nip-lab-2018

Prerequisites

- Either one of those:
  - Visual Studio 2017 version 15.7 or later with:
    - ASP.NET and web development
    - .NET Core cross-platform development
  - Visual Studio Code
    - Plugin: C# for Visual Studio Code
    - IF ERRORS then use C# for Visual Studio Code version 1.15.2 since there is a bug when older version of VS is installed alongside.
      a. Download that vsix file, go to VS Code > Extensions > ... > Install from VSIX...
      b. VS Code > Extensions > ... > Disable Auto Updating Extensions
    - .NET Core 2.1 SDK or later
- Postman

API Requirements (RESTful)

- API:

API	Description	Request body	Response body	HTTP status code
GET /api/v1/blogposts	Get all blog posts	-	{ array of posts }	200 OK
GET /api/v1/blogposts/{id}	Get a post by id	-	{ post }	200 OK
POST /api/v1/blogposts	Add a new blog post	{ body }	{ post }	201 Created
PUT /api/v1/blogposts/{id}	Update an existing blog post	{ post }	-	200 OK
DELETE /api/v1/blogposts/{id}	Delete a blog post	-	-	204 No Content

- Anonymous, but secured access (use SSL for any publicly exposed API)
- Document your API
- Version API via the URL
- Default to JSON for both request and response
- API should always return sensible HTTP status codes. API errors typically break down into 2 types: 400 series status codes for client issues & 500 series status codes for server issues. At a minimum, the API should standardize that all 400 series errors come with consumable JSON error representation

Laboratories

Setup steps

- Using Visual Studio
  i. From the File menu, select New > Project > Other Project Types > Blank Solution.
  ii. Name it Nip.Blog, this follows the common patter `<company>.<product>.*`
  iii. Right click on solution name and add folder path `Services/Posts`
  iv. Right click on `Services/Posts` , select Add > New project...

v. Select the *ASP.NET Core Web Application* template. Name the project Posts.Api and click OK.

vi. In the *New ASP.NET Core Web Application* dialog, choose 2.1 or later as the ASP.NET Core version. Select the API template and click OK. Do not select Enable Docker Support.

vii. *(optinal) Rename the default namespace (everywhere) from* `Posts.API` *to* `Nip.Blog.Services.Posts.API` *for consistency (* `<company>.<product>.<>` *)*

viii. Right click the `ValuesController.cs` and rename it to `BlogPostsController.cs`

ix. At the toolbar switch the debug target from `IIS Express` to `Nip.Blog.Services.Posts.API` and run it. *(the reason behind this is that we want to use .net core runtime to run the server for us and not the IIS)*

x. Before the debug starts VS should ask you for permission to add "fake" SSL certificate to the cert store.

- *(optional)* Using Visual Studio Code
  i. Create `Services/Posts` subfolder, navigate there and run commands:

```
dotnet new webapi -o Posts.API
dotnet dev-certs https --trust
code Posts.API
```

  ii. Accept any popups that appear (like `Restore`)

  iii. Right click the `ValuesController.cs` and rename it to `BlogPostsController.cs`

  iv. Start the server by either
    - Running the Debug (F5)
    - Running `dotnet run` from VS Code Terminal
    - Running `dotnet run` when in project folder
    - Running `dotnet Posts.API.dll` when in /bin/Debug folder
- Two app URLs should be available at this point: secure `https://localhost:5001` if SSL is enabled; unsecure `http://localhost:5000` that redirects to secured URL if SSL is enabled, otherwise just works fine. *Note: If someone did choose to use IIS instead, the ports should be different but the app should behave the same.*
  i. Open web browser and navigate to `https://localhost:5001/api/blogposts`
- Using Postman
  i. Navigate to Settings and turn off "SSL certificate verification"

  ii. Create new collection

  iii. Add two new requests and validate they work (Send & Save): `GET http://localhost:5000/api/blogposts` and `GET https://localhost:5001/api/blogposts`

Exercise set #1 - in-memory store & basic CRUD

- API versioning is important for consumers, thus change routing for our endpoint from `/api/blogposts` to `/api/v1/blogposts` . Validate using Postman that endpoints still work.
- Using Visual Studio
  i. Add new model BlogPost under Models folder (`./Models/BlogPost.cs`): right-click the project, select Add > New Folder and name it Models, then right-click the Models folder and select Add > Class (or New File when using VS Code). Name the class BlogPost and click Add. Update the model with the following properties/fields:

```
namespace Nip.Blog.Services.Posts.API.Models
{
        public class BlogPost
        {
                public long Id { get; set; }
                public string Title { get; set; }
                public string Description { get; set; }
        }
}
```

  ii. The easiest way to create an in-memory store is to create EntityFramework database context for BlogPost data model. Create `Data/BlogPostContext.cs` class that derive from `Microsoft.EntityFrameworkCore.DbContext` .

```
using Microsoft.EntityFrameworkCore;
using Nip.Blog.Services.Posts.API.Models;

namespace Nip.Blog.Services.Posts.API.Data
{
        public class BlogPostContext : DbContext
        {
                public BlogPostContext(DbContextOptions<BlogPostContext> options)
                        : base(options)
                {
                        // nop
                }

                public DbSet<BlogPost> BlogPosts { get; set; }
        }
}
```

iii. In ASP.NET Core dependencies are driven via built-in Dependency Injection (DI) Container. We have to register our DB context in the `IServiceCollection` service container. Update `Startup.cs` :

```
public void ConfigureServices(IServiceCollection services)
{
        services.AddDbContext<BlogPostContext>(opt => opt.UseInMemoryDatabase("BlogPosts"));
        ...
}
```

iv. Then we can use it in our BlogPostsController:

```
public class BlogPostsController : ControllerBase
{
        private readonly BlogPostContext _postsDbContext;

        public BlogPostsController(BlogPostContext postsDbContext)
        {
                _postsDbContext = postsDbContext;
        }
        ...
```

Note: The controller's constructor uses Dependency Injection to inject the database context (BlogPostsContext) into the controller.

v. Allow retrieval of all the blog posts. Update `BlogPostsController.cs`

```
// GET api/blogposts
[HttpGet]
public ActionResult<IEnumerable<BlogPost>> Get()
{
        return Ok(_postsDbContext.BlogPosts.ToList());
}
```

vi. Allow rerieval of specific blog post.

```
// GET api/blogposts/5
[HttpGet("{id}", Name = "GetBlogPost")]
public ActionResult<BlogPost> Get(long id)
{
        var item = _postsDbContext.BlogPosts.Find(id);
        if (item == null)
        {
                return NotFound();
        }
        else
        {
                return Ok(item);
        }
}
```

vii. Allow creation of a new blog post.

```
// POST api/blogposts
[HttpPost]
public IActionResult Post([FromBody] BlogPost post)
{
        _postsDbContext.BlogPosts.Add(post);
        _postsDbContext.SaveChanges();

        return CreatedAtRoute("GetBlogPost", new { id = post.Id }, post);
}
```

viii. Allow modification of an existing blog post.

```
// PUT api/blogposts/5
[HttpPut("{id}")]
public IActionResult Put(long id, [FromBody] BlogPost updatedPost)
{
        var post = _postsDbContext.BlogPosts.Find(id);
        if (post == null)
        {
                return NotFound();
        }
        else
        {
                post.Title = updatedPost.Title;
                post.Description = updatedPost.Description;

                _postsDbContext.BlogPosts.Update(post);
                _postsDbContext.SaveChanges();

                return NoContent();
        }
}
```

ix. Allow removal of a blog post

```
// DELETE api/blogposts/5
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
        var post = _postsDbContext.BlogPosts.Find(id);
        if (post == null)
        {
                return NotFound();
        }
        else
        {
                _postsDbContext.BlogPosts.Remove(post);
                _postsDbContext.SaveChanges();

                return NoContent();
        }
}
```

- Using Postman:
  i. Send and notice the 404 HTTP status code:
  `GET https://localhost:5001/api/v1/blogposts/923829`
  ii. Send and notice the 400 HTTP status code:
  `GET https://localhost:5001/api/v1/blogposts/' or 1=1`
  iii. Add "POST" request and validate it work (don't forget to hit Save!):
  `POST https://localhost:5001/api/v1/blogposts { "title": "New Post", "description": "Lorem ipsum..." }`
  iv. Notice the 201 status code and generated route for the new blog post (Headers -> Location). Validate the URI is accessible.

v. Add "PUT" request and validate it work:
`PUT https://localhost:5001/api/v1/blogposts/1 { "title": "New Post With Fixed Title", "description": "Lorem ipsum..." }`

vi. Add "DELETE" request and validate it work:
`DELETE https://localhost:5001/api/v1/blogposts/1`

vii. Try sending invalid models, like: empty one; with changed field names; with changed field value types, i.e: array [] or object {}.

Exercise set #2 - asynchronous IO access & asynchronous API

- *Forget for a moment that we currently have in-memory database...*
- Database context can be accessed asynchronously (`async` / `await`) since most of its operations are waiting for commends being executed on an actual database. They are usually not CPU intensitive, thus asynchronous operations allow the caller to do whatever he wants while waiting for the database query result.
- Usually there is a limited number of threads servicing requests. The benefit of using async over sync is that instead of blocking the thread while it is waiting for the database call to complete in sync implementation, the async will free the thread to handle more requests or assign it what ever process needs a thread. Once IO (database) call completes, another thread will take it from there and continue with the implementation. Async will also make your api run faster if your IO operations take longer to complete.
- ***One of prominent best practices in async programming is Async all the way i.e. you shouldn't mix synchronous and asynchronous code without carefully considering the consequences.***
- Using Visual Studio:
  i. Change all the calls to database context (`_postsDbContext`) to async representatives, for example:

```
public async Task<ActionResult<IEnumerable<BlogPost>>> Get()
{
        return Ok(await _postsDbContext.BlogPosts.ToAsyncEnumerable().ToList());
}

public async Task<IActionResult> Post([FromBody] BlogPost post)
{
        await _postsDbContext.BlogPosts.AddAsync(post);
        await _postsDbContext.SaveChangesAsync();
        return CreatedAtRoute("GetBlogPost", new { id = post.Id }, post);
}
```

  ii. All `GET/POST/PUT/DELETE` action methods should be `async` .
- Using Postman:
  i. Confirm nothing is broken by rerunning previous `POST/GET/PUT/DELETE` (preferably in that order) requests.

Exercise set #3 - document API

- One of the most widely used API specifications is OpenAPI. Swagger UI is an open source project to visually render documentation for an API defined with the OpenAPI (Swagger) Specification.
- The usual operation flow for usig Swagger looks like this:
  i. We register the Swagger generator which scans our APIs (controllers) and defines 1 or more Swagger documents.
  ii. Special middleware does serve generated Swagger as a JSON endpoint.
  iii. Special middleware does serve swagger-ui (HTML, JS, CSS, etc.) specifying the Swagger JSON endpoint.
- Using Visual Studio:
  i. Install `Swashbuckle.AspNetCore` nuget package by right-click on the project in Solution Explorer > Manage NuGet Packages
  ii. *(optional) you can avhieve the same via View > Other Windows > Package Manager Console and typing in* `Install-Package Swashbuckle.AspNetCore`
  iii. Register Swagger generator in the services collection (`Startup.cs > ConfigureServices()`)

```
services.AddSwaggerGen(c =>
{
```

```
        c.SwaggerDoc("v1", new Info { Title = "Blog Posts API", Version = "v1" });
});
```

iv. Enable the middleware for serving the generated JSON document and the Swagger UI (`Startup.cs > Configure()`)

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "Blog Posts API v1");
});
```

v. Run the server.

- Using browser:
  i. Navigate to API documentation in JSON format
  `https://localhost:5001/swagger/v1/swagger.json`
  ii. Navigate to Swagger UI documentation
  `https://localhost:5001/swagger/`
  iii. Since the latter one is interacive, you can send the same GET/POST/PUT/DELETE requests.
- Using Visual Studio:
  i. Extend SwaggerDoc(...) with more info like description, contact & license.
  ii. Add additional data annotations on the BlogPost model (`Models/BlogpPost.cs`) to inform about the constraints. Be careful because this does influence build-in model validation logic.

```
[Required]
[StringLength(32, MinimumLength = 3)]
[RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$", ErrorMessage = "Should start from capital letter and consist
public string Title { get; set; }

[StringLength(4096)]
public string Description { get; set; }
```

- Using browser:
  i. Confirm the documentation for the BlogPost model was updated *(bottom of the page)*
- One of the API requirements is to provide valid HTTP response codes and document them. We should at lest describe response types for each action.
- Use the HTTP Status Codes
  - **200 OK** - Response to a successful GET, PUT, PATCH or DELETE. Can also be used for a POST that doesn't result in a creation.
  - **201 Created** - Response to a POST that results in a creation. Should be combined with a Location header pointing to the location of the new resource.
  - **204 No Content** - Response to a successful request that won't be returning a body (like a DELETE request).
  - **400 Bad Request** - The request is malformed, such as if the body does not parse.
  - **401 Unauthorized** - When no or invalid authentication details are provided. Also useful to trigger an auth popup if the API is used from a browser.
  - **403 Forbidden** - When authentication succeeded but authenticated user doesn't have access to the resource.
  - **404 Not Found** - When a non-existent resource is requested.
- Using Visual Studio:
  i. Decorate/annotate each action method with appropriate `[ProducesResponseType]` attribute. Example:

```
[HttpPost]
[ProducesResponseType(201, Type = typeof(BlogPost))]
[ProducesResponseType(400)]
public async Task<IActionResult> Post([FromBody] BlogPost post)
{ ...
```

- Using browser:

i. Open Swagger UI and check if HTTP response codes are now listed.

Exercise set #4 - Global exception handler

- It is a good practice to always return consumable JSON error representation when building RESTful WebAPI.
- Using Visual Studio:
  i. Update `Startup.cs` and configure the HTTP request pipeline to redirect to `/api/v1/Error` controller whenever unhandled exception happens. This should work only if not in development mode, since in development we should get dev exception page with all sort of details (especially when queried with "?throw=true").

```
app.UseExceptionHandler("/api/Error");
```

  ii. Create new controller named `ErrorController`. Setup `Index()` method that return status 500 and display friendly message in JSON format. Hint:

```
{ "error": "Unhandled exception" }
```

```
 return StatusCode(..., new {...})
```

  iii. Fake one of the methods in `BlogPostsController` and throw exception from within, i.e:

```
public async Task<ActionResult<IEnumerable<BlogPost>>> Get()
{
        throw new BlogPostsDomainException("No posts atm");
}
```

  iv. Query it with Postman and confirm you get 500 status code with JSON body.
  v. *(optional) Try returning exception details and URI path from where it was issued. Google `HttpContext.Features.Get<IExceptionHandlerPathFeature>()` ...*
  vi. Remove the exception from faked method.

Exercise set #5 - Logging

- Few hints about logging
  - Never log any sensitive data (i.: usernames) at any level that goes to production. Trace/Debug level is less restrictive as long as it is only for development environment and does not involve "live" data.
  - **Trace** - debugging only, allow developer to track program execution, like begin/end of a method, bigger steps in algorithm, loops.
  - **Debug** - strictly for development & debugging purposes, like reading variable/model values, list sizes, consitions.
  - **Info** - usually contains information available in production so it is visible by OPS, used to track calls between "systems", like querying another api, processing a request, bigger steps in algorithms.
  - **Warn** - errors or unexpected behaviors which can be handled by application, like handled exceptions, invalid arguments.
  - **Error** - scenarios that are unrecoverable but can perform another work, like generic error handler.
  - **Critical/Fatal** - you should terminate right after this one... like no disk space, no space on the heap, lack of network connection.
- Using Visual Studio:
  i. Update `Program.cs` and append to the builder pipeline those lines:

```
.ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        });
```

ii. Using DI feed `Startup`, `BlogPostsController` and `ErrorController` classes with either `ILogger<T>` or `ILoggerFactory`.

iii. Apply appropriate logging to each of those classes with appropriate levels. Example:

```
public async Task<IActionResult> Put(long id, [FromBody] BlogPost updatedPost)
{
        _logger.LogInformation("Updating post {0}", id);
        _logger.LogDebug("Received post id {0} with new title: {1}'", id, updatedPost.Title);

        var post = await _postsDbContext.BlogPosts.FindAsync(id);
        if (post == null)
        {
                _logger.LogWarning("Post {0} not found", id);
                return NotFound();
        }
        ...
```

iv. *(optional) Try adding a provider that support writing logs to a file. Google* `Logging.AddFile(...);`

Exercise set #6 - persistent store & initialization

- ORM frameworks like EntityFramework Core not only allow you to effectively map database objects to models, but also easily swap database providers when needed, i.e: from SQLite > MsSQL > PostgreSQL > MySQL. It also handle database consistency and upgrades/downgrades (aka migrations).
- Using Visual Studio:
  i. Update `Startup.cs` and switch from in-memory database to MsSQL provider.

```
var connection = @"Server=
(localdb)\mssqllocaldb;Database=BlogPostsDb;Trusted_Connection=True;ConnectRetryCount=0";
services.AddDbContextPool<BlogPostContext>(options => options.UseSqlServer(connection));
```

  ii. Ensure that project builds.

  iii. We are going to use Code First workflow and generate database schema from our code (DBContexts and models). Add initial database migration and update local instance of MsSQL database (you should have it installed) accordingly.

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

  iv. Check the autogenerated migration files under `Migrations` folder. Those are the instructions how to change the databases with each version. There should be information like: tables and their columns, primary keys, indexes... That is being read by the database provider and translated to database instructions.

  v. Note! Later in the code, remember that whenever you update DBContext files, models or their relations then you should create new EF migrations.

  vi. Run the server and check if everything works. Now whenever you query the web API, you should be able to see generated SQL commands in the console window.

  vii. Add around 5 new posts using `POST https://localhost:5001/api/v1/blogposts`

  viii. Restart the server and confirm all the posts were stored in the databsae even though the server was stopped. Run `GET https://localhost:5001/api/v1/blogposts`

  ix. Now let us switch to SQLite. Update `Startup.cs` and switch from MsSQL provider database to SQLite provider.

```
var connection = @"Data Source=Data/Posts.db";
services.AddDbContextPool<BlogPostContext>(opt => opt.UseSqlite(connection))
```

  x. For this to work you need to install `Microsoft.EntityFrameworkCore.Sqlite` nuget package, either by installing via Nuget Package Manager or using command line: `dotnet add package Microsoft.EntityFrameworkCore.Sqlite`

  xi. Ensure that project builds.

  xii. We need to update selected SQLite database file with the migration instructions generated before.

```
dotnet ef database update
```

xiii. Run the server and check if everything works. Now whenever you query the web API, you should be able to see generated SQLLite commands in the console window.

xiv. Add X new posts using `POST https://localhost:5001/api/v1/blogposts`

xv. Download portable version of SQLiteStudio and open the `Data/Posts.db` file we used as persistent store. Confirm blog posts are in the `BlogPost` table.

xvi. When developing the web API, it is a good idea to initialize (aka seed) empty database with test data. Do so either in `Program.cs` (harder but recommended) or in `Startup.cs` (easier):

```
if (!env.IsDevelopment())
{
        context.Database.EnsureCreated();
        if(!context.BlogPosts.Any())
        {
                var posts = new List<BlogPost>
                {
                ...
                };
                context.BlogPosts.AddRange(posts);
                context.SaveChanges();
        }
}
```

xvii. Delete `Data/Posts.db` file and run the server. Database should be recreated using migration instructions and should also be populated with initial data. Run `GET https://localhost:5001/api/v1/blogposts` to confirm.

Exercise set #7 - repository pattern

- *For larger RESTful web API we could use `Controllers - Services - Repositories - Database` architecture, which provides maximum decoupling of application layers and makes it easy to develop and test the application. For the sake of this exercise we will skip `Services` layer.*
- Repository pattern adds fine abstraction level between the database and the controller. Repositories should encapsulate all the logic needed for accessing the database. If no `Services` layer is present, theb they should return collections as IEnumerable or IAsyncEnumerable (IQueryable is used otherwise).
- Using Visual Studio:
  i. Somewhere under `/Repositories` folder create `IBlogPostRepository` interface and `BlogPostRepository` class that implements it. The `IBlogPostRepository.cs` can look like this:

```
public interface IBlogPostRepository
{
        Task<BlogPost> GetAsync(long id);
        IAsyncEnumerable<BlogPost> GetAllAsync();
        Task AddAsync(BlogPost post);
        Task UpdateAsync(BlogPost post);
        Task DeleteAsync(long id);
}
```

  ii. Extract out from `BlogPostsController.cs` everything that is connected with accessing database under `BlogPostRepository.cs`, long story short: now the only class that should use `BlogPostContext` is `BlogPostRepository`

```
private readonly BlogPostContext _context;
public BlogPostRepository(BlogPostContext context) {
        _context = context;
}

public IAsyncEnumerable<T> GetAllAsync() {
        return _dbSet.ToAsyncEnumerable();
}
```

```
public async Task<T> GetAsync(long id) {
...
```

iii. The `BlogPostsController` should now expect `IBlogPostRepository` to be injected by the build-in Dependency Injection Container, hence its constructor should look like this:

```
private readonly ILogger<BlogPostsController> _logger;
private readonly IBlogPostRepository _postsRepo;
public BlogPostsController(ILogger<BlogPostsController> logger, IBlogPostRepository repo)
{
        _logger = logger;
        _postsRepo = repo;
}
```

iv. Remember that in order for the Dependency Injection Container to inject proper interface implementation to the constructors, you need to register them under `Startup > ConfigureServices`:

```
services.AddScoped<IBlogPostRepository, BlogPostRepository>();
```

v. Check if everything works now. Everything should behave the same as before. We soon will extend the repository with few more methods.

Updated requirements

- API v2

API	Description	Request body	Response body	HTTP status code
GET /api/v2/blogposts[?pageIndex={idx}&pageSize={size}]	Get blog posts page	-	{ one page of posts }	200 OK

- When requesting a page, there should be a link to the next page in the response body.

Exercise set #8 - new web API version - v2

- We talked about API versioning and how much we care about backward portability to make consumers/clients of our RESTful APIs happy.
- We are going to add v2 controller that will alter behaviour for retrieving collection of blog posts and we will use paging instead.
- Using Visual Studio:
  i. Before we proceed we need to install two nuget packages: `Microsoft.AspNetCore.Mvc.Versioning` and `Microsoft.AspNetCore.Mvc.Versioning.ApiExplorer` using either NuGet Package Manager or using cmd:

```
dotnet add package Microsoft.AspNetCore.Mvc.Versioning
dotnet add package Microsoft.AspNetCore.Mvc.Versioning.ApiExplorer
```

  ii. Update `Startup > ConfigureServices` and "teach" Swagger generator how to create separate documentation for each version of the controller endpoints:

```
services.AddMvcCore().AddVersionedApiExplorer(
        options => {
            options.GroupNameFormat = "'v'VVV";
            options.SubstituteApiVersionInUrl = true;
        });
        services.AddApiVersioning(options => options.ReportApiVersions = true);
services.AddSwaggerGen(
        options => {
            var provider = services.BuildServiceProvider().GetRequiredService<IApiVersionDescriptionProvider>(
            foreach (var description in provider.ApiVersionDescriptions)
            {
```

```
                options.SwaggerDoc(description.GroupName,  new Info{ ... Version = description.ApiVersion.ToSt
            }
        });
```

iii. Update `Startup > Configure` and tell Swagger UI where to find documentations:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, IApiVersionDescriptionProvider apiVers
        app.UseSwaggerUI(c => {
                foreach (var description in apiVersionDescProvider.ApiVersionDescriptions) {
                    c.SwaggerEndpoint($"/swagger/{description.GroupName}/swagger.json", description.GroupName
                }
                c.RoutePrefix = string.Empty; // serve the Swagger UI at the app's root
        });
...
```

iv. Modify `BlogPostsController` and replace `[Route("api/v1/[controller]")]` with this:

```
[ApiVersion("1")]
[Route("api/v{version:apiVersion}/BlogPosts")]
```

v. Build & run the server. Everything should work as before. Confirm Swagger UI generated same page under `https://localhost:5001/` .

vi. Clone `BlogPostsController.cs` and rename it to `BlogPostsV2Controller.cs` . Open it and change everything that was connected with v1 to v2. Examples:

- `[ApiVersion("2")]`
- `[HttpGet("{id}", Name = "GetBlogPostV2")]`
- `return CreatedAtRoute("GetBlogPostV2", new { id = post.Id }, post);`

vii. Build & run the server. Everything should work as before with exception that in Swagger UI there should be a dropdown at the top right corner where you can see specification for either V1 or V2 of the API.

- Using Postman:
  i. Duplicate collection of all your previous requests and rename it to `<collection> v2`
  ii. Update all the hyperlinks under that collection to target v2 version of the API now.
  iii. Confirm `GET POST PUT DELETE` requests work and that `POST` redirects you to the newly created resource under v2 endpoint, i.e: `Location: https://localhost:5001/api/v2/BlogPosts/6`

Exercise set #8 - paging

- From now on we will focus on v2 version of the controller.
- Using Visual Studio:
  i. Add new model called `PaginatedItems` :

```
public class PaginatedItems<T>
{
        public int PageIndex { get; set; }
        public int PageSize { get; set; }
        public long TotalItems { get; set; }
        public IEnumerable<T> Items { get; set; }
        public string NextPage { get; set; }
}
```

ii. Update `Get()` method in `BlogPostsV2Controller` to handle pagination when requested. You should be able to fill the gaps in the code:

```
// GET api/v2/blogposts[?pageIndex=3&pageSize=10]
[HttpGet]
[ProducesResponseType(400)]
[ProducesResponseType(200, Type = typeof(IEnumerable<BlogPost>))]
[ProducesResponseType(200, Type = typeof(PaginatedItems<BlogPost>))]
```

```
public async Task<IActionResult> Get([FromQuery]int pageIndex = -1, [FromQuery]int pageSize = 5)
{
        var posts = await _postsRepo.GetAllAsync().ToList();
        if (pageIndex < 0) {
                return Ok(posts);
        } else {
                ...
                var pagedPosts = new PaginatedItems<BlogPost>{ ... };
                return Ok(pagedPosts);
        }
}
```

iii. Chunking the list of posts using LINQ may look like this:

```
pagedPosts.Items = posts.OrderByDescending(c => c.Id).Skip(pageIndex * pageSize).Take(pageSize),
```

iv. Generating next URL should look like this:

```
pagedPosts.NextPage = (!isLastPage ? Url.Link(null, new { pageIndex = pageIndex + 1, pageSize = pageSize }) :
```

v. Build and run the server

- Using Postman:
  i. Query `GET https://localhost:5001/api/v2/blogposts` and confirm all the posts are returned.
  ii. Query `GET https://localhost:5001/api/v2/blogposts?pageIndex=0&pageSize=5` and confirm that max 5 records are returned.
  iii. If your test data has only few posts, just add more.
  iv. Confirm valid `NextPage` URI is generated when there are more pages available. Should be null if not.

```
...
"NextPage": "https://localhost:5001/api/v2/blogposts?pageIndex=1&pageSize=5"
```

  v. Confirm that query `GET https://localhost:5001/api/v2/blogposts?pageIndex=9000&pageSize=9000` returns empty set.
- Using Visual Studio:
  i. Extract out the paging logic to the `BlogPostsRepository`. The new `IBlogPostsRepository` method can look like this:

```
Task<PaginatedItems<BlogPost>> GetAllPagedAsync(int pageIndex, int pageSize);
```

  ii. Make use of the fact that repository is working on IQueryable instead of IEnumerable. You can get post count as a separate query:

```
var totalItems = await _context.BlogPosts.CountAsync();
var posts = await _context.BlogPosts.OrderByDescending(c => c.Id).Skip(pageIndex * pageSize).Take(pageSize).To
```

  iii. Build and run the server
  iv. Request few pages using `GET https://localhost:5001/api/v2/blogposts?pageIndex=0&pageSize=5` and confirm nothing changed.

Updated requirements

- API v2 - another endpoint

API	Description	Request body	Response body	HTTP status code
GET /api/v2/blogposts/withtitle/{title}[?pageIndex={idx}&pageSize={size}]	Get page of blog posts filtered by title	-	{ one page of filtered posts }	200 OK

Exercise set #9 - filtering

- Using Visual Studio:
  i. Update `GetAllPagedAsync()` method in the IBlogPostsRepository interface. It can now look like this:

```
Task<PaginatedItems<BlogPost>> GetAllPagedAsync(int pageIndex, int pageSize, Expression<Func<BlogPost, bool>>
```

  ii. The logic for paging can be updated to include only results that fulfil filter conditions:

```
...
IQueryable<BlogPost> query = _context.BlogPosts;
if (filter != null)
        query = query.Where(filter);
var totalItems = await query.CountAsync();
var posts = query.OrderByDescending(c => c.Id).Skip(pageIndex * pageSize).Take(pageSize);
...
```

  iii. **Add** new `Get(...)` method to the `BlogPostsV2Controller` that will handle filtering:

```
[HttpGet("withtitle/{title:minlength(1)}")]
[ProducesResponseType(400)]
[ProducesResponseType(200, Type = typeof(PaginatedItems<BlogPost>))]
public async Task<IActionResult> Get(string title, [FromQuery]int pageIndex = 0, [FromQuery]int pageSize = 5)
        ...
        var pagedPosts = await _postsRepo.GetAllPagedAsync(pageIndex, pageSize, x => x.Title.Contains(title));
        ...
```

  iv. Build and run the server

- Using Postman:
  i. Request posts that contain some sequence of characters in the title. Example:

```
GET https://localhost:5001/api/v2/blogposts/withtitle/derp
```

  ii. Do the same request but with paging (add more posts that fit the condition if there are not too many results)

```
GET https://localhost:5001/api/v2/blogposts/withtitle/derp?pageIndex=0&pageSize=2
```

  iii. Confirm sending empty title filter does result in `400 bad request` status code.

Updated requirements

- API v2:

API	Description	Request body	Response body	HTTP status code
GET /api/v1/blogposts/{id}/comments	Get comments for blog post	-	{ array of comments }	200 OK
POST /api/v1/blogposts/{id}/comments	Add a new blog post comment	{ body }	{ comment }	201 Created

Exercise set #10 - add blog post comments

- Using Visual Studio:
  i. Update `BlogPost` model with another property representing collection of blog post comments:

```
public ICollection<BlogPostComment> Comments { get; set; }
```

ii. Add new `BlogPostComment` model that looks like this:

```
public long Id { get; set; }

[Required]
        [StringLength(24, MinimumLength = 3)]
public string Author { get; set; }

[StringLength(256)]
public string Content { get; set; }
```

iii. Add new Entity Framework migration in order to generate code responsible for upgrading the database and then update the database.

```
dotnet ef migrations add AddComments
dotnet ef database update
```

iv. Update `IBlogPostsRepository` and its implementation with two more methods:

```
Task<IEnumerable<BlogPostComment>> GetCommentsAsync(long blogPostId);
Task AddCommentAsync(long blogPostId, BlogPostComment comment);
```

v. When querying for blog post and all its comments, you need to explicitly tell EF Core to include those comments:

```
var post = await _context.BlogPosts.Include(x => x.Comments).Where( x => x.Id == blogPostId).FirstAsync();
```

vi. Add two more methods to the `BlogPostsV2Controller`, one for querying for all the comments from specific blog post and one for submitting new comment.

```
[...]
[HttpGet("{id}/comments", Name = "GetBlogPostComments")]
public async Task<ActionResult<IEnumerable<BlogPostComment>>> GetAllComments(long id) { ... }

[...]
[HttpPost("{id}/comments")]
public async Task<IActionResult> PostComment(long id, [FromBody] BlogPostComment comment){ ... }
```

vii. Build and run the server

- Using Postman:
  i. Submit few comments using `POST https://localhost:5001/api/v2/blogposts/0/comments`. Check if it does redirect you to a collection of comments for that specific blog post.
  ii. Query for all the comments for specific blog post: `GET https://localhost:5001/api/v2/blogposts/0/comments`
  iii. Query for comments from nonexisting blog post and confirm you get `404 status code`.

Exercise set #11 - add concurrency token, aka row-version

- A row-version is a property where a new value is generated by the database every time a row is inserted or updated. The property is also treated as a concurrency token. This ensures you will get an exception if anyone else has modified a row that you are trying to update since you queried for the data.
- Using Visual Studio:
  i. Update `BlogPost` model with row-version property:

```
[Timestamp]
[IgnoreDataMember]
public byte[] RowVersion { get; set; }
```

ii. Add new migration and update database
iii. Note: currently SQLite provider for EF Core does not support row-version and concurrency tokens. It order for it to work you need to switch to enterprise database like MsSQL. Only then you will see that with each update of any blog post the `RowVersion` column gets incremented.

iv. When race condition happens and one of the requests will try to modify row in outdated version, an `DbUpdateConcurrencyException` will be thrown. You may want to handle it in `ErrorController` and return 503 that tells the client to simply retry its request.

Exercise set #12 - add configuration file

- By default ASP.NET Core provides a configuration file that we can use for storing our own settings. Its named `appsettings.json` and it can differ a little for each environment, i.e: `appsettings.Development.json` will have different log levels set when debugging.
- Using Visual Studio:

i. We will add one more config file. Create `dbsettings.json` with such content:

```
{
        "SelectedDbType" :  "SQLite",
        "ConnectionStrings": {
                "MsSQLBlogPostsDatabase": "Server=(localdb)\\mssqllocaldb;Database=BlogPostsDb;Trusted_Connect
                "SQLiteBlogPostsDatabase": "Data Source=Data/Posts.db"
        }
}
```

ii. We will use it to configure or app to either use SQLite or MsSQL as persistent store.

iii. Update `Program.cs` and add to the web host builder those lines:

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    config.SetBasePath(Directory.GetCurrentDirectory());
    config.AddJsonFile("dbsettings.json", optional: false, reloadOnChange: true);
})
```

iv. Update `Startup > ConfigureServices` and this time read config and then decide if we are going to use SQLite or MsSQL.

```
var dbType = Configuration.GetValue<string>("SelectedDbType");
if(...){
        var connection = Configuration.GetConnectionString("MsSQLBlogPostsDatabase")
} else {
...
```

v. Build and run the server with one configuration. Confirm by looking at the console logs if correct database type was chosen.

vi. Do the same for the second database type.