16 PAGING

We have covered a lot of interesting features while creating our Web API project, but there are still things to do.

So, in this chapter, we're going to learn how to implement paging in ASP.NET Core Web API. It is one of the most important concepts in building RESTful APIs.

If we inspect the `GetEmployeesForCompany` action in the `EmployeesController`, we can see that we return all the employees for the single company.

But we don't want to return a collection of all resources when querying our API. That can cause performance issues and it's in no way optimized for public or private APIs. It can cause massive slowdowns and even application crashes in severe cases.

Of course, we should learn a little more about Paging before we dive into code implementation.

16.1 What is Paging?

Paging refers to **getting partial results from an API**. Imagine having millions of results in the database and having your application try to return all of them at once.

Not only would that be an **extremely ineffective** way of returning the results, but it could also possibly have **devastating effects on the application itself or the hardware it runs on**. Moreover, every client has limited memory resources and it needs to restrict the number of shown results.

Thus, we need a way to return a set number of results to the client in order to avoid these consequences. Let's see how we can do that.

16.2 Paging Implementation

Mind you, we don’t want to change the base repository logic or implement any business logic in the controller.

What we want to achieve is something like this: `https://localhost:5001/api/companies/companyId/employees?pageNumber=2&pageSize=2`. This should return the second set of two employees we have in our database.

We also want to constrain our API not to return all the employees even if someone calls `https://localhost:5001/api/companies/companyId/employees`.

Let's start with the controller modification by modifying the `GetEmployeesForCompany` action:

```
[HttpGet]
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId,
       [FromQuery] EmployeeParameters employeeParameters)
{
       var employees = await _service.EmployeeService.GetEmployeesAsync(companyId,
trackChanges: false);
       return Ok(employees);
}
```

A few things to take note of here:

- We’re using `[FromQuery]` to point out that we’ll be using query parameters to define which page and how many employees we are requesting.
- The `EmployeeParameters` class is the container for the actual parameters for the Employee entity.

We also need to actually create the `EmployeeParameters` class. So, let’s first create a `RequestFeatures` folder in the `Shared` project and then inside, create the required classes.

First the `RequestParameters` class:

```
public abstract class RequestParameters
```

```
{
    const int maxPageSize = 50;
    public int PageNumber { get; set; } = 1;

    private int _pageSize = 10;
    public int PageSize
    {
        get
        {
            return _pageSize;
        }
        set
        {
            _pageSize = (value > maxPageSize) ? maxPageSize : value;
        }
    }
```

And then the `EmployeeParameters` class:

```
public class EmployeeParameters : RequestParameters
{
}
```

We create an abstract class to hold the common properties for all the entities in our project, and a single `EmployeeParameters` class that will hold the specific parameters. It is empty now, but soon it won't be.

In the abstract class, we are using the `maxPageSize` constant to restrict our API to a maximum of 50 rows per page. We have two public properties – `PageNumber` and `PageSize`. If not set by the caller, `PageNumber` will be set to 1, and `PageSize` to 10.

Now we can return to the controller and import a using directive for the `EmployeeParameters` class:

```
using Shared.RequestFeatures;
```

After that change, let's implement the most important part — the repository logic. We need to modify the `GetEmployeesAsync` method in the `IEmployeeRepository` interface and the `EmployeeRepository` class.

So, first the interface modification:

```
public interface IEmployeeRepository
{
      Task<IEnumerable<Employee>> GetEmployeesAsync(Guid companyId,
             EmployeeParameters employeeParameters, bool trackChanges);
      Task<Employee> GetEmployeeAsync(Guid companyId, Guid id, bool trackChanges);
      void CreateEmployeeForCompany(Guid companyId, Employee employee);
      void DeleteEmployee(Employee employee);
}
```

As Visual Studio suggests, we have to add the reference to the **Shared** project.

After that, let's modify the repository logic:

```
public async Task<IEnumerable<Employee>> GetEmployeesAsync(Guid companyId,
    EmployeeParameters employeeParameters, bool trackChanges) =>
    await FindByCondition(e => e.CompanyId.Equals(companyId), trackChanges)
    .OrderBy(e => e.Name)
    .Skip((employeeParameters.PageNumber - 1) * employeeParameters.PageSize)
    .Take(employeeParameters.PageSize)
    .ToListAsync();
```

Okay, the easiest way to explain this is by example.

Say we need to get the results for the third page of our website, counting 20 as the number of results we want. That would mean we want to skip the first ((3 – 1) * 20) = 40 results, then take the next 20 and return them to the caller.

Does that make sense?

Since we call this repository method in our service layer, we have to modify it as well.

So, let's start with the **IEmployeeService** modification:

```
public interface IEmployeeService
{
       Task<IEnumerable<EmployeeDto>> GetEmployeesAsync(Guid companyId,
              EmployeeParameters employeeParameters, bool trackChanges);
       ...
}
```

In this interface, we only have to modify the **GetEmployeesAsync** method by adding a new parameter.

After that, let's modify the **EmployeeService** class:

```
public async Task<IEnumerable<EmployeeDto>> GetEmployeesAsync(Guid companyId,
       EmployeeParameters employeeParameters, bool trackChanges)
{
       await CheckIfCompanyExists(companyId, trackChanges);

       var employeesFromDb = await _repository.Employee
              .GetEmployeesAsync(companyId, employeeParameters, trackChanges);
       var employeesDto = _mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);

       return employeesDto;
}
```

Nothing too complicated here. We just accept an additional parameter and pass it to the repository method.

Finally, we have to modify the **`GetEmployeesForCompany`** action and fix that error by adding another argument to the **`GetEmployeesAsync`** method call:

```
[HttpGet]
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId,
        [FromQuery] EmployeeParameters employeeParameters)
{
        var employees = await _service.EmployeeService.GetEmployeesAsync(companyId,
                employeeParameters, trackChanges: false);

        return Ok(employees);
}
```

16.3 Concrete Query

Before we continue, we should create additional employees for the company with the id: **`C9D4C053-49B6-410C-BC78-2D54A9991870`**. We are doing this because we have only a small number of employees per company and we need more of them for our example. You can use a predefined request in Part16 in Postman, and just change the request body with the following objects:

{
    "name": "Mihael Worth",
    "age": 30,
    "position": "Marketing expert"
}
	{
	    "name": "John Spike",
	    "age": 32,
	    "position": "Marketing expert II"
	}
		{
		    "name": "Nina Hawk",
		    "age": 26,
		    "position": "Marketing expert II"
		}

{
    "name": "Mihael Fins",
    "age": 30,
    "position": "Marketing expert"
	{
	    "name": "Martha Grown",
	    "age": 35,
		{
		    "name": "Kirk Metha",
		    "age": 30,
		    "position": "Marketing expert"

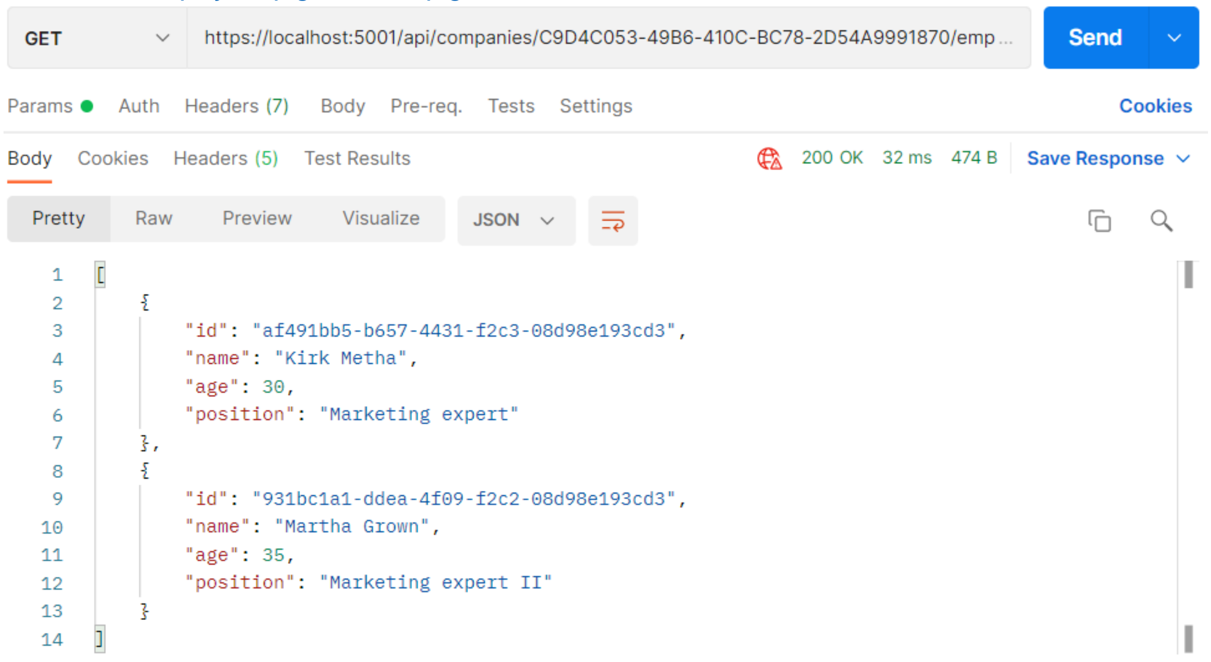
}	"position": "Marketing expert II" }	}

Now we should have eight employees for this company, and we can try a request like this:

https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?pageNumber=2&pageSize=2

So, we request page two with two employees:

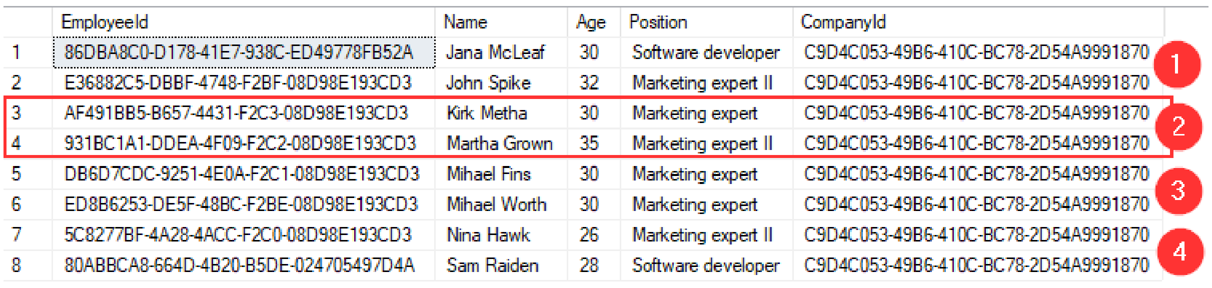
https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?pageNumber=2&pageSize=2



```
[
    {
        "id": "af491bb5-b657-4431-f2c3-08d98e193cd3",
        "name": "Kirk Metha",
        "age": 30,
        "position": "Marketing expert"
    },
    {
        "id": "931bc1a1-ddea-4f09-f2c2-08d98e193cd3",
        "name": "Martha Grown",
        "age": 35,
        "position": "Marketing expert II"
    }
]
```

If that's what you got, you're on the right track.

We can check our result in the database:



	EmployeeId	Name	Age	Position	CompanyId
1	86DBA8C0-D178-41E7-938C-ED49778FB52A	Jana McLeaf	30	Software developer	C9D4C053-49B6-410C-BC78-2D54A9991870
2	E36882C5-DBBF-4748-F2BF-08D98E193CD3	John Spike	32	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
3	AF491BB5-B657-4431-F2C3-08D98E193CD3	Kirk Metha	30	Marketing expert	C9D4C053-49B6-410C-BC78-2D54A9991870
4	931BC1A1-DDEA-4F09-F2C2-08D98E193CD3	Martha Grown	35	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
5	DB6D7CDC-9251-4E0A-F2C1-08D98E193CD3	Mihael Fins	30	Marketing expert	C9D4C053-49B6-410C-BC78-2D54A9991870
6	ED8B6253-DE5F-48BC-F2BE-08D98E193CD3	Mihael Worth	30	Marketing expert	C9D4C053-49B6-410C-BC78-2D54A9991870
7	5C8277BF-4A28-4ACC-F2C0-08D98E193CD3	Nina Hawk	26	Marketing expert II	C9D4C053-49B6-410C-BC78-2D54A9991870
8	80ABBCA8-664D-4B20-B5DE-024705497D4A	Sam Raiden	28	Software developer	C9D4C053-49B6-410C-BC78-2D54A9991870

And we can see that we have the correct data returned.

Now, what can we do to improve this solution?

16.4 Improving the Solution

Since we're returning just a subset of results to the caller, we might as well have a `PagedList` instead of `List`.

`PagedList` will inherit from the `List` class and will add some more to it. We can also move the skip/take logic to the `PagedList` since it makes more sense.

So, let's first create a new `MetaData` class in the `Shared/RequestFeatures` folder:

```
public class MetaData
{
    public int CurrentPage { get; set; }
    public int TotalPages { get; set; }
    public int PageSize { get; set; }
    public int TotalCount { get; set; }

    public bool HasPrevious => CurrentPage > 1;
    public bool HasNext => CurrentPage < TotalPages;
}
```

Then, we are going to implement the `PagedList` class in the same folder:

```
public class PagedList<T> : List<T>
{
    public MetaData MetaData { get; set; }

    public PagedList(List<T> items, int count, int pageNumber, int pageSize)
    {
        MetaData = new MetaData
        {
            TotalCount = count,
            PageSize = pageSize,
            CurrentPage = pageNumber,
            TotalPages = (int)Math.Ceiling(count / (double)pageSize)
        };

        AddRange(items);
    }

    public static PagedList<T> ToPagedList(IEnumerable<T> source, int pageNumber, int
pageSize)
    {
        var count = source.Count();
        var items = source
```

```
            .Skip((pageNumber - 1) * pageSize)
            .Take(pageSize).ToList();

        return new PagedList<T>(items, count, pageNumber, pageSize);
    }
}
```

As you can see, we've transferred the skip/take logic to the static method inside of the `PagedList` class. And in the `MetaData` class, we've added a few more properties that will come in handy as metadata for our response.

`HasPrevious` is true if the `CurrentPage` is larger than 1, and `HasNext` is calculated if the `CurrentPage` is smaller than the number of total pages. `TotalPages` is calculated by dividing the number of items by the page size and then rounding it to the larger number since a page needs to exist even if there is only one item on it.

Now that we've cleared that up, let's change our `EmployeeRepository` and `EmployeesController` accordingly.

Let's start with the interface modification:

```
Task<PagedList<Employee>> GetEmployeesAsync(Guid companyId,
       EmployeeParameters employeeParameters, bool trackChanges);
```

Then, let's change the repository class:

```
public async Task<PagedList<Employee>> GetEmployeesAsync(Guid companyId,
    EmployeeParameters employeeParameters, bool trackChanges)
{
    var employees = await FindByCondition(e => e.CompanyId.Equals(companyId),
trackChanges)
        .OrderBy(e => e.Name)
        .ToListAsync();

    return PagedList<Employee>
        .ToPagedList(employees, employeeParameters.PageNumber,
employeeParameters.PageSize);
}
```

After that, we are going to modify the `IEmplyeeService` interface:

```
Task<(IEnumerable<EmployeeDto> employees, MetaData metaData)> GetEmployeesAsync(Guid
companyId, EmployeeParameters employeeParameters, bool trackChanges);
```

Now our method returns a Tuple containing two fields – employees and metadata.