20 Data Shaping

In this chapter, we are going to talk about a neat concept called data shaping and how to implement it in ASP.NET Core Web API. To achieve that, we are going to use similar tools to the previous section. Data shaping is not something that every API needs, but it can be very useful in some cases.

Let's start by learning what data shaping is exactly.

20.1 What is Data Shaping?

Data shaping is a great way to reduce the amount of traffic sent from the API to the client. It **enables the consumer of the API to select (shape) the data by choosing the fields through the query string**.

What this means is something like:

```
https://localhost:5001/api/companies/companyId/employees?fields=name,age
```

By giving the consumer a way to select just the fields it needs, we can potentially **reduce the stress on the API**. On the other hand, **this is not something every API needs**, so we need to think carefully and decide whether we should implement its implementation because it has a bit of reflection in it.

And we know for a fact that reflection takes its toll and slows our application down.

Finally, as always, data shaping should work well together with the concepts we've covered so far – paging, filtering, searching, and sorting.

First, we are going to implement an employee-specific solution to data shaping. Then we are going to make it more generic, so it can be used by any entity or any API.

Let's get to work.

20.2 How to Implement Data Shaping

First things first, we need to extend our **RequestParameters** class since we are going to add a new feature to our query string and we want it to be available for any entity:

```
public string? Fields { get; set; }
```

We've added the **Fields** property and now we can use fields as a query string parameter.

Let's continue by creating a new interface in the **Contracts** project:

```
public interface IDataShaper<T>
{
       IEnumerable<ExpandoObject> ShapeData(IEnumerable<T> entities, string
fieldsString);
       ExpandoObject ShapeData(T entity, string fieldsString);
}
```

The **IDataShaper** defines two methods that should be implemented — one for the single entity and one for the collection of entities. Both are named **ShapeData**, but they have different signatures.

Notice how we use the **ExpandoObject** from **System.Dynamic** namespace as a return type. We need to do that to shape our data the way we want it.

To implement this interface, we are going to create a new **DataShaping** folder in the **Service** project and add a new **DataShaper** class:

```
public class DataShaper<T> : IDataShaper<T> where T : class
{
    public PropertyInfo[] Properties { get; set; }

    public DataShaper()
    {
        Properties = typeof(T).GetProperties(BindingFlags.Public |
BindingFlags.Instance);
    }

    public IEnumerable<ExpandoObject> ShapeData(IEnumerable<T> entities, string
fieldsString)
    {
```

```
            var requiredProperties = GetRequiredProperties(fieldsString);

            return FetchData(entities, requiredProperties);
        }

        public ExpandoObject ShapeData(T entity, string fieldsString)
        {
            var requiredProperties = GetRequiredProperties(fieldsString);

            return FetchDataForEntity(entity, requiredProperties);
        }

        private IEnumerable<PropertyInfo> GetRequiredProperties(string fieldsString)
        {
            var requiredProperties = new List<PropertyInfo>();

            if (!string.IsNullOrWhiteSpace(fieldsString))
            {
                var fields = fieldsString.Split(',',
StringSplitOptions.RemoveEmptyEntries);

                foreach (var field in fields)
                {
                    var property = Properties
                        .FirstOrDefault(pi => pi.Name.Equals(field.Trim(),
StringComparison.InvariantCultureIgnoreCase));

                    if (property == null)
                        continue;

                    requiredProperties.Add(property);
                }
            }
            else
            {
                requiredProperties = Properties.ToList();
            }

            return requiredProperties;
        }

        private IEnumerable<ExpandoObject> FetchData(IEnumerable<T> entities,
IEnumerable<PropertyInfo> requiredProperties)
        {
            var shapedData = new List<ExpandoObject>();

            foreach (var entity in entities)
            {
                var shapedObject = FetchDataForEntity(entity, requiredProperties);
                shapedData.Add(shapedObject);
            }

            return shapedData;
        }

        private ExpandoObject FetchDataForEntity(T entity, IEnumerable<PropertyInfo>
requiredProperties)
        {
            var shapedObject = new ExpandoObject();
```

```
        foreach (var property in requiredProperties)
        {
            var objectPropertyValue = property.GetValue(entity);
            shapedObject.TryAdd(property.Name, objectPropertyValue);
        }

        return shapedObject;
    }
}
```

We need these namespaces to be included as well:

```
using Contracts;
using System.Dynamic;
using System.Reflection;
```

There is quite a lot of code in our class, so let's break it down.

20.3 Step-by-Step Implementation

We have one public property in this class – **Properties**. It's an array of PropertyInfo's that we're going to pull out of the input type, whatever it is — Company or Employee in our case:

```
public PropertyInfo[] Properties { get; set; }

public DataShaper()
{
    Properties = typeof(T).GetProperties(BindingFlags.Public | BindingFlags.Instance);
}
```

So, here it is. In the constructor, we get all the properties of an input class.

Next, we have the implementation of our two public **ShapeData** methods:

```
public IEnumerable<ExpandoObject> ShapeData(IEnumerable<T> entities, string
fieldsString)
{
    var requiredProperties = GetRequiredProperties(fieldsString);

    return FetchData(entities, requiredProperties);
}

public ExpandoObject ShapeData(T entity, string fieldsString)
{
    var requiredProperties = GetRequiredProperties(fieldsString);

    return FetchDataForEntity(entity, requiredProperties);
}
```

Both methods rely on the `GetRequiredProperties` method to parse the input string that contains the fields we want to fetch.

The `GetRequiredProperties` method does the magic. It parses the input string and returns just the properties we need to return to the controller:

```
private IEnumerable<PropertyInfo> GetRequiredProperties(string fieldsString)
{
    var requiredProperties = new List<PropertyInfo>();

    if (!string.IsNullOrWhiteSpace(fieldsString))
    {
        var fields = fieldsString.Split(',', StringSplitOptions.RemoveEmptyEntries);

        foreach (var field in fields)
        {
            var property = Properties
              .FirstOrDefault(pi => pi.Name.Equals(field.Trim(),
StringComparison.InvariantCultureIgnoreCase));

            if (property == null)
                continue;

            requiredProperties.Add(property);
        }
    }
    else
    {
        requiredProperties = Properties.ToList();
    }

    return requiredProperties;
}
```

There's nothing special about it. If the `fieldsString` is not empty, we split it and check if the fields match the properties in our entity. If they do, we add them to the list of required properties.

On the other hand, if the `fieldsString` is empty, all properties are required.

Now, `FetchData` and `FetchDataForEntity` are the private methods to extract the values from these required properties we've prepared.

The `FetchDataForEntity` method does it for a single entity:

```
private ExpandoObject FetchDataForEntity(T entity, IEnumerable<PropertyInfo>
requiredProperties)
{
    var shapedObject = new ExpandoObject();
```

```
    foreach (var property in requiredProperties)
    {
        var objectPropertyValue = property.GetValue(entity);
        shapedObject.TryAdd(property.Name, objectPropertyValue);
    }

    return shapedObject;

}
```

Here, we loop through the `requiredProperties` parameter. Then, using a bit of reflection, we extract the values and add them to our `ExpandoObject`. ExpandoObject implements `IDictionary<string,object>`, so we can use the `TryAdd` method to add our property using its name as a key and the value as a value for the dictionary.

This way, we dynamically add just the properties we need to our dynamic object.

The `FetchData` method is just an implementation for multiple objects. It utilizes the `FetchDataForEntity` method we've just implemented:

```
private IEnumerable<ExpandoObject> FetchData(IEnumerable<T> entities,
IEnumerable<PropertyInfo> requiredProperties)
{
    var shapedData = new List<ExpandoObject>();

    foreach (var entity in entities)
    {
        var shapedObject = FetchDataForEntity(entity, requiredProperties);
        shapedData.Add(shapedObject);
    }

    return shapedData;
}
```

To continue, let's register the `DataShaper` class in the `IServiceCollection` in the `Program` class:

```
builder.Services.AddScoped<IDataShaper<EmployeeDto>, DataShaper<EmployeeDto>>();
```

During the service registration, we provide the type to work with.

Because we want to use the `DataShaper` class inside the service classes, we have to modify the constructor of the `ServiceManager` class first:

```
public ServiceManager(IRepositoryManager repositoryManager, ILoggerManager logger,
       IMapper mapper, IDataShaper<EmployeeDto> dataShaper)
{
       _companyService = new Lazy<ICompanyService>(() =>
              new  CompanyService(repositoryManager, logger, mapper));
       _employeeService = new Lazy<IEmployeeService>(() =>
              new EmployeeService(repositoryManager, logger, mapper, dataShaper));
}
```

We are going to use it only in the **EmployeeService** class.

Next, let's add one more field and modify the constructor in the **EmployeeService** class:

```
...
private readonly IDataShaper<EmployeeDto> _dataShaper;

public EmployeeService(IRepositoryManager repository, ILoggerManager logger,
       IMapper mapper, IDataShaper<EmployeeDto> dataShaper)
{
       _repository = repository;
       _logger = logger;
       _mapper = mapper;
       _dataShaper = dataShaper;
}
```

Let's also modify the **GetEmployeesAsync** method of the same class:

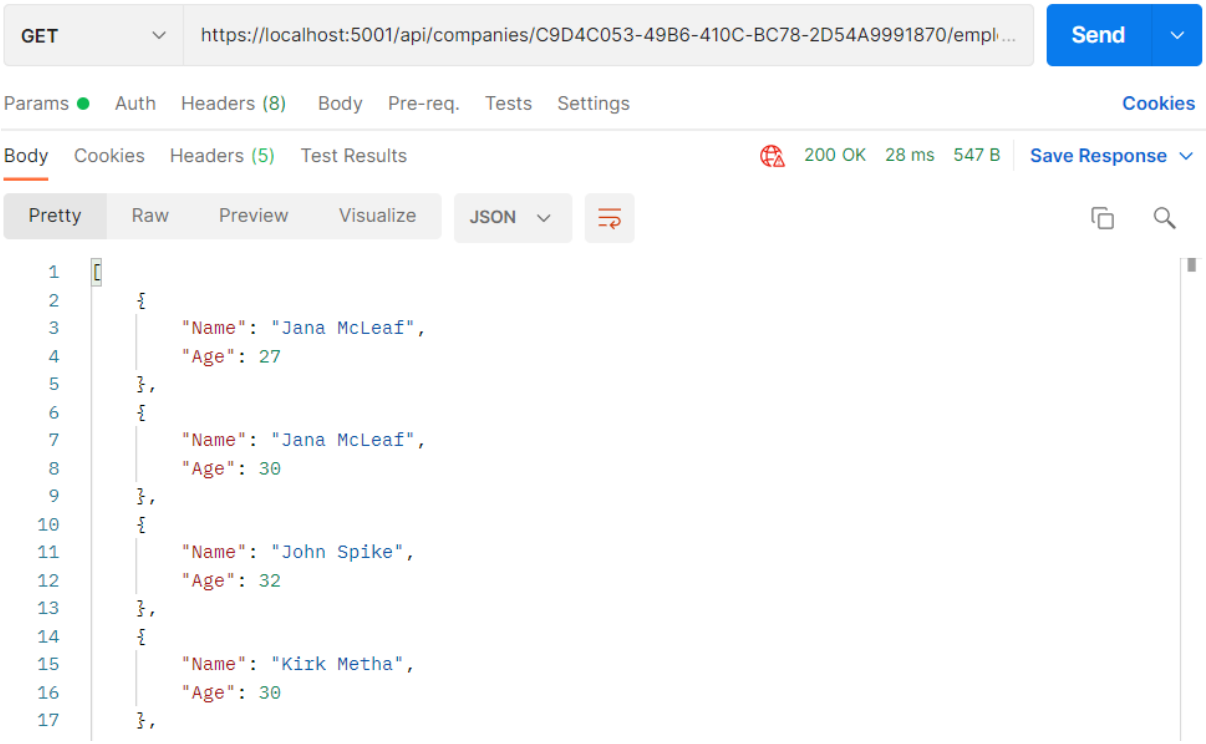
```
public async Task<(IEnumerable<ExpandoObject> employees, MetaData metaData)>
GetEmployeesAsync
       (Guid companyId, EmployeeParameters employeeParameters, bool trackChanges)
{
       if (!employeeParameters.ValidAgeRange)
              throw new MaxAgeRangeBadRequestException();

       await CheckIfCompanyExists(companyId, trackChanges);

       var employeesWithMetaData = await _repository.Employee
              .GetEmployeesAsync(companyId, employeeParameters, trackChanges);

       var employeesDto =
_mapper.Map<IEnumerable<EmployeeDto>>(employeesWithMetaData);
       var shapedData = _dataShaper.ShapeData(employeesDto,
employeeParameters.Fields);

       return (employees: shapedData, metaData: employeesWithMetaData.MetaData);
}
```

We have changed the method signature so, we have to modify the interface as well:

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

Now, we can test our solution:

https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?fields=name,age



It works great.

Let’s also test this solution by combining all the functionalities that we’ve implemented in the previous chapters:

https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?pageNumber=1&pageSize=4&minAge=26&maxAge=32&searchTerm=A&orderBy=name desc&fields=name,age

Excellent. Everything is working like a charm.

20.4 Resolving XML Serialization Problems

Let's send the same request one more time, but this time with the different accept header (text/xml):

```
<ArrayOfArrayOfKeyValueOfstringanyType xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://
    schemas.microsoft.com/2003/10/Serialization/Arrays">
    <ArrayOfKeyValueOfstringanyType>
        <KeyValueOfstringanyType>
            <Key>Name</Key>
            <Value xmlns:d4p1="http://www.w3.org/2001/XMLSchema" i:type="d4p1:string">Sam Raiden</Value>
        </KeyValueOfstringanyType>
        <KeyValueOfstringanyType>
            <Key>Age</Key>
            <Value xmlns:d4p1="http://www.w3.org/2001/XMLSchema" i:type="d4p1:int">28</Value>
        </KeyValueOfstringanyType>
    </ArrayOfKeyValueOfstringanyType>
    <ArrayOfKeyValueOfstringanyType>
        <KeyValueOfstringanyType>
            <Key>Name</Key>
            <Value xmlns:d4p1="http://www.w3.org/2001/XMLSchema" i:type="d4p1:string">Nina Hawk</Value>
        </KeyValueOfstringanyType>
        <KeyValueOfstringanyType>
            <Key>Age</Key>
            <Value xmlns:d4p1="http://www.w3.org/2001/XMLSchema" i:type="d4p1:int">26</Value>
        </KeyValueOfstringanyType>
    </ArrayOfKeyValueOfstringanyType>
```