Use Identity to secure a Web API backend for SPAs | Microsoft Learn https://share.google/4GYTMOXfvyHY6SUz9

* With IEnumerable<T>, all filtering and processing happens in memory
* With IQueryable<T>, filtering can happen at the database level

Eg:

1.var employees = \_context.Employees.ToList(); // Executes query, loads all employees into memory

var result = employees.Where(e => e.Age > 30); // Filter happens in memory

All employees are loaded from the database first → inefficient if table is large

2.IQueryable<Employee> employees = \_context.Employees; // Queryable

var result = employees.Where(e => e.Age > 30); // Translates to SQL

var list = result.ToList(); // Executes query in database, returns only filtered rows

EF Core converts the LINQ expression to SQL

Only the filtered rows are sent from database → much more efficient

ICollection<T> → can iterate + add/remove + count items

* SqlConnection - Creates connection to the SQL Server database
* SqlCommand - Will execute the stored procedure or query in the database
* cmd.ExecuteReaderAsync() - Will execute the command and return the data to SqlDataReader
* A screenshot of a computer

  AI-generated content may be incorrect.SqlDataReader - Has the data and .Read() will read the table row by row

The using statement automatically disposes (cleans up) objects when they're no longer needed, even if an exception occurs.

Key Features of DbContext

🔑 Key Features of DbContext in Entity Framework Core

1. 📡 Database Connection Management
   * Manages database connections automatically (open/close).
   * You don’t need to manually handle SqlConnection or SqlCommand.
   * Uses connection strings defined in appsettings.json.
2. 🧱 CRUD Operations (Create, Read, Update, Delete)
   * Allows you to perform CRUD without writing raw SQL.
   * Example:
   * context.Employees.Add(employee); // Create
   * context.Employees.Find(id); // Read
   * context.Employees.Update(employee); // Update
   * context.Employees.Remove(employee); // Delete
   * context.SaveChanges(); // Commit changes
3. 🧠 Change Tracking
   * Keeps track of changes made to entities while they are in memory.
   * When you call SaveChanges(), it automatically generates the right SQL commands.
   * Example: If you modify an entity’s property, EF knows which field changed and updates only that column.
4. 🏗️ Database Creation and Migration
   * Can create a database automatically based on your entity classes.
   * Supports code-first migrations to keep your schema in sync with your models.
   * Example commands:
   * dotnet ef migrations add InitialCreate
   * dotnet ef database update
5. 🧮 LINQ Query Support
   * You can query data using LINQ (Language Integrated Query) instead of SQL.
   * EF Core translates LINQ into SQL under the hood.
   * Example:
   * var employees = context.Employees.Where(e => e.Age > 25).ToList();
6. 🔗 Relationship Management
   * Handles relationships between entities (One-to-Many, Many-to-Many, etc.).
   * Automatically manages foreign keys, navigation properties, and cascading deletes.
   * Example:
   * var department = context.Departments.Include(d => d.Employees).FirstOrDefault();
7. 💾 Caching (First-Level Cache)
   * Tracks and caches entities within the same DbContext instance.
   * So multiple queries for the same entity in a single context won’t hit the database again.
8. 🧰 Interception and Logging (optional)
   * You can log SQL queries, intercept commands, or customize query execution.
   * Example:

optionsBuilder.LogTo(Console.WriteLine);

**🧩 1. SqlConnection — *Low-Level, Manual Control***

**💬 What it is**

SqlConnection is part of **ADO.NET** — it’s a **low-level** way to connect to SQL Server directly.

**⚙️ You do things manually:**

* Open/close the connection yourself.
* Write SQL queries or call stored procedures manually.
* Use SqlCommand, SqlDataReader, etc. to read/write data.

**🧠 Example:**

using (SqlConnection connection = new SqlConnection(\_connectionString))

{

connection.Open();

SqlCommand cmd = new SqlCommand("SELECT \* FROM Employees", connection);

SqlDataReader reader = cmd.ExecuteReader();

while (reader.Read())

{

Console.WriteLine(reader["Name"]);

}

}

**✅ Pros:**

* Full control over SQL commands.
* You can use **stored procedures** easily.
* Better performance for **complex or optimized SQL**.

**❌ Cons:**

* More code to write (boilerplate).
* No automatic change tracking or model mapping.
* Harder to maintain and test.

**🧩 2. DbContext — *High-Level, Object-Oriented Control***

**💬 What it is**

DbContext comes from **Entity Framework (EF) Core**, which is an **ORM (Object Relational Mapper)**.  
It maps your **C# classes** directly to **database tables**.

**⚙️ What it does automatically:**

* Opens and closes the database connection.
* Tracks changes to your entities.
* Converts C# LINQ queries to SQL automatically.
* Handles insert/update/delete without writing SQL manually.

**🧠 Example:**

var employees = await \_context.Employees.ToListAsync();

var employee = new Employee { Name = "John", Age = 30, Department = "IT" };

\_context.Employees.Add(employee);

await \_context.SaveChangesAsync();

**✅ Pros:**

* Less boilerplate code.
* No need to write SQL.
* Automatically tracks changes.
* Easier to maintain and test.

**❌ Cons:**

* Slightly slower for very complex queries.
* Harder to optimize for performance-heavy apps.
* Not great if you rely fully on stored procedures.

🧱 **SqlConnection** = You control everything.  
🧠 **DbContext** = Entity Framework controls everything.

**How SSL/TLS encryption works between browser and server**

1. **Server sends its SSL certificate to the browser**
   * The certificate contains the **server’s public key** (not the private key).
   * The **private key** always stays on the server and is never shared.
2. **Browser verifies the certificate**
   * Checks that the certificate is issued by a trusted Certificate Authority (CA).
   * Confirms that it really belongs to the website.
3. **Browser and server agree on a session key** (for symmetric encryption)
   * Here’s the subtle part: SSL/TLS **doesn’t usually encrypt all data directly with the public/private key** because asymmetric encryption is slow.
   * Instead, the browser generates a **random session key** and encrypts it using the **server’s public key** from the certificate.
   * The server decrypts it using its **private key**.
4. **Data encryption during the session**
   * Now the browser and server use that **session key** to encrypt all further communication with **fast symmetric encryption**.
   * This ensures your credit card info and other data are secure while traveling over the internet.

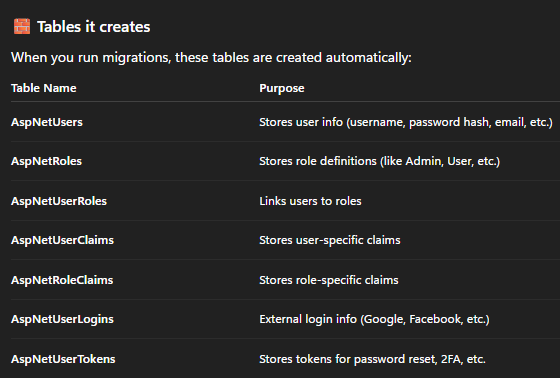
IDENTITY

When you use IdentityDbContext, it:

* Adds all the **Identity-related DbSets** automatically (like AspNetUsers, AspNetRoles, etc.).
* Configures their **relationships and schema**.
* Handles **migrations** for you when you run dotnet ef migrations add ....

So you don’t have to manually define DbSets or relationships for these tables — IdentityDbContext does that internally.

<https://andrewlock.net/exploring-the-dotnet-8-preview-introducing-the-identity-api-endpoints/>



**1. AddAuthorization()** Sets up the authorization system that controls what authenticated users can do. It enables you to use authorization policies, role-based access, and the [Authorize] attribute to protect endpoints.

**2. AddDbContext<AppDbContext>** Registers Entity Framework Core with SQLite as the database provider. This allows your application to:

* Use dependency injection to access the database context
* Connect to a SQLite database using the connection string named "DefaultConnection" from your configuration (appsettings.json)

**3. AddIdentityApiEndpoints<AppUser>().AddEntityFrameworkStores<AppDbContext>()** Configures ASP.NET Core Identity for user authentication and management:

* AddIdentityApiEndpoints<AppUser>() - Sets up built-in API endpoints for user registration, login, logout, token refresh, and user management (like /register, /login, etc.)
* AddEntityFrameworkStores<AppDbContext>() - Tells Identity to use your Entity Framework database context to store user data (users, passwords, roles, etc.)

app.MapIdentityApi<AppUser>() automatically registers a set of HTTP endpoints for user authentication and management. Here's what it does under the hood:

**It creates these API endpoints:**

1. **POST /register** - Creates a new user account
   * Validates email and password
   * Hashes the password securely
   * Stores user in the database
   * Returns success/error response
2. **POST /login** - Authenticates a user
   * Validates credentials
   * Generates a JWT access token (short-lived, ~15 minutes)
   * Generates a refresh token (long-lived, ~14 days)
   * Returns both tokens
3. **POST /refresh** - Refreshes an expired access token
   * Takes a refresh token
   * Validates it
   * Issues a new access token
4. **GET /confirmEmail** - Email confirmation endpoint
   * Confirms user's email address (if email confirmation is enabled)
5. **POST /resendConfirmationEmail** - Resends confirmation email
6. **POST /forgotPassword** - Initiates password reset
7. **POST /resetPassword** - Completes password reset
8. **POST /manage/2fa** - Two-factor authentication endpoints
9. **GET /manage/info** - Gets current user info (requires authentication)
10. **POST /manage/info** - Updates user info (requires authentication)

**In essence:** It's a shortcut that saves you from writing all this authentication logic manually. Instead of creating controllers and actions for register, login, token generation, password hashing, etc., this single line gives you a complete, production-ready authentication system that follows security best practices.

app.MapGroup("/account").MapIdentityApi<AppUser>();

api become /account/register

**1️⃣ builder.Services.AddIdentityApiEndpoints<User>()**

👉 **Registers all Identity services** (user management, password hashing, sign-in manager, token generator, etc.)  
and **adds built-in minimal API endpoints** for authentication.

**💡 What it does**

* Registers Identity core types:
  + UserManager<User>
  + SignInManager<User>
  + RoleManager<Role> (if used)
* Adds **ready-made routes** like:
  + /register
  + /login
  + /logout
  + /refresh
  + /manage/info
* Sets up **JWT token handling** for authentication APIs.

🧠 Think of this line as: “Turn on Identity system and prepare endpoints for login, register, etc.”

**🧩 2️⃣ .AddEntityFrameworkStores<AppDbContext>()**

👉 **Tells Identity where to store user data** — i.e., in your **AppDbContext** database using **Entity Framework Core**.

**💡 What it does**

* Registers **AppDbContext** as the database context that holds Identity tables.
* Enables Identity to **automatically create and use tables** like:
  + AspNetUsers
  + AspNetRoles
  + AspNetUserRoles
  + AspNetUserTokens
  + AspNetUserClaims
  + AspNetRoleClaims
* All user info, passwords, tokens, and roles get saved here.

🧠 Think of this line as: “Store all Identity users and roles in the database using EF Core.”

**🧩 3️⃣ app.MapIdentityApi<User>()**

👉 Actually **creates and maps the routes (endpoints)** for Identity into your app.

**💡 What it does**

* Exposes ready-to-use endpoints like:
  + POST /register
  + POST /login
  + POST /logout
  + POST /refresh
  + GET /manage/info
* These are the routes the frontend (like React) calls to handle user registration and login.

🧠 Think of this line as: “Make the login/register/logout routes available in your API.”