Milestone 3
1 Test Strategies

1.1 Testing Techniques:
We will be primarily focusing on whitebox testing for the moment and will start to integrate more blackbox testing when we are closer to the end of the project

1.2 Unit Testing:
We will be making use of the BUnit testing framework that leverages xUnit, NUnit and MSTests. It allows us to individually test Blazor components/pages and the methods within them as well as the methods in the helper classes.

For each component, we will develop unit tests and strive for 80% coverage on our tests. To do so, we will highlight critical code paths prior to testing and ensure they are explored when performing testing.

1.3 Integration Testing:
As we complete more components relevant to our different use cases, we will begin to do integration testing using the built in .dotnet workflow framework provided on github. The guidelines we shall follow are listed below:

1. Our Integration testing will follow the same guidelines as out unit testing requiring 80% coverage and will require us to highlight crucial interactions between component that must be covered in testing

2. For components within the same designated use case, continuous integration tests will be built to test their compatibility with each of the components for said use case

3. Once the components within a use case are shown to function with minimal issues, we shall create continuous integration tests across different use cases

4. Once inter-component testing is complete, we will begin testing UI interactions

1.4 Future Testing:
After all the components have been completed we will begin more whitebox testing, focusing on ensuring our functional, non functional and user requirements are all met with all the necessary specifications.

2 Design Patterns

2.1 Singleton Pattern (Creational):

The first design pattern that we will implement in our project is singleton, as it will help facilitate the usage of user accounts in the discord clone. Information about the current user will need to be accessible in many places within the application, including but not limited to account verification and modification. Thus, creating manager-style singleton classes that allow for global access, modification and movement of account information will ensure cleaner code and safer data.

Resource: https://refactoring.guru/design-patterns/singleton

2.2 Strategy Pattern (Behavioral):

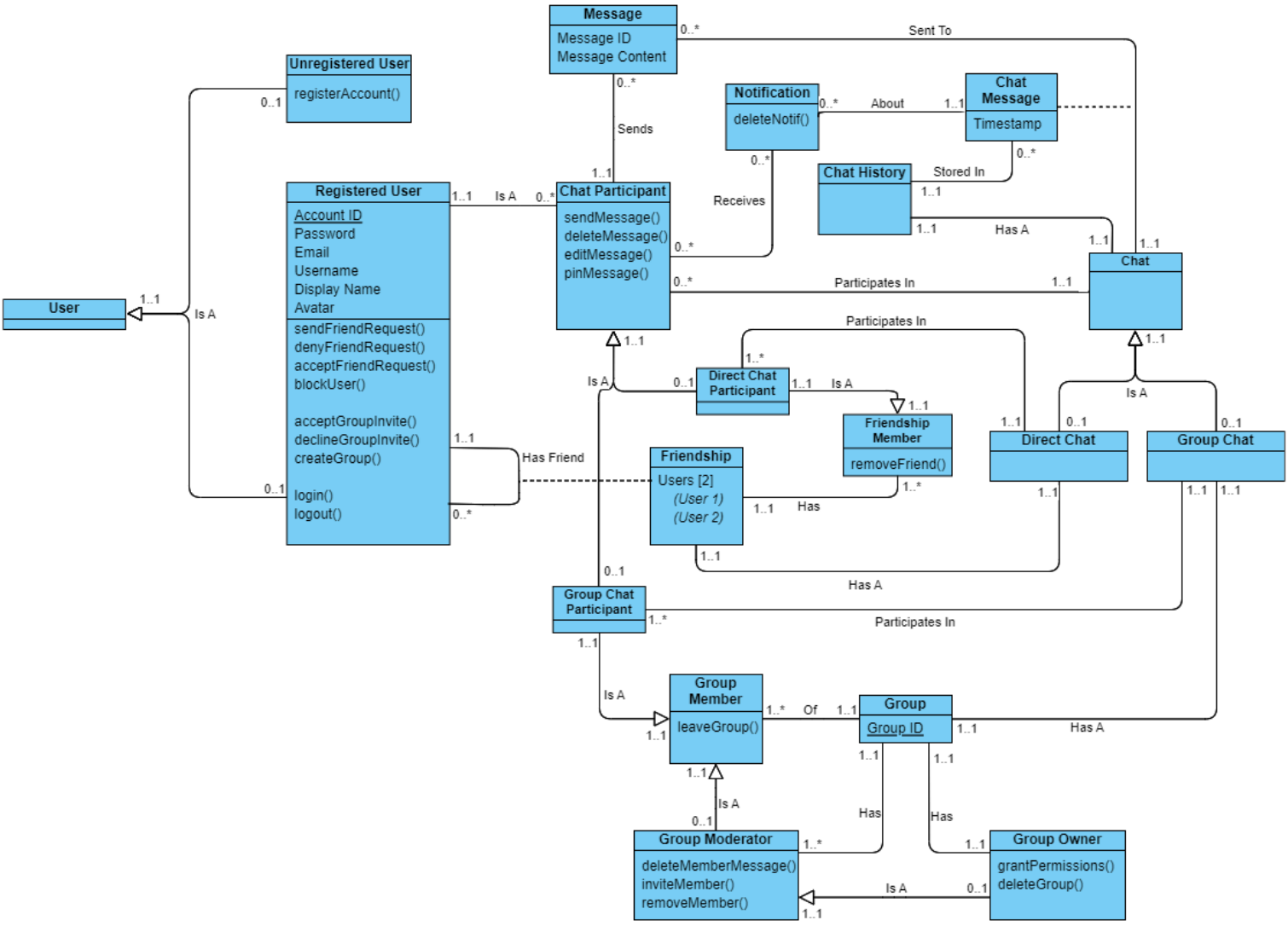
The second design pattern that we'll implement into our project is strategy, which will make sending and receiving user messages easier. Sent messages must be encoded and later decoded once received, however the encoding and decoding process will differ based on the message type (text, image, ect.). Handling each message type's encoding/decoding independently would produce large amounts of code bloat. Strategies will allow us to define general encode/decode interfaces that get implemented by classes with specific encode/decode algorithms, which removes unnecessary code variation outside the strategies.

Resource: https://refactoring.guru/design-patterns/strategy/csharp/example

Sources
https://bunit.dev

3 Diagrams

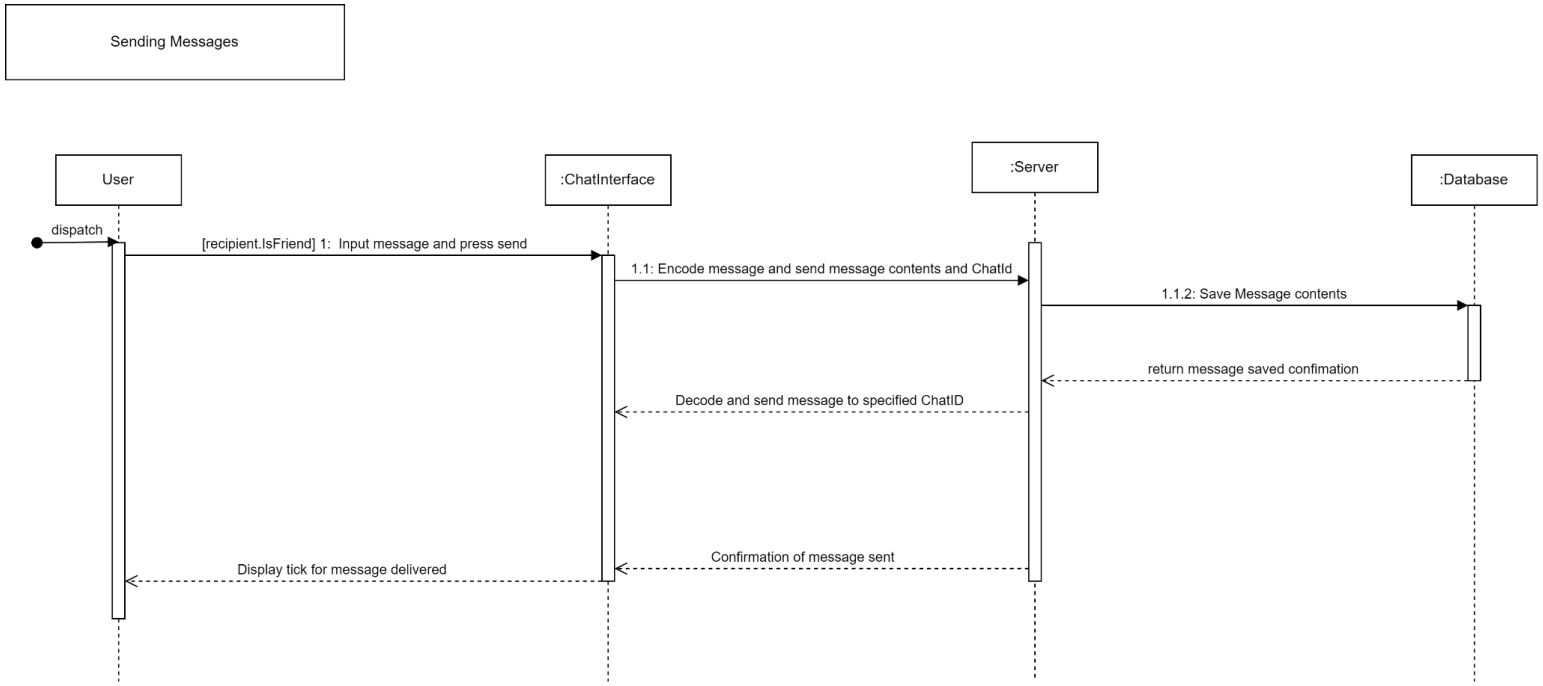
3.1 UML Class Diagram



3.1 Sequence Diagrams

We have created sequence diagrams for our 2 main use cases, sending messages and registration

3.1.1 Sending Messages



3.1.2 Registration