1. Inledning

Bakgrund och syfte:

- Detta projekt är en del av en skoluppgift där målet var att utveckla en kontaktapplikation i C#. Applikationen syftar till att ge praktisk erfarenhet av systemutveckling och att tillämpa designmönster som SOLID och principerna för objektorienterad programmering. Projektet inkluderar både konsolbaserade och grafiska applikationer för att förstå olika utvecklingsmetoder och användarupplevelser.

Kort sammanfattning av versionerna och arkitekturen:

- Projektet utvecklades med följande struktur och versioner:
    1. **Business Class Library:** En central komponent som innehåller **Core Files** (exempelvis datamodeller och gränssnitt) samt **Logic** (affärslogik och funktionalitet). Detta gör det möjligt att återanvända logiken mellan olika versioner av applikationen.
    2. **Testprojekt utan mocking:** Ett testprojekt för att verifiera applikationens funktionalitet med enhetstester som inte kräver mockade komponenter.
    3. **Testprojekt med mocking:** Ett separat testprojekt som använder mocking-ramverk för att testa applikationens beroenden och säkerställa robustheten hos enskilda moduler.
    4. **Console-version:** Den första versionen av applikationen som fokuserade på att hantera kontakter och spara data i en JSON-fil.
    5. **WPF-version:** Ett grafiskt användargränssnitt lades till för att ge en mer användarvänlig upplevelse och visuellt interagera med kontaktlistan.
    6. **WPF med MVVM:** Den slutliga versionen implementerade MVVM-arkitektur för att separera logik, data och gränssnitt, vilket resulterade i en modulär och skalbar applikation.

2. Business struktur

- Applikationen är uppdelad i en **Business Class Library** som innehåller två huvudsakliga komponenter: **CoreFiles** och **Logic**. Denna struktur följer principerna för **Dependency Injection** och **modulär arkitektur**, vilket möjliggör tydlig separation av ansvar och återanvändbarhet av kod.

CoreFiles

- **CoreFiles** innehåller grundläggande element som definierar applikationens kärna. Filstrukturen är dokumenterad i **CoreFileStructure.txt** (se Bilagor), och innehåller följande viktiga delar:

1. **Crud:** Interface för CRUD-operationer (`ICreate`, `IRead`, `IUpdate`, `IDelete`), vilka används för att definiera standardiserade metoder.
2. **Databases:** Hanterar JSON-baserad lagring genom `JsonFileDb` och exempeldata i `JsonDb.json`.
3. **Factory:** Implementerar Factory-mönstret för att skapa användare dynamiskt med klasser som `UserCreate` och `ExampleUserCreate`.
4. **Helpers:** Innehåller hjälpfunktioner som `PasswordHash` för lösenordshantering och `GuidGenerator` för att generera unika ID:n.
5. **Models:** Definierar applikationens datamodeller, t.ex. `Contact`, `FavoriteContact`, samt användarroller som `AdminRole` och `DefaultRole`.

Logic

- **Logic** innehåller applikationens affärslogik och ansvarar för att hantera dataflödet mellan CoreFiles och de olika gränssnitten (Console och WPF). Filstrukturen är dokumenterad i **LogicFileStructure.txt** (se Bilagor) och är uppdelad i följande kategorier:
    1. **Services:**

    - `ContactService`: Ansvarar för kontaktrelaterade operationer och implementerar `IContactService`.
    - `UserService`: Hanterar användarrelaterad logik via `IUserService`.
2. **Repositories:**
    - `ContactRepository` och `UserRepository`: Implementerar interface (`IContactRepository`, `IUserRepository`) för datalagring och datahämtning.
    - `JsonRepository`: Tillhandahåller en generisk implementation för att interagera med JSON-filer via `IJsonRepository`.

Dependency Injection

- **Dependency Injection (DI)** används för att koppla samman logik och datalager, vilket förbättrar testbarhet och kodunderhåll. DI-flödet är illustrerat i **DependencyInjection.png** (se Bilagor).

Designmönster

- Följande designmönster och principer har använts i projektet:
    1. **SOLID-principerna:**
       Dessa principer har tillämpats för att säkerställa att koden är strukturerad, lätt att underhålla och skalbar:
        a. **Single Responsibility Principle (SRP):** Varje klass och metod har ett enda ansvar, t.ex. separata repository-klasser för datalagring och service-klasser **för affärslogik.**
        b. **Open/Closed Principle (OCP):** Klasser är öppna för utökning men stängda för modifiering, vilket uppnås genom användning av interfaces och arv.
        c. **Liskov Substitution Principle (LSP):** Subklasser kan ersätta basklasser utan att bryta funktionaliteten, t.ex.

användningen av arv i rollhantering (`AdminRole`, `DefaultRole`).

   d. **Interface Segregation Principle (ISP):** Gränssnitt är specifika och små, t.ex. separata interface för CRUD-operationer (`ICreate`, `IRead`, etc.).
   e. **Dependency Inversion Principle (DIP):** Moduler förlitar sig på abstraktioner istället för konkreta implementationer, vilket möjliggörs med Dependency Injection.

2. **Repository Pattern:**
   Separerar datalagringslogik från affärslogik och gör det enkelt att byta ut datakällan vid behov.
3. **Service Pattern:**
   Hanterar affärslogik och fungerar som en mellanhand mellan repositories och applikationsgränssnitt.
4. **Factory Pattern:**
   Används för att skapa användarobjekt dynamiskt, vilket gör att skapandeprocessen är flexibel och lätt att utöka.
5. **Dependency Injection:**
   Används för att minska beroenden mellan komponenter och för att underlätta testning och utbytbarhet.

Exempeldata

- Ett exempel på den slutliga JSON-strukturen som används för att lagra kontakter finns i **FinalJsonExample.txt** (se Bilagor).

Klassdiagram

- Ett komplett klassdiagram för Business Class Library finns dokumenterat i **Business Class Diagram** (se Bilagor).

---

3. Tests och Mocking

Översikt av testning

- I projektet har två olika typer av tester använts för att tillämpa och demonstrera kunskaper inom testning:

1. **Enhetstester (Unit Tests):**
   Tester som fokuserar på att validera funktionaliteten hos individuella metoder och komponenter.
2. **Mocking:**
   Tester med mockade repositories och services för att hantera beroenden och simulera interaktioner mellan olika delar av applikationen.

Verktyg och ramverk

- **xUnit:** Ett testramverk som användes för att skriva och köra alla tester.
- **Moq:** Ett mocking-bibliotek som användes för att skapa simulerade implementationer av gränssnitt och klasser.

Varför mocking användes

- Mocking användes som en del av utbildningsmomentet för att:
  1. Simulera beteenden hos beroenden som repositories.
  2. Lära sig hur man verifierar interaktioner mellan tjänster och deras beroenden

---

4. Console Version

Översikt av konsolversionen

- Konsolversionen av applikationen implementerar en enkel textbaserad meny för att hantera användare och kontakter. Syftet med konsolversionen är att ge en grundläggande interaktion med affärslogiken i projektet utan grafiska komponenter. Den följer en modulär struktur med separata klasser för att hantera olika funktioner.

Funktionalitet och hur användaren interagerar med applikationen

- Användaren interagerar med applikationen genom en menystruktur med flera val:
  1. **Användarhantering:**
     - Skapa, uppdatera, läsa och radera användare.
     - Logga in som en specifik användare.
  2. **Kontakthantering:**
     - Skapa, uppdatera, läsa och radera kontakter kopplade till en inloggad användare.
  3. **Exempelanvändare (Example User):**

    - En fördefinierad användare med exempeldata visas vid applikationens start.

- **Filstrukturen** för konsolapplikationen inkluderar:
    1. **ContactMenu.cs:** Hanterar kontakter för den inloggade användaren.
    2. **UserMenu.cs:** Tillhandahåller menyer för användarhantering.
    3. **ExampleUserFrontpage.cs:** Introducerar och hanterar exempelanvändaren.
    4. **Program.cs:** Startpunkten för applikationen som sätter upp Dependency Injection och startar applikationen.

Hur affärslogiken används i konsolversionen

- Konsolversionen använder affärslogiken via följande steg:
    1. **Dependency Injection:**
       Tjänster och repositories som `UserService` och `ContactService` registreras och tillhandahålls via DI-container (Microsoft.Extensions.DependencyInjection).
    2. **Affärslogik:**
       Metoder i tjänster som `CreateUser`, `UpdateUser`, `CreateContact`, och `ReadAllContacts` anropas direkt från menyerna för att hantera data. Exempelvis skapas nya användare och kontakter via `UserService` och `ContactService`.
    3. **Datahantering:**
       Data lagras och hämtas från en JSON-fil via repositories som `JsonRepository`.

Begränsningar jämfört med WPF-versionerna

- Konsolversionen har följande begränsningar:
    1. **Ingen grafisk användarupplevelse:**
       Konsolversionen är helt textbaserad och erbjuder en enkel, linjär användarinteraktion.
    2. **Manuell navigering:**
       Användaren måste navigera genom menyer genom att skriva in numeriska val. Detta kan vara mindre intuitivt jämfört med knappar och visuella element i WPF-versionen.
    3. **Begränsad validering:**
       Felhantering och validering av användarinmatning är grundläggande och ger inte alltid detaljerad feedback.

4. **Ingen realtidsuppdatering:**
   Data uppdateras endast efter användarens aktiva val och visas inte dynamiskt, vilket är möjligt i WPF-versioner med datakontext och bindningar.

---

5. WPF Version

Översikt av den första WPF-versionen

- Den första WPF-versionen av applikationen är en grafisk implementation som bygger vidare på funktionaliteten i konsolversionen. Den introducerar ett GUI (Graphical User Interface) för att förbättra användarupplevelsen och göra det enklare att navigera mellan olika funktioner. Trots dessa förbättringar valde projektet att övergå till en MVVM-baserad version (WPF MVVM) eftersom den erbjuder bättre struktur och skalbarhet.

Grundläggande funktioner och hur de skiljer sig från konsolversionen

1. **Grafiskt användargränssnitt (GUI):**
   - Användare kan nu navigera mellan sidor som `LoginPage`, `RegisterPage`, `ContactsPage` och `FavoritesPage` istället för att använda en textbaserad meny.
   - Användaren interagerar med knappar, formulär och listor i stället för kommandoradsinmatningar.
2. **Hantering av kontakter och användare:**
   - Funktionaliteten för att skapa, uppdatera och ta bort kontakter och användare är identisk med konsolversionen men presenteras visuellt.
3. **Skillnader jämfört med konsolversionen:**
   - Navigering mellan funktioner sker via sidor (`Pages`) istället för menyer.
   - Data presenteras i listor i stället för att visas direkt i konsolen.
   - Felhantering och validering visas direkt i UI:t via meddelanden, textrutor eller färgändringar.

UI-design och layout

- Filstrukturen för WPF-versionen inkluderar:
  1. **MainWindows:**

    a. `HomePage.xaml`: Den första sidan som startas, där användaren kan välja att logga in eller registrera sig via knappar.
    b. `LoginPage.xaml` och `RegisterPage.xaml`: Hanterar autentisering och skapande av nya användare.
2. **Pages:**
    a. `ContactsPage.xaml`: Visar en lista över användarens kontakter.
    b. `FavoritesPage.xaml`: Visar användarens favoritkontakter.
    c. `AddContactPage.xaml` och `EditContactPage.xaml`: Tillåter skapande och redigering av kontakter.
    d. `AddFavoritePage.xaml` och `EditFavoritePage.xaml`: Hanterar favoritkontakter.
3. **Helpers:**
    a. `FavoriteToColorConverter.cs`: Konverterar data till UI-värden (t.ex. färger för att indikera favoriter).

UI-designen använder WPF:s styrkor, såsom:

1. **DataTemplates och Styles:** För att förbättra UI-layouten och skapa enhetliga designelement.
2. **Grid-layout:** För att placera komponenter såsom listor, knappar och formulär.

Begränsningar och eventuella problem i denna version

1. **Ingen MVVM-struktur:**
   Denna version använder code-behind för att hantera logik, vilket gör koden svårare att underhålla och återanvända. Detta var en av de huvudsakliga anledningarna till övergången till MVVM.
2. **Grundläggande design:**
   Även om UI:t är grafiskt är det inte lika flexibelt eller dynamiskt som i MVVM-versionen. Det saknas avancerade funktioner som datakoppling (Data Binding) och commands.
3. **Begränsad skalbarhet:**
   Att lägga till nya funktioner eller uppdatera befintliga sidor kräver ofta större ändringar, eftersom logik och UI är tätt sammankopplade.

---

6. WPF Mvvm Version

Beskrivning av hur MVVM implementerats

- I den uppdaterade WPF MVVM-versionen används MVVM (Model-View-ViewModel)-arkitekturen för att separera affärslogik från användargränssnittet. Detta möjliggör bättre struktur, testbarhet och underhåll av koden.
    1. **Struktur**
        a. **View (Vy):** Representeras av XAML-filer som `LoginPage.xaml` och `RegisterPage.xaml`. Dessa definierar användargränssnittet.
        b. **ViewModel:** Hanterar logik och användarinteraktioner via bindningar och kommandon. Exempel: `LoginPageViewModel`, `RegisterPageViewModel`.
        c. **Model:** Representerar applikationens data och affärslogik, tillhandahållen via Business-projektet, inklusive `UserService` och `ContactService`.
    2. **CommunityToolkit.Mvvm**
        a. Verktyget används för att förenkla implementationen av MVVM, t.ex. via `ObservableObject` för notifiering och `RelayCommand` för kommandon.

Databindning och Commands

1. **Databindning**
    a. Egenskaper som `Email` och `Password` är databundna till UI-komponenter som `TextBox` och `PasswordBox`.
    b. Exempel: Inmatad text i `TextBox` uppdateras direkt i `LoginPageViewModel` och valideras därefter.
2. **Commands**
    a. Kommandon som `LoginCommand` och `CreateCommand` kopplas till knappar, vilket minskar beroendet av code-behind och centraliserar logiken i ViewModel.
    b. Exempel: Vid klick på "Login" körs metoden `OnLoginClicked` i `LoginPageViewModel`.

Fördelar med MVVM jämfört med tidigare versioner

1. Separation av ansvar: Logiken är frikopplad från gränssnittet, vilket gör koden mer modulär och lättare att testa.

2. Återanvändbarhet: ViewModels kan användas mellan olika Views med minimal kodförändring.
3. Skalbarhet: Det är enklare att lägga till nya funktioner genom att skapa nya Views och ViewModels.

Hur affärslogiken integreras med MVVM

1. **Tjänster:** Affärslogiken hanteras av tjänster som `UserService` och `ContactService`, vilka exponerar kärnfunktioner till ViewModels.
2. **Integrationsexempel:**
    a. `LoginPageViewModel` använder `UserService` för att validera användarinloggning och navigerar användaren till `ContactWindow` vid framgångsrik inloggning.
    b. `RegisterPageViewModel` använder `UserService` för att skapa en ny användare och navigerar till `HomePage` efter registrering.

Eventuella lärdomar från denna version

1. Implementationen av MVVM har resulterat i en mer strukturerad applikation med förbättrad testbarhet och skalbarhet.
2. Användningen av `CommunityToolkit.Mvvm` har förenklat utvecklingen genom att tillhandahålla färdiga verktyg för notifieringar och kommandon.
3. Separat ViewModel för varje sida gör det enkelt att underhålla och modifiera enskilda funktioner utan att påverka andra delar av applikationen.

UI-design och förbättringar

- UI-layouten har vidareutvecklats med bilder och stilar för att skapa en mer professionell och användarvänlig upplevelse.
    - Bilder, t.ex. `frontpage.png`, används på startsidan för att skapa en visuell inbjudan.
    - Stilar definieras i resurser som `ButtonStyles.xaml` och `TextInputStyles.xaml` för att säkerställa enhetlighet i designen.
- Navigering hanteras effektivt via `NavigationService`.
    - Views inkluderar `HomePage`, `LoginPage`, `RegisterPage` och `ContactWindow`.
    - Dessa länkar till undersidor som `ContactsPage`, `FavoritesPage` och `MinaSidorPage`.

---

7. Framtida Utveckling

Funktioner som kan läggas till eller förbättras:

1. Implementera en funktion för att automatiskt lägga till favoriter från den vanliga kontaktlistan, vilket skulle förbättra användarupplevelsen.
2. Förbättra designen och användargränssnittet genom att göra det responsivt, så att applikationen anpassar sig till olika skärmstorlekar och enheter istället för att vara låst vid en viss höjd och bredd.

Planer för att optimera koden eller prestandan:

1. Separera applikationen ytterligare för att efterlikna en frontend/backend-struktur. Detta kan göras genom att flytta databasoperationer och relaterade repositories till en server, som hanteras via API:er. Användarens klientapplikation skulle då vara en lättviktig UI-applikation som endast innehåller nödvändiga tjänster som `UserService` och `ContactService`. Backend-servern skulle hanteras via en separat administratörsapplikation som uppdaterar och administrerar databasen.

Förslag på hur projektet kan skalas upp eller anpassas till andra plattformar:

1. **Använda .NET MAUI:**
   För mindre företag eller team kan .NET MAUI vara en effektiv lösning för att centralisera frontend-utvecklingen och stödja flera plattformar, såsom Windows, macOS, Android och iOS, med en gemensam kodbas. Detta skulle underlätta för mindre appar eller program att nå fler användare med minimala resurser.

2. **Backend som plattformsoberoende lösning:**
   Backend-servern kan förbli oberoende av specifika plattformar och hantera alla databasoperationer via API:er. Detta möjliggör en tydlig separation mellan frontend och backend, vilket gör det enklare att skala applikationen eller integrera fler klienter.

3. **Struktur för större företag:**

   För större företag kan projektet organiseras med MAUI för att skapa en gemensam kodbas som fungerar för flera plattformar samtidigt. Utvecklarna fokuserar på att utveckla funktionaliteten enhetligt med hjälp av MAUI, utan att optimera individuella plattformar.

   För att hantera plattformsspecifika problem kan en ansvarig person utses för varje plattform (t.ex. Windows, Android, iOS, macOS). Denna person är inte en utvecklare för själva applikationen, utan ansvarar för att identifiera och lösa problem som är specifika för sin plattform, såsom

buggar, prestandafrågor eller kompatibilitetsproblem relaterade till MAUI:s implementation.

---

8. ChatGPT - 4o & AI Användning

Användning av AI i projektet:

AI, specifikt ChatGPT-4o, har spelat en betydande roll under utvecklingen av projektet. Verktyget har använts på följande sätt:

1. **Kodgenerering och optimering:**
   AI har använts för att generera kodsnuttar för specifika funktioner, såsom databindning, kommandon och valideringslogik. Detta har påskyndat utvecklingsprocessen och gett förslag på förbättrade lösningar.
2. **Dokumentation:**
   ChatGPT har hjälpt till att formulera och strukturera dokumentationen för projektet. Detta inkluderar beskrivningar av funktioner, arkitektur och framtida utvecklingsplaner.
3. **Debugging och felsökning:**
   Vid komplexa fel och buggar har AI föreslagit potentiella lösningar och förbättringar baserat på logik och kodexempel som tillhandahållits.
4. **UI/UX-förbättringar:**
   AI har rekommenderat designprinciper och layoutlösningar för att förbättra användarupplevelsen, särskilt vid implementeringen av MVVM-arkitekturen.

AI som stöd och begränsningar:

1. **Stöd:**
   AI har fungerat som en virtuell handledare genom hela projektet, vilket har minskat behovet av externa resurser och dokumentationssökningar. Verktyget har också varit till hjälp för att snabbt förstå och tillämpa koncept som Dependency Injection och Design Patterns.
2. **Begränsningar:**
   AI-genererad kod kräver granskning och anpassning för att säkerställa att den passar projektets specifika krav. Det är viktigt att verifiera att den genererade koden är kompatibel med den befintliga arkitekturen och uppfyller de krav som ställs av uppgiften.

Sammanfattning:

AI, inklusive ChatGPT-4o, har varit en ovärderlig resurs för detta projekt. Genom att integrera AI i utvecklingsprocessen har det blivit möjligt att arbeta mer effektivt, lösa komplexa problem och skapa en mer strukturerad och professionell applikation.

---

9. Referenser och Resurser

Dokumentation, artiklar och tutorials som använts:

1. **Microsoft Dokumentation:**
   Officiell dokumentation för .NET, WPF och relaterade teknologier.
   Microsoft Learn
2. **Udemy - Lärare Frank Liu:**
   Kurs: *Learn .Net MAUI, Events Driven, MVVM, Clean Architecture, Creating and Consuming ASP.NET Core minimal API*
   Kursinnehåll inkluderade MVVM, Dependency Injection, Clean Architecture och databaslösningar.
   Kursbetyg: 4.6 av 5 med 363 omdömen och över 2 493 studenter.
3. **ChatGPT-4o:**
   AI-stöd användes för kodgenerering, problemlösning och dokumentationsstrukturering.

Verktyg och ramverk som användes i projektet:

1. **.NET och WPF:**
   Ramverk för utveckling av både konsol- och GUI-applikationer.
2. **CommunityToolkit.Mvvm:**
   Ett bibliotek som underlättar implementering av MVVM-arkitektur via klasser som `ObservableObject` och `RelayCommand`.
3. **xUnit:**
   Ett testramverk som användes för enhetstester utan behov av mock-objekt.
4. **Moq:**
   Ett mocking-ramverk för att simulera beroenden vid testning.

5. **Visual Studio:**
   Huvudverktyg för utveckling och debugging av applikationen.

---

10. Bilagor

Mappstruktur och relaterade filer:

1. **CoreFiles:**
   Dokumentation över mappstrukturen för applikationens kärnkomponenter, tillgänglig i filen `CoreFileStructure.txt`.
   Exempel: Datamodeller, gränssnitt och hjälpfunktioner som `GuidGenerator`.
2. **Logic:**
   Mappstrukturen för applikationens logik, dokumenterad i filen `LogicFileStructure.txt`.
   Exempel: Services, Repositories och deras implementationer.
3. **JSON-struktur:**
   Ett exempel på den JSON-struktur som används för att lagra data i applikationen, beskrivet i filen `FinalJsonExample.txt`.

Diagram och visuella hjälpmedel:

1. **Dependency Injection-flöde:**
   Diagrammet `DependencyInjection.png` visar hur olika tjänster och komponenter integreras i applikationen.
2. **Business Class Library - Klassdiagram:**
   Diagram som illustrerar relationerna mellan klasser i Business-projektet.
   `Business-ClassDiagram.drawio.png`
3. **InterfaceImplementeringar:** Hur interfaces och deras implementeringar kan kommunicera med varandra i ett större strukturerat nät.
   `InterfaceImplementeringar.png`

Testresultat och prestandamätningar:

1. **Testresultat och kodtäckning:**
   Testrapporter och kodtäckning visualiseras i filen `Testrapport.png`, som sammanfattar enhetstester och integrationstester.

Skärmdumpar av applikationen:

1. **WPF GUI - Viktiga sidor:**
    - `HomePage.png`
    - `LoginPage.png`
    - `RegisterPage.png`
    - `ContactsPage.png`
    - `MinaSidor.png`