**Shahar Sadeh – 203903505, Maayan Hadar - 206015976**

Design Patterns Course – Exercise 2

# App Features Description (form previous exercise):

### Feature 1 – Match Maker:

Provides the user with the ability to find a friend from his friends list which made the highest number of likes to his posts.

The friends list can be filter by gender and desired age range, according to the user’s preferences.

The app will return the Matched soul mate with his full details (including profile picture, relationship status, location, status and more).

### Feature 2 – Events Finder:

Provides the user with the ability to find Events to which his friends are attending, according to desired list of filters (Event time, Friend’s gender, Friend’s age range, etc.).

The app will return a list of matching events.

# First Pattern – Facade:

#### Purpose:

We chose to implement a Façade pattern into our project due to the following reasons:

* Provide a single API to be used by the UI layer to access all data and logic operations more easily.
* Solve the problem of UI and logic components dependency by creating separation between the UI layer (view) and Logic layer (Controllers and Models) – improving app readability and maintainability.
* Hide from the UI layer inner logic components that it doesn’t need and shouldn’t be exposed to – thus holding to the Encapsulation and Aggregation concepts of Object Oriented Programming.
* Allow future expansion and replacement of UI and Logic components to the app more easily – improving scalability.

#### implementation description:

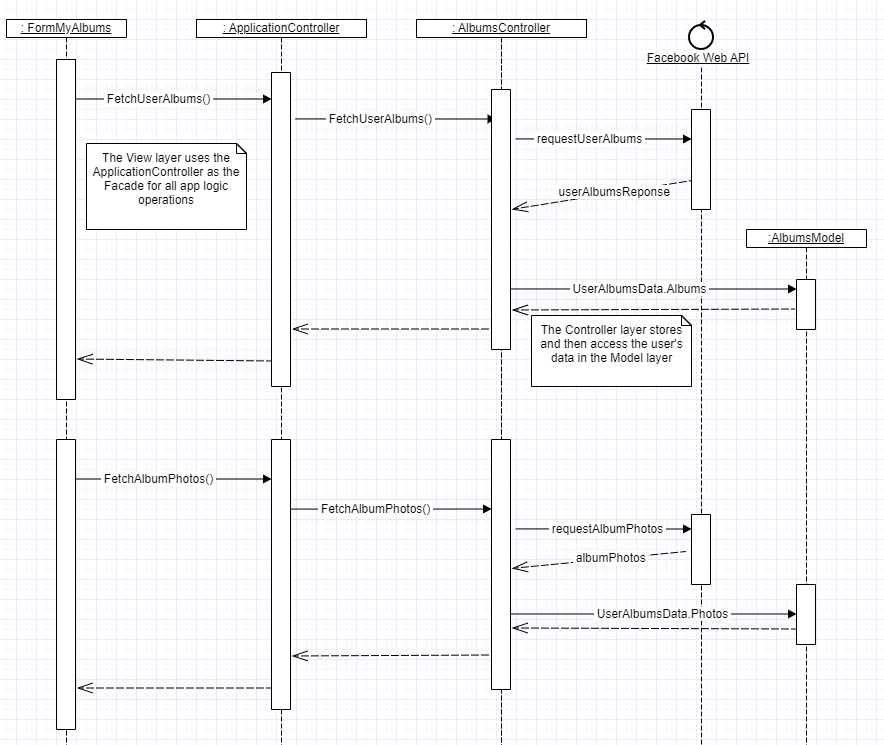
1. All logic operations of the app were encapsulated into a single façade class called **ApplicationController**.
2. The ApplicationController aggregates all the app logic controllers classes (**AlbumsController**, **FriendsController**, **PostsController**, etc.), each responsible for logic and data operations of a certain category (User’s albums, friends, posts respectively).
3. The controllers perform the communication with Facebook server, and also stores the user’s fetched data (albums, friends list, posts,.. ) into the Models classes (**AlbumsModel**, **FriendsModel**, **PostsModel** respectively).
4. The UI uses the ApplicationController in each of its forms (**AlbumsForm**, **FriendsForm**,. ..), thus creating a separation layer between the Model (data source) and the View (UI).

#### App components:

**FacebookLogic -> Controllers ->** ApplicationController, AlbumsController, EventByParametersController, FriendController, MatchMakerController, NewsFeedController, PostsController

**FacebookLogic -> Models ->** AlbumsModel, EventByParametersModel, FriendsModel, MatchMakerModel, NewsFeedModel, PostsModel, UserModel

## Sequence Diagram (façade described by FormMyAlbums (view) – AlbumsModel interaction):



## Class Diagram

# Second Pattern – Builder:

#### Purpose:

Our app includes a UI component called **EventCustomedItem** that represents an Event in **FormEventByParamers** feature, where the user can get a list of events filtered by parameters.

The data source of this item (where the event’s properties are stored in the app) is implemented by **CustomizedEventModel** .

The model is comprised with many properties that represents the event’s details (name, description, Location, Venue, Time, Etc.).

We chose to implement a Builder pattern so that we can construct different representation of **CustomizedEventModel** in the future, which will able us to expand and/or replace the Event’s properties if needed.

For example:

* Facebook can release new Event properties in the future (e.g. “event star rating”), so we will be able to support them.
* Facebook can modify types and formats of existing Event properties, such as date and time being displayed in DD/MM/YYYY or DD-MM-YY formats.

This will also allow us to replace the current Builder with a new and upgraded one in the future (that supports performance improvement like Lazy Loading for example).

#### Implementation:

1. We created an interface called **IBuilder** that the defines the Builder methods.
2. We created a **CustomizedEventBuilder** class the implements the **IBuilder** interface. This class represents the ConcreteBuilder in the pattern.
3. We created a Director class that serve’s as the pattern director, which receives the polymorphic base **IBuilder** and contains the **ContructCustomizedEvent**() method.

This method executes the set of methods defined in the interface, so it can construct the **CustomizedEventModel.**

1. Finally the client receives the **CustomizedEventModel** that is built upon demand. The object is returned via customizedEventBuilder.CustomizedEvent property that serves as the pattern’s GetResult().

#### App components:

FacebookLogic -> **CustomizedEventBuilder.cs, IBuilder.cs, Dorector.cs**

FacebookLogic -> Models **-> CustomizedEventModel.cs**

## Sequence Diagram:

## Class Diagram

# Third Pattern – Singleton:

#### Purpose:

We wanted to allow only a single user to be connected to the app at the same time.

And also to create a single login management tool that will handle all login and sign in operation with Facebook API, during the application lifetime.

Therefore, only single instance of LoginManager can be allowed to run in the app, so for this reason we chose to implement the Singleton pattern.

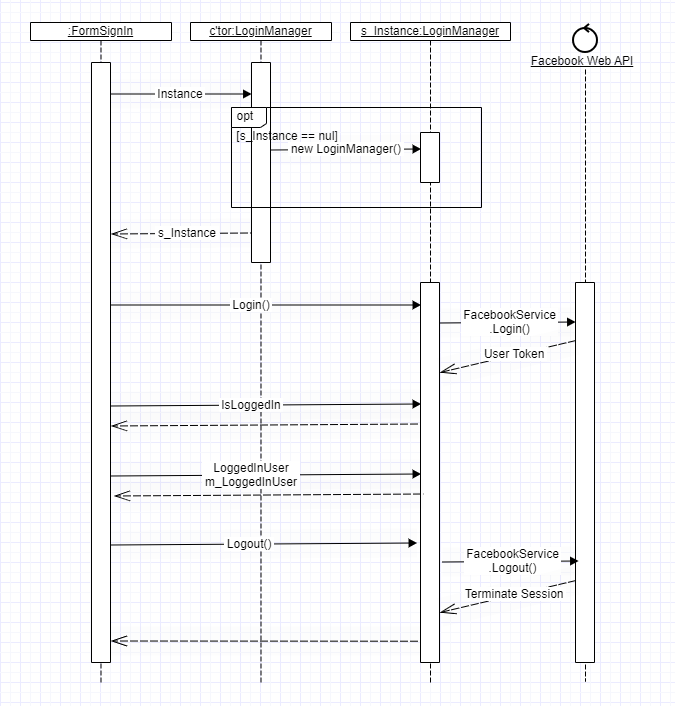
#### Implementation:

1. We added a s\_**Instance** field to the **LoginManager** class
2. We set the class c’tor access modifier to **private**
3. We added a **static get property** that returns the LoginManager
4. We also implemented **Thread-Safe** and **Double Check Lock** mechanisms.

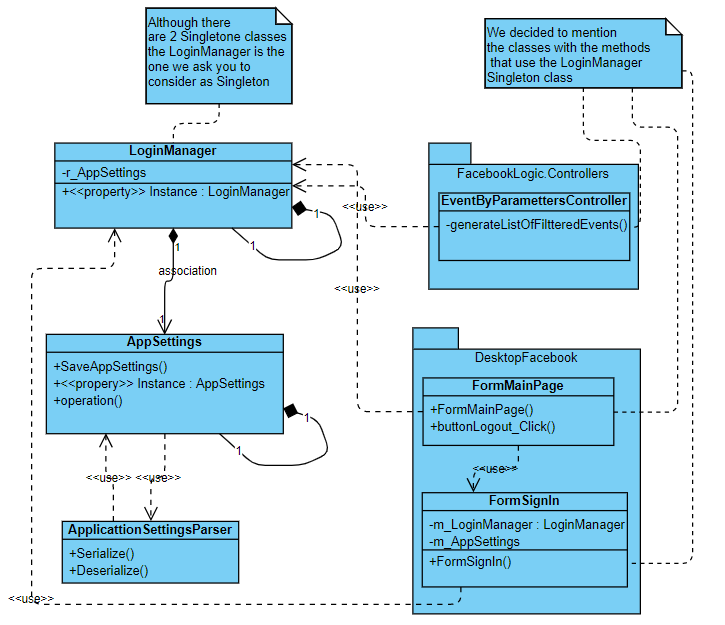
#### App components:

FacebookLogic -> **LoginManager.cs**

## Sequence Diagram:



## Class Diagram:



# Asynchronous Implementation:

We chose to implement asynchronous communication with Facebook API by creating different Threads that responsible for fetching the data from the server.

This in order to allow other app operations to be performed while the data is loaded from the server, and also to load the data asynchronously (so we don’t have to wait until all items are fetched).

The following methods are called in dedicated Threads:

FetchUserPosts() - FormPosts.cs line 24

FetchUserNewsFeed() – FormNewsFeed.cs line 19

FetchUserAlbums() – FormMyAlbums.cs line 18

# Data Binding:

Data Binding was implemented for FriendsController class:

bindFriendsourceBinding() – line 30