Metajestic Multiplayer API

v1.200

Contents

Introduction on use 5
Websocket Connections 6
Socket.io Client Implementations: 6
JavaScript 8
Example Code: 8
Unreal Engine: 9
Socket.io WebSocket: 9
Raw WebSockets: 11
Unity 15
Socket.io WebSocket: 15
Raw WebSocket: 18
Python 20
Socket.io WebSocket: 20
Data Structures 23
Structured Data - persistent 23
Unstructured Data - persistent 23
Example of a user data set: 23
Data which is ephemeral (non-persistent) 23
Receiving Messages 24
Message types: 24
error: 24
connect_error 24
connect 24
disconnect 24
my join collision 24
my join 24
user join 24
user disconnect 24
closest users 25
users by distance 25
update state 25
update state out of range 25

my update state 25
update attributes 25
my update attributes 25
update action 25
message 26
send chat 26
get location 26
Example Receiving Message Processing: 27
Sending Messages 33
Message types: 33
update state 33
update attributes 33
update action 33
send chat 33
get location 34
getClosestUsers 34
getUsersByDistance 34
Example Sending Message Processing: 35
Message Data Schemas - Receive 37
Message Type: “my join” 37
Message Type: “user join” 38
Message Type: “my update state” 38
Message Type: “update state” 38
Message Type: “my update attributes” 39
Message Type: “update attributes” 39
Message Type: “send action” 39
Message Type: “send chat” 40
Message Type: “get location” 40
Message Type: “message” 40
Message Type: “closest users” 41
Message Type: “users by distance” 42
Message Data Schemas - Send 43
Send Message Type: “update state” 43

Send Message Type: “update attributes”....43
Send Message Type: “update action”....43
Send Message Type: “send chat” 44
Send Message Type: “get location”....44
Send Message Type: “getClosestUsers”....44
Send Message Type: “getUsersByDistance”....44
Send Message Type: “getNumberOfUsers” 44
Information and Support....45

Introduction on use

Metajestic Multiplayer Engine (MME) is designed using standard interfacing technology to 3rd party development environments. In addition, we are actively developing our libraries and SDKs to make it even easier.

The basic interface is via a standard socket.io websocket or raw websocket implementation (raw udp and raw tcp socket connections are under development). You simply connect your front-end digital world or game providing the required attributes and authentication, and your players are connected. Once connected it's easy to update the players position, rotation, status, etc., and to add your custom attributes.

It is recommended that the rate of updates sent from the client is regulated and only reflects changes. This is due to a phenomenon referred to as socket flooding. Socket flooding on the client occurs when the client is sending too many messages over too short a time period, and the result is that the client socket gets congested and unresponsive. The MME can sustain very high volumes of both input and output socket messages (and will queue them when required), but the client cannot. It is therefore recommended that positional/rotational updates are sent only when changes in position/rotation occur, and are sent on a frequency of 4 times per second (or even 2 times per second) – but this will be dependent on the motion required on the client side (2 times, 4 times, 8 times, … per second). Sending the update every frame, for example, will cause the client-side socket flooding.

In order to use the MME, the client must have a valid account with a valid plan. Prior to completing any websocket calls, the client must already have a valid API Key. Via the website, the client logs in and receives a valid API key (see Settings once logged in on the site). Using the API key the client's users can make the websocket connection call. The API key, along with other data must be in the websocket connect query string.

The MME follows a conventional send/receive message paradigm – the user client sends messages and receives messaged to/from the MME. Depending on what is sent to the MME, it will automatically carry out the required processes in order to fulfill the request (ex. update of a user's position ➔ adjust if there is a collision ➔ relay the information back to the sending user that the update is successful ➔ send the update to all other users of interest ➔ recalculate all affected users of interest ➔ etc.) - the user client only needs to send the update and look for the acknowledgment of success, the other users will receive the update.

Websocket Connections

All WebSocket connection calls should be directed to the base WebSocket server allocated to you by Metajestic. It will be of the form:

wss://XXXX.metajestic.io

where XXXX is your allocated subdomain.

Once the initial connection has been authenticated in the MME, and if you have user-authentication turned on and the MME has authenticated the user against your backend end point, the user can begin to send and receive messages. If you do not have user-authentication turned on, once the MME has authenticated the connection, the user can begin to send and receive messages.

The MME currently uses socket.io or raw webSockets for client connections. In order to connect to the MME via Socket.io WebSockets, you can use the following client implementations. There are also examples for connecting via javascript, Unreal Engine and Unity later in the document.

Socket.io Client Implementations:

Language	Website
JavaScript (browser, Node.js or React Native)	- Installation steps - API - Source code
JavaScript (for WeChat Mini-Programs)	https://github.com/weapp-socketio/weapp.socket.io
Java	https://github.com/socketio/socket.io-client-java
C++	https://github.com/socketio/socket.io-client-cpp

Language	Website
Swift	https://github.com/socketio/socket.io-client-swift
Dart	https://github.com/rikulo/socket.io-client-dart
Python	https://github.com/miguelgrinberg/python-socketio
.Net	https://github.com/doghappy/socket.io-client-csharp
Rust	https://github.com/1c3t3a/rust-socketio
Kotlin	https://github.com/icerockdev/moko-socket-io
PHP	https://github.com/ElephantIO/elephant.io
Golang	https://github.com/maldikhan/go.socket.io

JavaScript:

Example Code:

Add scripts to browser:
<script src="https://cdn.socket.io/4.5.4/socket.io.min.js"></script>
<script src="https://metajestic.io/assets/metajestic.js"></script>

The metajestic.js script provides functions that allow for easy handling of the low-level socket communication. The primary entry point is the function:

startMetajesticSocket(apiKey, multiplayerService, roomName, myPartyId, jwt, position, attributes, processMetajesticMessage)

- See below for definitions
- The metajestic.js script provides the socket connection as: socketMetajestic , which can be used to directly send (emit) messages from the user client to the MME, or the preferred method of sending is using the built-in sendMetajesticMessage(sendMessageType, message), which includes error checking etc.
- processMetajesticMessage is a callback function you provide to handle the messages from the MME – it can be any function name you define, but must be passed with the startMetajesticSocket(…) call.

Define socket connector and other connection variables:
let apiKey = your apiKey (usually not in the code but rather obtained from your backend service on login by the user)
let multiplayerService = the MME url that you will connect to (this is also usually not in the code but rather obtained from your backend service on login by the user)
let x=0,y=0,z=0,rx=0,ry=0,rz=0,rw=0;//the initial positional and rotational data for this user
let attributes = {};// any initial client defined attributes (for example: {avatar: “avatarFile”, volume: 23})
let roomName = the room you want this user to join
let partyId= this is the Unique User Id from Your System
let jwt = this User JWT obtained from Your System (leave blank if you are not authenticating your users)
…
In a startup function, usually after successful login (example below), call the startMetajesticSocket function:

```
Function login(){
        … call your login end point/service …
        …. success …
        let position ={x:x,y:y,z:z,rx:rx,ry:ry,rz:rz,rw:rw};
        apiKey = returned apiKey from successful login
        multiplayerService = returned multiplayer service url (MME) you are connecting to from login
        roomName = usually returned from login, but could be in code
        myPartyId = usually returned from login – note this must be unique for each user
        jwt = usually returned from login – if you are not authenticating your users, leave blank
        // replace processMetajesticMessage with your function used as the callback to
        // process incoming messages from the MME (see example below)
        startMetajesticSocket(apiKey,multiplayerService,roomName,myPartyId,jwt,position,attributes,processMetajesticMessage)
}
Login();
```

Unreal Engine:

Socket.io WebSocket:

In order to connect to the MME via Socket.io WebSockets, you can use the following repository for socket.io WebSocket connectivity for Unreal Engine - provided as an example at: [GitHub - getnamo/SocketIOClient-Unreal: Socket.IO client plugin for the Unreal Engine.]

Example websocket (wss) connection query string:
' wss://XXXX.metajestic.io/?roomName=Room%20A&apiKey=99e8dcb3-&partyId=6113-098d-47dcb214&jwt=JhbGciOiJIUzI1kpXVCJ9.JqdU3MJ9.WYOBJ-diZaLaOkWJd5Rn8-Odk&x=5.332&y=4.6767&z=0.078&rx=0&ry=0&rz=22&rw=23&v=0&attributes=%7B%22avatar%22%3A%22https%3A%2F%2Fwww.avatar.com%2F47dc-b214ceb406276c61%2Favatar.glb%22%2C%22volume%22%3A23%2C%22displayName%22%3A%22Jane%22%7D&EIO=4&transport=websocket'

Steps to Connect (using):

1. Install the Plugin:
   Copy the "SocketIOClient-Unreal" plugin folder into your Unreal Engine project's "Plugins" directory. Restart your Unreal Engine project.
2. Add the Socket.IO Client Component:
   Add a "SocketIOClient" component to an actor in your scene. This component will handle the connection to the Socket.IO server.
3. Configure the Connection:
   In the component's settings, you can specify the Socket.IO server's address (URL) and port and query connect parameters (see above). You can also choose to automatically connect on begin play or connect manually via a blueprint node.
4. Establish the Connection:
   You can either let the component auto-connect on game start or manually trigger the connection using a blueprint. If using manual connection, you'll need to call the "Connect" function on the SocketIOClient component.
5. Sending and Receiving Data:
   Sending Data (Emitting Events):
      Use the "Emit" function to send data to the Socket.IO server. You can specify the event name and the data to be sent.
   Receiving Data (Listening for Events):
      You can create a "Receive To Delegate" event to listen for events from the server. When an event is received, the corresponding delegate function will be executed.
   Data Format:

The plugin supports sending and receiving various data types, including strings, numbers, and JSON objects. For MME communication, JSON objects should be used.

JSON Decoding:

If you're receiving complex data, you can use functions to decode JSON objects into Unreal Engine structs or other data structures.

Example (Blueprint):

1. Add a SocketIOClient component to an actor.
2. Configure the server address and port.
3. Add a "BeginPlay" event and call the "Connect" function on the SocketIOClient component.
4. Add a "Receive To Delegate" event and connect it to a custom event.
5. In the custom event, you can process the received data.

// When the user sends a chat message (Example)
Event: User Sends Message (Custom Event)

- Emit (SocketIOClient):
  - Event Name: "update state"
  - Data: “{x:23.4545,z:12.7878}”

// When the client receives a chat message (Example)
Event: OnEvent (from SocketIOClient)

- Get Event Data (S2C): Convert the received JSON data (if it's complex) or directly use the string data.
- Print: Display the chat message to the user.

Raw WebSockets:

In order to connect to the MME via webSockets, you can use either the built-in API (IWebSocket) or one of several plugins in the Unreal Engine Marketplace (and on the Epic Developer Community Forums) that offer WebSocket functionality (ex. UEWebsocket, Internet Protocol Client, and others).

Using IWebSocket interface (part of the Runtime.WebSockets module) for creating and managing WebSocket connections:

Example websocket (wss) connection query string:
' wss://XXXX.metajestic.io/?roomName=Room%20A&apiKey=99e8dcb3-&partyId=6113-098d-47dcb214&jwt=JhbGciOiJIUzI1kpXVCJ9.JqdU3MJ9.WYOBJ-diZaLaOkWJd5Rn8-Odk&x=5.332&y=4.6767&z=0.078&rx=0&ry=0&rz=22&rw=23&v=0&attributes=%7B%22avatar%22%3A%22https%3A%2F%2Fwww.avatar.com%2F47dc-b214ceb406276c61%2Favatar.glb%22%2C%22volume%22%3A23%2C%22displayName%22%3A%22Jane%22%7D&EIO=4&transport=websocket'

Steps to Connect (Using IWebSocket):

1. Enable the Runtime.WebSockets Module:
   Make sure the Runtime.WebSockets module is enabled in your project settings.
2. Create a WebSocket Instance:
   Use the IWebSocketsManager (or its equivalent in the specific plugin you choose) to create a IWebSocket instance.
3. Establish the Connection:
   Use the Connect() method (or similar) of the IWebSocket instance to connect to the specified WebSocket server URL.
4. Handle Connection Events:
   Implement event callbacks to handle connection success, failure, and data reception.
5. Send and Receive Data:
   Use SendMessage() and SendRawMessage() to send data, and OnMessage to receive data from the server.
6. Manage the Connection:
   Use the Close() method (or similar) to disconnect when necessary.

Example Code (using IWebSocket):

Header File (MyWebSocket.h):

```
#pragma once
#include "CoreMinimal.h"
#include "WebSockets/IWebSocket.h"
#include "WebSockets/IWebSocketClient.h"

class UMyWebSocket : public UObject
{
public:
   UMyWebSocket();
   ~UMyWebSocket();

   void ConnectToServer(const FString& ServerURL, const FString& ServerProtocol);
   void SendMessage(const FString& Message);
   void SendRawMessage(const TArray<uint8>& Message);

private:
   TSharedPtr<IWebSocket> MyWebSocket;
   bool bIsConnected = false;
};
```

CPP File (MyWebSocket.cpp):

```
#include "MyWebSocket.h"

UMyWebSocket::UMyWebSocket()
{
}

UMyWebSocket::~UMyWebSocket()
{
}

void UMyWebSocket::ConnectToServer(const FString& ServerURL, const FString&
ServerProtocol)
{
   // Create the WebSocket
   MyWebSocket = FWebSocketsModule::Get().CreateWebSocket();
```

```
    // Set the protocol (optional)
    MyWebSocket->SetProtocol(ServerProtocol);

    // Set up connection/message callbacks
    MyWebSocket->OnConnectionOpen.AddLambda([this](const FString& ServerURL)
    {
        UE_LOG(LogTemp, Log, TEXT("Connected to: %s"), *ServerURL);
        bIsConnected = true;
    });

    MyWebSocket->OnConnectionClosed.AddLambda([this](const FString& ServerURL, const
int32 CloseCode, const FString& Reason)
    {
        UE_LOG(LogTemp, Warning, TEXT("Disconnected from: %s, Code: %d, Reason: %s"),
*ServerURL, CloseCode, *Reason);
        bIsConnected = false;
        MyWebSocket = NULL;
    });

    MyWebSocket->OnMessage.AddLambda([this](const FString& Message)
    {
        UE_LOG(LogTemp, Log, TEXT("Received message: %s"), *Message);
    });

    MyWebSocket->OnRawMessage.AddLambda([this](const TArray<uint8>& Message)
    {
        UE_LOG(LogTemp, Log, TEXT("Received raw message"));
        // Process raw binary data
    });

    // Connect to the server
    MyWebSocket->Connect(ServerURL);
}

void UMyWebSocket::SendMessage(const FString& Message)
{
    if (bIsConnected)
    {
        MyWebSocket->SendMessage(Message);
    }
}
```

```
void UMyWebSocket::SendRawMessage(const TArray<uint8>& Message)
{
   if (bIsConnected)
   {
      MyWebSocket->SendRawMessage(Message);
   }
}
```

Example Usage:

```
// In a UActor or UGameInstance
UMyWebSocket* MyWebSocket = NewObject<UMyWebSocket>();
MyWebSocket->ConnectToServer(TEXT("wss://XXXX.metajestic.io/?..."), TEXT("wss"));

// Later, to send a message
MyWebSocket->SendMessage(TEXT("{type,msg}"));

// Remember to include the WebSocketsModule in your project's build configuration
```

Unity

Socket.io WebSocket:

In order to connect a Unity client to the MME Socket.io WebSocket service, you'll typically use a WebSocket client library like SocketIOClient. You'll need to create a new WebSocket client, connect it to the MME, and handle events like opening, closing, receiving messages, and errors.

Example websocket (wss) connection query string:
' wss://XXXX.metajestic.io/?roomName=Room%20A&apiKey=99e8dcb3-&partyId=6113-098d-47dcb214&jwt=JhbGciOiJIUzI1kpXVCJ9.JqdU3MJ9.WYOBJ-diZaLaOkWJd5Rn8-Odk&x=5.332&y=4.6767&z=0.078&rx=0&ry=0&rz=22&rw=23&v=0&attributes=%7B%22avatar%22%3A%22https%3A%2F%2Fwww.avatar.com%2F47dc-b214ceb406276c61%2Favatar.glb%22%2C%22volume%22%3A23%2C%22displayName%22%3A%22Jane%22%7D&EIO=4&transport=websocket'

Steps to Connect:

1. Import the WebSocket Library:
   Import the SocketIOClient into your Unity project.
2. Create a WebSocket Client:
   Use the WebSocket client library's constructor to create a new WebSocket instance, providing the server's address (see example connection query string above)
3. Handle Connection Events:
   OnOpen: Called when the connection is established.
   OnError: Called when an error occurs during the connection or communication.
   OnClose: Called when the connection is closed.
   OnMessage: Called when a message is received from the server.
4. Connect to the Server:
   Use the client's Connect() method to establish the WebSocket connection.
5. Send Messages:
   Use the client object's Emit() or EmitAsync() method to send events and data to the server.

Example Code (using SocketIOClient):

```
using System;
using System.Collections.Generic;
using SocketIOClient;
using SocketIOClient.Newtonsoft.Json; // If using Newtonsoft.Json
using UnityEngine;
using UnityEngine.UI;
```

```
using Newtonsoft.Json.Linq; // If using Newtonsoft.Json

public class SocketIOClientExample : MonoBehaviour
{
    private SocketIOUnity socket;
    private string serverUrlLink = "your_server_url"; // Replace with your server's URL

    void Start()
    {
        var uri = new Uri(serverUrlLink);
        socket = new SocketIOUnity(uri);

        // Optional: Configure JSON serializer if needed (e.g., for Newtonsoft.Json)
        // socket.JsonSerializer = new NewtonsoftJsonSerializer();

        // Connect to the server
        socket.Connect();

        // Register event handlers
        socket.OnConnected += (sender, e) => {
            Debug.Log("Socket.IO connected!");
        };

        socket.OnDisconnected += (sender, e) => {
            Debug.Log("Socket.IO disconnected!");
        };

        socket.OnError += (sender, e) => {
            Debug.LogError("Socket.IO error: " + e);
        };

        // Register for custom events
        socket.On("message", response => {
            Debug.Log("Received message: " + response.GetValue<string>());
        });
    }

    // Example of sending a message
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
```

```
            socket.EmitAsync("message", "Hello, server!"); // Replace with your message
        }
    }

    void OnDestroy()
    {
        // Dispose of the socket connection when the object is destroyed
        if (socket != null)
        {
            socket.Dispose();
        }
    }
}
```

Raw WebSocket:

In order to connect a Unity client to the MME WebSocket service, you'll typically use a WebSocket client library like WebSocketSharp or NativeWebSocket. You'll need to create a new WebSocket client, connect it to the MME, and handle events like opening, closing, receiving messages, and errors.

Example websocket (wss) connection query string:
' wss://XXXX.metajestic.io/?roomName=Room%20A&apiKey=99e8dcb3-&partyId=6113-098d-47dcb214&jwt=JhbGciOiJIUzI1kpXVCJ9.JqdU3MJ9.WYOBJ-diZaLaOkWJd5Rn8-Odk&x=5.332&y=4.6767&z=0.078&rx=0&ry=0&rz=22&rw=23&v=0&attributes=%7B%22avatar%22%3A%22https%3A%2F%2Fwww.avatar.com%2F47dc-b214ceb406276c61%2Favatar.glb%22%2C%22volume%22%3A23%2C%22displayName%22%3A%22Jane%22%7D&EIO=4&transport=websocket'

Steps to Connect:

1. Import the WebSocket Library:
   - WebSocketSharp: Add the WebSocketSharp.dll file to your Unity project's Assets folder.
   - NativeWebSocket: You can either install it via the Unity Package Manager (UPM) or download the source code and copy it to your project.
2. Create a WebSocket Client:
   - Use the WebSocket client library's constructor to create a new WebSocket instance, providing the server's address (see example connection query string above)
3. Handle Connection Events:
   - OnOpen: Called when the connection is established.
   - OnError: Called when an error occurs during the connection or communication.
   - OnClose: Called when the connection is closed.
   - OnMessage: Called when a message is received from the server.
4. Connect to the Server:
   - Use the client's Connect() method to establish the WebSocket connection.
5. Send Messages:
   - Use the client's Send() method to send data to the server.

Example Code (using NativeWebSocket):

```
using UnityEngine;
using NativeWebSocket;

public class WebSocketClient : MonoBehaviour
{
```

```
    private WebSocket websocket;

    async void Start()
    {
        websocket = new WebSocket("wss://XXXX.metajestic.io"); // Replace with your server's address

        websocket.OnOpen += () => { Debug.Log("Connection open!"); };
        websocket.OnError += (e) => { Debug.Log("Error: " + e); };
        websocket.OnClose += (e) => { Debug.Log("Connection closed!"); };
        websocket.OnMessage += (bytes) =>
        {
            Debug.Log("Received message: " + System.Text.Encoding.UTF8.GetString(bytes));
        };

        await websocket.Connect();
    }

    void Update()
    {
        if (websocket != null && websocket.State == WebSocketState.Open)
        {
            if (Input.GetKeyDown(KeyCode.Space)) // Example: send a message on spacebar press
            {
                websocket.SendText("Hello from Unity!");
            }
        }
        #if !UNITY_WEBGL || UNITY_EDITOR
        websocket.DispatchMessage();
        #endif
    }

    async void OnApplicationQuit()
    {
        if (websocket != null)
        {
            await websocket.Close();
        }
    }
}
```

Python

Socket.io WebSocket:

In order to connect a Python client to the MME Socket.io WebSocket service, you'll typically use a WebSocket client library like the socket.io. You'll need to create a new WebSocket client, connect it to the MME, and handle events like opening, closing, receiving messages, and errors.

Example websocket (wss) connection query string:
' wss://XXXX.metajestic.io/?roomName=Room%20A&apiKey=99e8dcb3-&partyId=6113-098d-47dcb214&jwt=JhbGciOiJIUzI1kpXVCJ9.JqdU3MJ9.WYOBJ-diZaLaOkWJd5Rn8-Odk&x=5.332&y=4.6767&z=0.078&rx=0&ry=0&rz=22&rw=23&v=0&attributes=%7B%22avatar%22%3A%22https%3A%2F%2Fwww.avatar.com%2F47dc-b214ceb406276c61%2Favatar.glb%22%2C%22volume%22%3A23%2C%22displayName%22%3A%22Jane%22%7D&EIO=4&transport=websocket'

Steps to Connect:

1. Install the WebSocket Library:
   pip install "python-socketio[client]".
2. Create a WebSocket Client:
   Use the WebSocket client library's constructor to create a new WebSocket instance, providing the server's address (see example connection query string above)
3. Handle Connection Events:
   connect: Called when the connection is established.
   error: Called when an error occurs during the connection or communication.
   disconnect: Called when the connection is closed.
   message: Called when a message is received from the server.
4. Connect to the Server:
   Use the client's connect() method to establish the WebSocket connection.
5. Send Messages:
   Use the client object's emit() method to send events and data to the server.

Example Code (using socket.io client):

```
#
# Python script to connect to Metajestic Multiplayer Engine
#
# Uses socket.io as example
# install: pip install "python-socketio[client]"
#       or if instead you plan on using the asyncio client, then use this:
#       pip install "python-socketio[asyncio_client]"
#
import socketio
from urllib.parse import quote
sio = socketio.Client()
```

```
# user variables examples
apiKey = "99e8dcb3-3ea0…"
jwt = ""
partyId = "Python-f8eae3f3-03b0-440e-8e20-e917b11cf…"
wsHost = "ws://xxxx.metajestic.io…"
roomName = "The Bridge"
x=5.332
y=4.6767
z=0.078
rx=0
ry=0
rz=22
rw=23
attributes=quote('{ "avatar": "https://www.avatar.com/Python-f8eae3f3-03b0-440e-8e20-e917b11cf16d/avatar.glb",
"volume": 23, "displayName": "PythonTom" }')
metajesticSocketConnectString =
wsHost+'/?roomName='+quote(roomName)+'&apiKey='+apiKey+'&partyId='+partyId+'&jwt='+jwt+'&x='+str(x)+'&y='+st
r(y)+'&z='+str(z)+'&rx='+str(rx)+'&ry='+str(ry)+'&rz='+str(rz)+'&rw='+str(rw)+'&attributes='+attributes

# example receiving the connect event from the server
@sio.event
def connect():
    print('connected to server')
    print('connected to room',roomName)

# example receiving the disconnect event from the server
@sio.event
def disconnect(reason):
    print('disconnected from server, reason:', reason)

# example receiving the message event from the server
@sio.event
def message(data):
    print('message',data)

# example receiving the 'my join collision' event from the server
# this is received if the user attempts to position themselves
# in a location that is already occupied by another user
# It provides the new location of this user in the data as newX, newY, newZ
@sio.on('my join collision')
def on_message(data):
    print('my join collision',data)

# example receiving the 'my join' event from the server
# the user will always receive this event when they have successfully
# connected and been positioned in the 3d space.
# it also contains the array of the set of closest users
@sio.on('my join')
def on_message(data):
    print('my join',data)
    # example showing sending an 'update state' event with
    # associated data of a change in x and z coordinates
```

```
    sio.emit('update state',{ "x": 23, "z":12 })

# example receiving the 'my update state' event from the server
# check for the collision element (boolean) if set to false, the
# user has successfully updated their state (ex. position) in the engine
# and the other users have been notified via the 'update state' event
@sio.on('my update state')
def on_message(data):
    print('my update state',data)
    print('my update state - collision:',data["data"]["collision"])

# example receiving the 'my update attributes' event from the server
@sio.on('my update attributes')
def on_message(data):
    print('my update attributes',data)

# catch-all for all other events, passing them to the processMessage function
# note this structure could be used for the above events also, which are included
# as examples for handling events that deal with this user specifically
@sio.on('*')
def any_event(event, data):
    processMessage(event,data)
    pass

def processMessage(event,data):
    # process messages
    match event:
        case "user join":
            print("user join",data)
        case "update state":
            print("update state",data)
        case "update state out of range":
            print("update state out of range",data)
        case "update attributes":
            print("update attributes",data)
        case "closest users":
            print("closest users",data)
        case "users by distance":
            print("users by distance",data)
        case "send action":
            print("update action",data)
        case "send chat":
            print("send chat",data)
        case "get location":
            print("get location",data)
        case _:
            print("unprocessed event",event,"with data",data)


if __name__ == '__main__':
    sio.connect(metajesticSocketConnectString,transports='websocket')
    sio.wait()
```

Data Structures

The data used in the MME is both structured and unstructured (extensible and defined by the client) and persistent or non-persistent. The structured data is a preset defined data set, whereas the unstructured data allows the client to define the elements and their values.

Structured Data - persistent

The structured data stores:

- PartyId, the unique identifier of the party as defined by the client: partyId: “abcdef”
- Positional data as x:0,y:0,z:0
- Rotational data as: rx:0.ry:0,rz:0,rw:0
- “v” element which can be used to store velocity or volume

Unstructured Data - persistent

The unstructured data is stored with the structured data, “follows” the user, and is stored in an element called “attributes”. Examples could be:

- attributes: { displayName: “Tom”, avatar: “https://www.avatar.io/tom.glb”, volume: 22 }

Example of a user data set:

```
{
        x:x,
        y:y,
        z:z,
        rx:rx,
        ry:ry,
        rz:rz,
        rw:rw,
        v:v,
        attributes: {
                avatar: "https://www.avatar.com/"+myPartyId+"/avatar.glb",
                volume: 23,
                displayName: myName
                }
}
```

Data which is ephemeral (non-persistent)

Other data that is ephemeral includes the data of message types of chat, actions, “message”, number of users, etc. are not stored in the MME.

Receiving Messages

Message types:

These are defined as an array called: receiveMessageType in the metajestic.js script.

error:

This message is received if there is an error detected on the socket connection.

connect_error

This message is received if there is an error on connecting the socket.

connect

This message is received if the socket connection was successful in connecting to the service.

disconnect

This message is received if the socket has been disconnected from the server side.

my join collision

This message is received when a user is attempting to connect to the service and the MME has detected that the user is trying to connect at the same spatial location as another user already occupies. It will automatically determine and provide the new coordinates of a location that the user has been moved to that is unoccupied by another user in order to join.

my join

This message is received when a user successfully joins the room in a service and there is no collision (see above). It contains the user's spatial position, rotational position, defined attributes, etc. of the joining user, and provides an array of closest users to them.

user join

This message is received when another user joins the same room/zone that has joined in an area of interest to the receiving user – they are in the sphere of interest, and the front end should decide where/how to render this new user.

user disconnect

This message is received when another user disconnects (leaves) a room/zone. The front end should decide what to do with this disconnected user (ex. delete them from view).

closest users

This message is received both periodically and on demand. It provides an array of the users that are closest to you.

users by distance

This message is received when a user sends a request for the users that are within a given distance. It provides an array of users that are within that specified (or defaulted) distance. Note the distance is calculated based on 3d space (x,y,z) coordinates.

update state

This message is received when another user updates their state (position, rotation, etc. but not attributes). The front end should update the position/rotation/etc. of this user based on the incoming data.

update state out of range

This message is received when another user updates their state to be “out of range” based on the specified distance. If the user receives this message, the front end should likely remove this user from view.

my update state

This message is received as a confirmation that the MME has received and validated no collisions with other users (if turned on), and this user is at that location according to the MME (and all other users in the room/zone are updated with the new state).

update attributes

This message is received when another user updates their attributes. The front end should update the attributes of this user based on the incoming data.

my update attributes

This message is received as a confirmation that the MME has received and validated this user updates to their attributes. All other users of interest in the room/zone are updated with the new/updated attributes.

update action

This message is received when another user sends an action that they are currently doing. Examples are jump, running on the spot, pushups, bow, etc. These are definable by the client and these are non-persistent data – the action is sent to the other users of interest at the moment and not stored in the MME.

message

This message is received when the server sends a message to the user. The message has a message type and the message content.

send chat

This message is received when another user sends a text chat message This message is non-persistent, no history of chat is currently stored on the MME.

get location

This message is received when the user has requested getting the location of another user, whether the other user is in their room or not, as long as the other user is in the same client spatial web (metaverse).

Example Receiving Message Processing:

The following are some examples of processing, using the callback function processMetajesticMessage:

```
// define a local map to hold all the players (users) of interest (including the current user)
let players = new Map();
let myPosition = {};
let myAttributes = {};
let x=5,y=4,z=0,rx=0,ry=0,rz=22,rw=23;
let distanceOfInterest = 100;
let initialConnection = false;
let reconnection = false;

function processMetajesticMessage(msgType,msg){
	let player;
	let position = {};
	if ( receiveMessageType.indexOf(msgType) >= 0 ){
	   console.log("Processing",msgType,msg);
	   switch (msgType) {
	      case "connect_error":
	         console.log(data || 'connect_failed');
	         break;
	      case "error":
	         console.log(data || 'error');
	         break;
	      case "disconnect":
	         console.log("multiplayer socket connected:",socketMetajestic.connected);
	         break;
	      case "connect":
	         if ( !initialConnection ) initialConnection = true;
	         console.log("multiplayer socket connected:",socketMetajestic.connected);
	         break;
	      case "my join collision":
	         // tried to join at a location another player was at
	         // engine found vacant location for me, so set my internal coords to new values
	                  if ( !reconnection && msg.data.collision ){
	                           x = msg.data.newX;
	                           y = msg.data.newY;
	                           z = msg.data.newZ;
	                  }
	                    myPosition = { "x": x, "y": y, "z": z, "rx": rx, "ry": ry, "rz": rz, "rw": rw};
	         break;
	      case "my join":
	                  // successfully joined
	                  // check if reconnection
	                  if (reconnection) { reconnection = false; reconnectionUpdates();}
	                  // add me to local map array in this example
```

```
                players.set(myPartyId,{ "x": x, "y": y, "z": z, "rx": rx, "ry": ry, "rz": rz, "rw": rw, "partyId": myPartyId,
"res": [] });
                myPosition = { "x": x, "y": y, "z": z, "rx": rx, "ry": ry, "rz": rz, "rw": rw};
                // add to visual display …
                // the my join array also includes an array of the initial set of other users that are closest to this user
                // add the other players to local map-array in this example
                for ( let i=0; i < msg.data.length; ++ i ){
                        // for this example, we check to see if the user is within a certain
                        // distanceOfInterest (set locally) if so, add to local player map
                        if ( isPlayerOfInterest(distanceOfInterest,msg.data[i].x,msg.data[i].y,msg.data[i].z) ){
                                players.set(msg.data[i].partyId,msg.data[i]);
                                // add to visual display …
                                position =
                        {x:msg.data[i].x,y:msg.data[i].y,z:msg.data[i].z,text:"P"+i,partyId:msg.data[i].partyId};
                }
           break;
        case "user join":
                // a new user has joined the room
                // determine whether or not the user is of interest, if so add
                // note you will only receive “user join” when the MME has determined that the user
                // is or may be of interest to you
                if ( msg.info.partyId != myPartyId ) {
                        if ( isPlayerOfInterest(distanceOfInterest,msg.info.x,msg.info.y,msg.info.z) );
                                players.set(msg.info.partyId,msg.info);
                                // add to visual display
                                position =
                        {x:msg.info.x,y:msg.info.y,z:msg.info.z,text:”name”,partyId:msg.info.partyId};
                        }
                }
           break;
        case "closest users":
                // Note: the client will receive automatic “closest users” messages periodically calculated by the
                // MME. The client will need to choose what to do with these updates. Typically, they are used
                // to update the current states of players, the visual state, verify that the local user list is in sync
                // with the MME, etc.
                //
                // in the msg.data component of “closest users” message is an array of the X closest users to you
                //
                // since there are no guarantees that sent json is in the exact same order as the received json
                // sort the users by distance from me
                let origin={ "x": x, "y": y, "z": z, "rx": rx, "ry": ry, "rz": rz, "rw": rw, "partyId": myPartyId, "res": [] }
                let sortedArray = msg.data.filter(Boolean).sort((pointa, pointb) => sqDist(origin,pointa)-
                sqDist(origin,pointb));
                // loop over the array
                for (let i=0;i < sortedArray.length;++i){
                        if ( sortedArray[i].partyId != myPartyId ) {
                                position =
                        {x:sortedArray[i].x,y:sortedArray[i].y,z:sortedArray[i].z,text:"P"+i,partyId:sortedArray[i].party
                        Id};
                                // here you could update the players local map to reflect the latest
                        positions/attributes of the closest users
                        }
                }
           break;
```

```
        case "users by distance":
                // in the msg.data component of “users by distance” message is an array of users
                // that are at or less than a specified distance from you. This distance can either be
                // sent in the message request to the MME (see Sending Messages) or is defaulted
                // from settings in your console settings.
                // This can be used in a similar way to closest users, and could be used to redraw
                // the visual space based on users within the specific distance.
                // For example, you could request users that are at a distance of 20 units or less – and with
                // the result set of users, you could enable audio on only that set.
           break;
        case "update state":
                //
                // This message provides update of state from another user (“my update state” is used for this
                // user as below).
                // Note: only the changed data is received in this message. For example, if the user only moves in
                // the x and y coordinates, only the x and y coordinates will be in the message
                //
                // Example:
                // get the player’s most recent data from the local map
                player = players.get(msg.info.partyId);
                if ( player ){
                        Object.keys(msg.info).forEach(key => {
                                player[key] = msg.info[key];
                        });
                        // check if player is still of interest
                        if ( isPlayerOfInterest(distanceOfInterest,player.x,player.y,player.z) ){
                                players.set(msg.info.partyId,player);
                                position =
                        {x:player.x,y:player.y,z:player.z,text:"PlayerName”,partyId:player.partyId};
                        } else {
                                // delete the player from view and remove from local players Map
                                players.delete(player.partyId);
                        }
                } else {
                        // is it an old player that is back in the sphere of interest, need to add to local map
                        let playerInfo = msg.info;
                        // check that the update has all the required fields prior to adding
                        if ( playerInfo.hasOwnProperty("partyId") && playerInfo.hasOwnProperty("x") &&
playerInfo.hasOwnProperty("y") && playerInfo.hasOwnProperty("z") && playerInfo.hasOwnProperty("rx") &&
playerInfo.hasOwnProperty("ry") && playerInfo.hasOwnProperty("rz") && playerInfo.hasOwnProperty("rw") &&
playerInfo.hasOwnProperty("v") ){
                                players.set(playerInfo.partyId,playerInfo);
                                position =
                        {x:msg.info.x,y:msg.info.y,z:msg.info.z,text:"PlayerName"+xData.length,partyId:msg.info.p
                        artyId};
                        }
                }
           break;
        case "update state out range":
                // this message is received when a user that was of interest to you updates their position
                // so that they are no longer of interest. This is sent immediately after the successful move
                // by the other user.
                //
                // Example: get the player from the local map and remove them
```

```
        player = players.get(msg.info.partyId);
        if ( player ){
                // delete the player from view and remove from local players Map
                players.delete(player.partyId);
        }
    break;
case "my update state":
        // this message is received after the user sends an “update state” message to the MME.
        // It will let the user know if there was a collision, and if so the MME does not update the
        // user information on the MME, and does not send an “update state” to all users of interest.
        // If there is no collision, the updated state information is in the msg.data element, and an
        // “update state” message is sent to all users of interest, so they know of your movement.
        //
        if ( msg.data.collision == false ){
                msg.data.collision = null;
                delete msg.data.collision;
                // no collision, update my position
                 player = players.get(msg.data.partyId);
                 if ( player ){
                        Object.keys(msg.data).forEach(key => {
                            player[key] = msg.data[key];
                            if ( positionArray.includes(key) ){
                                eval(key + "=" + msg.data[key]);
                            }
                        });
                   }
                   players.set(myPartyId,player);
                   myPosition = { "x": x, "y": y, "z": z, "rx": rx, "ry": ry, "rz": rz, "rw": rw, "v": v };
        else {
                // you collided with another player
                // multiplayer engine has left you in your previous position
        }
    break;
case "update attributes":
        // this message is received when a user of interest updates their custom user defined
        // attributes.
        // Note these user defined attributes are stored in the MME on the user data, and “follow”
        // the user updates.
        //
        player = players.get(msg.info.partyId);
        if ( player ){
                // if the player does not have any attributes defined, set the default empty set
                if ( !player.attributes ) player.attributes = {};
                // since these attributes are user defined, we iterate through the data
                Object.keys(msg.info.attributes).forEach(key => {
                         player.attributes[key] = msg.info.attributes[key];
                });
                players.set(msg.info.partyId,player);
        }
    break;
case "my update attributes":
        // this message confirms the user has updated their attributes
        // get current data, update this user
        player = players.get(myPartyId);
```

```
                    if ( player ){
                        if ( !player.attributes ) player.attributes = {};
                        Object.keys(msg.data.attributes).forEach(key => {
                            player.attributes[key] = msg.data.attributes[key];
                        });
                        players.set(myPartyId,player);
                    }
                break;
            case "user disconnect":
                    // this message lets the user know that another user has disconnect from the room and/or service
                    //delete the player from view and remove from local players Map
                    players.delete(msg.info.partyId);
                break;
            case "message":
                    // messages from the server
                    console.log("processing metajestic received message",msg.type,msg.message);
                break;
        }
    }
}



function reconnectionUpdates(){
        // socket was disconnected and is now connected - update position and attributes
        let myPlayerInfo = players.get(myPartyId);
        console.log("processing reconnectionUpdates",myPlayerInfo)
        sendMetajesticMessage("update
state",{x:myPlayerInfo.x,y:myPlayerInfo.y,z:myPlayerInfo.z,rx:myPlayerInfo.rx,ry:myPlayerInfo.ry,rz:myPlayerInfo.rz,rw:
myPlayerInfo.rw});
        let attributes = myPlayerInfo.attributes;
        if ( attributes ) sendMetajesticMessage("update attributes",attributes);
        reconnection = false;
}
function changeRoom(newRoom){
        if ( newRoom != roomName ){
            roomName = newRoom;
            startMetajesticSocket(apiKey,multiplayerService,roomName,myPartyId,jwt,myPosition,myAttribute
        s,processMetajesticMessage)
            // reset visuals, removing all other players
            players = new Map();
            players.set(myPartyId,{ "x": x, "y": y, "z": z, "rx": rx, "ry": ry, "rz": rz, "rw": rw, "partyId": myPartyId,
"attributes": myAttributes });
        }
}
  //
  // get users that are closest to you
  // and get users that are within a certain distance from you
  //
  function getClosestUsers(){
    sendMetajesticMessage("getClosestUsers");
  }
```

```
function getUsersByDistance(){
   sendMetajesticMessage("getUsersByDistance");
}
function calculateDistance(x2, y2, z2) {
   const deltaX = x2 - x;
   const deltaY = y2 - y;
   const deltaZ = z2 - z;
   return Math.sqrt(deltaX * deltaX + deltaY * deltaY + deltaZ * deltaZ);
}
//
// when looking at users, are they of interest as determined by their distance from you
//
function isPlayerOfInterest(doi, x2, y2, z2) {
   let interest = false;
   if ( calculateDistance(x2,y2,z2) <= doi ) interest = true;
   return interest;
}
```

Sending Messages

To send messages to the MME simply call sendMetajesticMessage(messageType,message) with the appropriate parameters for the given message or send an “emit” directly on the metajestic socket object (socketMetajestic if you are using the default from the metajestic.js library)

Message types:

These are defined as an array called: sendMessageType in the metajestic.js script.

NOTE that the user joining a given room/zone is handled in the connection query string over the websocket connection.

update state

Sending this message sends the updated state (position, rotation, etc.) to the MME. You should wait for the confirmation to be returned (“my update state”) from the MME before updating locally, but will depend on how your front-end application handles state updates. Note the MME will determine if there is a collision or not (if activated) and if no collision all other users will be sent the updated state.

update attributes

Sending this message sends the updated attributes (user defined) to the MME. You should wait for the confirmation to be returned (“my update attributes”) from the MME before updating locally, but will depend on how your front-end application handles attribute updates. Note the MME will send the update to all other users of interest. Also note, the “displayName” attribute associated to the user which is returned in various receiving messages (send chat, get location, …) is defined in the attributes of that user, example: attributes: { displayName: “Tom”, … }.

update action

Sending this message sends the updated action (user defined) to the MME. Note the MME will send the update to all other users of interest. Actions are non-persistent.

send chat

Sending this message sends the chat message to the MME. The MME will send the chat to the target user provided in the send chat message. Text chats are non-persistent.

get location

Sending this message sends a request to the MME to determine where within the client spatial web (metaverse) the target user is located. If the target user is somewhere in the client spatial web, the request will be returned with coordinates within the room/zone location they are in.

getClosestUsers

Sending this message sends a request to get the ‘X’ number of closest users. If the number is not specified it will default to the setting the client has set in the client settings interface. It returns an array of the ‘X’ closest users to the sending user.

getUsersByDistance

Sending this message sends a request to get users that are within a given distance from the sending user. If distance is not specified it will default to the setting the client has set in the client settings interface. It returns an array of the ‘X’ users that are within the provided distance to the sending user. Note the calculations are 3d distances (x,y,z).

Example Sending Message Processing:

```
let minimumMovement = .01;// set to allow updates in position to occur only if this minimum is achieved
let minimumRotation = 5;// set to allow updates in rotation to occur only if this minimum is achieved
function isMinimumMovement(pos){
	if ( Math.abs(pos.x - x) >= minimumMovement || Math.abs(pos.y - y) >= minimumMovement ||
Math.abs(pos.z - z) >= minimumMovement) {
		return true;
	} else { return false }
}
function isMinimumRotation(pos){
	if ( Math.abs(pos.rx - rx) >= minimumRotation || Math.abs(pos.ry - ry) >= minimumRotation ||
Math.abs(pos.rz - rz) >= minimumRotation) {
		return true;
	} else { return false }
}
let frameCounter = 0;
let fps = 30; // if rendering at 30 frames per second
function updateState(pos){
	//
	// note you need to provide an object that contains the updated position-rotation
	// you do not need to include any data that has not changed
	// for example, if the user only moves their x and y coordinates, you only need to
	// send the x and y coordinate updates
	//
	// NOTE: this updateState should only be executed periodically, not at every frame (see Introduction on Use)
	//    For example: it should sample the frames per second and send changes 4 times per second (or what is
	//        required)
	//
	//          frameCounter ++;
	//          if ( frameCounter/fps >= .25 ) {
	//                  frameCounter = 0;
	//                  … then test for movement
	//                          … then send updated movement(s)
	//
	if ( isMinimumMovement(pos) || isMinimumRotation(pos) ) {
		sendMetajesticMessage("update state",pos);
	}
}

function updateAttributes(){
	//
	// updating user defined attributes
	// User defined attributes can be anything you wish to add to a user
	// These attributes are attached to the user and all other users can see
	// these attributes. In addition, all users of interest are sent these updates.
	// Attributes can be updated using the same call – in the example below
	//    the initial value for volume is set to 22. This can be updated by
	//    sending “update attributes” with the value { volume: 44 } as the parameter
	//
```

```
        sendMetajesticMessage("update attributes", { displayName: “Tom”,
avatar:"https://www.avatar.com/files/"+myPartyId+".glb", volume: 22});
}
function getClosestUsers(){
        //
        // Get a list of users that are closest to you, irrespective of their distance
        //    from you.
        // Can be sent with a parameter defining the number of users to return
        // For example, if you send (“getClosestUsers”, {numberOfUsers:100}), it will return
        //     the 100 closest users to you. The default is set in your console, but can
        //     be overridden with this parameter.
        //
        sendMetajesticMessage("getClosestUsers");
}
function getUsersByDistance(){
        //
        // Get users that are within a certain distance from you
        // The default is set in your console, but can be overridden by sending
        //    (“getUsersByDistance”,{distance:25}) which will return all users within 25 units
        //    of the user. Note this is a 3d calculation of spatial distance.
        //
        sendMetajesticMessage("getUsersByDistance");
}
```

Message Data Schemas - Receive

The base message structure is:
Message type (receiveMessageType) and message content (msg).

Message types are covered off earlier in the document. If you are using raw websockets to connect to the MME, if follows the standard format of:

{ “event”: “messageType”, “message” : msg) in the structures as outlined below.

Message Type: “my join”

msg (data array of the users of interest):


```
{
   "data": [
      {
         "x": 4,
         "y": 5,
         "z": 9,
         "rx": 0,
         "ry": 0,
         "rz": 22,
         "rw": 23,
         "v": 0,
         "partyId": "Browser-388fe76b-717c-433f-bf08-d5552427b802"
      },
      {
         "x": 7,
         "y": 8,
         "z": 9,
         "rx": 0,
         "ry": 0,
         "rz": 24,
         "rw": 23,
         "v": 0,
         "partyId": "UE-488fe76b-717c-433f-bf08-d5552427b802"
      },  …
   ]
}
```

Message Type: “user join”

msg (single dataset of the user who just joined the room):


```
{
   "info": {
      "x": 4,
      "y": 5,
      "z": 9,
      "rx": 0,
      "ry": 0,
      "rz": 22,
      "rw": 23,
      "v": 0,
      "partyId": "Browser-388fe76b-717c-433f-bf08-d5552427b802"
   }
}
```


Message Type: “my update state”

msg (single dataset of my updated state successfully updated – changes only):


```
{
   "data": {
      "x": 5.582,
      "y": 4.9267,
      "z": 20.078,
      "rx": 79,
      "ry": 0,
      "rz": 22,
      "rw": 23,
      "v": 0,
      "partyId": "Browser-32ab395f-ba5f-4013-9be2-02562a9d3884",
   }
}
```


Message Type: “update state”

msg (single dataset of the user who just updated their state – changes only):


```
{
   "info": {
      "z": 30.078,
      "rx": 155,
      "partyId": "Browser-32ab395f-ba5f-4013-9be2-02562a9d3884"
   }
}
```

Message Type: “my update attributes”

msg (single dataset of my updated attributes successfully updated):

```
{
  "data": {
        "attributes": {
                "avatar": "https://www.avatar.com/files/Browser-14425dd6-4a9c-4568-a209-
8c85188e54c4.glb",
                "volume": 22
                },
        "partyId": "Browser-14425dd6-4a9c-4568-a209-8c85188e54c4"
        }
}
```

Message Type: “update attributes”

msg (single dataset of the updates for a user who just updated their attributes – changes only):

```
{
   "info": {
                {
                "avatar": "https://www.avatar.com/files/Browser-8dbdb7fc-009f-48f3-9e85-
3e5b556df450.glb",
                "volume": 22
                },
                "partyId": "Browser-388fe76b-717c-433f-bf08-d5552427b802"
        }
}
```

Message Type: “send action”

msg (single dataset of action of another user):

```
{
        “info”:
                {
                        "action": "roll",
                        "partyId": "Browser-8dbdb7fc-009f-48f3-9e85-3e5b556df450"
                }
}
```

Message Type: “send chat”

msg (single dataset of chat from another user):


```
{
   "type": "text",
   "fromParty": "Browser-8dbdb7fc-009f-48f3-9e85-3e5b556df450",
   "fromDisplayName": "Tom",
   "fromRoom": "Room A",
   "chatMessage": "hi"
}
```


Message Type: “get location”

msg (single dataset of the other user):


```
{
  “x”: 9,
  “y”: 16,
  “z”: 34,
  “message”: ‘Browser-8dbdb7fc-009f-48f3-9e85-3e5b556df450 is not in your room, they are in room Room A at position: {"x":9,"y":16,"z":34}',
  “partyId”: ‘Browser-8dbdb7fc-009f-48f3-9e85-3e5b556df450',
  “displayName”: 'Tom',
  “roomName”: 'Room A',
  “distance”: -1
}
```


Message Type: “message”

msg (single dataset):


```
{
   "type": "STATUS",
   "message": "Authorized and connected"
}
```

Message Type: “closest users”

msg (data array of the closest users):


```
{
   "data": [
      {
         "x": 4,
         "y": 5,
         "z": 9,
         "rx": 0,
         "ry": 0,
         "rz": 22,
         "rw": 23,
         "v": 0,
         "partyId": "Browser-388fe76b-717c-433f-bf08-d5552427b802"
      },
      {
         "x": 7,
         "y": 8,
         "z": 9,
         "rx": 0,
         "ry": 0,
         "rz": 24,
         "rw": 23,
         "v": 0,
         "partyId": "UE-488fe76b-717c-433f-bf08-d5552427b802"
      },  …
   ]
}
```

Message Type: “users by distance”

msg (data array of the users within a certain distance):


```
{
   "data": [
      {
         "x": 4,
         "y": 5,
         "z": 9,
         "rx": 0,
         "ry": 0,
         "rz": 22,
         "rw": 23,
         "v": 0,
         "partyId": "Browser-388fe76b-717c-433f-bf08-d5552427b802"
      },
      {
         "x": 7,
         "y": 8,
         "z": 9,
         "rx": 0,
         "ry": 0,
         "rz": 24,
         "rw": 23,
         "v": 0,
         "partyId": "UE-488fe76b-717c-433f-bf08-d5552427b802"
      },  …
   ]
}
```

Message Data Schemas - Send

The base message structure is:
Message type (sendMessageType) and message content (msg).

Send message types are covered off earlier in the document.

If you are using raw websockets to communicate with the MME, the send message content follows the following format:
{ “event”: “messageType”, “message” : msg) in the structures as outlined below.

If you are using the Metajestic js library, you can use the following call:

```
sendMetajesticMessage(sendMessageType,{msg});
```

Send Message Type: “update state”

msg:

```
{
    x:newX,
    y:newY,
    z:newZ,
    rx:newRx,
    …
}
```

Send Message Type: “update attributes”

msg:

```
{
    avatar:"https://www.avatar.com/files/"+myPartyId+".glb",
    volume: 22,
    displayName: “Tom”,
    …
}
```

Send Message Type: “update action”

msg:

```
{
    action: “jump”
}
```

Send Message Type: “send chat”

msg:
{
type: "text",
to: partyId,
chatMessage: “Hi”
}

Send Message Type: “get location”

msg:
{
ofPartyId: partyId
}

Send Message Type: “getClosestUsers”

msg:
{
numberOfUsers:100
}

Send Message Type: “getUsersByDistance”

msg:
{
distance:100
}

Send Message Type: “getNumberOfUsers”

msg:
{}

Information and Support

For any questions regarding this document or Metajestic Multiplayer Engine, please refer to our website at:

www.Metajestic.io

Or contact our support team at:

support@metajestic.io