DRAFT - IT Project Guidance

.NET.Core EF & ADO Queryability

Version:

0.1

## Purpose

This document serves as a practical guide for developers and architects, both technical and non-technical, to understand how custom-developed APIs using .NET Core can deliver maintainable, queryable, standards based and secure data access. The guidance is intended to show how decades of tooling and best practices have made the custom route simpler, safer, and ultimately more sustainable than connector-driven integration platforms that appear simple but mask long-term cost and rigidity.

This document serves as a practical guide for developers looking to implement efficient, queryable, and maintainable APIs using .NET Core, Entity Framework Core (EF Core), AutoMapper, and OData. It aims to bridge the gap between basic RESTful API development and fully queryable, domain-driven API architecture, demonstrating how AutoMapper’s ProjectTo<> improves performance and how OData enhances flexibility and standardization. By following this guide, developers can build APIs that are scalable, secure, and optimized for data-driven applications.

## Synopsis

This document provides a structured introduction to how to build queryable, efficient, and standards-based APIs using .NET Core, Entity Framework Core (EF Core), AutoMapper, and OData, without sacrifice of security or performance. It covers the transition from basic RESTful API endpoints to queryable DTO based APIs to fully standards based queryable OData endpoints, leveraging AutoMapper’s powerful ProjectTo<>. The guide ensures security, performance, and interoperability, helping developers implement scalable, maintainable, and flexible data-driven APIs.

## Contents

[Purpose 1](#_Toc193994910)

[Synopsis 1](#_Toc193994911)

[Contents 2](#_Toc193994912)

[Introduction 4](#_Toc193994913)

[Development Concepts 4](#_Toc193994914)

[Domains, Entities and Value Objects 4](#_Toc193994915)

[APIs 5](#_Toc193994916)

[Data Transfer Objects (DTOs) 6](#_Toc193994917)

[Technologies 6](#_Toc193994918)

[.NET Core LINQ 6](#_Toc193994919)

[EF Core 7](#_Toc193994920)

[Reputation 7](#_Toc193994921)

[EF’s Schema Development 7](#_Toc193994922)

[AutoMapper 8](#_Toc193994923)

[Demonstration 9](#_Toc193994924)

[Information 9](#_Toc193994925)

[Schema Creation 9](#_Toc193994926)

[Entity Definition 9](#_Toc193994927)

[Domain Schema Development 11](#_Toc193994928)

[Review by Specialists 12](#_Toc193994929)

[LINQ based Querying 12](#_Toc193994930)

[Developing APIs 13](#_Toc193994931)

[Entity REST Controller and endpoints 13](#_Toc193994932)

[DTO based REST API Controllers and Endpoints 14](#_Toc193994933)

[DTOs 14](#_Toc193994934)

[DTO enabled Controllers 15](#_Toc193994935)

[Registering AutoMapper Conversions 16](#_Toc193994936)

[Refactor the Controllers and Endpoints 16](#_Toc193994937)

[Queryable DTO API Endpoints 17](#_Toc193994938)

[Security 20](#_Toc193994939)

[Standards-Based Approach 20](#_Toc193994940)

[ODATA Controller 21](#_Toc193994941)

[Outputs 24](#_Toc193994942)

[Standard REST Response 25](#_Toc193994943)

[ODATA Response 25](#_Toc193994944)

[Final Thoughts 26](#_Toc193994945)

[Conclusion 27](#_Toc193994946)

[Appendices 28](#_Toc193994947)

[Appendix A - Document Information 28](#_Toc193994948)

[Author & Collaborators 28](#_Toc193994949)

[Versions 28](#_Toc193994950)

[Images 28](#_Toc193994951)

[Tables 28](#_Toc193994952)

[References 28](#_Toc193994953)

[Review Distribution 28](#_Toc193994954)

[Audience 28](#_Toc193994955)

[Structure 28](#_Toc193994956)

[Diagrams 29](#_Toc193994957)

[Acronyms 29](#_Toc193994958)

[API 29](#_Toc193994959)

[GUI 29](#_Toc193994960)

[ICT 29](#_Toc193994961)

[IT 29](#_Toc193994962)

[UI 29](#_Toc193994963)

[Terms 29](#_Toc193994964)

[Application Programming Interface 29](#_Toc193994965)

[User 29](#_Toc193994966)

[User Interface 29](#_Toc193994967)

# Introduction

NET Core is a modern, enterprise-grade cross platform development stack built to deliver scalable, secure, and performant applications across platforms. While many technologies can be used to integrate systems, few match the maturity and structured extensibility of the .NET ecosystem, particularly in how it treats information as structured, queryable data.

An enterprise-grade development stack means it is designed to handle large-scale, mission-critical workloads, support robust security and authentication mechanisms, and ensure maintainability, scalability, and integration with existing systems. Few technology stacks meet these criteria. Java, particularly with the Spring Boot framework, is a strong alternative, offering similar capabilities for large-scale enterprise applications.While Python and JavaScript (Node.js) are widely used, they typically require additional frameworks and services to match the robustness, scalability, and structured development approach that .NET Core and Java offer out of the box. Python excels in data science and automation, while JavaScript dominates front-end and lightweight back-end applications but lacks a comparable enterprise-oriented stack of 25 years encoded enterprise and government experience.

### Integration Capabilities

Unlike integration suites that attempt to mask complexity using connectors and no-code tools, a well-structured .NET Core solution allows developers to express their integration domain model precisely, provide them via secure APIs endpoints, and keep the system open to new requirements without rebuilding endpoints every time.

Central to this approach is the use of Language Integrated Query (LINQ), Entity Framework Core (EF Core), Domain Driven Design Entities, DTOs, AutoMapper’s ability to broker both entities and queries between them, and optionally OData for consistent, standards-based queryability.

## Development Concepts

When developing non-trivial – i.e., beyond proof-of-concept or demo -- systems, Object-Oriented (OO) principles and the core concepts of Domain-Driven Design (DDD) remain the most recommended organisational approach to follow.

### Domains, Entities and Value Objects

While describing DDD principles is beyond the scope of this paper, touching lightly on the matter of Domains, Entities and Value objects is appropriate.

In DDD, logical domains are used to group related entities into distinct areas of purpose.

Every system has a foundational "System" domain, for managing core system functionalities including configuration, diagnostics, settings, error recording, sessions, users, etc.

And mature systems have a separate "Social" logical domain for managing persons, relationships, groups and roles.

On top of these two common base domains one or more business domains are developed, containing the actual entities relevant to the application’s purpose (commerce, contracts, education, health, etc.).

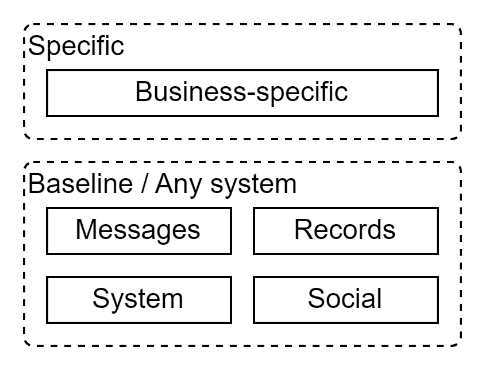


Figure 1: Common Domains stacks

Within logical domains, Entities represent unique, identifiable objects, each possessing a distinct identity and lifecycle. Value objects, are defined solely by their attributes and their managing entity’s ID, rather than having their own identity.

Entities and value objects are linked using one-to-one, one-to-many, or many-to-one relationships, forming structured associations that define the system’s data model and behaviour.

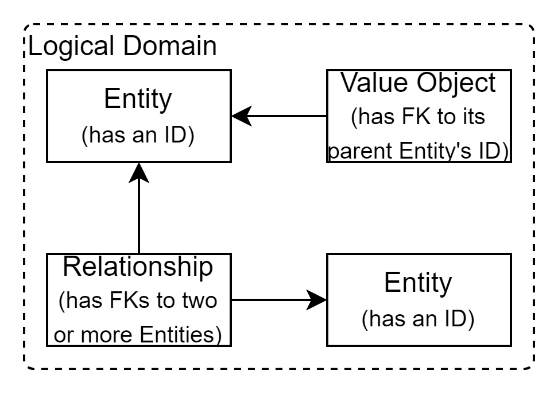


Figure 2: Domain Entities, Value Objects, Relationships

With these very simple but powerful models, any enterprise domain can be mapped and entities be developed as needed.

public class Person {

public int Id { get; set; }

public string FName { get; set; }

public string LName { get; set; }

public DateTime? DOB { get; set; }

public List<Membership> Memberships { get; set; } = new();

}

### APIs

APIs are a form of interface that provide remote third-party access to a system’s functionality. The predominant approach to developing externally accessible APIs for third parties is Representational State Transfer (REST).

This approach involves developing endpoints (URLs) to permit operations to Create new, Update existing, Retrieve (either a single item or a list), or [logically] Delete resources (mostly DB records)[[1]](#footnote-2).

**Note:** While an industry term, the term "Application Programming Interface (API)" is misleading. A more correct descriptive term could have been "Remote Resource Interface Layer" as integration APIs are mostly used to allow remote systems to interact with resources managed by the system, not change its programming.

### Data Transfer Objects (DTOs)

A key IT security concept is to not divulge to external nefarious actors the inner workings of systems.

DTOs is a shape of data *tailored to an external consumer*. They serve as a fundamental design pattern to expose only necessary data structures, preventing over-sharing, all while shielding internal domain models from visibility and manipulation by external consumers.

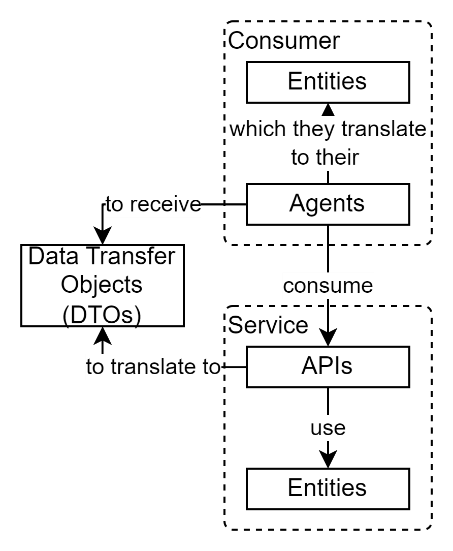


Figure 3: DTOs are used instead of internal Entities

By decoupling the API’s contract from internal modelling, DTOs allow greater flexibility in evolving the underlying model without breaking external obligations (integration contracts).

This also ensures that sensitive or irrelevant internal data is not inadvertently exposed, enforcing a clear separation between the internal logic and external API consumers.

public class StudentDto {

public int Id { get; set; }

public string GivenName { get; set; }

public string SurName { get; set; }

public int Age { get; private set; }

public List<EnrollmentDto> Enrollments { get; set; } = new();

}

public class EnrollmentDto {

public string SchoolName { get; set; }

public DateTime EnrolledFrom { get; set; }

public DateTime? EnrolledTo { get; set; }

}

The above DTOs allow the system to express a specific view over the Person entity, using service consumer terminology. In this case, they provide a view of a learner within an educational system, including their associated schools. Internal structures (e.g. foreign keys) are hidden.

They do not preclude the Person being reused for associations to other groups within other sectors (e.g., health, employment, military, etc.)

To our knowledge no commercial integration platform provides the separation of concerns of DTOs, and via their simplistic no-code development approach are only capable of exposing the internal Entities directly. This security concern requires monitoring over time.

## Technologies

Below are listed a small number of technologies relevant to understanding the approach recommended within this paper.

### .NET Core LINQ

.NET Core’s Language Integrated Query (LINQ) is an integral part of the .NET languages. It permits querying both collections of models in a structured, SQL-like manner, providing filtering, ordering, and projection in a fluent API.

An example would be filtering a collection of students aged over 14 and under 16, projecting them into new objects that contain only their given name, surname, and age, ordering by last name, and taking only the second page of 20 records:

var students = studentList

.Where(s => s.Age > 14 && s.Age < 16)

.Select(s => new { s.GivenName, s.Surname, s.Age })

.OrderBy(s => s.LName)

.Skip(20)

.Take(20)

.ToList();

LINQ is such a fundamental aspect of .NET Core that one finds it also within in two areas of development key to this document – datastore access via EF Core, and *making it accessible to external users via API endpoints*.

### EF Core

EF Core (EF) is the default ORM package for .NET projects. It acts as an Object-Relational Mapper (ORM), allowing developers to interact with databases using strongly typed entities and relationships. EF’s design aligns with Domain-Driven Design (DDD) objectives and therefore the management of Entities, Value Objects (VOs) using indexed tables, and using join tables to persist and support many-to-many relationships in EF Core.

#### Reputation

ORMS are now widely respected for eliminating SQL injection vulnerabilities and their enforcing a mature and maintainable DDD based approach to system development.

However they also have a mixed reputation among data specialists, partly because they contribute to their displacement, and partly as ORMs make schema development accessible to application developers who may lack expertise in indexing or query optimization, leading to suboptimal database performance.

While the industry does not advocate returning to traditional Model-First or Database-First schema development, it does recommend involving data specialists in schema reviews. The fundamental logic behind efficient database design remains unchanged, and experienced data professionals can quickly identify and correct flaws, even if modern tools have altered the implementation process.

Another source of friction is that databases were traditionally maintained and directly accessed via SQL by DBAs. Modern system design, however, treats the datastore as an implementation detail rather than the primary point of interaction. The maintenance, operation and development focus has shifted to the system as a whole, ensuring consistency, maintainability, and security through application-layer APIs or ORM-managed queries. This transition moves away from direct SQL access, reducing reliance on DBAs for routine operations and enabling greater flexibility in schema evolution.

With this current perspective, the physical, internal, datastore schema should not prioritise optimization for human readability but rather for efficient access patterns, scalability, and transactional integrity as dictated by the application.

#### EF’s Schema Development

A key feature of EF is its ability to manage schemas, supporting legacy Database-First, Model-First, and current Code-First development approaches.

Code-First flips the traditional model of designing a database schema separately (Database-First) and then adapting the code to match it. This eliminates the friction between distinct development groups and automates database management. Code-First defines database structures through C# classes and migrations, replacing the need for separate schema design tools.

As automated deployment practices dominate modern development, Code-First has become the preferred and widely adopted approach. However, to maximize its benefits, developers *really* should seek input from database specialists to refine schema designs.

### AutoMapper

AutoMapper is a widely adopted library in .NET Core that simplifies mapping between domain entities and DTOs and back again.

var studentDto = \_mapper.Map<StudentDto>(studentEntity);

One of its most powerful features, exposed via its ProjectTo<>() extension, allows mapping not only between DTOs and entities but also transforming queries into DTOs without exposing the internal domain models. This ensures security by preventing unintended data exposure while enabling efficient data retrieval and transformation for API consumers.

# Demonstration

The following is a demonstration at how to follow DDD patterns, to expose APIs that while providing Queryability.

## Information

To illustrate these concepts, consider the following example Education Sector hierarchy of entities:

+-- ABC School

|

+-- StudentAt (Join Table)

|

+-- Person (Student)

## Schema Creation

Entities, Value Objects, and Domains are fundamentally interrelated in a structured system. Entities represent uniquely identifiable objects with a lifecycle, while Value Objects define immutable characteristics that add meaning to entities. Domains group these entities and value objects into logical areas of responsibility, ensuring that business logic and data integrity remain well-organized and maintainable. Together, they form the backbone of Domain-Driven Design, providing a structured approach to managing complex data relationships.

### Entity Definition

Using EF Core’s Code-First approach, the entities are defined through C# classes:

public class Person {

public int Id { get; set; }

public string FName { get; set; }

public string LName { get; set; }

public DateTime DOB { get; set; }

public ICollection<Membership> Memberships { get; set; }

}

public class StudentAt

{

public int Id { get; set; }

public int PersonId { get; set; }

public Person Person { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public int YearLevel { get; set; }

public DateTime? StartDate { get; set; }

public DateTime? EndDate { get; set; }

}

public class School

{

public int Id { get; set; }

public string Name { get; set; }

}

### Domain Schema Development

Entities and value objects are not isolated elements but are part of a broader domain logical model that is translates to a database ‘schema’ when persisted there.

As the entities are registered within the model, indexes are added for efficient querying, value types and size constraints are added for storage efficiency, and the entities are related correctly together to make a cohesive and coherent model of a specific domain.  
  
This information can be gleaned from attributes attached to Entity properties or done in a ‘fluent’ manner when developing the schema model[[2]](#footnote-3) as per below, which shows the development of a model for an education domain:

public class ApplicationDbContext : DbContext

{

public DbSet<Person> Persons { get; set; }

public DbSet<School> Schools { get; set; }

public DbSet<StudentAt> StudentAt { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Person>(entity =>

{

entity.HasKey(e => e.Id);

entity.Property(e => e.Id).ValueGeneratedOnAdd();

entity.Property(e => e.FirstName)

.IsRequired()

.HasMaxLength(50);

entity.Property(e => e.LastName)

.IsRequired()

.HasMaxLength(50);

entity.Property(e => e.DateOfBirth);

entity.HasMany(e => e.StudentAt)

.WithOne(e => e.Person)

.HasForeignKey(e => e.PersonId)

.OnDelete(DeleteBehavior.Cascade);

});

modelBuilder.Entity<School>(entity =>

{

entity.HasKey(e => e.Id);

entity.Property(e => e.Id).ValueGeneratedOnAdd();

entity.Property(e => e.Name)

.IsRequired()

.HasMaxLength(100);

});

modelBuilder.Entity<StudentAt>(entity =>

{

entity.HasKey(e => e.Id);

entity.Property(e => e.Id).ValueGeneratedOnAdd();

entity.Property(e => e.PersonId).IsRequired();

entity.Property(e => e.SchoolId).IsRequired();

entity.Property(e => e.YearLevel).IsRequired();

entity.Property(e => e.StartDate);

entity.Property(e => e.EndDate);

entity.HasOne(e => e.Person)

.WithMany(e => e.StudentAt)

.HasForeignKey(e => e.PersonId);

entity.HasOne(e => e.School)

.WithMany()

.HasForeignKey(e => e.SchoolId);

});

}

}

#### Review by Specialists

While Code-First allows developers to define the database schema without relying on a database specialist, indexing strategies require expertise that many developers may lack. Poor indexing can lead to performance issues, slowing queries and reducing efficiency. To avoid bottlenecks and optimize data retrieval, we can’t stress enough how *strongly* recommended it is to use a database specialist at this stage as they can best suggest what to index while taking into context execution cost impacts.

### LINQ based Querying

Once the entities and domain are created the application tier can process EF returned collections with LINQ operators in the same way as was demonstrated earlier with in-memory collections.

For example querying the database for students older than 14, ordering by last name, and retrieving the third page of 20 records would be as follows:

using (var context = new ApplicationDbContext())

{

var students = context.Persons

.Where(p => p.StudentAt.Any() && EF.Functions.DateDiffYear(p.DateOfBirth, DateTime.Today) > 14)

.OrderBy(p => p.LastName)

.Skip(40) // Third page of 20 records

.Take(20)

.ToList();

}

### Developing APIs

The above created entities and domain are accessible internally by the application’s logic – but not yet accessible in any manner from outside the app.

For that to happen APIs need to be registered first.

### Entity REST Controller and endpoints

We start with a traditional REST endpoint, exposing Entities we defined earlier.

These controllers in turn define endpoints for one or more of the four basic actions (Create, Retrieve, Update, Delete).

[ApiController]

[Route("api/persons")]

public class PersonController : ControllerBase

{

private readonly ApplicationDbContext \_context;

public PersonController(ApplicationDbContext context)

{

\_context = context;

}

[HttpGet]

public async Task<ActionResult<IEnumerable<Person>>> GetPersons()

{

var persons = await \_context.Persons.ToListAsync();

return Ok(persons);

}

}

Invoking the above api/rest/Person/ endpoint, returns a list of students:

[

{

"id": 1,

"fName": "John",

"lName": "Doe",

"dob": "2005-08-15T00:00:00"

},

{

"id": 2,

"fName": "Jane",

"lName": "Smith",

"dob": "2006-02-21T00:00:00"

}

… etc. until the end of the table (could be long)

]

The above approach – while common in PoCs and published learning material – is not suitable for production code because it violates a number of fundamental security principles.

One issue discussed earlier is that it exposes the internal EF-managed entities and values.

The second is that there is no limiting of the size of the query. If the table has 8 million rows, the result will be a json file of 8 million records, that might time out one of the tiers, but certainly tie up precious resources.

Beyond security, there is the interoperability expectations: if a developer changes the entity *in any way*, this will change the shape of what is exposed to clients. In other words, the change will break the contract established between two parties.

Finally, while using "fName" and "lName" may be acceptable among developers, in professional and official government contexts, the preferred long term option is to use correct terminology. In this case, "GivenName" and "Surname."

### DTO based REST API Controllers and Endpoints

The issues can be addressed and resolved by implementing DTOs. To achieve this, we need to take a more structured approach and introduce DTOs that decouple external exposure from internal entity structures.

#### DTOs

Instead of directly exposing internal EF entities, we define DTOs that provide a controlled representation of data. The following DTO replaces the informal internal use of "FName" and "LName" with "GivenName" and "Surname" while also ensuring only necessary data is exposed, but also removing data that may not be required to be exposed:

public class StudentDto

{

public int Id { get; set; }

public string GivenName { get; set; }

public string Surname { get; set; }

public bool WithinSchoolAge { get; set; } // Derived from DOB instead of exposing it directly

}

#### DTO enabled Controllers

Adding DTOs to a project requires refactoring the API controllers to return DTOs rather than the internal entities they keep internal.

This ensures a cleaner separation between the external API contracts and the underlying internal data model, improving security, maintainability, and flexibility.  
  
The changes are not major – all one has to do is map the returned objects before returning those instead of the source entities.

Below is an example of doing this kind of mapping manually:

using Microsoft.AspNetCore.OData.Query;

using Microsoft.AspNetCore.OData.Routing.Controllers;

using Microsoft.AspNetCore.Mvc;

using System.Linq;

using System;

[Route("api/rest/[controller]")]

[ApiController]

public class StudentsController : ODataController {

private readonly IStudentRepository \_studentRepository;

public StudentsController(IStudentRepository studentRepository) {

\_studentRepository = studentRepository;

}

[EnableQuery]

[HttpGet]

public IActionResult Get() {

var students = \_studentRepository.GetAllStudents()

.AsQueryable()

.Select(s => new StudentDto {

Id = s.Id,

GivenName = s.FName,

Surname = s.LName,

FullName = $"{s.FName} {s.LName}",

DOB = s.DOB,

OfEducationAge = s.DOB.AddYears(7) <= DateTime.Today && s.DOB.AddYears(21) > DateTime.Today

})

.ToList();

return Ok(students);

}

}

#### Registering AutoMapper Conversions

Mapping entities by hand quickly becomes a maintenance issue for a number of reasons. The first, the sheer quantity of properties involved leads to making errors. The second is that while mapping the top element (e.g.: Person), one has to also consider and account for all the myriad related objects (e.g.: Addresses, Schools, Profiles, etc.). The task is never ending.

To facilitate mapping between entities and DTOs, you can do this by hand – but in large apps, this becomes cumbersome and prone to error. We recommend *strongly* using mapping service, such as AutoMapper.

It allows seamless transformations, ensuring DTOs receive properly formatted values while preventing direct exposure of the internal domain model.

public class StudentProfile : Profile {

public StudentProfile() {

CreateMap<Student, StudentDto>()

.ForMember(dest => dest.FullName,

opt => opt.MapFrom(src => $"{src.GivenName} {src.Surname}"))

.ForMember(dest => dest.OfEducationAge,

opt => opt.MapFrom(src => src.DOB.AddYears(7) <= DateTime.Today && src.DOB.AddYears(21) > DateTime.Today));

}

}

Note:   
while there are other mappers, there is a specific capability that we will be using later that AutoMapper has that others don’t.

#### Refactor the Controllers and Endpoints

Adding DTOs to a project requires refactoring the API controllers to return DTOs rather than the internal entities they hide.

This ensures a cleaner separation between the external API contracts and the underlying internal data model, improving security, maintainability, and flexibility.  
  
The changes are not major – the key change is passing the returned object (a Student) through AutoMapper to get back a list of equivalent DTOs (in this case StudentDto).

Note that because the response is correctly wrapped in an ActionResult<T> derived Ok() wrapper, one doesn’t even need to change the endpoint’s return type.

[Route("api/[controller]")]

[ApiController]

public class StudentsController : ControllerBase

{

private readonly IStudentRepository \_studentRepository;

public StudentsController(IStudentRepository studentRepository) {

\_studentRepository = studentRepository;

}

[HttpGet]

public IActionResult GetStudents() {

var students = \_studentRepository.GetAllStudents()

.Select(s => Mapper.Map<StudentDto>(s));

return Ok(students);

}

}

Now when you invoke the endpoint, you’ll get back a list of StudentDTOs (notice the values returned are labelled Given/Surname and Flag, instead of the names of the internal entity properties, FName, LName and DOB).

[

{

"id": 1,

"givenName": "John",

"surname": "Doe",

"dob": "2005-08-15T00:00:00",

"fullName": "John Doe",

"ofEducationAge": true

},

{

"id": 2,

"givenName": "Jane",

"surname": "Smith",

"dob": "2006-02-21T00:00:00",

"fullName": "Jane Smith",

"ofEducationAge": true

}

]

### Queryable DTO API Endpoints

If you invoke the above controller, you only have the endpoints defined. If you want an endpoint that permits paging, you will have to add either arguments and/or alternate endpoints for each common operation (paging, sorting, filtering, etc.) on each get endpoint.

A key issue with that approach is that you’ll be defining your own syntax for these operations, making it less intuitive, increasing documentation requirements and tech support costs as you guide end users through your custom implementation of basic universal concepts of paging, sorting, filtering, projecting, etc.

Not only is this frustrating for end users – having to ask for a new feature and have it prioritised to being delivered months after they have worked around the issue or moved on – but it is also wasteful in terms of ongoing resources and development, testing, and change control costs.

There is a much more practical way of delivering this functionality.

You add queryability to the endpoint by decorating the GET endpoint with the [EnableQuery] attribute.

[Route("api/[controller]")]

[ApiController]

public class StudentsController : ControllerBase {

…

[HttpGet]

[EnableQuery]

public IActionResult GetStudents()

{

var studentDtos = \_studentRepository.GetAllStudents()

.Select(s => Mapper.Map<StudentDto>(s));

return Ok(studentDtos);

}

}

With this single attribute in place, one can now include a number of operations to perform on the endpoint’s results:

* $filter: Filters records based on a condition.  
  Example: api/students?$filter=OfEducationAge eq true
* $orderby: Sorts results based on a field.  
  Example: api/students?$orderby=Surname asc
* $top: Limits the number of records returned.  
  Example: api/students?$top=20
* $skip: Skips a specified number of records (for pagination).  
  Example: api/students?$skip=40
* $select: Returns only specific fields.  
  Example: api/students?$select=GivenName,Surname
* $expand: Expands related entities.  
  Example: api/students?$expand=Courses

The following is an example of applying some of the above:

GET /api/students?$filter=dob lt 2008-01-01&$orderby=surname asc&$skip=40&$top=20

Adding queryability a big leap forward – in that you don’t have to define arguments and/or method overrides for all the possible filtering, sorting, etc operations to perform on the returned list of DTOs –

However, there is a subtle but impactful issue at play: the query is requesting the results ordered by a DTO property (surname), and ProjectTo<> is not yet being used.

What will happen is that the operation will be done on the mapped entities \*in memory\* *after* the set has been returned. That will use far more resources than intended or expected.

Note:  
Mistakes like this is why ORMs and related technologies, in the hands of inexperienced developers, has sometimes gained an unfair reputation.   
However, the issue -- once spotted -- is easily corrected.

The correct way to bring all 3 concepts together (DTO, APIs, Queryability) is with the use of a 4th concept: AutoMapper’s ProjectTo<> statement:

The correct way to do the controller is as below.

[Route("api/[controller]")]

[ApiController]

public class StudentsController : ControllerBase {

private readonly IMapper \_mapper;

private readonly IStudentRepository \_studentRepository;

public StudentsController(IStudentRepository studentRepository, IMapper mapper){

\_studentRepository = studentRepository;

\_mapper = mapper;

}

[HttpGet]

[EnableQuery]

public IActionResult GetStudents() {

var students = \_studentRepository.GetQueryableStudents()

.ProjectTo<StudentDto>(\_mapper.ConfigurationProvider);

return Ok(students);

}

}

What happens now is that the query is projected onto the *internal entities* – so that the operation happens \*in the database\*, such that only the matching Students are returned, and in doing so, resources are not wasted.

#### Security

Using ProjectTo<> removes one of the two approaches a DoS attack can be attempted, as it limits waste of the application’s memory.

However, there is another method that has to be countered – the payload size between tiers. This you solve by adding an upper PageSize limit, so that even if an attacker – or careless developer – requests all requests in one go, the query applied to the database is always limited to 100 records.

[HttpGet]

[EnableQuery(PageSize=100)]

public IActionResult GetStudents() {

var students = \_studentRepository.GetQueryableStudents()

.ProjectTo<StudentDto>(\_mapper.ConfigurationProvider);

return Ok(students);

}

}

Standards-Based Approach  
Enabling queryable endpoints was a significant improvement, offering greater flexibility for users. While some aspects of this functionality are inspired by the OData standard, it does not fully adhere to any established standard, including OData.

Another key limitation is that each GET endpoint has to be decorated with [EnableQuery] and it is almost certain some will be missed.

Another key limitation is that each endpoint operates independently, focusing only on individual entities rather than addressing the needs of the entire domain. This fragmented approach can lead to inconsistencies and inefficiencies in how data is accessed and managed.

Another key limitation – especially for government agencies -- adherence to recognized standards is essential due to the long service lifespans of their systems. Following established standards ensures interoperability, security, and maintainability over time. Rather than relying on [EnableQuery], transitioning to *full* OData adoption provides a more robust and structured solution.

### ODATA Controller

A more mature option and compliant method is to develop full ODATA enabled Controllers.

OData offers standardized, secure, and efficient querying, making it particularly well-suited for handling large datasets and supporting external consumers with advanced filtering and query capabilities.

But it goes beyond that. It makes the endpoints become part of a cohesive whole, a domain of endpoints.

This is done by refactoring the api another time, this time to inherit from the richer ODataController.

[ApiController]

[Route("api/rest/[controller]")] // REST Route

[ODataRoutePrefix("Students")] // OData Route

public class StudentsController : ODataController

{

private readonly IStudentRepository \_studentRepository;

private readonly IMapper \_mapper;

public StudentsController(IStudentRepository studentRepository, IMapper mapper)

{

\_studentRepository = studentRepository;

\_mapper = mapper;

}

// REST: GET /api/rest/students

// OData: GET /api/odata/students

[HttpGet]

[ODataRoute]

[EnableQuery] // Needed to be Queryable as a standard REST controller

public IActionResult Get(ODataQueryOptions<StudentDto> options)

{

var studentsDto = \_studentRepository.GetAllStudents().AsQueryable();

.ProjectTo<StudentDto>(\_mapper.ConfigurationProvider);

var filteredStudentsDto = Request.ApplyQuery(studentsDto); // Apply filters

Response.Headers["X-Total-Count"] = filteredStudentsDto.Count.ToString();

return Ok(filteredStudentsDto);  
 }

// REST: GET /api/rest/students/{id}

// OData: GET /api/odata/students({id})

[HttpGet("{id}")]

[ODataRoute("({id})")]  
 [EnableQuery] // Needed to be Queryable as a standard REST controller

public IActionResult Get(int id)

{

var studentDtoQuery = \_studentRepository.GetAllStudents()

.Where(s => s.Id == id)

.ProjectTo<StudentDto>(\_mapper.ConfigurationProvider) // Project first

.AsQueryable();

var filteredStudentDto =

Request.ApplyQuery(studentDtoQuery)  
 .FirstOrDefault(); // Apply OData $select or other

return filteredStudentDto == null ? NotFound() : Ok(filteredStudentDto);

}

// REST: POST /api/rest/students

// OData: POST /api/odata/students

[HttpPost]

[ODataRoute]

public IActionResult Post([FromBody] StudentDto studentDto)

{

if (!ModelState.IsValid)

return BadRequest(ModelState);

var student = \_mapper.Map<Student>(studentDto);

\_studentRepository.AddStudent(student);

return Created(student);

}

// REST: PUT /api/rest/students/{id}

// OData: PUT /api/odata/students({id})

[HttpPut("{id}")]

[ODataRoute("({id})")]

public IActionResult Put(int id, [FromBody] StudentDto studentDto)

{

if (!ModelState.IsValid)

return BadRequest(ModelState);

var existingStudent = \_studentRepository.GetAllStudents().FirstOrDefault(s => s.Id == id);

if (existingStudent == null)

return NotFound();

\_mapper.Map(studentDto, existingStudent);

\_studentRepository.UpdateStudent(existingStudent);

return NoContent();

}

// REST: DELETE /api/rest/students/{id}

// OData: DELETE /api/odata/students({id})

[HttpDelete("{id}")]

[ODataRoute("({id})")]

public IActionResult Delete(int id)

{

var student = \_studentRepository.GetAllStudents().FirstOrDefault(s => s.Id == id);

if (student == null)

return NotFound();

\_studentRepository.DeleteStudent(student);

return NoContent();

}

}

However even with this change to the Controller – enheriting from ODataController – it still isn’t really an OData controller until you *register* it as part of an API *Domain*.

….

using Microsoft.AspNet.OData.Builder;

using Microsoft.AspNet.OData.Extensions;

using Microsoft.OData.Edm;

public static class ODataConfig

{

public static IEdmModel GetEdmModel()

{

var builder = new ODataConventionModelBuilder();

// Register DTOs instead of Entity Models

builder.EntitySet<StudentDTO>("Students");

builder.EntitySet<PeopleDTO>("People");

builder.EntitySet<SchoolDTO>("Schools");

return builder.GetEdmModel();

}

}

And you register the whole domain in one go under another base route (‘api/odata/’) defining what operations you wish to permit end users.

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers().AddOData(opt =>

opt.Count().Filter().OrderBy().Expand().Select().SetMaxTop(100)

.AddRouteComponents("api/odata", ODataConfig.GetEdmModel()));

services.AddScoped<IStudentRepository, StudentRepository>();

services.AddAutoMapper(typeof(Startup));

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseRouting();

app.UseEndpoints(endpoints =>

{

// Standard REST routes

endpoints.MapControllers();

// OData routes

endpoints.Select().Filter().OrderBy().Expand().Count();

endpoints.MapODataRoute("ODataRoute", "api/odata", ODataConfig.GetEdmModel());

});

}

With the above in place the above controller are accessible under both simple REST and ODATA compliant rest, as per below.  
REST API (api/rest/):

* GET /api/rest/students
* GET /api/rest/students/{id}
* POST /api/rest/students
* PUT /api/rest/students/{id}
* DELETE /api/rest/students/{id}

OData API (api/odata/)

* GET /api/odata/students
* GET /api/odata/students({id})
* POST /api/odata/students
* PUT /api/odata/students({id})
* DELETE /api/odata/students({id})

### Outputs

While both versions of the GET endpoint will supports $filter, $orderby, $top, $expand, $select, and $count on DTOs, the key difference will be the output.

#### Standard REST Response

So for the following Request:

GET /api/rest/students

The following will be the respone:

[

{

"id": 1,

"givenName": "John",

"surname": "Doe",

"dob": "2005-08-15T00:00:00",

"fullName": "John Doe",

"ofEducationAge": true

},

{

"id": 2,

"givenName": "Jane",

"surname": "Smith",

"dob": "2006-02-21T00:00:00",

"fullName": "Jane Smith",

"ofEducationAge": true

}

]

#### ODATA Response

OData’s output is richer, so for the following Request:

GET /api/odata/students?$select=givenName,fullName  
the Response will be:

{

"@odata.context": "https://example.com/api/odata/$metadata#Students(givenName,fullName)",

"value": [

{

"givenName": "John",

"fullName": "John Doe"

},

{

"givenName": "Jane",

"fullName": "Jane Smith"

}

]

}

But it goes beyond the difference in output. While Queryable REST controllers provide some of OData’s operations, a crucial one that it doesn’t do is $expand. That alone makes it worth it. There’s also, $count, and couple of others that just don’t exist in the simpler REST implementation.

## Final Thoughts

The above was an example using a single domain – objects in an fictitious Education schema/domain. Other objects in the system would be defined under different domains – e.g. “System” (Diagnostics, Errors, Sessions, Operations, Permissions, System Roles, Users, etc.) or “Social” (People, Relationships, Groups, Group Roles) domain. It will depend on the system type and development capabilities as to how far to go with precision maintainable development versus just getting something good enough to produce value.

# Conclusion

Each capability is valuable in its own right – but when combined, produce a truly advantageous outcome. One that remains

* secure to government expectations in that it remains robust against exposure of internal workings and attempts at DoS attacks,
* permitting maintainability and modifiability of internal logic and entities without risk of breaking contracts with external API service consumers
* by making the end points queryable permits providing a means to address a vast pallet of yet unknown use cases in the future, with no added investment in this area.

In other words, secure, maintainable, usable and compliant.

Appendices

Appendix A - Document Information

### Author & Collaborators

Sky Sigal, Solution Architect

### Versions

* 1. Initial Draft

### Images

[Figure 1: Common Domains stacks 5](#_Toc195693805)

[Figure 2: Domain Entities, Value Objects, Relationships 5](#_Toc195693806)

[Figure 3: DTOs are used instead of internal Entities 6](#_Toc195693807)

### Tables

**No table of figures entries found.**

### References

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
| Russell Campbell,  Project Manager |  |
| Duncan Watson,  Enterprise Architect |  |
| Amy Orr,  Data Enterprise Architect |  |
| Carl Klitcher, Solution Architect |  |
| Vincent Wierdsma,  Lead Developer |  |

### Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

### Structure

Where possible, the document structure is guided by either ISO-\* standards or best practice.

### Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

### Acronyms

API

: [Application Programming Interface](#Term_ApplicationProgrammingInterface).

GUI

: [Graphical User Interface](#Term_ApplicationProgrammingInterface). A form of [UI](#Acronym_UI).

ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. [IT](#Acronym_IT) is a subset of ICT.

IT

: acronym for Information, using Technology to automate and facilitate its management.

UI

: User Interface. Contrast with [API](#Acronym_API).

### Terms

Refer to the project’s Glossary.

Application Programming Interface

: an Interface provided for other systems to invoke (as opposed to User Interfaces).

User

: a human user of a system via its UIs.

User Interface

: a system interface intended for use by system users. Most computer system UIs are Graphics User Interfaces ([GUI](#Acronym_GUI)) or Text/Console User Interfaces (TUI).

1. Ie, “CRUD” Operations. See also the BREAD acronym. [↑](#footnote-ref-2)
2. The fluent approach has more capability than the simpler attribute based approach and is therefore the approach we recommend. [↑](#footnote-ref-3)