Anderson.NET Developer Documentation*

Václav Luňák

August 12, 2019

*written for version 0.1.0

Contents

1 Overview

1.1 About Matrix

Matrix is an open standard for secure, decentralized real-time communication. It is maintained by the Matrix.org Foundation and currently in version 1.0. Visit Matrix.org for more information about Matrix.

1.2 About Anderson.NET

Anderson.NET is an experimental Windows client for the Matrix standard. It is written in C# using the WPF framework and relying on the Matrix .NET SDK, originally created by Will Hunt.[1].

This program was written by me as part of my .NET university courses.

2 Structure

2.1 Solution structure

The project solution[2] consists of three separate projects. In addition to these projects, a modified version of the Matrix SDK[3] is required to build it. The SDK repository and the Anderson repository must be part of the same top-level directory in order to successfully build the solutions.

2.2 Program architecture

The application is created according to the MVVM paradigm. This separates the code into three parts: the view, representing the actual user interface; the model, representing business logic and network connections; and the view model, which serves as a programmatic representation of the view data and connects it to the model without being tied to a specific UI representation.

In addition to the above, the project contains a collection of custom structures used. A dedicated section below will explain in more detail the function of each of these parts.

3 The Model

3.1 Function

In Anderson.NET, the model connects to the SDK and provides data to the application. It consists of two classes: LoginModel, providing user authentication functionality, and RoomModel, responsible for sending and receiving server events, getting user and room information etc.

[1] https://github.com/Half-Shot
[2] https://github.com/lunakv/Anderson.NET
[3] https://github.com/lunakv/matrix-dotnet-sdk

3.2 Client provider

In some situations, the MatrixClient instance whose API the model uses needs to be destroyed. This can happen for example when a user logs out of their current session. In some cases, such as another login, a new instance must be provided for the newly created session. To assure consistent state, all models must share the same MatrixClient instance at all times. The ClientProvider class exists as a solution to this problem.

Instead of registering a MatrixClient directly, the models are injected with an instance of ClientProvider. This class provides the underlying registered client as well as methods for registering new clients and disposing of old ones. Any time the current client is replaced, it raises the **ClientStarted** event, which models can attach to in order to maintain their state.

As a consequence, models mustn't save or directly access any MatrixClient instances. They must use the ClientProvider.Api property instead. If a method can be run asynchronously, a null check on this property is encouraged in case the client is in the middle of resetting.

3.3 Asynchronous methods

To improve user experience, both model classes provide asynchronous versions of most of their methods. These methods are non-blocking, operate on a separate worker thread, and raise an appropriate event upon completion. These methods are intended to be the default in a UI application.

To see an example, let's look at the **ConnectToServer** method. The LoginModel provides the following signature

```
public delegate void ConnectHandler(string error, string url);

public string ConnectToServer(string url);
public void ConnectToServerAsync(string url);
public event ConnectHandler ConnectCompleted;
```

The synchronous **ConnectToServer** method runs synchronously on the calling thread and returns null if successful and an error message otherwise.

The asynchronous **ConnectToServerAsync** method creates a separate thread, on which it invokes its synchronous counterpart. Upon completion, it raises the **ConnectCompleted** event with both the result and the original argument.

All time consuming operations should provide an asynchronous version following the usage and the naming convention of the above example.

4 The View Models

4.1 Function

The view models contain the data displayed in the application and the interaction logic. They transfer information between the models and the views without relying on their

implementation (with a single exception - see 4.4).

4.2 Delegation

The main view model of the application is ApplicationViewModel. This view model doesn't represent any particular view; instead, it creates all other view models, binds on the application window and delegates which view model is active at any given moment.

The active view model is set in the **CurrentPageViewModel** property of ApplicationViewModel. To change its value, the ViewModelBase base class contains the **ViewChanged** event. This event is parametrized by the ViewID property of the target view model. This property must be unique for each view model.

On startup, the ApplicationViewModel registers a handler to this event for each view model it creates. These view models can then access it via the **RaiseViewChanged** base method. After the change is completed, the **SwitchedToThis** virtual method is called on the new view model to activate it.

4.3 Data binding

The view must be notified whenever a view model property it is bound to is changed. This is why ViewModelBase implements the INotifyPropertyChanged interface. Any property that can be changed from within the view model must call the OnPropertyChanged in its setter, parametrized by its name.

Any collection property that can change its items must be of type ObservableCollection (or implement INotifyPropertyChanged and INotifyCollectionChanged).

4.4 Password extracting

For security related reasons, the WPF PasswordBox has no bindable properties on its content. This means the LoginViewModel has no simple way of obtaining the user password.

In Anderson.NET, this issue was solved by passing the entire PasswordBox object as an argument to the login command and accessing its properties programmatically. While this breaks the separation of responsibilities, as the LoginViewModel now relies on a specific implementation of the password dialog, it was deemed an acceptable compromise for this scenario.

5 The Views

5.1 Function

The views provide the user interface to the application. As such, they should not contain any application logic. Instead, they bind to properties of the view models and activate its commands. This means that, with a single exception, they consist of pure XAML with no code behind.

5.2 Data templates and conversion

The main application view, ApplicationWindow, is bound to the ApplicationViewModel. It defines an appropriate view as a data template for each available view model, then binds its content on the **CurrentPageViewModel** property. Thus, whenever the property is changed, it is reflected by displaying the corresponding view. The LoginView uses this principle of templating view models as views to display saved login tokens, while the UserView employs it for showing new invites.

In addition, data templates for the current message history (saved in an AndersonRoom) are included. Since the time information is saved as a DateTime instance, which has no bindable properties, a converter was added to display it in a desired string format.

5.3 Resizing

The WPF Grid element provides a way to create responsive applications viewable at almost any screen size or ratio. Right now, the views in Anderson.NET do not take advantage of these features, using instead hardcoded positional values for all its elements.

Because of this, the application window can not be maximized or resized in any way. Minimization of the window is of course still possible.

6 Structures

6.1 Function

Anderson.NET uses a custom set of classes to hold data about messages, logins, and invites.

6.2 Message classes

A single text message is stored in an AndersonMessage instance. Apart from the message itself, the class contains information about its sender, sent time and sending status (for your own pending messages, currently not implemented).

Multiple messages from the same sender in a short time frame are grouped into an AndersonParagraph. Apart from this collection, it includes a display name of said sender.

AndersonRoom is the set of AndersonParagraphs belonging to a single room, representing its message history. In addition, it provides an easy way of adding new messages to this collection.

6.3 Token keys

A single user can have at most one login token from any given server. Token keys are used to uniquely identify each server-user pair.