


HANZA Mentors Documentation

Welcome to the documentation for HANZA Mentors, a comprehensive mentoring management system designed specifically for Hanza Mechanics. This system has been carefully designed to improve the mentoring management for HRs at Hanza Mechanics. It simplifies the process of managing mentors and mentees as well as the process of issuing and signing mentoring-related documents. A time-saving solution that automates manual tasks, boosting HR efficiency.

 Currently, the work is still in process, and the documentation is updating regularly to reflect changes

Project Objective

The genesis of HANZA Mentors traces back to a pivotal moment at Hanza Mecha..... khmm. The project's primary objective was set clear by client, a senior HR at the Hanza Narva factory: to develop a mentoring management system that would not only simplify the mentoring management for HRs but also enhance efficiency and collaboration on the HR/mentor/mentee line.

The client identified specific goals for the application, each aimed at addressing corresponding challenges:

1. Managing Mentorship Processes

- **Problem:** Difficulty in effectively managing mentorship processes.
- **Solution:** The system aims to empower HRs by facilitating the creation and modification of mentorship programs tailored to different mentee types. This includes features such as automatic hour recalculations for adjustments like sick leaves, ensuring smoother management.

2. Documents Creation Automation

- **Problem:** Time-consuming manual creation of mentoring-related documents.
- **Solution:** Automation of document creation is crucial. By implementing templates that dynamically fill with outcomes from each mentorship period, the system reduces HR personnel's time and effort in document preparation.

3. Documents Storing in Database

- **Problem:** Difficulty in accessing and retrieving mentoring-related documents.
- **Solution:** Seamless storage within a centralized database. Storing documents electronically ensures effortless access and retrieval, eliminating the need for burdensome manual filing systems.

4. Uploading New Samples of Documents

- **Problem:** Adapting to changing document requirements and best practices.
- **Solution:** Providing HRs with the capability to upload new document samples as needed fosters adaptability. This flexibility enables the system to evolve alongside changing requirements and emerging best practices.

5. Digital or Physical Document Signing

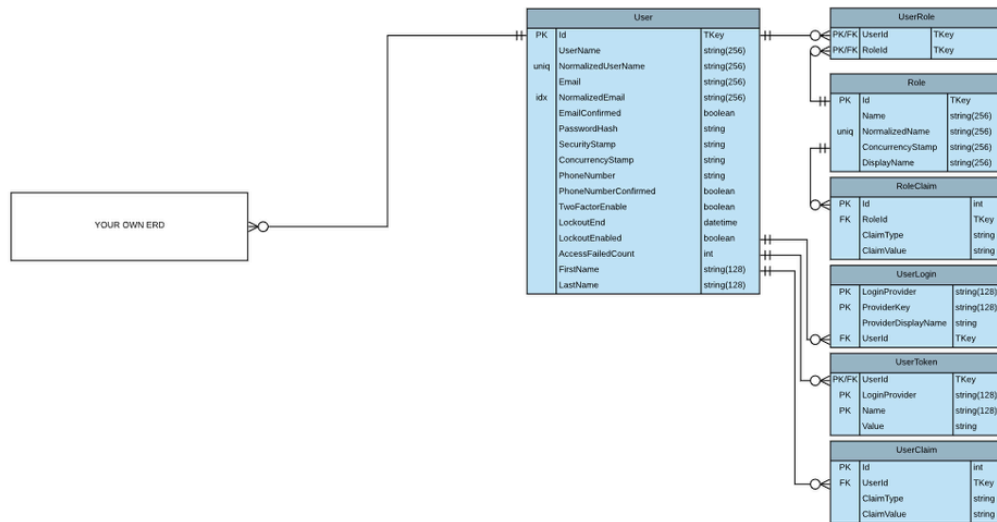
- **Problem:** Accommodating varying preferences for document signing.
- **Solution:** Offering both digital and physical document signing options caters to diverse preferences. The system facilitates flexibility, ensuring users can sign documents in their preferred manner and time.

6. User-Friendly Interface:

- **Problem:** Usability issues coming out from a lack of a user-friendly interface.

- In summary, HANZA Mentors aims to simplify mentoring processes at Hanza Mechanics by organizing programs effectively, automating document management, and providing flexible signing options. The system prioritized user-friendly design to ensure ease of use, aiming to enhance user satisfaction.

HANZA Mentors ERD



ASP.NET Identity ERD

ASP.NET Identity documentation - [Identity model customization in ASP.NET Core](#)

Table Purposes:

- **Employee:** This table serves to manage various details regarding employees within an organization, such as their roles, positions, and other information.
- **Intern:** This table is dedicated to storing essential data concerning interns, including their status, program details, and relevant associations within the organization.
- **AppUser:** This table is fundamental for managing user accounts within the application, storing user-related information such as names, emails, and encrypted passwords. Additionally, it extends the User entity provided by ASP.NET Core Identity, enhancing its functionality within the application.
- **FactorySupervisor:** Within this table, information regarding factory supervisors, including their names and potentially other identifying details, is stored.
- **MenteeSickLeave:** This table is used to track instances of sick leave taken by mentees, providing a record of such absences within the organization. The table includes mentees of all types, since sickleave is not a common phenomenon in an enterprise and there is no point in creating separate tables for intern-mentees and employee-mentees.
- **InternMentorshipUntilDate:** Here, the end dates of mentorship programs for interns are recorded, aiding in the management and tracking of such programs. This table is essential due to the potential for multiple changes in the end date of mentoring, allowing for accurate tracking of these changes over time.
- **EmployeeMentorshipDocument:** This table facilitates the storage and retrieval of documents pertinent to employee mentorship programs, aiding in the organization and management of program-related materials. It's separate from InternMentorshipDocument table because one employee mentoring period involves receiving between 1 to 3 documents for employee-mentee and employee-mentor each. This setup allows for better documents organization, making it easier to keep track of them during the employee mentoring process.
- **EmployeeMentorship:** Within this table are stored details regarding mentorship relationships involving employees, providing insights into the mentorship dynamics within the organization.
- **EmployeeMentorshipUntilDate:** This table tracks the end dates of mentorship programs for employees, providing valuable information for program management and evaluation. This table is essential due to the potential for multiple changes in the end date of mentoring, allowing for accurate tracking of these changes over time.
- **InternMentorship:** This table manages and tracks mentorship relationships specifically tailored to interns, aiding in the organization and oversight of intern mentorship programs.

- **InternSupervisor:** Information regarding supervisors responsible for overseeing interns is stored within this table, facilitating effective supervision and support for interns.
- **SickLeaveType:** This table stores information about various types of sick leave available to employees or interns, enabling accurate tracking and management of sick leave policies.
- **InternMentorshipDocument:** This table is dedicated to managing documents related to intern mentorship programs, ensuring the availability and accessibility of program-related materials. It's separate from EmployeeMentorshipDocument table because one intern mentoring period involves receiving between 1 to 3 documents for intern-mentee and employee-mentor each. This setup allows for better documents organization, making it easier to keep track of them during the intern mentoring process.
- **DocumentSample:** Within this table, templates of documents used within the organization are stored, aiding in standardization and consistency across document types.
- **Mentor:** This table serves to manage information about mentors participating in mentorship programs, providing insights into mentorship roles and responsibilities within the organization. The table includes mentors of all types (intern-mentor/employee-mentor) because within the Narva factory, there are no more than 25 active mentors at one time, so there is no need to store different mentors across two tables.

Attributes and Their Purposes in Each Table

1. Employee:

- Id: Unique identifier for each employee.
- AppUserId: Foreign key referencing the user ID in the AppUser table.
- EmployeeType: Type of employee (full-time, rental, probation period).
- Profession: Profession of the employee.

2. Intern:

- Id: Unique identifier for each intern.
- AppUserId: Foreign key referencing the user ID in the AppUser table.
- InternType: Type of intern (EUIF-estonian unemployment insurance fund/töötukassa) intern, vocational school intern).
- StudyProgram: Program of study for the intern.

3. AppUser:

- Id: Unique identifier for each user.
- FirstName: First name of the user.
- LastName: Last name of the user.
- PersonalCode: Personal identification(isikukood) code of the user. Length - 11 numbers.
- Email: Email address of the user.

4. FactorySupervisor:

- Id: Unique identifier for each factory supervisor.
- FullName: Full name of the factory supervisor.

5. MenteeSickLeave:

- Id: Unique identifier for each sick leave entry.
- InternMentorshipId: Foreign key referencing the intern mentorship. If the row in this table is related to intern mentoship then this attribute will be set and EmployeeMentorshipId will be null.
- EmployeeMentorshipId: Foreign key referencing the employee mentorship. If the row in this table is related to employee mentoship then this attribute will be set and InternMentorshipId will be null.
- SickLeaveTypeId: Foreign key referencing the type of sick leave.
- FromDate: Start date of the sick leave.

- UntilDate: End date of the sick leave.

6. InternMentorshipUntilDate:

- Id: Unique identifier for each intern mentorship end date.
- InternMentorshipId: Foreign key referencing the intern mentorship.
- UntilDate: End date of the mentorship.

7. EmployeeMentorshipDocument:

- Id: Unique identifier for each mentorship document.
- EmployeeMentorshipId: Foreign key referencing the employee mentorship.
- ReceiverId: ID of the receiver of the document. The receiver can either be an employee-mentor or an employee-mentee. To determine the receiver's ID, we look up information from the EmployeeMentorship table. This can be directly through the EmployeeId from the EmployeeMentorship table, or indirectly through the EmployeeMentorshipId and Mentor table to retrieve the EmployeeId.
- DocumentSampleId: ID referencing the sample document.
- Base64Code: Base64-encoded document content.
- DocumentStatus: Status of the document.

8. EmployeeMentorship:

- Id: Unique identifier for each employee mentorship.
- EmployeeMenteeId: ID of the employee mentee.
- FactorySupervisorId: ID of the factory supervisor.
- FromDate: Start date of the mentorship.
- TotalHours: Total hours spent on mentorship.

9. EmployeeMentorshipUntilDate:

- Id: Unique identifier for each employee mentorship end date.
- EmployeeMentorshipId: Foreign key referencing the employee mentorship.
- UntilDate: End date of the mentorship.

10. InternMentorship:

- Id: Unique identifier for each intern mentorship.
- InternId: ID of the intern.
- InternSupervisorId: ID of the intern supervisor.
- FactorySupervisorId: ID of the factory supervisor.
- FromDate: Start date of the mentorship.
- TotalHours: Total hours spent on mentorship.

11. InternSupervisor:

- Id: Unique identifier for each intern supervisor.
- FullName: Full name of the intern supervisor.
- Type: Type of intern supervisor.

12. SickLeaveType:

- Id: Unique identifier for each sick leave type.
- Type: Type of sick leave.

13. InternMentorshipDocument:

- Id: Unique identifier for each mentorship document.
- InternMentorshipId: Foreign key referencing the intern mentorship.

- ReceiverId: ReceiverId: ID of the receiver of the document. The receiver can either be an employee-mentor or an intern-mentee. To determine the receiver's ID, we look up information from the InternMentorship table. This can be directly through the EmployeeId from the InternMentorship table, or indirectly through the InternMentorshipId and Intern table to retrieve the InternId.
- DocumentSampleId: ID referencing the sample document.
- Base64Code: Base64-encoded document content.
- DocumentStatus: Status of the document.

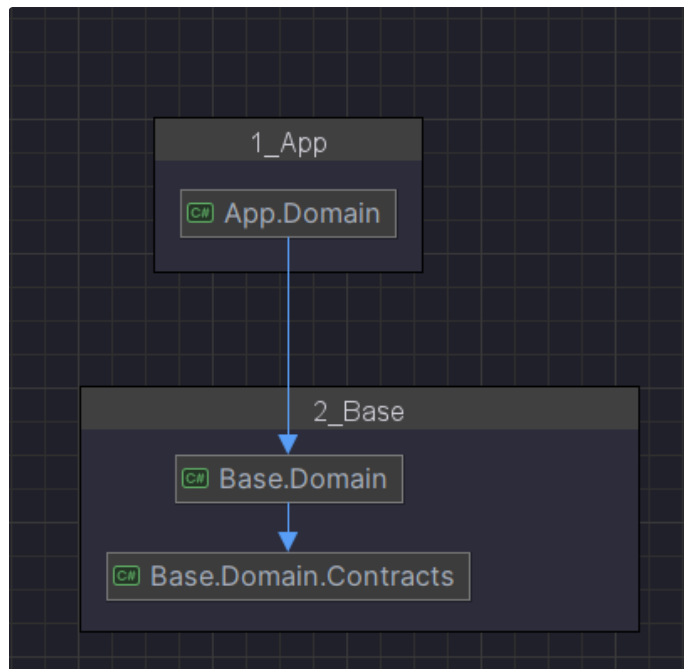
14. **DocumentSample:**

- Id: Unique identifier for each document sample.
- Title: Title or name of the document.
- Base64Code: Base64-encoded document content.

15. **Mentor:**

- Id: Unique identifier for each mentor.
- MentorId: ID of the mentor.
- InternMentorshipId: Foreign key referencing the intern mentorship. If the row in this table is related to intern mentorship then this attribute will be set and EmployeeMentorshipId will be null.
- EmployeeMentorshipId: Foreign key referencing the employee mentorship. If the row in this table is related to employee mentorship then this attribute will be set and InternMentorshipId will be null.
- FromDate: Start date of the mentorship.
- UntilDate: End date of the mentorship.
- PaymentAmount: Amount paid for the mentorship.
- PaymentOrderDate: Date of the payment order.
- ChangeReason: Reason for any changes that occur during a mentorship period, particularly if one mentor is switched to another. If mentoring process ended without mentor switching, this attribute stays null.

Application Domain Layer Structure



Overall Domain structure

2_Base Solution:

Base.Domain.Contract Project:

This project defines contract - interface that is common across different domains within the application.

```
1 namespace Base.Domain.Contracts;
2
3 public interface IBaseEntityId : IBaseEntityId<Guid>
4 {
5
6 }
7
8 public interface IBaseEntityId<TKey>
9     where TKey : IEquatable<TKey>
10 {
11     public TKey Id { get; set; }
12 }
```

• IBaseEntityId Interface:

The `IBaseEntityId` interface defines the contract for entities with a unique identifier. It allows for abstraction over different types of identifiers by providing a generic definition.

Interfaces:

1. `IBaseEntityId` (Non-generic):
 - Inherits: `IBaseEntityId<Guid>`
 - Purpose: Acts as a marker interface to represent entities with a unique identifier of type `Guid`.
2. `IBaseEntityId<TKey>` (Generic):
 - Type Parameter: `TKey` - Represents the type of the entity's identifier.
 - Constraints: Requires `TKey` to implement the `IEquatable<TKey>` interface.
 - Properties:

- `Id` : Getter and setter for the unique identifier of the entity.

Base.Domain Project:

This project contains base class and common functionalities shared across different domains.

```

1 using Base.Domain.Contracts;
2
3 namespace Base.Domain;
4
5 public abstract class BaseEntityId : BaseEntityId<Guid>, IBaseEntityId
6 {
7 }
8
9 public abstract class BaseEntityId<TKey>: IBaseEntityId<TKey>
10     where TKey : IEquatable<TKey>
11 {
12     public TKey Id { get; set; } = default!;
13 }

```

• BaseEntityId Class:

The `BaseEntityId` class serves as a foundational abstraction for entities with unique identifiers within the domain model.

Classes:

1. `BaseEntityId` (Non-generic):
 - Inherits: `BaseEntityId<Guid>`, `IBaseEntityId`
 - Purpose: This abstract class extends the generic `BaseEntityId<TKey>` class to provide a default implementation for entities with `Guid` identifiers. It serves as a convenient base for such entities, inheriting the behavior defined in the generic class while specifying `Guid` as the default type for identifiers.
2. `BaseEntityId<TKey>` (Generic):
 - Type Parameter: `TKey` - Represents the type of the entity's identifier.
 - Constraints: Requires `TKey` to implement the `IEquatable<TKey>` interface.
 - Properties:
 - `Id` : Getter and setter for the unique identifier of the entity.
 - Purpose: This abstract class defines the structure for entities with unique identifiers of a generic type. It enforces the presence of an `Id` property, allowing subclasses to specify the type of identifier they require. By providing a default value for the `Id` property, it ensures that entities always have a valid identifier.

2_App Solution:

App.Domain Project:

This project contains domain-specific 17 entities (14 base + 3 identity) representing various concepts within the application.

```

1 using System.ComponentModel.DataAnnotations;
2 using App.Domain.Identity;
3 using Base.Domain;
4
5 namespace App.Domain;
6
7 public class Employee : BaseEntityId
8 {
9     public Guid? AppUserId { get; set; }
10    public AppUser? AppUser { get; set; }
11
12    public int EmployeeType { get; set; }
13 }

```



```

14     [MaxLength(50)]
15     public string? Profession { get; set; }
16 }

```

Example Employee class

- Represents an `Employee` entity within the application.
- Inherits from `BaseEntityId`, indicating it has a unique identifier.
- Contains properties such as `AppUserId`, `EmployeeType`, and `Profession`.
- Utilizes data annotations for validation and database schema definition.

App.Domain.Identity folder:

Identity folder within the App.Domain project includes custom entity implementations tailored to integrate seamlessly with the application's specific database context (`AppDbContext`).

```

1 using System.ComponentModel.DataAnnotations;
2 using Base.Domain.Contracts;
3 using Microsoft.AspNetCore.Identity;
4
5 namespace App.Domain.Identity;
6
7 public class AppUser : IdentityUser<Guid>, IBaseEntityId
8 {
9     [MaxLength(128)]
10    public string FirstName { get; set; } = default!;
11
12    [MaxLength(128)]
13    public string LastName { get; set; } = default!;
14
15    public int PersonalCode { get; set; }
16 }

```

• AppUser Class:

- Represents a user entity in the application's identity system.
- Inherits from `IdentityUser<Guid>` and implements the `IBaseEntityId` interface, ensuring it has a unique identifier.
- Includes properties for user details such as `FirstName`, `LastName`, and `PersonalCode`.

```

1 using Base.Domain.Contracts;
2 using Microsoft.AspNetCore.Identity;
3
4 namespace App.Domain.Identity;
5
6 public class AppRole : IdentityRole<Guid>, IBaseEntityId
7 {
8
9 }

```

• AppRole Class:

- Represents a role entity in the application's identity system.
- Inherits from `IdentityRole<Guid>` and implements the `IBaseEntityId` interface.
- Provides support for assigning roles to users within the application.

```

1 using Microsoft.AspNetCore.Identity;
2
3 namespace App.Domain.Identity;
4

```

```

5 public class AppUserRole : IdentityUserRole<Guid>
6 {
7
8 }

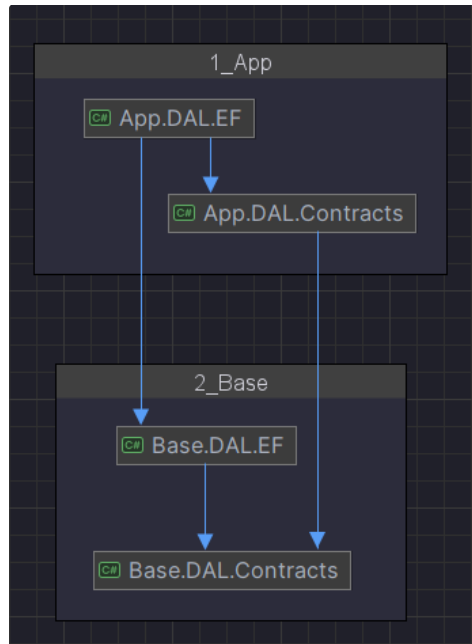
```

- **AppUserRole Class:**

- Represents the association between users and roles in the application's identity system.
- Inherits from IdentityUserRole<Guid>.
- Facilitates the management of user-role relationships.

This domain layer structure promotes code reusability and maintainability by providing a clear separation of concerns and standardizing common functionalities across different domains and identity management within the application.

Application Data Access Layer Structure [↗](#)



Overall DAL structure

2_Base Solution:

Base.DAL.Contracts Project:

Contains interfaces defining the contracts for data access operations.

```

1 using Base.Domain.Contracts;
2
3 namespace Base.DAL.Contracts;
4
5 public interface IBaseEntityRepository<TEntity> : IBaseEntityRepository<TEntity, Guid>
6     where TEntity : class, IBaseEntityId
7 {
8 }
9
10 public interface IBaseEntityRepository<TEntity, TKey>
11     where TEntity : class, IBaseEntityId<TKey>

```

```

12     where TKey : IEquatable<TKey>
13 {
14     Task<IEnumerable<TEntity>> AllAsync();
15     Task<TEntity?> FindAsync(TKey id);
16     TEntity Add(TEntity entity);
17     TEntity Update(TEntity entity);
18     TEntity Remove(TEntity entity);
19     Task<TEntity?> RemoveAsync(TKey id);
20
21 }

```

• IBaseEntityRepository Interface:

The `IBaseEntityRepository` interface defines the contract for a repository responsible for performing CRUD (Create, Read, Update, Delete) operations on entities. It is designed to work with entities that have unique identifiers.

◦ IBaseEntityRepository<TEntity>:

- This generic interface extends the more specific interface `IBaseEntityRepository<TEntity, Guid>`. It is parameterized by the `TEntity` type, representing the entity type that the repository will handle.
- Constraints: `TEntity` must be a reference type (`class`) and must implement the `IBaseEntityId` interface. Additionally, the `TKey` type is specified as `Guid`.

◦ IBaseEntityRepository<TEntity, TKey>:

- This interface further specifies the repository contract by including methods for CRUD operations.
- Constraints: `TEntity` must be a reference type (`class`) and must implement the `IBaseEntityId<TKey>` interface. `TKey` is constrained to types that implement `IEquatable<TKey>`, ensuring that the identifier type supports equality comparison.

Methods:

◦ AllAsync():

- Returns all entities asynchronously.

◦ FindAsync(TKey id):

- Retrieves an entity by its unique identifier asynchronously.

◦ Add(TEntity entity):

- Adds a new entity to the repository.

◦ Update(TEntity entity):

- Updates an existing entity in the repository.

◦ Remove(TEntity entity):

- Removes an entity from the repository.

◦ RemoveAsync(TKey id):

- Removes an entity by its unique identifier asynchronously.

```

1 namespace Base.DAL.Contracts;
2
3 public interface IBaseUnitOfWork
4 {
5     Task<int> SaveChangesAsync();
6 }

```

• IBaseUnitOfWork Interface:

The `IBaseUnitOfWork` interface represents the contract for a unit of work in the data access layer of the application. It defines a method for saving changes asynchronously to the underlying data storage.

- The primary purpose of the `IBaseUnitOfWork` interface is to abstract the concept of a unit of work, which encapsulates a series of related operations performed within a single transaction. It provides a standardized way to manage transactions and commit changes to the database.

Method:

- `SaveChangesAsync()` :
 - Asynchronously saves all changes made in the current unit of work to the underlying database.
 - Returns: A `Task<int>` representing the asynchronous operation, which completes when the changes have been successfully saved to the database. The integer result typically represents the number of affected entities or rows.

Base.DAL.EF Project:

Implements the data access layer using Entity Framework Core.

```
1 using Base.DAL.Contracts;
2 using Base.Domain.Contracts;
3 using Microsoft.EntityFrameworkCore;
4
5 namespace Base.DAL.EF;
6
7 public class BaseEntityRepository : BaseEntityRepository,
8     IBaseEntityRepository
9     where TEntity : class, IBaseEntityId
10    where TDbContext : DbContext
11    {
12        public BaseEntityRepository(TDbContext dataContext) : base(dataContext)
13        {
14        }
15    }
16
17
18 public class BaseEntityRepository : IBaseEntityRepository
19     where TEntity : class, IBaseEntityId<TKey>
20     where TKey : struct, IEquatable<TKey>
21     where TDbContext : DbContext
22    {
23        protected TDbContext DbContext;
24        protected DbSet<TEntity> DbSet;
25
26        public BaseEntityRepository(TDbContext dataContext)
27        {
28            DbContext = dataContext ?? throw new ArgumentNullException(nameof(dataContext));
29            DbSet = DbContext.Set<TEntity>();
30        }
31
32        public virtual async Task<IEnumerable<TEntity>> AllAsync()
33        {
34            return await DbSet.ToListAsync();
35        }
36
37        public virtual async Task<TEntity?> FindAsync(TKey id)
38        {
39            return await DbSet.FindAsync(id);
40        }
41
42        public virtual TEntity Add(TEntity entity)
43        {
44            return DbSet.Add(entity).Entity;
45        }
46
47        public virtual TEntity Update(TEntity entity)
48        {
49        }
```

```

49         return DbSet.Update(entity).Entity;
50     }
51
52     public virtual TEntity Remove(TEntity entity)
53     {
54         return DbSet.Remove(entity).Entity;
55     }
56
57     public virtual async Task<TEntity?> RemoveAsync(TKey id)
58     {
59         var entity = await FindAsync(id);
60         return entity != null ? Remove(entity) : null;
61     }
62
63 }

```

• BaseEntityRepository<TEntity, TDbContext> Class:

The `BaseEntityRepository<TEntity, TDbContext>` class serves as a base implementation for repositories that handle data access operations for entities in the data access layer of the application. It provides common functionality for interacting with the underlying database context and performing CRUD operations on entities.

- The primary purpose of the `BaseEntityRepository` class is to encapsulate common data access logic and provide a consistent interface for accessing and manipulating entities within the context of a specific database context (`TDbContext`).

Generic Parameters:

- `TEntity` : The type of entity that the repository operates on. It must implement the `IBaseEntityId` interface, representing an entity with a unique identifier.
- `TDbContext` : The type of the database context used by the repository. It must derive from `DbContext` , representing the entity framework database context used for data access.

Constructor:

- `BaseEntityRepository(TDbContext dbContext)` :
 - Initializes a new instance of the `BaseEntityRepository` class with the specified database context (`dbContext`).

Properties:

- `DbContext` : The database context used by the repository to interact with the underlying database.
- `DbSet` : The `DbSet<TEntity>` representing the collection of entities of type `TEntity` in the database context.

Methods:

- `AllAsync()` :
 - Retrieves all entities of type `TEntity` from the database asynchronously.
- `FindAsync(TKey id)` :
 - Finds an entity with the specified primary key (`id`) asynchronously.
- `Add(TEntity entity)` :
 - Adds a new entity to the repository and returns the added entity.
- `Update(TEntity entity)` :
 - Updates an existing entity in the repository and returns the updated entity.
- `Remove(TEntity entity)` :
 - Removes an entity from the repository and returns the removed entity.
- `RemoveAsync(TKey id)` :
 - Removes an entity with the specified primary key (`id`) asynchronously.

```

1 using Base.DAL.Contracts;
2 using Microsoft.EntityFrameworkCore;

```

```

3
4 namespace Base.DAL.EF;
5
6 public abstract class BaseUnitOfWork<TAppDbContext> : IBaseUnitOfWork
7     where TAppDbContext : DbContext
8 {
9     protected readonly TAppDbContext UowDbContext;
10
11     protected BaseUnitOfWork(TAppDbContext dbContext)
12     {
13         UowDbContext = dbContext;
14     }
15
16     public async Task<int> SaveChangesAsync()
17     {
18         return await UowDbContext.SaveChangesAsync();
19     }
20 }
21

```

• BaseUnitOfWork<TAppDbContext> Class:

The `BaseUnitOfWork<TAppDbContext>` class represents a base implementation of the Unit of Work pattern for managing database transactions and coordinating changes across multiple repositories within the data access layer of the application. It provides a consistent interface for saving changes to the underlying database context asynchronously.

- The primary purpose of the `BaseUnitOfWork` class is to encapsulate the logic for managing database transactions and providing a centralized mechanism for committing changes to the database context.

Generic Parameters:

- `TAppDbContext` : The type of the application-specific database context used by the unit of work. It must derive from `DbContext`, representing the entity framework database context used for data access.

Constructor:

- `BaseUnitOfWork(TAppDbContext dbContext)` :
 - Initializes a new instance of the `BaseUnitOfWork` class with the specified database context (`dbContext`).

Properties:

- `UowDbContext` : The database context instance associated with the unit of work, which is used for managing database transactions and saving changes.

Methods:

- `SaveChangesAsync()` :
 - Asynchronously saves all changes made in this context to the underlying database and returns the number of state entries written to the database.

1_App Solution:

App.DAL.Contracts Project:

Contains interfaces (14 base + 1 uow) defining specific repository contracts for entities within the application domain.

```

1 using Base.DAL.Contracts;
2 using App.Domain;
3
4 namespace App.DAL.Contracts.Repositories;
5
6 public interface IDocumentSampleRepository : IBaseEntityRepository<DocumentSample>
7 {

```

```
8
9 }
```

- **Example IDocumentSampleRepository Interface:**

The `IDocumentSampleRepository` interface defines the contract for repositories handling operations related to the `DocumentSample` entity within the application's data access layer. It extends the `IBaseEntityRepository` interface, specifying additional methods or behaviors specific to managing `DocumentSample` entities.

- The primary purpose of the `IDocumentSampleRepository` interface is to provide a standardized set of methods for performing CRUD (Create, Read, Update, Delete) operations and querying `DocumentSample` entities within the data access layer.

```
1 using App.DAL.Contracts.Repositories;
2 using Base.DAL.Contracts;
3
4 namespace App.DAL.Contracts;
5
6 public interface IUnitOfWork : IBaseUnitOfWork
7 {
8     IDocumentSampleRepository DocumentSamples { get; }
9
10    IInternRepository Interns { get; }
11    IInternMentorshipRepository InternMentorships { get; }
12    IInternSupervisorRepository InternSupervisors { get; }
13    IInternMentorshipDocumentRepository InternMentorshipDocuments { get; }
14    IInternMentorshipUntilDateRepository InternMentorshipUntilDates { get; }
15
16    IEmployeeRepository Employees { get; }
17    IEmployeeMentorshipRepository EmployeeMentorships { get; }
18    IFactorySupervisorRepository FactorySupervisors { get; }
19    IEmployeeMentorshipDocumentRepository EmployeeMentorshipDocuments { get; }
20    IEmployeeMentorshipUntilDateRepository EmployeeMentorshipUntilDates { get; }
21
22    IMentorRepository Mentors { get; }
23    IMenteeSickLeaveRepository MenteeSickLeaves { get; }
24    ISickLeaveTypeRepository SickLeaveTypes { get; }
25 }
```

- **IUnitOfWork Interface:**

The `IUnitOfWork` interface represents a unit of work pattern within the application's data access layer. It defines a contract for a cohesive set of repository instances, each responsible for managing a specific type of entity. This interface extends the `IBaseUnitOfWork` interface, providing additional repository properties tailored to the application's domain entities.

- The primary purpose of the `IUnitOfWork` interface is to encapsulate multiple repository instances and provide a single entry point for accessing and coordinating database operations related to various domain entities.

Base Interface:

- `IBaseUnitOfWork`: Inherits from the `IBaseUnitOfWork` interface, which defines a standard method for saving changes to the underlying data store asynchronously.

App.DAL.EF Project:

Implements the data access layer using Entity Framework Core, tailored to the application's domain. Have in it `Migrations` folder and `Repositiroies`, which are used in the `UnitOfWork` class

```
1 using App.Domain;
2 using App.Domain.Identity;
3 using Microsoft.AspNetCore.Identity;
4 using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
```

```

5 using Microsoft.EntityFrameworkCore;
6
7 namespace App.DAL.EF;
8
9 public class AppDbContext : IdentityDbContext<AppUser, AppRole, Guid, IdentityUserClaim<Guid>, AppUserRole,
10     IdentityUserLogin<Guid>, IdentityRoleClaim<Guid>, IdentityUserToken<Guid>>
11 {
12     public DbSet<DocumentSample> DocumentSamples { get; set; } = default!;
13
14     public DbSet<Intern> Interns { get; set; } = default!;
15     public DbSet<InternMentorship> InternMentorships { get; set; } = default!;
16     public DbSet<InternSupervisor> InternSupervisors { get; set; } = default!;
17     public DbSet<InternMentorshipDocument> InternMentorshipDocuments { get; set; } = default!;
18     public DbSet<InternMentorshipUntilDate> InternMentorshipUntilDates { get; set; } = default!;
19
20     public DbSet<Employee> Employees { get; set; } = default!;
21     public DbSet<EmployeeMentorship> EmployeeMentorships { get; set; } = default!;
22     public DbSet<FactorySupervisor> FactorySupervisors { get; set; } = default!;
23     public DbSet<EmployeeMentorshipDocument> EmployeeMentorshipDocuments { get; set; } = default!;
24     public DbSet<EmployeeMentorshipUntilDate> EmployeeMentorshipUntilDates { get; set; } = default!;
25
26     public DbSet<Mentor> Mentors { get; set; } = default!;
27     public DbSet<MenteeSickLeave> MenteeSickLeaves { get; set; } = default!;
28     public DbSet<SickLeaveType> SickLeaveTypes { get; set; } = default!;
29
30
31     public AppDbContext(DbContextOptions options) : base(options)
32     {
33     }
34 }
35 }

```

• AppDbContext Class:

The `AppDbContext` class represents the database context for the application's data access layer. It inherits from `IdentityDbContext`, which is a part of ASP.NET Core Identity framework and is tailored for managing user authentication and authorization. This context is responsible for providing access to the database tables corresponding to various domain entities and identity-related tables.

- The primary purpose of the `AppDbContext` class is to define the structure and relationships of the application's database schema, including domain entities and identity-related entities.

DbSets:

- `DocumentSamples` : DbSet representing the `DocumentSample` entity in the database.
- `Interns` : DbSet representing the `Intern` entity in the database.
- `InternMentorships` : DbSet representing the `InternMentorship` entity in the database.
- `InternSupervisors` : DbSet representing the `InternSupervisor` entity in the database.
- `InternMentorshipDocuments` : DbSet representing the `InternMentorshipDocument` entity in the database.
- `InternMentorshipUntilDates` : DbSet representing the `InternMentorshipUntilDate` entity in the database.
- `Employees` : DbSet representing the `Employee` entity in the database.
- `EmployeeMentorships` : DbSet representing the `EmployeeMentorship` entity in the database.
- `FactorySupervisors` : DbSet representing the `FactorySupervisor` entity in the database.
- `EmployeeMentorshipDocuments` : DbSet representing the `EmployeeMentorshipDocument` entity in the database.
- `EmployeeMentorshipUntilDates` : DbSet representing the `EmployeeMentorshipUntilDate` entity in the database.
- `Mentors` : DbSet representing the `Mentor` entity in the database.
- `MenteeSickLeaves` : DbSet representing the `MenteeSickLeave` entity in the database.

- `SickLeaveTypes` : DbSet representing the `SickLeaveType` entity in the database.

Constructor:

- `AppDbContext(DbContextOptions options)` : Initializes a new instance of the `AppDbContext` class with the specified options, typically provided during application startup for configuring the database connection.

```

1 using Microsoft.EntityFrameworkCore;
2 using Microsoft.EntityFrameworkCore.Design;
3
4 namespace App.DAL.EF;
5
6 public class AppDbContextFactory : IDesignTimeDbContextFactory<AppDbContext>
7 {
8     public AppDbContext CreateDbContext(string[] args)
9     {
10         var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>();
11         optionsBuilder.UseNpgsql("Host=localhost;Port=5267;Database=downloads;Username=jhe;Password=opy");
12
13         return new AppDbContext(optionsBuilder.Options);
14     }
15 }

```

• AppDbContextFactory:

The `AppDbContextFactory` class is responsible for creating instances of the `AppDbContext` class during design time, typically used for database migrations and other design-time operations. It implements the `IDesignTimeDbContextFactory` interface provided by Entity Framework Core.

- The main purpose of the `AppDbContextFactory` class is to provide a factory method for creating instances of the `AppDbContext` class with specified `DbContextOptions` during design time.

Implemented Interface:

- `IDesignTimeDbContextFactory<AppDbContext>` : Indicates that the class can create instances of a specific `DbContext` during design time.

Methods:

- `CreateDbContext(string[] args)` : This method is called by Entity Framework Core to create a new instance of `AppDbContext` . It receives an array of string arguments (typically command-line arguments) but doesn't use them in this implementation. It configures `DbContextOptionsBuilder` with the necessary database connection information (in this case, PostgreSQL connection string) and returns a new instance of `AppDbContext` initialized with these options.

```

1 using App.DAL.Contracts;
2 using App.DAL.Contracts.Repositories;
3 using App.DAL.EF.Repositories;
4 using Base.DAL.EF;
5
6 namespace App.DAL.EF;
7
8 public class UnitOfWork : BaseUnitOfWork<AppDbContext>, IUnitOfWork
9 {
10     public UnitOfWork(AppDbContext dbContext) : base(dbContext)
11     {
12     }
13
14     private IDocumentSampleRepository? _documentSamples;
15     public IDocumentSampleRepository DocumentSamples =>
16         _documentSamples ?? new DocumentSampleRepository(UowDbContext);
17
18     private IInternRepository? _interns;

```

```

19     public IInternRepository Interns =>
20         _interns ?? new InternRepository(UowDbContext);
21
22     private IInternMentorshipRepository? _internMentorships;
23     public IInternMentorshipRepository InternMentorships =>
24         _internMentorships ?? new InternMentorshipRepository(UowDbContext);
25
26     private IInternSupervisorRepository? _internSupervisors;
27     public IInternSupervisorRepository InternSupervisors =>
28         _internSupervisors ?? new InternSupervisorRepository(UowDbContext);
29
30     private IInternMentorshipDocumentRepository? _internMentorshipDocuments;
31     public IInternMentorshipDocumentRepository InternMentorshipDocuments =>
32         _internMentorshipDocuments ?? new InternMentorshipDocumentRepository(UowDbContext);
33
34     private IInternMentorshipUntilDateRepository? _internMentorshipUntilDates;
35     public IInternMentorshipUntilDateRepository InternMentorshipUntilDates =>
36         _internMentorshipUntilDates ?? new InternMentorshipUntilDateRepository(UowDbContext);
37
38     private IEmployeeRepository? _employees;
39     public IEmployeeRepository Employees =>
40         _employees ?? new EmployeeRepository(UowDbContext);
41
42     private IEmployeeMentorshipRepository? _employeeMentorships;
43     public IEmployeeMentorshipRepository EmployeeMentorships =>
44         _employeeMentorships ?? new EmployeeMentorshipRepository(UowDbContext);
45
46     private IFactorySupervisorRepository? _factorySupervisors;
47     public IFactorySupervisorRepository FactorySupervisors =>
48         _factorySupervisors ?? new FactorySupervisorRepository(UowDbContext);
49
50     private IEmployeeMentorshipDocumentRepository? _employeeMentorshipDocuments;
51     public IEmployeeMentorshipDocumentRepository EmployeeMentorshipDocuments =>
52         _employeeMentorshipDocuments ?? new EmployeeMentorshipDocumentRepository(UowDbContext);
53
54     private IEmployeeMentorshipUntilDateRepository? _employeeMentorshipUntilDates;
55     public IEmployeeMentorshipUntilDateRepository EmployeeMentorshipUntilDates =>
56         _employeeMentorshipUntilDates ?? new EmployeeMentorshipUntilDateRepository(UowDbContext);
57
58     private IMentorRepository? _mentors;
59     public IMentorRepository Mentors =>
60         _mentors ?? new MentorRepository(UowDbContext);
61
62     private IMenteeSickLeaveRepository? _menteeSickLeaves;
63     public IMenteeSickLeaveRepository MenteeSickLeaves =>
64         _menteeSickLeaves ?? new MenteeSickLeaveRepository(UowDbContext);
65
66     private ISickLeaveTypeRepository? _sickLeaveTypes;
67     public ISickLeaveTypeRepository SickLeaveTypes =>
68         _sickLeaveTypes ?? new SickLeaveTypeRepository(UowDbContext);
69
70 }
71

```

- **UnitOfWork Class:**

The `UnitOfWork` class serves as the concrete implementation of the unit of work pattern for the application's data access layer. It coordinates the repository instances and provides a single entry point to interact with the database. This class extends the `BaseUnitOfWork` class and implements the `IUnitOfWork` interface.

- The main purpose of the `UnitOfWork` class is to manage the lifecycle of the database context (`AppDbContext`) and provide access to repository instances for various entities within the application.

Inheritance:

- `BaseUnitOfWork<AppDbContext>`: Inherits from the base unit of work class provided by the Base solution, specialized for the `AppDbContext`.

Implemented Interface:

- `IUnitOfWork`: Provides the contract for the unit of work pattern, exposing repository properties for different entity types.

Properties:

- The `UnitOfWork` class contains properties for each repository interface defined in the `IUnitOfWork` interface. Each repository property is initialized lazily, ensuring that only one instance of each repository is created per unit of work instance.

Constructor:

- `UnitOfWork(AppDbContext dbContext)`: Constructor that initializes the `UnitOfWork` class with an instance of `AppDbContext`, which is then passed to the base class constructor.

```

1 using App.DAL.Contracts.Repositories;
2 using App.Domain;
3 using Base.DAL.Contracts;
4 using Base.DAL.EF;
5
6 namespace App.DAL.EF.Repositories;
7
8 public class DocumentSampleRepository : BaseEntityRepository<DocumentSample, AppDbContext>, IDocumentSampleRepository
9 {
10     public DocumentSampleRepository(AppDbContext dbContext) : base(dbContext)
11     {
12     }
13 }

```

• App.DAL.EF.Repositories: Example DocumentSampleRepository Class:

The `DocumentSampleRepository` class represents a repository responsible for interacting with the `DocumentSample` entity within the application's data access layer. It implements the `IDocumentSampleRepository` interface and extends the `BaseEntityRepository` class provided by the Base solution, specialized for the `DocumentSample` entity and the `AppDbContext`.

- The primary purpose of the `DocumentSampleRepository` class is to encapsulate database operations related to the `DocumentSample` entity, such as querying, adding, updating, and deleting `DocumentSample` records.

Inheritance:

- `BaseEntityRepository<DocumentSample, AppDbContext>`: Inherits from the base entity repository class provided by the Base solution, specialized for the `DocumentSample` entity and the `AppDbContext`.

Implemented Interface:

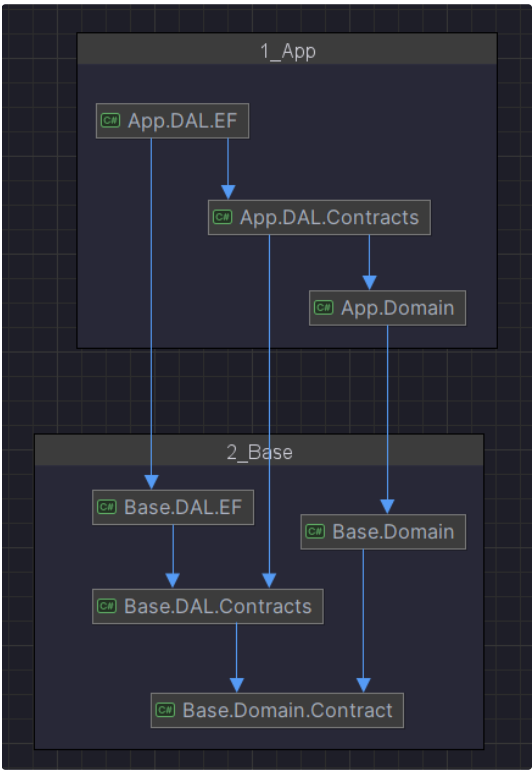
- `IDocumentSampleRepository`: Declares the contract for repository operations specific to the `DocumentSample` entity.

Constructor:

- `DocumentSampleRepository(AppDbContext dbContext)`: Constructor that initializes the `DocumentSampleRepository` class with an instance of `AppDbContext`, which is then passed to the base class constructor.

The data access layer structure ensures separation of concerns and facilitates efficient data access operations within the application. It leverages the unit of work pattern and generic repository implementations to provide a consistent and scalable approach to database interactions.

Domain-DAL Relationship [↗](#)



Domain-DAL Relationship

