Engineer Coding Exercise for Team Health

Prompt:

1. Create .Net Core or .Net 5 API for a hypothetical blogging application that will have the capacity to retrieve a list of blog posts and insert new posts.
2. The class structure of the post is up to you, Include common fields are Id, Title, Body, and AuthorID. Do not include any related entities like comments, etc.

No authorization/authentication is required. Your Solution should utilize EF Core to seed sample post data in the database.

3. Implement API endpoints for:
    a. Get Posts
        i. Return a list of posts in JSON format.
        ii. Handle pagination for limiting the number of records returned
        iii. specifying the starting record offset (ex. offset=1&limit=10) as parameters
        iv. Specifying the sort order (ex: sortby=Title) and Sort direction (ex: sort=asc)
    b. Insert New Post
        i. Insert a new post into the database
        ii. Use postman testing tool
        iii. Send an Email to the blog owner advising that a new post was added
        iv. Use **smtp** server of your choice to send message or Local **smtp** testing utility such as **Papercut**

Solution.

The point-by-point solution is described below, according to the requirement specified above.

1. Create .Net Core or .Net 5 API for a hypothetical blogging application that will have the capacity to retrieve a list of blog posts and insert new posts.
    a. The API was made usin .NET 5

2. The class structure of the post is up to you, Include common fields are Id, Title, Body, and AuthorID. Do not include any related entities like comments, etc.
    a. The Class BlogPost under Models Folder defines the Entity (table) for the blog posting

```
public class BlogPost
```

```
  {
      [Key] // it makes primary key and Identity incremented by 1
      public int BlogPostId { get; set; }

      [Required] // it makes not null constraint
      [StringLength(150)] // it makes nvarchar(150)
      //[MaxLength(150)]
      public string Title { get; set; }

      [Required]
      public string Body { get; set; } // nvarchar(max) by default

      ////Foreign key for Author
      public int AuthorId { get; set; }
      //public Author Author { get; set; }

      [Required]
      //[DatabaseGenerated(DatabaseGeneratedOption.Computed)] // for default value
      public DateTime CreatedAt { get; set; } // = DateTime.Now;

      [Required]
      public DateTime UpdatedAt { get; set; }
  }
```

No authorization/authentication is required. Your Solution should utilize EF Core to seed sample post data in the database.

3. Implement API endpoints for:
    a. Get Posts
        i. Return a list of posts in JSON format.

            1. To make sure that the format response is in JSON the line below was added in the WebApiConfig.cs to set up JSON Format:

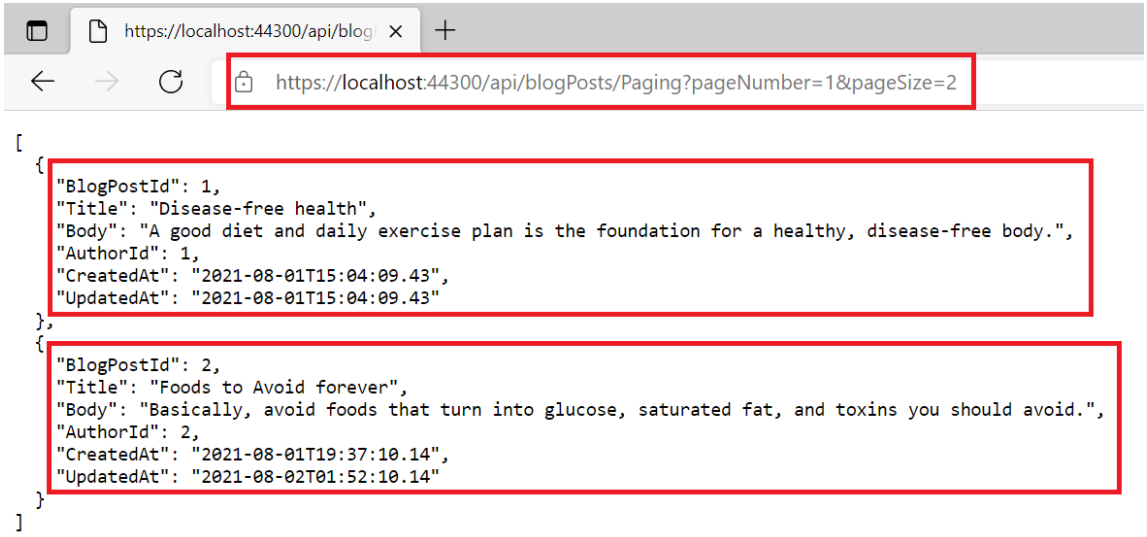
```
config.Formatters.Remove(config.Formatters.XmlFormatter);
```

Removing the Xml format, the JSON format remains by default

        ii. Handle pagination for limiting the number of records returned
            1. A new function was created for pagination as follows:

```
[HttpGet]
      [Route("api/blogposts/Paging/{pageNumber=1}/{pageSize=5}")]
      public IHttpActionResult PagingBlogPosts(int pageNumber, int
pageSize)
      {
```

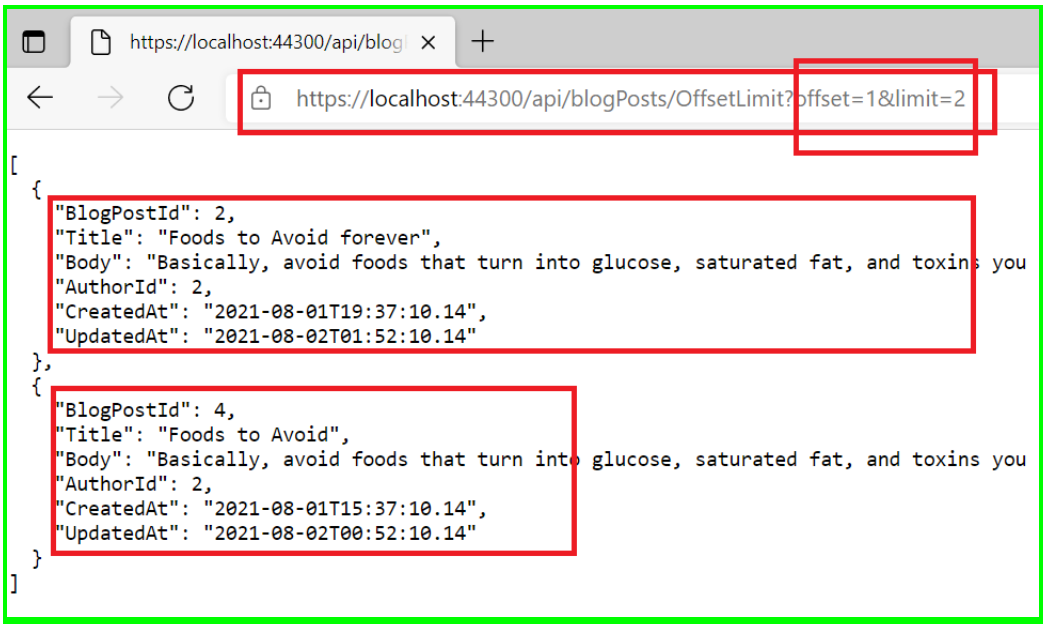
```
            var blogPosts = blogPostsDbContext.BlogPosts.OrderBy(q =>
q.BlogPostId);
            return Ok(blogPosts.Skip((pageNumber - 1) *
pageSize).Take(pageSize));
        }
```



iii. specifying the starting record offset (ex. offset=1&limit=10) as parameters

1. A new function was created for Offset and Limit as follows:

```
[HttpGet]
[Route("api/blogposts/OffsetLimit/{offset=0}/{limit=2}")]
public IHttpActionResult OffsetLimitBlogPosts(int offset, int limit)
        {
            var blogPosts = blogPostsDbContext.BlogPosts.OrderBy(q =>
q.BlogPostId);
            return Ok(blogPosts.Skip(offset).Take(limit));
        }
```

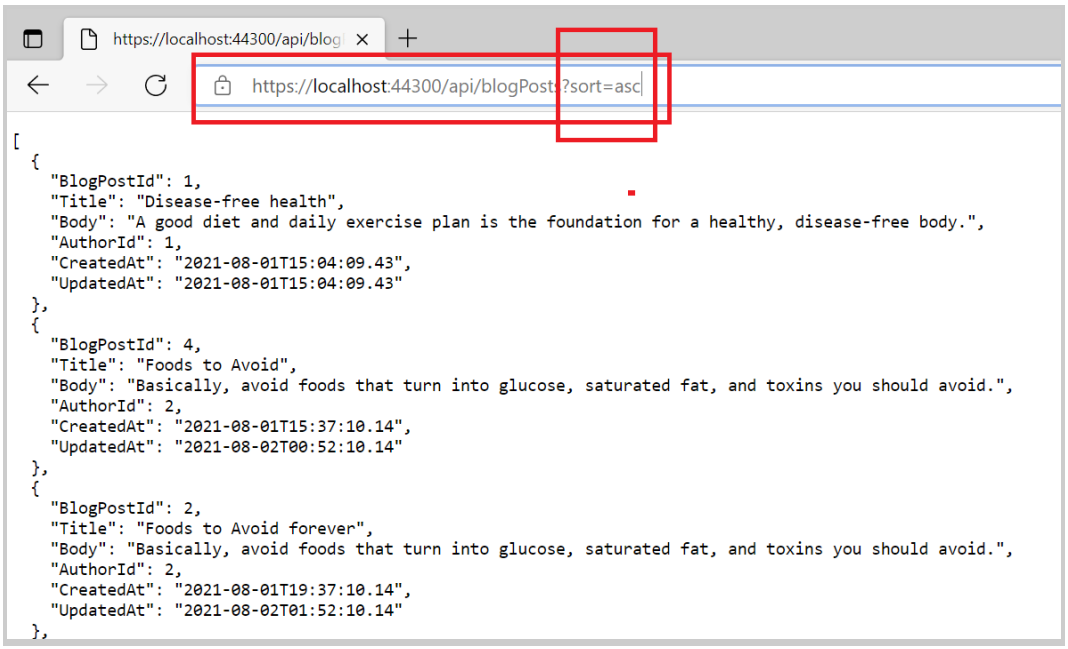


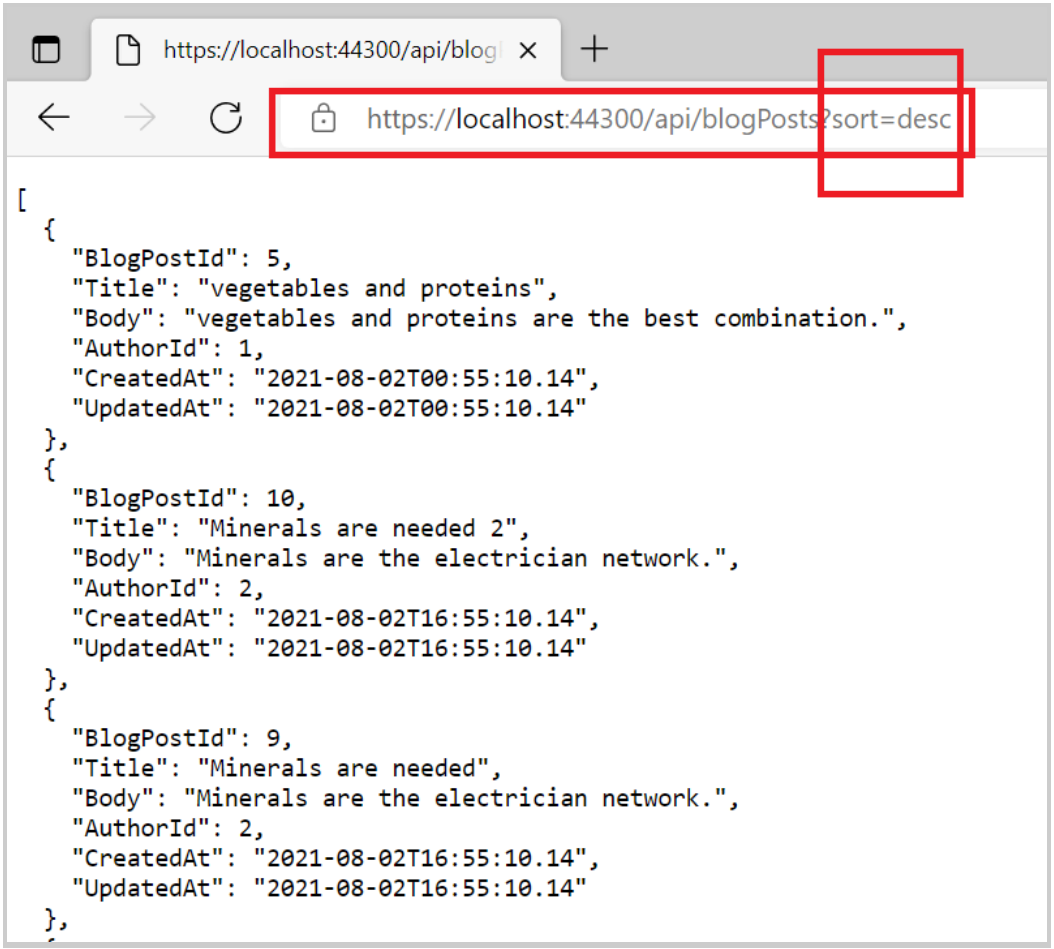
iv. Specifying the sort order (ex: sortby=Title) and Sort direction (ex: sort=asc)

1. A new function was created for Sorting as follows:

```
        [HttpGet]
        public IHttpActionResult LoadBlogPosts(string sort)
        {
        IQueryable<BlogPost> blogPosts;
        switch (sort)
        {
            case "desc":
                blogPosts =
blogPostsDbContext.BlogPosts.OrderByDescending(q => q.Title);
                break;
            case "asc":
                blogPosts = blogPostsDbContext.BlogPosts.OrderBy(q => q.Title);
                break;
            default:
                blogPosts = blogPostsDbContext.BlogPosts;
                break;

        }
        //var blogPosts = blogPostsDbContext.BlogPosts;
        return Ok(blogPosts);
    }
```



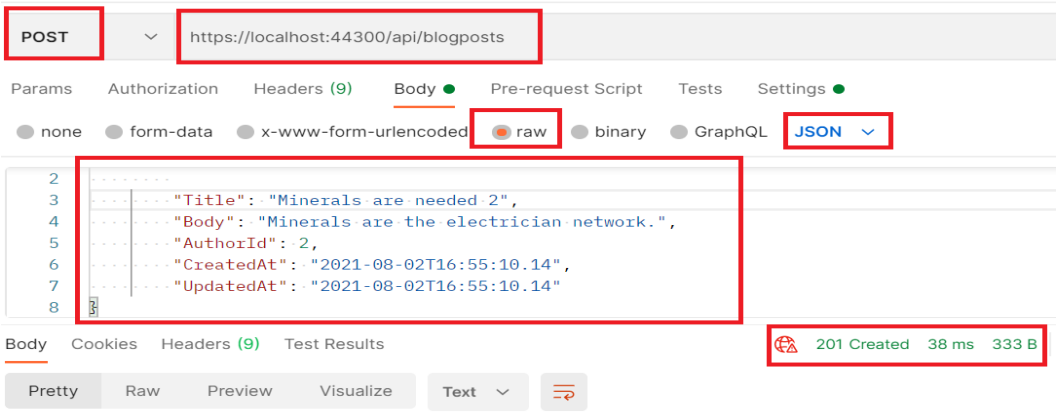


b. Insert New Post
    i. Insert a new post into the database

1. The POST implementation is below:

```
// POST: api/BlogPosts
        [HttpPost]
        public IHttpActionResult Post([FromBody]BlogPost blogPost)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            blogPostsDbContext.BlogPosts.Add(blogPost);
            blogPostsDbContext.SaveChanges();
            return StatusCode(HttpStatusCode.Created);
        }
```

ii. Use postman testing tool



iii. Send an Email to the blog owner advising that a new post was added. Use **smtp** server of your choice to send message or Local **smtp** testing utility such as **Papercut**

1. For this part, I used "smtp Sql Server". I set up the “database mail service” with the following script below.

```
-- To enable Database Mail, run the following code:
sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO

-- This is going to happen from time to time because this is an advanced option. To fix this, we need to change the show advanced options default value from 0 to 1.
-- To do this run the following code:
sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
```

```
GO

sp_configure 'Database Mail XPs', 1;
GO
RECONFIGURE
GO

-- To create a new Database Mail profile named ‘Notifications’ we will use the
sysmail_add_profile_sp stored procedure and the following code:
-- Create a Database Mail profile
EXECUTE msdb.dbo.sysmail_add_profile_sp
    @profile_name = 'Notifications',
    @description = 'Profile used for sending outgoing notifications using Gmail.' ;
GO

-- To grant permission for a database user or role to use this Database Mail profile, we will use the
sysmail_add_principalprofile_sp stored procedure and the following code:
-- Grant access to the profile to the DBMailUsers role
EXECUTE msdb.dbo.sysmail_add_principalprofile_sp
    @profile_name = 'Notifications',
    @principal_name = 'public',
    @is_default = 1 ;
GO


-- To create a new Database Mail account holding information about an SMTP account, we will
use the sysmail_add_account_sp stored procedure and the following code:
-- Create a Database Mail account
EXECUTE msdb.dbo.sysmail_add_account_sp
    @account_name = 'Gmail',
    @description = 'Mail account for sending outgoing notifications.',
    @email_address = 'lcruz@cavanny.com',
    @display_name = 'Automated Mailer',
    @mailserver_name = 'smtp.gmail.com',
    @port = 587,
    @enable_ssl = 1,
    @username = 'lcruz@cavanny.com',
    @password = 'DutyFree123' ;
GO

-- To add the Database Mail account to the Database Mail profile, we will use the
sysmail_add_profileaccount_sp stored procedure and the following code:
-- Add the account to the profile
EXECUTE msdb.dbo.sysmail_add_profileaccount_sp
    @profile_name = 'Notifications',
    @account_name = 'Gmail',
    @sequence_number =1 ;
GO

-- Test
EXEC msdb.dbo.sp_send_dbmail
     @profile_name = 'Notifications',
     @recipients = 'lcgusa64@gmail.com',
     @body = 'The database mail configuration was completed successfully.',
     @subject = 'Automated Success Message';
GO
```

2. I created a database trigger Post-Insert in the blogPost table; When a new row of blog-post is inserted the notification email is triggered.

```
USE [BlogPostsDb]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Author:                  Lorenzo Cruz
-- Create date: 08/02/2021
-- Description:     Send an Email to the blog owner advising that a new post was added
-- =============================================
ALTER TRIGGER SendEmail_to_BlogOwner
  ON  [dbo].[BlogPosts]
  AFTER INSERT
AS
BEGIN

DECLARE @query NVARCHAR(1000),
      @Title NVARCHAR(150),
                  @Body NVARCHAR(MAX),
                  @Email NVARCHAR(250),
                  @BlogPostId int,
                  @AuthorId int,
                  @session_usr NVARCHAR(60) = SESSION_USER

          SET NOCOUNT ON;

   SELECT  @BlogPostId = I.BlogPostId,
              @Title = I.Title,
              @Body = I.Body,
                        @Email = AU.[Email],
                        @AuthorId = I.AuthorId
    FROM  Inserted I
          INNER JOIN [dbo].[Authors] AU
          ON AU.[AuthorId] = I.AuthorId

          INSERT INTO [dbo].[BlogPosts_Audit] -- Auditing Table
          (BlogPostId,  Title,  Body,  AuthorId,  Email,  session_usr,  CreatedAt) values
          (@BlogPostId, @Title, @Body, @AuthorId, @Email, @session_usr, getdate())

          set @query='msdb.dbo.sp_send_dbmail
@profile_name="Notifications",@recipients='" + @Email +
                  '",@subject='" + @Title +
                                    '",@body='" + @Body +
                                    ""

    EXEC @query

END
GO
```

3.