Cosmonaut Documentation

Nick Chapsas

Jul 12, 2020

User Documentation

Cosmonaut is a supercharged SDK with object mapping capabilities that enables .NET developers to work with CosmosDB. It eliminates the need for most of the data-access code that developers usually need to write while providing a fluent API for all of it's operations and ORM support.

Github page: https://github.com/Elfocrash/Cosmonaut

Nuget: https://www.nuget.org/packages/Cosmonaut

New to Cosmonaut? Check out the *Getting Started Guide* page first.

CHAPTER 1

Getting Started Guide

1.1 What is Cosmonaut?

Cosmonaut is a supercharged Azure CosmosD DB SDK for the SQL API with ORM support. It eliminates the need for most of the data-access code that developers usually need to write and it limits the unit of work scope to the object itself that the developer needs to work with.

1.2 Why use Cosmonaut?

The official Cosmos DB SDK has a ton of features and it can do a lot of things, but there is no clear path when it comes to doing those things. There is object mapping but the scope always stays the same.

Cosmonaut limits the scope from the Database account level to the `CosmosStore`. The `CosmosStore`'s context is a single collection or part of a collection when using the collection sharing feature. That way, we have an entry point with a single responsibility and authority to operate to only what it needs to know about.

1.3 How do I use Cosmonaut

The idea is pretty simple. You can have one CosmosStore per entity (POCO/dtos etc). This entity will be used to create a collection or use part of a one in CosmosDB and it will offer all the data access for this object.

Registering the CosmosStores in ServiceCollection for DI support

```
 var cosmosSettings = new CosmosStoreSettings("<<databaseName>>", "<<cosmosUri>>", "<
↪<authkey>>");

serviceCollection.AddCosmosStore<Book>(cosmosSettings);

//or just by using the Action extension
```

(continues on next page)

(continued from previous page)

```
serviceCollection.AddCosmosStore<Book>("<<databaseName>>", "<<cosmosUri>>", "<
↪<authkey>>", settings =>
{
    settings.ConnectionPolicy = connectionPolicy;
    settings.DefaultCollectionThroughput = 5000;
    settings.IndexingPolicy = new IndexingPolicy(new RangeIndex(DataType.Number, -1),
        new RangeIndex(DataType.String, -1));
});

//or just initialise the object

ICosmosStore<Book> bookStore = new CosmosStore<Book>(cosmosSettings)
```

To use the `AddCosmosStore` extension methods you need to install the `Cosmonaut.Extensions.Microsoft.DependencyInjection` package.

```
Install-Package Cosmonaut.Extensions.Microsoft.DependencyInjection
or
dotnet add package Cosmonaut.Extensions.Microsoft.DependencyInjection
```

1.4 Resources

- How to easily start using CosmosDB in your C# application in no time with Cosmonaut
- (Video) Getting started with .NET Core and CosmosDB using Cosmonaut
- (Video) How to save money in CosmosDB with Cosmonaut's Collection Sharing
- CosmosDB Fluent Pagination with Cosmonaut
- Implementing server side CosmosDB pagination in a Blazor Web App (Part 1: Page Number and Page Size)
- Implementing server side CosmosDB pagination in a Blazor Web App (Part 2: Next/Previous Page)
- The `samples` folder in this project
- Web app server-side pagination for CosmosDB

CHAPTER 2

The CosmosStore

2.1 What is it and why do I care?

The main data context you will be working with while using Cosmonaut is the CosmosStore. The CosmosStore requires you to provide the entity model that it will be working with.

For example if I only wanted to work with the class `Book` my CosmosStore initialisation would look like this:

```
ICosmosStore<Book> bookStore = new CosmosStore<Book>(cosmosSettings)
```

But what is the context of the CosmosStore? What will I get if I query for all the items in a CosmosStore?

The CosmosStore's boundaries can be one of two.

- One entity is stored in it's own collection (ie books)
- One entity is stored in a shared collection that other entities live as well (ie library)

The choice to go with one or the other is completely up to you and it comes down to partitioning strategy, cost and flexibility when it comes to scaleability.

2.2 A single mandatory property

In order for an entity to be able to be stored, manipulated and retrieved in a CosmosStore it is required to have a mandatory property. This is the `id` property.

It needs to be a `string` property with the name either being `Id` (with any capitalisation) or any other name but decorated with the `[JsonProperty("id")]` attribute. Even though not neccessary, when the property is named Id, you should also decorate it with the `[JsonProperty("id")]` attribute. This is neccessary if you want to do any querying based on the id (querying NOT reading). It will also help with unintented behavour when it comes to object mapping and LINQ to SQL transformations.

2.3 CosmosStore's lifetime

CosmosStores should be registered as *singletons* in your system. This will achieve optimal performance. If you are using a dependency injection framework make sure they are registered as singletons and if you don't, just make sure you don't dispose them and you keep reusing them.

2.4 CosmosStoreSettings

The CosmosStore can be initialised in multiple ways but the recommended one is by providing the `CosmosStoreSettings` object.

The `CosmosStoreSettings` object can be initialised requires 3 parameters in order to be created. The database name, the Cosmos DB endpoint Uri and the auth key.

```
var cosmosSettings = new CosmosStoreSettings("<<databaseName>>", "<<cosmosUri>>", "<
↪<authkey>>");
```

There are other optional settings you can provide such as:

- `ConnectionPolicy` - The connection policy for this CosmosStore.
- `ConsistencyLevel` - The level of consistency for this CosmosStore.
- `IndexingPolicy` - The indexing policy for this CosmosStore if it's collection in not yet created.
- `DefaultDatabaseThroughput` - The default database level throughput. No database throughput by default.
- `OnDatabaseThroughput` - The action to be taken when the collection that is about to be created is part of a database that has RU/s provisioned for it. `UseDatabaseThroughput` will ignore the `DefaultCollectionThroughput` and use the database's RUs. `DedicateCollectionThroughput` will provision dedicated RUs for the collection of top of the database throughput with the value of `DefaultCollectionThroughput`.
- `DefaultCollectionThroughput` - The default throughput for this CosmosStore if it's collection in not yet created.
- `JsonSerializerSettings` - The object to json serialization settings.
- `InfiniteRetries` - Whether you want infinite retries on throttled requests.
- `CollectionPrefix` - A prefix prepended on the collection name.
- `ProvisionInfrastructureIfMissing` - Whether the `CosmosStore` will automatically provision the infrastructure when the `CosmosStore` is instantiated. Default `true`.

  Note: In some scenarios, especially with .NET Framework apps, you might notice that inintialisation of the `CosmosStore` can cause a deadlock. This is due to it's call from the UI thread and a synchronisation context issue. To work around that, you can simply set the `ProvisionInfrastructureIfMissing` to `false` and then use the `CosmosStore`'s `EnsureInfrastructureProvisionedAsync` method awaited properly.

2.5 CosmosResponse and response handling

By default, Cosmos DB throws exceptions for any bad operation. This includes reading documents that don't exist, pre condition failures or trying to add a document that already exists.

This makes response handing really painful so Cosmonaut changes that.

Instead of throwing an excpetion Cosmonaut wraps the responses into it's own response called `CosmosResponse`.

This object contains the following properties:

- IsSuccess - Indicates whether the operation was successful or not
- CosmosOperationStatus - A Cosmonaut enum which indicates what the status of the response is
- ResourceResponse - The ResourceResponse that contains things like RU charge, Etag, headers and all the other info that the response would normally have
- Entity - The object you used for this operation
- Exception - The exception that caused this response to fail

It also has an implicit operation which, if present, will return the entity itself.

2.5.1 CosmosOperationStatus

The CosmosOperationStatus operation status can have one of 5 values.

- Success - The operation was successful
- RequestRateIsLarge - Your CosmosDB is under heavy load and it can't handle the request
- ResourceNotFound - The item you tried to update/upsert was not found
- PreconditionFailed - The Etag that you provided is different from the document one in the database indicating that it has been changed
- Conflict - You are trying to add an item that already exists

2.6 Notes

The CosmosStore also exposes the underlying CosmonautClient that it's using to perform operations so you can use that for any other operations you want to make against Cosmos DB. You need to know though that the CosmosStore's context is only limited for it's own methods. Once you use the CosmonautClient or the DocumentClient you are outside of the limitations of CosmosStore so be careful.

CHAPTER 3

The CosmonautClient

3.1 What is it and why do I care?

The CosmonautClient is a wrapper around the DocumentClient that comes from the CosmosDB SDK. It's main purpose is to abstract some of the things you really don't need to know about when it comes to using Cosmos DB. An example would be the UriFactory class.

Normally the DocumentClient call requires you to provide a Uri to resources in order to perform operations. You shouldn't care. All you need to care about is that this call needs a `databaseId` and a `collectionId`. This client wrapper does that.

It also wraps the calls to Cosmos and it profiles them in order to provide performance metrics. This will only happen when you have an active event source. You can learn more about this in the Logging section.

Something worth noting is that the CosmonautClient won't throw an exception for not found documents on methods that the response is `ResourceResponse` but instead it will return null in order to make response handling easier.

Any method that returns a `CosmosResponse` will still obay all the rules described on the "CosmosResponse and response handling" section of the "The CosmosStore" page.

CHAPTER 4

Preparing your classes for the CosmosStore

There are some things you need to get familiar with when it comes to using Cosmonaut. You see, Cosmonaut it doing a lot of things for you behind the scenes but it's performing even better if you do just a little bit of configuration.

4.1 Key attributes

4.1.1 [CosmosCollection]

The `CosmosCollection` attribute is an optional attribute that you can decorate your entity's class with. It has two purposes.

First it allows you to override the default Cosmonaut collection naming bahaviour, which is to name your collections as a lowercase plularised version of the class name.

Second it allows you to map your class to a pre existing collection with a different name.

> You can of course do a further override of the `CosmosStore` target name at the `CosmosStore` constructor level by providing the `overriddenCollectionName` parameter.

4.1.2 [SharedCosmosCollection]

In order to enable collection sharing you all you have to do is have your POCO implement the `ISharedCosmosEntity` interface and decorate the class with the `SharedCosmosCollectionAttribute`.

This attribute has one mandatory and one optional parameter namely the `SharedCollectionName` and the `EntityName`.

```
[SharedCosmosCollection("shared", "somebooks")]
```

The first parameter at the `SharedCosmosCollection` attribute with value `shared` represents the shared collection name that this object will use. This is the only mandatory parameter for this attribute. The second one with value `somebooks` represents the value that `CosmosEntityName` will automatically be populated with. If you don't set this value then a lowercase pluralised version of the class name will be used instead.

> You can of course do a further override of the `CosmosStore` target name at the `CosmosStore` constructor level by providing the `overriddenCollectionName` parameter.

4.1.3 [CosmosPartitionKey]

This is a parameterless attribute. It is used to decorate the property that represents the partition key definition of your collection.

```
public class Book
{
    [CosmosPartitionKey]
    public string Name { get; set; }

    [JsonProperty("id")]
    public string Id { get; set; }
}
```

In this case I have a partition key definition in my collection named `/Name`.

By decorating the `Name` property with the `CosmosPartitionKey` attribute I enable Cosmonaut to do a lot of behind the scenes operation optimisation where it will set the partition key value for you if present, speeding up the operation and making it more cost efficient.

4.1.4 [JsonProperty("id")]

This is not a Cosmonaut attribute and it is coming from JSON.NET which the underlying Cosmos DB SDK is using.

Even though not required I strongly recommend that you decorate your property which represents the id of the entity with the `[JsonProperty("id")]` attribute. This will prevent any unwanted behaviour with LINQ to SQL conversions and the id property not being mapped propertly.

CHAPTER 5

Reading and querying entities

Cosmonaut provides a set of easy to use methods to read and query entities for both CosmosStore and CosmonautClient.

5.1 CosmosStore

5.1.1 Reading an entity

In order to perform a direct read against a container you can to use the `FindAsync` method. This method has three overloads.

- `FindAsync(documentId, RequestOptions?, CancellationToken?)`
- `FindAsync(documentId, partitionKeyValue, CancellationToken?)`

The first one will use the document id to read the entity. It will *automatically* add the partition key value in the request options if `CosmosPartitionKey` attribute is decorating the id property of the object. In other words, if your id is your partition key and it's decorated with `CosmosPartitionKey` attribute Cosmonaut will automatically set it for you.

The second one is a shorthand of the first one in the sense that if you only need to specify the partition key value without any other options you can use that.

5.1.2 Querying for entities

Cosmonaut supports two different types of querying when it comes to querying for entities.

- Fluent Querying with LINQ
- Using CosmosDB flavoured SQL

You can enter both modes using the `Query` method of the CosmosStore. There are 2 `Query` methods.

- `Query(FeedOptions?)` - Entry point for LINQ querying

- `Query(string sql, object parameters = null, FeedOptions?, CancellationToken?)` - Entry point for SQL querying

They both return an IQueryable which behind the scenes is a IDocumentQuery. However, if you try to add a `Where` clause or an `OrderBy` on the SQL variation of this method you will get an error. All your querying logic HAS to be in the sql text and it cannot be extended with LINQ after that. You can still use the `.WithPagination` method which we will talk about in the "Pagination" section of the docs.

Querying with LINQ

Returning all the items in the CosmosStore.

```
var books = await booksStore.Query().ToListAsync();
```

This method will execute a cross partition query without any pagination. This is generally not the recommended approach as it might take a long time to get all the results back. You can instead provide a specific partition key value and get all the books inside a logical partition.

```
var books = await booksStore.Query(new FeedOptions{PartitionKey = new PartitionKey(
↪"Stephen King")}).ToListAsync();
```

This query will perform way better as we are specifying a specific partition that we want to query.

You can also add a filter and ordering. Let's see how we can return all the books written in 1998 ordered by name.

```
var books = await booksStore.Query().Where(x => x.PublishedDate == 1998).OrderBy(x =>␣
↪x.Name).ToListAsync();
```

Simple as that. However you have to keep in mind that this query is again a cross partition query that also does ordering. This will be an inefficient and long running query. Ideally you want to provide the partition key value every time you query for items.

Querying with SQL

> The CosmosStore context is always limited to this entity. This means that even if you do `select * from c` from the CosmosStore you won't actually select everything from the collection but everything from the collection for that specific object.

The equivelant of the first LINQ example can be written in SQL like this:

```
var books = await booksStore.Query("select * from c").ToListAsync();
var books = await booksStore.QueryMultipleAsync("select * from c");
```

Both methods also accept the `FeedOptions` object that offers options for the query execution and cancellation tokens.

The reason why there are two methods that seem to be doing the same thing is because the first one can also use the `WithPagination` extension after that in order to be converted to a SQL query that also uses pagination. More about pagination can be found on the "Pagination" section of the docs.

Cosmonaut also supports parameterised queries in the same way that Dapper does. You can add the `@` symbol in the filter part of your sql query and then provide an object that contains a property that matches the name of the `@` prefixed paramater in order to validate and replace.

Example:

```
var user = await cosmoStore.Query("select * from c where c.LastName = @name", new {
↪name = "Smith" }).ToListAsync();

var user = await cosmoStore.QueryMultipleAsync("select * from c where c.LastName =
↪@name", new { name = "Smith" });
```

QuerySingleAsync and QueryMultipleAsync

As we saw above QueryMultipleAsync is a method that returns an IEnumerable of objects based on the query provided. There is also a second overload that accepts a generic T type. The purpose of this method is to allow you to map your SQL result to a different object. The main reason behind this features is that Cosmos DB SQL allows you to select only a few properties to return, so instead of returning a huge document you return only the properties you need.

```
var listOfFullNames = await cosmoStore.QueryMultipleAsync<FullName>("select c.
↪FirstName, c.LastName from c");
```

Of course you can achieve the same with LINQ by doing the following.

```
var listOfFullNames = await cosmoStore.Query().Select(x => new FullName{ FirstName =
↪x.FirstName; LastName = x.LastName }).ToListAsync();
```

The QuerySingleAsync methods are similar to the Multiple ones with the only difference being that they return a single value. If there are more than 1 value that this query returns then you will get an exception so you'll need to add a `select top 1` if you only need one item to be returned. If nothing is muched it retuns null.

> `ToListAsync` is only one of the asynchronous extension methods that Cosmonaut provides. Check the "Extensions" section of this page for a complete list of extension methods that you can use to query and retrieve data.

5.2 CosmonautClient

The CosmonautClient also offers a few new methods in order to make querying and reading easier.

5.2.1 Querying for documents/entities

- `Query<T>(databaseId, collectionId, FeedOptions?)` - Fluent LINQ querying entry point similar to the CosmosStore one but without the context limitations.
- `Query<T>(databaseId, collectionId, string sql, object parameters = null, FeedOptions?)` - Fluent SQL querying entry point similar to the CosmosStore one but without the context limitations.
- `QueryDocumentsAsync<T>(string databaseId, string collectionId, Expression<Func<Document, bool>> predicate = null, FeedOptions?, CancellationToken?)` - Queries all the documents that match the predicate providies. If the predicate is null then it queries all the documents in the collection. Returns results mapped to generic type `T`.
- `QueryDocumentsAsync(string databaseId, string collectionId, Expression<Func<Document, bool>> predicate = null, FeedOptions?, CancellationToken?)` - Queries all the documents that match the predicate provides. If the predicate is null then it queries all the documents in the collection. Returns `Document` results.

- `QueryDocumentsAsync<T>(databaseId, collectionId, string sql, object parameters = null, FeedOptions?, CancellationToken?)` - Similar to QueryMultipleAsync

5.3 Result Extensions

Cosmonaut has a set of extension methods that can be used in both LINQ and SQL based IQueryables in order to asynchronously query the containers.

- `ToListAsync<T>` - Asynchronously queries Cosmos DB and returns a list of all the data matching the query.
- `ToPagedListAsync<T>` - Asynchronously queries Cosmos DB and returns a `CosmosPagedResults` response containing a list of all the data matching the query and also whether there are more pages after this query and a continuation token for the next page.
- `FirstAsync<T>` - Asynchronously queries Cosmos DB and returns the first item matching this query. Throws exception if no items are matched.
- `FirstOrDefaultAsync<T>` - Asynchronously queries Cosmos DB and retuns the first item matching this query. Returns null if no items are matched.
- `SingleAsync<T>` - Asynchronously queries Cosmos DB and returns a single item matching this query. Throws exception if no items or more than one item are matched.
- `SingleOrDefaultAsync<T>` - Asynchronously queries Cosmos DB and returns a single item matching this query. Throws exception if more than one item are matched and returns null if no items are matched.
- `CountAsync<T>` - Asynchronously queries Cosmos DB and retuns the count of items matching this query.
- `MaxAsync<T>` - Asynchronously queries Cosmos DB and retuns the maximum value of item matched.
- `MinAsync<T>` - Asynchronously queries Cosmos DB and retuns the minimum value of item matched.

CHAPTER 6

Working with Entities using Cosmonaut

6.1 Adding an entity in the CosmosStore

```
var newUser = new User
{
    Name = "Nick"
};
var oneAdded = await cosmoStore.AddAsync(newUser);

var multipleAdded = await cosmoStore.AddRangeAsync(manyManyUsers);
```

Using the Range operation allows you to also provide a `Func` that creates a `RequestOptions` for every individual execution that takes place in the operation

6.2 Updating and Upserting entities

When it comes to updating you have two options.

Update example

```
var response = await cosmoStore.UpdateAsync(entity);
```

Upsert example

```
var response = await cosmoStore.UpsertAsync(entity);
```

The main difference is of course in the functionality. Update will only update if the item you are updating exists in the database with this id. Upsert on the other hand will either add the item if there is no item with this id or update it if an item with this id exists.

There are also `Range` variation of both of these methods.

Using one of the Range operations allows you to also provide a `Func` that creates a `RequestOptions` for every individual execution that takes place in the operation. This might be something the Etag in order to ensure that you are updatng the latest version of the document.

Example of an `UpdateRangeAsync` execution that ensures that the latest version of the document is being updated:

```
var updated = await booksStore.UpdateRangeAsync(objectsToUpdate, x => new␣
↪RequestOptions { AccessCondition = new AccessCondition
{
    Type = AccessConditionType.IfMatch,
    Condition = x.Etag
}});
```

6.3 Removing entities

There are multiple ways to remove an entity in Cosmonaut.

The simplest one is to use any of the overloads of the `RemoveByIdAsync` methods.

```
var removedWithId = await cosmoStore.RemoveByIdAsync("documentId");
var removedWithIdAndPartitionKey = await cosmoStore.RemoveByIdAsync("documentId",
↪"partitionKeyValue");
```

There is also the `RemoveAsync` method which uses an entity object to do the removal. However this object needs to have the id property populated and if it's a partitioned, it should also have the partition key value populated.

```
var removedEntity = await cosmosStore.RemoveAsync(entity);
```

Last but not least you can use the `RemoveAsync` method that has a predicate in it's signature. This will match all the documents that satistfy the predicate and remove them. You have to keep in mind that this method is doing a cross partition query behind the scenes before it does a direct delete per document. It's not very efficient and it should be used only in rare cases.

```
var deleted = await cosmoStore.RemoveAsync(x => x.Name == "Nick");
```

You can specify the `FeedOptions` for the query that takes place, potentially providing a partition key value to limit the scope of the request.

You to also provide a `Func` that creates a `RequestOptions` for every individual execution that takes place in the operation.

CHAPTER 7

Collection sharing

7.1 What is collection sharing?

When development on Cosmonaut started there was no option to provision RUs on the database level. Later this feature came in and it has a 50k RUs minimum. It was later reduced to 10k and now it's on 400 RUs for the whole database. That's fine and all but having collection level throughput is still to me the best way to go. It limits the scalability to a single collection.

Collection sharing is the consept of having multiple different types of objects sharing the same collection while Cosmonaut is able to operate on them as if they were in completely different containers.

The benefit of such a feature is that you don't need a single collection per entity type. You can simply have them sharing. If you are also good with your partitioning strategy you will be able to have multiple shared collections with different partition key definitions that make sense and provide optimal read and write performance.

7.2 How can I use Collection sharing?

In order to enable collection sharing you all you have to do is have your POCO implement the `ISharedCosmosEntity` interface and decorate the class with the `SharedCosmosCollectionAttribute`.

An example of an object that is hosted in a shared collection looks like this:

```
[SharedCosmosCollection("shared", "somebooks")]
public class Book : ISharedCosmosEntity
{
    [JsonProperty("id")]
    public string Id { get; set; }

    [CosmosPartitionKey]
    public string Name { get; set; }

    public string CosmosEntityName { get; set; }
}
```

The first parameter at the `SharedCosmosCollection` attribute with value `shared` represents the shared collection name that this object will use. This is the only mandatory parameter for this attribute. The second one with value `somebooks` represents the value that `CosmosEntityName` will automatically be populated with. If you don't set this value then a lowercase pluralised version of the class name will be used instead.

> Note: You do NOT need to set the `CosmosEntityName` value yourself. Leave it as it is and Cosmonaut will do the rest for you.

Even though this is convinient I understand that you might need to have a more dynamic way of specifying the collection that this object should use. That's why the `CosmosStore` class has some extra constructors that allow you to specify the `overriddenCollectionName` property. This property will override any collection name specified at the attribute level and will use that one instead.

> Note: If you have specified a CollectionPrefix at the CosmosStoreSettings level it will still be added. You are only overriding the collection name that the attribute would normally set.

If you want your shared collection to be partitioned then make sure than the partition key definition is the same in all the objects that are hosted in this collection.

You can also use the `SharedCosmosCollection` constructor overload that uses the `UseEntityFullName` boolean. By using that constructor Cosmonaut will automatically assign the full namespace of the entity as the discriminator value.

CHAPTER 8

Pagination

Cosmonaut supports two types of pagination.

- Page number + Page size
- ContinuationToken + Page size

Both of there methods work by adding the `.WithPagination()` method after you used any of the `Query` methods.

```
var firstPage = await booksStore.Query().WithPagination(1, 10).OrderBy(x=>x.Name).
↪ToListAsync();
var secondPage = await booksStore.Query().WithPagination(2, 10).OrderBy(x => x.Name).
↪ToPagedListAsync();
var thirdPage = await booksStore.Query().WithPagination(secondPage.NextPageToken, 10).
↪OrderBy(x => x.Name).ToPagedListAsync();
var fourthPage = await thirdPage.GetNextPageAsync();
var fifthPage = await booksStore.Query().WithPagination(5, 10).OrderBy(x => x.Name).
↪ToListAsync();
```

`ToListAsync()` on a paged query will just return the results. `ToPagedListAsync()` on the other hand will return a `CosmosPagedResults` object. This object contains the results but also a boolean indicating whether there are more pages after the one you just got but also the continuation token you need to use to get the next page.

8.1 Pagination recommendations

Because page number + page size pagination goes though all the documents until it gets to the requested page, it's potentially slow and expensive. The recommended approach would be to use the page number + page size approach once for the first page and get the results using the `.ToPagedListAsync()` method. This method will return the next continuation token and it will also tell you if there are more pages for this query. Then use the continuation token alternative of `WithPagination` to continue from your last query.

Keep in mind that this approach means that you have to keep state on the client for the next query, but that's what you'd do if you where using previous/next buttons anyway.

CHAPTER 9

Dependency Injection

Cosmonaut also has a separate package that adds extensions on top of the .NET Standard Dependency injection framework.

Nuget package: Cosmonaut.Extensions.Microsoft.DependencyInjection

Installing this package will add a set of methods for `IServiceCollection` called `AddCosmosStore`.

```
var cosmosSettings = new CosmosStoreSettings("<<databaseName>>", "<<cosmosUri>>", "<
↪<authkey>>");

serviceCollection.AddCosmosStore<Book>(cosmosSettings);

// or override the collection name

serviceCollection.AddCosmosStore<Book>(cosmosSettings, "myCollection");

//or just by using the Action extension

serviceCollection.AddCosmosStore<Book>("<<databaseName>>", "<<cosmosUri>>", "<
↪<authkey>>", settings =>
{
    settings.ConnectionPolicy = connectionPolicy;
    settings.DefaultCollectionThroughput = 5000;
    settings.IndexingPolicy = new IndexingPolicy(new RangeIndex(DataType.Number, -1),
        new RangeIndex(DataType.String, -1));
});
```

CHAPTER 10

Logging

10.1 Event source

Cosmonaut uses the .NET Standard's `System.Diagnostics` to log it's actions as dependency events.

By default, this system is deactivated. In order to activated and actually do something with those events you need to create an `EventListener` which will activate the logging and give you the option do something with the logs.

10.2 Cosmonaut.ApplicationInsights

By using this package you are able to log the events as dependencies in Application Insights in detail. The logs are batched and send in intervals OR automatically sent when the batch buffer is filled to max.

Just initialise the AppInsightsTelemetryModule in your Startup or setup pipeline like this. Example:

```
AppInsightsTelemetryModule.Instance.Initialize(new TelemetryConfiguration(
↪"InstrumentationKey"));
```

If you already have initialised `TelemetryConfiguration` for your application then use `TelemetryConfiguration.Active` instead of `new TelemetryConfiguration` because if you don't there will be no association between the dependency calls and the parent request.

```
AppInsightsTelemetryModule.Instance.Initialize(new␣
↪TelemetryConfiguration(TelemetryConfiguration.Active));
```