

PG3302

SOFTWARE DESIGN EXAM

By Elin Eujung Park & Ivan Kovals

Table of contents

Introduction

- 0. About our project
- 1. The Process of solving the exam
- 2. About our solution
 - 2.1 Technique and tools for collaboration
 - 2.2 UML class diagram for our solution
 - 2.3 Code review & analysis
 - 2.4 SOLID principles
 - 2.5 Layering
 - 2.6 Data storage
 - 2.7 Design patterns
 - 2.8 Event-based code
 - 2.9 Testing
 - 2.10 Refactoring
- 3. User Interface
 - 3.1 Design
 - 3.2 Usage
 - 3.3 Examples
- 4. Limitation
- 5. Source reference

0. About our project

Our exam project, "**Digital Emotion Diary**," is a diary application designed to allow users to publish, manage, and engage with diary entries. Users can view public entries from other app users while maintaining control over their private entries. The initial concept of a "Digital Diary" was proposed by Elin, and Ivan enriched the idea by introducing the "emotional" aspect and adding tags to enhance entry categorization.

The application was designed with the following key features in mind:

1. Emotion and Background Color

Given the emphasis on emotions in the app's name, each diary entry is associated with a background color that represents a specific emotion. For example:

- **Happy mood** – Yellow
- **Energized** – Orange
- **Stressed** – Red
- **Annoyed** – Black

If the application were to gain widespread use, we could conduct user surveys to refine these color-emotion combinations and potentially expand the options.

2. Image Upload

To make entries more expressive, users have the option to upload at least one image per diary entry. Images can convey emotions or moments better than text in some cases. This feature is optional, allowing flexibility in user preferences.

3. Likes and Comments

To increase user engagement, the app includes features for liking and commenting on diary entries:

- **Likes:** A diary entry can receive likes from multiple users.
- **Comments:** Users can leave comments on entries to interact with one another.

4. Tags

Tags enhance the organizational aspect of the app. We implemented a **many-to-many relationship** between tags and diary entries:

- A single diary entry can have multiple tags.
- A single tag can be associated with multiple entries.

5. Filtering Options

Users can filter their entries by mood or tags, enabling them to revisit specific emotional states or topics easily.

6. Command-Line Interface (CLI)

The app features a menu-driven interface in the command line, allowing users to navigate and execute operations such as creating, updating, and filtering entries.

Project Kickoff

After finalizing the core idea, we began planning our workflow using **Notion** and **Trello** to organize our actions and track progress.

Database Initialization

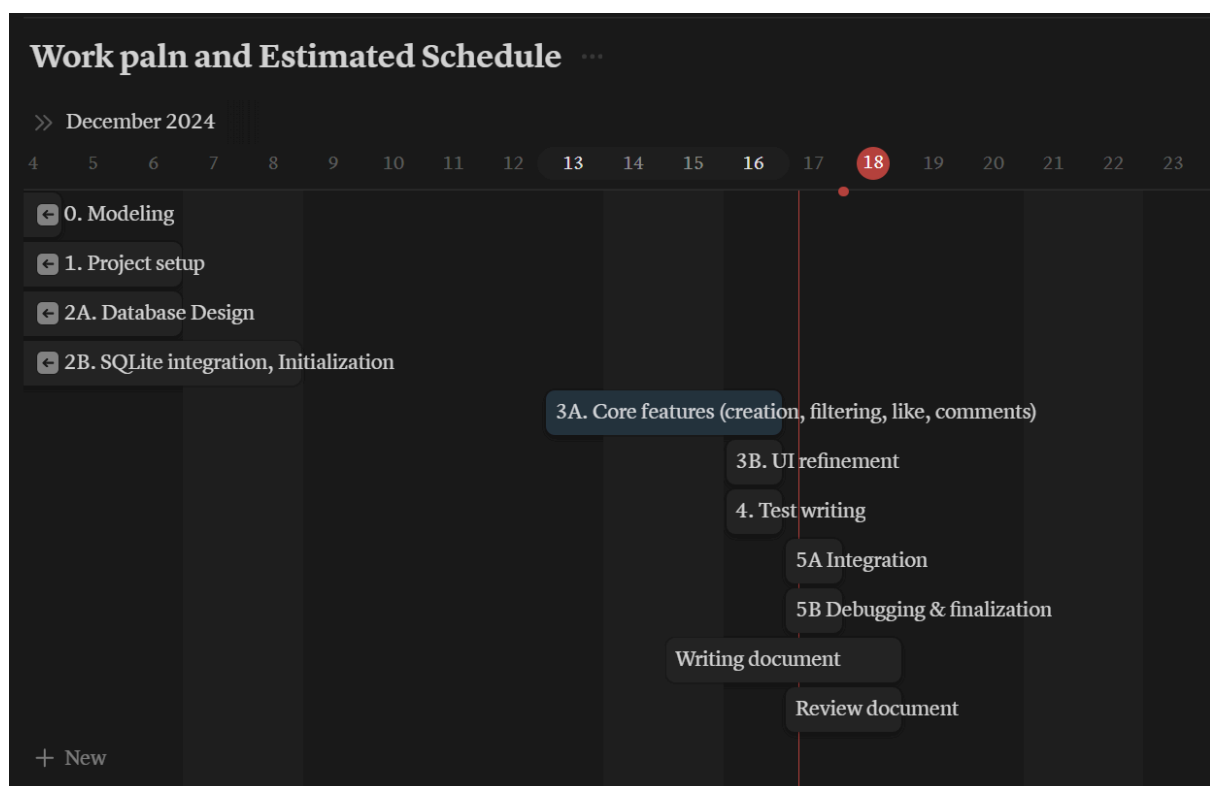
For this project, we initialized the database with two static users, laying the foundation for user-related functionalities.

	123 Id	A-Z UserName	A-Z Email	A-Z Password	A-Z ProfileImagePath
1	1	Elin	elin@gmail.com	pass1	elinProfile.png
2	2	Ivan	ivan@gmail.com	pass2	ivanProfile.png

1. The Process of solving the exam

After our initial meeting, we planned the working process and created a schedule as below, taking into account the upcoming vacation period. However, the work plan had to be adjusted after our flight schedules, which left us with significantly less time to work on the project.

We made a concerted effort to complete most of the work before the vacation started. Unfortunately, our progress was impacted by overlapping exams during the same period and unexpected illness for Ivan, which made it challenging to stick to the original plan. We used Notion and Trello for planning and tracking a progress on exam:



Detailed planning in Notion as below:

1. Project Setup

- ✓ Initialize Console Application.
- ✓ Create key classes and files (e.g., `Program.cs`, `DiaryEntry.cs`).
- ✓ Create ~~NU~~Unit Test Project and set up the environment.
- ✓ Prepare SQLite DB file and perform basic connection tests.

2. Database Design and Configuration

A. Data Model Design

- Design SQLite tables:

- ✓ `DiaryEntries` : ID, date, emotion, background-color, visibility...
- + :: ✓ `Likes` : ID, ~~DiaryEntryID~~, ~~UserID~~.
- ✓ `Comments` : ID, ~~DiaryEntryID~~, content.

B. Database Integration and Initialization

- ✓ Write SQLite connection and integration code.
- ✓ Create database initialization logic.
- ✓ Populate database with sample data.

3. Feature Implementation

A. Core Feature Implementation

- ✓ Implement diary entry creation.
- ✓ Implement `background-color` selection and storage.
- ✓ Implement filtering by `emotion` (monthly/yearly).
- ✓ Implement filtering by `tag`.
- ✓ Implement `public/private` settings.

2. About our Solution

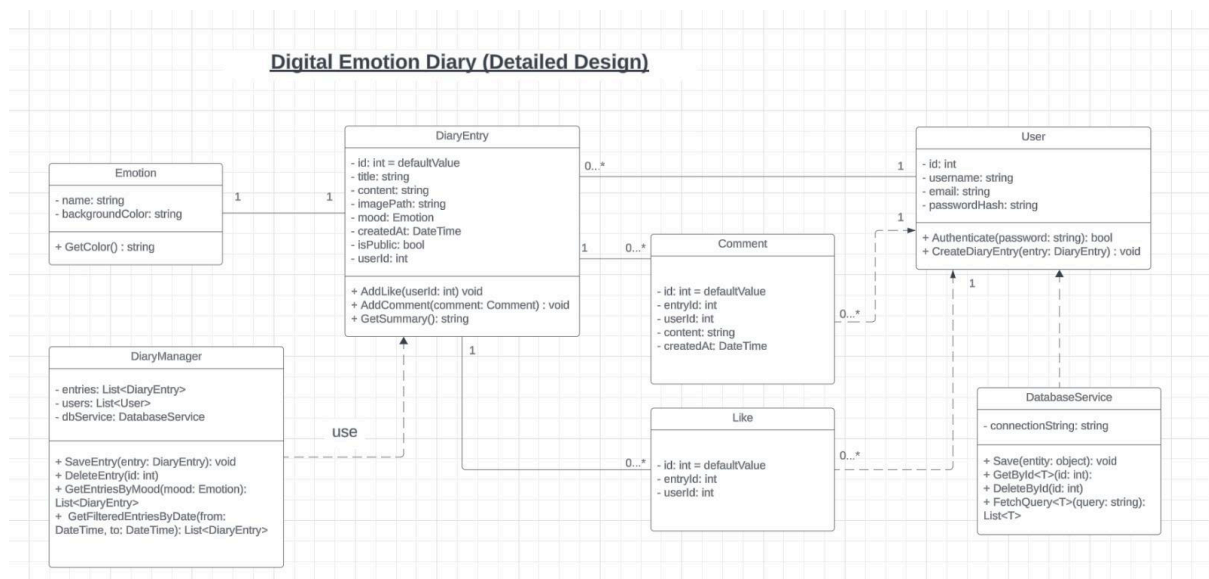
2.1 Technique and tools for collaboration

- **Github:** We used Github for version control, benefiting from C#'s built-in Git integration for seamless usage. To minimize merge conflicts, we tried not to work on the same directory to avoid collision.
- **Discord:** Meetings and code collaborations were conducted via Discord. Its screen-sharing feature proved especially useful when working from different locations, enabling real-time collaboration and troubleshooting.
- **Notion:** Notion served as a central tool for logging our process throughout the project. We consolidated all ideas, plans, processes and documentation into a single page.
- **Trello:** Trello was used as a supplementary tool for planning and tracking our progress, allowing us to visualise our tasks and milestones during the exam.
- **Lucid chart:** We experimented with UML (Unified Modeling Language) Class Diagram using Lucidchart to design and refine the structure of our application.
- **Google docs:** Documentation for the project was written collaboratively on Google Docs. Its simultaneous editing features made it a practical choice for teamwork.
- **Code review:** Although we initially planned to conduct systematic peer reviews of our code, time constraints made it challenging to implement this process effectively.
- **Code comments :** We prioritized adding concise comments in the code to separate sections and explain the intention behind specific implementations, ensuring clarity for future reference.

2.2 UML class diagram for our solution

2.2.0 Initial version of UML class diagram

In the Initial stage of the project, we planned for users to create DiaryEntry with the ability to associate an Emotion and a Background color to reflect their mood. Additionally users would be able to interact with these diary entries by adding Comments and Likes. To support these features, we included two core components (DiaryManager and DatabaseService) in our design.

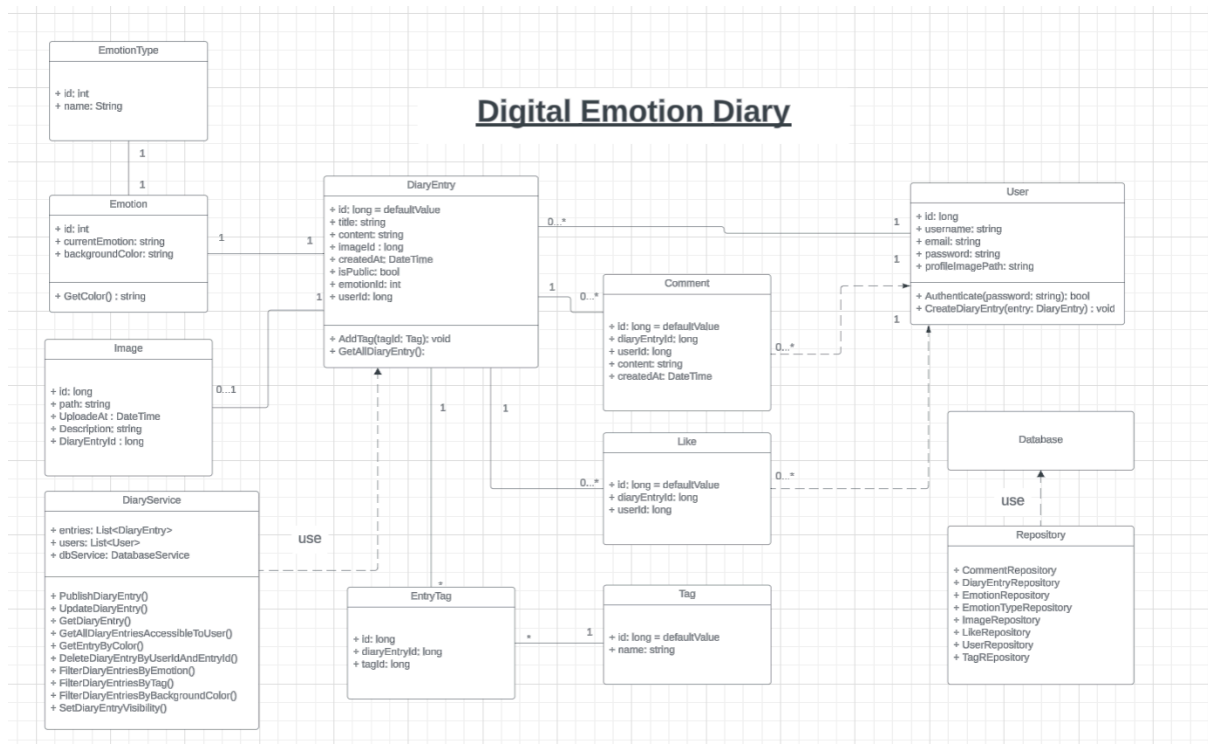


2.2.1 Final version of UML class diagram

During the development process, we decided to expand our initial concept by adding the Tag feature. To manage the many-to-many relationship between DiaryEntry and Tag (where multiple diary entries can have multiple tags and vice versa), we created a join class called EntryTag.

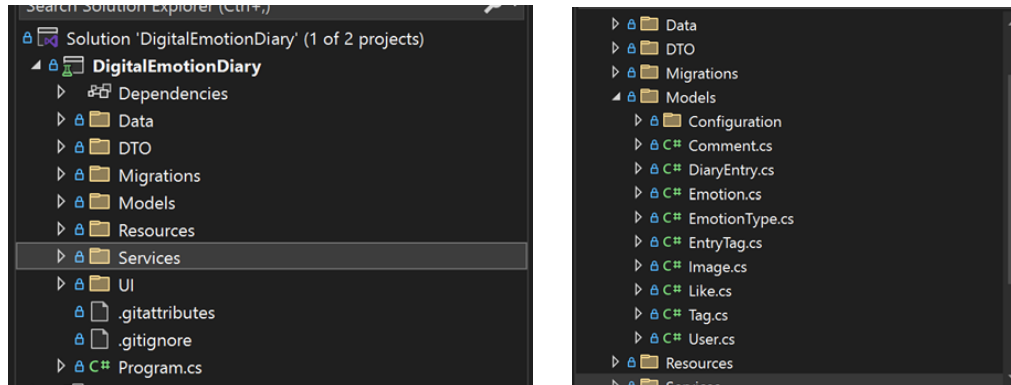
We also enhanced the project by allowing users to add images to their diary entries, enriching the user experience. For handling emotion more effectively, we introduced an `EmotionType` class, which categorizes and organizes different emotions. `EmotionType` class could be enum, but we decided to switch field value to string for future scalability,

Additionally, users have the option to add a profile picture to their account. However, this functionality is managed without a separate class, simplifying its implementation.



2.3 Code review & analysis

Next step is writing entities in C# using VisualStudio 2022 IDE. We will try to be short while going through these classes. First, I would like to start with folder structure, where our classes are stored inside folder Models:



2.3.1 User class

```
namespace DigitalEmotionDiary.Models
{
    21 references
    public class User
    {
        7 references
        public long Id { get; set; }
        5 references
        public required string UserName { get; set; }
        5 references
        public required string Email { get; set; }
        5 references
        public required string Password { get; set; }
        2 references
        public string? ProfileImagePath { get; set; }

        // Navigation property for related DiaryEntries
        2 references
        public ICollection<DiaryEntry> DiaryEntries { get; set; } = new List<DiaryEntry>(); // One to Many
        2 references
        public ICollection<Comment> Comments { get; set; } = new List<Comment>(); // One to Many
        2 references
        public ICollection<Like> Likes { get; set; } = new List<Like>();
    }
}
```

A simple class with fields Id, UserName, Email, Password and ProfileImagePath (optional). We used properties with getter and setter for each variable. Required keyword is used to make sure variables are being set during object initialization.

Since C# is a new language for us, we keep fields public for now, to provide availability across the application. The plan is to change them to private or protected later, if we get enough time, and ensure encapsulation by that.

Fields after // Navigation property are used for handling relations in DB.

For future updates we could think about hashing password, and simple validation for some fields, probably using specification patterns.

2.3.2 DiaryEntry class

```
namespace DigitalEmotionDiary.Models
{
    31 references
    public class DiaryEntry
    {
        7 references
        public long Id { get; set; }
        5 references
        public required string Title { get; set; }
        6 references
        public required string Content { get; set; }

        7 references
        public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
        6 references
        public bool IsPublic { get; set; } = false;

        // Foreign key
        12 references
        public long UserId { get; set; }
        7 references
        public int EmotionId { get; set; }

        // Optional Foreign Key
        4 references
        public long? ImageId { get; set; } = null;

        // Navigation Properties
        2 references
        public User User { get; set; }
        1 reference
        public Emotion Emotion { get; set; }
        2 references
        public Image Image { get; set; }
        3 references
        public ICollection<EntryTag> EntryTags { get; set; } = new List<EntryTag>(); // Middle table for Many to many
        2 references
        public ICollection<Comment> Comments { get; set; } = new List<Comment>(); // One to Many
        2 references
        public ICollection<Like> Likes { get; set; } = new List<Like>();
    }
}
```

DiaryEntry class is the biggest entity class in our application, though it is quite straightforward and easy to understand. Core fields are Id, Title and Content, and CreatedAt & IsPublic fields help to save important info about objects. *Properties* are used here also. Foreign key section contains fields used to store many-to-one relations with tables User and Emotion. ImageId presents here as well, that is a one-to-one relation.

Also here we handle relations in our DB with fields declared in // Navigation properties section.

Can make a note, that in Java we don't need to use both User and UserId fields for the same relation, it is enough with just User, which is handled by JPA. Maybe it also can be done in C#.

2.3.3 EntryTag class

```
namespace DigitalEmotionDiary.Models
{
    14 references
    public class EntryTag
    {
        4 references
        public long Id { get; set; }
        7 references
        public long DiaryEntryId { get; set; }
        6 references
        public long TagId { get; set; }

        // Navigation Property
        1 reference
        public DiaryEntry? DiaryEntry { get; set; } // One to many
        3 references
        public Tag? Tag { get; set; } // One to many
    }
}
```

That little class is worth mentioning because it actually is a join table, which connects Tag and DiaryEntry. Properties Id, DiaryEntryId and TagId are being used here. That class does it job for now, in future we could improve it in following areas:

- DiaryEntry? and Tag? are optional for now. Probably these two need to be defined anyway, so that tuple in that entity makes sense.
- Am a bit in doubt of having Id here as well, since DiaryEntryId and TagId could be used as a compound primary key.

We can skip going through other classes in the Models folder to save time. Worth to mention that db relations follow Entity Framework as well as Domain Design pattern.

2.3.4 UserConfiguration class

```
public class UserConfiguration : IEntityTypeConfiguration<User>
{
    0 references
    public void Configure(EntityTypeBuilder<User> builder)
    {
        builder.HasKey(User u => u.Id);

        builder.HasMany(User u => u.DiaryEntries)
            .WithOne(DiaryEntry u => u.User)
            .HasForeignKey(DiaryEntry d => d.UserId) // DiaryEntry should have UserId as a foreign key
            .OnDelete(DeleteBehavior.Cascade);

        builder.HasMany(User u => u.Comments)
            .WithOne(Comment c => c.User)
            .HasForeignKey(Comment c => c.UserId)
            .OnDelete(DeleteBehavior.Restrict);

        builder.HasMany(User u => u.Likes)
            .WithOne(Like l => l.User)
            .HasForeignKey(Like l => l.UserId)
            .OnDelete(DeleteBehavior.Restrict);

        builder.HasData(
            new User
            {
                Id = 1,
                UserName = "Elin",
                Email = "elin@gmail.com",
                Password = "pass1",
                ProfileImagePath = "elinProfile.png",
            },
            new User
            {
                Id = 2,
                UserName = "Ivan",
                Email = "ivan@gmail.com",
                Password = "pass2",
                ProfileImagePath = "ivanProfile.png",
            }
        );
    }
}
```

UserConfiguration class is an Entity Framework Core API configuration for user entity. At the beginning it defines an Id as a primary key. Further it sets one-to-many relationships, with specific cascading behavior on deletion. Deleting a user will delete that user's diary entries, but will not delete the user's comments and likes, it's a design choice. At the end of class we add some initial data to our table, these credentials can be used for testing the app. Without even noticing that we used a Data Seeder Pattern here. I think a Builder Pattern is also used here, though indirectly.

2.3.5 CommentDTO class.

```
namespace DigitalEmotionDiary.DTO
{
    0 references
    public class CommentDTO
    {
        0 references
        public long Id { get; set; }
        0 references
        public string Content { get; set; }
        0 references
        public DateTime CreatedAt { get; set; }
        0 references
        public long UserId { get; set; }
        0 references
        public long DiaryEntryId { get; set; }

        // Related User Information
        0 references
        public string UserName { get; set; }
    }
}
```

Another common pattern is to use DTO (data transfer object). Here is an example from our project, CommentDTO class. With dto we can decouple comment class with all the fields from database version to external API communication version, and use only fields we would like to show in CLI. It can also be defined as separation of concerns pattern.

2.3.6 LikeRepository class

```
9 namespace DigitalEmotionDiary.Data.Repositories
10 {
11     3 references
12     public class LikeRepository
13     {
14         private readonly DigitalEmotionDiaryDbContext _dbContext;
15
16         0 references
17         public LikeRepository(DigitalEmotionDiaryDbContext dbContext)
18         {
19             _dbContext = dbContext;
20         }
21
22         1 reference
23         public void AddLikeToEntry(long entryId, long userId)
24         {
25             var Like? existingLike = GetLike(entryId, userId);
26             if (existingLike == null)
27             {
28                 var Like? like = new Like
29                 {
30                     DiaryEntryId = entryId,
31                     UserId = userId
32                 };
33                 _dbContext.Like.Add(like);
34             }
35         }
36
37         1 reference
38         public void RemoveLikeFromEntry(long entryId, long userId)
39         {
40             var Like? like = GetLike(entryId, userId);
41             _dbContext.Like.Remove(like);
42         }
43
44         2 references
45         public Like? GetLike(long diaryEntryId, long userId)
46         {
47             return _dbContext.Like.FirstOrDefault(Like l => l.DiaryEntryId == diaryEntryId && l.UserId == userId);
48         }
49     }
50 }
```

```

46
47 1 reference
48 public IEnumerable<Like> GetLikesByDiaryEntryId(long diaryEntryId)
49 {
50     return _dbContext.Like.Where(Like l => l.DiaryEntryId == diaryEntryId).ToList();
51 }
52
53 1 reference
54 public IEnumerable<Like> GetLikesByUserId(long userId)
55 {
56     return _dbContext.Like.Where(Like l => l.UserId == userId).ToList();
57 }
58
59 1 reference
60 public long CountLikesByDiaryEntryId(long diaryEntryId)
61 {
62     return _dbContext.Like.Count(Like l => l.DiaryEntryId == diaryEntryId);
63 }
64
65 2 references
66 public bool Exists(long diaryEntryId, long userId)
67 {
68     return _dbContext.Like.Any(Like l => l.DiaryEntryId == diaryEntryId && l.UserId == userId);
69 }
70
71 2 references
72 public void SaveChanges()
73 {
74     _dbContext.SaveChanges();
75 }

```

Here we can look at LikeRepository. That class takes care about getting the data from our database. It is a common practice in software design to use repo - service - controller layer. Repository pattern and again Separation of concerns pattern are used here. Some concrete methods from that class takes care about registering a new like, removing existing like, as well as getting likes by userId, diaryEntryId, or total amount of likes for that diaryEntry.

2.3.7 LikeService class

```

10 namespace DigitalEmotionDiary.Services
11 {
12     1 reference
13     public class LikeService
14     {
15         private readonly LikeRepository _likeRepository;
16
17         0 references
18         public LikeService(
19             LikeRepository likeRepository
20         )
21         {
22             _likeRepository = likeRepository;
23         }
24
25         0 references
26         public void LikeEntry(long diaryEntryId, long userId)
27         {
28             if(_likeRepository.Exists(diaryEntryId, userId))
29             {
30                 throw new InvalidOperationException("A like for this diary entry by the user already exists.");
31             }
32
33             var Like? like = new Like
34             {
35                 DiaryEntryId = diaryEntryId,
36                 UserId = userId
37             };
38
39             _likeRepository.AddLikeToEntry(diaryEntryId, userId);
40             _likeRepository.SaveChanges();
41         }
42     }
43 }

```

```

41 | 0 references
42 | public void UnlikeEntry(long diaryEntryId, long userId)
43 | {
44 |     _likeRepository.RemoveLikeFromEntry(diaryEntryId, userId);
45 |     _likeRepository.SaveChanges();
46 | }
47 | 0 references
48 | public IEnumerable<Like> GetLikesByDiaryEntryId(long diaryEntryId)
49 | {
50 |     return _likeRepository.GetLikesByDiaryEntryId(diaryEntryId);
51 | }
52 | 0 references
53 | public IEnumerable<Like> GetLikesByUserId(long userId)
54 | {
55 |     return _likeRepository.GetLikesByUserId(userId);
56 | }
57 | 0 references
58 | public long CountLikesByDiaryEntryId(long diaryEntryId)
59 | {
60 |     return _likeRepository.CountLikesByDiaryEntryId(diaryEntryId);
61 | }
62 | 0 references
63 | public bool Exists(long diaryEntryId, long userId)
64 | {
65 |     return _likeRepository.Exists(diaryEntryId, userId);
66 | }
67 | }
68 |

```

And here comes example of service layer class, again LikeService class, for consistency. Service layer is kind of a bridge between repo layer and business logic. It doesn't happen much here, looks like we just are calling methods from repo and returning a result. Nevertheless, it is important to have a service layer. We could add some logic before returning the result from repo for example. We could replace the repo class with another one, which connects to another database for instance. That would be an example of the Dependency Inversion Principle.

2.3.8 Business logic & UI

Classes LoginService, Command and UserInterface takes care of interacting with users. They are quite big and we can skip code review for them to save space. Usually we would use the Controller layer which takes care of required API endpoints and communication between backend and frontend. But our task for that exam was to do CLI communication. LoginService provides login and registration of users. UserInterface acts as an entry point to our application, welcomes users and calls relevant services to get or fetch the required data. Command class takes care of user command and appropriate arguments.

2.4 SOLID principles

2.4.1 Single Responsibility Principle (SRP)

"Each class should only focus on one area of responsibility."

Examples

1. DiaryEntry class focuses just on managing diary entries. A dedicated class for that purpose.
2. Even though there are many methods with the same name both in Repositories and Services. We separate the responsibility clearly. Repositories are mainly focused on database management. Services are focused on interaction between database and UI. For example, we created a SaveChange method in Repositories to make the repositories take charge of database handling. Service layers save changes by calling this method through repositories.

```
3 references
public void SaveChanges()
{
    _dbContext.SaveChanges();
}
```

```
References
public void PostComment(long diaryEntryId, long userId, string content)
{
    var Comment? comment = new Comment
    {
        Content = content,
        CreatedAt = DateTime.Now,
        DiaryEntryId = diaryEntryId,
        UserId = userId
    };

    _commentRepository.AddComment(comment);
    _commentRepository.SaveChanges();
}
```

2.4.2 Open/Closed Principle (OCP)

"Existing code should be able to be extended with more functionality, without having to rewrite existing code." -for instance using of inheritance (interfaces, abstract classes, and/or virtual methods)

Example

1. Command class has some useful user commands now. It could be expanded in future without modifying the original version.

2.4.3 Liskov Substitution Principle (LSP)

"Subclasses can replace base classes without "blowing up" the execution of the program"

Example: I am not sure where in our app we have used that principle now. Probably we could take into use some interfaces, like IUserService, and have LSP that way.

2.4.4 Interface Segregation Principle (ISP)

"No client should be forced to depend on methods it does not use."

Example: through our app we don't implement methods we don't use. So it is done kind of implicitly. Areas to improve - we could introduce more interfaces and use them.

2.4.5 Dependency Inversion Principle (DIP)

"High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces)" - for instance using of "Dependency injection", "Factory design"

"Abstraction should not depend on details. Details (concrete implementations) should depend on abstractions."

-> Program to an interface, not to a class

Am not sure if we use the DIP principle in our app now.

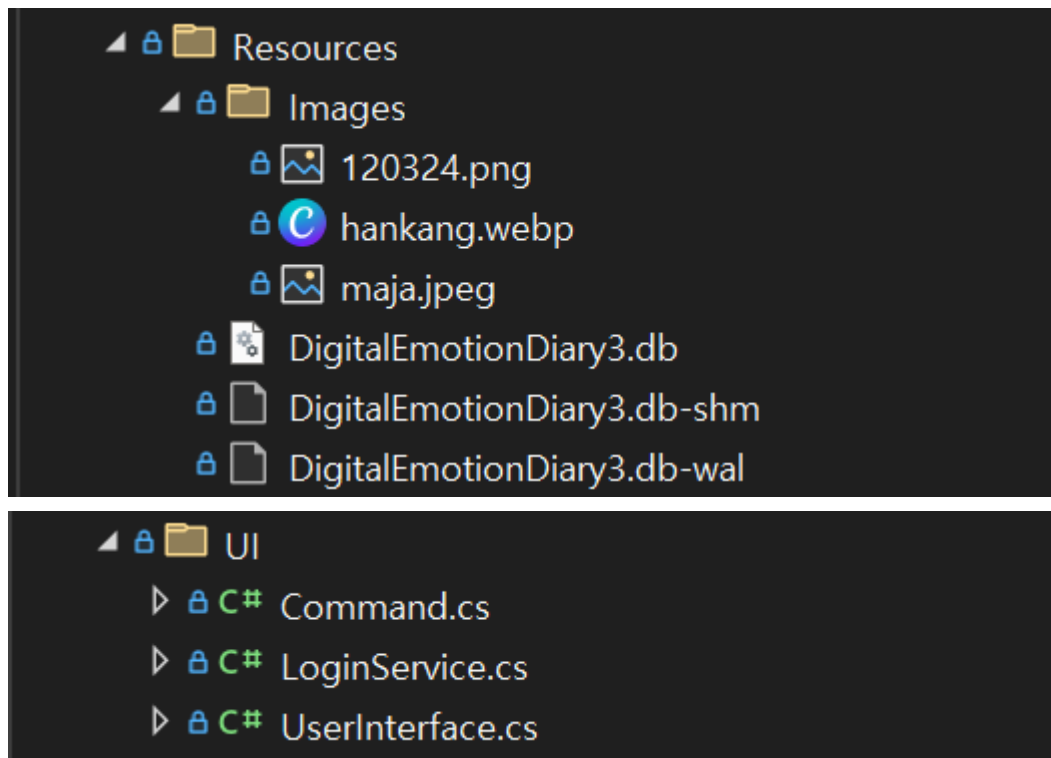
2.5 Layering

Our project's layering structure is as following:

1. User Interface (UI) Layer
2. Application Layer
3. Domain/Business Layer
4. Data Layer

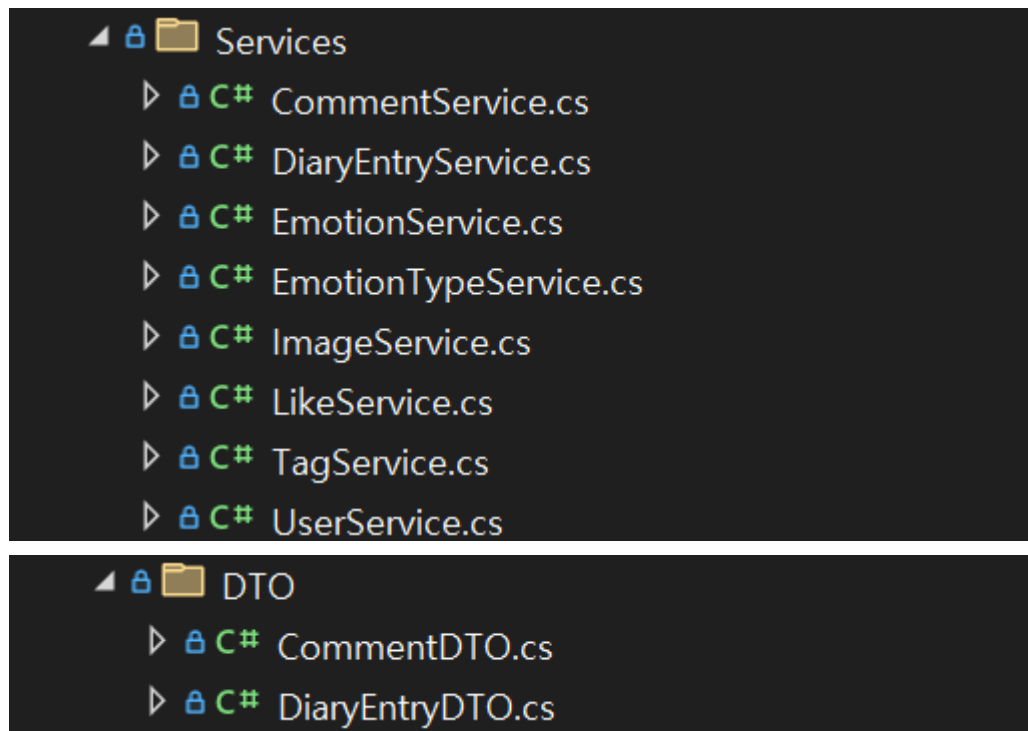
1. User Interface (UI) Layer

- Directory: **Resource, UI**
- **Resource** directory includes the Images folder (There are three static data), which are designed for interacting with UI.



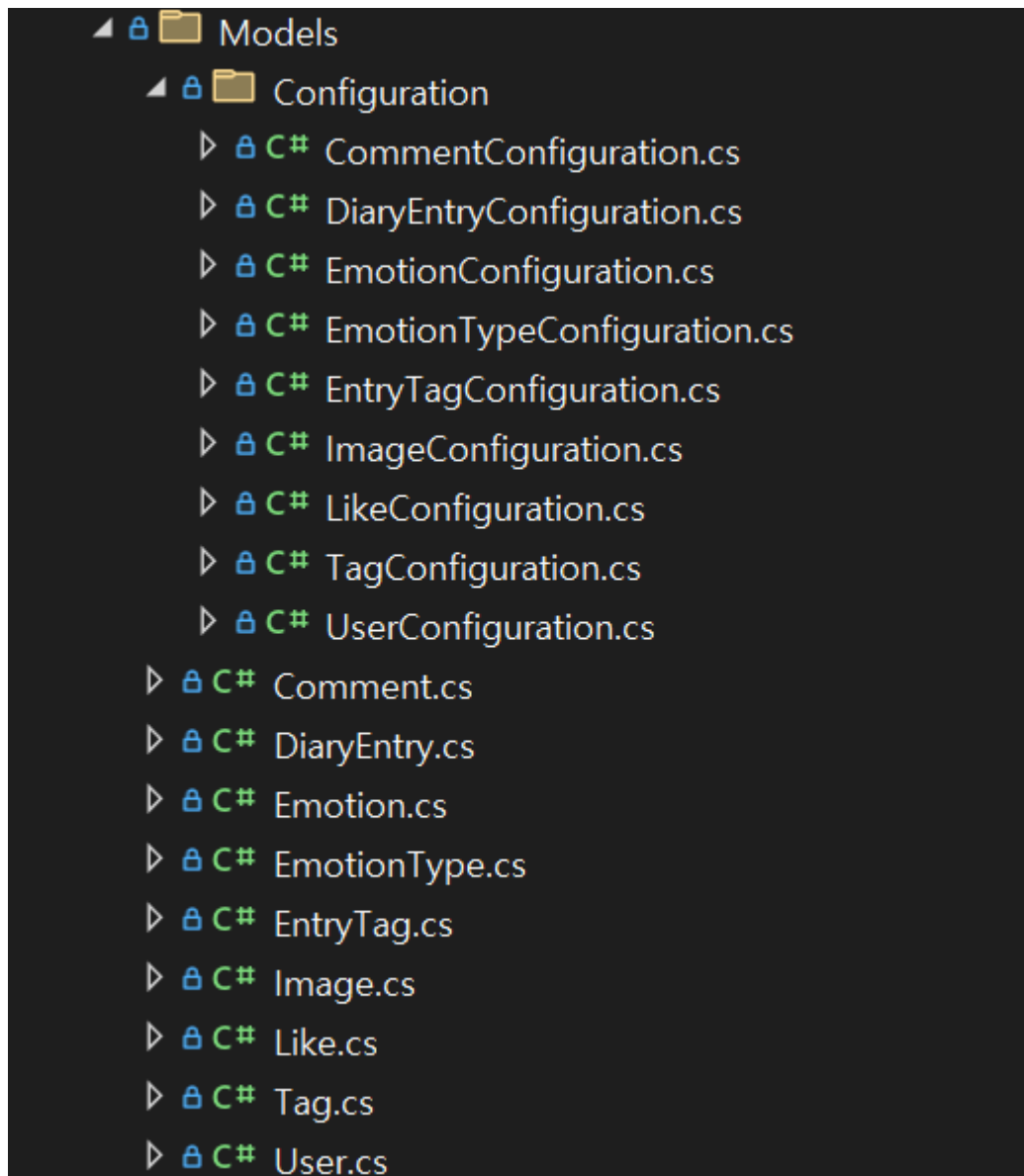
2. Application Layer

- Directory: **Services, DTO**
- **Services** implement key functionalities such as "PublishDiaryEntry()", "UpdateDiaryEntry", "GetDiaryEntry", "FilterDiaryEntriesByEmotion" etc.
- **DTO (Data Transfer Objects)** used for transferring data between layers.



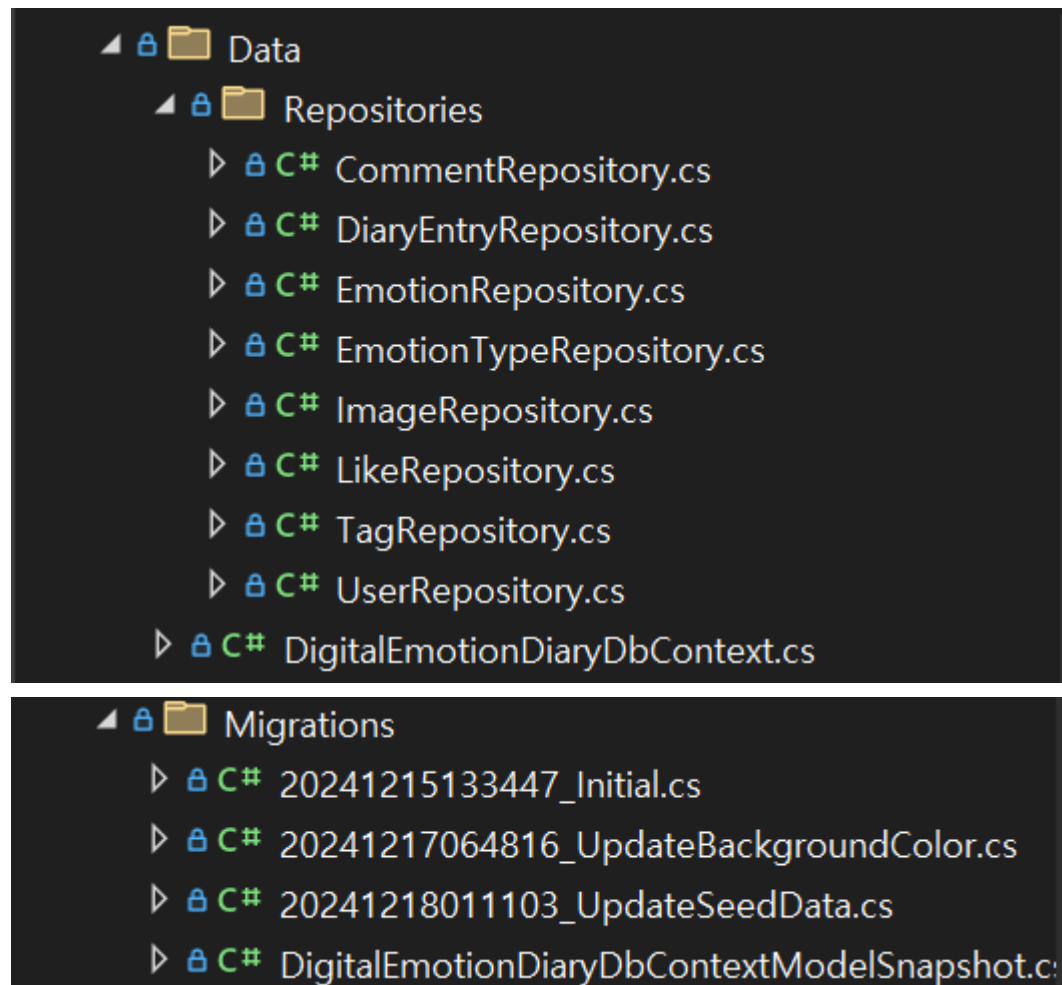
3. Domain/Business Layer

- Directory: **Models, Models/Configuration**
- **Models** define core information to create tables in the database.
- **Configuration** specifies how the entity properties are mapped to database columns and include static data for each entity/class.



4. Data Layer

- Directory: **Data, Migration**
- **Data** is responsible for database interaction and management. Repositories handle CRUD functions and custom functions for database management.
- **Migration** store scripts to create SQLite database schemas.



Layering structure as a whole:

Solution Explorer





Search Solution Explorer (Ctrl+;)



 Solution 'DigitalEmotionDiary' (2 of 2 projects)

 DigitalEmotionDiary

►  Dependencies

📁 Data

▶   Repositories

▷   DigitalEmotionDiaryDbContext.cs

▶ DTO

▶   Migrations

Models

▶ 🔒 📁 Configuration

▶ Comment.cs

▶ DiaryEntry.cs

▶  C# Emotion.cs

▷ EmotionType.cs

▶ EntryTag.cs

▶ C# Image.cs

▷   Like.cs

▶   Tag.cs

▷ User.cs

Resources

▶ 🔒 📁 Images

🔒 📄 DigitalEmotionDiary3.db

🔒 📄 DigitalEmotionDiary3.db-shm

🔒 📄 DigitalEmotionDiary3.db-wal

Services

▷ 🔒 📁 UI

```

└─ .gitattributes

```

 .gitignore

▶  C# Program.cs

▲ DigitalEmotionDiary.Tests

2.6. Data storage

As we learned in the course, we adapted the “Code-First” approach with **Entity Framework (EF) Core** to generate databases. Among them, we downloaded the following packages: **Microsoft.EntityFrameworkCore.Tools**, **Microsoft.EntityFrameworkCore.Sqlite**. We mapped each entity(DiaryEntry, User, Emotion, Like, Comment, Tag, EntryTag, Image) with `DbSet<T>` to the Database table.

To populate initial seed data (static data), we used the `hasData()` method in **DbContext** and migrated (Add-Migration, Update-database).

For `EntryTag` entity, we set one to many (1:N) relationship with `DiaryEntry` and `Tag` (`DiaryEntry ↔ EntryTag ↔ Tag`)

For `Emotion` entity, we set one to one (1:1) relationship with `EntityType`.

While the application is running, (dynamic data) can be generated more by the user's CRUD(`Get()`, `Add()`, `Update()`, `Delete()`) operations, or updated in run time by other dynamic events.

For database, we choose to use SQLite, and DBeaver for DB administration tool.












2.6.1 Seed data

Seed data is captured from Dbeaver screen as following:



2.6.1.1 DiaryEntity

	<small>123</small> Id	<small>44</small> Title	<small>44</small> Content	<small>44</small> CreatedAt	<small>123</small> IsPublic	<small>123</small> UserId	<small>123</small> EmotionId	<small>123</small> ImageId
1	1	Good news	Han Kang, South korean writer won the Nobel Prize in Lite	2024-12-18 01:11:02.8994779	1	1	1	1
2	2	A complete shock	An idiot declared martial law today, luckily paliament over	2024-12-18 01:11:02.8995984	1	1	4	2
3	3	Maya	I adapted new cat, she's so adorable!	2024-12-18 01:11:02.8995991	1	2	2	3
4	4	Kimchi day	Time to make kimchi for next year, I'm ready for the war!!	2024-12-18 01:11:02.8995996	1	1	1	[NULL]
5	5	Trip to Jeju	It's our family's yearly trip. Getting exeited!	2024-12-18 01:11:02.8996	1	1	1	[NULL]
6	6	Sickness	It's awful to get sick while I'm traveling.	2024-12-18 01:11:02.8996004	0	2	7	[NULL]
7	7	Party	We live only once in our life. Let's party today! yay!	2024-12-18 01:11:02.8996008	1	2	2	[NULL]
8	8	Shopping	So excited, I'm in the middle of shopping heaven!	2024-12-18 01:11:02.8996012	1	1	2	[NULL]
9	9	Chirstmas	So excited, I'm in the middle of shopping heaven!	2024-12-18 01:11:02.8996016	1	1	2	[NULL]
10	10	Oyster	So excited, I'm in the middle of shopping heaven!	2024-12-18 01:11:02.899602	1	1	2	[NULL]
11	11	Riga, my home	Food in Riga is best in my opinion!	2024-12-18 01:11:02.8996024	1	2	2	[NULL]
12	12	Healthy life	work out everyday is so important. I don't want to get sic	2024-12-18 01:11:02.8996028	0	2	8	[NULL]










2.6.1.2 Emotion

	 Id	 EmotionTypeId	 BackgroundColor
1	1	1 	Yellow
2	2	2 	Orange
3	3	3 	Beige
4	4	4 	Brown
5	5	5 	Red
6	6	6 	Grey
7	7	7 	Black
8	8	8 	White









2.6.1.3 Emotion Type

	 Id	 Name
1	1	Happy
2	2	Energized
3	3	Tired
4	4	Anxious
5	5	Stressed
6	6	Sad
7	7	Annoyed
8	8	Neutral






2.6.1.4 EntryTag

	 Id	 DiaryEntryId	 TagId
1	1	1 	1 
2	2	2 	2 
3	3	3 	1 


2.6.1.5 Image

	 Id	 Path	 UploadedAt	 Description	 DiaryEntryId
1	1	./Resources/Images/hankang.webp	2024-12-18 01:11:02.9155772	writer Han Kang	1 
2	2	./Resources/Images/120324.png	2024-12-18 01:11:02.915673	the night under martial law	2 
3	3	./Resources/Images/maja.jpeg	2024-12-18 01:11:02.9156736	maja	3 








2.6.1.6 Like

	 123 Id	123 DiaryEntryId	123 UserId
1	1	1 	2 
2	2	2 	2 


2.6.1.7 Tag

	 123 Id	A-Z Name
1	1	news
2	2	shocking

2.6.1.8 Comment

	 123 Id	A-Z Content	A-Z CreatedAt	123 DiaryEntryId	123 UserId
1	1	Congraturation! Woohoo!	2024-12-18 01:11:02.9074769	1 	2 
2	2	We need to fight for democracy!	2024-12-18 01:11:02.9075471	2 	2 
3	3	awwww so happy for you!	2024-12-18 01:11:02.9075475	2 	1 

2.6.1.9 User

	 123 Id	A-Z UserName	A-Z Email	A-Z Password	A-Z ProfileImagePath
1	1	Elin	elin@gmail.com	pass1	elinProfile.png
2	2	Ivan	ivan@gmail.com	pass2	ivanProfile.png

2.7 Design patterns

We have actually covered design patterns we used in our app in section 2.3 (Code review & analysis). I think it is better to do it that way and show concrete examples which patterns were used in code snippets. In summary, some design patterns we used:

- Singleton pattern. For example DiaryEntryService gets instantiated just once and the same object is reused throughout the whole application.
- Dependency Injection. We can use same example, DiaryEntryService here. It gets injected in places it is required.
- MVC pattern in our app. We used Model-View-Controller design pattern.
 1. Model layer contains data and business logic. For example Models, DTOs and all services.
 2. View. Takes care about user interface and representation of data from model layer. Command, UserInterface are classes from that layer.
 3. Controller manages interaction between view and model. I can mention LoginService as example here.

2.8 Event-based code

For this project, we prioritized implementing event-based code rather than multithreading to handle specific scenarios effectively. Below, we provide examples of two key scenarios that utilize event-based programming:

Scenario 1 : Handling User Input from the menu

When a user inputs a command from the CLI menu, the following sequence is triggered:

1. **Input Event:** The command entered by the user triggers the `ParseInputAsCommand` event.
2. **Validation:** The input is validated to ensure it matches the expected format and available commands.
3. **Command Processing:** Once validated, the corresponding command is processed.
4. **Data Handling:** The processed data is sent to the Service layer for execution.
5. **Feedback:** The result is returned and displayed to the user in the CLI.

This flow ensures the app responds to user commands while maintaining separation of concerns through clear layers.

```
private bool GetInputFromUser()
{
    var Command? UserCommand = Command.BlankCommand();
    while (true)
    {
        Console.Write("COMMAND: ");
        var string? userInput = Console.ReadLine();
        UserCommand = ParseInputAsCommand(userInput);
        if (!UserCommand.IsBlankCommand() && CommandIsValid(UserCommand) && CommandHasCorrectNrOfArguments(UserCommand))
        {
            ExecuteCommand(UserCommand);
        }
        if (UserCommand.IsQuitCommand() || UserCommand.IsLogoutCommand())
        {
            break;
        }
    }
    if (UserCommand.IsQuitCommand())
    {
        return true;
    }
    return false;
}
```

```
private Command ParseInputAsCommand(String userInput)
{
    if (userInput.Length < 1)
    {
        return Command.BlankCommand();
    }
    var string[]? commandWithArguments = userInput.Split(";");
    String command = commandWithArguments[0];
    String[] arguments = [];
    if (commandWithArguments.Length > 1) {
        arguments = new String[commandWithArguments.Length - 1];
        Array.Copy(commandWithArguments, 1, arguments, 0, arguments.Length);
    }
    return new Command(command, arguments);
}
```

```

private void ExecuteCommand(Command command)
{
    switch (command.GetName())
    {
        case WRITE_COMMAND :
            WriteDiaryEntry(command.GetArguments());
            break;
        case GET_COMMAND :
            GetDiaryEntry(command.GetArguments());
            break;
        case GET_ALL_COMMAND :
            GetAllDiaryEntries();
            break;
        case DELETE_COMMAND :
            DeleteDiaryEntry(command.GetArguments());
            break;
        case HELP_COMMAND :
            DisplayMainMenu();
            break;
        case GET_BY_COLOR_COMMAND :
            GetEntryByColor(command.GetArguments());
            break;
        case UPDATE_COMMAND :
            UpdateDiaryEntry(command.GetArguments());
            break;
        case LOGOUT_COMMAND :
            Logout();
            break;
        case QUIT_COMMAND :
            PrintQuitText();
            break;
        default:
            Console.WriteLine("Error, unknown command: " + command.GetName());
            break;
    }
}

```

Scenario 2: Capture unexpected errors

To handle exceptions and maintain robust error management, we implemented event-based triggers for specific error scenarios:

- **Missing required information:**
If a user omits critical fields (e.g., title, content, isPublic or emotionId) when publishing a diary entry, it triggers ArgumentException.

```

public void PublishDiaryEntry(long userId, DiaryEntryDTO dto)
{
    if (dto == null)
    {
        throw new ArgumentException("DiaryEntryDTO cannot be null.", nameof(dto));
    }

    var DiaryEntry? entry = new DiaryEntry
    {
        UserId = userId,
        Title = dto.Title,
        Content = dto.Content,
        CreatedAt = DateTime.Now,
        IsPublic = dto.IsPublic,
        EmotionId = dto.EmotionId,
        ImageId = dto.ImageId
    };

    _diaryEntryRepository.CreateDiaryEntry(entry);
    _diaryEntryRepository.SaveChanges();
}

```

- **Image upload errors:**

If a user fails to provide a file path for an image or attempts to upload a file that does not exist, it triggers either an `ArgumentException` or a `FileNotFoundException`.

```

0 references
public void UploadEntryImage(long userId, long entryId, string filePath, string? description = null)
{
    if (string.IsNullOrWhiteSpace(filePath))
        throw new ArgumentException("File path cannot be null or empty.", nameof(filePath));

    if (!System.IO.File.Exists(filePath))
        throw new FileNotFoundException("The specified file does not exist.", filePath);

    var Image? image = new Image
    {
        Path = filePath,
        UploadedAt = DateTime.Now,
        Description = description,
        DiaryEntryId = entryId,
    };

    _imageRepository.UploadEntryImage(image);
    _imageRepository.SaveChanges();
}

```

These events ensure that errors are caught and handled appropriately, provide meaningful feedback to users and prevent the application from crashing with unknown reason.

2.9 Testing

For testing we used NUnit testing framework. Here is a testing code sample from User class:

```
onDiaryNUnitTests EmotionDiaryNUnitTests.UserTests ShouldHaveEmptyProfileMa
10 [Test]
11 | 0 references
12 public void ShouldInitializeUser()
13 {
14     // Arrange + Act
15     var User? user = new User
16     {
17         UserName = "Test User",
18         Email = "user@test.com",
19         Password = "password1"
20     };
21     // Assert
22     Assert.AreEqual(0, user.Id);
23     Assert.AreEqual("Test User", user.UserName);
24     Assert.AreEqual("user@test.com", user.Email);
25     Assert.AreEqual("password1", user.Password);
26     Assert.IsNotNull(user.DiaryEntries);
27     Assert.IsNotNull(user.Comments);
28     Assert.IsNotNull(user.Likes);
29 }
30 [Test]
31 | 0 references
32 public void ShouldAddDiaryEntryToUser()
33 {
34     // Arrange
35     var User? user = new User
36     {
37         UserName = "Test User",
38         Email = "user@test.com",
39         Password = "password1"
40     };
41     var DiaryEntry? diaryEntry = new DiaryEntry
42     {
43         Title = "Test Entry",
44         Content = "First diary entry."
45     };
46     // Act
47     user.DiaryEntries.Add(diaryEntry);
48     // Assert
49     Assert.AreEqual(1, user.DiaryEntries.Count);
50     Assert.AreEqual("Test Entry", user.DiaryEntries.First().Title);
51 }
```

First test focuses on User object creation. In assert part we check that Collections gets initialized, so we don't have a null Pointer exception. We also check that fields do get assigned values, though it is actually obvious.

Second test asserts that diary entry gets added to user DiaryEntry list. For that we create both user and diary entry first, and then add diaryEntry to user. In assert part we check that user has that diaryEntry.

```

52
53 [Test]
54 | 0 references
55 public void ShouldAllowUserToLikeDiaryEntry()
56 {
57     // Arrange
58     var User? user = new User
59     {
60         UserName = "Test User",
61         Email = "user@test.com",
62         Password = "password1"
63     };
64     var DiaryEntry? diaryEntry = new DiaryEntry
65     {
66         Title = "Test Entry",
67         Content = "Entry with like"
68     };
69     var Like? like = new Like
70     {
71         User = user,
72         DiaryEntry = diaryEntry
73     };
74     // Act
75     user.Likes.Add(like);
76     // Assert
77     Assert.AreEqual(1, user.Likes.Count);
78     Assert.AreEqual("Entry with like", user.Likes.First().DiaryEntry.Content);
79 }

```

Third test have focus on adding a like. For that we need a user, diaryEntry, and like. Rest of test if similar to previous.


```

79
80 [Test]
81 | 0 references
82 public void ShouldAllowUserToDeleteDiaryEntry()
83 {
84     // Arrange
85     var User? user = new User
86     {
87         UserName = "Test User",
88         Email = "user@test.com",
89         Password = "password1"
90     };
91     var DiaryEntry? diaryEntry = new DiaryEntry
92     {
93         Title = "Test Entry",
94         Content = "Entry with like"
95     };
96     // Act
97     user.DiaryEntries.Add(diaryEntry);
98     // Assert
99     Assert.AreEqual(1, user.DiaryEntries.Count);
100
101     user.DiaryEntries.Remove(diaryEntry);
102     Assert.IsEmpty(user.DiaryEntries);
103 }
104
105 [Test]
106 | 0 references
107 public void ShouldHaveEmptyProfileImagePath()
108 {
109     // Arrange + Act
110     var User? user = new User
111     {
112         UserName = "Test User",
113         Email = "user@test.com",
114         Password = "password1"
115     };
116     // Assert
117     Assert.IsNull(user.ProfileImagePath);
118 }

```

And couple more tests. One will check that diaryEntry can be both added and removed from user diaryEntry list, and last one checks that profile image gets initial value, null.

We covered also DiaryEntry with several nUnit tests. Given short time, I tried to make integration test for one of the Repositories, but to be honest it was a bit time consuming and we could not risk to have some part of application which does not work before delivering assignment.

2.10 Refactoring

There are room for many improvements to the code in its current state.

- There is much code, especially in the service and repository classes that are effectively dead, because it's not being used as of now. Some examples are for instance "TagService.cs" and "TagRepository.cs". Although we did implement some functionality for basic CRUD on DiaryEntry Tags, we did not get around to employ them in any useful way and tie it together with other DiaryEntry-related functions (like sorting by Tags). This code should be removed, because it contributes no useful functionality, in order to keep the code in the repository minimal and easier to read and understand. We include the classes to give an idea of how we would go ahead if we actually were to implement this kind of functionality. That's why it's there, even though it gives little sense from a "clean code" perspective.
- The UserInterface.cs class / implementation is somewhat a bit heavy to read because a lot of (although related) functionality for parsing, validating and executing Commands from the user is clumped together in the same class. A better approach may be to split this functionality into separate modules, and tie them together in the UserInterface class. That would likely make the UI code easier to read and reason about.
- We have decided to implement the application in a way where we have the client (frontend-front) and services (backend-part) running in the same application. We did this due to time constraints, but would very much like to split the application in separate client and server (and possibly a middle api) applications. This would make it possible to run the UI / client part on a separate machine that doesn't demand the user to directly interact with the machine storing the database. This requires a major architectural overhaul though, so we have chosen the easy way in order to implement a working version.

3. User Interface

3.1 Design

The UI is implemented as a CLI-like interface in the `UserInterface.cs` file. It handles parsing, validation and execution of the available commands. It is bootstrapped in the `Program.cs` application entrypoint. The file is rather large, so we decided to isolate / abstract a “Command” as an own class, which has some utility functions like object-comparison, checking type of Command and getter and setter function on the fields.

3.2 How to use

On starting the application. The user is first prompted for a username and password. We have predefined some data through migrations to make sure there are two users available in the database on startup. These are:

User 1

Username: Elin

Password: pass1

User 2

Username: Ivan

Password: pass2

We choose to include these users in the migration, to avoid having to register new users on every startup. The “userId” for these users are 1 and 2 respectively.

After logging in with either of the username/password combinations, the user is presented with the following menu:

```
MENU
-----
WRITE      - Write entry to Diary
DELETE     - Delete entry from Diary
GET        - Get a specific entry from the Diary
GET_ALL    - Get all entries from the Diary
GET_BY_COLOR - Get entry by background color
UPDATE     - Update diary entry(title,content or emotion)
HELP       - Display this menu
LOGOUT     - Logout current user
QUIT      - Exit program
-----
```

And a prompt to enter a command which looks like this:

COMMAND: |

Everyone of the “Commands” listed above requires a specific number of arguments. It’s important to note that the arguments are separated with *semicolon (;)* and not whitespace, as is more common in traditional CLI’s. Following is a overview of the Commands and the arguments they require as well as explanations of them:

- **WRITE;<Title>;<Content>;<IsPublic>;<EmotionId>**
 - * <Title>: Title of the diary entry. Datatype is String
 - * <Content>: The main text content of a diary entry. Datatype is String
 - * <IsPublic>: Indicates whether the entry should be private or open to the public. The Datatype is boolean (in form of a String) - “true” or “false”
 - * <EmotionId>: The id of the emotion associated with the entry, valid number are In the range between 1-8. The Datatype is an integer.

The WRITE Command creates a new diary entry and stores it in the database. The entry is associated with the userId, and can only be retrieved by the user with that userId if it <IsPublic> is set to “false”.

- **DELETE;<EntryId>**
 - * <EntryId> The specific entry to delete.

Deletes the entry with <EntryId> as entryId. A user can only delete his/her own diary entries

- **GET;<EntryId>**
 - * <EntryId> entryId of diary entry to retrieve

Retrieves the specific diary entry with the <EntryId> as id. Can only be called on its own (created by the logged in user) diary entries.

- **GET_ALL**

This command has no arguments. It retrieves all the users' diary entries. As well as diary entries of other users that are marked as public (IsPublic set to “true”)

- **GET_BY_COLOR;<Color>**
 - * <Color>:Color of entries to fetch

Fetches all entries, created by the logged in user, which has the specified <Color> as background color. <Color> should be typed with text. The <Color> parameter is the (case-sensitive) name of the color (e.g. “Yellow” without parenthesis). Valid <Color> values are: Yellow, Orange, Beige, Brown, Red, Grey, Black, White.

- **UPDATE;<EntryId>;<Field>;<NewValue>**

- * <EntryId>: Id of the entry to be updated.

- * <Field>: Which field should be updated.

- * <NewValue>: The new value the field should be updated with.

Updates the given field of a DiaryEntry (<Field>) with a new value given by <NewValue>. The legal values of <Field> can be either of "TITLE", "CONTENT" or "EMOTION".

- **HELP**

Takes no arguments. Prints out the menu shown above.

- **LOGOUT**

Takes no arguments, Will log out the current user and return to the login prompt that appears when the application is initially started.

- **QUIT**

Takes no arguments. Exits and terminates the application.

3.3 Examples

The following paragraphs contains examples of some Commands and how they are executed:

COMMAND: WRITE;Diary title;Here comes the content;false;2

```
COMMAND: WRITE;Finally;It's about to deliver exam! 🤩;true;2
```

The command above creates a new private (only visible to the creator) Diary Entry with the title "Diary title", content equal to "Here comes the content" and an Emotion set to 2 (State: Energized / Background color: Orange).

COMMAND: GET;1

This will get the DiaryEntry with id equal to 1 and print out its content (given that it has been created by the user. Otherwise nothing will be printed). And example of its output is:

```
COMMAND: GET;1
ENTRY[2024-12-18 오전 1:11:02]
Title: Good news
Content: Han Kang, South korean writer won the Nobel Prize in Literature! I'm so proud of her
```

COMMAND: GET_ALL

This command will print out all the user's diary entries, as well as all entries marked by other authors as public. An example output follow below:

```
COMMAND: GET_ALL
ENTRY[2024-12-18 오전 1:11:02]
Title: Good news
Content: Han Kang, South korean writer won the Nobel Prize in Literature! I'm so proud of her

ENTRY[2024-12-18 오전 1:11:02]
Title: A complete shock
Content: An idiot declared martial law today, luckily paliament overruled it in two hours, could save our democracy at the end. What a drama!

ENTRY[2024-12-18 오전 1:11:02]
Title: Kimchi day
Content: Time to make kimchi for next year, I'm ready for the war!!

ENTRY[2024-12-18 오전 1:11:02]
Title: Trip to Jeju
Content: It's our family's yearly trip. Getting excited!

ENTRY[2024-12-18 오전 1:11:02]
Title: Shopping
Content: So excited, I'm in the middle of shopping heaven!

ENTRY[2024-12-18 오전 1:11:02]
Title: Chirstmas
Content: So excited, I'm in the middle of shopping heaven!

ENTRY[2024-12-18 오전 1:11:02]
Title: Oyster
Content: So excited, I'm in the middle of shopping heaven!

ENTRY[2024-12-18 오후 3:30:38]
Title: Finally
Content: It's about to deliver exam! 🥰

ENTRY[2024-12-18 오전 1:11:02]
Title: Maya
Content: I adapted new cat, she's so adorable!

ENTRY[2024-12-18 오전 1:11:02]
Title: Party
Content: We live only once in our life. Let's party today! yay!

ENTRY[2024-12-18 오전 1:11:02]
Title: Riga, my home
Content: Food in Riga is best in my opinion!
```

COMMAND: GET_BY_COLOR;<Color>

This command will print out entries with background color.

```
COMMAND: GET_BY_COLOR;Brown
ENTRY[2024-12-18 오전 1:11:02]
Title: A complete shock
Content: An idiot declared martial law today, luckily paliament overruled it in two hours, could save our democracy at the end. What a drama!
```

UPDATE;<EntryId>;<Field>;<NewValue>

UPDATE command demands three fields of data. It should be typed as below:

```
COMMAND: UPDATE;1;TITLE;Great News;
```

HELP

This command will print out menu as below:

```
COMMAND: HELP
MENU
-----
WRITE          - Write entry to Diary
DELETE         - Delete entry from Diary
GET            - Get a specific entry from the Diary
GET_ALL        - Get all entries from the Diary
GET_BY_COLOR   - Get entry by background color
UPDATE         - Update diary entry(title,content or emotion)
HELP           - Display this menu
LOGOUT         - Logout current user
QUIT           - Exit program
-----
```

LOGOUT

This command will log out the current user and go back to the login prompt as initial status.

```
COMMAND: LOGOUT
Please log in. Enter your username:
```

QUIT

This command terminates the program. It look as below:

```
COMMAND: QUIT
Exiting program...
```

4. Limitation

API

- For further improvements, we are planning to make controllers and API with endpoints.

Class: EmotionType

- It could be enum, but we changed it to type string for scalability in the future.

Features : Like, Comment, Tag, Image

- We aimed to implement CLI, but couldn't manage it due to the time limit. Also it was quite limited to show it in console based frontend.

UI

- UI/Login service has both UI logic and user authentication, we could improve that.

Database

- We experienced multiple errors while database migration, therefore we often choose to delete the migrated db, and migrate again. Everytime we did it, the field Date.now in DiaryEntry was updated. It became difficult to track when the original entry was created.

CLI

- We could make a new flag for the Admin role that is access for both private and public entries. In our setting with users, it's not available for private entries.
- When we try to delete diaryEntry, we need to know about entryId. But it's unable to search for an entryId for a specific entry by using GET_ALL currently.

Security

- We could add password hash for adding security, but we decided not to take this option this time.

5. Source reference

- Example Codes from the lectures
- StackOverflow
- https://www.youtube.com/watch?v=uvqAGchg8bc&ab_channel=MatthiWare
A nice tutorial about unit tests.
- <https://www.w3schools.com/cs/index.php>