



Realtime Messaging for Unity

About

Realtime Messaging is a global Pub/Sub (publish/subscribe) messaging service.

Realtime Cloud Messaging uses the Pub/Sub (publish/subscribe) messaging pattern, where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers.

Requirements

Unity 5

Supported Platforms

Standalone, Webplayer, Android, iOS, Windows Store

Installation

The Realtime Messaging Client for Unity3d is distributed as a **.unityPackage**. Here are the steps needed to install the Realtime Messaging Client.

Installation

- Download the Realtime Messaging Client **.unityPackage**
- Open Unity. It is suggested you start a new project.
- Double click the **.unityPackage**
- You will be prompted with an import dialog. Import the package.
- Move the Plugin Package to root
 - From /Assets/Realtime/Plugins --> /Assets/Plugins

Configuration

Now that the product has been installed, you must configure it. Configuration is managed by a scriptable object. This object may be accessed by using the **Realtime/Messenger Settings** main menu command.

Account

- **Application Key** : Identifies your application. This may be acquired from the your account information on the Realtime website.
- **Private Key** : A secret token used for authorizing clients. If you want to use network security and wish to authenticate directly from the client you will need this.

Service Url

- **Service Url** : The location of the realtime server
- **Is Cluster** : Is this server a cluster instance ?

Authentication

- **Authentication Time** : If using Client Authentication, this is how long a authentication session will be valid. This is a sliding expression.
 - **Is Private** : Limits the authentication token to a single client instance.

Quick Start

Life cycle

The general life cycle of the messenger client is as follows

- **Create** a new messenger client
- **Connect** the client to the gateway
- **AddListeners** to receive messages by the client
- **RemoveListeners** to stop receipt of messages
- **Subscribe** the client to various topic channels
- **Send** messages
- When complete, **Disconnect** the client
- If the application is pausing call **Pause**. Pause will disconnect the client but cache the subscribed channels.
- When the application resumes call **Resume**. This will reconnect the client and resubscribe to subscribed channels

Tasks

The messenger client makes use of tasks as a strategy to deal with long running operations. Tasks are like Coroutines with return results. In your code it is suggested you use coroutines when interacting with the messenger. Take for instance the connection task.

```
IEnumerator Connect()
{
    // Create the task
    var task = Messenger.Connect(AuthToken);

    // Wait for the task to complete
    yield return StartCoroutine(task.WaitRoutine());

    if (task.IsFaulted)
        Debug.LogException(task.Exception);
}
```

Example Use

Under **Assets/RealtimeDemos** there is a demonstration scene named "**RealtimeTest**" this scene shows all functionality of the realtime messaging. This scene has a single relevant script "RealTimeTest.cs" . The remainder of this documentation will explain this script.

Create

On Awake we create an instance of our RealTimeMessenger object. This object is the main interface object to the realtime network.

```
private RealtimeMessenger Messenger { get; set; }

protected void Awake()
{
    Messenger = new RealtimeMessenger();
}
```

Next, It is suggested that you subscribe to the messenger's events. This will allow you to read messages from the network as they are published.

```
Messenger.OnMessage += OnMessage;

void OnMessage(string channel, string message)
{
    // Hello World
}
```

Authorize

If you require authentication, now is the time to authenticate. Authentication, like most Messenger methods make use of tasks. Tasks require to co-routines to start properly.

```
IEnumerator Auth()
{
    // Set the AuthenticationToken
    // This token is a unique key that the client may use to authorize itself.
    Messenger.AuthenticationToken = AuthToken;

    // Create the task
    var task = Messenger.PostAuthentication(Permissions);

    // Wait for it
    yield return StartCoroutine(task.WaitRoutine());

    if (task.IsFaulted)
        // Handle Errors
    else
        // Handle Success
}
```

Connect

Now that the client has been authorized, it is time to connect to the network.

```
IEnumerator Connect()
{
    // Set the ConnectionMetadata
    // Encode identifying data such as Username here
    Messenger.ConnectionMetadata = ClientMetaData;

    // Create the task
    var task = Messenger.Connect(AuthToken);

    // Wait for it
    yield return StartCoroutine(task.WaitRoutine());

    if (task.IsFaulted)
        // Handle Errors
    else
        // Handle Success
}
```

Disconnect

Likewise, disconnection works the same way.

```
IEnumerator Disconnect()
{
    var task = Messenger.Disconnect();

    yield return StartCoroutine(task.WaitRoutine());

    if (task.IsFaulted)
        // Handle Errors
    else
        // Handle Success
}
```

Subscribe / Unsubscribe

Realtime uses a pub/sub messaging pattern. You will need to subscribe to channels to begin receiving messages. You may also unsubscribe to channels at will.

```
IEnumerator Subscribe()
{
    var task = Messenger.Subscribe(Channel);

    yield return StartCoroutine(task.WaitRoutine());
}

IEnumerator Unsubscribe()
{
    var task = Messenger.Unsubscribe(Channel);

    yield return StartCoroutine(task.WaitRoutine());
}
```

Sending

Now it is time to send messages.

```
IEnumerator Send()
{
    var task = Messenger.Send(Channel, Message);

    yield return StartCoroutine(task.WaitRoutine());
}
```

Presence

Presence is a way of knowing who is online. The presence response includes the client metadata.

```
IEnumerator RequestPresence()
{
    var task = Messenger.GetPresence(AuthToken, Channel);

    yield return StartCoroutine(task.WaitRoutine());

    if (task.IsFaulted)
        // Handle Errors
    else
    {
        Debug.Log(String.Format("Subscriptions {0}", task.Result.Subscriptions));
        Debug.Log(String.Format("Metadatas {0}", task.Result.Metadata.Count));

        if (task.Result.Metadata != null)
        {
            foreach (var metadata in task.Result.Metadata)
            {
                Debug.Log(metadata.Key + " - " + metadata.Value);
            }
        }
    }
}
```