Let's get to work.

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` as a return type. We need to do that in order to shape our data the way we want it.

To implement this interface, we are going to create the new folder `DataShaping` in the `Repository` project and the new class `DataShaper`:

```
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;
    }
}
```

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

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.