below is a complete, ready-to-use **Ticket Management (Booking) Web API** implementation in .NET that includes:

* Booking model
* ApplicationDbContext in dotnetapp.Data
* BookingController implementing GET/POST/PUT/DELETE with proper HTTP status codes
* EF Core setup (DbSet + sample seed data via HasData)
* Swagger with XML comments (so endpoints show documentation & response types)
* appsettings.json sample connection string and .csproj changes needed for XML docs
* Commands to run migrations and the application

You can copy the files into your dotnetapp project structure and run the commands at the end.

**File: Models/Booking.cs**

using System;

using System.ComponentModel.DataAnnotations;

namespace dotnetapp.Models

{

/// <summary>

/// Represents a booking/ticket reservation.

/// </summary>

public class Booking

{

[Key]

public int BookingId { get; set; }

[Required]

[MaxLength(200)]

public string CustomerName { get; set; } = string.Empty;

[Required]

[MaxLength(200)]

public string EventName { get; set; } = string.Empty;

[Required]

public DateTime BookingDate { get; set; }

[Required]

[Range(1, 1000)]

public int NumberOfTickets { get; set; }

}

}

**File: Data/ApplicationDbContext.cs (namespace dotnetapp.Data)**

using Microsoft.EntityFrameworkCore;

using dotnetapp.Models;

using System;

namespace dotnetapp.Data

{

public class ApplicationDbContext : DbContext

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)

: base(options) { }

public DbSet<Booking> Bookings { get; set; } = null!;

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

// Seed static sample data for testing (will be applied with migrations)

modelBuilder.Entity<Booking>().HasData(

new Booking { BookingId = 1, CustomerName = "Alice", EventName = "Rock Concert", BookingDate = new DateTime(2025, 10, 01), NumberOfTickets = 2 },

new Booking { BookingId = 2, CustomerName = "Bob", EventName = "Comedy Night", BookingDate = new DateTime(2025, 11, 05), NumberOfTickets = 3 },

new Booking { BookingId = 3, CustomerName = "Charlie", EventName = "Tech Talk", BookingDate = new DateTime(2025, 12, 15), NumberOfTickets = 1 }

);

base.OnModelCreating(modelBuilder);

}

}

}

**File: Controllers/BookingController.cs**

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using dotnetapp.Data;

using dotnetapp.Models;

namespace dotnetapp.Controllers

{

[ApiController]

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

public class BookingController : ControllerBase

{

private readonly ApplicationDbContext \_db;

public BookingController(ApplicationDbContext db)

{

\_db = db;

}

/// <summary>

/// Get all bookings.

/// </summary>

/// <returns>List of bookings or 204 NoContent if none exist.</returns>

[HttpGet]

[ProducesResponseType(StatusCodes.Status200OK)]

[ProducesResponseType(StatusCodes.Status204NoContent)]

public async Task<IActionResult> GetBookings()

{

var bookings = await \_db.Bookings.ToListAsync();

if (bookings == null || bookings.Count == 0)

return NoContent();

return Ok(bookings);

}

/// <summary>

/// Get booking by id.

/// </summary>

/// <param name="id">Booking id</param>

[HttpGet("{id:int}", Name = "GetBooking")]

[ProducesResponseType(StatusCodes.Status200OK)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

public async Task<IActionResult> GetBooking(int id)

{

var booking = await \_db.Bookings.FindAsync(id);

if (booking == null)

return NotFound(new { Message = $"Booking with id {id} not found." });

return Ok(booking);

}

/// <summary>

/// Create a new booking.

/// </summary>

/// <param name="booking">Booking payload</param>

[HttpPost]

[ProducesResponseType(StatusCodes.Status201Created)]

[ProducesResponseType(StatusCodes.Status400BadRequest)]

public async Task<IActionResult> CreateBooking([FromBody] Booking booking)

{

if (booking == null)

return BadRequest(new { Message = "Booking payload is null." });

if (!ModelState.IsValid)

return BadRequest(ModelState);

\_db.Bookings.Add(booking);

await \_db.SaveChangesAsync();

// CreatedAtAction returns 201 and Location header pointing to GetBooking

return CreatedAtAction(nameof(GetBooking), new { id = booking.BookingId }, booking);

}

/// <summary>

/// Update an existing booking.

/// </summary>

/// <param name="bookingId">Booking id</param>

/// <param name="updatedBooking">Updated booking payload</param>

[HttpPut("{bookingId:int}")]

[ProducesResponseType(StatusCodes.Status204NoContent)]

[ProducesResponseType(StatusCodes.Status400BadRequest)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

public async Task<IActionResult> UpdateBooking(int bookingId, [FromBody] Booking updatedBooking)

{

if (updatedBooking == null)

return BadRequest(new { Message = "Booking payload is null." });

if (bookingId != updatedBooking.BookingId)

return BadRequest(new { Message = "Route id and payload id do not match." });

var existing = await \_db.Bookings.FindAsync(bookingId);

if (existing == null)

return NotFound(new { Message = $"Booking with id {bookingId} not found." });

// Update properties

existing.CustomerName = updatedBooking.CustomerName;

existing.EventName = updatedBooking.EventName;

existing.BookingDate = updatedBooking.BookingDate;

existing.NumberOfTickets = updatedBooking.NumberOfTickets;

try

{

await \_db.SaveChangesAsync();

}

catch (DbUpdateConcurrencyException)

{

// If record disappeared between read and update

if (!await \_db.Bookings.AnyAsync(b => b.BookingId == bookingId))

return NotFound(new { Message = $"Booking with id {bookingId} not found (concurrency)." });

else

throw;

}

return NoContent();

}

/// <summary>

/// Delete a booking.

/// </summary>

/// <param name="bookingId">Booking id</param>

[HttpDelete("{bookingId:int}")]

[ProducesResponseType(StatusCodes.Status204NoContent)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

public async Task<IActionResult> DeleteBooking(int bookingId)

{

var existing = await \_db.Bookings.FindAsync(bookingId);

if (existing == null)

return NotFound(new { Message = $"Booking with id {bookingId} not found." });

\_db.Bookings.Remove(existing);

await \_db.SaveChangesAsync();

return NoContent();

}

}

}

**File: Program.cs (for .NET 6/7+)**

using Microsoft.EntityFrameworkCore;

using Microsoft.OpenApi.Models;

using dotnetapp.Data;

var builder = WebApplication.CreateBuilder(args);

// Add services

var connectionString = builder.Configuration.GetConnectionString("DefaultConnection")

?? "User ID=sa;password=examlyMssql@123; server=localhost;Database=TicketDB;trusted\_connection=false;Persist Security Info=False;Encrypt=False";

builder.Services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(connectionString));

builder.Services.AddControllers()

.AddJsonOptions(opts => opts.JsonSerializerOptions.PropertyNamingPolicy = null);

builder.Services.AddEndpointsApiExplorer();

// Swagger with XML comments

builder.Services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Title = "Ticket Management API", Version = "v1", Description = "API for managing ticket bookings" });

// Set the comments path for the Swagger JSON and UI.

var xmlFile = $"{System.Reflection.Assembly.GetExecutingAssembly().GetName().Name}.xml";

var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);

if (File.Exists(xmlPath))

c.IncludeXmlComments(xmlPath);

});

var app = builder.Build();

// Apply pending migrations automatically in development (optional)

using (var scope = app.Services.CreateScope())

{

var db = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();

// Note: For production you should manage migrations explicitly.

db.Database.Migrate();

}

if (app.Environment.IsDevelopment() || true) // keep swagger available for test environment

{

app.UseSwagger();

app.UseSwaggerUI(c =>

{

c.RoutePrefix = "swagger"; // swagger/index available at /swagger/index.html

c.SwaggerEndpoint("/swagger/v1/swagger.json", "Ticket Management API v1");

});

}

app.UseHttpsRedirection();

app.MapControllers();

app.Run();

**File: appsettings.json (sample)**

{

"ConnectionStrings": {

"DefaultConnection": "User ID=sa;password=examlyMssql@123; server=localhost;Database=TicketDB;trusted\_connection=false;Persist Security Info=False;Encrypt=False"

},

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

}

}

**.csproj changes (enable XML docs for Swagger)**

Add inside the <PropertyGroup> of your .csproj:

<GenerateDocumentationFile>true</GenerateDocumentationFile>

<NoWarn>$(NoWarn);1591</NoWarn>

This ensures XML comments are generated and included in Swagger. 1591 suppresses missing XML comments warnings.

**Migration & EF Core tool notes (as requested)**

If you don't already have the EF Core CLI locally in the repo, follow the commands in your prompt (example for EF tool manifest local install):

cd dotnetapp

dotnet new tool-manifest # if you don't have one already

dotnet tool install --local dotnet-ef --version 6.0.6

# Add migration

dotnet dotnet-ef migrations add InitialSetup

# Apply migration to create DB and seed data

dotnet dotnet-ef database update

If you prefer dotnet-ef global install you can use that instead.

**Commands to run the project (summary)**

cd dotnetapp

dotnet restore

dotnet build

dotnet run

Open Swagger UI at:

http://localhost:8080/swagger/index.html

# or if running locally with default Kestrel port: http://localhost:<port>/swagger/index.html

(Make sure port settings match your environment. If your environment pre-selects port 8080, Kestrel will serve there.)

**Notes & Tips**

* I seeded static data using HasData so your GET endpoints are non-empty immediately after database update.
* Program.cs auto-applies migrations during startup (convenient in dev/test). In production prefer explicit migration deployment.
* The controller returns:
  + 200 OK with data when present
  + 204 NoContent when list is empty (GET all)
  + 404 NotFound when entity not found
  + 201 Created for new booking via CreatedAtAction
  + 400 BadRequest for invalid payloads or id mismatches
  + 204 NoContent for successful update/delete
* Swagger XML comments were added to controller & model so the UI shows summaries, parameter types, and response codes.
* Keep the appsettings.json connection string safe; for local dev the provided sample uses sa credentials per your note — in shared environments use secrets or environment variables.

If you want, I can:

* Add unit tests (xUnit) and sample dotnet test testcases for the controller using InMemoryDatabase.
* Provide a Postman collection or sample curl commands for each endpoint.
* Convert the project to use an in-memory EF provider for easier local test runs (no SQL Server needed).

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using dotnetapp.Data;

using dotnetapp.Models;

namespace dotnetapp.Controllers

{

[ApiController]

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

public class BookingController : ControllerBase

{

private readonly ApplicationDbContext \_db;

public BookingController(ApplicationDbContext db)

{

\_db = db;

}

// GET: api/Booking

[HttpGet]

public async Task<IActionResult> GetBookings()

{

var bookings = await \_db.Bookings.ToListAsync();

if (bookings == null || bookings.Count == 0)

return NoContent();

return Ok(bookings);

}

// GET: api/Booking/{id}

[HttpGet("{id:int}")]

public async Task<IActionResult> GetBooking(int id)

{

var booking = await \_db.Bookings.FindAsync(id);

if (booking == null)

return NotFound();

return Ok(booking);

}

// POST: api/Booking

[HttpPost]

public async Task<IActionResult> CreateBooking([FromBody] Booking booking)

{

if (booking == null)

return BadRequest();

\_db.Bookings.Add(booking);

await \_db.SaveChangesAsync();

return CreatedAtAction(nameof(GetBooking), new { id = booking.BookingId }, booking);

}

// PUT: api/Booking/{bookingId}

[HttpPut("{bookingId:int}")]

public async Task<IActionResult> UpdateBooking(int bookingId, [FromBody] Booking updatedBooking)

{

if (updatedBooking == null)

return BadRequest();

// The tests for UpdateBooking\_ValidId and UpdateBooking\_InvalidId

// expect NOT to check payload ID equality — remove that condition.

var existing = await \_db.Bookings.FindAsync(bookingId);

if (existing == null)

return NotFound();

// Update entity fields

existing.CustomerName = updatedBooking.CustomerName;

existing.EventName = updatedBooking.EventName;

existing.BookingDate = updatedBooking.BookingDate;

existing.NumberOfTickets = updatedBooking.NumberOfTickets;

await \_db.SaveChangesAsync();

// Return exactly 204 NoContent for successful update

return NoContent();

}

// DELETE: api/Booking/{bookingId}

[HttpDelete("{bookingId:int}")]

public async Task<IActionResult> DeleteBooking(int bookingId)

{

var existing = await \_db.Bookings.FindAsync(bookingId);

if (existing == null)

return NotFound();

\_db.Bookings.Remove(existing);

await \_db.SaveChangesAsync();

return NoContent();

}

}

}