ASP.NET CORE ESSENTIALS

Module 11-15

Table of Contents

Module 11: API Basics in ASP.NET Core

Modern web applications often need to expose functionality to clients like mobile apps, SPAs (Single Page Applications), or third-party services. This is done using APIs (Application Programming Interfaces).

ASP.NET Core provides a **robust framework for building RESTful APIs**, leveraging MVC patterns, routing, model binding, and built-in features for validation and response formatting.

This module covers:

- What an API is and why we need it
- Differences between MVC controllers and API controllers
- Setting up an API project
- Routing and model binding
- Returning JSON responses
- Best practices for basic API design

11.1 What Is an API?

An API is a **contract** between the server and clients, allowing **data exchange** over HTTP. Key characteristics:

- Stateless: Each request is independent
- Uses HTTP verbs (GET, POST, PUT, DELETE) to define actions
- Returns data, usually in **JSON** format
- Can be consumed by multiple clients (web, mobile, IoT)

11.2 MVC Controller vs API Controller

Feature	MVC Controller	API Controller
Return type	Views (HTML)	JSON, XML, or other data formats
[ApiController]	Optional	Required for modern API features
Model binding	Works with forms	Works with JSON, query string, route, etc.
Automatic validation	Needs manual check	Automatic ModelState validation

API controllers are designed for **data-only responses**, not HTML views.

11.3 Setting Up an API Controller

1. Create a new **ASP.NET Core Web API project**.
2. Add `[ApiController]` attribute and route prefix.

```
using Microsoft.AspNetCore.Mvc;

[ApiController]
[Route("api/[controller]")]
public class StudentsController : ControllerBase
{
    private readonly SchoolDbContext _context;

    public StudentsController(SchoolDbContext context)
    {
        _context = context;
    }
}
```

- `[ApiController]` → Enables automatic model validation and response formatting
- `[Route("api/[controller]")]` → Base route (e.g., `api/students`)
- `ControllerBase` → Provides only API functionality (no views)

11.4 API Routing

API routing can be **conventional** or **attribute-based**.

11.4.1 Attribute Routing

```
[HttpGet]                   // GET api/students
public IActionResult GetAll()
{
    var students = _context.Students.ToList();
    return Ok(students);
}

[HttpGet("{id}")]           // GET api/students/1
public IActionResult GetById(int id)
{
    var student = _context.Students.Find(id);
    if (student == null) return NotFound();
    return Ok(student);
}
```

- `[HttpGet]`, `[HttpPost]`, `[HttpPut]`, `[HttpDelete]` → map to HTTP verbs

- `{id}` → route parameter, automatically bound to method argument

---

11.5 Model Binding in API

ASP.NET Core automatically binds incoming data from:

- **Query strings** → `?name=John`
- **Route parameters** → `/api/students/1`
- **Request body (JSON)** → `[FromBody]` attribute

Example:

```
[HttpPost]
public IActionResult Create([FromBody] StudentDTO dto)
{
    if (!ModelState.IsValid) return BadRequest(ModelState);

    var student = new Student
    {
        Name = dto.Name,
        Email = dto.Email
    };

    _context.Students.Add(student);
    _context.SaveChanges();

    return CreatedAtAction(nameof(GetById), new { id = student.StudentId }, student);
}
```

- `[FromBody]` binds JSON from request body
- `CreatedAtAction` → Returns HTTP 201 with location header

11.6 Returning Responses

API responses can be returned using:

Method	Description
`Ok(object)`	HTTP 200 with data
`CreatedAtAction()`	HTTP 201 for new resources
`BadRequest()`	HTTP 400 for validation errors
`NotFound()`	HTTP 404 if resource not found
`NoContent()`	HTTP 204 when no data is returned

Consistently using proper HTTP status codes improves client-side error handling.

11.7 Automatic Model Validation

With `[ApiController]`, ASP.NET Core automatically:

- Validates the model using Data Annotations
- Returns **HTTP 400** if `ModelState` is invalid

Example:

```
[HttpPost]
public IActionResult Create([FromBody] StudentCreateDTO dto)
{
    // No need to manually check ModelState
    var student = _mapper.Map<Student>(dto);
    _context.Students.Add(student);
    _context.SaveChanges();
    return CreatedAtAction(nameof(GetById), new { id = student.StudentId }, student);
}
```

This reduces boilerplate code and ensures **consistent validation behavior**.

11.8 Example: Simple CRUD API

```
[HttpGet]
public IActionResult GetAll() => Ok(_context.Students.ToList());

[HttpGet("{id}")]
public IActionResult GetById(int id)
{
    var student = _context.Students.Find(id);
    if (student == null) return NotFound();
    return Ok(student);
}

[HttpPost]
public IActionResult Create([FromBody] StudentCreateDTO dto)
{
    if (!ModelState.IsValid) return BadRequest(ModelState);

    var student = _mapper.Map<Student>(dto);
    _context.Students.Add(student);
    _context.SaveChanges();

    return CreatedAtAction(nameof(GetById), new { id = student.StudentId }, student);
```

```
}

[HttpPut("{id}")]
public IActionResult Update(int id, [FromBody] StudentUpdateDTO dto)
{
    if (!ModelState.IsValid) return BadRequest(ModelState);

    var student = _context.Students.Find(id);
    if (student == null) return NotFound();

    _mapper.Map(dto, student);
    _context.SaveChanges();

    return NoContent();
}

[HttpDelete("{id}")]
public IActionResult Delete(int id)
{
    var student = _context.Students.Find(id);
    if (student == null) return NotFound();

    _context.Students.Remove(student);
    _context.SaveChanges();

    return NoContent();
}
```

11.9 Best Practices for API Basics

1. Always use **[ApiController]** for modern APIs
2. Use **DTOs** instead of EF entities
3. Return **appropriate HTTP status codes**
4. Validate inputs using **Data Annotations**
5. Keep endpoints **RESTful** and consistent
6. Use **AutoMapper** to reduce repetitive mapping code
7. Keep controller methods **small and focused**

11.10 Summary of API Basics

In this module, you learned:

- What an **API** is and why it's needed
- Differences between **MVC controllers and API controllers**
- How to **set up an API controller** in ASP.NET Core
- Routing, model binding, and HTTP verbs for RESTful endpoints

- Returning responses with proper **HTTP status codes**
- Automatic **model validation** with `[ApiController]`
- Example of a simple CRUD API with **DTOs and AutoMapper**
- Best practices for maintainable and clean API design

By mastering these basics, you are ready to build **full-featured REST APIs** that can interact with **web apps, mobile apps, and third-party clients**.

Module 12: API Development in ASP.NET Core CRUD

APIs are the backbone of modern web applications, enabling communication between web clients, mobile apps, and third-party services. ASP.NET Core makes it easy to build **RESTful APIs** with clean architecture, DTOs, validation, and proper HTTP responses.

This module covers:

- Setting up API controllers
- CRUD operations with DTOs and AutoMapper
- Model validation
- Returning proper HTTP status codes
- Best practices for maintainable APIs

12.1 Setting Up an API Controller

Create a controller for your API endpoints:

```
using Microsoft.AspNetCore.Mvc;

[ApiController]
[Route("api/[controller]")]
public class StudentsController : ControllerBase
{
    private readonly SchoolDbContext _context;
    private readonly IMapper _mapper;

    public StudentsController(SchoolDbContext context, IMapper mapper)
    {
        _context = context;
        _mapper = mapper;
    }
}
```

- `[ApiController]` enables automatic validation and response formatting
- `[Route("api/[controller]")]` defines the base URL (`api/students`)
- `ControllerBase` is used instead of `Controller` since APIs don't return views

12.2 CRUD Operations Using DTOs

Using **DTOs** ensures that sensitive database fields are never exposed and allows for consistent API responses.

12.2.1 GET: Retrieve All Students

```
[HttpGet]
public IActionResult GetAll()
{
    var students = _context.Students.ToList();
    var studentDTOs = _mapper.Map<List<StudentDTO>>(students);
    return Ok(studentDTOs);
}
```

- `Ok()` returns HTTP 200
- AutoMapper converts entities to DTOs

12.2.2 GET: Retrieve by ID

```
[HttpGet("{id}")]
public IActionResult GetById(int id)
{
    var student = _context.Students.Find(id);
    if (student == null) return NotFound(); // HTTP 404
    return Ok(_mapper.Map<StudentDTO>(student));
}
```

12.2.3 POST: Create a New Student

```
[HttpPost]
public IActionResult Create(StudentCreateDTO dto)
{
    if (!ModelState.IsValid) return BadRequest(ModelState);

    var student = _mapper.Map<Student>(dto);
    _context.Students.Add(student);
    _context.SaveChanges();

    var resultDTO = _mapper.Map<StudentDTO>(student);
    return CreatedAtAction(nameof(GetById), new { id = student.StudentId }, resultDTO); //
HTTP 201
}
```

- binds JSON payload to DTO
- `CreatedAtAction` sets **Location header** for the new resource

12.2.4 PUT: Update a Student

```
[HttpPut("{id}")]
public IActionResult Update(int id, [FromBody] StudentUpdateDTO dto)
{
    if (!ModelState.IsValid) return BadRequest(ModelState);

    var student = _context.Students.Find(id);
    if (student == null) return NotFound();

    _mapper.Map(dto, student);
    _context.SaveChanges();

    return NoContent(); // HTTP 204
}
```

- `NoContent()` indicates successful update without returning body

12.2.5 DELETE: Remove a Student

```
[HttpDelete("{id}")]
public IActionResult Delete(int id)
{
    var student = _context.Students.Find(id);
    if (student == null) return NotFound();

    _context.Students.Remove(student);
    _context.SaveChanges();

    return NoContent(); // HTTP 204
}
```

12.3 Model Validation

Data annotations on DTOs allow **automatic validation**:

```
public class StudentCreateDTO
{
    [Required]
    public string Name { get; set; }

    [Required, EmailAddress]
    public string Email { get; set; }

    [Required, MinLength(6)]
    public string Password { get; set; }
}
```

With `[ApiController]`, **invalid models automatically return HTTP 400** with validation errors.

12.4 Using AutoMapper for Mapping

AutoMapper converts **DTOs ↔ Entities** to avoid repetitive code:

```
var student = _mapper.Map<Student>(dto);         // DTO → Entity
var resultDTO = _mapper.Map<StudentDTO>(student); // Entity → DTO
```

Benefits:

- Cleaner controllers
- Consistent mapping logic
- Easy to extend for complex relationships

12.5 Returning Proper HTTP Status Codes

Operation	Status Code	Notes
GET all / GET by ID	200 OK	Success
POST (create)	201 Created	Include Location header
PUT (update)	204 No Content	Success, no body
DELETE	204 No Content	Success, no body
Validation error	400 Bad Request	ModelState invalid
Resource not found	404 Not Found	Non-existent ID

Returning correct HTTP codes improves client-side handling and API consistency.

12.6 Error Handling

Wrap database calls in **try-catch** if needed:

```
try
{
    _context.SaveChanges();
}
catch (Exception ex)
{
    return StatusCode(500, "Internal server error: " + ex.Message);
}
```

- Helps log unexpected errors
- Avoids leaking sensitive information

12.7 Best Practices for API Development

1. Always use **DTOs** for requests and responses
2. **Validate models** with Data Annotations
3. Use **AutoMapper** to simplify mapping logic
4. Return **proper HTTP status codes** consistently
5. Keep **controllers thin**; move business logic to **BLL or services**
6. Document APIs (Swagger/OpenAPI) for clients

12.8 Summary of Module 12

In this module, you learned:

- How to create an **API controller** in ASP.NET Core
- CRUD operations using **DTOs and AutoMapper**
- How to **validate input models** automatically
- How to **return proper HTTP status codes**
- Best practices for **maintainable, secure, and clean APIs**

By applying these concepts, you can build **robust RESTful APIs** suitable for web apps, mobile apps, and third-party integrations.

Module 13: 3-Tier Architecture Basics in ASP.NET Core

Modern applications benefit from **layered architecture**. In ASP.NET Core, a **3-tier architecture** separates responsibilities into three distinct layers:

1. **Presentation Layer** → Handles user interactions, HTTP requests, and responses.
2. **Business Logic Layer (BLL)** → Contains business rules, validations, and orchestrates data access.
3. **Data Access Layer (DAL)** → Handles communication with the database.

This separation improves **maintainability, testability, and modularity**.

13.1 Why 3-Tier Architecture?

- **Separation of concerns:** Each layer focuses on a single responsibility.
- **Modular structure:** Each layer can be developed and tested independently.
- **Reusability:** BLL and DAL can be reused across multiple presentation projects (Web API, MVC, Blazor).
- **Easy maintenance:** Changing database technology or business rules requires minimal changes in other layers.

13.2 Project Structure and Types

To implement a 3-tier architecture, create **separate projects for each layer** in the same solution:

Layer	Project Type	Purpose
Presentation Layer	ASP.NET Core Web API / MVC Project	Handles HTTP requests, returns views or JSON responses.
Business Logic Layer (BLL)	Class Library Project	Implements business rules, validations, and orchestrates data operations.
Data Access Layer (DAL)	Class Library Project	Implements database interactions using Entity Framework Core or other ORM.

Example Solution Structure:

```
MySolution/
├─ MyApp.API       (Presentation Layer)
├─ MyApp.BLL       (Business Logic Layer)
├─ MyApp.DAL       (Data Access Layer)
├─ MyApp.Common    (Optional: DTOs, Enums, Shared Models)
```

Each layer references the layer directly below it, but **lower layers do not reference higher layers**.

13.3 Dependency Flow

The **dependency direction** is always top-to-bottom:

```
Presentation Layer (API / MVC)
        ↓
Business Logic Layer (BLL)
        ↓
Data Access Layer (DAL)
```

- The **API layer** depends on BLL interfaces.
- The **BLL layer** depends on DAL interfaces.
- This ensures **decoupling**; each layer can be tested independently.

13.4 Dependency Injection (DI)

ASP.NET Core has built-in **Dependency Injection**. Using DI ensures that:

- Layers remain **loosely coupled**.
- Services can be replaced easily for testing or new implementations.

13.4.1 Registering Services

In `Program.cs` of the API project:

```
// Add DbContext (DAL)
builder.Services.AddDbContext<SchoolDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

// Register BLL and DAL services
builder.Services.AddScoped<StudentService>();   // BLL
builder.Services.AddScoped<StudentRepository>(); // DAL
```

13.4.2 Using Services in API Controllers

```
[ApiController]
[Route("api/[controller]")]
public class StudentsController : ControllerBase
{
    private readonly StudentService _studentService;

    public StudentsController(StudentService studentService)
    {
        _studentService = studentService;
    }

    [HttpGet]
    public IActionResult GetAll()
    {
        var students = _studentService.GetAllStudents();
        return Ok(students);
    }
}
```

The API controller **does not create BLL or DAL instances manually**. DI injects the required instances automatically.

13.5 Summary of 3-Tier Basics

1. **Presentation Layer** → Handles HTTP requests and responses.
2. **Business Logic Layer (BLL)** → Handles validations, rules, and orchestrates data access.
3. **Data Access Layer (DAL)** → Handles database operations.
4. **Dependency Injection** → Connects layers without tight coupling.
5. **Project Type** → API/MVC for presentation, Class Library for BLL and DAL.

With this structure in place, your application is **modular, maintainable, and ready for adding more advanced patterns like repository and unit of work in later modules**.

Module 14: Repository Pattern with Generic Interface

When building applications, **data access code** often becomes repetitive. For example, every entity (Student, Course, Teacher) needs methods like **GetAll, GetById, Add, Update, Delete**. Writing this for each entity leads to **duplicate code**.

The **Generic Repository Pattern** solves this by providing a **single reusable repository interface and implementation** that can handle **any entity type**.

14.1 Benefits of Generic Repository

- Reduces repetitive CRUD code
- Promotes **consistency** across data access
- Makes **unit testing easier**
- Encapsulates data access logic in **DAL**, leaving BLL clean
- Works with **any entity type**

14.2 Structure in 3-Tier Architecture

```
Presentation Layer (API / MVC)
        ↓
Business Logic Layer (BLL)
        ↓
Data Access Layer (DAL)
  ├─ Generic Repository Interface (IRepository<T>)
  ├─ Generic Repository Implementation (Repository<T>)
  └─ Entity-specific classes (Student, Course, etc.)
```

- **DAL** contains the generic repository and DbContext
- **BLL** consumes the repository through interfaces
- **Presentation layer** calls BLL services

14.3 Creating the Generic Repository Interface

```
namespace MyApp.DAL
```

```
{
    public interface IRepository<T> where T : class
    {
        T GetById(int id);
        List<T> GetAll();
        void Add(T entity);
        void Update(T entity);
        void Delete(T entity);
    }
}
```

- `<T>` allows the repository to work with **any entity type**
- Methods are **standard CRUD operations**

14.4 Implementing the Generic Repository

```
using Microsoft.EntityFrameworkCore;

namespace MyApp.DAL
{
    public class Repository<T> : IRepository<T> where T : class
    {
        private readonly SchoolDbContext _context;
        private readonly DbSet<T> _dbSet;

        public Repository(SchoolDbContext context)
        {
            _context = context;
            _dbSet = context.Set<T>();
        }

        public void Add(T entity) => _dbSet.Add(entity);

        public void Delete(T entity) => _dbSet.Remove(entity);

        public List<T> GetAll() => _dbSet.ToList();

        public T GetById(int id) => _dbSet.Find(id);

        public void Update(T entity) => _dbSet.Update(entity);
    }
}
```

- `DbSet<T>` dynamically maps to any entity type in DbContext
- BLL or services do **not need to know about EF Core** internals

---

14.5 Integrating with BLL

In **BLL**, inject the generic repository:

```
using MyApp.DAL;

namespace MyApp.BLL
{


    public class StudentService
    {
        private readonly Repository<Student> _studentRepo;
        private readonly IMapper _mapper;

        public StudentService(Repository<Student> studentRepo, IMapper mapper)
        {
            _studentRepo = studentRepo;
            _mapper = mapper;
        }

        public void CreateStudent(StudentDTO dto)
        {
            var student = _mapper.Map<Student>(dto);
            _studentRepo.Add(student);
        }

        public IEnumerable<StudentDTO> GetAllStudents()
        {
            var students = _studentRepo.GetAll();
            return _mapper.Map<IEnumerable<StudentDTO>>(students);
        }

        public StudentDTO GetStudent(int id)
        {
            var student = _studentRepo.GetById(id);
            return _mapper.Map<StudentDTO>(student);
        }
    }
}
```

BLL now works with **generic repository interface**, not DbContext directly.

14.6 Dependency Injection

In `Program.cs` of the API project:

```
builder.Services.AddScoped(Repository<>);
builder.Services.AddScoped<StudentService>();
```

- `Repository<>` maps for all entity types

14.7 Advantages of This Approach

1. **DRY principle** → Avoid repeated CRUD code
2. **Consistency** → All entities follow the same CRUD contract
3. **Decoupling** → BLL only depends on interfaces
4. **Testable** → Easy to mock `IRepository<T>` in unit tests
5. **Extensible** → New entities immediately benefit from generic repository

---

14.8 Summary of Generic Repository

In this module, you learned:

- What is **Generic Repository** and why it is useful
- How to define **IRepository<T> interface**
- How to implement **Repository<T> using EF Core DbSet<T>**
- How BLL consumes the repository through **dependency injection**
- Advantages of **DRY, consistency, and decoupling** in a 3-tier architecture

Using the generic repository pattern simplifies **data access across multiple entities** and keeps **BLL clean and focused on business logic**.

Module 15: EF Core – Database-First & Code-First in 3-Tier Architecture

---

15.1 Solution & Project Setup

In a 3-tier setup:

```
MySolution/
├─ App       (Presentation Layer, Startup Project)
├─ BLL       (Business Logic Layer)
├─ DAL       (Data Access Layer, contains DbContext & Entities)
├─ Common    (DTOs, Enums, Shared Models)
```

Key Points:

- **DbContext lives in DAL**.
- **BLL** uses DAL interfaces/services.
- **App** is the **startup project** where migrations are applied.

EF Core tooling must know **where DbContext is** (`--project`) and **which project to use as startup** (`--startup-project`).

15.2 Database-First Approach

Database-First is useful when you already have an existing database.

15.2.1 Install EF Core Packages in DAL

```
Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.EntityFrameworkCore.Design
dotnet tool install --global dotnet-ef
```

These packages go into **DAL**, because DbContext is there.

15.2.2 Scaffold the Database

Suppose database `SchoolDB` exists:

```
dotnet ef dbcontext scaffold "Data Source=DESKTOP-GAKQQ5S\SQLEXPRESS;Initial Catalog=
SchoolDB;TrustServerCertificate=True;Integrated Security=True;"
Microsoft.EntityFrameworkCore.SqlServer --project DAL --startup-project App --context-dir
EF --output-dir EF/Tables
```

Explanation:

- `--output-dir EF/Tables` → output folder inside DAL for entity classes
- `--context-dir EF` → output folder inside DAL for context class
- `--project DAL` → **project containing DbContext**
- `--startup-project App` → **startup project** for connection string & configuration

After this, **DAL contains entities and DbContext**, ready for BLL.

15.2.3 Using DB-First in BLL & API

- BLL receives **DbContext via dependency injection or repository interface**.
- API controllers use BLL services only.
- **No DbContext usage directly in API**.

15.3 Code-First Approach

Code-First is useful when **you want to define entities in code** and generate the database.

15.3.1 Define Entities in DAL

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
}
```

Create **DbContext**:

```
using Microsoft.EntityFrameworkCore;
```

```
public class SchoolDbContext : DbContext
{
    public SchoolDbContext(DbContextOptions<SchoolDbContext> options) : base(options) { }
    public DbSet<Student> Students { get; set; }
}
```

15.3.2 Configure Connection in API Project

In `Program.cs`:

```
builder.Services.AddDbContext<SchoolDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
```

The **API project** provides the startup configuration for EF Core.

15.3.3 Add Migration from API Project

Since **DbContext is in DAL**, you must specify:

```
dotnet ef migrations add InitialCreate --project DAL --startup-project App
```

Explanation:

- `--project DAL` → project where **DbContext** exists
- `--startup-project App` → **startup project** that contains appsettings.json / connection string

---

15.3.4 Update Database

```
dotnet ef database update --project DAL --startup-project App
```

EF Core applies migrations to the database using **API project as startup** while **DAL contains the context**.

15.4 Notes & Best Practices

1. **Separate projects** require careful use of `--project` (DbContext project) and `--startup-project` (Startup project).
2. **DAL** always contains **DbContext and entities**.

3. **BLL** uses **DAL interfaces or services**.
4. **API** is the **startup project** for EF Core commands, and for DI configuration.
5. Store **connection strings** in API project (`appsettings.json`).

15.5 Summary

- **Database-First:** Scaffold from existing database into **DAL**. Use API project as startup.
- **Code-First:** Create entities in **DAL**, generate migrations, and update database via API project as startup.
- **3-tier structure** keeps **DAL, BLL, and API decoupled**.
- EF Core commands must always specify **DbContext project (--project)** and **startup project (--startup-project)**.

Following this approach ensures a **clean, maintainable, and scalable 3-tier architecture**, ready for **repositories, DTO mapping, and service layers**.