Analysis and Design Document

Student: Stancu Gabriel - Iulian

**Group: 30432**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 01/Apr/2021 | 1.0 | Project Specification and Elaboration - Iteration 1.1 | Stancu Gabriel - Iulian |
| 23/Apr/2021 | 2.0 | Added the Iteration 1.2 | Stancu Gabriel - Iulian |
| 18/May/2021 | 3.0 | Completed the Iteration 2 and added the testing documentation | Stancu Gabriel - Iulian |
|  |  |  |  |

Table of Contents

I. Project Specification 4

II. Elaboration – Iteration 1.1 4

1. Domain Model 4

2. Architectural Design 5

2.1 Conceptual Architecture 5

2.2 Package Design 5

2.3 Component and Deployment Diagrams 6

III. Elaboration – Iteration 1.2 7

1. Design Model 7

1.1 Dynamic Behavior 7

1.2 Class Design 8

2. Data Model 9

3. Unit Testing 9

IV. Elaboration – Iteration 2 10

1. Architectural Design Refinement 10

2. Design Model Refinement 12

V. Construction and Transition 12

1. System Testing 12

2. Future improvements 13

VI. Bibliography 13

# Project Specification

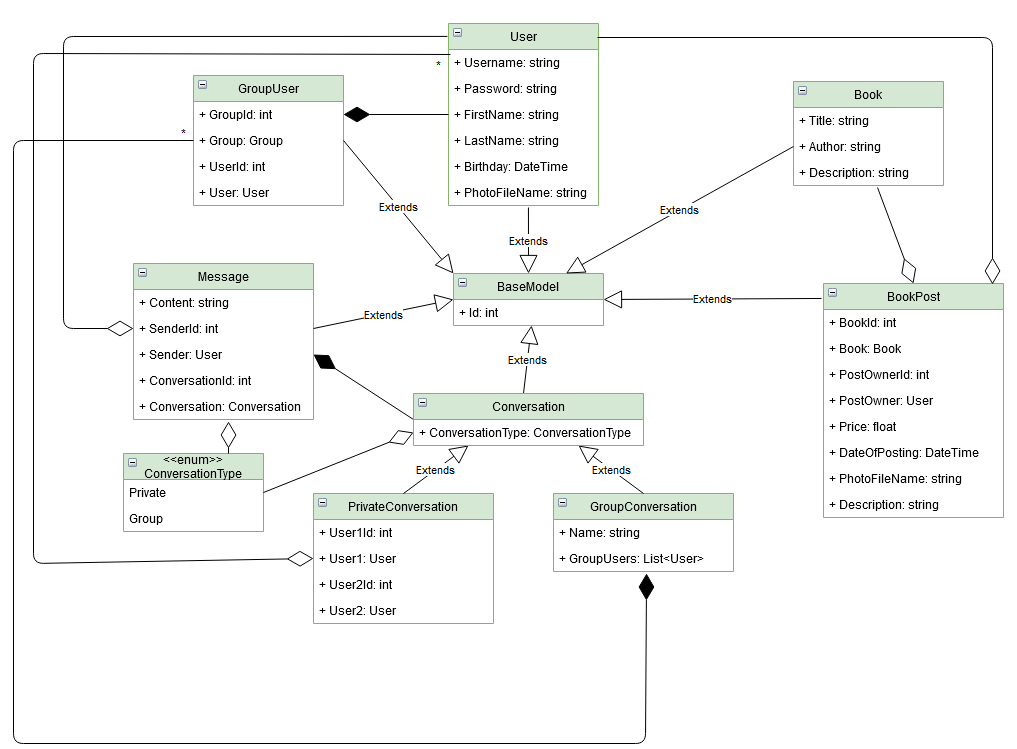
The final form of the project will be a hub for readers, where they can exchange books (trade, sell, buy, borrow etc.) but they can also exchange opinions and have discussions based on their readings. The project aims to build a whole community based on a common hobby: books reading.

For this matter, the final application will have to allow online, real-time communication between its users. Considering this, the first component that needs to be considered is the server of the application. The server should allow authentication, account details settings, data storage (user and books information). Because the application includes a chat between its users, the messages, conversations, groups etc. will require storage space as well. For the communication aspect, a hub needs to be established on the server, to serve this purpose.

The client application will be a phone application at first, eventually transitioning into a web application as well (not for this project, however). The application will target both Android and iOS devices and will allow its users the usual account functionality (login, signup, password recovery, personal settings etc.) together with searching, filtering, grouping the book articles of all the app users and finally the communication functionality, which will consist of chats creation, group chats, contacts managing.

# Elaboration – Iteration 1.1

# Domain Model

The main concepts around whom the domain model revolves will be the User and the Book entities. The User entity will represent the final user of the application, with associated personal information and account information. Similarly, the Book entity will contain data describing the book it refers to. Other entities for the project would be Conversation (Private & Group), Message, BookPost, GroupUser (handles many to many relaytionship between GroupConversation and User).

# Architectural Design

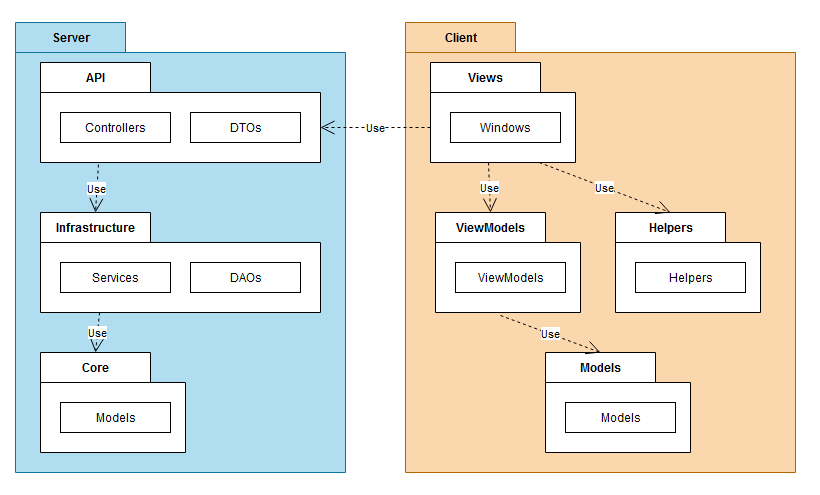
## Conceptual Architecture

The overall structure of the project will be built around the Client – Server pattern. The server will handle database communication and client request handling, while the client will display the request results and will provide input controls for the user to interact with the system.

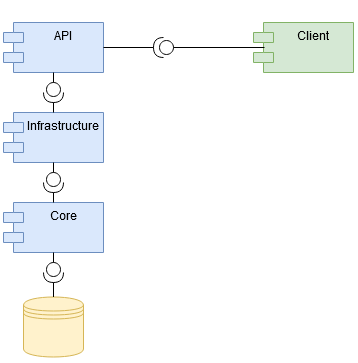
Because the server needs to communicate and access a database, the pattern used for developing the server will be Layered. Initially, we think of the following three layers: API layer (the Controllers that receive the http requests, the hit-points of the server, DTOs), the Core layer (consists of the database entities models) and the Infrastructure layer (that will contain database accessors, services and business logic – related classes).

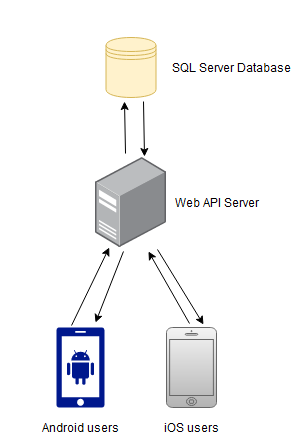
For the client application, which will be developed using Xamarin Forms, we will use the MVVM pattern, as this feels natural when working with Xamarin Forms projects, whose structure is similar with the WPF framework. By using this pattern, we make full use of the chosen framework.

## Package Design



## Component and Deployment Diagrams

Component Diagram:

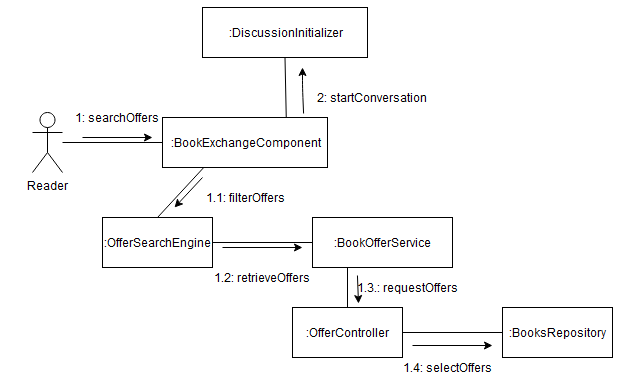
Deployment Diagram:

# Elaboration – Iteration 1.2

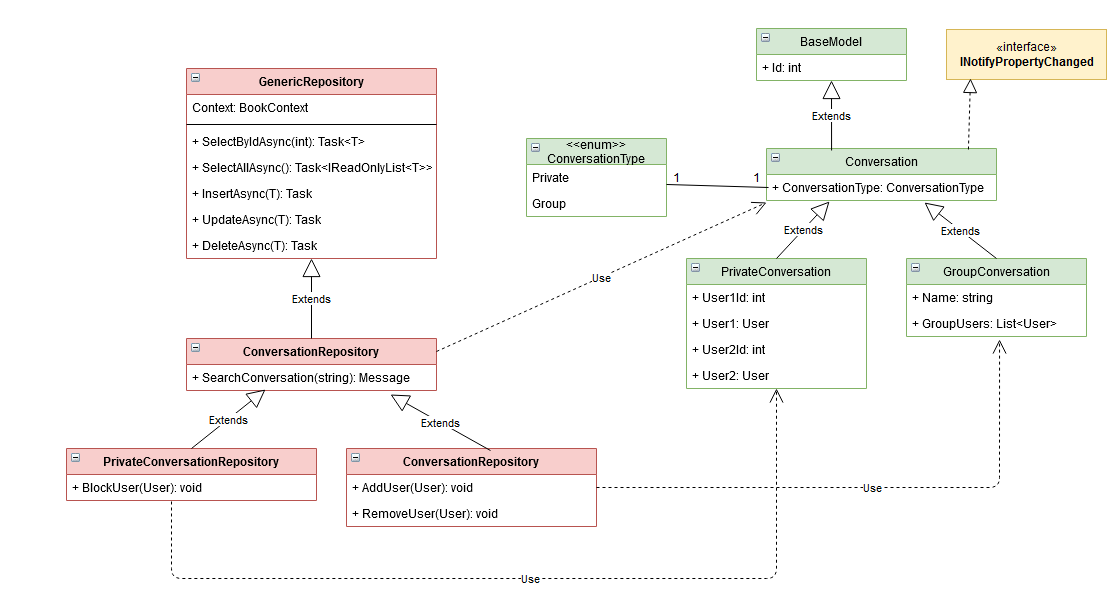
# Design Model

## Dynamic Behavior

## 



## Class Design

**

On the presented diagram we are displaying three of the design patterns we are going to use for our server API. They are colored differently for easier separation.

The red design pattern is the Repository design pattern. Because we are communicating with a database and the framework used for this purpose is Entity Framework Core, it felt naturally to pair the context functionality with the repository implementation. This way, we can create the needed queries in the server classes, by writing native (C#) code.

The green design pattern is the Player – Role design pattern. Because we have two types of conversations (group and private conversations) which in a future update might allow the switching between the types (a private conversation might become a group one if a third user is added to it, for instance), we are preparing this functionality from now, by implementing this design pattern on the conversation tree.

Finally, because we want to update the conversations each time a message is sent, we will implement the .NET version of the Observer Design Pattern, which is implementing the INotifyPropertyChanged interface.

# Data Model

# Unit Testing

From the unit testing point of view, we will test all the classes that provide logical functionality, covering as many cases as possible (combinations of inputs for all methods). This will apply both on the server and on the client, generating the required data where a certain component is missing (for instance we try to access the database from the client and the server is down, or we generate input from the client on the server). To avoid altering the production data, we will generate and test further on a test database.

For the moment, because of time constraints, we only provided a demo of testing for one of the server classes, which is the MessageRepository. We test both available methods (get all messages for a conversation and get last message from each conversation for a user) with valid data (the user exists, he has conversations with other users etc.), but future versions will include malicious cases, when received data might have been altered, thus producing unforeseen events. For both tests, we begin by creating the context required for communicating with the database, then we generate the required entities, we add them to the database, then we query against them and compare with the values we expect to get back, based on what we generated before.

The same principles will apply on other classes from other layers as well, both from the server and from the client application. Because most of the client logic is written in other classes than the pages themselves, all except a limited number of features will be testable too (except user alerts for instance).

# Elaboration – Iteration 2

# Architectural Design Refinement

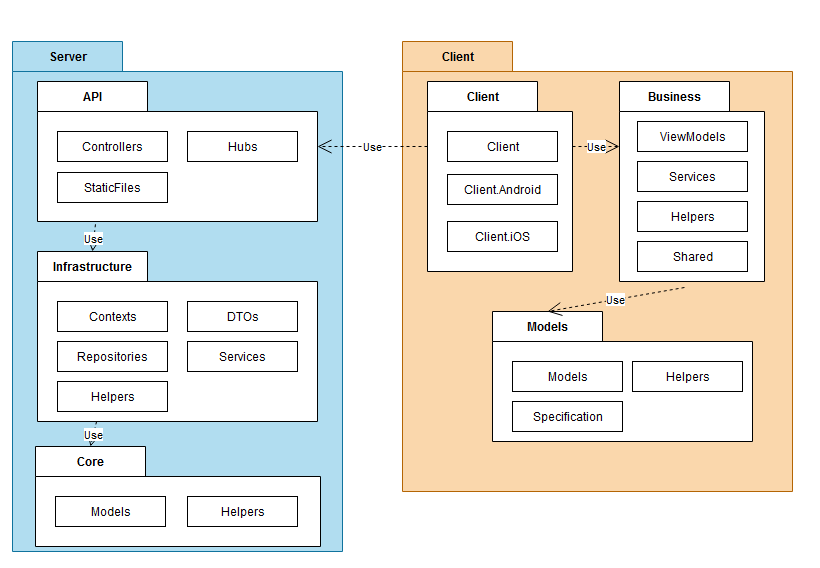
Regarding the conceptual architecture, we did not produce major changes. As it can be seen on the diagrams below (component and deployment diagrams), we did not change the overall layout of our project. However, at a more discrete level (package level and later classes level), we produced some changes.

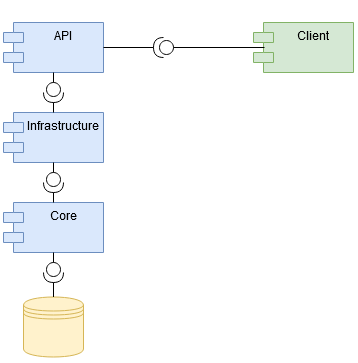
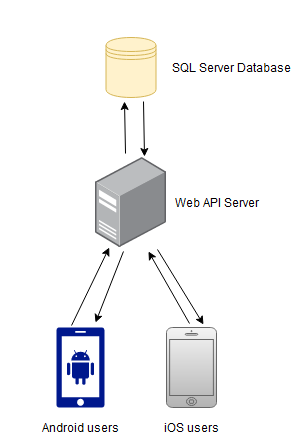
We divided both projects (the client and the server) into more packages and we eventually distributed them in the layers / components they belong to.

From the client’s point of view, the ViewModels package from the previous iterations is now included in a Business package/assembly, that also includes services (used for media content sharing, internet communication with the server etc.), a Helpers package (containing classes for different purposes, such as mapping between classes) and a Shared package, that acts as a storage for the running session of the app. In future versions this will be replaced with a state management policy. All these sub-packages were brought in the Business assembly with the ViewModels because of their connection (they are usually used together, so they were brought together). The Models assembly now contains packages for Helpers (model enhancements, for instance) and Specification (used to build the filtering criteria in the UI).

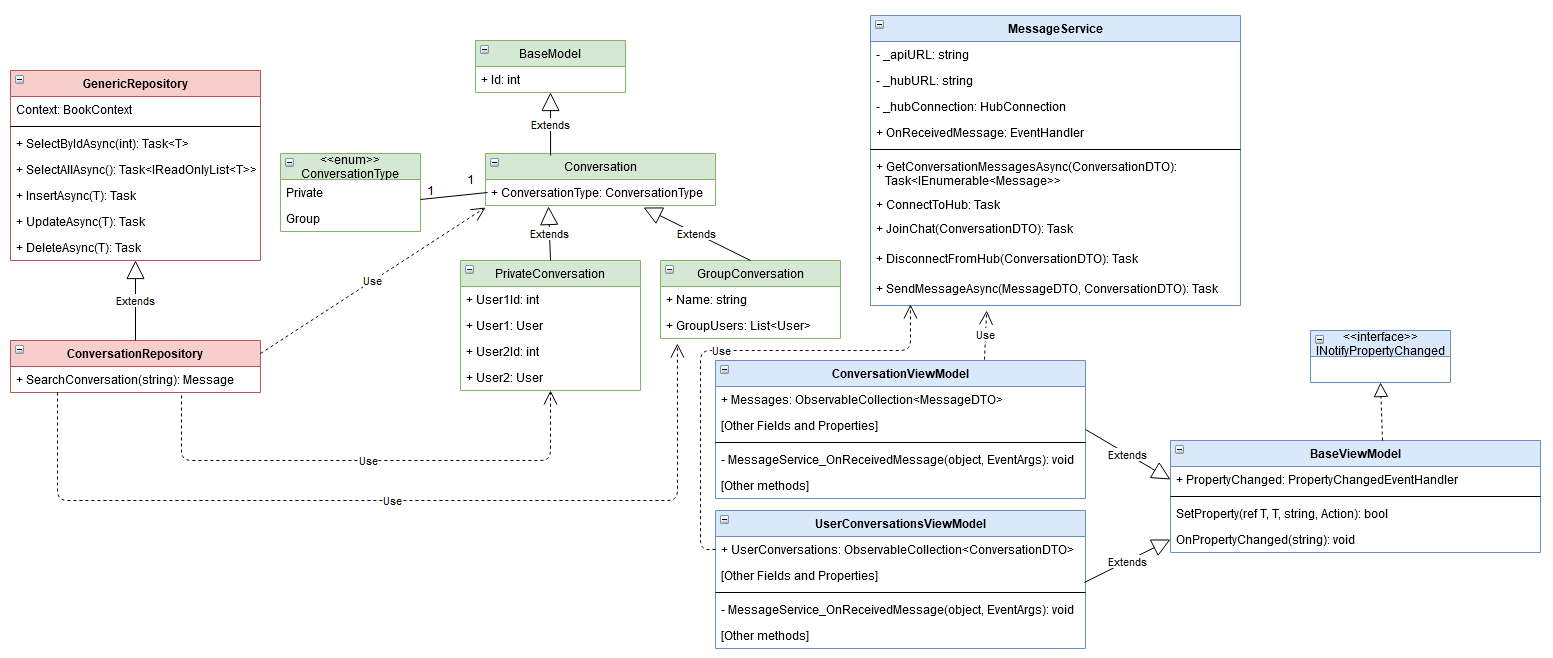
From the server’s point of view, the API layer also contains a package for the hubs (they are similar to controllers, but used for instant messaging communication between clients) and static files (photos of books from posts, users profile pictures etc.). At the Infrastructure layer, the DAOs were replaced by Repositories. The database contexts were also brought in this layer in order to be as close as possible to the repositories, which are the only classes that work with them. Finally, both the Infrastructure and the Core layers have a Helpers package, which enhance the functionalities of the classes in that layer.

As stated before, we produced no change with respect to the component and deployment diagrams.

******Package Diagram:

Component & Deployment Diagrams:

# Design Model Refinement

****

From the previous iteration, we have removed the PrivateConversationRepository and the GroupConversationRepository classes, thus reducing the number or dependencies, making the ConversationRepository acting more uniform on the two types of conversations. Also the INotifyPropertyChanged implementation was moved from this level (which resides on the server) to the ViewModel level on the client.

The Player-Role design pattern regarding pricate and group conversations remains unchanged.

On the client, the MessageService together with the ConversationViewModel and the UserConversationsViewModel implement a flavor of the Observer design pattern. Because the first mentioned class is a service, however, we did not want to add dependencies in it to view models, which are classes residing close to the presentation / UI layer of the application. We reversed the dependency direction by having an event in the service that is fired each time a new message is sent from the hub. The two view models subscribe to this event, providing a handler that performs a specific operation on each event firing: either update the list of messages in the current conversation, or update the last message in the overall user conversations page, to show the user an update took place. Both updates are done via binding, which is achieved by extending the BaseViewModel class, that implements the INotifyPropertyVhanged.

# Construction and Transition

# System Testing

From the integration testing point of view, we aim to ensure correctness for larger parts of our application. On the server, for instance, we could test each service individually. However, this should be covered by the unit testing, so the integration testing might cover even larger parts of the app, such as the controllers functionalities. This strategy was followed for demonstration purposes and we tested the login controller with valid and invalid data. The same logic will be applied on the other controllers.

From the client’s point of view, we might consider testing a whole functionality, from the moment it is fired in the ViewModel level (the first one under the Presentation level). Services on the client should be covered by unit testing as well.

For even larger scopes, we might consider testing both the client and the server together, by using an automated UI commands generation, for instance, or by starting from the ViewModels level as well.

# Future improvements

We are planning huge future updates for the app. One of the highest impact changes that will be performed is the enhancement of the security. Although no password is coming back to the client from the server, the passwords that go from the client to the server (when logging in, signing up) are not encrypted. They are also brutely stored in the database, with no security operation performed on them. The same applies to the messages between users. The first concern in future versions will be the security of the user data. Another huge update we are planning for the app is the including of the “rate a book” feature, which will allow the users to make posts visible to other users where they share their opinions, thoughts and they rate books they read.

Smaller updates are also coming along the way, such as displaying the connection status of users (connected, disconnected etc.), the status of a message (not sent, delivered, received, read), sharing media content within conversations, using emojis, gifs, stickers in the chat. We are also planning on bringing user contacts, such that users can have a list of “friends”, but also to improve the creation of a group and the start of a new conversation experiences.

Finally, we plan on testing as much of the app as possible, such that future releases of the application will be easier to deliver, backed by the “confidence” gained from automated testing.

# Bibliography

1. <https://mindofai.github.io/Building-a-Real-Time-Chat-App-With-SignalR-and-Xamarin/>
2. <https://ilclubdellesei.blog/2018/02/14/how-to-upload-images-to-an-asp-net-core-rest-service-with-xamarin-forms/>
3. <https://codeburst.io/getting-started-with-xamarin-development-as-of-june-2018-5552bb89bbbd>
4. <https://docs.microsoft.com/en-us/xamarin/xamarin-forms/data-cloud/web-services/rest>
5. <https://marcstan.net/blog/2020/01/connecting-to-localhost-from-a-xamarin-app/?fbclid=IwAR35sPzVXrKHQGH948Z5695gKidv9rijrQaX5ZCNt17frIRAozlUaIvmrVM>