Chat Program Documentation

# Server

To start the server service, open up Developer Command Prompt in Administrator, cd to the release directory, and run:

Installutil Server.exe

To uninstall the service, run:

Installutil /u Server.exe

# Code Contracts

If you want the full experience of debugging this solution, install this Code Contracts extension:

<http://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970>

This will give you static code contract analysis to make sure that you’re not passing in values that will upset your preconditions. Your build may complain if you don’t have this extension as it will build the assemblies using CCRewrite.exe, and whenever it passes a contract it will get upset.

More information on Visual Studio Code Contracts here

<http://devjourney.com/blog/2014/02/12/code-contracts-part-1-introduction/>

# Domain ID Allocation

A domain object has to have a unique ID. This ID is generated in the Server. Each Domain object has a respective IDGenerator which is in charge of creating a unique ID for the object, and only the Server has instances of these generators. This is created in the EntityGeneratorFactory.

Notice that this is not a static class. The Server has to have an instance of all generators for NextID to correctly iterate for each new entity. This is so that this functionality can be unit tested.

The IDGenerator was created to make a seperate location for Domain ID creation. ID iteration could not be done inside the domain class as domain objects were recreated in both Server and Client. This meant that Domain objects that were equal were getting independent IDs. To solve this problem, only the Server could have permission to create and allocate Domain IDs using the IDGenerators. Then when a Domain object was allocated an ID, it could just pass this ID along with the object creation elsewhere. For example, when a new Client has logged in and the Server has allocated the Client a unique User ID, the Client must create its own instance of the User with the same unique ID. For this to happen, the User object is overloaded with the following:

var newUser = new User(clientLogin.User.Username,

entityIDGenerator.GetEntityID<User>(), ConnectionStatus.Connected);

This creates a new User with a username, a new unique ID, and a connection status of connected.

# Login Procedure Sequence Diagram (outdated)

The login procedure has grown significantly since the start of development. Whereas before the client would send a LoginRequest and assume it was connected without any verification from the server. Now there are a handful of protocol messages related to the client login procedure.

Firstly the client sends a login request and the server responds by creating a User object with the client’s username and a unique ID created by the server’s UserFactory. The server then sends a new UserNotification to all currently connected clients with the user, adds the new user to the server’s list of connected clients and a LoginResponse to the newly connected user with the unique User ID. Once completed, the newly connected client sends a UserSnapshotRequest message to the server and the server will then give a list of connected Users back to the new client.

The purpose of these processes is to get the new client, the server and the existing clients all synchronised with the same user data.

The sequence diagram below shows this process.



# New Conversation Sequence Diagram (outdated)

This sequence of events makes sure that the server and the appropriate clients know that there is a conversation.

Each client has their own list of Conversations, and a ConversationNotification message lets the client know that they are in a new conversation.

