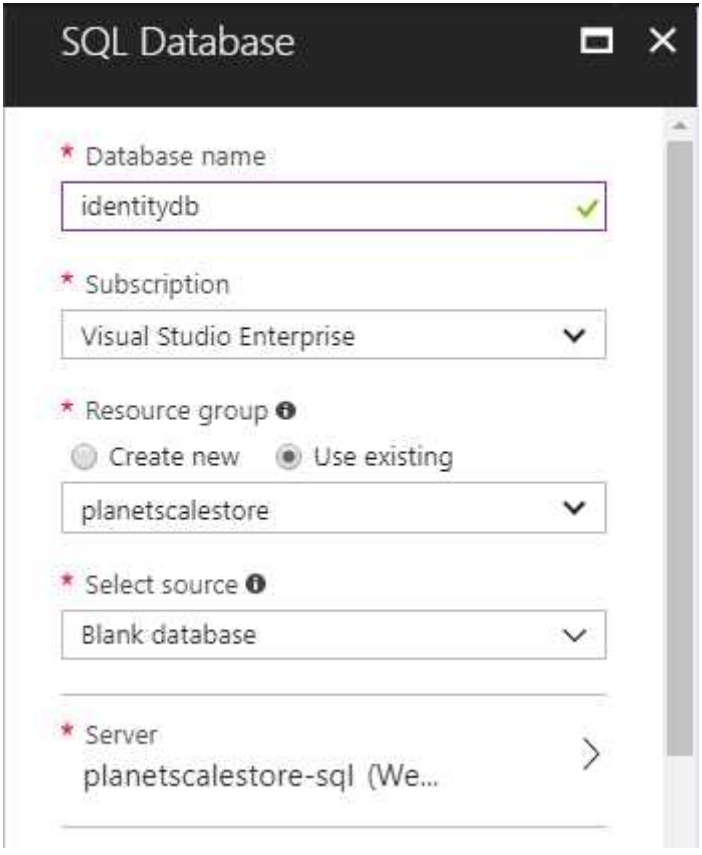
Create Identity SQL Database

In Azure Portal select `SQL Databases` from the left panel and then click the `Add` button and fill the form as follow:

1. **Database name**: identitydb
2. **Subscription**: <Select-Your-Subscription>
3. **Resource group**: Use existing <Primary-Resource-Group>
4. **Select source**: Blank database
5. **Configure required settings**: Click to create a new server as follow
    - **Server name**: <Primary-Name>-sql
    - **Server admin login**: <your-admin-login>
    - **Password**: <your-admin-password>
    - **Confirm password**: <your-admin-password>
    - **Location**: <same-as-primary-location>
    - **Allow azure services to access server**: Checked
6. **Want to use SQL elastic pool?**: Not now
7. **Pricing Tier**: Basic: 5 DTU, 500 MB
8. **Collation**: SQL_Latin1_General_CP1_CI_AS



Primary Resource Group SQL Server



Primary Resource Group identitydb SQL Database

When the new resources are provisioned, follow the same steps you followed in the previous section and add a `Firewall Rule` for your `Client IP` in the new SQL Server created in the primary resource group. Next find the **connection string** for the `identitydb` database following the same steps you followed for the previous database *(but this time for identitydb)*. Set that value as the value for the *ConnectionStrings:IdentityConnection* in the **secrets.json** file. Make sure to replace the placeholders **{your_username}** and **{your_password}** with those you have setup when created the primary SQL Server.

```
{
  ...
  "ConnectionStrings:DefaultConnection": "<connection-string>",
  "ConnectionStrings:IdentityConnection": "<identity-connection-string>",
  ...
}
```

The last thing remained to complete the `ASP.NET Identity` configuration is to update the schema in the `identitydb` database. Using either `SQL Server Management Studio` or `Visual Studio` open a connection to the `identitydb` database on your new SQL Server and run the **identity-migrations.sql** query which you can find in the `Online.Store/App_Data/db` folder. At this point, you should be able to register an account in the `Online.Store` application. After registering the account, login using your **username** and **password** set during registration.

Azure Active Directory B2C

Overview

`Azure Active Directory Business-to-Consumer` is a cloud, global identity service for your consumer facing applications across all platforms web, mobile or desktop. Consumers have the choice to use either their **social accounts** such as `Facebook`, `Google`, `Microsoft`, `Amazon` or `LinkedIn` or create new credentials named **local accounts** to access your applications. A third option is to use open standard protocols, OpenID Connect or SAML **(Enterprise Accounts)**.

In terms of availabilty, `Azure Active Directory B2C` service is backed by a global highly available and secure cloud identity service and directory and can serve hundreds of millions of users per day. While `Azure AD B2C` manages your consumer identities for you, you own your consumers data and have the full control of the branding and experiences they go through. You can actually manage your user accounts programatically using the Azure AD Graph API. It's a `pay-as-you-go` service which means you only pay for what you use.

The process for using `Azure Active Directory B2C` starts by creating a `Azure AD B2C tenant`. Next you register your applications and configure them how you wish to be accesed. `Online.Store` application gives you the choice to use `Azure AD B2C` **local accounts** for authenticating users. If you wish to follow this membership scheme you need to set `UseIdentity` property equal to `false` in the **appsettings.json** file of the `Online.Store` web application project.

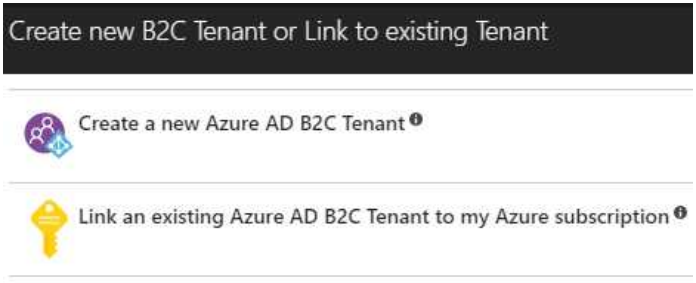
```
{
  ...
  "UseIdentity": false
}
```

Azure Portal

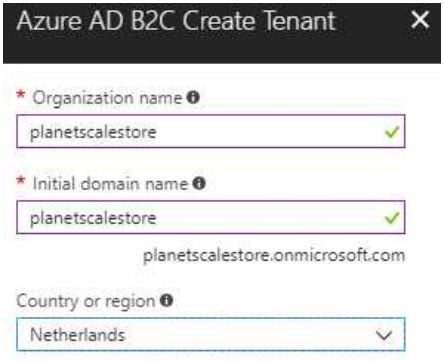
Create Azure Active Directory B2C

In Azure Portal click `New` from the left panel, search for and select `Azure Active Directory B2C`. In the new blade select `Create a new Azure AD B2C Tenant` and fill the form as follow:

1. **Organization name**: <Primary-Name> *(or name it as you wish)*
2. **Initial domain name**: <Primary-Resource-Group>
3. **Country or region**: Choose as you wish. The choice will determine the datacenter for your directory.

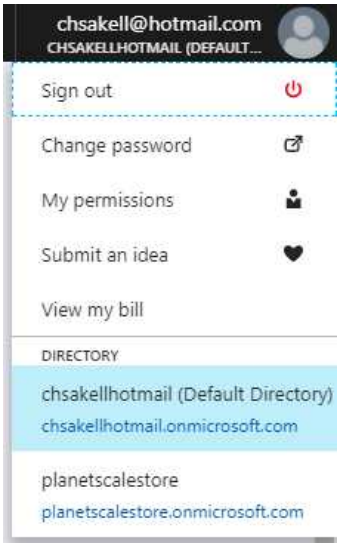


Create Azure Active Directory



Azure Active Directory Details

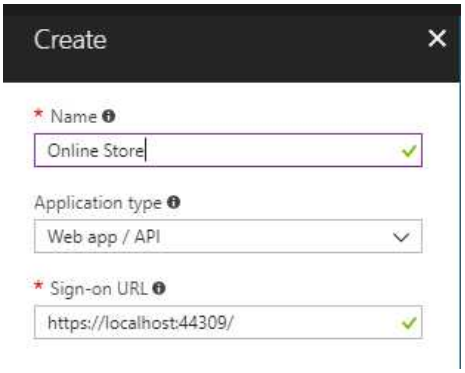
When your new Active Directory is ready, sign out and sign in again to Azure Portal. Click on your email on the upper right and you will see that there is a new `Azure Active Directory` entry. Click it and you will switch to that directory.



Switch Azure Active Directory

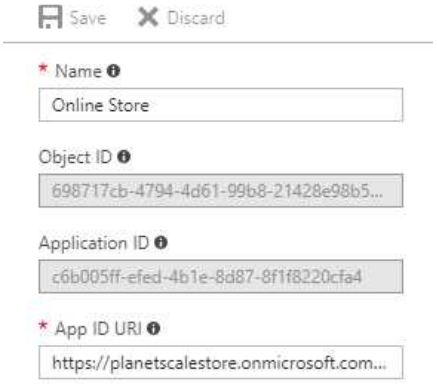
Register the Online.Store application

1. Select `Azure Active Directory` from the left menu and in the new blade click `App registrations`
2. Click `New application registration`, fill the form as follow and click *Create*:
    - **Name**: Online Store
    - **Application type**: Web app / API
    - **Sign-on URL**: https://localhost:44309/



Application Registration

When the application is created click on it and in the `Settings` blade select `Properties`.



Application Properties

Copy the `Application ID` value and set is as the value for *AzureAd:ClientId* property in the **appsettings.json** file.

```
{
 {
  "AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    "Domain": "",
    "TenantId": "",
    "ClientId": "<application-id>",
    "CallbackPath": "/signin-oidc"
  }
  ...
}
```

From the `App ID URI` field, copy the part till *onmicrosoft.com* but without the *https://* part *(e.g. planetscalestore.onmicrosoft.com)*. Set this as the value for the *AzureAd:Domain* property in the **appsettings.json** file.

```
{
 {
  "AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    "Domain": "<paste-value-here>",
    "TenantId": "",
    "ClientId": "<application-id>",
    "CallbackPath": "/signin-oidc"
  }
  ...
}
```

Click `Reply URLs`, add the value **https://localhost:44309/signin-oidc** and click save.

Warning

This is a crusial part to remember. Any web app that needs to authenticate using this `Azure Directory Application` needs to be listed here. Port **:44309** is used when you deploy the app from Visual Studio but when you run it through the .NET CLI it uses a different port. For example if port *:57278* is used you need to add `https://localhost:57278/signin-oidc` in the list as well. When you deploy the app in Azure, each App Service needs to be added on the list as well.

Back in the `Settings` blade select `Keys` and create a key as follow:

- **Description**: app login
- **Expires**: Never expires

Click `Save` and wait for the new key to be created. Copy the key value and set is as the value for the *AzureAd:ClientSecret* in the **secrets.json** file.

```
{
  ...
 "AzureAd:ClientSecret": "<paste-key-here>",
  ...
}
```

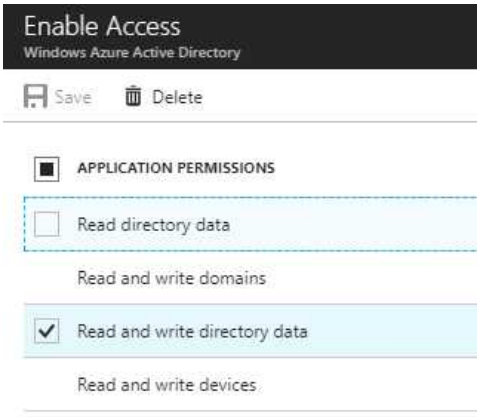
The equivalent `dotnet` command is the following:

```
dotnet user-secrets set "AzureAd:ClientSecret" "<paste-key-here>"
```

Back in the main blade of the new `Azure Active Directory` *(just click Azure Active Directory in the left panel)*, select `Properties`. Copy the `Directory ID` value and set it as the value for the property *AzureAd:TenantId* in the **appsettings.json** file.

```
{
 {
  "AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    "TenantId": "<directory-id>",
    ..
  }
  ...
}
```

The last thing remained to complete the `Azure Active Directory` configuration is to **Grant Permissions** to your app so that it can Read and Write users to directory. In the main blade of the directory select `Required persmissions`. There should be already an entry of API type `Windows Azure Active Directory`. Click this entry and in the new blade under the *APPLICATION PERMISSIONS* section, check **Read and write directory data** and click save.

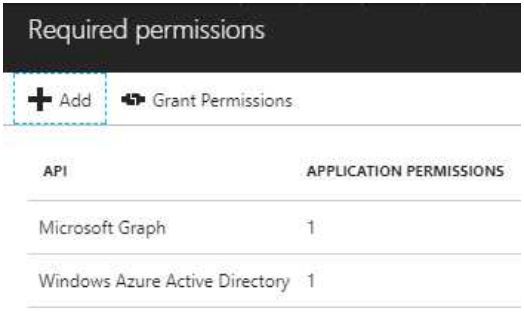


Application Required Permissions

Repeat the same but this time for a new `API` of type **Microsoft Graph**.

1. In the `Required Permissions` blade click `Add`
2. Click `Select an API` and click `Microsoft Graph`
3. In `Select permissions` blade check **Read and write directory data**

After saving the permissions you have to grant them as well. It usually takes a while for the permissions to take effect.



Application Grant Permissions

At this point you should be able to **login** in `Online.Store`. Keep in mind that there is already a user registered, having the same email and password with the one you used for signing in Azure Portal. In case you are already logged in in the portal with that user and you try to

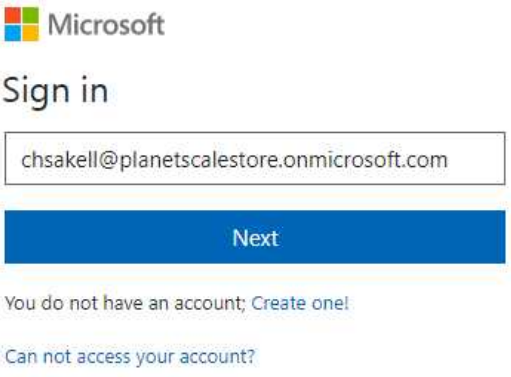
signin in `Online.Store`, you will automatically login in the app as well. Try the following scenario to confirm that you have setup `Azure Active Directory` correctly.

Register and Sign in with a new account

1. Open the app in an `incognito` window, or make sure to logout from Azure Portal, clear your browser's cache and cookies
2. Click the `Account` button on the upper right to register a new user as follow:
   - **Email**: Enter an email. The email MUST end with **@<AzureAd:TenantId>**. You have set this value in **appsettings.json** file before. For example if `AzureAd:TenantId` is *planetscalestore.onmicrosoft.com* then a valid email address for registration form would be `chsakell@planetscalestore.onmicrosoft.com`
   - **Username**: Enter a username
   - **Password**: Enter your password
   - **Confirm password**: Enter your password

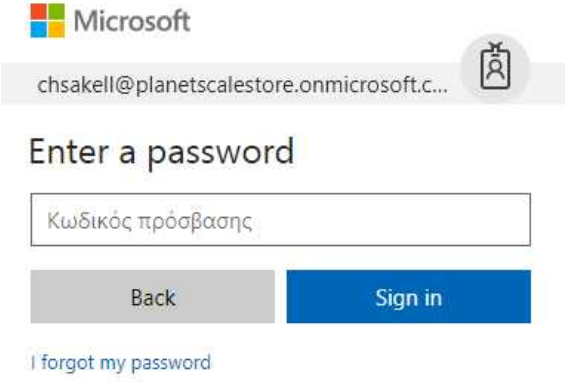
User Registration

Uppon successfull registration you will automatically redirected to a Microsoft's signin page. If not just click the `Sign in` button in the upper right.



Application Sign in - User Email

Enter your password and you will be signed in `Online.Store` app.

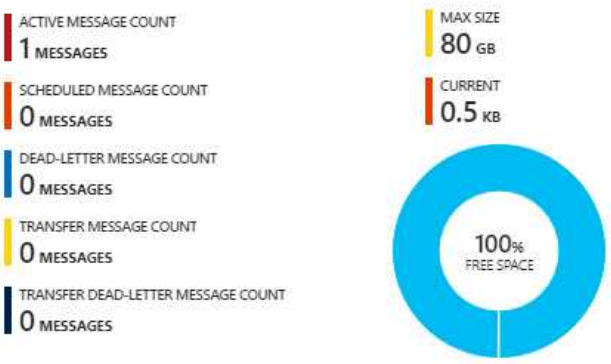


Application Sign in - User Password

After signing in `Online.Store` application, add some products in the cart and finally try to checkout. When you click the `Checkout` button the order will be submited to the `Service Bus` *orders* queue. Go to that resource and confirm that the message has been sent.

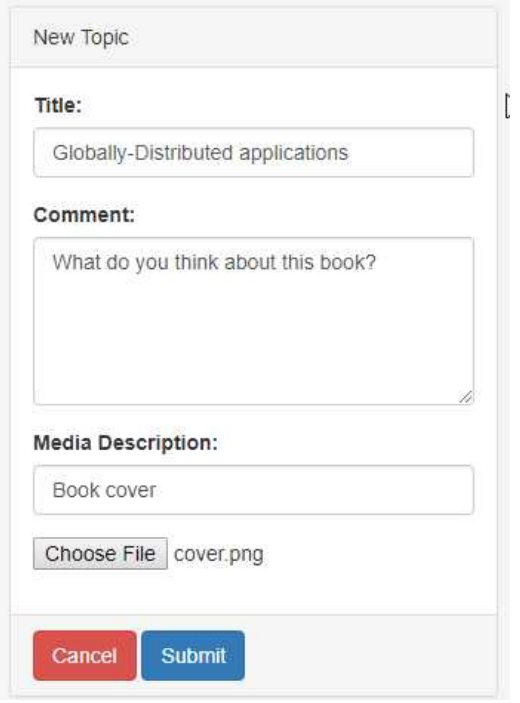
Azure Portal Resources

You won't found the resources you created till now in the new `Azure Active Directory`. You need to switch back to your default directory first.



Orders queue Status

You orders will not appear till you finish the `Web Job` section where will create a Web Job that listens for messages in the *orders* queue and submits them in the database. Another thing you can test now is creating topics. Try to create one and optionally submit an image. Topics and their responses are stored in `Azure Cosmos DB` **Forum** collection.



Create Topic

WebJobs

Overview

`Azure App Services` have a feature named `WebJobs` which allow you to run background tasks in the same context as a web app, API app, or mobile app. A `WebJob` can be an executable or a script with one of the following file types:

- .cmd, .bat, .exe (using Windows cmd)
- .ps1 (using PowerShell)
- .sh (using Bash)
- .php (using PHP)
- .py (using Python)
- .js (using Node.js)
- .jar (using Java)

To add a `WebJob` in an App Service, you simply upload a **zip** file containing your executable or script file to your `App Service`. There are two different types for WebJobs, *continuous* and *triggered*:

Continuous	Triggered
Starts immediately when the WebJob is created. To keep the job from ending, the program or script typically does its work inside an endless loop. If the job does end, you can restart it.	Starts only when triggered manually or on a schedule.
Runs on all instances that the web app runs on. You can optionally restrict the WebJob to a single instance.	Runs on a single instance that Azure selects for load balancing.
Supports remote debugging.	Doesn't support remote debugging.

You can also monitor your `WebJobs` status in the relevant blade of your `App Service`. Any

logs your executable or scrips produce, appear on the selected `WebJob` screen. `Online.Store` application uses `WebJobs` to listen for order messages from the `Service Bus` deployed in the same region with the `App Service`.

Local configuration

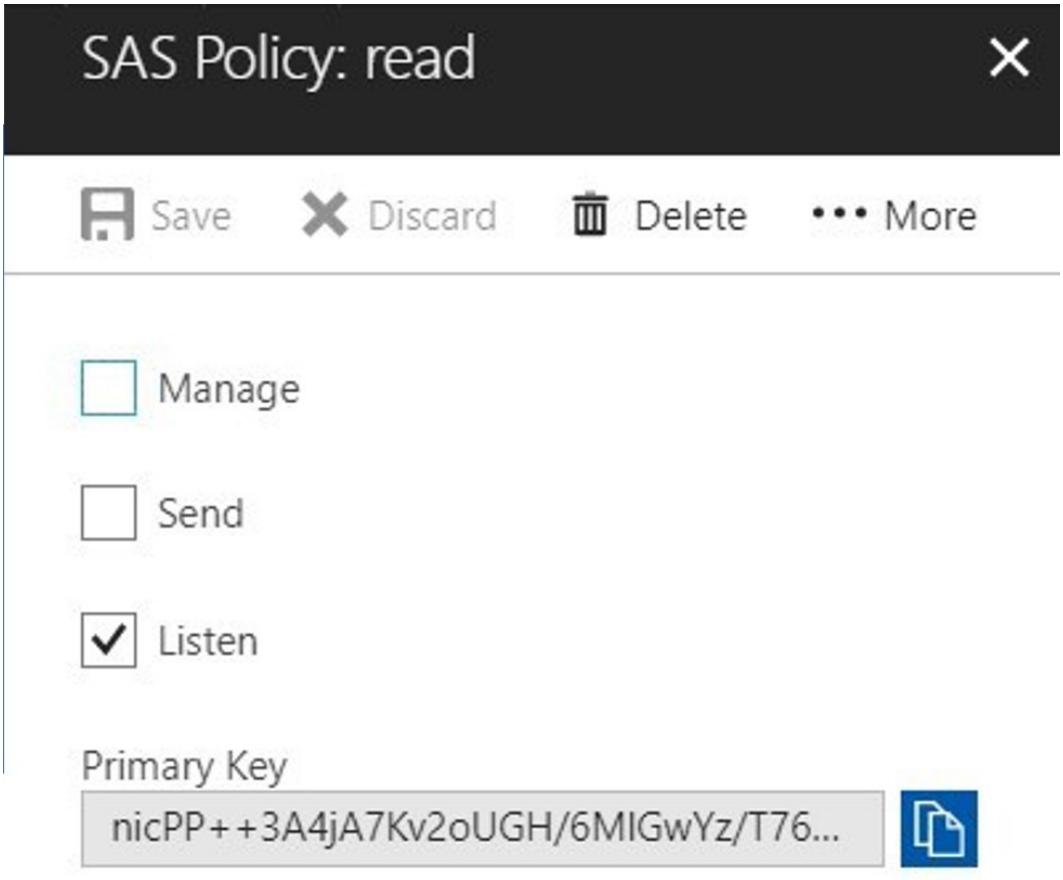
The WebJob library project for submitting orders in database is the `Online.Store.WebJob`. It's a simple .NET Core console application project that uses the Microsoft.Azure.ServiceBus NuGet package to listen messages on the Service Bus *orders* queue. The reason you see a **appsettings.json.template** file in this project and not the usual `appsettings.json` is that the latter shouldn't tracked by the source control. In contrast with the `Online.Store` web project you cannot use a `secrets.json` file or use the **application settings** stored in the App Service's settings *(this is where you save secrets in Azure Portal, more on this on the next section)*. Add a new file named **appsettings.json** in the `Online.Store.WebJob` project and copy the contents from the *appsettings.json.template*.

```
{
  "ConnectionStrings": {
    "DefaultConnection": ""
  },
  "ServiceBus:Namespace": "",
  "ServiceBus:Queue": "orders",
  "ServiceBus:ReadAccessKeyName": "Read",
  "ServiceBus:ReadAccessKey": ""
}
```

Set the above configuration properties as follow:

WebJob appsettings.json configuration

1. **ConnectionStrings:DefaultConnection**: Same connection string you used in the **secrets.json** file for `Online.Store` web application. It's the connection string to the database you created
2. **ServiceBus:Namespace**: <parent-resource-group>-servicebus. It's the same value you used for the same property in the **appsettings.json** file for `Online.Store` web application.
3. **ServiceBus:ReadAccessKey**: In Azure Portal open the *orders* queue in the `Service Bus` namespace you created in **Performance Optimization** part. Select and view the `Shared access policies` for the queue. Recall that you had created two access policies, one named `Write` for which you used its Primary Key in the `Online.Store` application to send messages to the orders queue. Now you need to get the Primary Key for the **Read** access policy in order to read messages from the queue. Click in the `Read` policy and copy the **Primary Key**. Set it as the value for this property



Service Bus Read access policy Primary Key

Your webjob's `appsettings.json` file should look like the following *(e.g. for planetscale-store)*:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=tcp:planetscalestore-westeurope-app-sql-a.database..\
."
  },
  "ServiceBus:Namespace": "planetscalestore-westeurope-servicebus",
  "ServiceBus:Queue": "orders",
  "ServiceBus:ReadAccessKeyName": "Read",
  "ServiceBus:ReadAccessKey": "N0Lxa5YkHiHF3eXAEqDu2n8hMHMCckoUPi7ntxvND0k="
}
```

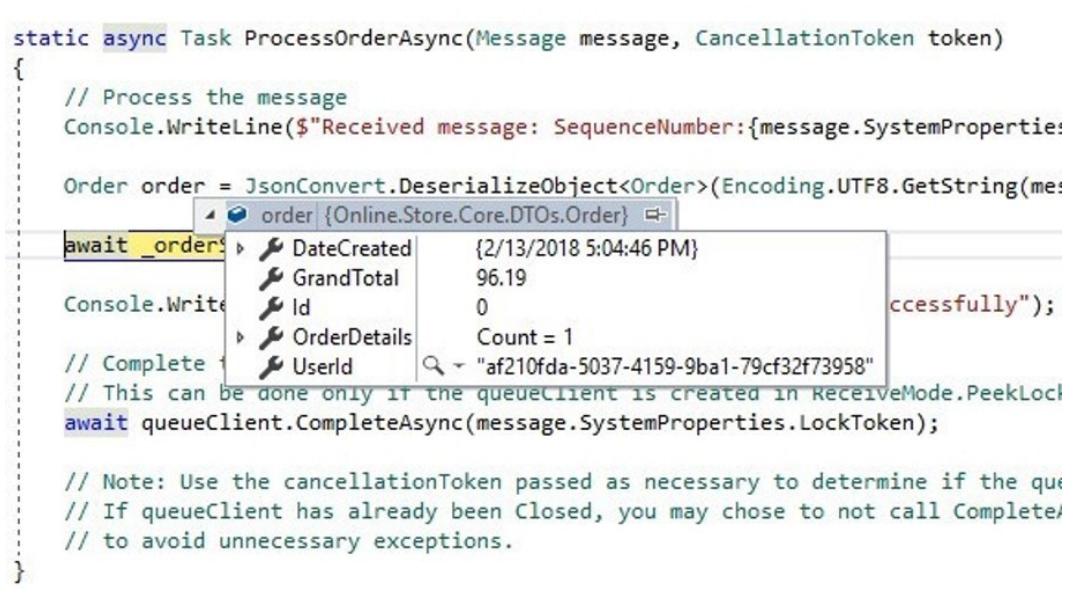
The console application has a simple `OrderService` class to save orders in the database. It uses the `ApplicationDbContext` **Entity Framework** context class referenced from the `Online.Store.SqlServer` project.

```
public class OrderService : IOrderService
{
    ApplicationDbContext _context;

    public OrderService(ApplicationDbContext context)
    {
        _context = context;
    }
    public async Task AddOrderAsync(Order order)
    {
        _context.Orders.Add(order);
        await _context.SaveChangesAsync();
    }
}
```

In `Program.cs` an instance of `QueueClient` is used to register a **message handler** for receiving Service Bus queue messages.

```
static void StartReceivingOrders(string queue)
{
    queueClient = new QueueClient(_serviceBusConnString, queue);

    // Configure the MessageHandler Options in terms of exception handling,
    // number of concurrent messages to deliver etc.
    var messageHandlerOptions
        = new MessageHandlerOptions(ExceptionReceivedHandler)
    {
        // Maximum number of Concurrent calls to the callback
        // `ProcessMessagesAsync`, set to 1 for simplicity. Set it according
        //  to how many messages the application wants to process in parallel.
        MaxConcurrentCalls = 1,

        // Indicates whether MessagePump should automatically complete
        // the messages after returning from User Callback.
        // False below indicates the Complete will be handled by the
        // User Callback as in `ProcessMessagesAsync` below.
        AutoComplete = false
    };

    // Register the function that will process orders
    queueClient.RegisterMessageHandler(ProcessOrderAsync, messageHandlerOptions);
}
```

The proccessing method **deserializes** the message from the queue, saves it in the database and finally removes the message from the queue.



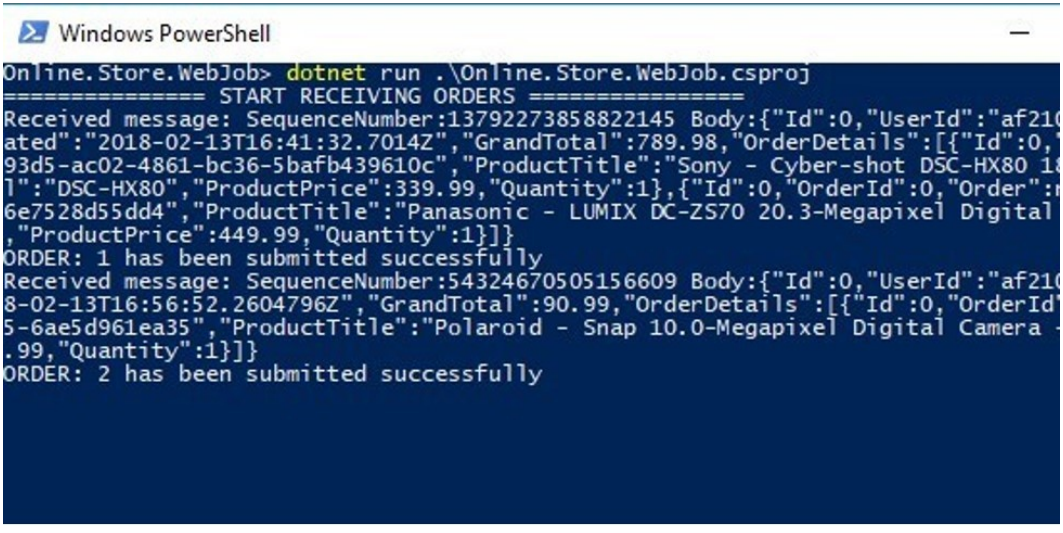
WebJob - Receive messages

```
static async Task ProcessOrderAsync(Message message, CancellationToken token)
{
    // Process the message
    Order order = JsonConvert
        .DeserializeObject<Order>(Encoding.UTF8.GetString(message.Body));

    await _orderService.AddOrderAsync(order);

    // Complete the message so that it is not received again.
    // This can be done only if the queueClient is created in
    // ReceiveMode.PeekLock mode (which is default).
    await queueClient.CompleteAsync(message.SystemProperties.LockToken);
}
```

Run `Online.Store.WebJob` console app using one of the following methods:

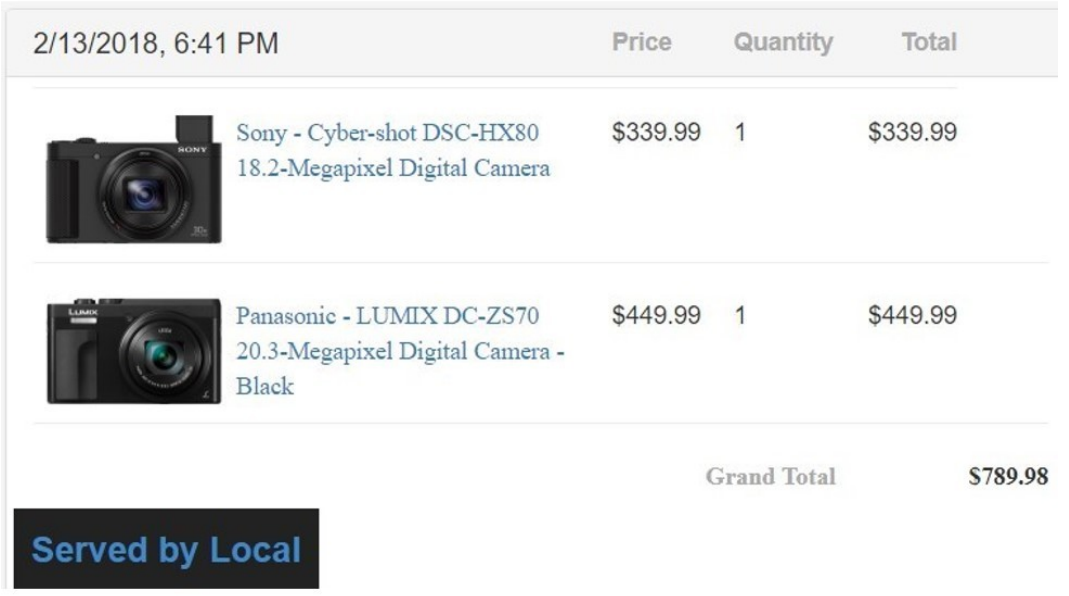
1. **Visual Studio**
   - Right click `Online.Store.WebJob` application
   - Select `Debug -> Start new instance`
2. **.NET Core CLI**
   - Open a terminal and `cd` to `Online.Store.WebJob` application's folder
   - Run `dotnet run .\Online.Store.WebJob.csproj`

It should receive all the order messages exist in the Service Bus queue and process them. When a message is received the app logs related information. This information will be usefull when you will deploy the web job up in Azure.



Web Job - Log messages

If all the configuration have been set properly, you should see the orders in the `Online.Store` interface as well.



Web Job - Receive messages

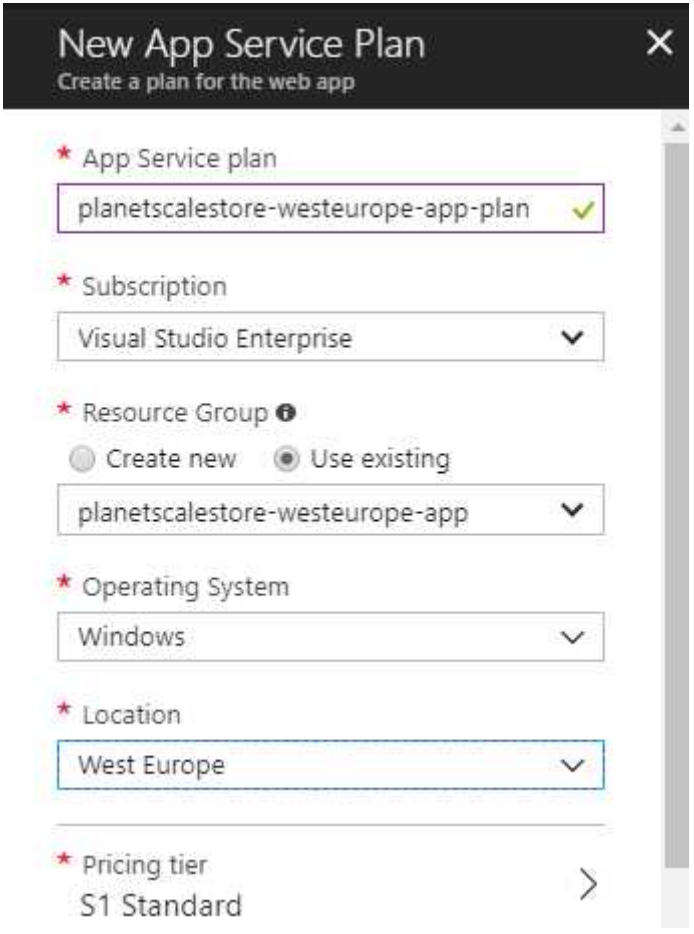
Deployment

App Service

Create App Service Plan

In Azure Portal select `New` from the left panel, search for and select `App Service Plan`. Click the `Create` button and fill the form as follow:

1. **App Service Plan**: <Child-Resource-Group>-plan
2. **Subscription**: <Select-Your-Subscription>
3. **Resource group**: Use existing <Child-Resource-Group>
4. **Operating System**: Windows
5. **Location**: same as <Child-Resource-Group>
6. **Pricing tier** S1 Standar



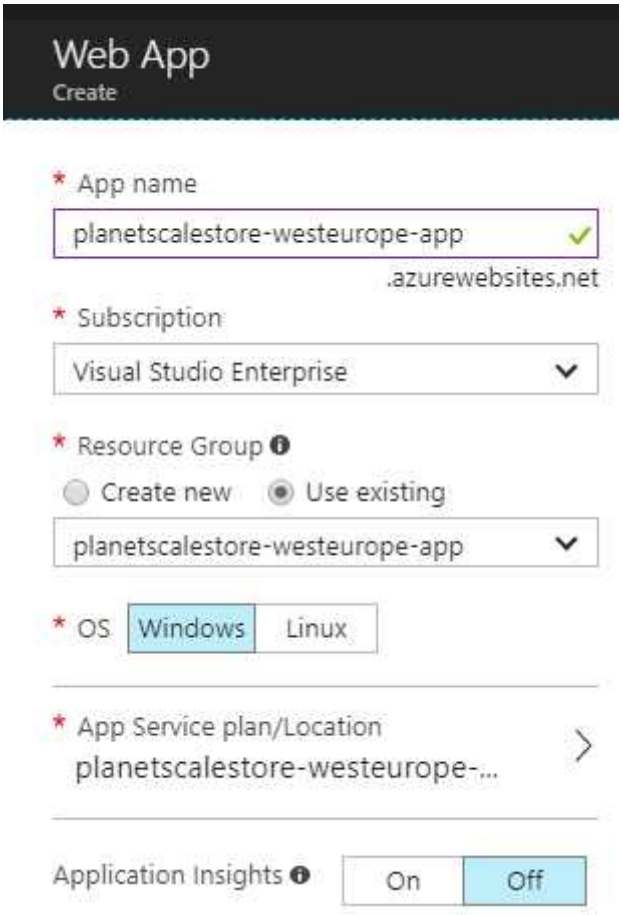
Create App Service Plan

Create App Service

In Azure Portal select `App Services` from the left panel and click the `Add` button. Search for and select to create a **Web App**. You will find this option at the `Web Apps` section. If `App Services` don't exist on the left panel, just click the `New` button and search for `Web App` Click the `Create` button and fill the form as follow:

1. **App name**: <Child-Resource-Group>
2. **Subscription**: <Select-Your-Subscription>
3. **Resource group**: Use existing <Child-Resource-Group>
4. **Operating System**: Windows
5. **App Service plan/Location** Click and select the plan you created before
6. **Application insights**: Off

Click `Create` to create the App Service.



Create App Service

When ths App Service is created click to view the resource. You can to this either by:

- Select `Resource Groups` from the left menu and then click your **child** resource group. Find and click the resource of type `App Service`
- In the left panel select `App services`

In the main blade of your App Service and under the `Settings` section, select `Application settings` and scroll down to `App settings` section. Here is where you put all your secrets keys you have in **secrets.json** file except the connection string(s) which is placed in the `Connection strings` section below. Your app settings should look like this:

Property	Value
WEBSITE_NODE_DEFAULT_VERSION	6.11.2
Region	West Europe
AzureAd:ClientSecret	<your-client-secret> *(if AD authentication selected)*
DocumentDB:Key	<your-documentdb-primary-key>
DocumentDB:ConnectionPolicies	West Europe
Storage:AccountKey	<storage-key1-key-value>
SearchService:Name	<parent-resource-group>-search
SearchService:ApiKey	<search-service-primary-key>
RedisCache:Endpoint	<parent-resource-group>-rediscache
RedisCache:Key	<redis-cache-primary-key>
ServiceBus:Namespace	<parent-resource-group>-servicebus
ServiceBus:Queue	orders
ServiceBus:WriteAccessKeyName	Write
ServiceBus:WriteAccessKey	<write-access-policy-key>
UseIdentity	true or false

You may see some new properties introduced here.

1. **WEBSITE_NODE_DEFAULT_VERSION**: The `NODE` version needed for the Angular app.
2. **Region**: An indicator for each App Service deployed to show which App Service serves a request. Later on, you will deploy an App Service in `West Central US`, the value for this property will be `West Central US`. This value is shown at the bottom of the page in `Online.Store` application *(it is Local for localhost)*
3. **DocumentDB:ConnectionPolicies**: This value indicates the priorities for accessing the `DocumentDB account`. Since at the moment the account is deployed in `West Europe` without any replication, the value for the property is `West Europe`. Later on, you will replicate the data in a second region *(West Central US)* and you may change the value to `West Europe,West Central US` for the App Service deployed in `West Europe` and `West Central US,West Europe` for the one deployed in `West Central US`

Secret keys VS App settings

You may have noticed that the above table has some settings that you configured in the **appsettings** file and not in **secrets.json**, such as the `SearchService:Name` or the `RedisCache:Endpoint`. Anything that is declared in the `App settings` table of your provisioned App Service will override the value in the **appsettings.json** file. You do this because App Services deployed in different regions have different values for those properties, for example the **<parent-resource-group>** for an App Service deployed in `West Europe` may be `planetscalestore-westeurope` but the App Service deployed in `West Central US` will be `planetscalestore-westcentralus`. All those values will be set for you automatically using Powershell scripts.

Right after the `App settings` in the same blade, there is a `Connection strings` section. Add a connection string as follow:

1. **Name**: DefaultConnection
2. **Value**: <connection-string> - same with the one you entered in **secrets.json** file

Warning

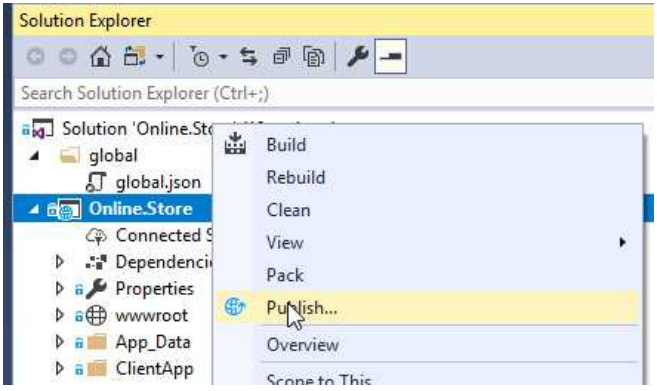
Notice that in the portal the property for the connection string is just `DefaultConnection` while in **secrets.json** file is `ConnectionStrings:DefaultConnection`

In case you want to use `ASP.NET Identity` for authentication make sure to add the `IdentityConnection` connection string as well.

It is high time you deploy the `Online.Store` application up in Azure. There are two ways to do this, a simple one using **Visual Studio** and a more complex to set but yet automated through AppVeyor. `AppVeyor` deployment process will be described in the `DevOps` section.

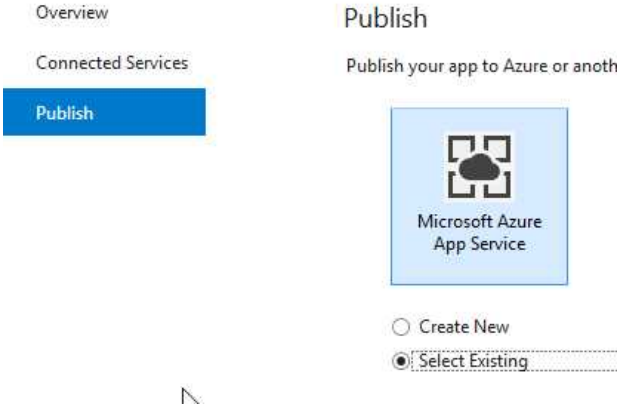
Deploy using Visual Studio 2017

Open `Online.Store` solution in Visual Studio. Make sure you are logged in with the account you used in Azure Portal to create the resources Right click the `Online.Store` web application project and click `Publish..`



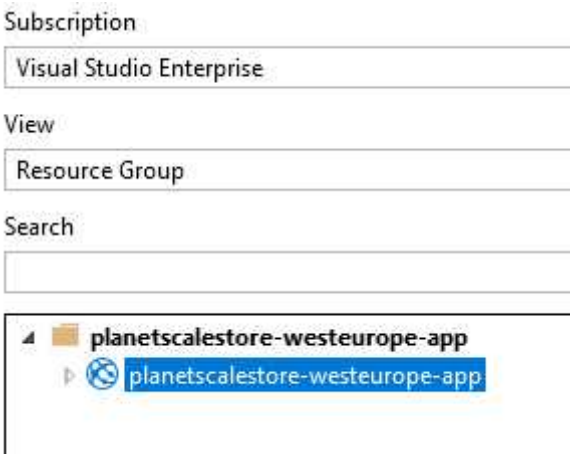
Right click and publish

In the new window select the `Microsoft Azure App Service` box, check `Select existing` and click `Publish`



Select existing App Service

In the next window find the `App service` you created before and click `Publish`. If Visual Studio asks you to re-enter your Azure credentials first, do it

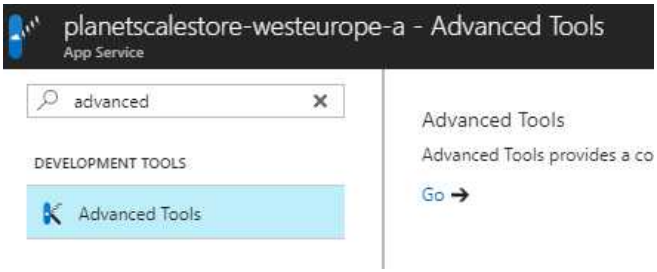


Start publishing App Service

It will take a while for Visual Studio to complete publishing the app in Azure. When it finishes it will open the app's URL in your default browser. The URL should be something like `https://<primary-name>-westeurope-app.azurewebsites.net/`. You should receive an error though because you haven't finished all the steps. The **Angular** app in `Online.Store/ClientApp` requires some `npm` packages which aren't installed in the deployed app. You can do this manually using the Kudu engine.

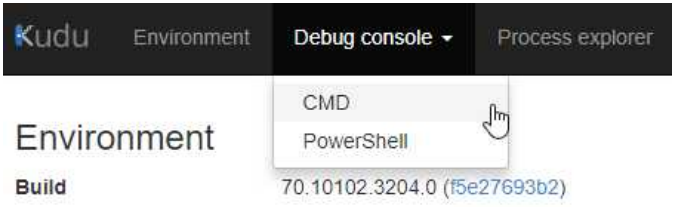
Install NPM packages using KUDU

Back in the portal click your App Service resource and in the main blade, search for and click `Advanced tools`. In the blade opened click the `Go` link



Open Kudu Environment

Clicking the link opens a new tab where you can access your real application files. Open the `Debug console` dropdown and select `CMD`



Open Kudu CMD

The site is deployed in `site/wwwroot` folder so navigate to that folder either by clicking the folders in the interface or using the terminal at the bottom. In the terminal run the command `npm install` and wait till finishes *(if it's the first time then it will take about 10 minutes..)*

```
Kudu Remote Execution Console
Type 'exit' then hit 'enter' to get a new CMD process.
Type 'cls' to clear the console

Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

D:\home>
D:\home\site>
D:\home>
D:\home\site>
D:\home\site\wwwroot>npm install
```

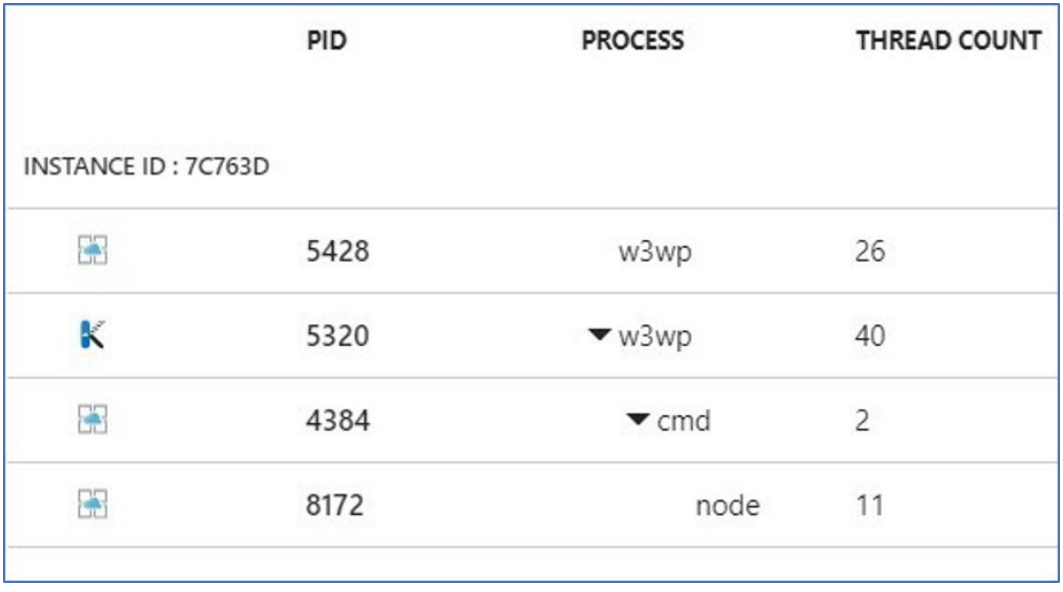
Run npm install in Kudu terminal

When installation finishes, refresh the page in your browser and confirm that the app works fine.

NPM install fails

Sometimes `npm install` might fail in Kudu enviroment. Some common errors are that it fails to rename or delete folders in `node_modules` and it asks for permissions. All you have to do is **kill** the `cmd` process that Kudu uses. To do this follow the next steps:

1. In the main blade of your `App Service` search for and select `Process explorer`
2. Locate and click the `cmd` process
3. Click `Kill` to kill the process



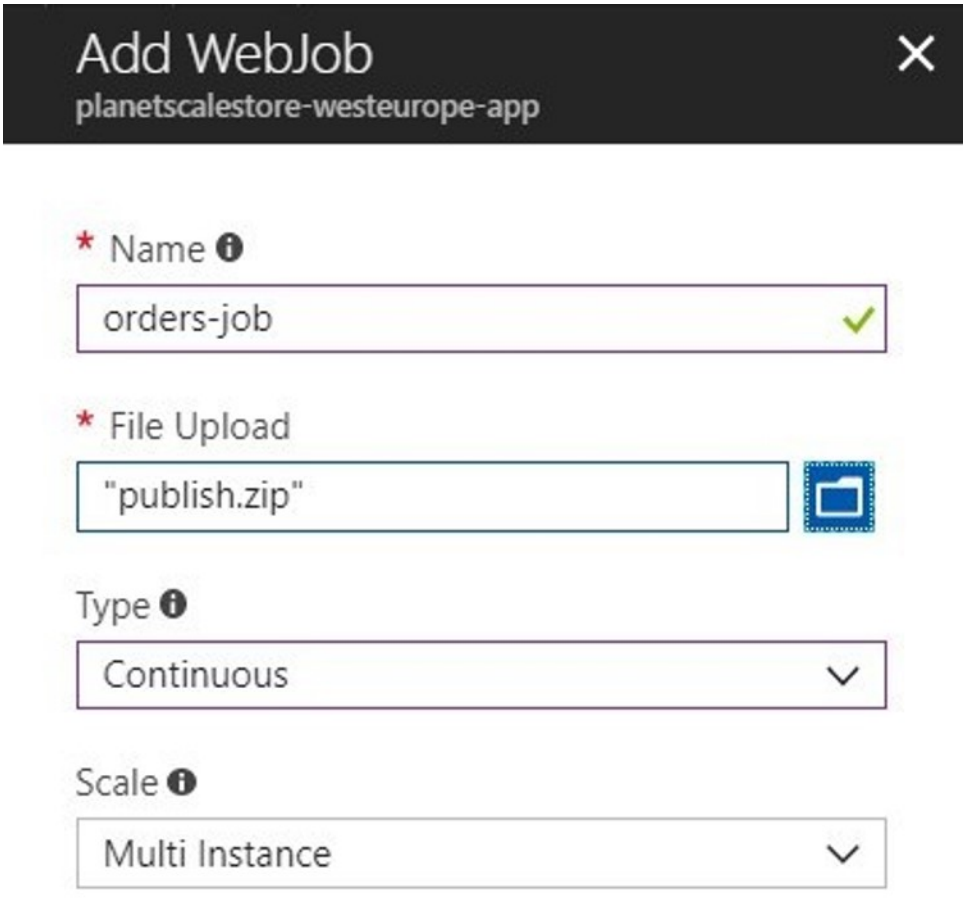
Kill Kudu cmd process

WebJob

Before uploading the web job make sure that you have properly setup the **appsettings.json** file in the `Online.Store.WebJob` project as discussed in previous section.

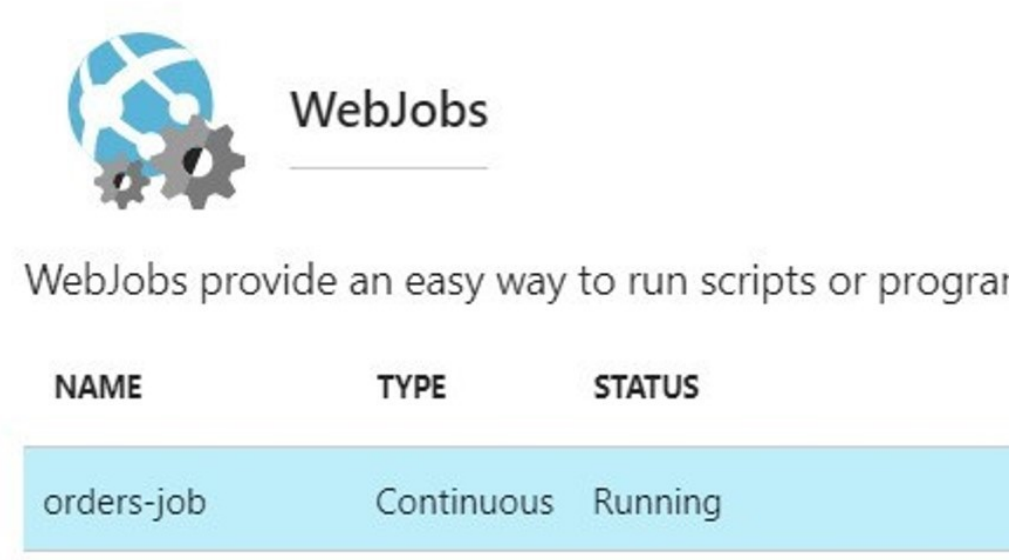
Deploy WebJob to App Service

1. Open a terminal and `cd` to the `Online.Store.WebJob` project's folder
2. Run `dotnet publish -c Release`
3. Go to `Online.Store.WebJob\bin\Release\netcoreapp2.0\publish` and **ZIP** the folder's contents. It doesn't matter how you name the ZIP file but make sure that inside the ZIP file are the contents of the `publish` folder directly and **not the folder *publish***. Also it needs to be only `ZIP`, not `rar` or some other archive tool's extension
4. Switch to Azure Portal and in the main blade of your `App Service` search for and select `WebJobs`. In the blade opened click `Add`, fill the form as follow and click `OK`:
    - **Name**: orders-job *(or anything you like, just make it descriptive)*
    - **File Upload**: Click the folder icon, and locate the **ZIP** file you created before
    - **Type**: Continuous
    - **Scale**: multi Instance



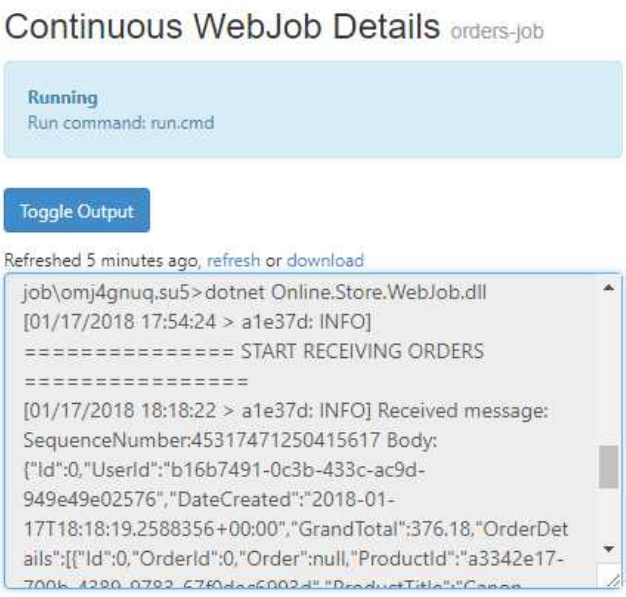
Publish WebJob

When the WebJob is deployed you should see it in the list with status `running`.



WebJob Status

Select it from the list and click `Logs`. Any messages the `Online.Store.WebJob` console app logs, appear on that screen. At the very least and if everything has been setup correctly you should see the `START RECEIVING ORDERS` message. Try to submit orders from the `App Service` URL and check they are saved in database as well.

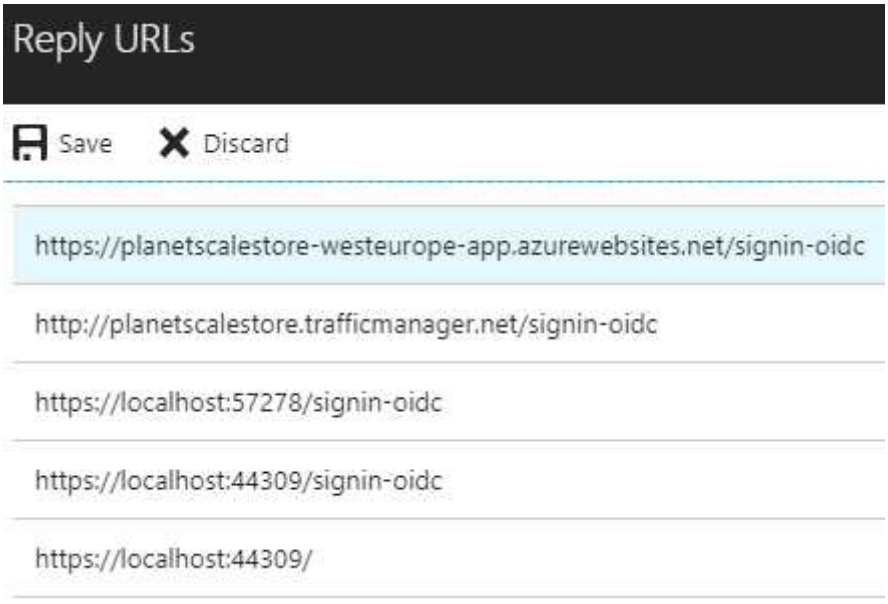


WebJob Logs

Reminder if Azure Directory authentication used

In order to login from an Azure `App Service` URL such as `https://planetscalestore-westeurope-app.azurewebsites.net` you have to add it to the **Reply URLs** of the registered application in the `Azure Active Directory` you created for authenticating users. More specifically you add this URL plus **/signin-oidc**. Check the following example:

```
https://planetscalestore-westeurope-app.azurewebsites.net/signin-oidc
```



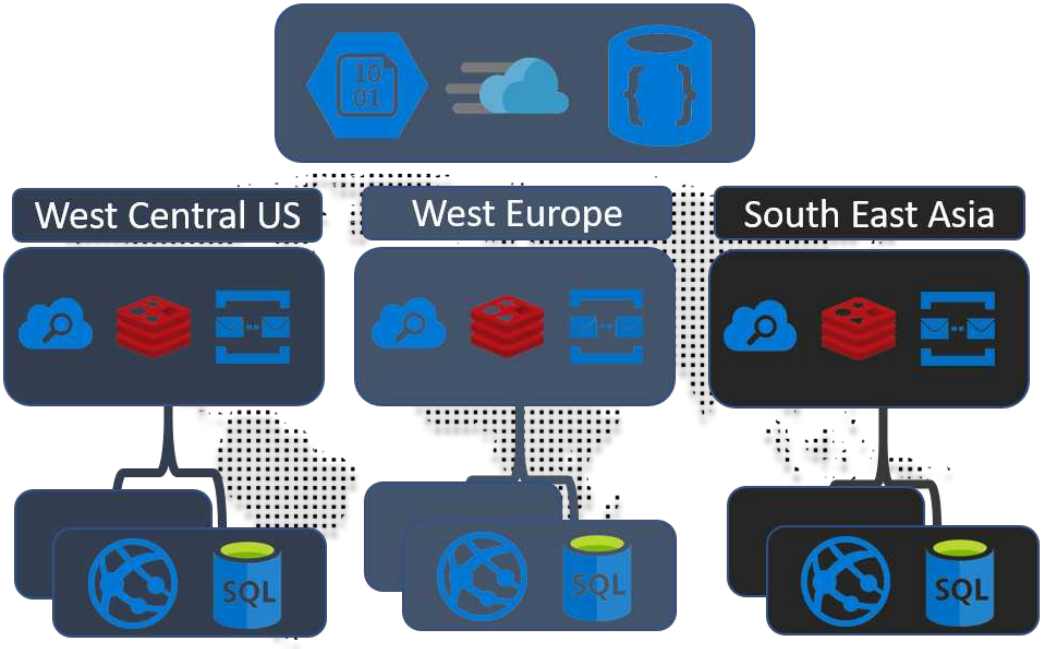
App Service URL - Reply URLs

How did Azure knew how to start the `WebJob`?

Inside the `Online.Store.WebJob` project you will find a **run.cmd** file having the command `dotnet Online.Store.WebJob.dll`. As soon as you deploy the ZIP file Azure will run the **run.cmd** file and hence the WebJob will start running

Global Scale

At this moment you should have `Online.Store` deployed in one region *(West Europe)* but still, you haven't leveraged its global-scale architecture. In this part you will understand the reason why resource groups and their resources have been grouped in such manner. You will see how easy is to `go global` in case your application goes viral in a specific region over the globe. In the following sections you will learn how to create *replicas* of parent and child resource groups in different regions, how to set `Active Geo-Replication` for databases in the new child resource groups and how to use the `Azure Traffic Manager` to distribute traffic between your `App Services` provisioned in different datacenters.



Resource Groups - Replicas

Region Replicas

Architecture

Assuming that you wish to decrease the web app's response delay for users leaving in `United States`, you make the decision to deploy the required resources in `West Central US` Azure region. So the big question here is what exactly do you have to provision and what you don't. Fortunately, you have already designed your resource groups in such way that makes quite easy to answer the question. All you have to do is provision a **Parent Resource Group** *(if not already exists)* and a **Child Resource Group** for that region. These groups should have the exact type of resources with those groups you created in `West Europe`.

For `West Europe` region you created a parent resource group named `<primary-resource-group>-westeurope` and you added three resources of type `Redis Cache`, `Search Service` and `Service Bus`. Those resources were named after their resource group they belong to plus a dash and a prefix. For example, assuming that the primary resource group is named *planetscalestore* then the parent resource group resources in `West Europe` would look like this:

NAME	TYPE	LOCATION
planetscalestore-westeurope-rediscache	Redis Cache	West Europe
planetscalestore-westeurope-search	Search service	West Europe
planetscalestore-westeurope-servicebus	Service Bus	West Europe

Following the same naming convesion, the parent resource group in `West Central US` region should be named **planetscalestore-westcentralus** and its resources would look like this:

NAME	TYPE	LOCATION
planetscalestore-westcentralus-rediscache	Redis Cache	West Central US
planetscalestore-westcentralus-search	Search service	West Central US
planetscalestore-westcentralus-servicebus	Service Bus	West Central US

Everything you did in the **Performance Optimization** part is related to `Parent Resource`

Groups so go ahead and repeat the same but this time for West Central US region.

For the Child Resource Group in West Central US you will do **almost** the same as you did for the child resource group you created in West Europe. Following the *planetscalestore* example the child resource group is named <parent-resource-group>-app and its resources of type SQL server, SQL Database, App Service plan and App Service would look like this:

NAME	TYPE	LOCATION
planetscalestore-westeurope-app	App Service	West Europe
planetscalestore-westeurope-app-plan	App Service plan	West Europe
planetscalestore-westeurope-app-sql-a	SQL server	West Europe
planetscalestore	SQL database	West Europe

Following the same naming convension the child resource group in West Central US should be named <parent-resource-group>-app that is planetscalestore-westcentralus-app.

Following the example of planetscalestore the child resources for West Central US would look like this:

NAME	TYPE	LOCATION
planetscalestore-westcentralus-app	App Service	West Central US
planetscalestore-westcentralus-app-plan	App Service plan	West Central US
planetscalestore-westcentralus-app-sql-a *(Do not create it yet)*	SQL server	West Central US
planetscalestore *(Do not create it yet)*	SQL database	West Central US

Everything you did in the **Web Application** part is related to Child Resource Groups but you will not repeat all the steps for West Central US region *(this is what the almost was all about)*. As far as Active Directory section, in case you don't use ASP.NET Core Identity for membership, the only thing you need to do is add the new App Service URL in the **Reply URLs** as described so far. The main difference though is that **you won't create the SQL Database**. The reason is that Online.Store application uses the SQL Active Geo-Replication feature which means that there is only one **Read-Write** database, currently located in West Europe and all others will be **Read-Only**. This doesn't mean that the child resource group in West Central US won't have SQL Server and Database, it just means that the SQL Database will be created through the SQL Active Geo-Replication feature.

Be carefull when setting the App Settings and the Connection string in your new App

Service in `West Central US`. Those settings should be *bound* to the new parent resource group in `West Central US` and not the one in `West Europe`. This means that at least the following properties must be different:

- **SearchService:Name**: e.g.: <primary-name>-westcentralus-search
- **SearchService:ApiKey**: <different>
- **RedisCache:Key**: <different>
- **RedisCache:Endpoint**: <primary name>-westcentralus-rediscache
- **DocumentDB:ConnectionPolicies**: West Central US,West Europe
- **Region**: westcentralus
- **ServiceBus:WriteAccessKey**: <different>
- **ServiceBus:Namespace**: <primary-name>-westcentralus-servicebus

The following properties may remain the same:

- **AzureAd:ClientSecret**: <same>
- **Storage:AccountKey**: <same>
- **DocumentDB:Key**: <same>
- **ServiceBus:Queue**: orders
- **WEBSITE_NODE_DEFAULT_VERSION**: 6.11.2
- **ServiceBus:WriteAccessKeyName**: write

The `DefaultConnection` connection string should point to the new **Secondary** database which you will create later on. The value will be slightly different from the one you used in the first App Service, meaning that only the name of the SQL Server changes (*instead of `<primary name>-westeurope-app-sql-a` set `<primary name>-westcentralus-app-sql-a`*) since you will use the same credentials for that server as well.

For the `WebJob` deployed in `West Central US` change the following `2` property values in `appsettings.json` file before run the `publish` process:

- **ServiceBus:Namespace**: <primary name>-westcentralus-servicebus
- **ServiceBus:ReadAccessKey**: <different>

All others may remain the same especially the **connection string** which must point to the `Read-Write` primary database in `West Europe`.

Azure Cosmos DB replication

In case you set the property `DocumentDB:ConnectionPolicies` equal to `West Central US,West Europe` for the App Service in `West Central US`, this means that you intend to replicate your Azure Cosmos DB account's data to `West Central US`. The App Service will send all requests to `West Central US` and if not available it will try the next priority set which is the default in `West Europe`. Follow the next steps to replicate your Azure Cosmos Account data in `West Central US`.

Replicate Azure Cosmos DB globally

1. In Azure Portal find and open your `Azure Cosmos DB` account resource
2. In its main blade select `Replicate data globally`
3. This will open a new blade with a map that shows all the regions available to replicate your data. Select the `West Central US` and that's it!

Region not available

Sometimes, the region you wish to replicate your data to, may not be available. That's OK, just select another region closest to the one you wished. But make sure you adjust the `DocumentDB:ConnectionPolicies` property value accordingly

West Central US - Not Available

At this point, you can proceed with the following:

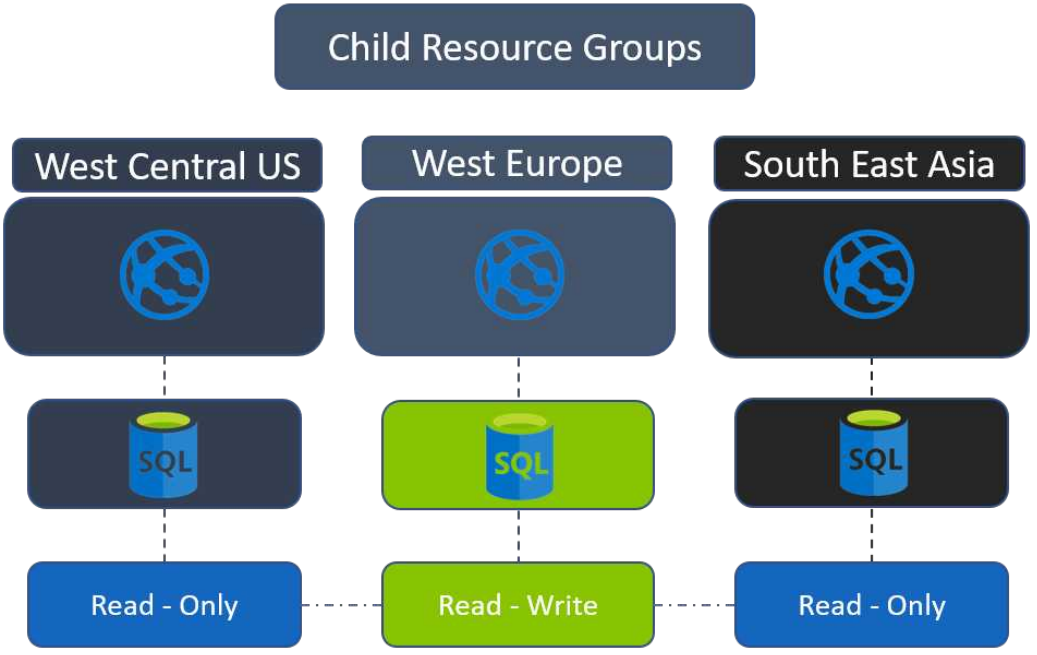
1. Create the parent resource group in `West Central US` and all its resources *(Search Service, Redis Cache, Service Bus)*
2. Set their settings in the **appsettings.json** file locally so that you can deploy later the app in the `App Service` provisioned in `West Central US`
3. Create a child resource group in `West Central US`, add the `App Service Plan` and the `App Service` to it
4. In Azure Portal, open the new `App Service` resource and set its `App Settings` as discussed previously
5. Use Visual Studio and deploy `Online.Store` application in the `App Service` provisioned in `West Central US`. Make sure you run `npm install` in the Kudu environment when the deployment finished

After completing all these, you will continue by creating a new `SQL Server` and a database in the new child resource group. This will be the SQL Server that the new `App Service` will connect to.

SQL Active Geo-Replication

Overview

`SQL Active Geo-Replication` is one of the most powerfull features `Azure SQL Databases` service offers which allows you to configure up to four readable secondary databases in the same or different data center locations. The way that `SQL Active Geo-Replication` works is that you have a primary database that continously sends commited transactions asynchronously to its secondary databases so that they are up to date. Secondary databases are `Read-Only` and available for querying and **failover** as well. This means that if there's a data center outage or inability to connect to the primary database you can initiate a failover to one of your secondary databases. When the failover completes, the connection endpoint to the new `Read-Write` primary database changes and you have to manually apply any required changes to your apps. All other secondary databases are automatically linked to the new primary. If you wish to have more than four secondary databases you can just create secondary of a secondary database, a process known as *chaining*.



SQL Active Geo-Replication

`Online.Store` application uses `SQL Active Geo-Replication` to replicate the primary's database data to other regions around the world. Each `App Service` **reads** orders data from the database deployed in the same region and hence keeps low the latency for reads operations. Write operations are comitted asynchronously by `WebJobs` deployed globally.

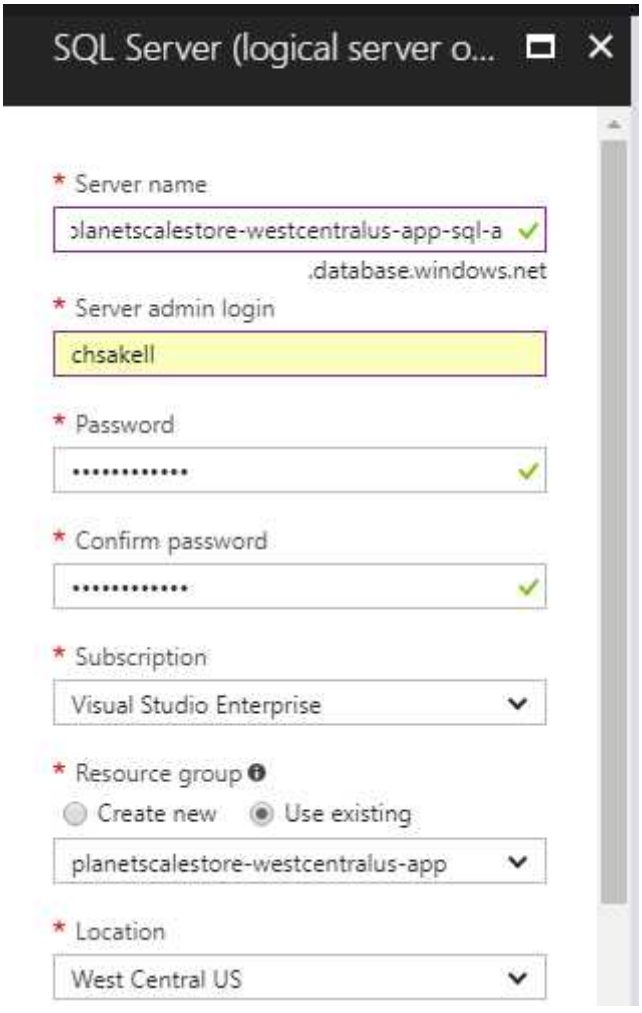
Azure Portal

Assuming that you have created a parent *(e.g. planetscalestore-westcentralus)* and a child resource group *(e.g. planetscalestore-westcentralus-app)* in `West Central US`, and also created the `App Service plan` and the `App Service` in the new child resource group, follow the next steps to create the `SQL Server`:

Create SQL Server

- In the left panel click `New`, search for and select `SQL server (logical server)`. Click Create an fill the form as follow:
- **Server name**: <Child-Resource-Group>-sql-a *(the one in `West Central US`)*
- **Server admin login**: <your-admin-login>
- **Password**: <your-admin-password>
- **Confirm password**: <your-admin-password>
- **Subscription**: <Select-Your-Subscription>
- **Resource group**: Use existing <Child-Resource-Group> *(the one in `West Central US`)*
- **Location**: <same-as-parent-location> *(West Central US)*
- **Allow azure services to access server**: Checked

Make sure to use the same credentials you used for the SQL Server in `West Europe`.

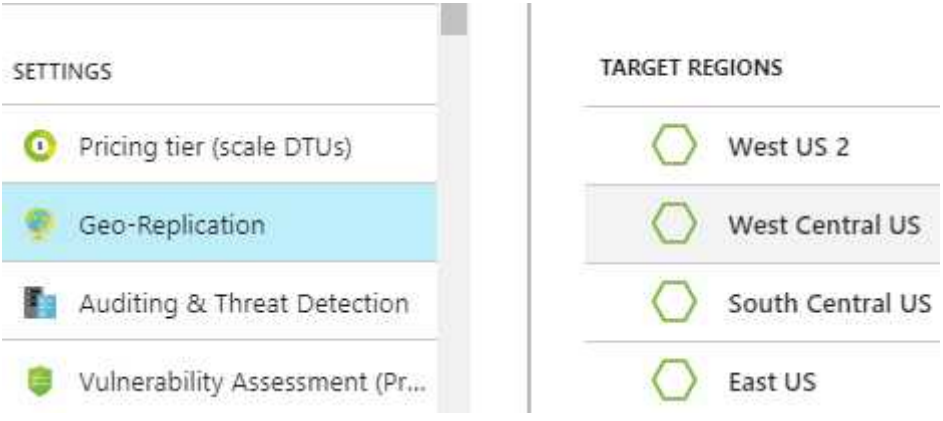


Create SQL Logical Server

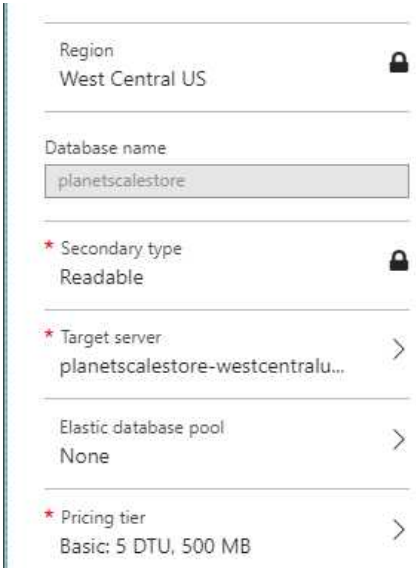
Next procceed to enable Active Geo-Replication. In Azure Portal locate and open the SQL Database resource located in `West Europe` and follow the next steps:

Enable Geo-Replication

1. In the main blade of the SQL Database resource, select `Geo-Replication`
2. In the blade opened, scroll down and in the `TARGET REGIONS` section select `West Central US`
3. The next blade looks very similar with the one you saw when you created the SQL Database for `West Europe` except that this time most of the fields are locked. The only thing that you can actually configure is the **Target server** field. Click `Target server` and select the `SQL Server` you created in the previous step. Finally click OK



Target Regions

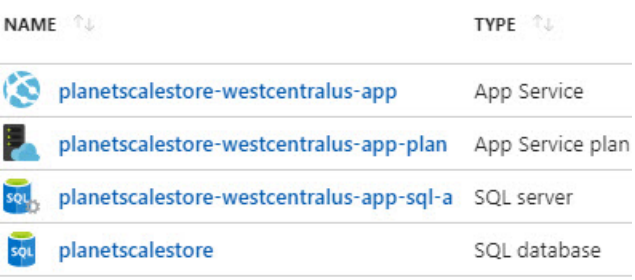


Create Secondary Database

When you click OK you will see the deployment process on the blade's map.

Starting Geo-Replication

Wait until the provisions are completed and confirm that the new resources have been set as as intended on your new Child Resource group in `West Central US`.



West Central US child resource group

Azure Traffic Manager

Overview

`Azure Traffic Manager` is used to distribute traffic between service endpoints in different datacenters. Endpoints can be either Azure endpoints such as Azure VMs, Web Apps, and cloud services or externals non-Azure endpoints. `Traffic Manager` uses DNS and a traffic-routing method of your choice to direct client requests. The four traffic routing methods available are:

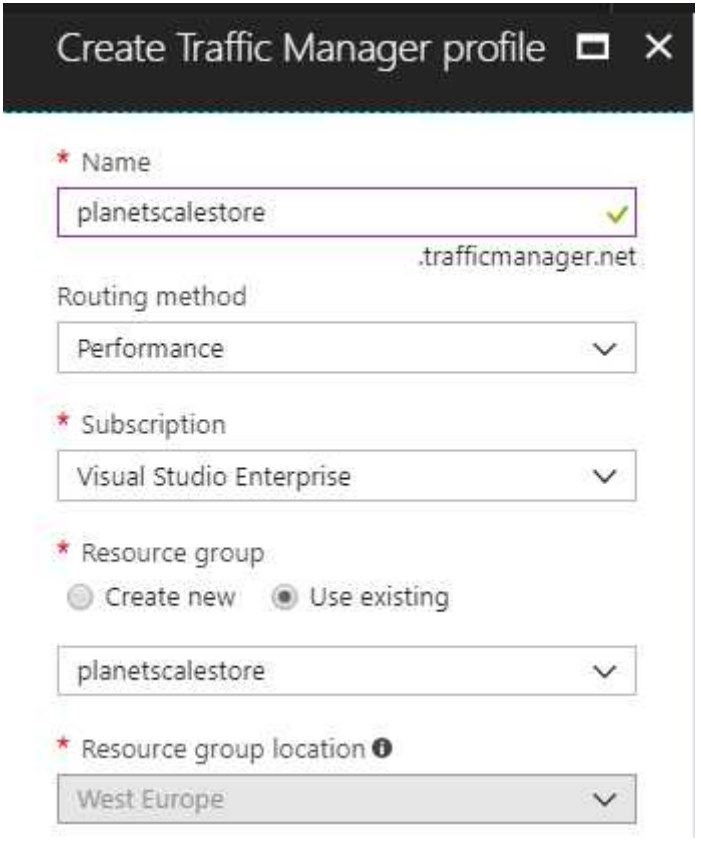
- **Priority**: When you want to use a primary service endpoint for all traffic, and provide backups in case the primary or the backup endpoints are unavailable
- **Weighted**: When you want to distribute traffic across a set of endpoints, either evenly or according to weights, which you define
- **Performance**: When you have endpoints in different geographic locations and you want end users to use the "closest" endpoint in terms of the lowest network latency.
- **Geographic**: When you want users to be directed to specific endpoints (Azure, External, or Nested) based on which geographic location their DNS query originates from

`Azure Traffic Manager` can help you increase the availability of your applications by continuously monitoring your endpoints and properly directing users to those that are healthy. For globally distributed applications such as `Online.Store` provides a **consistent** responsiveness by directing users to the endpoint with the lowest network latency. `Online.Store` application uses the `Performance` routing method since the web app is deployed to more tha one regions around the world. This way, users from United States are served by the `App Service` provisioned in a `US` region and users from Europe are served from the `App Service` provisioned in one of the datacenters in Europe.

Azure Portal

Create Traffic Manager Profile

- In the left panel click `New`, search for and select `Traffic Manager Profile`. Click Create an fill the form as follow:
- **Name**: <Primary-Name>
- **Routing method**: Performance
- **Subscription**: <Select-Your-Subscription>
- **Resource group**: Use existing <Primary-Resource-Group>
- **Resource group location**: <same-as-primary-location> *(Will be global actually..)*



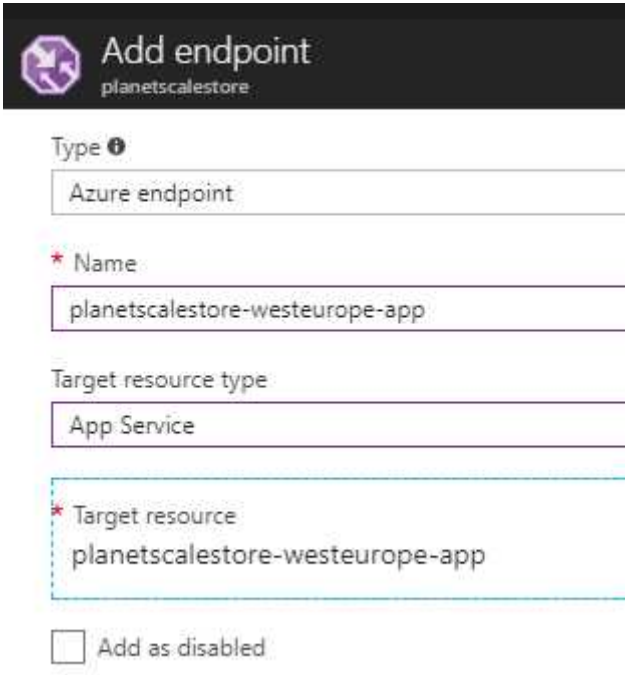
Create Traffic Manager Profile

The `Traffic Manager Profile` is created once and is common for all child `App Services` and this is why is placed under the primary resource group. After creating the profile go to resource. You can to this either by:

- Select `Resource Groups` from the left menu and then click your **primary** resource group. Find and click the resource of type `Traffic Manager profile`
- Click `More services` from the left menu, search for and click `Traffic Manager profiles`. The new blade will list all the `Traffic Manager profiles` you have deployed. The next thing to do is add the `Azure App Services` endpoints into the endpoints collection of the traffic manager profile

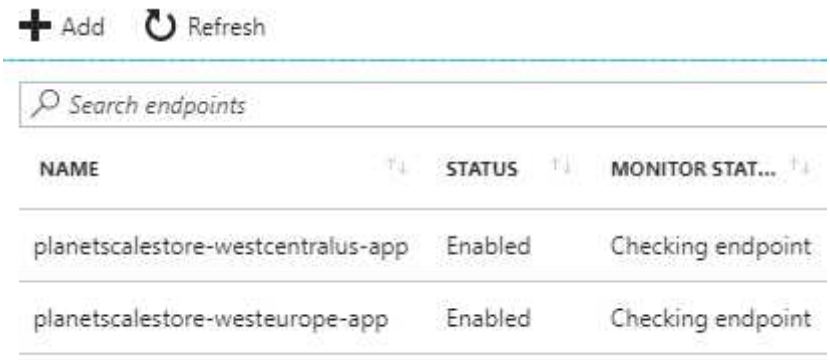
Create Traffic Manager Endpoint

1. In the main blade of your Traffic Manager profile select `Endpoints` and click `Add`
2. Select `Azure endopoint` for Type
3. For each `App Service` you have created *(two so far)* set as follow:
    - **Name**: <app-service-name> *(e.g. planetscalestore-westeurope-app)*
    - **Target resource type**: App Service
    - **Target resource**: Click *Choose an app service* and select the proper App Service respectively
    - **Add as enabled**: Unchecked



Add Traffic Manager Endpoint

As soon as you create the endpoints, traffic manager profile will check their status and normally after a few seconds they should be `Online`.



Traffic Manager Endpoints Status

In the main blade of your Traffic Manager profile select `Overview` and then click the `DNS name`. This is your Traffic Manager's URL that eventually internally resolves to one of your two App Services URLs. Your DNS name should be something like `http://<primary-name>.trafficmanager.net`. When you open that url in a browser Traffic Manager should resolve to the App Service that can serve you faster. You can always check which App Service served you at the bottom left of `Online.Store` application's page.

GeoPeeker

You can use a tool named GeoPeeker to view how your site appears in different locations over the world

Paste your Traffic Manager's DNS name and check the response. You will see that the server in `Ireland` was served by the App Service in `West Europe` while most of the others served by the one in `West Central US`

Active Directory Reply URLs

In case you use `Active Directory B2C` for authentication you have to add `http://<primary-name>.trafficmanager.net/signin-oidc` at the `Reply URLs` of your registered application

Scalability

Having your web application deployed all around the world doesn't ensure that can handle by default any performance requirements. You start by provisioning resources and selecting the `basic` or `standard` tier but as your application goes viral, you should be able to dynamically allocate resources to match the new demands. For example you may have `Online.Store` application deployed in `West Europe` and `West Central US` and the app goes viral on `United States`. The web app should be able to adapt to the new demands in `US` either automatically or manually, maintain the desired performance levels and satisfy service-level agreements (SLAs). As demand slackens and the additional resources are no longer needed, they can be de-allocated to minimize costs.

App Services

`App Services` gives you two options to scale your app, `Scale up` or `Scale out`. With `Scale up` you get more CPU, memory, disk space, and many extra features. By `Scaling out` you increase the number of instances running your web app.

Provioning new instances

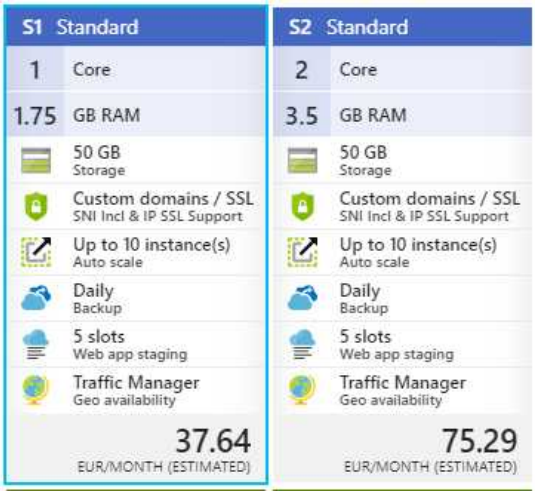
While new instances are provisioned your application continues running without interruption. Also, you are not required to change your code or redeploy the application

You scale up by changing the pricing tier of the `App Service plan` that your app belongs to. `Scale up` is also known as `Vertical scaling`.

Scale up App Service plan

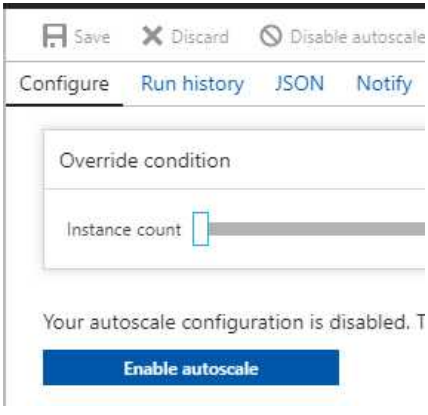
1. In Azure Portal open the `App Service` resource in the region you wish to scale up
2. In the main blade of the `App Service` select `Scale up (App Service plan)`

You will find plenty of pricing tiers to choose. The recommendation is to start scaling up one tier at a time and see if that works for you



App Service plan - Scale up

In `Scale out` which is also known as `Horizontal scale` you can choose either to manually set the number of instances running your web app or enable a feature called **autoscale**. In `autoscale` mode you can add **rules** defining how you wish to increase or decrease instances for your app based on certain `metrics`. For example you can set that you wish to provision `2` more instances every time the average CPU is greater than `70%`.

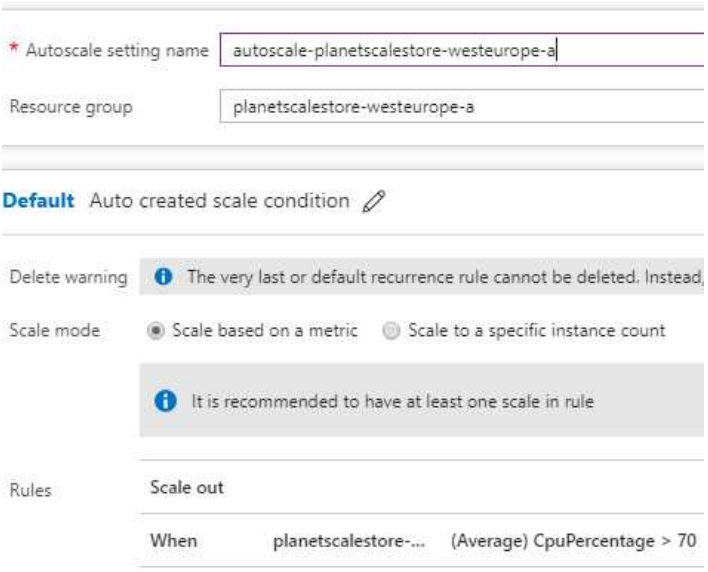


App Service plan - Scale out

Scale out App Service plan - Autoscale

1. In Azure Portal open the `App Service` resource in the region you wish to scale out
2. In the main blade of the `App Service` select `Scale out (App Service plan)`
3. In the new blade fill the form as follow:

- **Autoscale setting name**: autoscale-<child-resource-group>
- **Resource group**: <child-resource-group>
- **Scale mode**: Check `Scale based on a metric` and click `Add a rule`. Fill the `Scale rule` form as follow and click `Add`:
  - **Metric source**: Current resource
  - **Resource type**: App service plans
  - **Resource**: <child-resource-group>
  - **Time aggregation**: Average
  - **Metric name**: CU Percentage
  - **Time grain statistic**: average
  - **Operator**: Greater than
  - **Threashod**: 70
  - **Duration**: 10
  - **Action**: Increase count by
  - **Instance count**: 2
  - **Cool down (minutes)**: 5
- **Instance limits**: Min 1 - Max 10 - Default 1



App Service plan - Autoscale

You can also add a rule that defines how and when you wish to decrease instances of your web app.

WebJobs

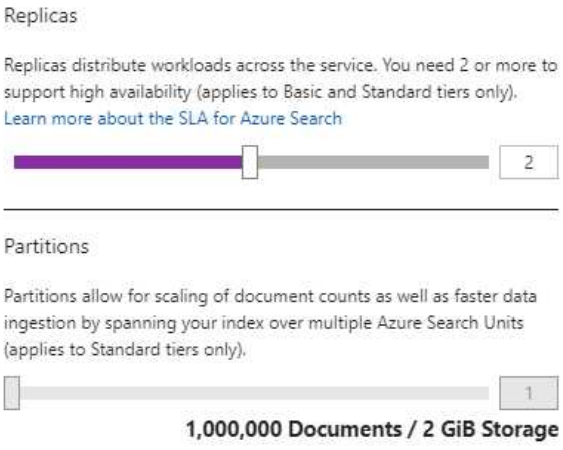
You don't have to do anything about your `WebJobs` since are part of the `App Service`. As long as the deployed `WebJob` is configured as **Continuous**, then it will run on all instances that the web app runs on

Azure Search

If your web app instances increase, it means that more and more users visiting your website in a specific region and eventually sending more search queries to your `Azure Search Service`. The query latency of your `Search Service` is a good indicator that you need to add more of that resource. In order to improve the query performance of your `Search Service` you **add replicas**. Based on the pricing tier you have selected for your Search Service you can add a specific maximum amount of replicas. Microsoft recommends that you add the replicas through the Azure Portal.

Scale Search Service

1. In Azure Portal open the `Search Service` resource in the region you wish to add the replicas
2. In the main blade of the `Search Service` select `Scale`
3. In the new blade increase the number of the replicas



Search Service

You can increase either the number of the replicas or the number of the partitions. Increasing the replicas means you increase the instances of the `Search Service` something that helps to properly load balance the query operations when query volumes get large. Increase the partitions when your application requires near real-time data refresh.

Azure Redis Cache

`Azure Redis Cache` is a scalable service as well. By monitoring your `Redis Cache` in Azure you can understand when scaling is required. In `Azure Portal` select the instance of the `Redis Cache` you wish to monitor and in the main blade select `Overview` or `Metrics`. For following table lists all the metrics you can monitor for your `Redis Cache`.

Cache Hits	Cache Misses	Connected Clients
Evicted Keys	Expired Keys	Total Keys
Get operations	Redis Server Load	Set operations to the cache
Total Operations	Used Memory	Used Memory RSS
CPU utilization	Amount of data read from the cache	Cache Write

The most important metrics that you should be aware of their values are the *Redis Server Load*, *Memory Usage*, *Network Bandwidth* and *CPU Usage*. When you decide to scale your `Redis Cache` simply select `Scale` from its main blade and select the desired cache pricing tier.

Pricing tiers

Mind that you can't scale from a higher pricing tier to a lower pricing tier. For example you cannot scale from a `Premium cache` down to a `Standard` or a `Basic` cache

Azure SQL Databases

More users in your website means more queries to your `SQL Databases`. Luckily, running your databases on Microsoft `Azure SQL Databases` allows you to dynamically manage them and adapt to changing workload demands. At any time you can either `scale-up` or `scale-down` the DTUs *(CPU/memory/IOPS resources)* of your databases with a few clicks. But how do you know when to `scale up` or `scale down` anyway?

In the main blade of your `SQL Database` select `Overview` and you will see the database's resource utilization in terms of DTUs percentage. If this percentage is getting close to 100% then you know that it's time to `scale up`. Another thing you can do is create an **alert rule**. In the main blade again, select `Alert rules`. You can add several alerts based on the following metrics:

- Blocked by Firewall
- Failed Connections
- Successful Connections
- CPU percentage
- Deadlocks
- DTU percentage

- DTU limit
- DTU used
- Log IO percentage
- Data IO percentage
- Sessions percentage
- Total database size
- Database size percentage
- Workers percentage
- In-Memory OLTP storage percent

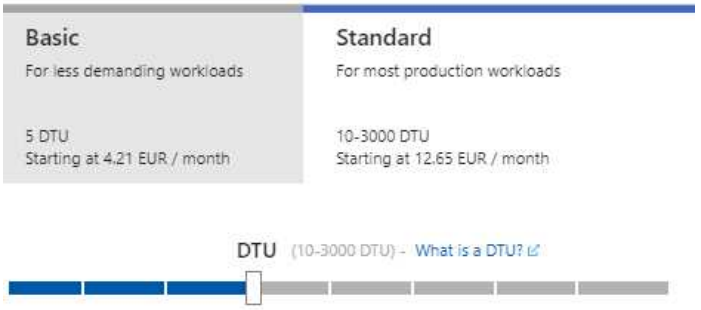
You can set for example an alert to inform you when the `DTU percentage` is greater than 70% or the database's size has reached the 90% percentage of the total size. In order to scale an Azure SQL Database all you have to do is select a different pricing tier. In the main blade of your SQL Database select `Pricing tier (scale DTUs)`. Select the pricing tier that suits your needs.

SQL Active Geo-Replication

If you use `SQL Active Geo-Replication` *(e.g. the case of Online.Store app)* the highest pricing tier you can select for a primary database is the lowest current edition of its secondary dataabases. For example if the current primary's and secondary's tier is the `Basic` then you cannot select the `Standard` for the primary database till all of the secondaries have a higher tier

SQL Databases - Scale not available for primary database

The solution is to change the pricing tier of your secondary databases first and finally your primary's. In the previous example, after changing the secondary's pricing tier to `Standard`, this tier got instantly available for the primary as well. `Premium` though was still unavailable.



SQL Databases - Scale

DevOps Automation

Introduction

After completing all the previous parts you will have realized that provision resources in Azure Portal is a **time consuming** and some times **error prone**. You might for example paste a key in a textbox and forget the last letter or even entirely forget to set a property. Moreover most of the operations you make in the portal are the same over and over and only some parameters change. Wouldn't be awesome to `automate` all these operations? Fortunately this is possible and it can be done in three ways:

1. Azure PowerShell
2. Azure REST API
3. Azure CLI

In this book you learn how to use `Azure Powershell` to automate resource provisioning and release processes. You don't have to be an expert in `PowerShell` to understand the scripts. Most of them include calls to `PowerShell` cmdlets and the only thing required from you is to pass the correct parameters.

Powershell

Before starting scripting with `Azure PowerShell` module you need to have `Powershell` installed. You can find step by step instructions in the official PowerShell Installation page. To follow along with the scripts in the book make sure you have version **5.1.1** or later installed on your machine. After installing `PowerShell`, open a PowerShell session and continue with the `Azure PowerShell` installation instructions below.

Install Azure Powershell

The following instructions come from the official Azure PowerShell Installation page

Install PowerShellGet: Run the following command to see if you have PowerShellGet installed on your system.

```
Get-Module PowerShellGet -list | Select-Object Name,Version,Path
```

If `PowerShellGet` is not installed check the How to Get PowerShellGet section

Install Azure PowerShell: Installing Azure PowerShell from the PowerShell Gallery requires elevated privileges. Run the following command from an elevated PowerShell session:

```
# Install the Azure Resource Manager modules from the PowerShell Gallery
Install-Module AzureRM -AllowClobber
```

If asked to trust the repository answer `'Yes'` or `'Yes to All'` to continue with the installation.

Elevated PowerShell session

`Elevated` session means you have to open `PowerShell` as administrator

Load the AzureRM module: Once the module is installed, you need to load the module into your PowerShell session. You can switch back to a normal - non admistrator PowerShell session again from now on

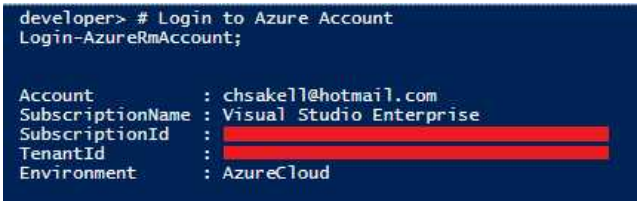
```
Import-Module AzureRM
```

Login to your Azure Account

To manage or provision new resources on `Azure` you have to tell `PowerShell` who you are. Run the following command and enter your Azure credentials you used throught this book:

```
# Login to Azure Account
Login-AzureRmAccount;
```

The command will open a window and ask for your credentials. After entering your credentials successfully, `PowerShell` will log your account details.



Login to your Azure Account

You may have more than one subscriptions so select the **SubscriptionId** you wish to use while operating through `PowerShell`.

```
# Select Subscription
Select-AzureRmSubscription -SubscriptionID "YOUR-SUBSCRIPTION-ID"
```

Save account details

Each time you open a new `PowerShell` session you have to login and select a subscription. Most of the times though you will probably use the same account so it would be nice to save this information for all new sessions. The following command allows the azure credential, account and subscription information to be saved and automatically loaded when you open a PowerShell window.

```
Enable-AzureRmContextAutosave
```

Now you won't have to login over and over again.

In the following sections you will try to provision all the resources you created via Azure Portal through `PowerShell`, so go ahead and **Delete** all the resource groups you created till now.

Delete Resource Group

1. In the left panel, select `Resource groups`
2. For each resource group you wish to delete, click on the three dots on the right and select `Delete resource group`
3. In the blade opened, you will be asked to type either the `Resource Name` you wish to delete or just `yes`. Do as asked and click the `Delete` button

`Powershell` scripts for `Online.Store` application exist in the `Online.Store` project at the `App_Data/devops` folder. Open a `PowerShell` session and `cd` to that folder before proceeding with the next section. In the following sections you will study the code for each of those scripts and at the end of each section you will find instructions on how to run them.

Primary Resource Group

The primary resource group is the base resource group for `Online.Store` application. It contains the resources that are common for all *sub-resource groups* and once it is created, it should't change often. The 4 types of resources contained in the primary resource group are:

- Storage Account
- CDN Profile
- Azure Cosmos DB account
- Traffic Manager Profile
- Identity SQL Server and Database *(optional)*

The primary resource group has a **unique** name and this name will be the base for constructing all other resource groups and resources type names *(based on region, type, version..)*. This will help you locate and monitor resources more easily and of course build more generic PowerShell scripts. The property used for the primary resource group name in the PowerShell scripts that follow, is `PrimaryName`. But why `PrimaryName` should be unique? Most of the resources have **endpoints** which must be globally unique. Take a look how service endpoints look like for the following types in the primary resource group:

- **Storage Account**: https://<primary-name>storage.blob.core.windows.net/
- **CDN Profile**: https://<primary-name>-cdn.azureedge.net
- **Azure Cosmos DB**: https://<primary-name>-cosmosdb.documents.azure.com:443/
- **Traffic Manager Profile**: http://<primary-name>.trafficmanager.net

The prefixes you have added will certainly reduce the possibility for conficts. During the examples of this book, **planetscalestore** has been used as the primary name. Each time you see the word `planetscalestore` in the scripts that follow, make sure to replace it with your own primary name.

Init Primary Resources Script

Ideally, you would like to run a custom script to initialize the primary resource group and all of its resources. This is what the **init-primary-resources.ps1** PowerShell script is for. You will find the script and all others inside the `Online.Store/App_Data/devops` folder. The script accepts 5 parameters of which the last 3 are optional:

1. **PrimaryName**: The primary name to be used
2. **ResourceGroupLocation**: Azure region for the resource group and its resources
3. **CreateIdentityDatabase**: Create or not the SQL Server & database required for ASP.NET Identity ***(optional)***. If parameter is used, possible values are `$true`, `$false`
4. **SqlServerLogin**: SQL Server login for the primary SQL Server created in case `CreateIdentityDatabase` is `True` ***(optional)***
5. **SqlServerPassword**: SQL Server password for the primary SQL Server created in case `CreateIdentityDatabase` is `True` ***(optional)***

PowerShell optional parameters

Optional parameters can be omitted when running a PowerShell script

To list all available resource group location values run the following command in PowerShell.

```
Get-AzureRmLocation | select Location
```

Available Locations for Resource Groups

eastasia	southeastasia	centralus	eastus
eastus2	westus	northcentralus	southcentralus
northeurope	westeurope	japanwest	japaneast
brazilsouth	australiaeast	australiasoutheast	southindia
centralindia	westindia	canadacentral	canadaeast
uksouth	ukwest	westcentralus	westus2
koreacentral	koreasouth		

About locations

This book has followed a `naming convension` where parent and child resource groups belong to the same region. The thing is that some times not all type of resources are available to a specific region. For example, assuming that you want to provision the primary resource group and its resources to `West Europe` but **at that time** `Azure Cosmos DB` may not be available **for that region**. If you run the `init-primary-resources` script, it will fail when try to create the Cosmos DB account.

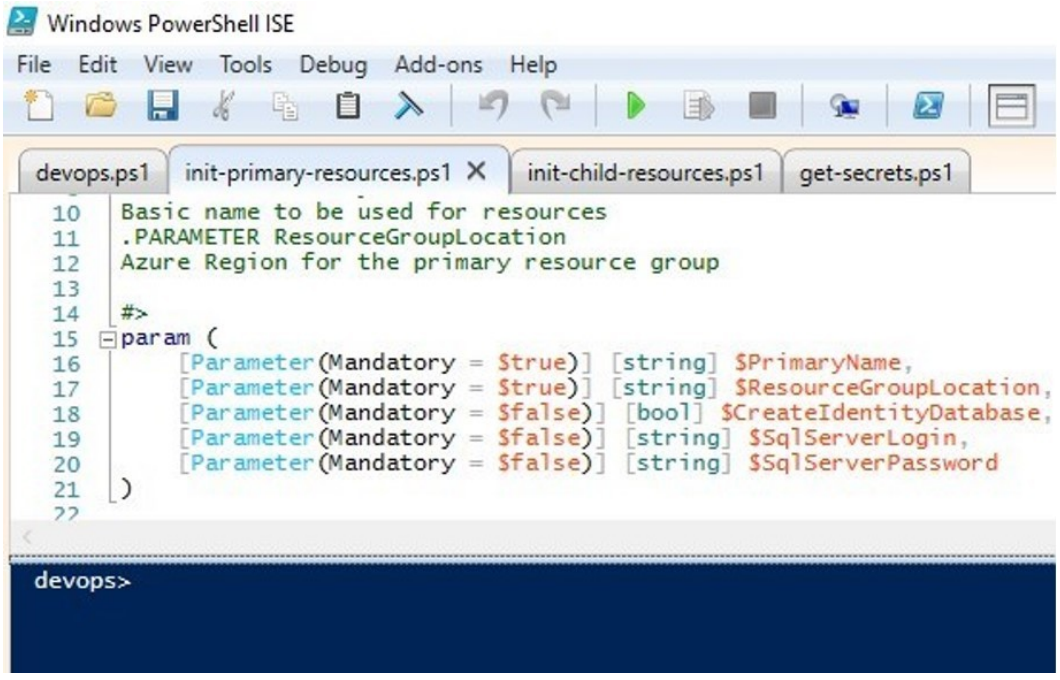
You could catch some of those errors before starting the provisioning process by checking the Products available by region page. The following regions have been tested many times during the development of `Online.Store` app and are highly recommended

1. westcentralus
2. southcentralus
3. eastus
4. westeurope
5. southeastasia

Open the `init-primary-resources.ps1` script to examine it.

Opening PowerShell scripts

A very usefull tool for viewing and running PowerShell scrips is the Windows Powershell ISE. Most of the times you will find it installed along with PowerShell in Windows OS. On this `Integrated Scripting Environment` you can have the script opened in one editor and viewing its results in the integrated command line. More over you get a powerfull intellisense experience.



Windows Powershell ISE

Create Primary Resource Group

```
Get-AzureRmResourceGroup -Name $PrimaryName -ev notPresent -ea 0

if ($notPresent)
{
    # ResourceGroup doesn't exist
    Write-Host "Trying to create Resource Group: $PrimaryName "
    New-AzureRmResourceGroup -Name $PrimaryName -Location $ResourceGroupLocation
}
else
{
    # ResourceGroup exist
    Write-Host "Resource Group:  $PrimaryName  already exists.."
}
```

Get-AzureRmResourceGroup **cmdlet** is used to get the resource group named `PrimaryName` in your account. If not exists New-AzureRmResourceGroup is used to create one at the location

you provided. Notice that all your scripts should be `idempotent` meaning that you can run the same script multiple times but it will only make the changes once. In other words, if the resource group already exists it will not try to create it again. The same logic must be followed for every Azure Resource provisioning.

Create Storage Account

The storage account hosts the images for `Online.Store` application. Any containers required *(such as product-images if you recall)* are created at application startup.

```
$storageAccountName = "$PrimaryName" + "$storagePrefix";

Get-AzureRmStorageAccount -ResourceGroupName $PrimaryName `
                          -Name $storageAccountName -ev storageNotPresent -ea 0

if ($storageNotPresent)
{
    Write-Host "Creating Storage Account $storageAccountName"
    $skuName = "Standard_GRS"

    # Create the storage account.
    $storageAccount = New-AzureRmStorageAccount -ResourceGroupName $PrimaryName `
      -Name $storageAccountName `
      -Location $ResourceGroupLocation `
      -SkuName $skuName

    Write-Host "Storage Account $storageAccountName successfully created.."
}
else
{
    Write-Host "Storage Account $storageAccountName already exists.."
}
```

Get-AzureRmStorageAccount is used to retrieve the storage account named `PrimaryName` in your primary resource group. If not exists New-AzureRmStorageAccount cmdlet is used to create one.

SKU name

Specifies the SKU name of the storage account that this cmdlet creates. The acceptable values for this parameter are:

- **Standard_LRS**: Locally-redundant storage
- **Standard_ZRS**: Zone-redundant storage
- **Standard_GRS**: Geo-redundant storage
- **Standard_RAGRS**: Read access geo-redundant storage
- **Premium_LRS**: Premium locally-redundant storage

Create CDN Profile

Following the storage account the script creates the `CDN Profile` *bound* to that storage account. Not only it provisions the CDN Profile but also configures it to cache the **blobs** on the storage account created before. This is how images in `Online.Store` application will be served by POPs of your CDN Profile.

```
$cdnProfileName = "$PrimaryName-$cdnPrefix";

Get-AzureRmCdnProfile -ProfileName $cdnProfileName `
    -ResourceGroupName $PrimaryName -ev cdnNotPresent -ea 0
if ($cdnNotPresent)
{
    Write-Host "Creating CDN profile $cdnProfileName.."
    # Create a new profile
    New-AzureRmCdnProfile -ProfileName $cdnProfileName `
                          -ResourceGroupName $PrimaryName `
                          -Sku Standard_Verizon -Location $ResourceGroupLocation

    Write-Host "CDN profile $cdnProfileName succesfully created.."

    # Create a new endpoint
    $cdnEnpointName = "$PrimaryName-$endpointPrefix";
    $endpointHost =  "$storageAccountName.blob.core.windows.net"

    $availability =
        Get-AzureRmCdnEndpointNameAvailability -EndpointName $cdnEnpointName
```

```
    if($availability.NameAvailable) {
        Write-Host "Creating endpoint..."

        New-AzureRmCdnEndpoint -ProfileName $cdnProfileName `
         -ResourceGroupName $PrimaryName `
         -Location $ResourceGroupLocation -EndpointName $cdnEnpointName `
         -OriginName "$storageAccountName" `
         -OriginHostName $endpointHost `
         -OriginHostHeader $endpointHost
    }
}
else
{
    Write-Host "CDN profile $cdnProfileName already exists.."
}
```

Get-AzureRmCdnProfile cmd checks if the profile already exists. In not New-AzureRmCdnProfile is used to create one. Next you need to add a CDN endpoint that points to the `Blob Service Endpoint` of your storage account. The URI of this endpoint is <primary-name>storage.blob.core.windows. Before adding this endpoint using New-AzureRmCdnEndpoint, it checks endpoint's availability using the Get-AzureRmCdnEndpointNameAvailability cmdlet.

CDN Endpoint & Storage Account

You need to set `OriginHostHeader` equal to `OriginHostName` in the `New-AzureRmCdnEndpoint` cmdlet for storage account endpoints

Create Cosmos DB account

The script continues with creating the `Azure Cosmos DB account`. The database account name matches the `DocumentDB:DatabaseId` property in the **appsettings.json** file.

```
$documentDbDatabase = "$PrimaryName-$cosmosDbPrefix";

$query = Find-AzureRmResource -ResourceNameContains $documentDbDatabase `
    -ResourceType "Microsoft.DocumentDb/databaseAccounts"

if (!$query)
{
    Write-Host "Creating DocumentDB account $documentDbDatabase.."
    # Create the account

    # Write and read locations and priorities for the database
    $locations = @(@{"locationName"= $ResourceGroupLocation;
                     "failoverPriority"=0})

    # Consistency policy
    $consistencyPolicy = @{"defaultConsistencyLevel"="Session";}

    # DB properties
    # https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels
    $DBProperties = @{"databaseAccountOfferType"="Standard";
                              "locations"=$locations;
                              "consistencyPolicy"=$consistencyPolicy}

    # Create the database
    New-AzureRmResource -ResourceType "Microsoft.DocumentDb/databaseAccounts" `
                        -ApiVersion "2015-04-08" `
                        -ResourceGroupName $PrimaryName `
                        -Location $ResourceGroupLocation `
                        -Name $documentDbDatabase `
                        -PropertyObject $DBProperties

    Write-Host "DocumentDB account $documentDbDatabase succesfully created.."
}
else
{
    Write-Host "DocumentDB account $documentDbDatabase already exists.."
}
```

The script searches for a resource of type `Microsoft.DocumentDb/databaseAccounts` using the Find-AzureRmResource cmdlet. You can use this cmdlet to search for many resource types in Azure. If not found, it creates the `Cosmos DB database account` using the New-AzureRmResource cmdlet. The `$locations` property determines if you wish to replicate data

in other regions and their failover priorities. You will replicate the account's data in a different region through the portal where you can also visually see the available locations at that time.

Create Traffic Manager profile

Last but not least, the script creates the `Traffic Manager profile`. The Traffic Manager profile will be used to control the distribution of traffic to your Azure website endpoints.

```
$tmpProfileName = "$PrimaryName";
$tmpDnsName = "$PrimaryName";

Get-AzureRmTrafficManagerProfile -Name $tmpProfileName `
    -ResourceGroupName $PrimaryName -ev tmpNotPresent -ea 0
if($tmpNotPresent) {
    Write-Host "Creating Traffic Manager Profile $tmpProfileName.."

    New-AzureRmTrafficManagerProfile -Name $tmpProfileName `
    -ResourceGroupName $PrimaryName -TrafficRoutingMethod Performance `
    -RelativeDnsName $tmpDnsName -Ttl 30 -MonitorProtocol HTTP `
    -MonitorPort 80 -MonitorPath "/"

    Write-Host "Traffic Manager Profile created successfully.."
}
else {
    Write-Host "Traffic Maanger $tmpProfileName already exists.."
}
```

The script checks if the traffic manager profile exists using the Get-AzureRmTrafficManagerProfile cmdlet and if not, it creates it using New-AzureRmTrafficManagerProfile. At this point you don't have any App Services to add. The App Service endpoints are added to the Traffic Manager profile at the time being provisioned as `disabled` *(more on this in Child Resources section).*

Create SQL Server and Database for ASP.NET Identity *(optional)*

If you pass the parameter `CreateIdentityDatabase` as `$true`, the script will create a logical SQL Server named <PrimaryName>-sql.

```
$serverName =  "$PrimaryName-$sqlServerPrefix";
$resourceGroupName = $PrimaryName;

$serverInstance = Get-AzureRmSqlServer -ServerName $serverName `
                                       -ResourceGroupName $resourceGroupName `
                                       -ErrorAction SilentlyContinue
if ($serverInstance) {
    Write-Host "SQL Server $serverName already exists..."
}
else {
    Write-Host "Trying to create SQL Server $serverName.."

    New-AzureRmSqlServer -ResourceGroupName $resourceGroupName `
    -ServerName $serverName `
    -Location $ResourceGroupLocation `
    -SqlAdministratorCredentials $(New-Object `
        -TypeName System.Management.Automation.PSCredential `
        -ArgumentList $SqlServerLogin, $(ConvertTo-SecureString `
        -String $SqlServerPassword -AsPlainText -Force))

    Write-Host "SQL Server $serverName successfully created..."

    # Allow access to Azure Services
    Write-Host "Allowing access to Azure Services..."

    New-AzureSqlDatabaseServerFirewallRule -ServerName $serverName `
                                           -AllowAllAzureServices
}
```

The script checks if the SQL Server exists using the Get-AzureRmSqlServer cmdlet and if not it creates it using New-AzureRmSqlServer. At the end adds a new Firewall Rule so that Azure Services such as App Services can access the server. In case `CreateIdentityDatabase` is `$true` the script will continue by checking the `identitydb` database.

```
$Database = "identitydb";

if($CreateIdentityDatabase) {
    $azureDatabase = Get-AzureRmSqlDatabase `
        -ResourceGroupName $resourceGroupName `
        -ServerName $serverName -DatabaseName $Database `
        -ErrorAction SilentlyContinue

    if ($azureDatabase) {
        Write-Host "Azure SQL Database $Database already exists..."
    }
    else {
        Write-Host "Creating SQL Database $Database at Server $serverName.."

        New-AzureRmSqlDatabase  -ResourceGroupName $resourceGroupName `
        -ServerName $serverName `
        -DatabaseName $Database `
        -RequestedServiceObjectiveName "Basic" `
        -MaxSizeBytes 524288000

        Write-Host "Azure SQL Database $Database successfully created..."

    }
}
```

First it checks if `identitydb` database exists on the previous created server using the Get-AzureRmSqlDatabase cmdlet and if not, it creates it using New-AzureRmSqlDatabase .

Running the script

While in `PowerShell` make sure that your working directory is the `Online.Store/App_-Data/devops` folder where all the scripts exist.

Create a `devops` script

To make it easier for you running the scripts in the `Online.Store/App_-Data/devops` folder, create a `PowerShell` script named **devops.ps1** inside that folder. Don't worry, this file will not be tracked by Git *(check .gitignore)*. There you can write all the calls to your scripts you want to run for the `Online.Store` application.

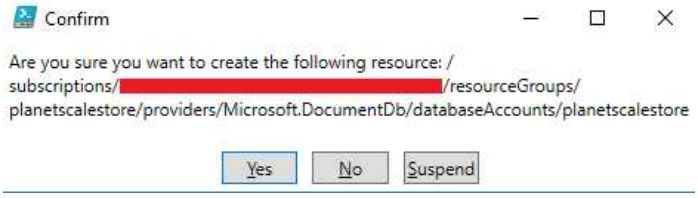
Run the `init-primary-resources.ps1` script as follow:

```
.\init-primary-resources.ps1 -PrimaryName "<primary-name>" `
                             -ResourceGroupLocation "<region>"
```

Make sure to use your primary name and the region you want the resources to provision to. For primary name `planetscalestore` and region `westeurope` the call would look like this:

```
.\init-primary-resources.ps1 -PrimaryName "planetscalestore" `
                             -ResourceGroupLocation "westeurope"
```

During the creation of the Cosmos DB account you will be asked to confirm the creation. Click `Yes` to continue.



Create Azure Cosmos DB account confirmation

Creating Azure Cosmos DB for the first time

The **first time** you create an `Azure Cosmos DB` in a subscription using the Azure portal, the portal registers the `Microsoft.DocumentDB` namespace for that subscription. If you attempt to create the first Cosmos DB account in a subscription using `PowerShell`, you must first register the namespace using the following command:

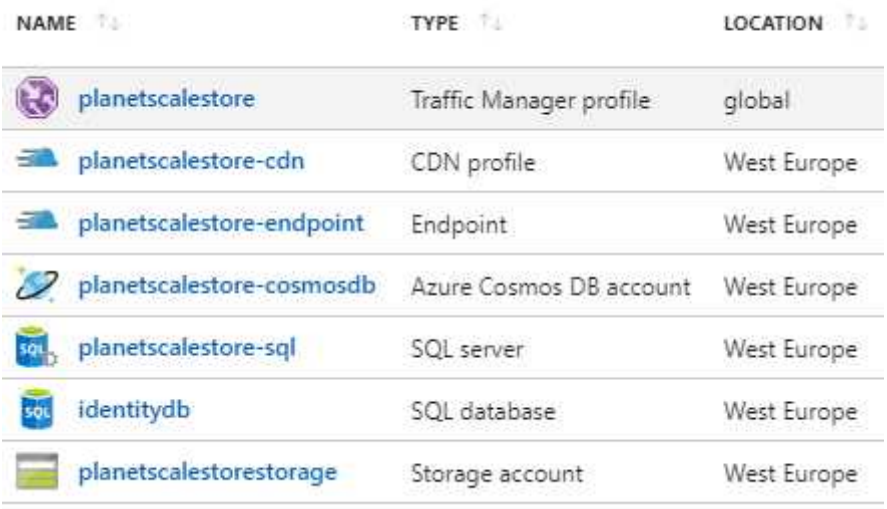
```
Register-AzureRmResourceProvider -ProviderNamespace "Microsoft.DocumentDB"
```

otherwise the script will fail to create the account. If you have followed along with the book, you have already created at least one `Azure Cosmos DB` account so you don't need to worry about that

At the end of each script you will here a `beep` sound indicating that the script has been executed. In case you want to use ASP.NET Identity for authentication in `Online.Store` application run the script as follow:

```
.\init-primary-resources.ps1 -PrimaryName "<primary-name>" `
                             -ResourceGroupLocation "<region>" `
                             -CreateIdentityDatabase $true `
                             -SqlServerLogin "<sql-server1-login>" `
                             -SqlServerPassword "<sql-server-password>"
```

Go back in the portal and confirm that all resources have been provisioned properly.



NAME	TYPE	LOCATION
planetscalestore	Traffic Manager profile	global
planetscalestore-cdn	CDN profile	West Europe
planetscalestore-endpoint	Endpoint	West Europe
planetscalestore-cosmosdb	Azure Cosmos DB account	West Europe
planetscalestore-sql	SQL server	West Europe
identitydb	SQL database	West Europe
planetscalestorestorage	Storage account	West Europe

Primary Resource Group

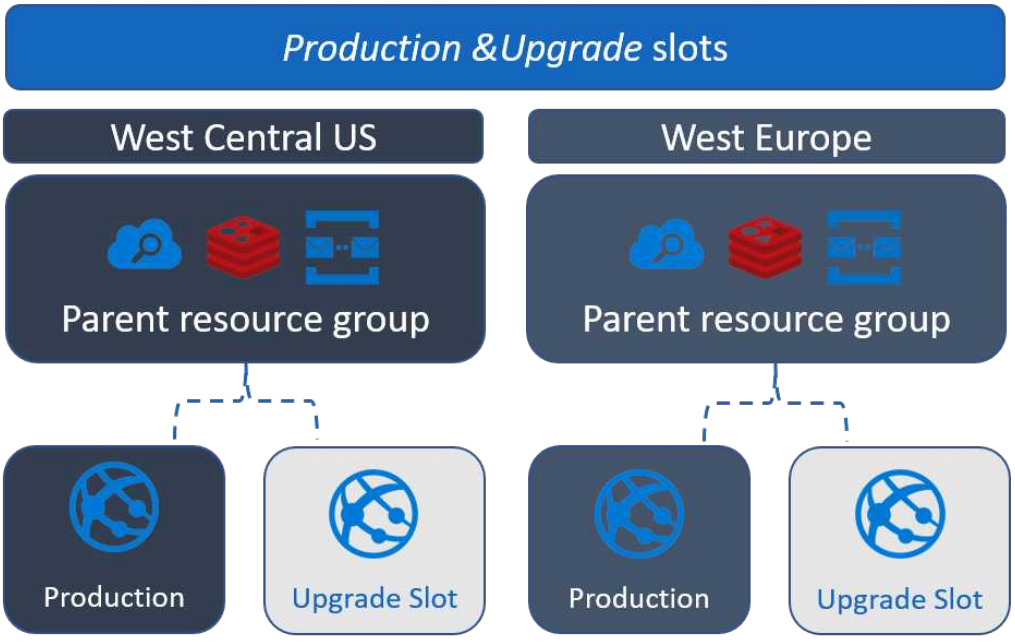
The script will create the SQL Server plus the `identitydb` database. Don't forget to run the **identity-migrations.sql** SQL script to update the schema in the `identitydb` database. This database will be a stand-alone database used by all regions for authenticating users.

Parent Resource Group

A `Parent Resource Group` in `Online.Store` application hosts the following resource types used by web applications deployed in the same region:

1. Service Bus
2. Redis Cache
3. Search Engine

A parent resource group, as you will find out in the `Release Process` section, plays a **crusial** role for achieving *Business continuity* during software updates. When starting a new release you deploy your updates in an App Service Slot which connects to the same parent resouce group resources with the production one. Which means that all cart items exist in the `Redis Cache` or all messages in the `Service Bus` queue will be able to be processed by the new version of your software deployed in the slot.



App Services slots - Connection to same parent resources

Init Parent Resources Script

Open the **init-parent-resources.ps1** PowerShell file to examine its contents. The script accepts the same two parameters:

1. **PrimaryName**: The primary name to be used
2. **ResourceGroupLocation**: Azure region for the resource group and its resources

The `PrimaryName` parameter will be used to constuct the parent's resource group name and hence the name for the resources in it *(following the naming convesion)*. The parent's resource group name is equal to:

```
$resourceGroupName = "$PrimaryName-" + "$ResourceGroupLocation";
```

For `PrimaryName` = *planetscalestore* and `ResourceGroupLocation` = *westeurope* the result name will be **planetscalestore-westeurope**. You can think this naming convension as a type of `hierarchy` where you can quickly understand where a specific resource belongs to.

- <PrimaryName>: Highest level
- <ParentName>: Middle level
- <ChildName>: Lowest level

Create Parent Resource Group

The script starts again by creating the Resource Group for the parent resources. The code is identical with the one you saw for the primary resource group.

```
Get-AzureRmResourceGroup -Name $resourceGroupName -ev notPresent -ea 0

if ($notPresent)
{
    # ResourceGroup doesn't exist
    Write-Host "Trying to create Resource Group: $resourceGroupName "
    New-AzureRmResourceGroup -Name $resourceGroupName `
                             -Location $ResourceGroupLocation
}
else
{
    # ResourceGroup exist
    Write-Host "Resource Group:  $resourceGroupName  already exists.."
}
```

Create Service Bus namespace

The script continues with the `Service Bus` namespace. Not only provisions the namespace but also creates the *orders* queue with the required *Read* and *Write* access policies described in the `Service Bus` section. Let's examine it step by step.

```
$serviceBusNameSpace
    = "$PrimaryName-$ResourceGroupLocation-$serviceBusPrefix";
$serviceBusExists = Test-AzureName -ServiceBusNamespace $serviceBusNameSpace
# Check if the namespace already exists or needs to be created
if ($serviceBusExists)
{
    # Report what was found
    Get-AzureRMServiceBusNamespace -ResourceGroup $resourceGroupName `
                                   -NamespaceName $serviceBusNameSpace
}
```

```
else
{
    New-AzureRmServiceBusNamespace -ResourceGroup $resourceGroupName `
     -NamespaceName $serviceBusNameSpace -Location $ResourceGroupLocation

    $namespace = Get-AzureRMServiceBusNamespace `
                        -ResourceGroup $resourceGroupName `
                        -NamespaceName $serviceBusNameSpace
# code omitted
```

After constructing the name for the `Service Bus` namespace it checks if already exists using the Test-AzureName cmdlet. If it does, reports the details using Get-AzureRMServiceBusNamespace. If the namespace not found it creates it using the New-AzureRmServiceBusNamespace cmdlet. Notice that after creating it, it retrieves a reference to it to be used later for the *orders* queue.

Create Service Bus queue

After creating the namespace, the script creates the *orders* queue.

```
$queueName = "orders"
$ordersQueue = Get-AzureRmServiceBusQueue -ResourceGroup $resourceGroupName `
                                          -NamespaceName $serviceBusNameSpace `
                                          -QueueName $queueName `
                                          -ErrorAction SilentlyContinue

if($ordersQueue)
{
    Write-Host "The queue $queueName already exists.."
}
else
{
    Write-Host "The $queueName queue does not exist."
    Write-Host "Creating the $queueName queue..."
    New-AzureRmServiceBusQueue -ResourceGroup $resourceGroupName `
        -NamespaceName $serviceBusNameSpace -QueueName $queueName `
        -EnablePartitioning $True
    $ordersQueue = Get-AzureRmServiceBusQueue -ResourceGroup $resourceGroupName `
                    -NamespaceName $serviceBusNameSpace -QueueName $queueName
}
```

First it checks if the *orders* queue exists using the Get-AzureRmServiceBusQueue cmdlet. If not found, it creates it using the New-AzureRmServiceBusQueue and also gets a refence to it in order to set the access policies later on.

Create Service Bus queue authorization rules

The script continues by creating the access policies for sending and listening messages from/to the *orders* queue. The access keys for these policies will be used by the following scripts:

1. **Write** access policy: `init-child-resources` script for setting Application Settings in App Service
2. **Read** access policy: `deploy-webjob` script for setting the `ServiceBus:ReadAccessKey` value in the `Online.Store.WebJob` project before publishing it

```
$writeRule = "write"
$readRule = "read"

$authWriteRule
    = Get-AzureRmServiceBusAuthorizationRule -ResourceGroup $resourceGroupName `
                                             -Namespace $serviceBusNameSpace `
                                             -Queue $queueName -Name $writeRule `
                                             -ErrorAction SilentlyContinue
if(!$authWriteRule) {
    "Write Rule not found. Creating rule in namespace $serviceBusNameSpace.."
    New-AzureRmServiceBusAuthorizationRule -ResourceGroup $resourceGroupName `
                                           -Namespace $serviceBusNameSpace `
                        -Queue $queueName -Name $writeRule -Rights @("Send")
}
else {
    Write-Host "Write Rule already exists..."
}

$authReadRule
    = Get-AzureRmServiceBusAuthorizationRule -ResourceGroup $resourceGroupName `
                                             -Namespace $serviceBusNameSpace `
                                             -Queue $queueName -Name $readRule `
                                             -ErrorAction SilentlyContinue
if(!$authReadRule) {
```

```
    "Read Rule not found. Creating rule in namespace $serviceBusNameSpace.."
    New-AzureRmServiceBusAuthorizationRule -ResourceGroup $resourceGroupName `
                                           -Namespace $serviceBusNameSpace `
                        -Queue $queueName -Name $readRule -Rights @("Listen")
}
else {
    Write-Host "Read Rule already exists..."
}
```

The pattern for creating the `Write` and `Read` rules is the same. First it checks if the rule exists using the Get-AzureRmServiceBusAuthorizationRule cmdlet. This cmdlet needs to know the queue and the name of the rule that is looking for. If the rule not found it creates it using the New-AzureRmServiceBusAuthorizationRule cmdlet. The most important parameter for this cmdlet is the `Rights` where you define if you wish to *Send*, *Read* or *Manage* messages in the queue. Possible values are:

1. Listen
2. Send
3. Manage

Create Redis Cache

Each parent resource group has a `Redis Cache` instance to keep track and save the user carts before checking out. Creating a `Redis Cache` through PowerShell is quite easy:

```
$cacheName
    = "$PrimaryName-$ResourceGroupLocation-$redisCachePrefix";

$redisCache = Get-AzureRmRedisCache -Name $cacheName `
    -ResourceGroupName $resourceGroupName -ErrorAction SilentlyContinue

if($redisCache) {
    Write-Host "Redis Cache $cacheName already exists.."
}
else {
    Write-Host "The $cacheName Redis Cache does not exist."
    Write-Host "Creating the $cacheName Redis Cache..."

    New-AzureRmRedisCache -ResourceGroupName $resourceGroupName `
                        -Name $cacheName -Location $ResourceGroupLocation `
```

```
                        -Size 250MB

    Write-Host "$cacheName Redis Cache successfully created.."
    Get-AzureRmRedisCache -Name $cacheName -ResourceGroupName $resourceGroupName
}
```

Get-AzureRmRedisCache cmdlet is used to get a reference to the Redis Cache instance. If it doesn't exist the script creates it using the New-AzureRmRedisCache.

Creating Redis Cache for the first time

The **first time** you create a `Redis cache` in a subscription using the Azure portal, the portal registers the `Microsoft.Cache` namespace for that subscription. If you attempt to create the first Redis cache in a subscription using `PowerShell`, you must first register the namespace using the following command:

```
Register-AzureRmResourceProvider -ProviderNamespace "Microsoft.Cache"
```

otherwise cmdlets such as `New-AzureRmRedisCache` and `Get-AzureRmRedisCache` fail.

If you have followed along with the book, you have already created at least one `Redis Cache` instance in the related chapter so you don't need to worry about that

Create Search Service

The last service the `init-parent-resources.ps` script creates is the `Search Service`. This service provides fast and accurate results when searching for products. Each region must have its own instance of `Search Service`.

```
$searchEngineName
    = "$PrimaryName-$ResourceGroupLocation-$searchServicePrefix";
$sku = "basic" # or "basic" or "standard" for paid services

$query = Find-AzureRmResource -ResourceNameContains $searchEngineName `
                        -ResourceType "Microsoft.Search/searchServices"

if (!$query) {

# Create a new search service
    Write-Host "Creating Search Engine $searchEngineName..."

    New-AzureRmResourceGroupDeployment `
    -ResourceGroupName $resourceGroupName `
    -TemplateUri "https://gallery.azure.com/artifact/20151001/Microsoft.Search.1.0.9/\
DeploymentTemplates/searchServiceDefaultTemplate.json" `
    -NameFromTemplate $searchEngineName `
    -Sku $sku `
    -Location $ResourceGroupLocation `
    -PartitionCount 1 `
    -ReplicaCount 1
}
else {
    Write-Host "Search Engine $searchEngineName already exists.."

    Get-AzureRmResource `
    -ResourceType "Microsoft.Search/searchServices" `
    -ResourceGroupName $resourceGroupName `
    -ResourceName $searchEngineName `
    -ApiVersion "2015-08-19" `
}
```

The script searches for a resource of type `Microsoft.Search/searchServices` using the Find-AzureRmResource cmdlet. If not found it creates it using the New-AzureRmResourceGroupDeployment.

Search Service SKU

The SKU parameter is the pricing tier to select when provisioning a `Search Service`. There are three available options with multiple configurations and capacities:

1. Free
2. Basic
3. Standard

The following table provides full descriptions for each of the above tiers.

Tier	Primary scenarios
Free	A shared service, at no charge, used for evaluation, investigation, or small workloads. Because it's shared with other subscribers, query throughput and indexing varies based on who else is using the service. Capacity is small (50 MB or 3 indexes with up 10,000 documents each).
Basic	Small production workloads on dedicated hardware. Highly available. Capacity is up to 3 replicas and 1 partition (2 GB).
S1	Standard 1 supports flexible combinations of partitions (12) and replicas (12), used for medium production workloads on dedicated hardware. You can allocate partitions and replicas in combinations supported by a maximum number of 36 billable search units. At this level, partitions are 25 GB each.
S2	Standard 2 runs larger production workloads using the same 36 search units as S1 but with larger sized partitions and replicas. At this level, partitions are 100 GB each.
S3	Standard 3 runs proportionally larger production workloads on higher end systems, in configurations of up to 12 partitions or 12 replicas under 36 search units. At this level, partitions are 200 GB each.
S3 HD	Standard 3 High Density is designed for a large number of smaller indexes. You can have up to 3 partitions, at 200 GB each.

Running the script

While in `PowerShell` make sure that your working directory is the `Online.Store/App_-Data/devops` folder where all the scripts exist.

Run the `init-parent-resources.ps1` script as follow:

```
.\init-parent-resources.ps1 -PrimaryName "<primary-name>" `
                            -ResourceGroupLocation "<region>"
```

Make sure to use the **same** primary name you used when you run the `init-primary-resources` so you can follow the `naming-convension`. For primary name `planetscalestore` and region `westeurope` the call would look like this:

```
.\init-parent-resources.ps1 -PrimaryName "planetscalestore" `
                            -ResourceGroupLocation "westeurope"
```

Parent Resource Group

Child Resource Group

A `Child Resource Group` in `Online.Store` application *hosts* the following 3 resource types:

1. App Service Plan
2. App Service
3. SQL Server
4. SQL Database

The name of a `child resource group` is `<Primary-Name>-<region>-app` or in other words `<Parent-Name>-app.Online.Store` web application is deployed in `App Services` provisioned in child resource groups around the globe. Each `App Service` connects to the database provisioned in the same child resource group for **Read-Only** access. There may be many child resource groups hosting a group of an App Service, SQL Server and an SQL Database but only one of them hosts the **Read-Write** access database. The *init-child-resources.ps1* script accepts a boolean `CreateDatabase` parameter that defines if this is the **main** child resource group that hosts the `Read-Write` database. If it's not the main child resource group then the database for that resource group will be created as a `Read Only` secondary database using a different script.

Init Child Resources Script

Open the **init-child-resources.ps1** PowerShell file to examine its contents. The purpose of the script is to create the resources and also configure the `App Service` settings as well. The script accepts the following parameters:

1. **PrimaryName**: The primary name to be used
2. **ResourceGroupLocation**: Azure region for the resource group and its resources
3. **SqlServerVersion**: Version for SQL Servers *(a or b)*
4. **SqlServerLogin**: SQL Server login
5. **SqlServerPassword**: SQL Server password
6. **CreateDatabase**: Create or do not create a Database *(bool)*

7. **Database**: Database name
8. **DocumentDBPolicies**: Priorities for accessing the Cosmos DB account
9. **AzureIDClientSecret**: The client secret for the registered application in the `Azure Active Directory` ***(optional)***
10. **UseIdentity**: `$true` or `$false` to indicate if `ASP.NET Identity` will be used for authentication ***(optional)***
11. **IdentitySqlServerLogin**: Login for accessing the SQL Server in the primary resource group which hosts the `identitydb` database ***(optional)***
12. **IdentitySqlServerPassword** Password for accessing the SQL Server in the primary resource group which hosts the `identitydb` database ***(optional)***

Create Child Resource Group

The script starts by constructing the resource group name and creating the child resource group if it doesn't exist.

```
$resourceGroupName = "$PrimaryName-$ResourceGroupLocation-$Version";

Get-AzureRmResourceGroup -Name $resourceGroupName -ev notPresent -ea 0

if ($notPresent)
{
    # ResourceGroup doesn't exist
    Write-Host "Trying to create Resource Group: $resourceGroupName "
    New-AzureRmResourceGroup -Name $resourceGroupName `
                             -Location $ResourceGroupLocation
}
else
{
    # ResourceGroup exist
    Write-Host "Resource Group:  $resourceGroupName  already exists.."
}
```

Create App Service Plan

Next it creates the `App Service Plan` that will *host* the actual `App Service`.

```
$appServicePlan = "$resourceGroupName-$appServicePlanPrefix";
$webappName = "$resourceGroupName";

Get-AzureRmAppServicePlan -ResourceGroupName $resourceGroupName `
                          -Name $appServicePlan  -ev planNotPresent -ea 0

if ($planNotPresent)
{
    # App Service Plan doesn't exist
    Write-Host "Trying to create App Service Plan $resourceGroupName "
    New-AzureRmAppServicePlan -Name $appServicePlan `
                              -Location $ResourceGroupLocation `
                              -ResourceGroupName $resourceGroupName `
                              -Tier Standard -WorkerSize Small
}
else
{
    # App Service Plan exist
    Write-Host "App Service Plan:  $appServicePlan  already exists.."
}
```

It looks for an `App Service Plan` using the Get-AzureRmAppServicePlan cmdlet and if not found it creates one using the New-AzureRmAppServicePlan.

Create App Service

Following next, the script creates the `App Service`.

```
$appServiceExists = Test-AzureName -Website $webappName

# Check if the namespace already exists or needs to be created
if ($appServiceExists) {
    # App Service Plan doesn't exist
    Write-Host "App Service $webappName already exists.."
}
else {
    # App Service Plan doesn't exist
    Write-Host "Trying to create App Service $webappName "

    New-AzureRmWebApp -Name $webappName `
                      -Location $ResourceGroupLocation `
```

```
                    -AppServicePlan $appServicePlan `
                    -ResourceGroupName $resourceGroupName

    Write-Host "App Service $webappName successfully created..."
}
```

Using the Test-AzureName cmdlet checks if the App Service already exists and if not it creates it using the New-AzureRmWebApp

Add Traffic Manager Endpoint

The script adds the `App Service`'s endpoint to the traffic manager profile provisioned in the primary resource group.

```
$TrafficManagerEndpoint
    = Get-AzureRmTrafficManagerEndpoint -Name $webappName `
                                -ProfileName "$PrimaryName" `
                                -ResourceGroupName $PrimaryName `
                                -Type AzureEndpoints `
                                -ErrorAction SilentlyContinue

if(!$TrafficManagerEndpoint) {

    $webAppResourceId
    = (Get-AzureRmResource -ResourceGroupName $resourceGroupName `
                           -ResourceName $webappName `
                           -ResourceType "Microsoft.web/sites").ResourceId

    Write-Host "Adding new Traffic Manager Endpoint [$webappName]..."

    New-AzureRmTrafficManagerEndpoint -EndpointStatus Disabled `
                -Name $webappName -ProfileName "$PrimaryName" `
                -ResourceGroupName $PrimaryName -Type AzureEndpoints `
                -TargetResourceId "$webAppResourceId"

    Write-Host "Endpoint [$webappName] added successfully.."
} else {
    Write-Host "Traffic Manager Endpoint [$webappName] already exists..."
}
```

First checks if the endpoint of type **AzureEndpoints** already exists in the traffic profile using the Get-AzureRmTrafficManagerEndpoint and if not it adds it as *disabled* using the New-AzureRmTrafficManagerEndpoint. You could add it as *enabled* but there is a different script for enabling or disabling endpoints.

Create Logical SQL Server

In the next step the `init-child-resources.ps1` script creates the `Logical SQL Server`. Each child resource group has a logical SQL Server whose name is <child-resource-group>-sql-<version>. It's important for the script to create the SQL Server at the child's resource group because when you set `Active Geo-Replication` this instance will be used as a parameter.

```
$serverName =  "$resourceGroupName-$sqlServerPrefix-$SqlServerVersion";

$serverInstance = Get-AzureRmSqlServer -ServerName $serverName `
                                       -ResourceGroupName $resourceGroupName `
                                       -ErrorAction SilentlyContinue
if ($serverInstance) {
    Write-Host "SQL Server $serverName already exists..."
}
else {
    Write-Host "Trying to create SQL Server $serverName.."

    New-AzureRmSqlServer -ResourceGroupName $resourceGroupName `
    -ServerName $serverName `
    -Location $ResourceGroupLocation `
    -SqlAdministratorCredentials $(New-Object `
    -TypeName System.Management.Automation.PSCredential `
    -ArgumentList $SqlServerLogin, $(ConvertTo-SecureString `
    -String $SqlServerPassword -AsPlainText -Force))

    Write-Host "SQL Server $serverName successfully created..."

    # Allow access to Azure Services
    Write-Host "Allowing access to Azure Services..."

    New-AzureSqlDatabaseServerFirewallRule -ServerName $serverName `
    -AllowAllAzureServices
}
```

The script checks if the `SQL Server` exists in the child resource group using the Get-AzureRmSqlServer cmdlet. If not, it creates it using the New-AzureRmSqlServer cmdlet. At

the end it creates a new Firewall Rule so that this server can be accessed through Azure Services. Notice that this is the place where the `SqlServerLogin` and `SqlServerPassword` parameters are used.

SQL Server Credentials

It is highly recommended that you use the same credentials for all SQL Servers in the child resource groups.

Create SQL Database *($false or $true)*

The SQL Database provision is **optional** in the `init-child-resources.ps1` script and can be set via the `CreateDatabase` parameter. The reason why this is optional is because **only one database can be Read-Write** in the `Online.Store` globally-distributed application while others will be `Read-Only.Online.Store` application doesn't write directly to the database but just reads from it and only `WebJobs` are allowed to write to the main `Read-Write` database. For the **main** child resource group *(hosts the Read-Write database)* this parameter should be set as `$true` while for the others `$false`. Secondary databases will be created through `SQL Active Geo-Replication` feature by different script.

```
if($CreateDatabase) {
    $azureDatabase
        = Get-AzureRmSqlDatabase -ResourceGroupName $resourceGroupName `
                                 -ServerName $serverName `
                                 -DatabaseName $Database `
                                 -ErrorAction SilentlyContinue

    if ($azureDatabase) {
        Write-Host "Azure SQL Database $Database already exists..."
    }
    else {
        Write-Host "Trying to create Azure SQL Database $Database.."

        New-AzureRmSqlDatabase  -ResourceGroupName $resourceGroupName `
                                -ServerName $serverName `
                                -DatabaseName $Database `
                                -RequestedServiceObjectiveName "Basic" `
                                -MaxSizeBytes 524288000
```

```
        Write-Host "Azure SQL Database $databaseName successfully created..."
    }
}
```

The script checks if the database exists using the Get-AzureRmSqlDatabase cmdlet. If it doesn't it creates it using the New-AzureRmSqlDatabase passing the previous created SQL Server as a parameter. All secondary databases will have the same name as the primary.

App Service Application Settings

One of the most important parts of the script is to automatically set for you all the secret keys in Azure. These are all the properties you set in the `Deployment` section. The script gets the name of the primary's and parent's group which is equal to `PrimaryName` and `<PrimaryName-ResourceGroupLocation>` respectively and retrieves the keys for the following resource types:

1. **Azure CosmosDB Account**: Primary resource group
2. **Storage Account**: Primary resource group
3. **Redis Cache**: Parent resource group
4. **Service Bus**: Parent resource group
5. **Search Service**: Parent resource group

After retrieving the keys *(code omitted)*, updates the settings up on Azure.

```
$settings = @{
    "WEBSITE_NODE_DEFAULT_VERSION" = "6.11.2";
    "DocumentDB:Key" = "$docDbPrimaryMasterKey";
    "DocumentDB:ConnectionPolicies" = "$DocumentDBPolicies";
    "SearchService:Name" = "$searchServiceInfo";
    "SearchService:ApiKey" = "$searchServicePrimaryKey";
    "Storage:AccountKey" = "$storageAccountKey";
    "RedisCache:Endpoint" = "$cacheName";
    "RedisCache:Key" = "$cachePrimaryKey";
    "Region"= "$region";
    "ServiceBus:Namespace" = "$serviceBusNameSpace";
    "ServiceBus:Queue" = "orders";
    "ServiceBus:WriteAccessKeyName" = "write";
    "ServiceBus:WriteAccessKey" = "$writeAccessKey";
```

```
}

$settings.Add("UseIdentity", $UseIdentity.ToString());

if($AzureIDClientSecret) {
  $settings.Add("AzureAd:ClientSecret", "$AzureIDClientSecret");
}

Write-Host "Setting App Settings for $webappName"
$printSettings = ConvertTo-Json $settings -Depth 2
$printSettings
Set-AzureRmWebApp -ResourceGroupName "$resourceGroupName" `
                  -Name "$webappName" `
                  -AppSettings $settings
Write-Host "App Settings updated successfully..."
```

If you wish you can add the following settings in the `$settings` variable as well:

```
"CDN:Endpoint" = "https://$PrimaryName-endpoint.azureedge.net";
"DocumentDB:Endpoint" = "$PrimaryName-$cosmosDbPrefix";
"DocumentDB:DatabaseId" = "$PrimaryName";
"Storage:AccountName" = "$PrimaryName" + "storage";
```

These properties are configured in `appsettings.json` file and are common for all `App Services`.

Database Connection String

Even if you pass `CreateDatabase` as `$false` *(not the main child resource group)* the script will set the `DefaultConnection` connection string on Azure. This is why it's important to pass the correct and the same `Database` parameter for all child resource groups. More over the script will check the `UseIdentity` parameter and if `$true` will add the `IdentityConnection` connection string as well.

```
# Create Hash variable for Connection Strings
$connString = @{}
# Add or Update a desired Connection String within the Hash collection
$connString["DefaultConnection"] =
@{
    Type = "SqlAzure";
    Value = "Server=tcp:$serverName.database.windows.net,1433;Initial Catalog=$Databa\
se;Persist Security Info=False;User ID=$SqlServerLogin;Password=$SqlServerPassword;Mu\
ltipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Tim\
eout=30;"
}

if($UseIdentity -and $IdentitySqlServerLogin -and $IdentitySqlServerPassword) {
    $identityServer = $PrimaryName;
    $IdentityDatabase = "identitydb";
    $connString["IdentityConnection"] = @{ Type = "SqlAzure"; Value = "Server=tcp:$id\
entityServer.database.windows.net,1433;Initial Catalog=$IdentityDatabase;Persist Secu\
rity Info=False;User ID=$IdentitySqlServerLogin;Password=$IdentitySqlServerPassword;M\
ultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Ti\
meout=30;" }
}

Write-Host "Setting Connection String for $webappName"
# Save Connection String to Azure Web App
Set-AzureRmWebAppSlot -ResourceGroupName $resourceGroupName `
                      -Name $webappName -Slot production `
                      -ConnectionStrings $connString
Write-Host "Connection String saved succeffully.."
```

After building the connection string in a `hash variable` updates the App Service connection strings using the Set-AzureRmWebAppSlot cmdlet. Note that you can add more than one connection strings at the `connString` variable.

App Service Always-On

The last property the script configures for your App Service is the `Always-On` property. This setting is required to be `True` for `Continuous WebJobs`.

```
# Set Always-On property True - Required for continuous Web Jobs
$WebAppPropertiesObject = @{"siteConfig" = @{"AlwaysOn" = $true}}
$WebAppResourceType = 'microsoft.web/sites'
$webAppResource = Get-AzureRmResource -ResourceType $WebAppResourceType `
                                      -ResourceGroupName $resourceGroupName `
                                      -ResourceName $webappName

$webAppResource | Set-AzureRmResource -PropertyObject $WebAppPropertiesObject `
                                      -Force
```

The script retrieves the resource of type `microsoft.web/sites` in the child resource group using Get-AzureRmResource cmdlet and sets the property using the Set-AzureRmResource.

Running the script

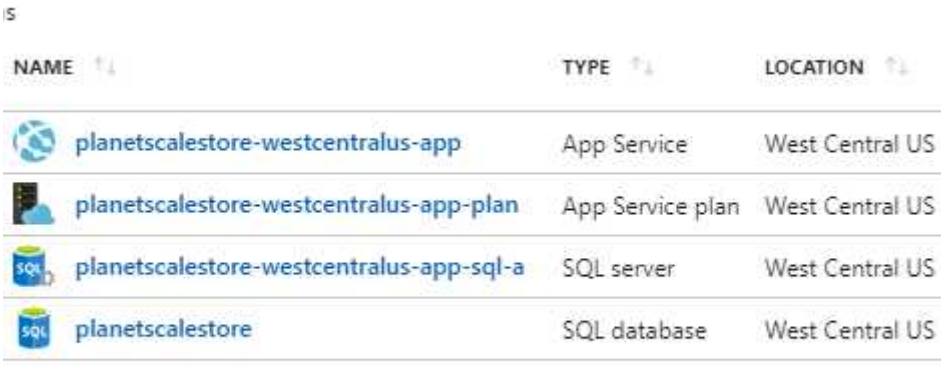
While in `PowerShell` make sure that your working directory is the `Online.Store/App_-Data/devops` folder where all the scripts exist.

For a **main** child resource group that hosts the `Read-Write` database, run the `init-child-resources.ps1` script as follow:

```
.\init-child-resources.ps1 `
                        -PrimaryName "<primary-name>" `
                        -ResourceGroupLocation "<region>" `
                        -SqlServerVersion "a" `
                        -SqlServerLogin "<slq-server-login>" `
                        -SqlServerPassword "<sql-server-password>" `
                        -CreateDatabase $true
                        -Database "<database-name>"
                        -DocumentDBPolicies "<policies-here>" `
                        -UseIdentity $true `
                        -IdentitySqlServerLogin "<identity-server-login>" `
                        -IdentitySqlServerPassword "<identity-server-password>"
```

Following the `PowerShell` example for primary name `planetscalestore` where primary and parent resource groups deployed in `West Europe` the script would look like this:

```
.\init-child-resources.ps1 `
                        -PrimaryName "planetscalestore" `
                        -ResourceGroupLocation "westeurope" `
                        -SqlServerVersion "a" `
                        -SqlServerLogin "<slq-server-login>" `
                        -SqlServerPassword "<sql-server-password>" `
                        -CreateDatabase $true
                        -Database "planetscalestore"
                        -DocumentDBPolicies "West Europe" `
                        -UseIdentity $true `
                        -IdentitySqlServerLogin "<identity-server-login>" `
                        -IdentitySqlServerPassword "<identity-server-password>"
```



Child Resource Group

`DocumentDBPolicies` property is a comma separated string with no spaces, declaring the priorities for accessing the Azure Cosmos DB account in case you replicate its data. The following table shows the available locations for the `DocumentDBPolicies`.

West US	Brazil South	UK South	UK West
Korea Central	Korea South	China East	China North
Germany Northeast	Germany Central	Canada Central	Canada East
West India	South India	Central India	Australia Southeast
Australia East	Japan West	Japan East	Southeast Asia
East Asia	North Europe	West Europe	North Central US
South Central US	Central US	East US 2	East US
West Central US	West US 2	USGov Arizona	USGov Texas

In case you wanted to use **Azure Active Directory** authentication instead of `ASP.NET Core Identity`, run the script as follow.

```
.\init-child-resources.ps1 -PrimaryName "planetscalestore" `
                           -ResourceGroupLocation "westeurope" `
                           -SqlServerVersion "a" `
                           -SqlServerLogin "<slq-server-login>" `
                           -SqlServerPassword "<sql-server-password>" `
                           -CreateDatabase $true
                           -Database "planetscalestore"
                           -DocumentDBPolicies "West Europe" `
                           -AzureIDClientSecret "<AzureID:ClientSecret>"
```

SQL Geo-Replication

Running the `init-child-resources` script with `CreateDatabase = True` means that you wish the new child resource group to be the `main child resource group` which hosts the **Read-Write** access database. Only one child resource group may have a `Read-Write` database while others have **Read-Only** replicas of the main database. The `Read-Only` databases are initialized using the **init-geo-replication.ps1** script. The script accepts the following `4` parameters:

1. Database
2. PrimaryResourceGroupName
3. PrimaryServerName
4. SecondaryResourceGroupName
5. SecondaryServerName

The script gets a reference to the `Database` in the `PrimaryServerName` and creates a secondary `Read-Only` replica database with the same name in the `SecondaryServerName`.

```
$primaryDatabase = Get-AzureRmSqlDatabase -DatabaseName $Database `
                    -ResourceGroupName $PrimaryResourceGroupName `
                    -ServerName $PrimaryServerName

$primaryDatabase | New-AzureRmSqlDatabaseSecondary `
    -PartnerResourceGroupName "$SecondaryResourceGroupName" `
    -PartnerServerName "$SecondaryServerName" -AllowConnections "All"
```

It uses the New-AzureRmSqlDatabaseSecondary cmdlet to create the secondary database and starts data replication at the same time.

Running the script

Assuming you have deployed the main child resource group in `West Europe` and a second child resource group in `West Central US` you can init the replication as follow:

```
# Init Geo-Replication
.\init-geo-replication.ps1 -Database "<primary-name>" `
            -PrimaryResourceGroupName "<primary-name>-westeurope-app" `
            -PrimaryServerName "<primary-name>-westeurope-app-sql-a" `
            -SecondaryResourceGroupName "<primary-name>-westcentralus-app" `
            -SecondaryServerName "<primary-name>-westcentralus-app-sql-a"
```

For primary name `planetscalestore` and child resource groups version `a` the script looks like this.

```
# Init Geo-Replication
.\init-geo-replication.ps1 `
    -Database "planetscalestore" `
    -PrimaryResourceGroupName "planetscalestore-westeurope-app" `
    -PrimaryServerName "planetscalestore-westeurope-app-sql-a" `
    -SecondaryResourceGroupName "planetscalestore-westcentralus-app" `
    -SecondaryServerName "planetscalestore-westcentralus-app-sql-a"
```

The script will create a secondary `Read-Only` database in `West Central US` and also start the replication. This means that you don't have to run the **migrations.sql** file in the secondary

databases *(you cannot run it anyway since they are Read-Only)*, schema and data will be synced asynchronously any time changes happen on the primary database. After running the script, back in the portal you can confirm that geo-replication has been successfully configured as you did in the `SQL Active Geo-Replication` section.

The following figure *(select SQL Databases from the left panel)* shows that a primary database *planetscalestore* provisioned in `West Europe` is being replicated in `West Central US`.

NAME	...	RE...	SERVER
identitydb	Online	None	planetscalestore-sql
planetscalestore	Online	Secondary	planetscalestore-westcentralus-app-sql-a
planetscalestore	Online	Primary	planetscalestore-westeurope-app-sql-a

SQL Geo-Replication

SQL Geo-Replication Map

App Service Deployment

In the Deployment section of the Web Application part you saw how to publish the Online.Store application through Visual Studio. Assuming that you want to provision 3 App Services in 3 different regions, the publishing process can be time consuming. Also recall that you might have to open the Kudu interface one by one for each App Service in order to install npm packages. The real disadvantage though is that you don't have full control of what is being deployed during the process. Ideally you would like to know more details about the deployment such as the build version to be released and if all the tests have been passed before the process starts. All these are possible using AppVeyor, a continuous integration solution capable of making the release process **fast** and more important **reliable**. AppVeyor is free for open-source projects but also provides several plans to meet the needs for enterprise solutions.

In this section you will complete everything remained to have the globally-distributed Online.Store application working properly in Microsoft Azure cloud. Scripts explained in this section:

- AppVeyor integration for automated deployment
- WebJob deployment
- Enabling & Disabling Traffic Manager profile endpoints

Integrating AppVeyor

There are 4 main steps you need to follow to configure a deployment environment for your project in AppVeyor.

1. Add the project to AppVeyor
2. Create a build definition *(appveyor.yml)*
3. Create a deployment environment
4. Publish artifacts

The first two steps, will be configured once while the 3rd and 4th are the actual PowerShell scripts that will automate the deployment process for your App Services. Before moving on to the next steps, visit AppVeyor and sign in. It is recommended that use your GitHub account and if you don't have any simply create one, it's free.

Add project

If you haven't **forked** the planet-scale-azure repository yet, now it's the right time to do so. To fork the repository, sign in to your GitHub account and visit the repository's url. Click the `Fork` button on the upper right and that's it. Back in `AppVeyor`, click `Projects` from the top menu, search for the forked repository in the `GitHub` tab and click the `Add` button on the right.

Add Project to AppVeyor

AppVeyor.yml

`AppVeyor` requires to add **appveyor.yml** file at the root of your repository that defines what do you want it to do with your project when you push changes. This file has been already there for you so go ahead and open it. Following are the most important parts of it.

```
version: '2.0.{build}'
image: Visual Studio 2017
branches:
  only:
  - master
```

The above section tells `AppVeyor` to use the `Visual Studio 2017` build worker image so that it can build the .NET Core `Online.Store` application. An image has a pre-installed software ready to be used for processing your projects. It also informs `AppVeyor` to run a build every time you push a commit in the `master` branch. Any commits to different branches will not trigger a build.

```
# Install scripts. (runs after repo cloning)
install:
  # Get the latest stable version of Node.js or io.js
  - cd Online.Store
  # install npm modules
  - ps: Install-Product node $env:nodejs_version
  - npm install
  - node node_modules/webpack/bin/webpack.js --config webpack.config.vendor.js --env.\
prod
```

The `install` instruction prepares the environment before the actual build starts. After changing the working directory to `Online.Store` web application project, it installs `node` and runs the `npm install` command. Next it runs `webpack` to produce the `vendors` packages file used by the Angular application.

```
before_build:
  # Display minimal restore text
  - cmd: dotnet restore --verbosity m
build_script:
  # output will be in ./Online.Store/bin/Release/netcoreapp2.0/publish/
  - cmd: dotnet publish -c Release
```

Before building the project it restores packages using the .NET Core CLI. Next it runs the `dotnet publish` command to produce the artifacts.

```
artifacts:
 - path: '\Online.Store\bin\Release\netcoreapp2.0\publish'
   name: WebSite
   type: WebDeployPackage
```

When `dotnet publish -c Release` command finishes, the produced artifacts are stored in the `\Online.Store\bin\Release\netcoreapp2.0\publish` folder. The instruction tells `AppVeyor` to name the artifacts `WebSite` and **ZIP** them in a file named `WebSite.zip`. Those artifacts will be available for deployment using the `Web Deploy` method.

Build artifacts

Every build produces its own artifacts. This means that using the build number you can deploy any version of your software at any time you want.

Try and push a simple change to your forked repository. `AppVeyor` will trigger a new build. To view your project's build history, in the `Projects` page click the forked repository and select the `History` tab.

Project's Build History

Web Deploy

You will use the Web Deploy method to publish the artifacts of your builds up on your App Services in Azure. This will be done through `AppVeyor` Deployment Environments. A deployment environment's role is to connect to a specific Azure App Service **slot** and publish the artifacts for the selected build. In order to connect though to an App Service it needs to know the `web deploy`'s credentials for that slot. Those credentials can be found in the **publish profile** of an App Service slot. This means that there will be different deployment environments in AppVeyor for each App Service you create. Before viewing the associated PowerShell script it's good to know how an xml publish profile looks like. In Azure Portal open an `App Service` resource and in the main blade select `Overview`. Click the `Get publish profile` button to download the App Service's publish profile. Part of the **xml** file *(for the planetscalestore-westeurope-app App Service)* looks like this:

```
<publishProfile
    profileName="planetscalestore-westeurope-app - Web Deploy"
    publishMethod="MSDeploy"
    publishUrl="planetscalestore-westeurope-app.scm.azurewebsites.net:443"
    msdeploySite="planetscalestore-westeurope-app"
    userName="$planetscalestore-westeurope-app"
    userPWD="<some-random-password>"
    destinationAppUrl="http://planetscalestore-westeurope-app.azurewebsites.net"
    SQLServerDBConnectionString="<connection-string>"
    mySQLDBConnectionString=""
    hostingProviderForumLink=""
    controlPanelLink="http://windows.azure.com"
    webSystem="WebSites"
    targetDatabaseEngineType="sqlazuredatabase"
    targetServerVersion="Version100">
</publishProfile>
```

AppVeyor's deployment environment needs to know the following properties in order to publish build artifacts to an Azure App Service:

1. publishUrl
2. userName
3. userPWD

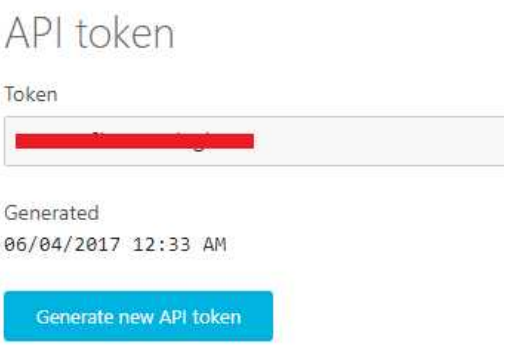
Create deployment environment

The script for creating an AppVeyor deployment environment is the **create-deployment-environment.ps1**. It requires the following 4 parameters:

1. **token**
2. **projectSlug**
3. **webappName**
4. **resourceGroupName**
5. **slot** *(optional)*

The `token` parameter is an AppVeyor's **API token** you need in order to use its REST APIs. To get one, make sure you are authenticated in AppVeyor and click your account on the upper

right. Select `API token` and copy the value you see over there. If no value exists click the `Generate new API token` button.



API token

The `projectSlug` parameter value comes from the project's URL in AppVeyor. More specifically it's the last part, after the account name. Assuming the project's URL in the AppVeyor is *https://ci.appveyor.com/project/chsakell/planet-scale-azure* the `projectSlug` is equal to **planet-scale-azure**. The `webappName` and `resourceGroupName` should be equal to the child's resource group name *(e.g. planetscalestore-westeurope-app)*.

The `slot` parameter is optional and if omitted then it means that you want to create a deployment environment for the default "production" slot. The release process though depends in App Service Slots where the idea is to deploy your updates over there and then swap them with the production slot. This means that an `App Service` will have two deployment slots, one for production and another one for releases.

Production slot

Basically, you never deploy your updates directly to the production slot. Instead, you deploy to a *staging* slot and then swap them

Open a `PowerShell` session and cd to `Online.Store/App_Data/devops` folder. Run the **create-deployment-environment.ps1** script file as follow:

```
.\create-deployment-environment.ps1 `
                        -token "<your-api-token>" `
                        -projectSlug "<project-slug>" `
                        -webappName "<child-resource-group>" `
                        -resourceGroupName "<child-resource-group>"
```

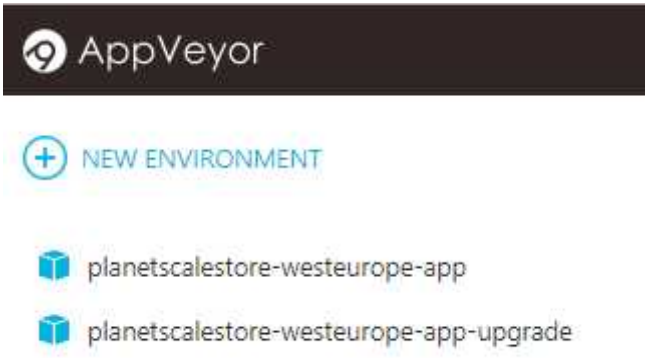
For a primary resource group named `planetscalestore`, an App Service in `West Europe` and the planet-scale-azure AppVeyor repository the call for creating the deployment envriron-ment for the production slot would look like this:

```
.\create-deployment-environment.ps1 `
                        -token "<random-password>" `
                        -projectSlug "planet-scale-azure" `
                        -webappName "planetscalestore-westeurope-app" `
                        -resourceGroupName "planetscalestore-westeurope-app"
```

Later on, in the `Release Process` after creating an App Service slot named `upgrade` you will create a deployment environment for that as well.

```
.\create-deployment-environment.ps1 `
                        -token "<random-password>" `
                        -projectSlug "planet-scale-azure" `
                        -webappName "planetscalestore-westeurope-app" `
                        -resourceGroupName "planetscalestore-westeurope-app" `
                        -slot "upgrade"
```

The only difference is that you added the `slot` parameter which is equal to the slot's name. When the script finishes confirm that the deployment environment has been created successfully. In `AppVeyor` click `Environments` from the top navbar and you should see the new environment.



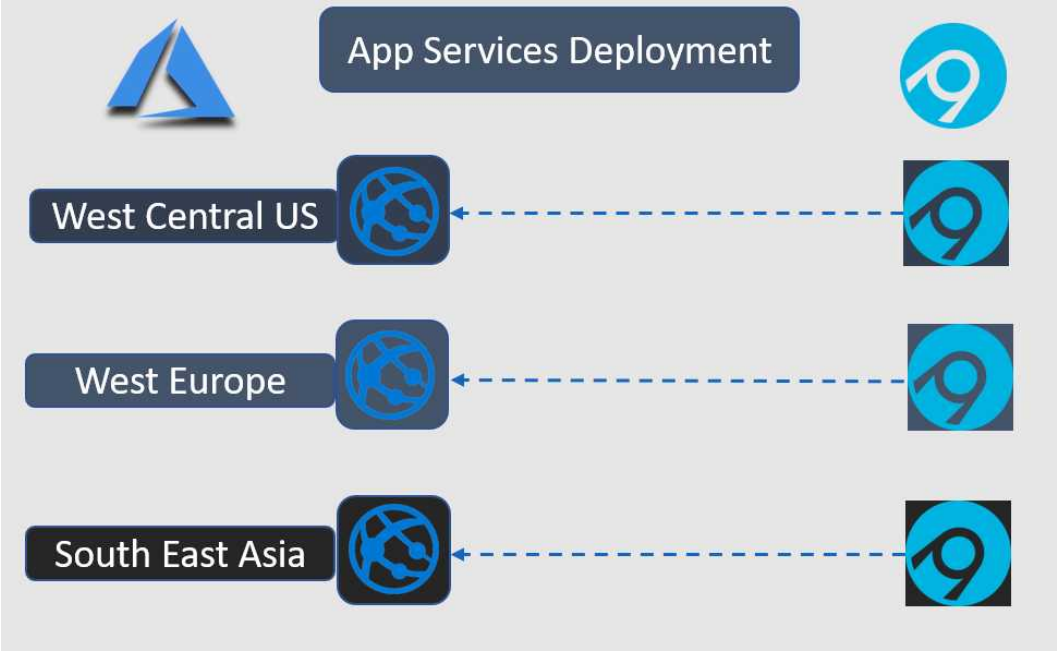
Web Deploy environments

What you see in the above screenshot is 2 different deployment environments for `Online.Store` application deployed in `West Europe`. The first one is the environment for the default production slot and the second one for a slot named `upgrade`.

Click in an environment to view its settings and the deployment history. The `create-deployment-environment.ps1` script is quite large to paste it here but in fact, it's very simple. After retrieving the credentials for publishing the `App Service` using `Web Deploy`, it invokes AppVeyor REST API to create or update a deployment environment.

```
# update environment
$createdEnvironment = Invoke-RestMethod
        -Uri 'https://ci.appveyor.com/api/environments' `
        -Headers $headers -Method Put -Body $updatedEnvironment

# create environment
$createdEnvironment = Invoke-RestMethod
        -Uri 'https://ci.appveyor.com/api/environments' `
        -Headers $headers -Method Post -Body $newEnvironment
```

Start deployment



AppVeyor Deployment Environments

The script for starting a deployment in an App Service is the **start-deployment.ps1**. The script accepts the following 8 parameters:

1. **token**
2. **accountName**
3. **projectSlug**
4. **webappName**
5. **resourceGroupName**
6. **deploymentEnvironment**
7. **build** *(optional)*
8. **slot** *(optional)*

Other that those you saw in the previous script, there are 2 new required parameters to start the deployment. The `accountName` is the **AppVeyor**'s account name and the `deploymentEnvironment` is the name of the deployment environment you want to use during publishing.

The `create-deployment-environment` script names the enviroment same as the `webappName`. You can run the `start-deployment` script as follow:

```
.\start-deployment.ps1  -token "<your-api-token>" -accountName "<account-name>" `
                        -projectSlug "<project-slug>" `
                        -webappName "<child-resource-group>" `
                        -resourceGroupName "<child-resource-group>" `
                        -deploymentEnvironment "<child-resource-group>"
```

An example for starting deploying to an App Service slot named *"upgrade"* in `West Central US` and primary name `planetscalestore` looks like this:

```
.\start-deployment.ps1  `
            -token "<random-token>" -accountName "chsakell" `
            -projectSlug "planet-scale-azure" `
            -webappName "planetscalestore-westcentralus-app" `
            -resourceGroupName "planetscalestore-westcentralus-app" `
            -deploymentEnvironment "planetscalestore-westcentralus-app-upgrade" `
            -slot "upgrade"
```

The script does quite a few things so it's better to break it down. First it gets the latest build in `AppVeyor` by invoking its API.

```
# Get project's last build

$latestBuild = Invoke-RestMethod
    -Uri "https://ci.appveyor.com/api/projects/$accountName/$projectSlug" `
    -Headers $headers -Method Get
```

> **Build number**
>
> If you wish you can the use the exact build version you wish to publish by passing the parameter `-build` which is optional.

Next it starts the actual publishing by using the deployment environment in AppVeyor.

```
# Start Deployment
$body = @{
    environmentName = "$deploymentEnvironment";
    accountName = "$accountName";
    projectSlug = "$projectSlug";
    buildVersion = $latestBuild.build.version;
}

if($build) {
    $body.buildVersion = "$build";
}

ConvertTo-Json $body

$StartDate=(GET-DATE)

$deployment = Invoke-RestMethod
                -Uri 'https://ci.appveyor.com/api/deployments' `
                -Headers $headers -Method Post `
                -Body (ConvertTo-Json $body)
```

Normally this is all you wanted to do but the thing is that starting a deployment in AppVeyor is an asynchronous operation, meaning that the call returns immediately while the publishing still runs. So what the script does is start querying for the deployment **status** till is finished or failed.

```
# Get Deployment - Post Build Events

$deploymentId = $deployment.deploymentId;
$runningDeployment = Invoke-RestMethod `
    -Uri "https://ci.appveyor.com/api/deployments/$deploymentId" `
    -Headers $headers -Method Get
$runningDeploymentStatus = $runningDeployment.deployment.status;

while($runningDeploymentStatus -eq "running" -or `
      $runningDeploymentStatus -eq "queued" ){
    Write-Host "Publishing artifacts to [$webappName]"
    # Sleep for 2 seconds..
    Start-Sleep -s 2
    $runningDeployment = Invoke-RestMethod `
        -Uri "https://ci.appveyor.com/api/deployments/$deploymentId" `
        -Headers $headers -Method Get
```

```
    $runningDeploymentStatus = $runningDeployment.deployment.status;
}
```

When AppVeyor finishes deploying the artifacts *(status: success)* a post deployment action starts. If you recall we needed to run `npm install` in the App Service's Kudu environment to ensure all npm packages are installed. This is done via Kudu REST API where you can simulate the exact thing you did via the interface.

```
Try {
    $response = Invoke-RestMethod -Uri $apiUrl `
        -Headers @{Authorization=("Basic {0}" -f $auth)} -Method Post `
        -ContentType "application/json" -Body $body -TimeoutSec 1500 `
        -ErrorAction Stop

    Write-Output $response
    Write-Output $response.Output
}
Catch {
    Write-Host "Waiting for npm install...";
}
```

The reason why you see a **Try/Catch** block is that if you make a call to Azure API and takes more than about 230 seconds without sending a response, it sends an error 500. So the first time the `npm install` command runs and as always it takes a little bit too long to finish, the catch block will catch the error. An output example of the `start-deployment.ps1` script looks like this.

```
{
    "buildVersion":  "2.0.10",
    "environmentName":  "planetscalestore-westcentralus-app",
    "accountName":  "chsakell",
    "projectSlug":  "planet-scale-azure"
}
Publishing artifacts to [planetscalestore-westcentralus-app] Status: running
Publishing artifacts to [planetscalestore-westcentralus-app] Status: running
Artifacts deployed succesfully..
Retrieving publish profile information..
Invoking 'npm install' on planetscalestore-westcentralus-app ...
.
.
```

```
Deployment finished succesfully..
Total Deployment Time: 11 minutes and 50 seconds
```

While the first time it takes about **11 minutes** for the deployment to finish *(due to `npm install` command)*, it takes less than **2 minutes** for feature deployments.

NPM packages

It will take a while for the `npm install` to finish the first time *(about 10 minutes)*. You can always check if the command has finished in the `Kudu` environment

WebJob Deployment

WebJob deployment should be scripted and automated as well. Before explaining the associated PowerShell script let's review what is the purpose of a WebJob in `Online.Store` application:

- Listen for messages in the *orders* queue in the `Service Bus` namespace belonging to the Parent Resource group
- Parse order messages and save them into `Read-Write` primary database

Having said that, the `deploy-webjob.ps1` script requires the following parameters:

1. PrimaryDatabaseServer
2. Database
3. SqlServerLogin
4. SqlServerPassword
5. WebappParentResourceGroup
6. WebappResourceGroup
7. WebjobAppLocation
8. DeploymentDestinationFolder
9. slot *(optional)*

The first 4 parameters are required to create the DefaultConnection connection string to the primary database. WebappParentResourceGroup is the parent resource group which will be used to locate the Service Bus namespace and start listening for messages in the *orders* queue. The WebappResourceGroup is the child resource group and equal to the App Service name. A WebJob is being deployed to a specific App Service slot. WebjobAppLocation parameter is the **full local path** of the Online.WebJob project on your machine. The last required parameter DeploymentDestinationFolder is a folder in your machine out of the Online.Store solution where the ZIP files will be saved. If slot parameter is omitted then the production slot will be used.

Deploy-WebJob script

First of all the script tries to set the correct settings in your local **appsettings.json** file in the Online.Store.WebJob project. In case you haven't created that file yet, make sure you do so by pasting the contents from the *appsettings.json.template* file.

```
{
  "ConnectionStrings": {
    "DefaultConnection": ""
  },
  "ServiceBus:Namespace": "",
  "ServiceBus:Queue": "orders",
  "ServiceBus:ReadAccessKeyName": "Read",
  "ServiceBus:ReadAccessKey": ""
}
```

If you followed along with the Deployment section in Web Application part, you saw that in order to deploy a WebJob to an App Service you need to do the following:

1. Run dotnet publish -c Release command
2. **ZIP** the artifacts in a file
3. Upload the ZIP file in the WebJobs blade of your App Service

All these are handled for you by the deploy-webjob.ps1 script. After zipping the artifacts in a folder of your preference, the orders.zip file is uploaded through Kudu REST API.

Running the script

You can run the deploy-webjob.ps1 script as follow:

```
.\deploy-webjob.ps1 `
    -PrimaryDatabaseServer "<main-child-resource-group>-sql-<version>" `
    -Database "<database>" `
    -SqlServerLogin "<login>" -SqlServerPassword "<password>" `
    -WebappParentResourceGroup "<parent-resourcegroup>" `
    -WebappResourceGroup "<child-resource-group>" `
    -WebjobAppLocation "C:\path-to\Online.Store.WebJob" `
    -DeploymentDestinationFolder "C:\path-to-zip-folder\"
```

An example for `planetscalestore` and an App Service in `West Europe` looks like this.

```
.\deploy-webjob.ps1 `
    -PrimaryDatabaseServer "planetscalestore-westeurope-app-sql-a" `
    -Database "planetscalestore" `
    -SqlServerLogin "chsakell" `
    -SqlServerPassword "<random-password>" `
    -WebappParentResourceGroup "planetscalestore-westeurope"`
    -WebappResourceGroup "planetscalestore-westeurope-app" `
    -WebjobAppLocation "C:\planet-scale-azure\Online.Store.WebJob" `
    -DeploymentDestinationFolder "C:\webjobs"
```

Confirm that the WebJob has been uploaded successfully by opening your `App Service` resource in Azure Portal and checking the WebJobs list in the `WebJobs` blade.

Traffic Manager Profile Endpoints

When an App Service is provisioned during the `init-child-resources.ps1` script, a Traffic Manager `Azure endpoint` is also created for that App Service. This endpoint is added as disabled meaning that Traffic Manager doesn't direct requests to the endpoint. Each endpoint has the same name with the `child resource group` and hence the App Service. You can use the **set-traffic-manager-endpoint.ps1** PowerShell script to enable or disable endpoints. The script requires the following 4 parameters:

1. **EndpointName**: App Service name *(<child-resource-group>)*
2. **TraficManagerProfile**: Traffic Manager Profile name *(<primary-name>)*
3. **ResourceGroupName**: Traffic Manager Profile resource group *(<primary-resource-group>)*
4. **Enable**: Enable or disable the endpoint. Possible values: $true, $false

```
if($Enable) {

Write-Host "Enabling $EndpointName...";

Enable-AzureRmTrafficManagerEndpoint -Name $EndpointName -Type AzureEndpoints `
                                     -ProfileName $TraficManagerProfile `
                                     -ResourceGroupName $ResourceGroupName
}
else {

Write-Host "Disabling $EndpointName...";

Disable-AzureRmTrafficManagerEndpoint -Name $EndpointName -Type AzureEndpoints `
                                     -ProfileName $TraficManagerProfile `
                                     -ResourceGroupName $ResourceGroupName

}
```

Enable-AzureRmTrafficManagerEndpoint and Disable-AzureRmTrafficManagerEndpoint cmdlets are used to enable or disable the `Azure endpoints` in the Traffic Manager profile respectively.

Running the script

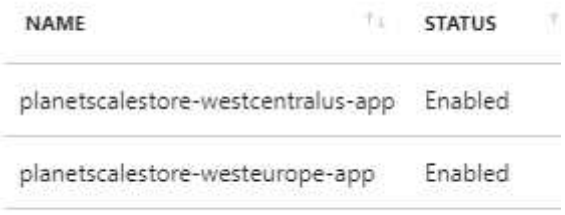
Run the `set-traffic-manager-profile` script as follow:

```
.\set-traffic-manager-endpoint.ps1 -EndpointName "<child-resource-group>" `
                            -TraficManagerProfile "<primary-name>" `
                            -ResourceGroupName "<primary-resource-group>" `
                            -Enable $true
```

For primary name `planetscalestore` and an App Service in `West Europe` the script would look like this.

```
.\set-traffic-manager-endpoint.ps1
                            -EndpointName "planetscalestore-westeurope-app" `
                            -TraficManagerProfile "planetscalestore" `
                            -ResourceGroupName "planetscalestore" `
                            -Enable $true
```

After enabling the endpoints, open the `Traffic Manager Profile` resource in the portal and confirm that they have been enabled.



NAME	STATUS
planetscalestore-westcentralus-app	Enabled
planetscalestore-westeurope-app	Enabled

Traffic Manager profile Endpoints

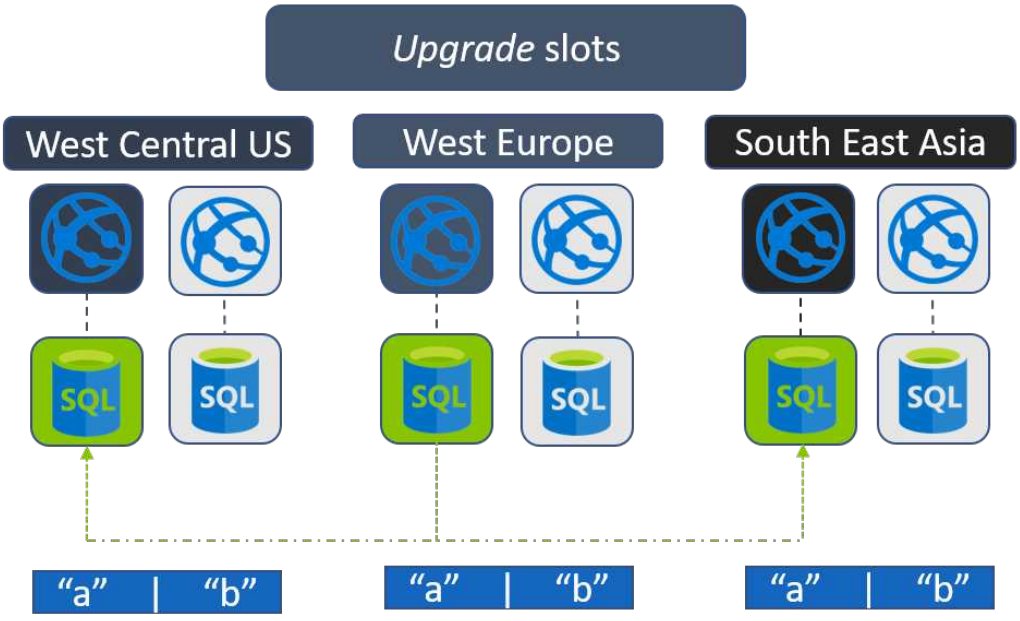
Browse your Traffic Manager DNS name *(should be something like https://<primary-name>.trafficmanager.net)* and you should be served by one of your App Services.

Release Process

At this point you should have `Online.Store` applicaton deployed at least in two geographic regions around the world. This chapter will describe a `release plan` that is the process you need to follow to release new versions of you software without affecting the end user's experience *(zero downtime)*. During the process you will be able to cancel or `roll back` your updates in case something went really bad. You will use many of the scripts you have seen in the previous chapters plus a few more. The release plan is described in the following steps:

1. Create the **upgrade** App Services slots *(if not already exist)* and configure them to connect to the new versions of SQL Servers
2. Deploy new code in the *upgrade* slots
3. Setup chaining SQL Geo-Replication from version `"a"` **Read-Write** database to version `"b"` databases *(or vice versa)*
4. Stop Geo-Replication between versions and switch primary/secondary roles
5. Run SQL upgrades *(schema and data updates)* in the **new** `Read-Write` version
6. Deploy WebJobs to `upgrade` slots
7. Swap `upgrade` and `production` slots

1. Create *upgrade* slots



Upgrade slots

Image description

The above image shows 3 `upgrade` slots *(grey globe boxes)*, one per each `App Service` connecting to the **b** versions of `SQL Servers`. These servers don't contain any database. Their databases will be initialized through `SQL Geo-Replication` starting from the Read-Write `"a"` primary database

The first step to start a new release is create if not already exists one staging slot named `upgrade` for each `App Service` you have provisioned around the globe. This can be done either from the portal or using the **add-child-slot.ps1** script. Each of these slots connects to a different SQL Server than the production one. If the **active** SQL Server version is *"a"* then they connect to *"b"* and vice versa. For example, if `Online.Store` application is deployed in `West Europe` and `West Central US`, currently the active SQL Servers are the following

(example for planetscalestore):

SQL Server	Region	Geo-Replication
planetscalestore-westeurope-app-sql-a	West Europe	Primary - Read-Write
planetscalestore-westcentralus-app-sql-a	West Central US	Secondary - Read-Only

This means that the `upgrade` slots will connect to the following new SQL Servers

- `planetscalestore-westeurope-app-sql-b`
- `planetscalestore-westcentralus-app-sql-b`

There isn't any dependency between those servers, they are just provisioned in case they haven't yet. The **add-child-slot.ps1** script does the following:

1. Creates if not already exists the `upgrade` slot to the `App Service`
2. Creates if not already exists the version of the SQL Server to be used for the next release
3. Configures the App Settings and connection strings for the `upgrade` slot

It accepts the following parameters:

1. PrimaryName
2. ResourceGroupLocation
3. SqlServerVersion
4. SqlServerLogin
5. SqlServerPassword
6. Database
7. DocumentDBPolicies
8. AzureIDClientSecret *(optional)*
9. UseIdentity *(optional)*
10. IdentitySqlServerLogin *(optional)*
11. IdentitySqlServerPassword *(optional)*

Notice that they are exactly the same parameters used by the `init-child-resources.ps1` with the only difference that the `CreateDatabase` is missing. The `SqlServerVersion` parameter defines the version of SQL Server that the slot will connect to. If this SQL Server doesn't exist in the child resource group it will be created. The script's code is identical to `init-child-resources.ps1` except the part where it creates the `upgrade` slot.

```
$appServicePlan = "$resourceGroupName-$appServicePlanPrefix";
$webappName = "$resourceGroupName";
$slotName = "upgrade";

$azureSlot = Get-AzureRmWebAppSlot -ResourceGroupName "$resourceGroupName" `
                     -Name "$webappName" -Slot "$slotName" `
                     -ErrorAction SilentlyContinue

if ($azureSlot)
{
    Write-Host "Azure Slot already exists.."
}
else
{
    # Upgrade slot doesn't exist..
    Write-Host "Creating Azure Slot to $webappName .."

    New-AzureRmWebAppSlot -ResourceGroupName "$resourceGroupName" `
                          -Name "$webappName" -AppServicePlan "$appServicePlan" `
                          -Slot "$slotName"

    Write-Host "$slotName Azure Slot created.."
}
```

The script checks if the `upgrade` slot exists for the `App Service` using the Get-AzureRmWebAppSlot and if it doesn't it creates it using the New-AzureRmWebAppSlot.

Running the script

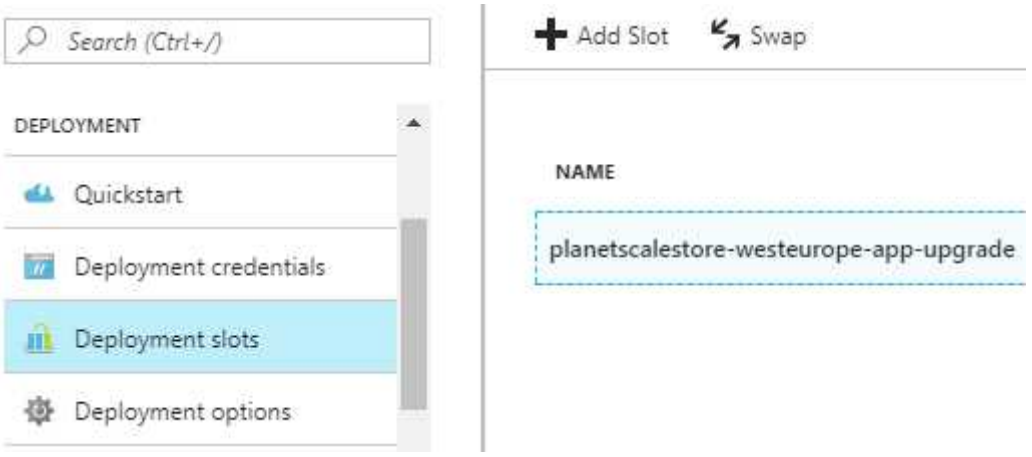
You can run the `add-child-slot` script as follow:

```
.\add-child-slot.ps1 `
        -PrimaryName "<primary name>" `
        -ResourceGroupLocation "<region>" `
        -SqlServerVersion "<version>" `
        -SqlServerLogin "<sql-server-login>" `
        -SqlServerPassword "<sql-server-password>" `
        -Database "<primary name>" `
        -DocumentDBPolicies "<policies>" `
        -UseIdentity $true `
        -IdentitySqlServerLogin "<identity-server-login>" `
        -IdentitySqlServerPassword "<identity-server-password>"
```

For the case of `planetscalestore` and assuming the new version of SQL Server is "b" the script would look like this.

```
.\add-child-slot.ps1 `
        -PrimaryName "planetscalestore" `
        -ResourceGroupLocation "westeurope" `
        -SqlServerVersion "b" `
        -SqlServerLogin "chsakell" `
        -SqlServerPassword "<password>" `
        -Database "planetscalestore" `
        -DocumentDBPolicies "West Europe" `
        -UseIdentity $true `
        -IdentitySqlServerLogin "chsakell" `
        -IdentitySqlServerPassword "<password>"
```

Confirm that both the slot and the new SQL Server have been created in the portal. In the main blade of your `App Service` select `Deployment slots`



Upgrade slot

Click on the `upgrade` slot and check its `App Settings` and connection strings. The only thing that actually changes is the `DefaultConnection` connection string that points to the new SQL Server.

Upgrade slot

You don't have to add a new entry in the `Traffic Profile Manager` for the `uprade` slots. When you swap them with the production, Azure makes the required changes internally

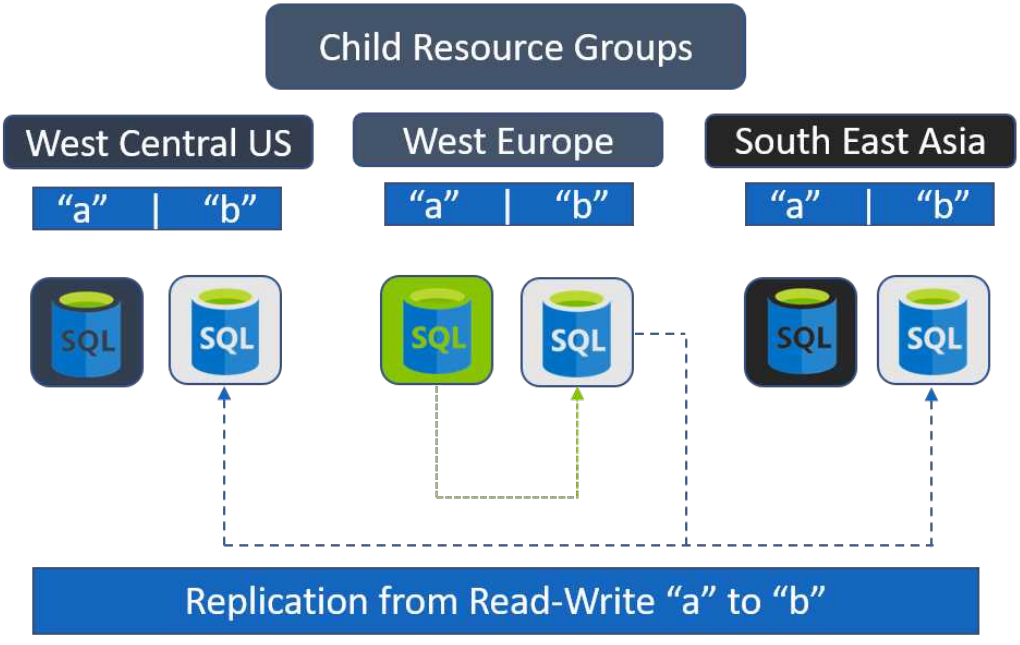
2. Deploy new code

Deploying your latest build to your `upgrade` slots. Use the **create-deployment-environment.ps1** script to create the new deployment environments and the **start-deployment.ps1** to deploy the code. For the `planetscalestore` example and the `upgrade` slot in `West Europe` the scripts would look like this.

```
.\create-deployment-environment.ps1 -token "<token>" `
                          -projectSlug "planet-scale-azure" `
                          -webappName "planetscalestore-westeurope-app" `
                          -resourceGroupName "planetscalestore-westeurope-app" `
                          -slot "upgrade"

.\start-deployment.ps1  `
                 -token "<token>" -accountName "chsakell" `
                 -projectSlug "planet-scale-azure" `
                 -webappName "planetscalestore-westeurope-app" `
                 -resourceGroupName "planetscalestore-westeurope-app" `
                 -deploymentEnvironment "planetscalestore-westeurope-app-upgrade" `
                 -slot "upgrade"
```

Repeat the scripts for all your `upgrade` slots over the globe.

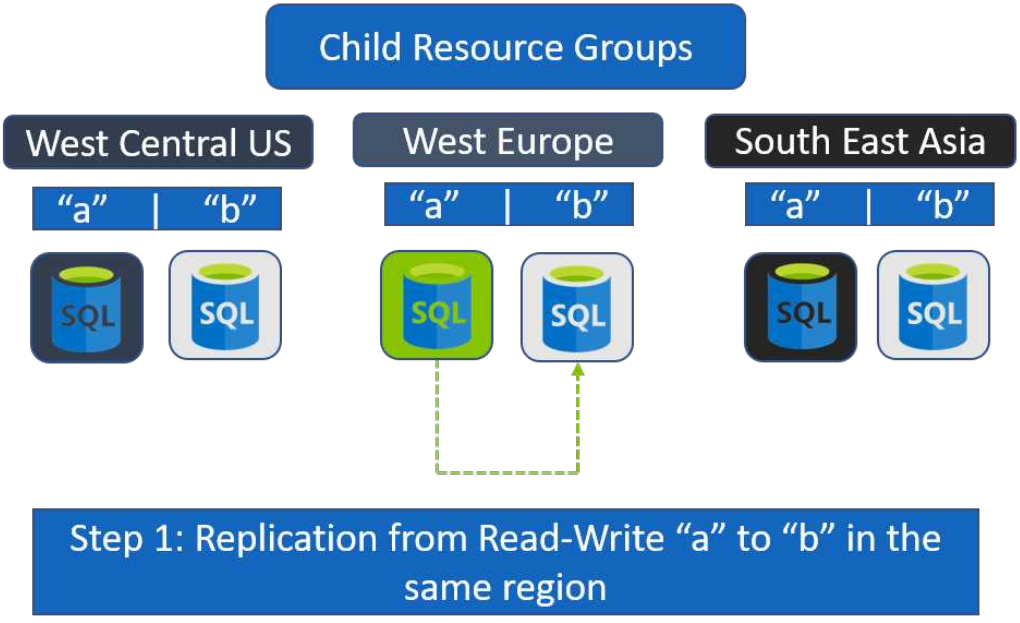
3. Setup chaining SQL Geo-Replication



Chaining SQL Geo-Replication

During this step you will create the following **chaining** SQL Geo-Replication:

1. Geo-Replication from main `Read-Write` database from SQL Server version `"a"` to the version `"b"` provisioned in the **same region**
2. Geo-Replication from the secondary database created in the previous step to the other SQL Servers with `"b"` version



Chaining SQL Geo-Replication - Step 1

For the first step you run the **init-geo-replication.ps1** script as follow.

```
.\init-geo-replication.ps1 `
        -Database "<primary name>" `
        -PrimaryResourceGroupName "<main-child-resource-group>" `
        -PrimaryServerName "<main-child-resource-group>-sql-a" `
        -SecondaryResourceGroupName "<main-child-resource-group>" `
        -SecondaryServerName "<main-child-resource-group>-sql-b"
```

For the case of `planetscalestore` where the primary database is provisioned in `West Europe` the script would look like this.

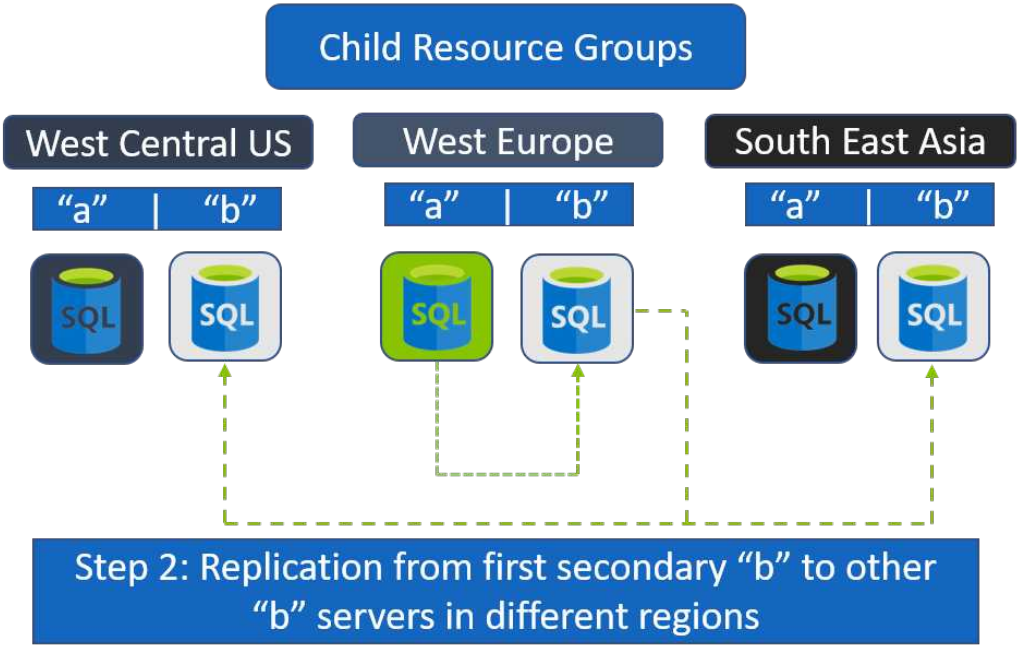
```
.\init-geo-replication.ps1 `
        -Database "planetscalestore" `
        -PrimaryResourceGroupName "planetscalestore-westeurope-app" `
        -PrimaryServerName "planetscalestore-westeurope-app-sql-a" `
        -SecondaryResourceGroupName "planetscalestore-westeurope-app" `
        -SecondaryServerName "planetscalestore-westeurope-app-sql-b"
```

The script will initiate SQL Geo-Replication from SQL Server `<main-child-resource-group>-sql-a` to SQL Server `<main-child-resource-group>-sql-b` for database `<primary name>`. A new database will be created in `<main-child-resource-group>-sql-b` SQL Server. At this moment that database is a secondary but later on, will be the primary.

NAME	RE...	SERVER
identitydb	None	planetscalestore-sql
planetscalestore	Secondary	planetscalestore-westcentralus-app-sql-a
planetscalestore	Primary	planetscalestore-westeurope-app-sql-a
planetscalestore	Secondary	planetscalestore-westeurope-app-sql-b

a to b SQL Geo-Replication

The second step is to create SQL Geo-Replication from the secondary database created before, to the other `"b"` SQL Servers used by the `upgrade` slots provisioned in child resource groups in other regions.



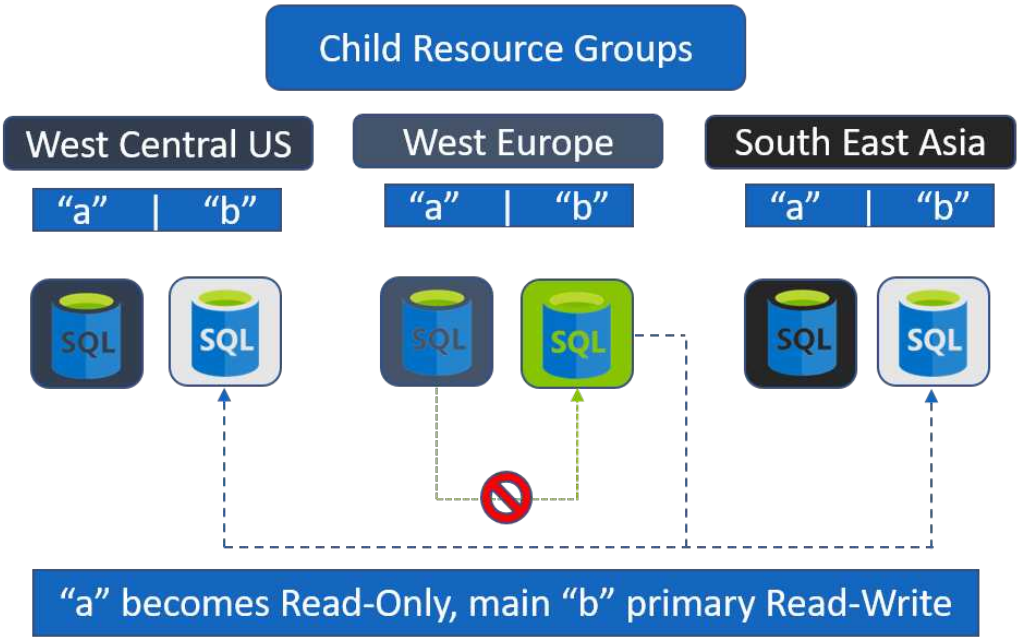
Chaining SQL Geo-Replication - Step 2

For the `planetscalestore` example there is one more *upgrade* slot in an `App Service` provisioned in a child resource group in `West Central US` so the script would look like this.

```
.\init-geo-replication.ps1 `
        -Database "planetscalestore" `
        -PrimaryResourceGroupName "planetscalestore-westeurope-app" `
        -PrimaryServerName "planetscalestore-westeurope-app-sql-b" `
        -SecondaryResourceGroupName "planetscalestore-westcentralus-app" `
        -SecondaryServerName "planetscalestore-westcentralus-app-sql-b"
```

At this point every update in the **Read-Write** database in version `"a"` is being replicated to all versions `"b"`.

4. Switch primary and secondary databases



Switch primary database from “a” to “b”

This is a very crusial step in the release process. The step can be splitted in 4 sub-tasks.

1. Make **Read-Only** the primary database in SQL Server version `"a"`
2. **Disconnect** the secondary database deployed in version `"b"` child resource group **in the same region**.
3. Make the disconnected secondary database **Read-Write**
4. Stop WebJobs deployed in production slots that connect to SQL Server version `"a"`

For the first sub-task, open a connection to your primary `Read-Write` database *(SQL Server Management Studio or Visual Studio)* and run the following query.

```
ALTER DATABASE [<database-here>] SET READ_ONLY
GO
```

Is `Online.Store` application operational at this point?

You may wonder what happens when you make the primary database `Read-Only`, will `Online.Store` application function properly? The answer is **YES**, end users won't understand any difference since orders are committed as `Service Bus` messages in their parent resource groups. The WebJobs in version `"a"` though may still listen the messages but they won't be able to commit the changes the database they connect to has become `Read-Only`. You can safely stop those WebJobs at the moment. The new orders will be saved by the new WebJobs

For the second step you will use a new script named **set-geo-replication.ps1** inside the `devops` folder. The script accepts `4` parameters:

1. **Database**: The database name
2. **PrimaryServerName**: The primary SQL Server that hosts the currently primary database
3. **SecondaryServerName**: The secondary SQL Server in the same region with the primary *(but version `"b"`)*
4. **Action**: Action to perform. Possible values *Stop* or *Start*

You can run the script as follow.

```
.\set-geo-replication.ps1 `
        -Database "planetscalestore" `
        -PrimaryServerName "<main-child-resource-group>-sql-a" `
        -SecondaryServerName "<main-child-resource-group>-sql-b" `
        -Action "Stop"
```

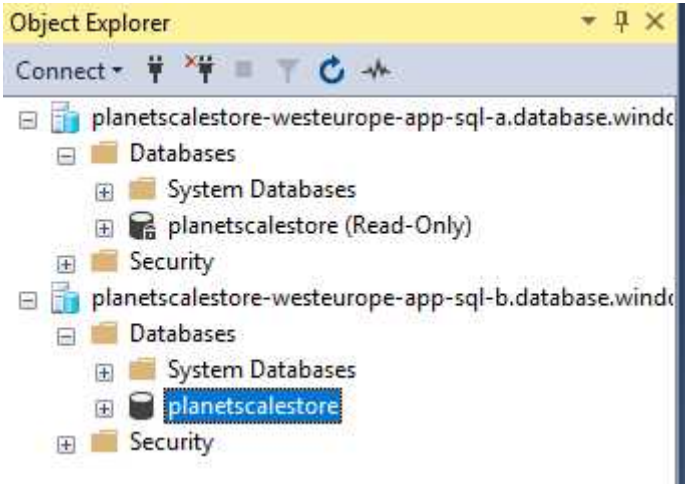
For the example of `planetscalestore` the command would look like this.

```
.\set-geo-replication.ps1 `
        -Database "planetscalestore" `
        -PrimaryServerName "planetscalestore-westeurope-app-sql-a" `
        -SecondaryServerName "planetscalestore-westeurope-app-sql-b" `
        -Action "Stop"
```

The script disconnects the secondary database using **planned termination** meaning it waits until all committed transactions on the source database at the time you run the cmdlet, have been replicated to the secondary database. When the termination completes, the secondary database automatically becomes a primary.

The last sub-task to complete this step is to make the new primary database **Read-Write**. Open a connection to the new primary database *(version b)* and run the following query.

```
ALTER DATABASE [<database>] SET READ_WRITE
GO
```



Switch primary database from "a" to "b"

5. Run SQL upgrades in new primary database

Now that the new primary database is **Read-Write** you can run any SQL schema or data updates. Any update on the primary database will be replicated in the other child resource groups of version `"b"`.

6. Deploy WebJobs

The next step is to deploy the WebJobs to your `upgrade` slots. You will use the **deploy-webjob.ps** script in the same way you used it for `production` slots with the difference that

this time you will use the "b" version SQL Server and the extra *slot* parameter. Following the example of the planetscalestore the script for deploying the WebJob in the upgrade slot in West Europe would look like this.

```
.\deploy-webjob.ps1 `
        -PrimaryDatabaseServer "planetscalestore-westeurope-app-sql-b" `
        -Database "planetscalestore" `
        -SqlServerLogin "chsakell" `
        -SqlServerPassword "<password>" `
        -WebappParentResourceGroup "planetscalestore-westeurope"`
        -WebappResourceGroup "planetscalestore-westeurope-app" `
        -WebjobAppLocation "C:\planet-scale-azure\Online.Store.WebJob" `
        -DeploymentDestinationFolder "C:\webjobs" `
        -slot "upgrade"
```

The script for deploying the WebJob for the upgrade slot in West Central US region..

```
.\deploy-webjob.ps1 `
        -PrimaryDatabaseServer "planetscalestore-westeurope-app-sql-b" `
        -Database "planetscalestore" `
        -SqlServerLogin "chsakell" `
        -SqlServerPassword "<password>" `
        -WebappParentResourceGroup "planetscalestore-westcentralus"`
        -WebappResourceGroup "planetscalestore-westcentralus-app" `
        -WebjobAppLocation "C:\planet-scale-azure\Online.Store.WebJob" `
        -DeploymentDestinationFolder "C:\webjobs" `
        -slot "upgrade"
```

Notice that both the scripts used the same new primary server in West Europe because that's the one who has Read-Write access.

7. Swap upgrade and production slots

The final step remained for completing the release process is **swap** the upgrade slots with the production ones using the Switch-AzureRmWebAppSlot cmdlet. For each upgrade slot and App Service you have provisioned run the following command.

```
Switch-AzureRmWebAppSlot -ResourceGroupName "<child-resource-group>" `
                         -Name "<child-resource-group>" `
                         -SourceSlotName "upgrade" `
                         -DestinationSlotName "production"
```

Warm the slot

Before making the swap it's good practice to *warm* it up by sending some requests to the slot. This way you will have zerotime downtime when completing the swap

For the case of `planetscalestore`, swapping the slots in `West Europe` and `West Central US` would look like this.

```
Switch-AzureRmWebAppSlot `
        -ResourceGroupName "planetscalestore-westeurope-app" `
        -Name "planetscalestore-westeurope-app" `
        -SourceSlotName "upgrade" `
        -DestinationSlotName "production"

Switch-AzureRmWebAppSlot `
        -ResourceGroupName "planetscalestore-westcentralus-app" `
        -Name "planetscalestore-westcentralus-app" `
        -SourceSlotName "upgrade" `
        -DestinationSlotName "production"
```

After swapping the slots users are served by the new version of your software *(new code, database changes in SQL Server b version plus new WebJobs)* while the `upgrade` slot keeps the older version used to be in production, in an `idle` state. In case something goes wrong and you wish to **roll back** to your previous version this is possible and easy to do. The only thing you need to do is swap again the slots.

```
Switch-AzureRmWebAppSlot `
        -ResourceGroupName "planetscalestore-westeurope-app" `
        -Name "planetscalestore-westeurope-app" `
        -SourceSlotName "upgrade" `
        -DestinationSlotName "production"

Switch-AzureRmWebAppSlot `
        -ResourceGroupName "planetscalestore-westcentralus-app" `
        -Name "planetscalestore-westcentralus-app" `
        -SourceSlotName "upgrade" `
        -DestinationSlotName "production"
```

Before swapping though you have to make the **main** database of the version `"a"` of you SQL Servers **Read-Write** because the previous production version of your code used to work with version `"a"` not `"b"`. Moreover, make sure to start the `WebJobs` which they used to save orders in the version `"a"`.

Assuming you had a succesfull release and user orders are comitted in the `"b"` version of the SQL Servers, go ahead and **remove** the SQL Servers of version `"a"`. You certainly don't want to get charged for resources you don't use. You can also delete the `upgrade` slots as well. The next time you start a release, you will set your `upgrade` slots to connect to a brand new SQL Server `"a"` version.

```
.\add-child-slot.ps1 `
        -PrimaryName "planetscalestore" `
        -ResourceGroupLocation "westeurope" `
        -SqlServerVersion "a" `
        -SqlServerLogin "chsakell" `
        -SqlServerPassword "<password>" `
        -Database "planetscalestore" `
        -DocumentDBPolicies "West Europe" `
        -UseIdentity $true `
        -IdentitySqlServerLogin "chsakell" `
        -IdentitySqlServerPassword "<password>"
```

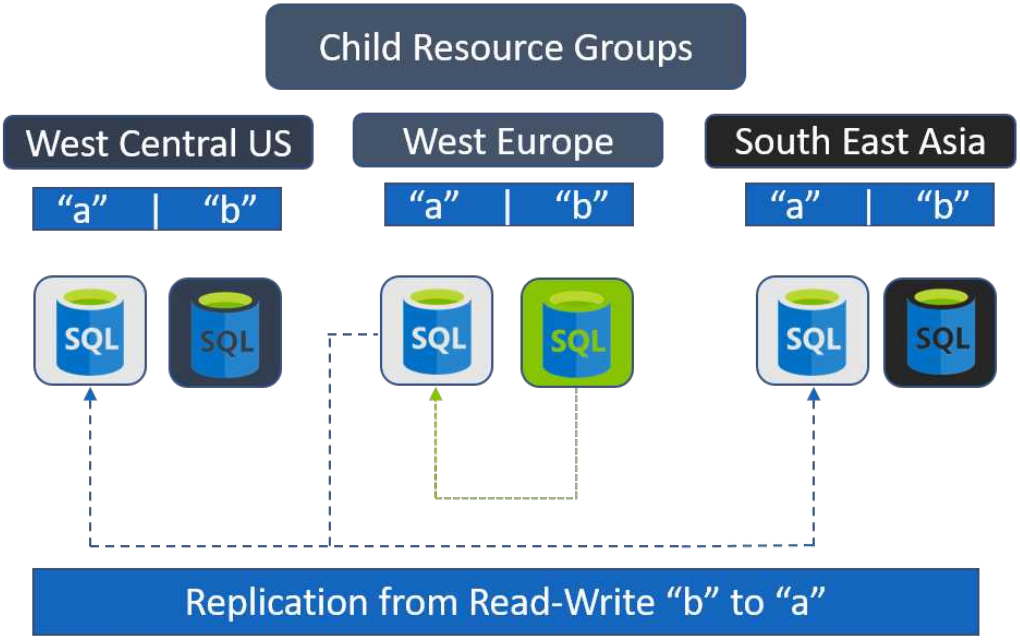
The script will create the SQL Server of version `"a"` and the `upgrade` slot if it doesn't exist. Next, you follow the exact same process you did but this time for SQL Server version `"a"`

```
.\add-child-slot.ps1 -PrimaryName "planetscalestore" `
                -ResourceGroupLocation "westeurope" `
                -SqlServerVersion "a" `
                -SqlServerLogin "chsakell" `
                -SqlServerPassword "<password>" `
                -Database "planetscalestore" `
                -DocumentDBPolicies "West Europe" `
                -UseIdentity $true `
                -IdentitySqlServerLogin "chsakell" `
                -IdentitySqlServerPassword "<password>"

.\init-geo-replication.ps1 `
        -Database "planetscalestore" `
        -PrimaryResourceGroupName "planetscalestore-westeurope-app" `
        -PrimaryServerName "planetscalestore-westeurope-app-sql-b" `
        -SecondaryResourceGroupName "planetscalestore-westeurope-app" `
        -SecondaryServerName "planetscalestore-westeurope-app-sql-a"

.\deploy-webjob.ps1 `
        -PrimaryDatabaseServer "planetscalestore-westeurope-app-sql-a" `
        -Database "planetscalestore" `
        -SqlServerLogin "chsakell" `
        -SqlServerPassword "<password>" `
        -WebappParentResourceGroup "planetscalestore-westeurope"`
        -WebappResourceGroup "planetscalestore-westeurope-app" `
        -WebjobAppLocation "C:\planet-scale-azure\Online.Store.WebJob" `
        -DeploymentDestinationFolder "C:\webjobs" `
        -slot "upgrade"
```

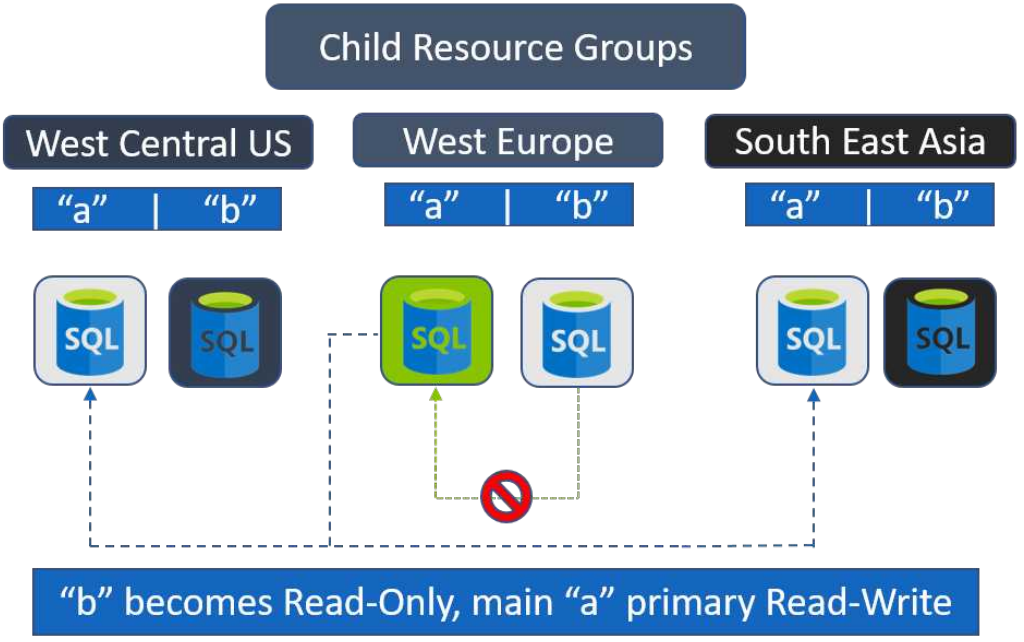
In case you re-create the `upgrade` slots, make sure to run the **create-deployment-environment** script to update the Web Deploy credentials in AppVeyor.

Init Geo-Replication

Before starting the replication the database on the secondary server must **NOT** exist



Chaining SQL Geo-Replication - b to a



a becomes primary again

Conclusion

Hopefully, you have managed to complete all of the exercises in the book. At this point you should be familiar with most of the `Azure Services` you have used during the book plus *configuring* them using `PowerShell` scripts. It is most likely that your application has different requirements than the `Online.Store` but this shouldn't stop you from building your own architecture and DevOps workflow.

What about other regions?

A lot of efforts have been made so that all the resources of `Online.Store` application can be grouped in `Primary`, `Parent` and `Child` *logical* resource groups. This architecture helped a lot to understand the dependencies between the services but also create more clean `PowerShell` scripts. On the other hand, it **restricted** the Azure Regions that you can use `Online.Store` application to those that all the resource types can actually provisioned on a specific region. But what if you want to use this architecture in regions that some of the resource types are not available? Well, this shouldn't be a problem at all, this is how cloud supposed to work from the beginning! Don't be afraid to provision a `Search Service` in a different region than the one you wanted but make sure the region you selected is close enough to the one you wished. The same applies for other services. Keep in mind that you will have to add extra parameters to the scripts so that can properly configure the app, for example you may have to add an extra parameter `SearchService:Region` to the **init-parent-resources** script

Let's finish this journey with 2 important tips.

Tips

Monitoring: Add Application Insights to your `App Services` to monitor your web sites. It helps to detect and diagnose exceptions or application performance issues

Disaster Recovery Plan: Replicate your primary resource group resources to a different region. While your application stays resilient if a datacenter outage happens in a datacenter other than the one where your primary resources are, this is not the case when the outage happens on the datacenter of your primary resources

In case you find any bugs in `Online.Store` application, don't hesitate to open an issue in the repository. Also, you can add comments for this book that may help to improve its content in the feedback page.