

Speed up ASP.NET Core WEB API application. Part 2

Eduard Silantiev, 13 Oct 2018

<https://www.linkedin.com/in/eduard-silantiev-012922148/>

<https://github.com/EduardSilantiev/Speed-up-ASP.NET-Core-WEB-API-app-Part2>

Using various approaches to increase ASP.NET Core WEB API application's productivity

Introduction

In Part 1 we have created a RESTful WEB API service using Asynchronous design pattern.

In Part 2 we will review the following:

- Application productivity;
- Asynchronous design pattern;
- Data normalization vs SQL queries efficiency;
- NCHAR vs NVARCHAR data type;
- Using Full-Text Engine of MSSQL server;
- Stored procedures;
- Optimize stored procedures;
- Precompilation and reusing stored procedures execution plans;
- Use the Entity Framework Core with full-text search;
- Entity Framework Core performance;
- Full-text search on the Prices table;
- Full-text search on numeric values;
- Changing a computed column formula;
- Caching results of data processing;
- Redis cache;
- Installing Redis on Windows;
- Redis Desktop Manager;
- Redis NuGet package;
- Caching expiration control;
- Where to apply caching?
- Caching implementation;
- Prepare data in advance concept;
- Prepare data in advance implementation;
- Thinking of microservices architecture;
- HttpClient issues;
- Manage HttpClient with HttpClientFactory.

Application productivity

There are some steps that can be performed to increase application productivity:

- Asynchronous design pattern;
- Denormalizing data;
- Full-Text searching;
- Optimize Entity Framework Core;
- Caching data processing results;
- Prepare data in advance.

Asynchronous design pattern

Asynchronously working is the first step of increasing the productivity of our application.

Asynchronous design pattern has been implemented in Part 1. It needs some extra coding, and generally works a little slower than a synchronous one, because it demands certain background activity of the system to provide asynchrony. So, in small applications without long I/O operations, asynchronous working can even decrease applications performance.

But in a heavily loaded application asynchronous can increase its productivity and resilience by using resources more efficiently. Let us observe how requests are being processed in ASP.NET Core:

Each request is being processed in an individual thread that is taken from the thread pool. If working synchronously and a long I/O operation occurs the thread waits till the end of the operation and returns to the pool after the operation completes. But during this wait, the thread is blocked and cannot be used by another request. So, for a new request, if no available thread is found in the thread pool, a new thread will be created to handle the request. It takes time to create a new thread and, with each blocked thread, there is also some blocked memory assigned to the thread. In a heavily loaded application, mass thread creation and blocking the memory can lead to a lack of resources and hence to a significant decrease of application and whole system productivity. It can even lead to applications crashing.

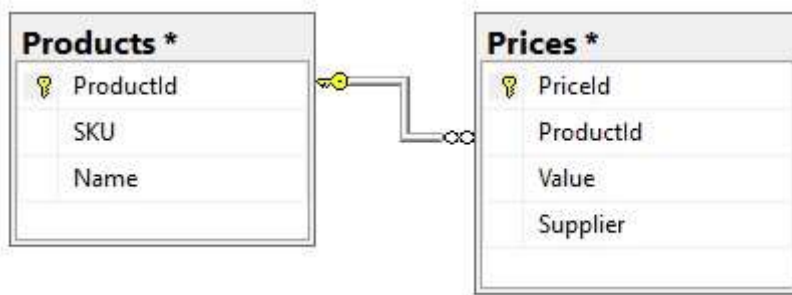
But if working asynchronously, just after I/O operation starts, a thread that handles the operation, returns to the thread pool and becomes available to handle another request.

So, an asynchronous design pattern increases application scalability by using resources more efficiently, thus making an application faster and more resilient.

Data normalization vs SQL queries efficiency

You may have noticed that SpeedUpCoreAPIExampleDB database structure almost entirely corresponds to expected output result. This means that taking data from the database and sending it to a user does not need any data transformation and thus provides the fastest result. We have achieved this by denormalizing the Prices table and using suppliers' names instead of suppliers' Ids.

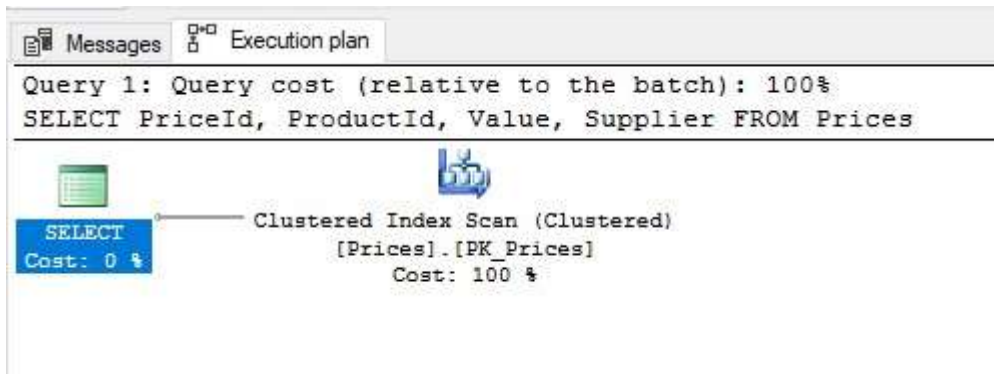
Our current database structure is:



All prices from the price table can be obtained by a request:

```
SELECT PriceId, ProductId, Value, Supplier FROM Prices
```

With an execution plan:



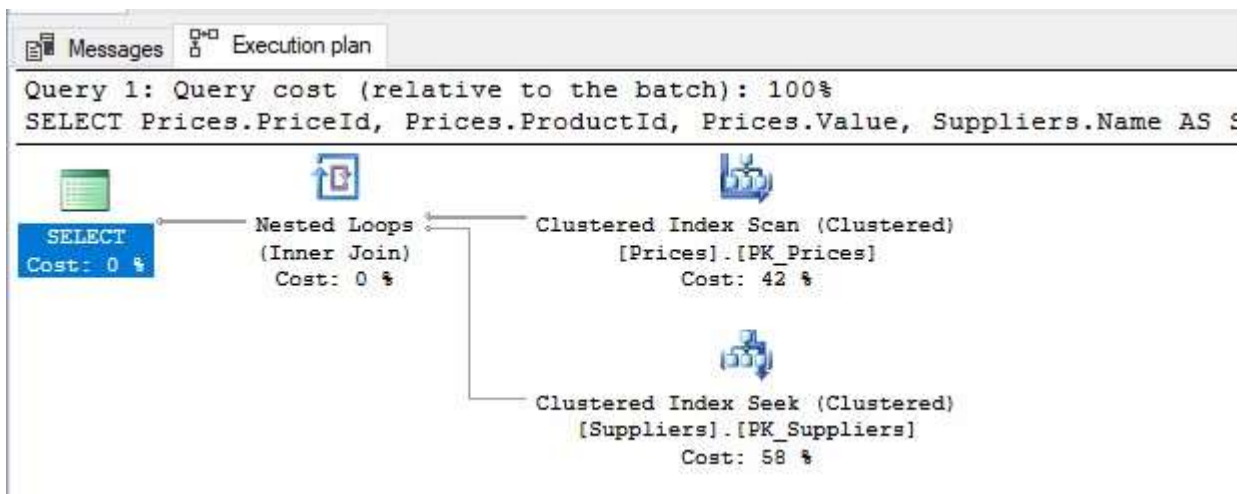
How our database structure could look like when fully normalized?



But in a fully normalized database the Prices and Suppliers tables should be joined in the SQL query that could have been like this:

```
SELECT Prices.PriceId, Prices.ProductId, Prices.Value, Suppliers.Name AS Supplier
FROM Prices INNER JOIN
      Suppliers ON Prices.SupplierId = Suppliers.SupplierId
```

With an execution plan:



The first query is evidently much faster because the Prices table has already been optimized for reading. But it is not so with a fully normalized data model that is optimized for storing complex objects but not for fast reading. So, with a fully normalized data we can have problems with SQL queries efficiency.

And notice, that the Prices table is optimized not only for reading, but also for filling with the data. For instance, a lot of pricelists nowadays are delivered with Excel files or a .csv files which can be easily obtained from Excel, any MS SQL Table or View and other sources. Usually such files have the following columns: Code; SKU; Product; Supplier; Price; where Supplier is a name, but not a code. If Code values in a file correspond to ProductId in the Products table, filling the Prices table with the data from such a file with millions of records can be performed in a few seconds by one line of T-SQL code:

```
EXEC('BULK INSERT Prices FROM ''' + @CsvSourceFileName + ''' WITH ( FORMATFILE = ''' + @FormatFileName + ''')');
```

Of course, denormalization has a price – doubling data and the necessity of solving issues with data consistency in the Prices and Suppliers tables. But is worth it if the goal is productivity.

Note! At the end of Part 1 we have tested the DELETE API. Your data can be different to our example. If so, please recreate the database from the script from Part 1

NCHAR vs NVARCHAR

In our database all the string fields have a NCHAR data type, which is obviously not the best solution. The fact is that the NCHAR is a fixed-length data type. It means, that SQL server reserves a place of fixed size (that we have declared for a field) for each field independently of the real length of fields content. For instance, the "Supplier" field in the Prices table has declared as:

```
[Supplier] NCHAR (50) NOT NULL
```

That is why, when we receive prices from the Prices table, the results look like:

```
[
{
```

```

    "PriceId": 7,
    "ProductId": 3,
    "Value": 160.00,
    "Supplier": "Bosch"
  },
  {
    "PriceId": 8,
    "ProductId": 3,
    "Value": 165.00,
    "Supplier": "LG"
  },
  {
    "PriceId": 9,
    "ProductId": 3,
    "Value": 170.00,
    "Supplier": "Garmin"
  }
]

```

To remove the trailing blank space in the Suppliers values we have to apply the Trim() method in the PricesService. The same for the SKU and Name results in the ProductsService. So, we have loss both in database size and application performance.

To resolve this issue, we can change NCHAR fields data type to NVARCHAR, which is variable-length string data type. For NVARCHAR fields SQL server allocates just the memory needed to hold the context of a field and does not add trailing spaces to a fields data.

We can change fields data types with the T-SQL script:

```

USE [SpeedUpCoreAPIExampleDB]
GO

ALTER TABLE [Products]
ALTER COLUMN SKU nvarchar(50) NOT NULL

ALTER TABLE [Products]
ALTER COLUMN [Name] nvarchar(150) NOT NULL

ALTER TABLE [Prices]
ALTER COLUMN Supplier nvarchar(50) NOT NULL

```

But the trailing spaces still remain because SQL server has not trimmed them in order to not lose the data. So, we should do the trimming intentionally:

```

USE [SpeedUpCoreAPIExampleDB]
GO

UPDATE Products SET SKU = RTRIM(SKU), Name = RTRIM(Name)
GO

UPDATE Prices SET Supplier = RTRIM(Supplier)
GO

```

Now we can remove all the .Trim() methods in the ProductsService and PricesService and the output results will be without the trailing spaces.

Use Full-Text Engine of MSSQL server

In case the Products table has a huge size, the speed of SQL query execution can be significantly increased by means of using the power of MSSQL server's Full-Text search Engine. FTS have only one limitation with the full-text search in MSSQL server – a text can only be searched by a prefix of a field. In other words, if apply full-text search for the SKU column and try to find records, which SKU contains "ab", only "abc", but not "aab" records can be found. If this search result is suitable for the application business logic, full-text search can be implemented.

So, a sku or its beginning part will be searched in the SKU column of Products table. For this, in our SpeedUpCoreAPIExampleDB database, we should create Full-Text catalog:

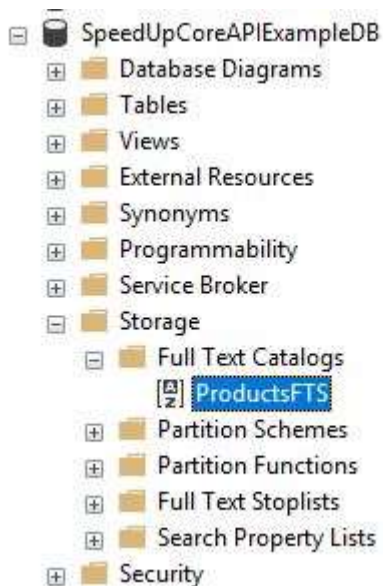
```
USE [SpeedUpCoreAPIExampleDB]
GO

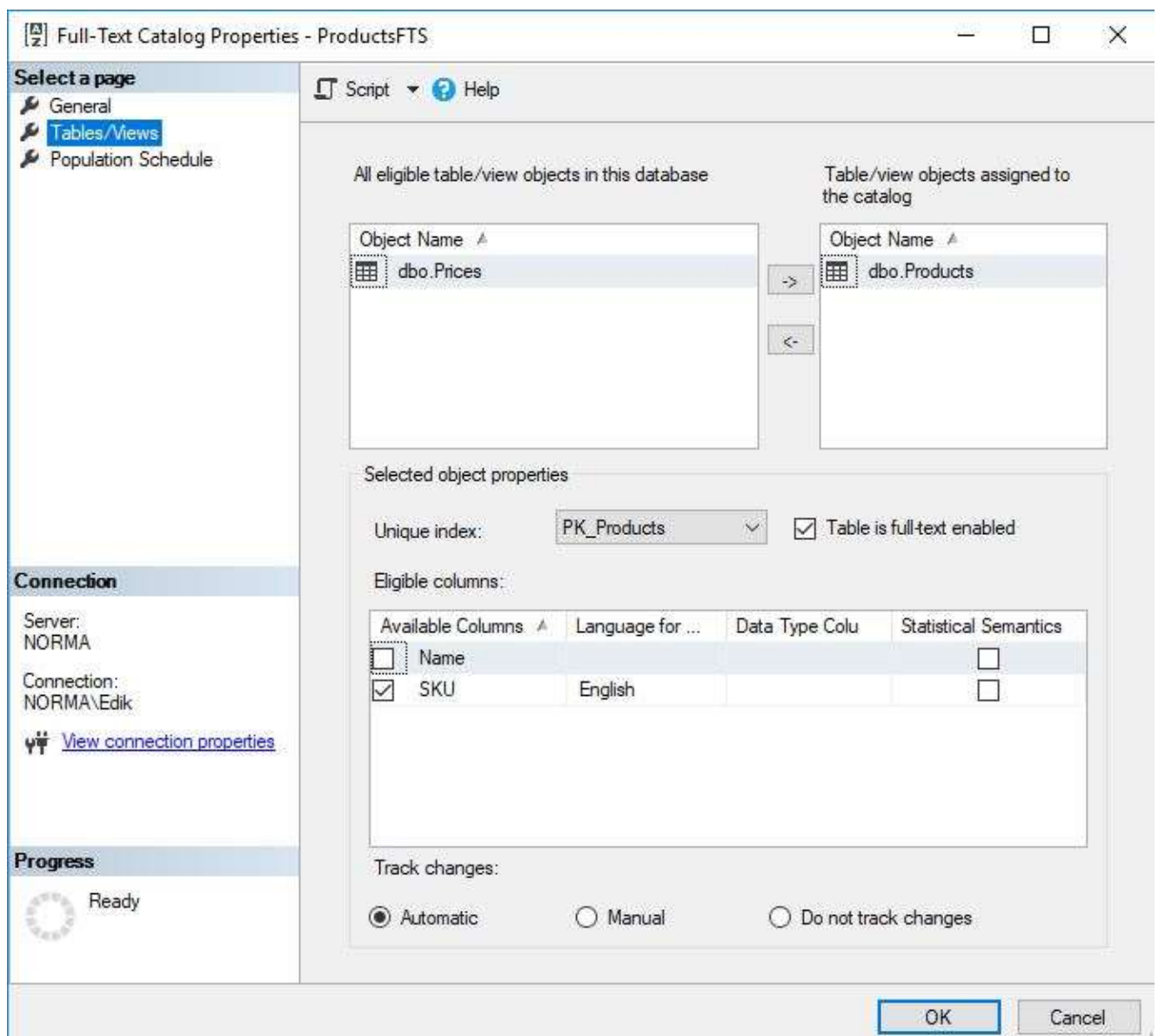
CREATE FULLTEXT CATALOG [ProductsFTS] WITH ACCENT_SENSITIVITY = ON
AS DEFAULT
GO
```

and then FULLTEXT INDEX in the ProductsFTS catalog

```
USE [SpeedUpCoreAPIExampleDB]
GO

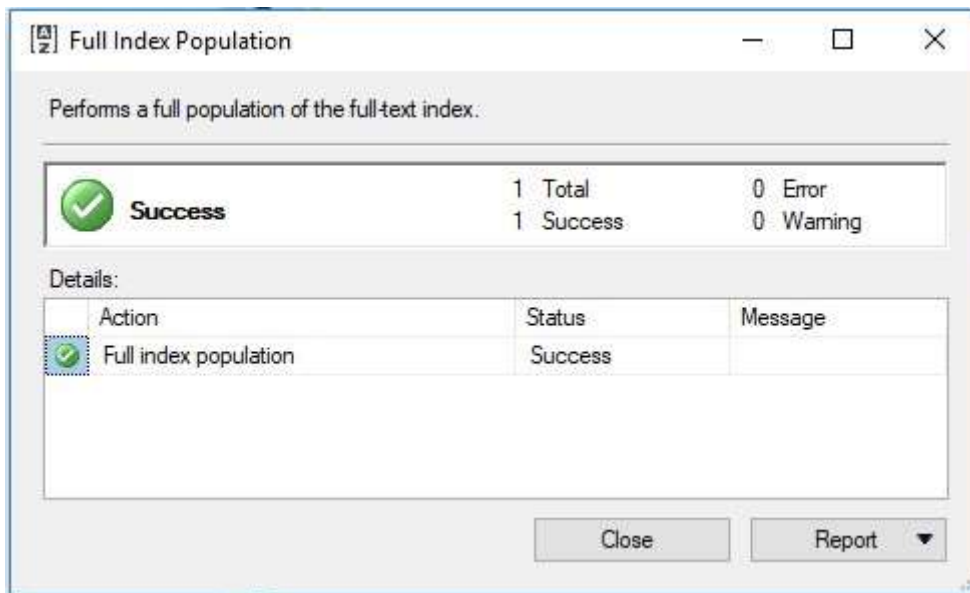
CREATE FULLTEXT INDEX ON [dbo].[Products]
(SKU LANGUAGE 1033)
KEY INDEX PK_Products
ON ProductsFTS
GO
```





The SKU column will be included in the full-text index. The Index will be populated automatically. But if you want to do this manually, just right click the Products table and select Full-Text index > Start Full population.

The result should be:



Let us create a stored procedure to examine how the full-text search works.

Stored procedure

```
USE [SpeedUpCoreAPIExampleDB]
GO

CREATE PROCEDURE [dbo].[GetProductsBySKU]
    @sku [varchar] (50)
AS
BEGIN
    SET NOCOUNT ON;

    Select @sku = '' + @sku + '*'

    -- Insert statements for procedure here
    SELECT ProductId, SKU, Name FROM [dbo].Products WHERE CONTAINS(SKU, @sku)
END
GO
```

Some explanation about the @sku format - to let the full-text search by prefix of the word, a search parameter should have the closing * wildcard: "aa*". So, the line `Select @sku = '' + @sku + '*'` just formats the @sku value.

Let us check how the procedure works:

```
USE [SpeedUpCoreAPIExampleDB]
GO

EXEC [dbo].[GetProductsBySKU] 'aa'
GO
```

The result will be:

| Results | | Messages | |
|---------|-----------|----------|----------|
| | ProductId | SKU | Name |
| 1 | 1 | aaa | Product1 |
| 2 | 2 | aab | Product2 |

Exactly as was expected.

Optimize the stored procedure

Do not forget to "SET NOCOUNT ON" to prevent unnecessary counting of processed records.

Note, that a query:

```
SELECT ProductId, SKU, [Name] FROM [dbo].Products WHERE CONTAINS(SKU, @sku)
```

is used, but not

```
SELECT * FROM Products WHERE CONTAINS(SKU, @sku)
```

Although the results of both queries will be equal, the first one works faster. Because if * wildcard is used instead of columns names, SQL server first searches for all the column names of the table and then replaces * wildcard with these names. If column names are stated explicitly, this extra job is omitted. And without stating table schema, [dbo] in our case, SQL server will search for a table in all the schemas. But if the schema is stated explicitly, SQL server searches for a table faster within this schema only.

Precompilation and reusing stored procedures execution plans

An important benefit of using a stored procedure is that before being executed for the first-time a procedure is compiled and its execution plan is created and put into a cache. Then, when the procedure is executed next time the compilation act is omitted and a ready execution plan is taken from the cache. All these make the request process much faster.

Let us make sure that SQL server reuses a procedure execution plan and a precompiled code. For this, first clear SQL server memory from all cached execution plans - in the Microsoft SQL Server Management Studio create new Query:

```
USE [SpeedUpCoreAPIExampleDB]
GO

--clear cache
DBCC FREEPROCCACHE
```

And check a cache state with a new query:

```

SELECT cplan.usecounts, cplan.objtype, qtext.text, qplan.query_plan
FROM sys.dm_exec_cached_plans AS cplan
CROSS APPLY sys.dm_exec_sql_text(plan_handle) AS qtext
CROSS APPLY sys.dm_exec_query_plan(plan_handle) AS qplan
ORDER BY cplan.usecounts DESC

```

The result will be:

| | usecounts | objtype | text | query_plan |
|---|-----------|---------|--|---|
| 1 | 1 | Adhoc | SELECT cplan.usecounts, cplan.objtype, qtext.te... | <ShowPlanXML xmlns="http://schemas.microsoft.com... |

Execute the stored procedure once again

```
EXEC [dbo].[GetProductsBySKU] 'aa'
```

and then inspect the cache:

| | usecounts | objtype | text | query_plan |
|---|-----------|---------|--|---|
| 1 | 1 | Proc | CREATE PROCEDURE [dbo].[GetProductsBySKU] @s... | <ShowPlanXML xmlns="http://schemas.microsoft.com... |
| 2 | 1 | Adhoc | SELECT cplan.usecounts, cplan.objtype, qtext.text, qplan.... | <ShowPlanXML xmlns="http://schemas.microsoft.com... |

We can see that a procedure execution plan is cached. Execute the procedure and check the information about currently cached plans once more:

| | usecounts | objtype | text | query_plan |
|---|-----------|---------|--|---|
| 1 | 2 | Proc | CREATE PROCEDURE [dbo].[GetProductsBySKU] @s... | <ShowPlanXML xmlns="http://schemas.microsoft.com... |
| 2 | 2 | Adhoc | SELECT cplan.usecounts, cplan.objtype, qtext.text, qplan.... | <ShowPlanXML xmlns="http://schemas.microsoft.com... |

In the "usecounts" field we can see how many times the plan has been reused. You can see in the "usecounts" field that the plan has been reused twice, proving that the execution plan caching really works for our procedure. In our example we have two times, this proves that the execution plan caching really works for our procedure.

Use the Entity Framework Core with full-text search

The last issue with the full-text search is how to use it with the Entity Framework Core. EFC generates queries to database by itself and does not take into account full-text indexes. There are some approaches how to fix it. The simplest way is to call our stored procedure GetProductsBySKU which has already implemented the full-text search.

To execute our stored procedure, we will use a FromSql method. This method is used in Entity Framework Core to execute the stored procedures and raw SQL queries that return sets of the data.

In the ProductsRepository.cs change the code of the FindProductsAsync method to:

```
public async Task<IEnumerable<Product>> FindProductsAsync(string sku)
{
    return await _context.Products.FromSql("[dbo].GetProductsBySKU @sku = {0}",
sku).ToListAsync();
}
```

Note, that to speed up the start of the procedure, we are using its fully qualified name [dbo].GetProductsBySKU, which includes the [dbo] schema.

A problem with using a stored procedure is that its code is out of the source control. To resolve this issue, you could call a raw SQL query with the same script rather than a stored procedure.

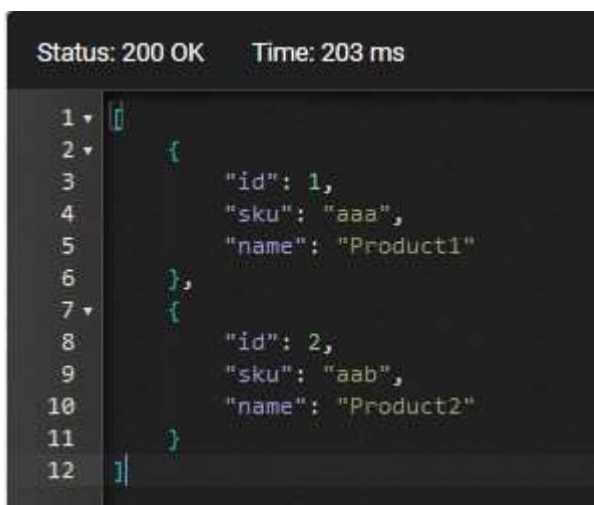
Note! Use only parameterized raw SQL queries to take advantage of execution plans reuse and to prevent SQL injection attacks.

But stored procedures are still faster because when calling a procedure, we pass only its name to SQL Server rather a full script text in case of calling a raw SQL query.

Let us check how the stored procedure and FTS work in our application. Start the application and test </api/products/find/>

<http://localhost:49858/api/products/find/aa>

The result will be the same as without full-text search:



Entity Framework Core performance

Since our stored procedure returns a list of expected entity type Product, EFC automatically preforms tracking to analyze which records were changed to update only those records. But we are not going to

change any data when obtaining a list of Products. So, it is reasonable to switch off tracking by using the `AsNoTracking()` method, which disables extra activity of EF and significantly increases its productivity.

The final version of the `FindProductsAsync` method without tracking is:

```
public async Task<IEnumerable<Product>> FindProductsAsync(string sku)
{
    return await _context.Products.AsNoTracking().FromSql("[dbo.GetProductsBySKU @sku = {0}]",
sku).ToListAsync();
}
```

We can also apply `AsNoTracking` in the `GetAllProductsAsync` method:

```
public async Task<IEnumerable<Product>> GetAllProductsAsync()
{
    return await _context.Products.AsNoTracking().ToListAsync();
}
```

And in the `GetProductAsync` method:

```
public async Task<Product> GetProductAsync(int productId)
{
    return await _context.Products.AsNoTracking().Where(p => p.ProductId ==
productId).FirstOrDefaultAsync();
}
```

Note, that with `AsNoTracking()` method EFC does not perform tracking of changed entities and you will not be able to save changes in an entity, found by the `GetProductAsync` method, if any, without attaching to the `_context`. But EFC still performs identity resolution, so we can easily delete a Product, found by the `GetProductAsync` method. That is why, our `DeleteProductAsync` method will work fine with the new version of `GetProductAsync` method.

Full-text search on the Prices table

We could significantly increase SQL query performance when fetching prices, if the `ProductId` was of `NVARCHAR` data type, because we could be able to apply Full-text search on the `ProductId` column. But its type is `INTEGER` because it is a foreign key to `ProductId` primary key of the `Products` table, which is integer with an auto increment identity.

One possible solution for this issue is to create a calculated column in the `Prices` table which will consist of `NVARCHAR` representation of the `ProductId` field and add this column to a full-text index.

Let us create a new calculated column named `xProductId`:

```
USE [SpeedUpCoreAPIExampleDB]
GO

ALTER TABLE [Prices]
ADD xProductId AS convert(nvarchar(10), ProductId) PERSISTED NOT NULL
```

We have marked the xProductId column as PERSISTED so that its values are physically stored in the table. If not persisted, xProductId column values will be recalculated every time they are accessed. These recalculations can also affect the performance of the SQL server.

Values in xProductId fields will be ProductId as a string:

The screenshot shows the SQL Server Enterprise Manager interface. At the top, the 'dbo.Prices' table is selected. Below it, a table structure view shows the following columns:

| Column Name | Data Type | Allow Nulls |
|-------------|----------------|--------------------------|
| PriceId | int | <input type="checkbox"/> |
| ProductId | int | <input type="checkbox"/> |
| Value | decimal(18, 2) | <input type="checkbox"/> |
| Supplier | nvarchar(50) | <input type="checkbox"/> |
| xProductId | | <input type="checkbox"/> |

The 'xProductId' column is highlighted. Below the table structure, the 'Column Properties' window is open for the 'xProductId' column. The 'Computed Column Specification' tab is selected, showing the following properties:

| Property | Value |
|--------------------------|---------------------------------------|
| (Name) | xProductId |
| Allow Nulls | No |
| Data Type | |
| Default Value or Binding | |
| Length | |
| Collation | <database default> |
| (Formula) | (CONVERT([nvarchar](10),[ProductId])) |
| Is Persisted | Yes |

New content of the table

| | PriceId | ProductId | Value | Supplier | xProductId |
|---|---------|-----------|--------|----------|------------|
| 1 | 1 | 1 | 100.00 | Bosch | 1 |
| 2 | 2 | 1 | 125.00 | LG | 1 |
| 3 | 3 | 1 | 130.00 | Gamin | 1 |
| 4 | 4 | 2 | 140.00 | Bosch | 2 |
| 5 | 5 | 2 | 145.00 | LG | 2 |
| 6 | 6 | 2 | 150.00 | Gamin | 2 |
| 7 | 7 | 3 | 160.00 | Bosch | 3 |
| 8 | 8 | 3 | 165.00 | LG | 3 |
| 9 | 9 | 3 | 170.00 | Gamin | 3 |

Then create new PricesFTS Full-text catalog with a FULLTEXT INDEX on the xProductId field:

```
USE [SpeedUpCoreAPIExampleDB]
GO

CREATE FULLTEXT CATALOG [PricesFTS] WITH ACCENT_SENSITIVITY = ON
AS DEFAULT
GO

CREATE FULLTEXT INDEX ON [dbo].[Prices]
(xProductId LANGUAGE 1033)
KEY INDEX PK_Prices
ON PricesFTS
GO
```

And finally, create a stored procedure to test the results:

```
USE [SpeedUpCoreAPIExampleDB]
GO

CREATE PROCEDURE [dbo].[GetPricesByProductId]
    @productId [int]
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @xProductId [NVARCHAR] (10)
    Select @xProductId = '' + CONVERT([nvarchar](10),@productId) + ''

    -- Insert statements for procedure here
    SELECT PriceId, ProductId, [Value], Supplier FROM [dbo].Prices WHERE
    CONTAINS(xProductId, @xProductId)
END
GO
```

In the stored procedure we have declared the @xProductId variable, converted @productId to NVARCHAR and performed full-text search.

Execute the GetPricesByProductId procedure:

```

USE [SpeedUpCoreAPIExampleDB]
GO

DECLARE @return_value int

EXEC @return_value = [dbo].[GetPricesByProductId]
    @productId = 1

SELECT 'Return Value' = @return_value

GO

```

But nothing has been found:



| PriceId | ProductId | Value | Supplier |
|---------|-----------|-------|----------|
|---------|-----------|-------|----------|

Full-text search on numeric values

The problem with full-text search on a string column that holds numeric values occurs in Microsoft SQL Server, starting from SQL Server 2012 due to its new version of the word breakers. Let us examine, how the full-text search engine parses xProductId value ("1", "2",...). Execute:

```

SELECT display_term FROM sys.dm_fts_parser (' "1" ', 1033, 0, 0)

```



| | display_term |
|---|--------------|
| 1 | 1 |
| 2 | nn1 |

You can see, that the parser has recognized value "1" both as a string in line 1 and as a number in line 2. This ambiguity did not allow the xProductId column values to be included in the full-text index. One possible way to resolve this issue is to ["Revert the Word Breakers Used by Search to the Previous Version"](#). But we have applied another approach – to start each value in the xProductId column with a char ("x", for example), to force the full-text parser to recognize values as strings. Let us make sure of this:

```

SELECT display_term FROM sys.dm_fts_parser (' "x1" ', 1033, 0, 0)

```



| | display_term |
|---|--------------|
| 1 | x1 |

There is no more ambiguity in the results.

Changing a computed column formula

The only possibility to alter a computed column is to remove the column and then to recreate it with other conditions.

As the ProductId column is enabled for Full-Text Search, we will not be able to remove the column, so we should remove a full-text index first:

```
USE [SpeedUpCoreAPIExampleDB]
GO

DROP FULLTEXT INDEX ON [Prices]
GO
```

Then remove the column:

```
USE [SpeedUpCoreAPIExampleDB]
GO

ALTER TABLE [Prices]
DROP COLUMN xProductId
GO
```

Then recreate the column with a new formula:

```
USE [SpeedUpCoreAPIExampleDB]
GO

ALTER TABLE [Prices]
ADD xProductId AS 'x' + convert(nvarchar(10), ProductId) PERSISTED NOT NULL
GO
```

Check the results:

```
USE [SpeedUpCoreAPIExampleDB]
GO

SELECT * FROM [Prices]
GO
```


| | PriceId | ProductId | Value | Supplier | xProductId |
|---|---------|-----------|--------|----------|------------|
| 1 | 1 | 1 | 100.00 | Bosch | x1 |
| 2 | 2 | 1 | 125.00 | LG | x1 |
| 3 | 3 | 1 | 130.00 | Garmin | x1 |
| 4 | 4 | 2 | 140.00 | Bosch | x2 |
| 5 | 5 | 2 | 145.00 | LG | x2 |
| 6 | 6 | 2 | 150.00 | Garmin | x2 |
| 7 | 7 | 3 | 160.00 | Bosch | x3 |
| 8 | 8 | 3 | 165.00 | LG | x3 |
| 9 | 9 | 3 | 170.00 | Garmin | x3 |

Recreate a full-text index:

```
USE [SpeedUpCoreAPIExampleDB]
GO

CREATE FULLTEXT INDEX ON [dbo].[Prices]
(xProductId LANGUAGE 1033)
KEY INDEX PK_Prices
ON PricesFTS
GO
```

Change our GetPricesByProductId stored procedure to add 'x' to a search pattern:

```
USE [SpeedUpCoreAPIExampleDB]
GO

ALTER PROCEDURE [dbo].[GetPricesByProductId]
    @productId [int]
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @xProductId [NVARCHAR] (10)
    Select @xProductId = "'x' + CONVERT([nvarchar](10),@productId) + '"

    -- Insert statements for procedure here
    SELECT PriceId, ProductId, [Value], Supplier FROM [dbo].Prices WHERE
CONTAINS(xProductId, @xProductId)
END
```

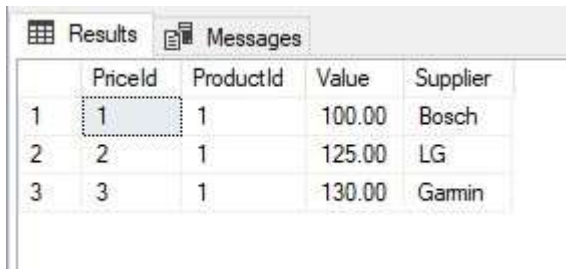
And finally, check the procedure working results:

```
USE [SpeedUpCoreAPIExampleDB]
GO

DECLARE @return_value int

EXEC    @return_value = [dbo].[GetPricesByProductId]
        @productId = 1
```

```
SELECT 'Return Value' = @return_value  
  
GO
```



| | PriceId | ProductId | Value | Supplier |
|---|---------|-----------|--------|----------|
| 1 | 1 | 1 | 100.00 | Bosch |
| 2 | 2 | 1 | 125.00 | LG |
| 3 | 3 | 1 | 130.00 | Garmin |

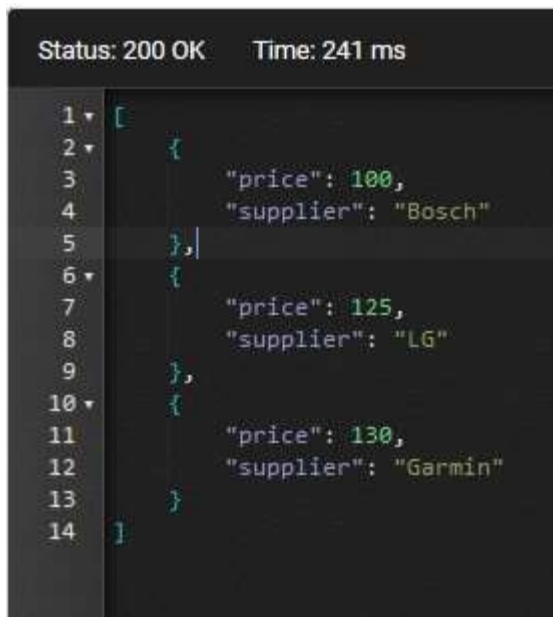
It works fine. Now let us change the GetPricesAsync method in the PricesRepository. Change the line:

```
return await _context.Prices.Where(p => p.ProductId == productId).ToListAsync();
```

to:

```
return await _context.Prices.AsNoTracking().FromSql("[dbo].GetPricesByProductId @productId =  
{0}", productId).ToListAsync();
```

Start the application and check the <http://localhost:49858/api/prices/1> results. The result will be the same as without full-text search:



Status: 200 OK Time: 241 ms

```
1 [
2   {
3     "price": 100,
4     "supplier": "Bosch"
5   },
6   {
7     "price": 125,
8     "supplier": "LG"
9   },
10  {
11    "price": 130,
12    "supplier": "Garmin"
13  }
14 ]
```

Caching results of data processing.

Look at the picture above again. In our case, the results of the <http://localhost:49858/api/prices/1> request can be cached for some time. On the next attempt to get prices of the Product1, the ready pricelist will be taken from a cache and sent to a user. If there is still no result for Id=1 in the cache,

prices will be taken from the database and put into the cache then. This approach will reduce the number of relatively slow database accesses in favor of fast retrieval of data from the cache in memory.

Redis cache

For caching, the Redis cache service will be used. The advantages of Redis cache are:

- Redis cache is in-memory storage of data, so it has a much higher performance than databases that store data on a disk;
- Redis cache implements IDistributedCache Interface. This means that we can easily change a cache provider to another IDistributedCache one, for example MS SQL Server, without the necessity to change cache management logic;
- In case of migrating the service to Azure cloud, it will be easy to switch to Redis cache for Azure.

Installing Redis on Windows

The latest release of Redis for Windows can be downloaded from <https://github.com/MicrosoftArchive/redis/releases>

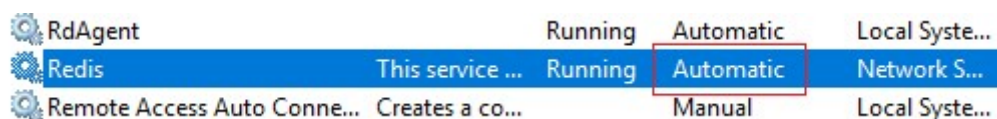
At the moment it is 3.2.100

Save and run Redis-x64-3.2.100.msi

The installation is quite standard. For testing purposes, you can leave all options by default. After installation, open Task Manager and check that the Redis service is running.



Also, be sure that the service is starting automatically. For this, open: Windows > Start menu > Administrative tools > Services.

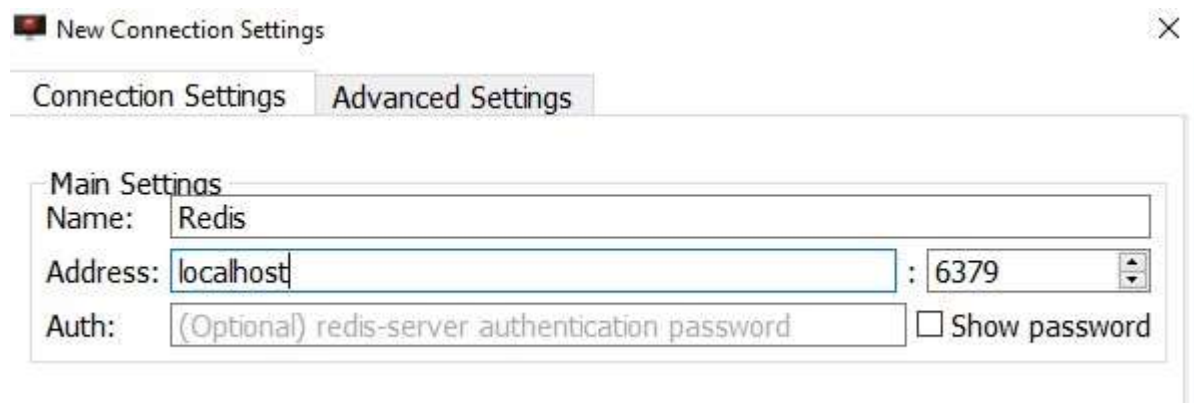


Redis Desktop Manager

For debugging purposes, it is convenient to have some client application for the Redis server to watch cached values. For this, Redis Desktop Manager can be used. You can download it from <https://redisdesktop.com/download>

The installation of Redis Desktop Manager is also very simple - everything is by default.

Open Redis Desktop Manager, click Connect to Redis server button and select Name: Redis and Address: localhost



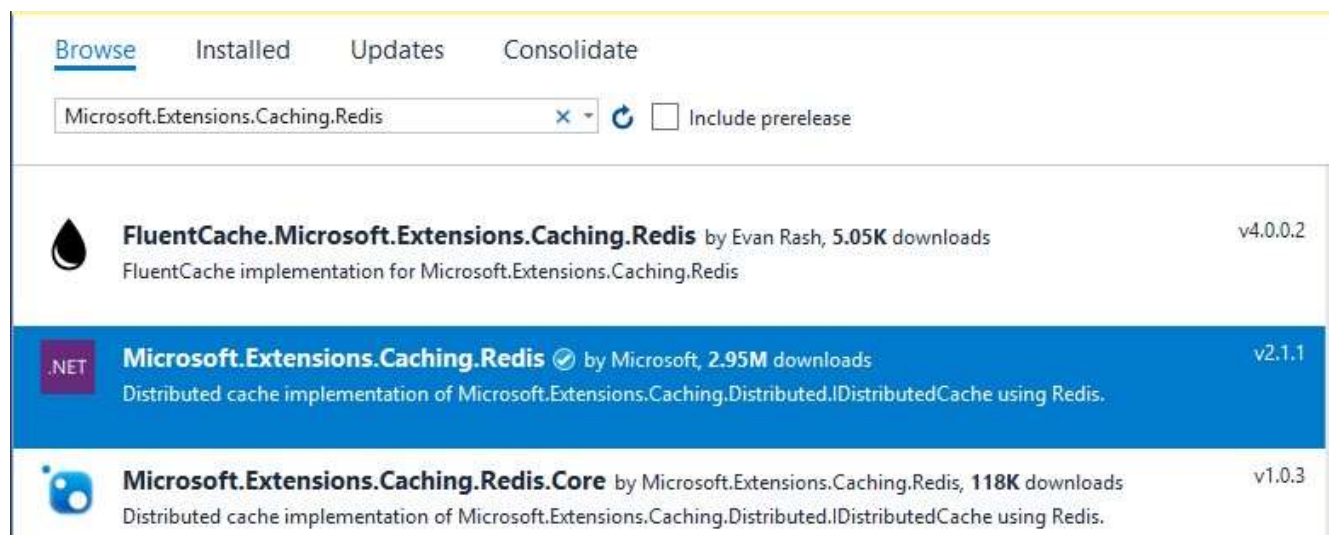
Then click OK button and you will see the content of Redis cache server.

Redis NuGet package

Add Redis NuGet package into our application:

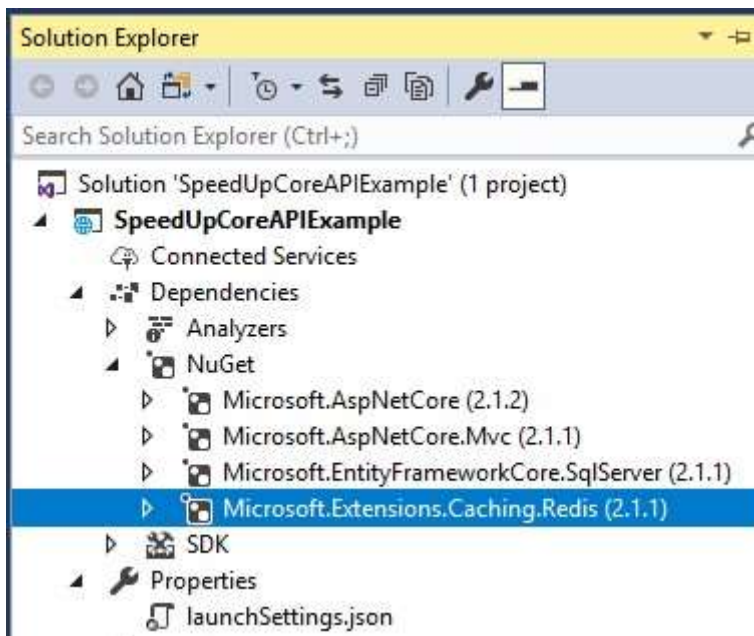
Main menu > Tools > NuGet Package Manager > Manager NuGet Packages For Solution

Input Microsoft.Extensions.Caching.Redis in the Browse field and select the package:



Note! Be sure to select exactly the official Microsoft package Microsoft.Extensions.Caching.Redis (but not Microsoft.Extensions.Caching.Redis.Core).

At this stage you must have the following packages installed:



Declare AddDistributedRedisCache before repositories in ConfigureServices methods of the Startup class

```
//Cache
services.AddDistributedRedisCache(options =>
{
    options.InstanceName = Configuration.GetValue<string>("Redis:Name");
    options.Configuration = Configuration.GetValue<string>("Redis:Host");
});
```

Add Redis connection settings in configuration file appsettings.json (and appsettings.Development.json)

```
"Redis": {
  "Name": "Redis",
  "Host": "localhost"
}
```

Caching expiration control

For caching either a Sliding or Absolute expiration model can be applied.

- Sliding expiration will be useful with prices when you have a huge list of Products, but only a small set of products is in great demand. So only prices of this set will be always cached. All the other prices will be removed from the cache automatically because they are rarely being requested and the Sliding expiration model continues caching only items that are re-requested within the specified period. This keeps memory free from unimportant data. The disadvantage of this method is that we have to implement some mechanism for removing items from the cache when prices are changed in the database.

- Absolute expiration model is what is used in the application. In this case all items will be equally cached for a specified period and then will be removed from the cache automatically. The problem of maintaining actual prices in the cache will be solved by itself, although perhaps with a slight delay.

Add a section for cache settings in the appsettings.json (and appsettings.Development.json) file.

```
"Caching": {  
  "PricesExpirationPeriod": 15  
}
```

Prices will be cached for 15 minutes.

Where to apply caching?

Since in the application architecture, services know nothing about the way of data storage, the proper place for caching is repositories, responsible for the infrastructure layer. For caching prices, RedisCache will be injected into PricesRepository with IConfiguration, which provides access to the cache settings.

Caching implementation

The last version of the PricesRepository class at this stage will be:

```
using Microsoft.EntityFrameworkCore;  
using Microsoft.Extensions.Caching.Distributed;  
using Microsoft.Extensions.Configuration;  
using Newtonsoft.Json;  
using SpeedUpCoreAPIExample.Contexts;  
using SpeedUpCoreAPIExample.Interfaces;  
using SpeedUpCoreAPIExample.Models;  
using System;  
using System.Collections.Generic;  
using System.Globalization;  
using System.Threading.Tasks;  
  
namespace SpeedUpCoreAPIExample.Repositories  
{  
    public class PricesRepository : IPricesRepository  
    {  
        private readonly Settings _settings;  
        private readonly DefaultContext _context;  
        private readonly IDistributedCache _distributedCache;  
  
        public PricesRepository(DefaultContext context, IConfiguration configuration,  
IDistributedCache distributedCache)  
        {  
            _settings = new Settings(configuration);  
  
            _context = context;  
            _distributedCache = distributedCache;  
        }  
  
        public async Task<IEnumerable<Price>> GetPricesAsync(int productId)  
        {  
            IEnumerable<Price> prices = null;
```

```

        string cacheKey = "Prices: " + productId;

        var pricesTemp = await _distributedCache.GetStringAsync(cacheKey);
        if (pricesTemp != null)
        {
            //Deserialize
            prices = JsonConvert.DeserializeObject<IEnumerable<Price>>(pricesTemp);
        }
        else
        {
            prices = await _context.Prices.AsNoTracking().FromSql("[dbo].GetPricesByProductId
@productId = {0}", productId).ToListAsync();

            //cache prices for PricesExpirationPeriod minutes
            DistributedCacheEntryOptions cacheOptions = new DistributedCacheEntryOptions()

                .SetAbsoluteExpiration(TimeSpan.FromMinutes(_settings.PricesExpirationPeriod));
            await _distributedCache.SetStringAsync(cacheKey,
                JsonConvert.SerializeObject(prices), cacheOptions);
        }

        return prices;
    }

    private class Settings
    {
        public int PricesExpirationPeriod = 15;        //15 minutes by default

        public Settings(IConfiguration configuration)
        {
            int pricesExpirationPeriod;
            if (Int32.TryParse(configuration["Caching:PricesExpirationPeriod"],
                NumberStyles.Any,
                NumberFormatInfo.InvariantInfo, out pricesExpirationPeriod))
            {
                PricesExpirationPeriod = pricesExpirationPeriod;
            }
        }
    }
}

```

Some explanation of the code:

In the constructor of the class DefaultContext, IConfiguration and IDistributedCache were injected. Then a new instance of class Settings (implemented at the bottom of the class PricesRepository) was created. Settings are used to reach the value of "PricesExpirationPeriod" in the section "Caching" of the configuration. In the Settings class type checking of PricesExpirationPeriod parameter is also provided. If a period is not integer, the default value (15 min) is used.

In GetPricesAsync method, we first try to get the prices list for a ProductId from the Redis cache, injected as IDistributedCache. If a value exists, we deserialize it and return a list of prices. If one does not exist, we get the list from the database and cache it for a number of minutes from PricesExpirationPeriod parameter of Settings.

Let us check how everything is working

In the Firefox or Chrome browser, start Swagger Inspector Extension (installed earlier) and call API <http://localhost:49858/api/prices/1>

API responds with Status: 200 OK and list of prices for the Product1:

```
Status: 200 OK    Time: 241 ms

1  [
2    {
3      "price": 100,
4      "supplier": "Bosch"
5    },
6    {
7      "price": 125,
8      "supplier": "LG"
9    },
10   {
11     "price": 130,
12     "supplier": "Garmin"
13   }
14 ]
```

Open the Redis Desktop Manager, connect to the Redis server. Now we can see a group RedisPrices and cached value for the key Prices: 1

The screenshot shows the Redis Desktop Manager interface. On the left, a tree view shows the hierarchy: Redis > db0 (1) > RedisPrices (1) > RedisPrices: 1. On the right, the 'HASH: RedisPrices: 1' section displays a table of cached data.

| row | key | value |
|-----|--------|--------------------------------------|
| 1 | sldexp | -1 |
| 2 | data | [{"PriceId":1,"ProductId":1,"Price": |
| 3 | absexp | 636669216997917288 |

Prices for the Product1 are cached and next calls of the API `api/prices/1` within 15 minutes will take them from the cache, not from the database.

Prepare data in advance concept

In a case when we have a huge database, or the prices are only basic and must be additionally recalculated for a specific user, the increase in the response speed could be much higher if we prepare prices before the users apply for them and cache precalculated prices for following requests.

Let us analyze `api/products/find` API results with the parameter "aa"

<http://localhost:49858/api/products/find/aa>

We can find two positions whose sku consists of "aa". At this stage, we do not know which one can be requested for prices by a user.

```
Status: 200 OK    Time: 203 ms

1  [
2  {
3      "id": 1,
4      "sku": "aaa",
5      "name": "Product1"
6  },
7  {
8      "id": 2,
9      "sku": "aab",
10     "name": "Product2"
11  }
12 ]
```

But if the parameter is "abc", we will obtain only one Product in response.

```
[
{
    "id": 3,
    "sku": "abc",
    "name": "Product3"
}
```

The most probable next step of the user will be requesting prices for this particular product. If we get prices for the product at this stage and cache the result, the next call of API <http://localhost:49858/api/prices/3> will take ready prices from the cache and save a lot of time and SQL Server activity.

Prepare data in advance implementation

For implementation of this idea, we create PreparePricesAsync method in PricesRepository and PricesService

First declare this method in the interfaces IPricesRepository and IPricesService. In both cases the method will return nothing.

```
using SpeedUpCoreAPIExample.Models;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace SpeedUpCoreAPIExample.Repositories
{
    public interface IPricesRepository
    {
        Task<IEnumerable<Price>> GetPricesAsync(int productId);
    }
}
```

```

        Task PreparePricesAsync(int productId);
    }
}

```

and

```

using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace SpeedUpCoreAPIExample.Interfaces
{
    public interface IPricesService
    {
        Task<ActionResult> GetPricesAsync(int productId);
        Task PreparePricesAsync(int productId);
    }
}

```

The PreparePricesAsync method of PricesService just calls the PreparePricesAsync of PricesRepository inside try-catch construction. Note that there is not any exception handling in PreparePricesAsync procedure, and that we just completely ignore possible errors. This is because we do not want to break the flow of the program in this place, as there is still a possibility that the user will never request prices of this product and an error message can be an undesirable obstacle in his work.

```

public async Task PreparePricesAsync(int productId)
{
    IEnumerable<Price> prices = null;

    string cacheKey = "Prices: " + productId;

    var pricesTemp = await _distributedCache.GetStringAsync(cacheKey);
    if (pricesTemp != null)
    {
        //already cached
        return;
    }
    else
    {
        prices = await _context.Prices.AsNoTracking().FromSql("[dbo].GetPricesByProductId @productId = {0}", productId).ToListAsync();

        //cache prices for PricesExpirationPeriod minutes
        DistributedCacheEntryOptions cacheOptions = new DistributedCacheEntryOptions()
            .SetAbsoluteExpiration(TimeSpan.FromMinutes(_settings.PricesExpirationPeriod));
        await _distributedCache.SetStringAsync(cacheKey, JsonConvert.SerializeObject(prices),
            cacheOptions);
    }
    return;
}

```

In PricesService.cs

```

using System;
...
public async Task PreparePricesAsync(int productId)

```

```

{
    try
    {
        await _pricesRepository.PreparePricesAsync(productId);
    }
    catch (Exception ex)
    {
    }
}

```

Let us check how the PreparePricesAsync methods work. First inject PricesService into ProductsService:

```

private readonly IProductsRepository _productsRepository;
private readonly IPricesService _pricesService;

public ProductsService(IProductsRepository productsRepository, IPricesService pricesService)
{
    _productsRepository = productsRepository;
    _pricesService = pricesService;
}

```

Note! We have injected PricesService into ProductsService for test purposes only. Coupling services this way is not a good practice because it will make things difficult if we decide to implement microservices architecture. In the ideal microservices world, services should not depend on each other.

But let us go further and create PreparePricesAsync method in the Product Service class. The method will be a Private and so it need not be declared in the IProductsRepository interface.

```

private async Task PreparePricesAsync(int productId)
{
    await _pricesService.PreparePricesAsync(productId);
}

```

The method does nothing but call the PreparePricesAsync method of the PricesService.

Then, in FindProductsAsync, check if there is only one item in the search result for products list. If there is only one, we call the PreparePricesAsync of PricesService for product Id of this single item. Note, that we call _pricesService.PreparePricesAsync before we return the products list to the user – the reason will be explained later.

```

public async Task<IActionResult> FindProductsAsync(string sku)
{
    try
    {
        IEnumerable<Product> products = await _productsRepository.FindProductsAsync(sku);

        if (products != null)
        {
            if (products.Count() == 1)
            {
                //only one record found - prepare prices beforehand
                await PreparePricesAsync(products.FirstOrDefault().ProductId);
            }
        }
    }
}

```

```

        return new OkObjectResult(products.Select(p => new ProductViewModel()
        {
            Id = p.ProductId,
            Sku = p.Sku,
            Name = p.Name
        }
        ));
    }
    else
    {
        return new NotFoundResult();
    }
}
catch
{
    return new ConflictResult();
}
}

```

And we can also add PreparePricesAsync in the GetProductAsync method.

```

public async Task<IActionResult> GetProductAsync(int productId)
{
    try
    {
        Product product = await _productsRepository.GetProductAsync(productId);

        if (product != null)
        {
            await PreparePricesAsync(productId);

            return new OkObjectResult(new ProductViewModel()
            {
                Id = product.ProductId,
                Sku = product.Sku,
                Name = product.Name
            }
            ));
        }
        else
        {
            return new NotFoundResult();
        }
    }
    catch
    {
        return new ConflictResult();
    }
}

```

Remove cached values from Redis cache, start the application and call <http://localhost:49858/api/products/find/abc>

Open the Redis Desktop Manager and check cached values. You can find the prices list for "ProductId":3

```

[
  {
    "PriceId": 7,
    "ProductId": 3,
    "Value": 160.00,

```

```

    "Supplier": "Bosch"
  },
  {
    "PriceId": 8,
    "ProductId": 3,
    "Value": 165.00,
    "Supplier": "LG"
  },
  {
    "PriceId": 9,
    "ProductId": 3,
    "Value": 170.00,
    "Supplier": "Garmin"
  }
]

```

Then check /api/products/3 API. Remove data from the cache and call <http://localhost:49858/api/products/3>

Check in Redis Desktop Manager and you will find that this API also caches prices properly.

But we have not achieved any gain in speed because we called the asynchronous method GetProductAsync synchronously - the application workflow waited until GetProductAsync prepared the price list. So, our API did the job of two APIs.

To solve the problem, we should execute GetProductAsync in a separate thread. In this case, the result of api/products will be delivered to a user immediately. At the same time, the GetProductAsync method will continue working until it prepares prices and caches the result.

For this, we have to change the declaration of PreparePricesAsync method a little – let it return void.

In ProductsService:

```

private async void PreparePricesAsync(int productId)
{
    await _pricesService.PreparePricesAsync(productId);
}

```

Add System.Threading namespace into ProductsService class.

```

using System.Threading

```

Now we can change calling of this method for a thread.

In FindProductsAsync method:

```

...
if (products.Count() == 1)
{
    //only one record found - prepare prices beforehand
    ThreadPool.QueueUserWorkItem(delegate
    {

```

```

        PreparePricesAsync(products.FirstOrDefault().ProductId);
    });
};
...

```

And in GetProductAsync method:

```

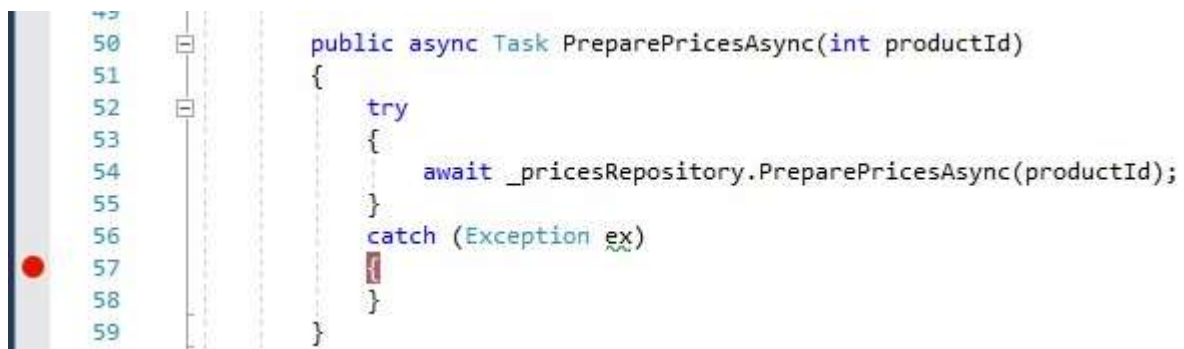
...
ThreadPool.QueueUserWorkItem(delegate
{
    PreparePricesAsync(productId);
});
...

```

Everything seems to be all right. Remove cached values from Redis cache, start the application and call <http://localhost:49858/api/products/find/abc>

The result status is Status: 200 OK, but the cache is still empty. So, some error occurred, but we cannot see it because we have not performed errors handling in the PricesService for the PreparePricesAsync method.

Let us set a breakpoint in PreparePricesAsync method of PricesService just after the catch statement:



```

49
50
51
52
53
54
55
56
57
58
59

public async Task PreparePricesAsync(int productId)
{
    try
    {
        await _pricesRepository.PreparePricesAsync(productId);
    }
    catch (Exception ex)
    {
    }
}

```

Then call API <http://localhost:49858/api/products/find/abc> ones again.

Now we have an exception and can check the details:

System.ObjectDisposedException: 'Cannot access a disposed object. A common cause of this error is disposing a context that was resolved from dependency injection and then later trying to use the same context instance elsewhere in your application. This may occur if you are calling Dispose() on the context, or wrapping the context in a using statement. If you are using dependency injection, you should let the dependency injection container take care of disposing context instances.'

This means, that when the result is sent to a user, we cannot use DbContext which was injected using dependency injection anymore, because the DbContext is already disposed at this time. And it does not matter how deep the DbContext injected in our chain of dependency injections.

Let us examine whether we can do the job without the dependency injections of DbContext. In PricesRepository.PreparePricesAsync, we will create DbContext dynamically and use it inside Using construction.

add EntityFrameworkCore namespace

```
using Microsoft.EntityFrameworkCore
```

The block of getting prices will look like that:

```
using Microsoft.EntityFrameworkCore
...

public async Task PreparePricesAsync(int productId)
{
    ...

    var optionsBuilder = new DbContextOptionsBuilder<DefaultContext>();
    optionsBuilder.UseSqlServer(_settings.DefaultDatabase);

    using (var _context = new DefaultContext(optionsBuilder.Options))
    {
        prices = await _context.Prices.AsNoTracking().FromSql("[dbo].GetPricesByProductId
@productId = {0}", productId).ToListAsync();
    }
    ...
}
```

and add two lines in the Settings class:

```
public string DefaultDatabase;
...

DefaultDatabase = configuration["ConnectionStrings:DefaultDatabase"];
```

Then start the application and try <http://localhost:49858/api/products/find/abc> ones again.

There are no errors now and the prices are cached in Redis cache. If we set breakpoints inside the PricesRepository.PreparePricesAsync method and call the API again, we can see, that the program stops at this breakpoint after the result is sent to a user. So, we achieved our aim – the prices are being prepared beforehand in background and this process does not block the flow of the application.

But this solution is not ideal. Some problems are:

- By injecting PricisService into ProductsService we couple services and so make it difficult to apply microservices architecture, if we want to;
- We cannot get advantages of dependency injections for a DbContext;
- Mixing approaches, we make our code less unified and therefore more confusing.

Thinking of microservices architecture

In this article, we are describing a monolith application, but after all productivity improvements have been done, a possible way to increase a heavily loaded application performance could be its horizontal scaling. For this, the application could probably be split into two microservices, ProductsMicroservice and PricesMicroservice. If the ProductsMicroservice wants to prepare prices in advance it will call the appropriate method of the PricesMicroservice. This method should be accessed via an API.

We will follow this idea, but implement it within our monolith application. First, we will create an API `api/prices/prepare` in the PricesController and then call this API from the ProductsService via Http request. This should resolve all the problems we are having with the dependency injections for a DbContext and prepare the application to the microservices architecture. And another benefit of using Http request even in a monolith is that in a multitenant application behind the load balancer this request might be processed by another instance of the application and thus, we will get the benefits of the horizontal scaling.

First, let us return our PricesRepository to the state before we started testing `PreparePricesAsync` method: in `PricesRepository.PreparePricesAsync` method, we remove "using" statement and leave just one line:

```
public async Task PreparePricesAsync(int productId)
{
    ...

    prices = await _context.Prices.AsNoTracking().FromSql("[dbo].GetPricesByProductId @productId"
= {0}", productId).ToListAsync();

    ...
}
```

and also remove `DefaultDatabase` variable from the `PricesRepository.Setting` class.

Creating API for prices preparation

In PricesController add the method:

```
// POST api/prices/prepare/5
[HttpPost("prepare/{id}")]
public async Task<IActionResult> PreparePricesAsync(int id)
{
    await _pricesService.PreparePricesAsync(id);

    return Ok();
}
```

Note, that the call method is POST, as we are not going to get any data with this API. And the API always returns OK – if some error occurs during API execution, it will be ignored as it is unimportant at this stage.

Clear Redis cache, start our application, call POST `http://localhost:49858/api/prices/prepare/3`

The API works fine - we have Status: 200 OK and pricelist for the Product3 is cached.

So, our intention is to call this new API from the code of the `ProductsService.PreparePricesAsync` method. To do this we must decide, how to get the URL of the API. We will obtain the URL in the `GetFullyQualifiedApiUrl` method. But how can we get the URL inside a service class if we do not have access to current `HttpContext` to find out the host and working protocol and port?

There are at least three possibilities we can use:

- Put fully qualified API URL into configuration file. This is the simplest way but can cause some problems in the future if we decide to move the application to another infrastructure – we shall have to care about actual URL in the config file;
- Current `HttpContext` is available at a Controller level. So, we can determine the URL there and pass it as a parameter to the `ProductsService.PreparePricesAsync` method, or even pass the `HttpContext` itself. Both options are not very good, as we do not want to implement any business logic in Controllers and from the point of view of a Service class it becomes dependent of a Controller and as a result tests of a Service will be much more difficult to establish;
- Use `HttpContextAccessor` service. It provides access to `HttpContext` anywhere in the application. And it can be injected via dependency injection. Of course, we choose this approach as a universal and native to ASP.NET Core.

To implement this, we register `HttpContextAccessor` in the `ConfigureServices` method of the `Startup` class:

```
using Microsoft.AspNetCore.Http;
...
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
    ...
}
```

The scope of the service should be `Singleton`.

Now we can use `HttpContextAccessor` in the `ProductService`. Inject `HttpContextAccessor` instead of `PriceService`:

```
using Microsoft.AspNetCore.Http;
...
public class ProductsService : IProductsService
{
    private readonly IProductsRepository _productsRepository;
    private readonly IHttpContextAccessor _httpContextAccessor;
    private readonly string _apiUrl;

    public ProductsService(IProductsRepository productsRepository, IHttpContextAccessor httpContextAccessor)
    {
        _productsRepository = productsRepository;
        _httpContextAccessor = httpContextAccessor;

        _apiUrl = GetFullyQualifiedApiUrl("/api/prices/prepare/");
    }
}
```

...

Add a method `ProductsService.GetFullyQualifiedApiUrl` with the code:

```
private string GetFullyQualifiedApiUrl(string apiRoute)
{
    string apiUrl = string.Format("{0}://{1}{2}",
        _httpContextAccessor.HttpContext.Request.Scheme,
        _httpContextAccessor.HttpContext.Request.Host,
        apiRoute);

    return apiUrl;
}
```

Note, that we set the value of the `_apiUrl` variable at the class constructor. And we decoupled the `ProductService` and `PricesService` by removing dependency injection of `PricesService` and changing the `ProductService.PreparePricesAsync` method – call new API instead of invoking `PriceService.PreparePricesAsync` method:

```
using System.Net.Http;
...
private async void PreparePricesAsync(int productId)
{
    using (HttpClient client = new HttpClient())
    {
        var parameters = new Dictionary<string, string>();
        var encodedContent = new FormUrlEncodedContent(parameters);

        try
        {
            var result = await client.PostAsync(_apiUrl + productId,
                encodedContent).ConfigureAwait(false);
        }
        catch
        {
        }
    }
}
```

In this method we call the API inside try-catch without errors handling.

Clear Redis cache, start our application, call <http://localhost:49858/api/products/find/abc> or <http://localhost:49858/api/products/3>

The API works fine - we have Status: 200 OK and the pricelist for the Product3 cached.

HttpClient issues

Using an `HttpClient` inside a "Using" construction is not the best solution and we have used it just as a proof of concept. There are two points where we can lose productivity:

- Each `HttpClient` has its own connection pool for storing and reusing connections. But if you create a new `HttpClient` for each request, the connection pools of previously created `HttpClient`s

cannot be reused by the new HttpClient. So, it has to waste time establishing a new connection to the same server;

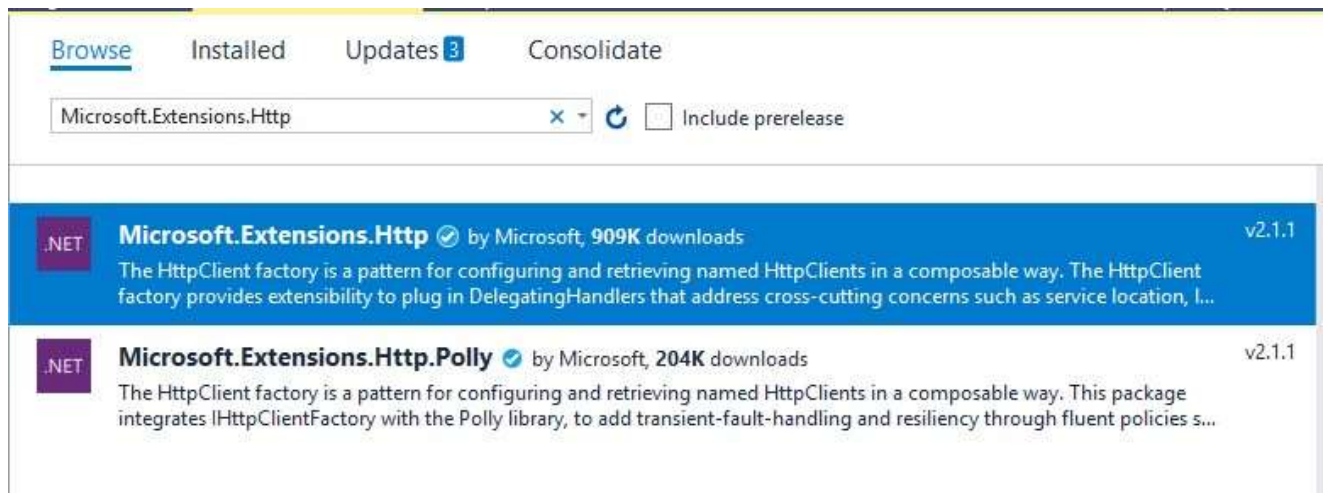
- After disposing of the HttpClient at the end of "Using" construction, its connections are not released immediately. Instead they wait for some time in TIME_WAIT state, blocking ports assigned to them. In a heavily loaded application a lot of connections are created for a short period of time but remain unavailable for reuse (for 4 minutes, by default). This inefficient use of the resources can cause a significant loss of productivity and even lead to a "socket exhaustion" problem and application crash.

One of the possible solutions for this problem is to have one HttpClient per Service and add the Service as Singleton. But we will apply another approach – using HttpClientFactory for managing our HttpClients in a proper way.

Manage HttpClients with HttpClientFactory

HttpClientFactory controls the lifetime of HttpClients' handlers, makes them reusable and thus prevents the application from inefficiently using the resources.

HttpClientFactory has been available since ASP.NET Core 2.1. To add it to our application we should install the Microsoft.Extensions.Http NuGet package:



Register the default HttpClientFactory in the Startup.cs file of the application by applying the AddHttpClient() method:

```
...
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
    services.AddHttpClient();
}
...
```

In the ProductsService class, inject HttpClientFactory via dependency injection:

```

...
private readonly IProductsRepository _productsRepository;
private readonly IHttpContextAccessor _httpContextAccessor;
private readonly IHttpClientFactory _httpClientFactory;

private readonly string _apiUrl;

public ProductsService(IProductsRepository productsRepository, IHttpContextAccessor
httpContextAccessor, IHttpClientFactory httpClientFactory)
{
    _productsRepository = productsRepository;
    _httpContextAccessor = httpContextAccessor;
    _httpClientFactory = httpClientFactory;

    _apiUrl = GetFullyQualifiedApiUrl("/api/prices/prepare/");
}
...

```

Correct the PreparePricesAsync method - remove "Using" construction and create an HttpClient by means of the .Create Client() method of the injected HttpClientFactory:

```

...
private async void PreparePricesAsync(int productId)
{
    var parameters = new Dictionary<string, string>();
    var encodedContent = new FormUrlEncodedContent(parameters);

    try
    {
        HttpClient client = _httpClientFactory.CreateClient();
        var result = await client.PostAsync(_apiUrl + productId,
encodedContent).ConfigureAwait(false);
    }
    catch
    {
    }
}
...

```

The .CreateClient() method reuses HttpClientHandlers by taking one from the pool and passing it to a newly created HttpClient.

The last stage is passed, our application prepares prices in advance and does this following the .NET Core paradigm in an effective and resilient way.

Summary

Finally, we have our application with various increasing productivity approaches applied.

Comparing to test application in Part 1, the latest version is much faster and uses the infrastructure much more effectively.

Points of Interest

During Part 1 and Part 2 we were developing the application step by step with the main focus on easiness to apply and examine different approaches, to modify code and check results. But now, after we have made a choice of approaches and implemented them, we can consider our application as a whole. And it becomes evident that the code requires some refactoring.

Therefore, there Part 3 named "Create a Perfect WEB API Core application" is coming in the near future. In Part 3 we will focus on a concise code, global error handling, input parameters validating, documenting, testing and other important features that a good written application must have.