Full stack WPF application

All the development from the database layer to the UI layer.

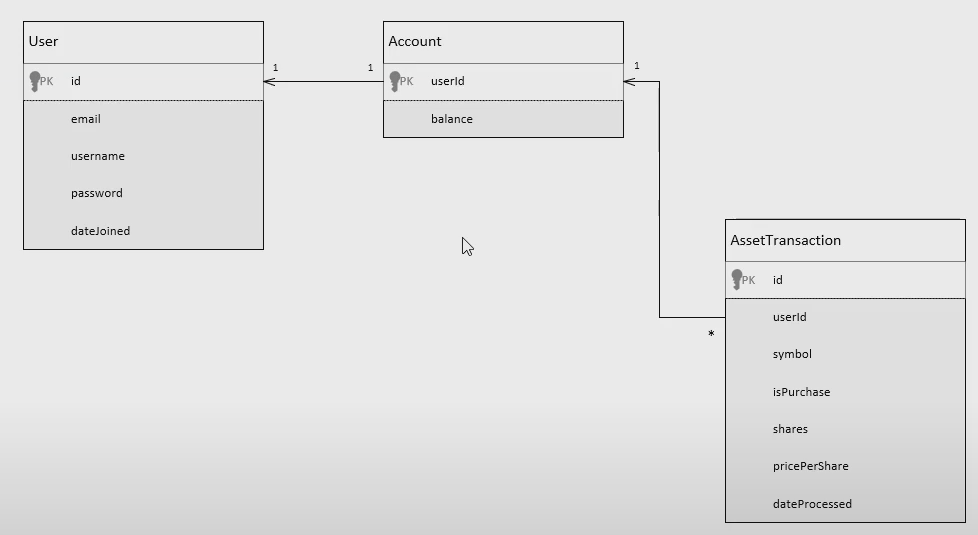
MVVM-architecture, dependency inversion + testing

**SimpleTrader** – a stock trading application

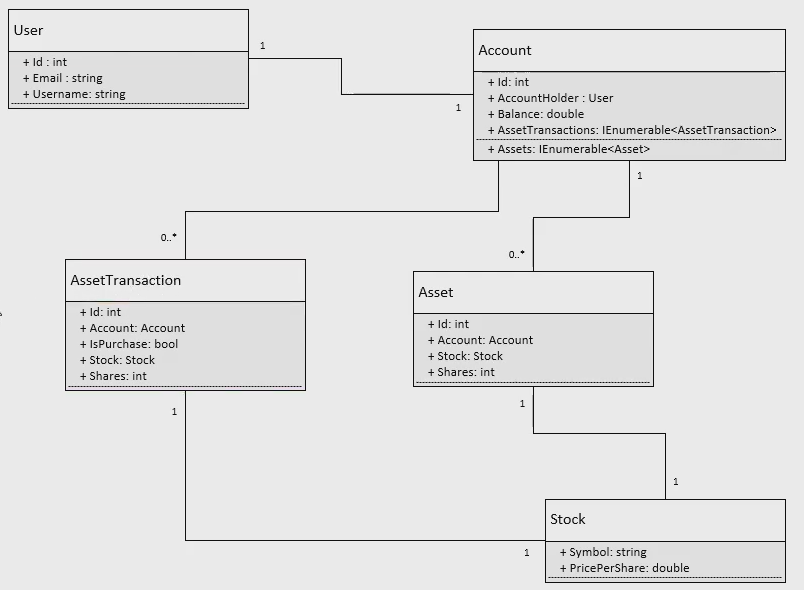
# Video #1

**Trading application beginning**

## Database



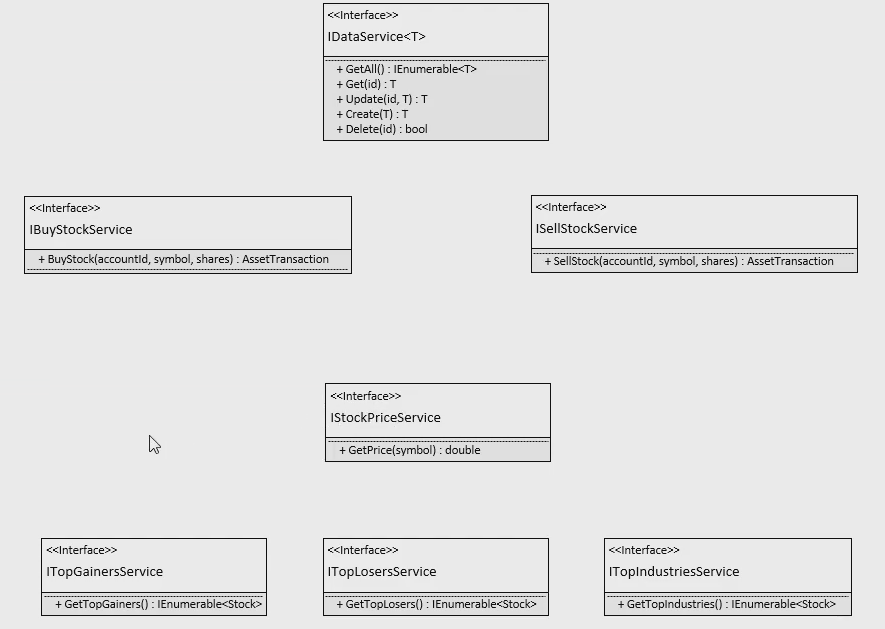
## Domain



These are the classes inside the “Models” folder inside the “SimpleTrader.Domain” project.

AssetTransaction: used to calculate total assets

## DomainServices



These are the interfaces inside the “Services” folder inside the “SimpleTrader.Domain” project.

## Create a Class Library project – SimpleTrader.Domain

Contains all domain logic inside a Models folder. At the beginning it only has the models:

* Account
* AssetTransaction
* Stock
* User

And an interface folder called “Services” which initially contains:

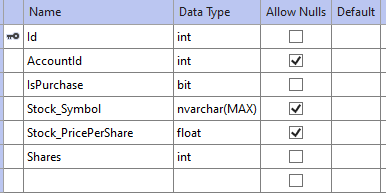
* IDataService – A service for accessing data

## Create a Class Library project – SimpleTrader.EntityFramework

Is dependent on the “Microsoft.EntityFrameworkCore”, “Microsoft.EntityFrameworkCore.SqlServer” and “Microsoft.EntityFrameworkCore.Tools” NuGet packages, all version 5.0.13, and the “SimpleTrader.Domain” project.

The class “SimpleTraderDbContext” is derived from “DbContext” and manages the interaction with the database with any framework. It needs to be told what kind of entities it’s going to store in the database. This will be done through DbSet. Each of the Database tables is given their own DbSet element.

In order to generate the database, migrations must be used. And for this a connection string must be provided. This is done in the “OnConfiguring” method inside the “SimpleTraderDbContext”.

The “OnModelCreating” inside “SimpleTraderDbContext” replaces the Stock element in “AssetTransaction” with the two elements from the Stock class. Think of the Stock in “AssetTransaction” as a foreign key to the Stock class.  
The table then looks like this:  


Using the Package Manager Console with SimpleTrader.EntityFramework as the Default project and uses the commands “add-migration initial” and “update-database”.  
The database is now created with the tables and elements defined in the Models in SimpleTrader.Domain.

The database is now set up.

## Video #2

**Talking to the database**

The database connection string inside the SimpleTraderDbContext class is hardcoded. Create a new class “SimpleTraderDbContextFactory” where it can be passed in the constructor.

Adding data services for the frameworks to the SimpleTrader.EntityFramework.

The “GenericDataService” is located inside this Services folder. This class implements the IDataService interface from the domain models and gives functionality to the tasks: Create, Delete, Get, GetAll and Update elements in the database.

It is now possible to add, delete, read and update the elements in the database, but error handling has not yet been implemented.

# Video #3

**Creating WPF project with MVVM**

## Creating necessary folders and files

Creating WPF project and removing the OnStartup from the App.xaml. Moving this functionality inside the App.xaml.cs --> OnStartup method.

Adding folders “View” and “ViewModel”.

Creating a “ViewModelBase” class and a “MainViewModel” class that extends this.

Creates NavigationBar.xaml (inside “Controls” folder) that will always be visible on top of the application window.

Creates Common.xaml inside “Styles” folder. “Style”-classes contains everything from colors to button styles etc. that is reused across the application.

## MVVM framework setup

Creates more ViewModels and corresponding Views.

Creates the “State” folder, with the “Navigators” folder inside. Contains one interface and class; INavigator and Navigator respectively. The Navigator holds the current view model that is displayed in the application and contains an ICommand command that will update the current view.  
An ICommand is used for the data binding in WPF, e.g., when a button is clicked, the binding runs the ICommand.

For this application, all commands are stored in the folder “Commands” and each unique ICommand will have their own class. This is done in order to make the code more decoupled and reusable.

The UpdateCurrentViewModelCommand (inside Commands) is created and its Execute method takes in a parameter. If the parameter exists in the INavigator ViewType, the corresponding view will become the current view.

Radiobuttons can now be used like this (after the xmlns nav path and command data context have been set):  


The Navigator class must also extend INotifyPropertyChanged with some methods to be able to update the view.

Creates a Models folder with the class ObservableObject, where the event PopertyChangedEventHandler and method OnPropertyChanged is moved to, and changes so that the ObservableObject extends INotifyPropertyChanged and the Navigator class extends ObservableObject.

Datatemplates for the views must also be defined inside the App.xaml.

# VIDEO #4

**API and Async ViewModel Loading**

This part is about connecting to an internet provided real-time index of stock market prices with an API.

Creates a new service (inside the SimpleTrader.Domain project) called IMajorIndexService and a corresponding MajorIndex class inside Models. The latter will store information about the price, changes in price and type (that is, the type of stock index).

Adds a new project for the stock market prices API called FinancialModelingPrepAPI.

The data is retrieved with a HttpResponseMessage in the MajorIndexService (inside FinancialModelingPrepAPI), and converted to JSON with the help of the NuGet package Newtonsoft.Json.  
The HttpResponseMessage requires an url as a string.

The WPF project class UpdateCurrentViewModelCommand gets data from the API project directly and is therefore dependent on this. This dependency remains forever (throughout all the remaining videos) and I’m unsure whether it is possible or not to get rid of this dependency (probably not).

# VIDEO #5

Stylization (not implemented)

# VIDEO #6

Creating a card control. That is, the elements on the Home View Model displaying info.

Data to the cards is loaded using binding via the MajorIndexViewModel.

Add isChecked logic to the radiobuttons at the top with the help of a converter class.

# VIDEO #7

Connecting another part of the API to retrieve stock information.

Combining some of the logic behind the MajorIndexService and StockPriceService.

API now requires a key. Had to create fake data that could be loaded.

Created a custom exception (inside domain).

# VIDEO #8

**Inserting user data – communicating (adding/removing) the database**

Lagde BuyStockService for å la brukere kjøpe. Handelen legges inn i dbo.AssetTransactions.

# VIDEO #9

**Dependency Injection Setup**

At this point all the services are created in the OnStartup method in App.xaml.cs. We want the dependency injection container to automatically create all the services instead.

Benefits with dependency injection:

* Give us services with dependencies already passed in
* Define all application dependencies in one isolated place (the dependency injection container)

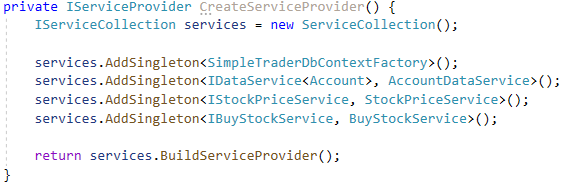
Using Microsoft.Extensions.DependencyInjection (built into .NET).

Three types of services in terms of lifetime:

* Singleton: one instance per application
* Transient: different instance everytime
* Scoped: one instance per “scope”

The accountService, stockPriceService and buyStockService no longer need to be “configured manually” like this:  


But rather like this:  


Using the method:  


We want to register as much as possible with our Dependency Injection container.

The great thing about the dependency injection container, is that let’s say the FinancialModelingPrepAPI disappears and we can’t use the StockPriceService anymore. By replacing the API we can implement a new StockPriceService and register it at the same place in the code.

In addition to the services, we’re also dependency injecting the ViewModels, Navigator and MainWindow.

The factories are created because we want different view models whenever we switch views (instead of always using the same ones). A factory will create an instance when we need it.

# VIDEO #10

Added a view for buying (but kept it empty). This view differs from the others because it is retained throughout the application runtime. Edited name of a factory (from abstract to root).

# VIDEO #11 - #15

**Not implemented**

# VIDEO #16

**Navigation**

Navigation to a new view from a view on the same level.

Removing the UpdateCurrentViewModelCommand from INavigator and Navigator. This makes sense because the Navigator is inside the State folder and should therefore only keep track of the state of the program, not perform commands. The navigator shall not have the responsibility to instantiate views.

Circular dependencies are a big warning sign of poor architecture.

Adding the IRenavigator, but never used since the Login page isn’t implemented.  
The renavigation allows for logic inside the currentview to change the current view (while the Navigator/INavigator only allows navigation to be executed from the navigation bar) (?).

# VIDEO #17

**Delegates**

Replacing the factories with delegates.  
A delegate describes a function.

The delegates allow us to handle viewmodel creation without having to create factory classes for the viewmodels. All of the factories are deleted.

Can now from the App.xaml.cs decide whether each viewmodel will be reused or a new one created every time a user opens a new view.

*Not in tutorial:  
Added a button in BuyViewModel that changes the view to HomeViewModel with the ViewModelDelegateRenavigator. In other words a button for switching views of the contentcontrol inside the contentcontrol itself.*