ASP.NET Core - True Ultimate Guide

Section 18: EntityFrameworkCore

Entity Framework Core

EF Core is a modern, lightweight, and extensible Object-Relational Mapper (ORM) framework for .NET. It simplifies database interactions by allowing you to work with data as .NET objects (entities) rather than raw SQL queries. This abstraction makes your code more maintainable, readable, and productive.

How EF Core Works

1. **Entities and DbContext:** You define classes that represent your database tables (entities) and a DbContext class that acts as a bridge between your entities and the database.
2. **Mapping:** EF Core handles the mapping between your entities and the database tables, including column names, data types, relationships, and constraints.
3. **Querying and Saving:** You use LINQ (Language Integrated Query) to query your data and interact with your entities. EF Core translates your LINQ queries into efficient SQL statements and executes them against the database. You can also add, update, and delete entities, and EF Core takes care of persisting these changes to the database.

Pros of EF Core

- **Developer Productivity:** Reduced boilerplate code for database interactions, allowing you to focus on your application's business logic.
- **Object-Oriented Approach:** Work with data using familiar object-oriented concepts, making your code more intuitive and easier to reason about.
- **Strongly Typed Queries:** LINQ provides compile-time type safety for your queries, reducing the risk of runtime errors.
- **Cross-Platform:** EF Core is cross-platform, supporting various database providers (SQL Server, SQLite, PostgreSQL, MySQL, etc.).
- **Automatic Change Tracking:** EF Core keeps track of changes made to entities, making it easy to persist those changes to the database.
- **Migrations:** The migrations feature simplifies database schema evolution, allowing you to incrementally update your database as your application's model changes.

Cons of EF Core

- **Abstraction Overhead:** The abstraction layer introduced by EF Core can sometimes lead to less optimized SQL queries compared to hand-written SQL. However, you can often mitigate this by understanding EF Core's behavior and using techniques like raw SQL queries or stored procedures when necessary.
- **Learning Curve:** While EF Core simplifies many aspects of data access, there is still a learning curve to understand its concepts and best practices.

NuGet Packages for EF Core

- **Microsoft.EntityFrameworkCore:** The core package containing the essential functionality of EF Core.
- **Microsoft.EntityFrameworkCore.SqlServer:** Database provider for SQL Server.
- **Microsoft.EntityFrameworkCore.Sqlite:** Database provider for SQLite.
- **Microsoft.EntityFrameworkCore.InMemory:** An in-memory database provider, primarily used for testing.
- **Microsoft.EntityFrameworkCore.Design:** Tools for working with migrations and scaffolding.
- **Microsoft.EntityFrameworkCore.Tools:** The dotnet ef command-line tools for managing migrations and database operations.
- **Other Providers:** There are also database providers for PostgreSQL, MySQL, and other databases.

Choosing a Database Provider

The choice of database provider depends on your project's requirements:

- **SQL Server:** A popular choice for enterprise applications with robust features and scalability.
- **SQLite:** A lightweight, file-based database suitable for small to medium-sized applications or embedded scenarios.
- **PostgreSQL:** A powerful open-source relational database known for its extensibility and standards compliance.
- **MySQL:** Another open-source relational database popular for web applications.
- **InMemory:** Ideal for testing and scenarios where you don't need data persistence.

Notes

- **ORM:** EF Core is an Object-Relational Mapper that simplifies database interaction.
- **Core Concepts:** Entities (DbContext), mapping, querying with LINQ, change tracking, migrations.
- **Pros:** Productivity, object-oriented, type safety, cross-platform, migrations.
- **Cons:** Potential for abstraction overhead, learning curve.
- **Packages:** The core package, database providers, design-time tools.
- **Choose the Right Database:** Consider factors like scalability, features, licensing, and your team's expertise when selecting a database and provider.

EF Core Architecture: A Three-Layer Approach

1. **Conceptual Model (Entity Model):**
    - This is your C# code representation of the database schema. You define entity classes that represent your database tables, along with their properties (columns) and relationships between entities.
    - The entity classes form the heart of your domain model, reflecting the real-world concepts your application deals with.
2. **Mapping:**
    - EF Core handles the mapping between your entity classes and the underlying database schema. This includes mapping property names to column names, data types, relationships (foreign keys), and constraints.
    - You can customize this mapping using fluent APIs or data annotations in your entity classes.
3. **Storage Model (Database Schema):**
    - This is the actual structure of your database (tables, columns, relationships). EF Core can either generate the database schema based on your entity model or work with an existing database.

How EF Core Works: A Simplified View

1. **DbContext:**
    - You create a DbContext class that acts as a session with the database. This class is responsible for tracking entity changes, managing transactions, and translating LINQ queries into SQL commands.

    - Think of it as a bridge between your C# code and the database.
2. **Querying:**
    - You write LINQ queries against your DbContext to fetch data from the database.
    - EF Core translates these LINQ queries into optimized SQL queries and executes them against the database.
    - It then materializes the results into your entity objects, which you can work with in your application.
3. **Saving Changes:**
    - When you modify entities in your code, EF Core tracks those changes.
    - When you call SaveChanges() on your DbContext, EF Core generates SQL commands to update the database based on the tracked changes.
    - This includes inserts, updates, and deletes to keep your database synchronized with your entity objects.

EF Core Approaches: Which One to Choose?

1. **Code First:**
    - You start by defining your entity classes, and EF Core creates the database schema based on those classes.
    - Use this approach when you are starting from scratch or have full control over your database schema.
    - It's flexible and well-suited for rapid development.
2. **Database First:**
    - You start with an existing database, and EF Core generates entity classes based on the schema.
    - Use this approach when you have a legacy database that you need to integrate with.
    - It can save time initially but may require manual adjustments to the generated code.
3. **Model First (Not in EF Core):**
    - This approach involves designing a visual model (EDMX) of your database schema, and EF generates the code from the model.
    - While it was available in earlier versions of Entity Framework, it's not supported in EF Core.

Notes

- **ORM:** EF Core is an Object-Relational Mapper, bridging the gap between your code and the database.
- **Key Components:** DbContext, entity classes, LINQ queries, SaveChanges().
- **Approaches:** Choose between Code First and Database First based on your project's starting point and requirements.
- **Benefits:** Increased productivity, type safety, simplified data access, and cross-platform compatibility.

DbContext

In EF Core, the DbContext class serves as a central hub for your database interaction. Think of it as a session with your database. It's responsible for:

1. **Connecting to the Database:** The DbContext establishes the connection to your database using the connection string you provide in your configuration.
2. **Managing Entities:** The DbContext tracks changes made to entity instances, manages their lifecycle (adding, deleting, updating), and coordinates the persistence of those changes back to the database.
3. **Querying Data:** You use LINQ (Language-Integrated Query) to formulate queries against your entities through the DbContext. EF Core then translates these queries into SQL statements, executes them against the database, and materializes the results into your entity objects.
4. **Change Tracking:** EF Core automatically tracks changes you make to your entities in memory. When you call SaveChanges(), EF Core detects these changes and generates the appropriate SQL commands (INSERT, UPDATE, DELETE) to persist them to the database.

DbSet: Your Entity Collection

DbSet<TEntity> represents a collection of a specific entity type within your DbContext. It exposes methods for querying, adding, updating, and deleting entities of that type.

- **Mapping:** Each DbSet property in your DbContext is mapped to a corresponding table in your database. EF Core takes care of this mapping based on conventions or explicit configurations you provide.

- **Usage:** You interact with the database through your DbSet properties. For example, context.Persons.Add(newPerson); would add a new Person entity to the database.

Code Example: PersonsDbContext

```
// PersonsDbContext.cs
using System;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;
using Entities;

namespace Entities
{
    public class PersonsDbContext : DbContext
    {
        public DbSet<Country> Countries { get; set; }
        public DbSet<Person> Persons { get; set; }

        // OnModelCreating is used to customize your model mappings, but for this example we won't change anything.
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            modelBuilder.Entity<Country>().ToTable("Countries");
            modelBuilder.Entity<Person>().ToTable("Persons");
        }
    }
}
```

In this example:

- PersonsDbContext derives from DbContext.

- Countries and Persons are DbSet properties representing the Country and Person entities respectively.
- OnModelCreating is overridden to customize the mapping between your entities and database tables (though we aren't customizing anything in this case).

Notes

- **DbContext:**
    - Represents a session with your database.
    - Handles entity management, change tracking, querying, and saving changes.
    - Often injected as a scoped service in your ASP.NET Core application.
- **DbSet:**
    - Represents a collection of a specific entity type.
    - Provides methods for querying, adding, updating, and deleting entities.
    - Each DbSet is mapped to a corresponding table in your database.

Important Considerations

- **Entity Classes:** Your entity classes define the shape of your data and should align with your domain model.
- **Data Annotations and Fluent API:** Use these techniques to configure the mapping between your entities and database tables.
- **Relationships:** EF Core supports relationships between entities (one-to-one, one-to-many, many-to-many), and you can define them using navigation properties or fluent API configuration.
- **Connection String:** Ensure you provide the correct connection string in your appsettings.json (or environment variables) so EF Core knows how to connect to your database.

Connection Strings

In the world of databases, a connection string is essentially the address your application uses to locate and connect to your database server. It contains vital information like:

- **Data Source (Server):** The name or IP address of the database server.
- **Initial Catalog (Database Name):** The specific database on the server you want to connect to.
- **Credentials (Optional):** If your database requires authentication, you'll include the username and password.
- **Additional Settings:** Options like connection timeout, encryption settings, and more.

SQL Server Connection String Format (Example)

Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=PersonsDatabase;Integrated Security=True;Connect Timeout=30;Encrypt=False;TrustServerCertificate=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False

- **Data Source=(localdb)\MSSQLLocalDB:** Specifies the server name. In this case, it's using the local SQL Server Express LocalDB instance.
- **Initial Catalog=PersonsDatabase:** Indicates that the database to connect to is named "PersonsDatabase."
- **Integrated Security=True:** Uses Windows authentication (the application's identity) to connect to the database.
- **Connect Timeout=30:** Sets the maximum time (in seconds) to wait for a connection to be established.
- **Encrypt=False, TrustServerCertificate=False:** Options related to encryption and certificate validation (can be set to true for production environments).
- **ApplicationIntent=ReadWrite:** Specifies the intended use of the connection (read/write in this case).
- **MultiSubnetFailover=False:** Relates to high availability scenarios (not relevant for most basic setups).

Storing Connection Strings in ASP.NET Core

1. **appsettings.json (Recommended):**
    - The preferred location for storing your connection string (and other configuration settings).
    - It's organized by sections (like "ConnectionStrings"):

```
{
  "ConnectionStrings": {
```

```
  "DefaultConnection": "..." // Your connection string here
 }
}
```

2. **Environment Variables:**
   - More secure for sensitive information, as environment variables are not stored in code.
   - Use the prefix ConnectionStrings__ for your connection string environment variable:

```
Bash
set ASPNETCORE_ConnectionStrings__DefaultConnection="..."  // In Command Prompt
$env:ASPNETCORE_ConnectionStrings__DefaultConnection = "..." // In PowerShell
```

3. **User Secrets (Development Only):**
   - Best for keeping sensitive information out of your source code during development.
   - Use the dotnet user-secrets commands to manage them.

Injecting and Using the Connection String in EF Core

```
// Program.cs
builder.Services.AddDbContext<PersonsDbContext>(options => {
  options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"));
});
```

- **AddDbContext<PersonsDbContext>():** Registers your DbContext with the DI container and configures it.
- **options.UseSqlServer(...):** Specifies that you're using SQL Server and provides the connection string.
- **builder.Configuration.GetConnectionString("DefaultConnection"):** Retrieves the connection string from the "ConnectionStrings:DefaultConnection" key in the configuration.

Best Practices

- **Separate Environments:** Store different connection strings for development, staging, and production environments (e.g., appsettings.Development.json, appsettings.Production.json).

- **Environment Variables in Production:** Use environment variables to store your production connection string for security.
- **User Secrets in Development:** Use user secrets during development to keep sensitive data out of source control.
- **Secure Storage:** Consider using Azure Key Vault or other secret management solutions for production environments.
- **Connection Resiliency:** For production, implement connection resiliency strategies to handle transient database errors.

Seed Data in EF Core

Seed data refers to the initial data that you populate your database with when it's created or when you apply a new migration. It's a crucial aspect of setting up a meaningful development or testing environment, providing sample data to work with or establishing default values for certain records.

Purpose

- **Initial Data:** Sets up your database with meaningful data for development, testing, or demonstration purposes.
- **Reference Data:** Populates tables with lookup data (e.g., countries, states, categories).
- **Default Values:** Establishes default values for specific columns.

How Seed Data Works in EF Core

1. **OnModelCreating Method:** You define your seed data within the OnModelCreating method of your DbContext class. This method is called when EF Core builds your model.
2. **HasData Method:** EF Core provides the HasData method on the EntityTypeBuilder class, which allows you to associate seed data with specific entities.
   - This method can be used with Code First or with Migrations (where the seed data is automatically integrated into your migration scripts).
3. **Migration Generation (Optional):** If you're using migrations, EF Core will automatically detect your seed data and generate the necessary SQL statements to insert the data into your database when you apply the migration.
4. **Direct Seeding (Optional):** You can also choose to directly seed your database by calling EnsureCreated on your DbContext or using custom code to insert the data.

Code Example

```
// PersonsDbContext.cs
using System;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;
using Entities;

namespace Entities
{
  public class PersonsDbContext : DbContext
  {
    public PersonsDbContext(DbContextOptions options) : base(options) { }
    public DbSet<Country> Countries { get; set; }
    public DbSet<Person> Persons { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      base.OnModelCreating(modelBuilder);

      modelBuilder.Entity<Country>().ToTable("Countries");
      modelBuilder.Entity<Person>().ToTable("Persons");

      // Seed Data for Countries
      string countriesJson = System.IO.File.ReadAllText("countries.json"); // Read country data
      List<Country> countries =
System.Text.Json.JsonSerializer.Deserialize<List<Country>>(countriesJson);
      foreach (Country country in countries)
```

```
            {
                modelBuilder.Entity<Country>().HasData(country); // Add each country as seed data
            }


            // Seed Data for Persons
            string personsJson = System.IO.File.ReadAllText("persons.json"); // Read person data
            List<Person> persons =
System.Text.Json.JsonSerializer.Deserialize<List<Person>>(personsJson);


            foreach (Person person in persons)
            {
                modelBuilder.Entity<Person>().HasData(person); // Add each person as seed data
            }
        }
    }
}
```

In this code:

- Countries and persons are read from external json files and seeded to database using HasData.
- HasData takes object of the specified entity type as parameter.

Best Practices

- **Separate Seed Data:** Store seed data in separate files (e.g., JSON, CSV) to keep your DbContext clean and maintainable.
- **Idempotent Seeds:** Design your seed data so that it can be applied multiple times without causing errors or duplicates.
- **Conditional Seeding:** Consider using environment checks to apply different seed data based on the environment (e.g., more comprehensive data for development).
- **Order of Seeding:** If you have relationships between entities, ensure the correct seeding order to avoid foreign key constraint violations.
- **Large Datasets:** For very large datasets, consider using bulk insert mechanisms for performance optimization.

Code First Migrations

Code First Migrations in EF Core provide a structured and automated way to manage changes to your database schema as your application evolves. They bridge the gap between your code-first entity models (C# classes) and the database structure, allowing you to keep them synchronized over time.

Purpose

- **Track Changes:** Migrations track the changes you make to your entity classes and generate migration scripts (C# code) that represent those changes.
- **Version Control:** Each migration has a unique version and can be tracked in source control, providing a clear history of your schema modifications.
- **Apply Changes:** You can use the dotnet ef command or the Package Manager Console to apply migrations to your database, updating the schema accordingly.
- **Rollbacks:** Migrations can be rolled back if needed, undoing previous schema changes.

When to Use Code First Migrations

- **Code First Approach:** Use migrations if you are following the Code First approach in EF Core, where you define your database model using C# classes and let EF Core create the database based on that model.
- **Evolving Schema:** Whenever you make changes to your entity classes (add properties, rename tables, modify relationships), you'll use migrations to apply those changes to your database.
- **Team Environments:** Migrations are essential for team collaboration, as they allow everyone to keep their databases in sync with the evolving codebase.

Using Code First Migrations with the Package Manager Console

1. **Enable Migrations:**

   ```
   Enable-Migrations
   ```

   This command is usually only needed once to set up the migrations infrastructure in your project.

2. **Add a Migration:**

```
Add-Migration <MigrationName>
```

- Replace <MigrationName> with a descriptive name for your migration (e.g., "AddProductsTable").
- EF Core will analyze your model changes and generate a migration file in the Migrations folder. This file contains Up and Down methods to apply and revert the changes, respectively.

3. **View Migration Details:**

   ```
   Get-Migrations
   ```

   Lists all the migrations in your project, their versions, and whether they have been applied to the database.

4. **Update the Database:**

   ```
   Update-Database
   ```

   Applies any pending migrations to your database, bringing it up to date with your model.

5. **Revert a Migration:**

   ```
   Update-Database –TargetMigration <MigrationName>
   ```

   Rolls back the database to the specified migration.

Important Considerations:

- **Backup Your Database:** Always make a backup before applying migrations, especially in production environments.
- **Seed Data:** If you need to seed data, include that logic in your migrations or create a separate seeder class.
- **Complex Changes:** For complex schema changes that can't be easily automated by migrations, consider writing custom SQL scripts or using tools like the Entity Framework Power Tools.
- **Version Control:** Check in your migration files into source control so that the entire team can track schema changes.
- **Naming Conventions:** Use clear and descriptive names for your migrations to make it easier to understand the history of your database schema.

Best Practices

- **Small, Focused Migrations:** Create small, incremental migrations that focus on a single feature or change. This makes them easier to understand, review, and potentially rollback if needed.

- **Descriptive Names:** Use clear and descriptive names for your migrations to reflect the changes they contain (e.g., "AddProductTable," "RenameCustomerColumn").
- **Data Preservation:** When possible, design your migrations in a way that preserves existing data in the database. Avoid destructive changes like dropping tables or columns if you can modify them instead.
- **Review Migrations:** Always review the generated migration code before applying it to your database, especially in production environments. Make sure the changes are what you expect.
- **Automate in CI/CD:** Integrate migration execution into your continuous integration and deployment (CI/CD) pipeline to ensure that your database schema stays in sync with your application code.

Controller Actions

- **Index (Read):**
  - **Purpose:** Displays a list of Person entities, optionally filtered and sorted.
  - **Key Steps:**
    1. **Retrieval:** Gets filtered and sorted person data from the PersonsService.
    2. **ViewBag Preparation:**
       - Populates ViewBag with search field options (SearchFields), the currently applied search filter (CurrentSearchBy, CurrentSearchString), and sorting information (CurrentSortBy, CurrentSortOrder).
    3. **View Rendering:** Returns the Index view, passing the filtered and sorted person data as the model.

- **Create (Create):**
  - **Purpose:** Handles both the display of the create form (GET) and the processing of the form submission (POST).

  - **Key Steps (GET):**
    1. **Country List Retrieval:** Fetches countries from CountriesService and prepares a SelectList for the dropdown in the form.
    2. **View Rendering:** Returns the Create view.
  - **Key Steps (POST):**
    1. **Model Binding:** Binds the submitted form data to a PersonAddRequest object.
    2. **Validation:** Checks if the model state is valid (using data annotations).
    3. **Creation and Redirect (if valid):** Calls _personsService.AddPerson to create the new person and redirects to the Index action.
    4. **Error Handling (if invalid):** Repopulates ViewBag with countries and error messages, then returns the Create view with the errors displayed.

- **Edit (Update):**
  - Similar to the Create action, it handles both GET (displaying the edit form) and POST (processing the form submission) requests.
  - Retrieves the person to edit from the database based on the personID route parameter.
  - Populates the form with the existing person's data.
  - Validates the form submission, updates the person if valid, and redirects to Index.

- **Delete (Delete):**
  - Similar to the Edit action, it also handles both GET and POST requests.
  - Retrieves the person to delete based on the personID route parameter.
  - Displays a confirmation view (GET) to the user.
  - If the user confirms, performs the deletion using _personsService.DeletePerson (POST) and redirects to Index.

Views

- **Index.cshtml:**
  - Displays a table of persons with filtering, sorting, and links to edit/delete actions.
  - Employs a partial view (_GridColumnHeader) to render sortable column headers.
  - Utilizes tag helpers for form creation and link generation.
- **Create.cshtml and Edit.cshtml:**
  - Render forms for creating/editing a person, using tag helpers (asp-for, asp-validation-for, etc.) for model binding and validation.
  - Includes a dropdown for country selection, populated from the ViewBag.Countries.
- **Delete.cshtml:**
  - Displays a confirmation message and a form with a hidden field for PersonID.

Enabling Client-Side Validation

- **Script Section:** The @section scripts block in Create.cshtml and Edit.cshtml includes scripts for jQuery, jQuery Validate, and jQuery Unobtrusive Validation.
- **Data Annotations:** The model classes (PersonAddRequest, PersonUpdateRequest) are decorated with data annotation attributes (e.g., [Required], [EmailAddress]) which are used by the client-side validation libraries to enforce the rules.

How HttpPost Action Method Submission Works

1. **Form Submission:** The user submits the form, triggering a POST request to the server.
2. **Model Binding:** ASP.NET Core's model binder extracts the data from the request and attempts to create an instance of the model specified in the action method parameter (e.g., PersonAddRequest).
3. **Model Validation:** The model binder runs the validation logic specified by data annotations and any custom validators.
4. **Action Execution (If Valid):** If ModelState.IsValid is true, the action method's logic executes (e.g., calling the _personsService.AddPerson method).
5. **Result (If Invalid):** If ModelState.IsValid is false, the action returns the same view, including error messages in ViewBag.Errors.

Key Points and Best Practices

- **Strong Typing:** Always prefer using strongly typed views and view models.
- **Thin Controllers:** Keep controller actions concise and delegate business logic to services.

- **Dependency Injection:** Inject services into your controllers for loose coupling.
- **DTOs:** Use DTOs to prevent overposting vulnerabilities.
- **Validation:** Implement both server-side and client-side validation.
- **Error Handling:** Handle exceptions gracefully and return appropriate status codes and error messages.

Stored Procedures in EF Core

Stored procedures are precompiled SQL code blocks stored within the database. While EF Core primarily emphasizes a Code First approach with LINQ (Language Integrated Query), integrating stored procedures can offer performance benefits, leverage existing database logic, or provide a way to execute complex database operations not easily expressed in LINQ.

Notes

- **Execution:** EF Core allows you to execute stored procedures using FromSqlRaw (for queries) or ExecuteSqlRaw (for non-query commands).
- **Parameterization:** You must use parameterized queries to prevent SQL injection vulnerabilities.
- **Limitations:** EF Core doesn't fully support mapping stored procedure results to complex entities with relationships (use result classes or projection for those cases).
- **Creating Stored Procedures:** You typically create stored procedures directly in your database (e.g., using SQL Server Management Studio) rather than from within EF Core.

Code Explanation

```
// PersonsDbContext.cs
public class PersonsDbContext : DbContext
{
  // ... DbSet properties and OnModelCreating ...

  public List<Person> sp_GetAllPersons()
  {
    return Persons.FromSqlRaw("EXECUTE [dbo].[GetAllPersons]").ToList();
  }
```

```
    public int sp_InsertPerson(Person person)
    {
        SqlParameter[] parameters = new SqlParameter[]
        {
            new SqlParameter("@PersonID", person.PersonID),
            // ... (Other parameters for PersonName, Email, etc.)
        };

        return Database.ExecuteSqlRaw("EXECUTE [dbo].[InsertPerson] @PersonID, @PersonName,
@Email, @DateOfBirth, @Gender, @CountryID, @Address, @ReceiveNewsLetters", parameters);
    }
}
```

In this code:

- **sp_GetAllPersons() Method:**
    - Executes a stored procedure named GetAllPersons.
    - Uses FromSqlRaw to treat the stored procedure's result set as a queryable collection of Person entities.
    - Returns a list of Person objects retrieved from the stored procedure's result set.

- **sp_InsertPerson() Method:**
    - Takes a Person object as input.
    - Creates an array of SqlParameter objects to pass values to the stored procedure.
    - Uses Database.ExecuteSqlRaw to execute the InsertPerson stored procedure, passing the parameterized values.
    - Returns the number of rows affected by the insert operation.

Best Practices

- **Parameterization:** Always parameterize your stored procedure calls to prevent SQL injection.
- **Result Classes (Optional):** If your stored procedure returns data that doesn't directly map to your entities, create separate result classes and use projection to map the data.

- **Naming:** Use a consistent naming convention for your stored procedure methods in the DbContext (e.g., the "sp_" prefix).
- **Transactions (Optional):** Wrap multiple stored procedure calls in a transaction to ensure data consistency.
- **Logging:** Consider adding logging to track stored procedure execution and any errors that might occur.

Fluent API

In EF Core, the Fluent API provides an alternative to data annotations for configuring your domain model and how it maps to the database schema. It allows you to define complex relationships, constraints, and other database-specific details that might not be easily expressed using attributes alone.

Why Use the Fluent API?

- **Flexibility:** It offers a broader range of configuration options than data annotations, enabling you to tackle more intricate scenarios.
- **Separation of Concerns:** Keeps your entity classes clean and focused on their domain logic, without cluttering them with database-specific attributes.
- **Readability:** The fluent API's method chaining syntax can be more expressive and readable than attribute-based configuration.

Using the Fluent API

You define Fluent API configurations within the OnModelCreating method of your DbContext class. This method is called when EF Core builds your model, giving you the opportunity to customize how entities and their properties are mapped to the database.

Important Fluent API Methods

1. **Entity-Level Configuration:**
   - modelBuilder.Entity<TEntity>(): This method gets an EntityTypeBuilder for a specific entity type (TEntity). It's the starting point for configuring an entity.
   - ToTable(string tableName): Configures the name of the database table for the entity.
   - HasKey(e => e.Property): Specifies the primary key property for the entity.
   - HasAlternateKey(e => e.Property): Specifies an alternate key (unique constraint) for the entity.

- Ignore(string propertyName): Excludes a property from being mapped to the database.
- HasQueryFilter(Expression<Func<TEntity, bool>> filter): Applies a global filter to the entity.

2. **Property-Level Configuration:**
   - Property(e => e.Property): Gets a PropertyBuilder for a specific property of the entity.
   - HasColumnName(string columnName): Specifies the column name in the database for the property.
   - HasColumnType(string typeName): Sets the data type of the column (e.g., nvarchar, int, datetime2).
   - HasMaxLength(int maxLength): Sets the maximum length for a string property.
   - IsRequired(): Makes the property required in the database (not nullable).
   - HasDefaultValue(object value): Sets the default value for the property.
   - HasDefaultValueSql(string sql): Sets the default value using a SQL expression.
   - ValueGeneratedOnAdd() or ValueGeneratedNever(): Controls how the value is generated (automatic, never, on update).

3. **Relationship Configuration:**
   - HasOne(e => e.NavigationProperty): Configures a one-to-one relationship.
   - WithMany(e => e.NavigationProperty): Configures a one-to-many relationship.
   - HasForeignKey(e => e.Property): Specifies the foreign key property for the relationship.
   - HasPrincipalKey(e => e.Property): Specifies the principal key property for the relationship.
   - WithMany().HasForeignKey(e => e.Property): Configures a many-to-many relationship.

4. **Index and Constraint Configuration:**
   - HasIndex(e => e.Property): Creates an index on a property or properties.
   - HasCheckConstraint(string name, string sql): Adds a check constraint to the table.

Code Example

```
// PersonsDbContext.cs
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  // ... table mappings and seed data ...

  // Fluent API Configuration
  modelBuilder.Entity<Person>()
    .Property(temp => temp.TIN) // Configure the TIN property
    .HasColumnName("TaxIdentificationNumber")
    .HasColumnType("varchar(8)")
    .HasDefaultValue("ABC12345");

  // ... other Fluent API configurations ...
}
```

In this code:

1. modelBuilder.Entity<Person>(): Gets the entity type builder for the Person entity.
2. .Property(temp => temp.TIN): Gets the property builder for the TIN property.
3. .HasColumnName("TaxIdentificationNumber"): Sets the column name in the database to "TaxIdentificationNumber".
4. .HasColumnType("varchar(8)"): Sets the column's data type to varchar(8).
5. .HasDefaultValue("ABC12345"): Sets the default value for the TIN column to "ABC12345".
6. .HasCheckConstraint("CHK_TIN", "len([TaxIdentificationNumber]) = 8"): This will apply a check constraint on the table, where the length of the Tax Identification number must be exactly 8.

});

Referential Integrity, Primary Keys, and Foreign Keys

- **Referential Integrity:** This is a database concept that ensures the consistency and validity of relationships between tables. It prevents actions that would break these relationships, such as deleting a record that is referenced by other records in another table.
- **Primary Key:** A unique identifier for each record in a table. It ensures that each row is uniquely identifiable and enforces entity integrity. Primary keys are typically of integer or GUID types.
  - In EF Core, you can mark a property as a primary key using the [Key] attribute or the Fluent API's HasKey() method.
- **Foreign Key:** A column (or set of columns) in a table that refers to the primary key of another table. Foreign keys establish relationships between tables and enforce referential integrity.
  - In EF Core, you define foreign keys using the [ForeignKey] attribute, the Fluent API's HasForeignKey() method, or by convention (if the property name follows a specific pattern).

Managing Relationships in EF Core Models

1. **Navigation Properties:**
   - These are properties in your entity classes that hold references to related entities.
   - They allow you to navigate from one entity to its related entities without writing explicit joins in your queries.
   - Declare navigation properties with the appropriate types: public virtual ICollection<Person>? Persons { get; set; } in the Country class.

2. **Fluent API:**
   - Use the Fluent API in your DbContext's OnModelCreating method to define relationships explicitly.

```
// PersonsDbContext.cs

modelBuilder.Entity<Person>(entity =>

{

  entity.HasOne(p => p.Country)

       .WithMany(c => c.Persons)

       .HasForeignKey(p => p.CountryID);

});
```

- This configuration establishes a one-to-many relationship between Country and Person. A country can have many persons, and a person belongs to one country. The CountryID property in the Person class is the foreign key pointing to the CountryID primary key in the Country class.

LINQ Queries with Table Relations

You can use LINQ to query across relationships using navigation properties.

```
// Find all persons from the USA

var peopleFromUSA = _dbContext.Persons

   .Where(p => p.Country.CountryName == "USA")

   .ToList();
```

Include() Method in LINQ Queries

The Include() method allows you to eagerly load related entities in a single query. This avoids the N+1 query problem, where you would otherwise have to make separate queries to fetch related data for each entity.

```
// Find all persons from the USA, including their country details

var peopleFromUSA = _dbContext.Persons

   .Include(p => p.Country)  // Eagerly load the Country entity

   .Where(p => p.Country.CountryName == "USA")

   .ToList();
```

Best Practices

- **Choose the Right Relationship:** Carefully consider the nature of your data to select the correct relationship type (one-to-one, one-to-many, many-to-many).
- **Cascading Behavior:** Decide how you want EF Core to handle cascading actions (e.g., when deleting a country, should it delete the associated persons?). You can configure this in the Fluent API using options like OnDelete.
- **Performance Considerations:** Avoid overusing Include() if you don't need the related data in your current operation. It can lead to fetching more data than necessary.
- **Explicit Loading:** If you need to load related data only in certain cases, use explicit loading (Load() method) instead of Include().

Things to Avoid

- **Lazy Loading (Default in EF Core):** By default, EF Core enables lazy loading, which means related entities are loaded on demand when you first access their navigation properties. This can lead to performance issues if you're not careful. You can disable lazy loading if it doesn't fit your needs.

- **Ignoring Referential Integrity:** Improperly configured relationships can lead to data inconsistency and unexpected behavior.

CRUD Operations with Entity Framework Core

EF Core, as an Object-Relational Mapper (ORM), simplifies database interactions by allowing you to work with data as C# objects (entities). Let's see how this translates into implementing CRUD operations within your controllers.

Controller Actions

- **Index (Read):**
  - **Purpose:** Display a list of entities.
  - **Data Source:** Retrieves Person entities using _personsService.GetAllPersons(), applying filters and sorting based on query parameters if provided.
  - **ViewBag:** Populates ViewBag with:
    - Search fields and their display names.
    - The current search and sort criteria for display in the view.
  - **View:** Returns the "Index" view with the retrieved and processed data.
  - **Error Handling:** None in this example, but ideally, you'd handle exceptions from the service and potentially display error messages in the view.

- **Create (Create):**
  - **Purpose:**
    - **(GET)** Displays a form for creating a new person.
    - **(POST)** Processes the submitted form data to add a new person.
  - **Data Source (GET):** Retrieves countries from _countriesService.GetAllCountries() to populate a country dropdown.
  - **Model Binding (POST):** Binds the incoming form data to a PersonAddRequest object.

- **Validation:**
    - **Server-Side:** Checks if ModelState.IsValid.
    - **Client-Side:** Uses jQuery Validation and Unobtrusive Validation to provide immediate feedback in the browser.
- **Logic:**
    - **(POST-Valid):** Calls _personsService.AddPerson to create the new person and redirects to the Index action.
    - **(POST-Invalid):** Repopulates the ViewBag (countries and errors) and re-renders the Create view with validation errors.

- **Edit (Update):**
    - **Purpose:**
        - **(GET)** Displays a form for editing an existing person.
        - **(POST)** Processes the submitted form data to update the person's details.
    - **Data Source (GET):**
        - Fetches the person to edit based on personID from the PersonsService.
        - Retrieves countries from _countriesService.GetAllCountries() for the dropdown.
    - **Model Binding (POST):** Binds the form data to a PersonUpdateRequest object.
    - **Validation & Logic:** Same as in the Create action.

- **Delete (Delete):**
    - **Purpose:**
        - **(GET):** Displays a confirmation page for deleting a person.
        - **(POST):** Deletes the person.
    - **Data Source (GET):** Fetches the person to delete based on personID.
    - **Model Binding (POST):** Binds only the PersonID from the form.
    - **Logic (POST):** Calls _personsService.DeletePerson and redirects to the Index action.

EF Core CRUD Operations (in the PersonsService):

- **AddPerson:**
    - Adds a new Person entity to the Persons DbSet of the DbContext.

    - Calls _db.SaveChanges() to persist the changes to the database.

- **GetAllPersons:** Retrieves all Person entities from the database using _db.Persons.Include("Country").ToListAsync(). The .Include("Country") ensures eager loading of the related Country entity for each person.
- **GetPersonByPersonID:** Retrieves a specific person based on their ID using _db.Persons.FirstOrDefaultAsync(temp => temp.PersonID == personID).
- **UpdatePerson:** Updates the properties of an existing person and calls _db.SaveChanges() to save the changes.
- **DeletePerson:** Removes a person from the database and calls _db.SaveChanges() to persist the deletion.

Best Practices

- **Asynchronous Operations:** The service methods in your example correctly use async and await for database operations, which helps to avoid blocking the main thread and improves the responsiveness and scalability of your application.
- **Dependency Injection:** The services (IPersonsService, ICountriesService) are injected into the controller, following the Dependency Inversion Principle (DIP) and making the code more testable.
- **Data Transfer Objects (DTOs):** The use of PersonAddRequest, PersonResponse, etc. helps to keep your domain model (the Person class) separate from your presentation layer, preventing overposting vulnerabilities and improving the maintainability of your code.
- **Validation:** Both client-side (jQuery Validate) and server-side (model state validation in the controller) are used to ensure data integrity.
- **Error Handling:** Exceptions are caught, and appropriate error messages or redirections are provided.

Generating PDFs in ASP.NET Core MVC

Creating PDF files directly within your ASP.NET Core MVC applications is a valuable feature for generating reports, invoices, tickets, or any other documents you need in a portable and widely supported format. Several libraries exist to simplify this process, and Rotativa is one popular choice.

Rotativa: Leveraging Wkhtmltopdf for PDF Generation

Rotativa is a .NET library that wraps the wkhtmltopdf tool, a command-line utility that converts HTML content into PDF documents. This makes it remarkably simple to generate PDFs in ASP.NET Core by leveraging your existing Razor views.

Notes About Rotativa

- **Installation:** Install the Rotativa.AspNetCore NuGet package.
- **ViewAsPdf:** Rotativa provides an ViewAsPdf action result that you can return from your controller actions. This action result takes your view name, model data, and optionally, custom settings.
- **Customization:** You can customize various aspects of the PDF, including margins, page orientation, header/footer content, and more.
- **Dependency on Wkhtmltopdf:** Rotativa depends on the wkhtmltopdf executable, which needs to be installed on your system (or accessible in your deployment environment).

Code Example:

Controller Action (PersonsController.cs)

```
[Route("PersonsPDF")]
public async Task<IActionResult> PersonsPDF()
{
  // Get list of persons
  List<PersonResponse> persons = await _personsService.GetAllPersons();


  // Return view as pdf
  return new ViewAsPdf("PersonsPDF", persons, ViewData) // Render the "PersonsPDF" view as a
PDF
  {
    PageMargins = { Top = 20, Right = 20, Bottom = 20, Left = 20 },
    PageOrientation = Orientation.Landscape // Set landscape orientation
  };
}
```

1. **Retrieves Data:** Fetches a list of PersonResponse objects from the _personsService.
2. **ViewAsPdf Action Result:** Creates a ViewAsPdf action result:

    - "PersonsPDF": Specifies the name of the view to render as a PDF.
    - persons: Passes the list of persons as the model to the view.
    - ViewData: Passes the ViewData object containing the page title.
3. **Customization:** Configures the PDF's margins and orientation.

View (PersonsPDF.cshtml)

```
@model IEnumerable<PersonResponse>
@{
  Layout = null; // Disable the layout for this view, since it is rendered as a PDF
}

<link href="@("http://" + Context.Request.Host.ToString() + "/Stylesheet.css")" rel="stylesheet" />
@* http://localhost:port/StyleSheet.css *@

<h1>Persons</h1>

<table class="table w-100 mt">
  <thead>
    <tr>
      <th>Person Name</th>
      <th>Email</th>
      <th>Date of Birth</th>
      <th>Age</th>
      <th>Gender</th>
      <th>Country</th>
      <th>Address</th>
      <th>Receive News Letters</th>
    </tr>
  </thead>
  <tbody>
    @foreach (PersonResponse person in Model)
```

```
      {
        <tr>
          <td style="width:15%">@person.PersonName</td>
          <td style="width:20%">@person.Email</td>
          <td style="width:13%">@person.DateOfBirth?.ToString("dd MMM yyyy")</td>
          <td style="width:9%">@person.Age</td>
          <td style="width:9%">@person.Gender</td>
          <td style="width:10%">@person.Country</td>
          <td style="width:15%">@person.Address</td>
          <td style="width:20%">@person.ReceiveNewsLetters</td>
        </tr>
      }
  </tbody>
</table>
```

- The view's content is plain HTML, using a foreach loop and razor syntax to loop through the data and create a table to display the results.

Best Practices

- **Separate PDF Views:** Create dedicated views for PDF generation to avoid cluttering your regular views with PDF-specific styling or layout.
- **CSS for Styling:** Use CSS (either inline or linked) to style your PDF content.
- **wkhtmltopdf Options:** Familiarize yourself with the available wkhtmltopdf options to customize the generated PDF (headers, footers, page size, etc.).
- **Deployment:** Ensure that wkhtmltopdf is installed on your production server if you're using Rotativa.

Alternative Libraries

- **iTextSharp/iText7:** A powerful library for creating and manipulating PDF documents programmatically.
- **QuestPDF:** A modern, fluent library for generating PDFs from C# code.
- **IronPDF:** Allows you to convert HTML to PDF with ease.

Generating CSV Files in ASP.NET Core MVC

Comma-Separated Values (CSV) files are a simple and widely supported format for exporting tabular data. They're often used for data exchange between systems, bulk imports/exports, or providing data downloads for users. ASP.NET Core MVC makes it straightforward to generate CSV files from your data, especially with the help of the CsvHelper library.

CsvHelper

CsvHelper is a popular and well-maintained .NET library designed for reading and writing CSV files with ease. It handles tasks like:

- **Reading:** Parsing CSV files into strongly typed objects or dynamic collections.
- **Writing:** Generating CSV files from your data, customizing delimiters, headers, and formatting.
- **Mapping:** Mapping your class properties to CSV columns using conventions or explicit configuration.

Notes About CsvHelper

- **Installation:** Install the CsvHelper NuGet package.
- **CsvWriter and CsvReader:** These are the core classes for writing and reading CSV data.
- **Customization:** You can customize how your CSV data is read or written using configuration options and mapping strategies.
- **Flexibility:** CsvHelper supports various CSV formats, delimiters, and encoding options.

Code Example:

Controller Action (PersonsController.cs)

```
[Route("PersonsCSV")]

public async Task<IActionResult> PersonsCSV()

{

   MemoryStream memoryStream = await _personsService.GetPersonsCSV(); // Get CSV data from the service

   return File(memoryStream, "application/octet-stream", "persons.csv");
}
```

1. **CSV Data Retrieval:** The action method calls _personsService.GetPersonsCSV() to get a MemoryStream containing the CSV data. This method handles the logic of fetching data from the database, formatting it as CSV, and writing it to the memory stream.
2. **File Result:** The File() method is used to return a FileContentResult action result. The arguments include:
   - memoryStream: The stream containing the CSV data.
   - "application/octet-stream": The content type for CSV files (forces download).
   - "persons.csv": The suggested filename for the downloaded file.

Service Method (PersonsService.GetPersonsCSV())

```
public async Task<MemoryStream> GetPersonsCSV()
{
  MemoryStream memoryStream = new MemoryStream();
  StreamWriter streamWriter = new StreamWriter(memoryStream);
  CsvWriter csvWriter = new CsvWriter(streamWriter, CultureInfo.InvariantCulture, leaveOpen: true);

  csvWriter.WriteHeader<PersonResponse>(); // Write CSV headers based on the PersonResponse class
  csvWriter.NextRecord();

  List<PersonResponse> persons = _db.Persons // Query persons from the database
    .Include("Country")
    .Select(temp => temp.ToPersonResponse()).ToList();

  await csvWriter.WriteRecordsAsync(persons); // Write person data as CSV rows

  memoryStream.Position = 0; // Reset the stream position
  return memoryStream;
}
```

1. **Create Stream:** A MemoryStream is created to hold the CSV data.

2. **CSV Writer:** A CsvWriter is initialized, using the memory stream and specifying the culture info (to ensure consistent number and date formatting).
3. **Headers:** csvWriter.WriteHeader<PersonResponse>() writes the CSV header row using the property names of the PersonResponse class.
4. **Data Retrieval:** A LINQ query retrieves all Person entities along with their related Country information (eager loading).
5. **Data Writing:** csvWriter.WriteRecordsAsync(persons) writes each PersonResponse object as a row in the CSV file.
6. **Reset Stream Position:** The memoryStream.Position is reset to the beginning so that the controller action can read the entire CSV content.

Best Practices

- **Choose the Right Library:** CsvHelper is a great choice, but explore other libraries if you have specific requirements (e.g., FastCsvParser for large files).
- **Streaming:** For large datasets, consider streaming the CSV generation process to avoid loading all data into memory at once.
- **Customization:** Customize the CSV format (delimiter, headers, etc.) as needed.
- **Error Handling:** Handle potential errors during CSV generation (e.g., invalid data).
- **Security:** If you are including sensitive information in the CSV, consider encryption or other security measures.

Key Points to Remember

Entity Framework Core (EF Core)

- **Purpose:** Object-Relational Mapper (ORM) that simplifies database interaction in .NET.
- **Core Concepts:**
    - **DbContext:** A session with the database, tracks changes, handles queries.
    - **DbSet<T>:** Represents a collection of a specific entity type.
    - **Entities:** C# classes that model your database tables.
    - **Mapping:** Defines how entities and database tables/columns correspond.
    - **LINQ Queries:** Query the database using C#.
    - **Change Tracking:** EF Core tracks changes to entities for efficient updates.

    - **Migrations:** Manages changes to your database schema over time.

EF Core Approaches

- **Code First:** Define your model with C# classes, EF Core creates the database.
- **Database First:** Start with an existing database, EF Core generates entity classes.

Fluent API

- **Purpose:** Alternative to data annotations for configuring the model in code.
- **Key Methods:**
    - modelBuilder.Entity<T>(): Configures an entity.
    - ToTable(), HasKey(), HasAlternateKey(): Table and key configuration.
    - Property(): Configures a property's column name, type, constraints, etc.
    - HasOne(), WithMany(): Configures relationships.
    - HasIndex(), HasCheckConstraint(): Creates indexes and constraints.

Relationships

- **Types:** One-to-one, one-to-many, many-to-many.
- **Navigation Properties:** Properties in your entities that reference related entities.
- **Foreign Keys:** Define using [ForeignKey], fluent API, or convention.
- **LINQ Queries:** Use navigation properties for querying related data.
- **Include():** Eagerly load related entities in a single query.

Code First Migrations

- **Purpose:** Manage database schema changes as your model evolves.
- **Commands (Package Manager Console):**
    - Add-Migration "Name": Creates a new migration.
    - Update-Database: Applies pending migrations to the database.
    - Remove-Migration: Reverts the last migration.

Seed Data

- **Purpose:** Populate the database with initial data.

- **HasData() Method:** Used within OnModelCreating to seed data.

Stored Procedures

- **FromSqlRaw:** Executes a stored procedure and maps results to entities.
- **ExecuteSqlRaw:** Executes a stored procedure that doesn't return results.
- **Parameterization:** Always use parameters to prevent SQL injection.

Async Operations

- **ToListAsync(), FirstOrDefaultAsync(), SaveChangesAsync():** Asynchronous versions of common EF Core methods.
- **Benefits:** Improved scalability and responsiveness by avoiding blocking the main thread.

Generating Files

- **PDFs (e.g., Rotativa):**
  - Install the library.
  - Use ViewAsPdf to render a view as a PDF.
- **CSVs (e.g., CsvHelper):**
  - Install the library.
  - Use CsvWriter to write data to a CSV file or stream.

Best Practices

- **Repository Pattern:** Consider using it to abstract data access logic.
- **Unit of Work Pattern:** Group multiple database operations into a single transaction.
- **Connection Resiliency:** Implement strategies to handle transient errors when connecting to the database.
- **Caching:** Cache query results where appropriate to improve performance.

Interview Tips

- **Concepts:** Explain the core concepts of EF Core and the benefits of using an ORM.
- **Relationships:** Demonstrate how to define and work with relationships between entities.

- **Migrations:** Discuss the importance of migrations for managing schema changes.
- **Best Practices:** Showcase your knowledge of best practices like using the repository pattern, asynchronous operations, and handling errors.