亚马逊云科技

Realtime Servers Developer Guide

Amazon GameLift Servers

Version

Amazon GameLift Servers: Realtime Servers Developer Guide

Table of Contents

What is Amazon GameLift Servers Realtime?

If you're looking for a game server solution for your multiplayer game, but you don't want to expend the time and resources to develop, test, and deploy a fully custom game server, consider using Amazon GameLift Servers Realtime. Realtime servers are lightweight, ready-to-go game servers that Amazon GameLift Servers provides for you.

Amazon GameLift Servers Realtimekey features

- Full network stack for game client and server interaction
- Core game server functionality
- Customizable server logic
- Live updates to Realtime configurations and server logic
- FlexMatch matchmaking
- Flexible control of hosting resources

How to set up Realtime servers

Setting up your game to use Realtime servers involves these tasks:

- Get the default Realtime script (JavaScript) and configure it for your game. You can optionally add server-side logic.
- Deploy a fleet of hosting resources with Realtime servers configured for your game.
- Create a simple backend service that your game client can use to find or start game sessions on your Realtime servers.
- Add functionality to your game client (using provided APIs) to request a game session, connect to it, and play the game.

How Amazon GameLift Servers Realtime servers work

A Realtime server acts as a stateless relay server, where the server relays packets of messages and game data between the game clients that are connected to the game. The Realtime server doesn't evaluate messages, process data, or perform any gameplay logic. Each game client maintains its own view of the game state and provides updates to other players through the relay server. Each game client is responsible for incorporating these updates and reconciling its own game state.

You set up Realtime servers by uploading a Realtime servers script to Amazon GameLift Servers and deploying it to a managed EC2 hosting fleet. The Realtime script is a custom configuration for your game. Amazon GameLift Servers uses the script's instructions to manage the tasks of setting up and running game servers for your players.

The default Realtime script is a set of JavaScript code. You configuration it for use with your game client. Amazon GameLift Servers defines a set of server-side callbacks for Realtime scripts. Implement these callbacks to add event-driven functionality to your server. For example, you can:

- Authenticate a player when a game client tries to connect to the server.
- Validate whether a player can join a group upon request.
- Determine when to deliver messages from a certain player or to a target player, or perform additional processing in response.
- Notify all players when a player leaves a group or disconnects from the server.
- View the content of game session objects or message objects, and use the data.

You can add custom logic for game session management by building it into the Realtime script. You can write code to access server-specific objects, add event-driven logic using callbacks, or add logic based on non-event scenarios. For example, you might add game logic to build a stateful game with a server-authoritative view of the game state.

A key advantage of Amazon GameLift Servers Realtime is the ability to update your scripts at any time. When you update a script, Amazon GameLift Servers distributes the new version to all hosting resources within minutes. After Amazon GameLift Servers deploys the new script, all new game sessions created after that point will use the new script version. (Existing game sessions will continue to use the original version.)

How Realtime clients and servers interact

During a game session, game clients interact by sending messages to the Realtime server through a backend service. The backend service then relays the messages among connected game clients to exchange activity, game state, and relevant game data. If you've added custom game logic to the Realtime script, a Realtime server might implement callbacks to start event-driven responses.

Communication protocol

Realtime servers and connected game clients communicate through two channels: a TCP connection for reliable delivery, and a UDP channel for fast delivery. When creating messages, game clients choose which protocol to use depending on the nature of the message. Message delivery is set to UDP by default. If a UDP channel isn't available, Amazon GameLift Servers sends messages using TCP as a fallback.

Message content

Message content consists of two elements: a required operation code (opCode) and an optional payload. A message's opCode identifies a particular player activity or game event, and the payload provides additional data related to the operation code. Both of these elements are developer-defined. Your game client acts based on the opCodes in the messages that it receives.

Player groups

Amazon GameLift Servers Realtime provides functionality to manage groups of players. By default, Amazon GameLift Servers places all players who connect to a game session into an "all players" group. In addition, developers can set up other groups for their games, and players can be members of multiple groups simultaneously. Group members can send messages and share game data with all players in the group. For example, if your gameplay relies on teams, you can use player groups to enable communication within team members as well as game-wide.

Amazon GameLift Servers Realtime pricing and cost planning

With Amazon GameLift Servers Realtime you pay for the managed EC2 hosting resources and services you use to host Realtime servers. There's no additional charge for use of the Realtime servers themselves.

For information about Amazon GameLift Servers hosting pricing and cost planning for managed EC2 hosting, see . Pricing for Amazon GameLift Servers. For details pricing information, including example pricing scenarios and pricing tools, see Amazon GameLift Servers Pricing.

New Amazon customers can use Amazon GameLift Servers without incurring charges under the Free Tier for up to 12 months. Stay within Free Tier usage limits to avoid charges.

Preparing your games for hosting with Amazon GameLift Servers Realtime

Get your multiplayer games ready for hosting with Amazon GameLift Servers Realtime. This section guides you through the steps required to create a hosting solution for your game using Realtime servers and Amazon GameLift Servers managed EC2 hosting in the cloud.

Your solution will have the following components:

- A Realtime server that you customize to work with your game. This component is in the form of a script containing JavaScript code that provides your custom configuration and optional extensions. This script tells Amazon GameLift Servers how to set up and run Realtime servers for you. To use this script, you upload it to the Amazon GameLift Servers service.
- A version of your game client that's been integrated with the Amazon SDK and the Amazon GameLift Servers Realtime client SDK. Use these tools to add the necessary functionality to communicate with the Amazon GameLift Servers service to start or join game sessions, as well as connect to and interact with Realtime servers to play the game.

Topics

Customize an Amazon GameLift Servers Realtime script

To use Amazon GameLift Servers Realtime servers for your game, you need to provide a script (in the form of JavaScript code) to configure and optionally customize how the Amazon GameLift Servers Realtime server run and interact with your game clients. When your script is ready, upload it to the Amazon GameLift Servers service (see Upload a script for Amazon GameLift Servers Realtime servers) and use it to create a fleet of game server hosts.

Start with the default Realtime script and configure it with the following functionality.

Manage game session life-cycle (required)

At a minimum, a Realtime script must include the `Init()` function, which prepares the Realtime server to start a game session. It is also highly recommended that you also provide a way to terminate game sessions, to ensure that new game sessions can continue to be started on your fleet.

The `Init()` callback function, when called, is passed a Realtime session object, which contains an interface for the Realtime server. See Amazon GameLift Servers Realtime interface for more details on this interface.

To gracefully end a game session, the script must also call the Realtime server's `session.processEnding` function. This requires some mechanism to determine when to end a session. The script example code illustrates a simple mechanism that checks for player connections and triggers game session termination when no players have been connected to the session for a specified length of time.

Amazon GameLift Servers Realtime with the most basic configuration--server process initialization and termination--essentially act as stateless relay servers. The Realtime server relays messages and game data between game clients that are connected to the game, but takes no independent action to process data or perform logic. You can optionally add game logic, triggered by game events or other mechanisms, as needed for your game.

Add server-side game logic (optional)

You can optionally add game logic to your Realtime script. For example, you might do any or all of the following. The script example code provides illustration. See Script reference for Amazon GameLift Servers Realtime.

- **Add event-driven logic.** Implement the callback functions to respond to client-server events. See Script callbacks for Amazon GameLift Servers Realtime for a complete list of callbacks.
- **Trigger logic by sending messages to the server.** Create a set of special operation codes for messages sent from game clients to the server, and add functions to handle receipt. Use the callback `onMessage`, and parse the message content using the `gameMessage` interface (see gameMessage.opcode).
- Enable game logic to access your other Amazon resources. For details, see Communicate with other Amazon resources from your fleets.

- Allow game logic to access fleet information for the instance it is running on. For details, see Get fleet data for an Amazon GameLift Servers instance.

Node.js runtime versions

Amazon GameLift Servers Realtime servers support multiple Node.js runtime versions. You can specify the Node.js version when creating a script. The supported versions are:

- **Node.js 10.x** (Default) - Runs on Amazon Linux 2. This version is nearing end of support as Amazon Linux 2 reaches end-of-life in June 2026. See Amazon Linux 2 FAQs for more details.
- **Node.js 24.x** - Runs on Amazon Linux 2023

To specify the Node.js version and add optional customizations like install scripts, see Upload a script for Amazon GameLift Servers Realtime servers.

Amazon GameLift Servers Realtime script example

This example illustrates a basic script needed to deploy Amazon GameLift Servers Realtime plus some custom logic. It contains the required `Init()` function, and uses a timer mechanism to trigger game session termination based on length of time with no player connections. It also includes some hooks for custom logic, including some callback implementations.

```
// Example Realtime Server Script
'use strict';

// Example override configuration
const configuration = {
    pingIntervalTime: 30000,
    maxPlayers: 32
};

// Timing mechanism used to trigger end of game session. Defines how long, in
 milliseconds, between each tick in the example tick loop
const tickTime = 1000;

// Defines how to long to wait in Seconds before beginning early termination check in
 the example tick loop
const minimumElapsedTime = 120;

var session;                        // The Realtime server session object
var logger;                          // Log at appropriate level
 via .info(), .warn(), .error(), .debug()
var startTime;                       // Records the time the process started
var activePlayers = 0;               // Records the number of connected players
var onProcessStartedCalled = false; // Record if onProcessStarted has been called

// Example custom op codes for user-defined messages
// Any positive op code number can be defined here. These should match your client
 code.
const OP_CODE_CUSTOM_OP1 = 111;
const OP_CODE_CUSTOM_OP1_REPLY = 112;
const OP_CODE_PLAYER_ACCEPTED = 113;
const OP_CODE_DISCONNECT_NOTIFICATION = 114;

// Example groups for user-defined groups
// Any positive group number can be defined here. These should match your client code.
// When referring to user-defined groups, "-1" represents all groups, "0" is reserved.
const RED_TEAM_GROUP = 1;
const BLUE_TEAM_GROUP = 2;

// Called when game server is initialized, passed server's object of current session
function init(rtSession) {
    session = rtSession;
    logger = session.getLogger();
}

// On Process Started is called when the process has begun and we need to perform any
// bootstrapping.  This is where the developer should insert any code to prepare
// the process to be able to host a game session, for example load some settings or set
 state
//
// Return true if the process has been appropriately prepared and it is okay to invoke
 the
// GameLift ProcessReady() call.
function onProcessStarted(args) {
    onProcessStartedCalled = true;
    logger.info("Starting process with args: " + args);
    logger.info("Ready to host games...");

    return true;
}

// Called when a new game session is started on the process
function onStartGameSession(gameSession) {
    // Complete any game session set-up

    // Set up an example tick loop to perform server initiated actions
    startTime = getTimeInS();
    tickLoop();
}

// Handle process termination if the process is being terminated by Amazon GameLift
 Servers
// You do not need to call ProcessEnding here
function onProcessTerminate() {
    // Perform any clean up
}

// Return true if the process is healthy
function onHealthCheck() {
    return true;
}

// On Player Connect is called when a player has passed initial validation
// Return true if player should connect, false to reject
function onPlayerConnect(connectMsg) {
    // Perform any validation needed for connectMsg.payload, connectMsg.peerId
    return true;
}

// Called when a Player is accepted into the game
function onPlayerAccepted(player) {
    // This player was accepted -- let's send them a message
    const msg = session.newTextGameMessage(OP_CODE_PLAYER_ACCEPTED, player.peerId,
                                             "Peer " + player.peerId + " accepted");
    session.sendReliableMessage(msg, player.peerId);
    activePlayers++;
}

// On Player Disconnect is called when a player has left or been forcibly terminated
// Is only called for players that actually connected to the server and not those
 rejected by validation
// This is called before the player is removed from the player list
function onPlayerDisconnect(peerId) {
    // send a message to each remaining player letting them know about the disconnect
    const outMessage = session.newTextGameMessage(OP_CODE_DISCONNECT_NOTIFICATION,
                                                session.getServerId(),
                                                "Peer " + peerId + " disconnected");
    session.getPlayers().forEach((player, playerId) => {
        if (playerId != peerId) {
            session.sendReliableMessage(outMessage, playerId);
        }
    });
    activePlayers--;
}

// Handle a message to the server
function onMessage(gameMessage) {
    switch (gameMessage.opCode) {
      case OP_CODE_CUSTOM_OP1: {
        // do operation 1 with gameMessage.payload for example sendToGroup
        const outMessage = session.newTextGameMessage(OP_CODE_CUSTOM_OP1_REPLY,
 session.getServerId(), gameMessage.payload);
        session.sendGroupMessage(outMessage, RED_TEAM_GROUP);
        break;
      }
    }
}

// Return true if the send should be allowed
function onSendToPlayer(gameMessage) {
    // This example rejects any payloads containing "Reject"
    return (!gameMessage.getPayloadAsText().includes("Reject"));
}

// Return true if the send to group should be allowed
// Use gameMessage.getPayloadAsText() to get the message contents
function onSendToGroup(gameMessage) {
    return true;
}

// Return true if the player is allowed to join the group
function onPlayerJoinGroup(groupId, peerId) {
    return true;
}

// Return true if the player is allowed to leave the group
function onPlayerLeaveGroup(groupId, peerId) {
    return true;
}

// A simple tick loop example
// Checks to see if a minimum amount of time has passed before seeing if the game has
 ended
async function tickLoop() {
    const elapsedTime = getTimeInS() - startTime;
    logger.info("Tick... " + elapsedTime + " activePlayers: " + activePlayers);

    // In Tick loop - see if all players have left early after a minimum period of time
 has passed
    // Call processEnding() to terminate the process and quit
    if ( (activePlayers == 0) && (elapsedTime > minimumElapsedTime)) {
        logger.info("All players disconnected. Ending game");
        const outcome = await session.processEnding();
        logger.info("Completed process ending with: " + outcome);
        process.exit(0);
    }
    else {
        setTimeout(tickLoop, tickTime);
    }
}

// Calculates the current time in seconds
function getTimeInS() {
    return Math.round(new Date().getTime()/1000);
}

exports.ssExports = {
    configuration: configuration,
    init: init,
    onProcessStarted: onProcessStarted,
    onMessage: onMessage,
    onPlayerConnect: onPlayerConnect,
    onPlayerAccepted: onPlayerAccepted,
    onPlayerDisconnect: onPlayerDisconnect,
    onSendToPlayer: onSendToPlayer,
    onSendToGroup: onSendToGroup,
    onPlayerJoinGroup: onPlayerJoinGroup,
    onPlayerLeaveGroup: onPlayerLeaveGroup,
    onStartGameSession: onStartGameSession,
    onProcessTerminate: onProcessTerminate,
    onHealthCheck: onHealthCheck
};
```

Integrate a game client for Amazon GameLift Servers Realtime

When using Realtime servers, you need to add functionality to your game client so that players can join and participate in game sessions hosted with Amazon GameLift Servers Realtime.

There are two sets of tasks needed to prepare your game client:

- Set up your game client to acquire information about existing games, request matchmaking, start new game sessions, and reserve game session slots for a player.
- Enable your game client to join a game session hosted on a Realtime server and exchange messages.

Find or create game sessions and player sessions

Set up your game client to find or start game sessions, request FlexMatch matchmaking, and reserve space for players in a game by creating player sessions.

Important

As a best practice, we highly recommend that your create a backend service to make all direct requests to the Amazon GameLift Servers service. Set up game client actions that initiate the backent service to make requests and then relay relevant responses back to the game client. For more information about setting up a backend service, see Design your backend service for Amazon GameLift Servers.

1. Add the Amazon SDK to your game client, initialize an Amazon GameLift Servers client, and configure it to use the hosting resources in your fleets and queues. The Amazon SDK is available in several languages; see the Amazon GameLift Servers SDKs Amazon GameLift Servers development tools for game clients.
2. Add Amazon GameLift Servers functionality to your backend service. For more detailed instructions, see Integrate your game client for Amazon GameLift Servers and (optionally) Adding FlexMatch matchmaking.

   Although not required, it is a best practice to use game session placement queues to create new game sessions. This method lets you take full advantage of Amazon GameLift Servers to choose the best possible locations to host new game sessions, based on a variety of factors, including player latency data. At a minimum, your backend service must use one of the available methods to request new game sessions and then handle game session data in response. You may also want to add optional functionality, such as to search for information on existing game sessions or to reserve player slots in a game session by creating individual player sessions.

3. Convey connection information back to the game client. The backend service receives game session and player session objects in response to requests to the Amazon GameLift Servers service. These objects contain information, in particular connection details (IP address and port) and player session ID, that the game client needs to connect to the game session running on a Realtime Server.

Connect to games on Amazon GameLift Servers Realtime

Enable your game client to connect directly with a hosted game session on a Realtime server and exchange messages with the server and with other players.

1. Get the client SDK for Amazon GameLift Servers Realtime, build it, and add it to your game client project. See the README file for more information on SDK requirements and instructions on how to build the client libraries.
2. Call Client() with a client configuration that specifies the type of client/server connection to use.
3. Add the following functionality to your game client. See the Amazon GameLift Servers Realtime client API (C#) reference for more information.
   - Connect to and disconnect from a game
     - Connect()
     - Disconnect()
   - Send messages to target recipients
     - SendMessage()
   - Receive and process messages
     - OnDataReceived()
   - Join groups and leave player groups
     - JoinGroup()
     - RequestGroupMembership()
     - LeaveGroup()
4. Set up event handlers for the client callbacks as needed. See Amazon GameLift Servers Realtime client API (C#) reference: Asynchronous callbacks.

Game client examples

Basic realtime client (C#)

This example illustrates a basic game client integration with the client SDK (C#) for Amazon GameLift Servers Realtime. As shown, the example initializes a Realtime client object, sets up event handlers and implements the client-side callbacks, connects to a Realtime server, sends a message, and disconnects.

```
using System;
using System.Text;
using Aws.GameLift.Realtime;
using Aws.GameLift.Realtime.Event;
using Aws.GameLift.Realtime.Types;

namespace Example
{
    /**
     * An example client that wraps the client SDK for Amazon GameLift Servers Realtime
     *
     * You can redirect logging from the SDK by setting up the LogHandler as such:
     * ClientLogger.LogHandler = (x) => Console.WriteLine(x);
     *
     */
    class RealTimeClient
    {
        public Aws.GameLift.Realtime.Client Client { get; private set; }

        // An opcode defined by client and your server script that represents a custom
 message type
        private const int MY_TEST_OP_CODE = 10;

        /// Initialize a client for Amazon GameLift Servers Realtime and connect to a
 player session.
        /// <param name="endpoint">The DNS name that is assigned to Realtime server</
param>
        /// <param name="remoteTcpPort">A TCP port for the Realtime server</param>
        /// <param name="listeningUdpPort">A local port for listening to UDP traffic</
param>
        /// <param name="connectionType">Type of connection to establish between client
 and the Realtime server</param>
        /// <param name="playerSessionId">The player session ID that is assigned to the
 game client for a game session </param>
```

```
        /// <param name="connectionPayload">Developer-defined data to be used during
client connection, such as for player authentication</param>
       public RealTimeClient(string endpoint, int remoteTcpPort, int listeningUdpPort,
ConnectionType connectionType,
                   string playerSessionId, byte[] connectionPayload)
        {
            // Create a client configuration to specify a secure or unsecure connection
type
            // Best practice is to set up a secure connection using the connection type
RT_OVER_WSS_DTLS_TLS12.
            ClientConfiguration clientConfiguration = new ClientConfiguration()
      {
                // C# notation to set the field ConnectionType in the new instance of
ClientConfiguration
          ConnectionType = connectionType
      };

            // Create a Realtime client with the client configuration
            Client = new Client(clientConfiguration);

            // Initialize event handlers for the Realtime client
            Client.ConnectionOpen += OnOpenEvent;
            Client.ConnectionClose += OnCloseEvent;
            Client.GroupMembershipUpdated += OnGroupMembershipUpdate;
            Client.DataReceived += OnDataReceived;

            // Create a connection token to authenticate the client with the Realtime
server
            // Player session IDs can be retrieved using Amazon SDK for Amazon GameLift
Servers
            ConnectionToken connectionToken = new ConnectionToken(playerSessionId,
connectionPayload);

            // Initiate a connection with the Realtime server with the given connection
information
            Client.Connect(endpoint, remoteTcpPort, listeningUdpPort, connectionToken);
        }

        public void Disconnect()
        {
            if (Client.Connected)
            {
                Client.Disconnect();
            }
        }

        public bool IsConnected()
        {
            return Client.Connected;
        }

        /// <summary>
        /// Example of sending to a custom message to the server.
        ///
        /// Server could be replaced by known peer Id etc.
        /// </summary>
        /// <param name="intent">Choice of delivery intent i.e. Reliable, Fast etc. </
param>
        /// <param name="payload">Custom payload to send with message</param>
        public void SendMessage(DeliveryIntent intent, string payload)
        {
            Client.SendMessage(Client.NewMessage(MY_TEST_OP_CODE)
                .WithDeliveryIntent(intent)
                .WithTargetPlayer(Constants.PLAYER_ID_SERVER)
                .WithPayload(StringToBytes(payload)));
        }

        /**
         * Handle connection open events
         */
        public void OnOpenEvent(object sender, EventArgs e)
        {
        }

        /**
         * Handle connection close events
         */
        public void OnCloseEvent(object sender, EventArgs e)
        {
        }

        /**
         * Handle Group membership update events
         */
        public void OnGroupMembershipUpdate(object sender, GroupMembershipEventArgs e)
        {
        }
        /**
         *  Handle data received from the Realtime server
         */
        public virtual void OnDataReceived(object sender, DataReceivedEventArgs e)
        {
            switch (e.OpCode)
            {
                // handle message based on OpCode
                default:
                    break;
            }
        }

        /**
         * Helper method to simplify task of sending/receiving payloads.
         */
        public static byte[] StringToBytes(string str)
        {
            return Encoding.UTF8.GetBytes(str);
        }

        /**
         * Helper method to simplify task of sending/receiving payloads.
         */
        public static string BytesToString(byte[] bytes)
        {
            return Encoding.UTF8.GetString(bytes);
        }
    }
}
```

Upload a script for Amazon GameLift Servers Realtime servers

When you're ready to deploy Amazon GameLift Servers Realtime for your game, upload completed Realtime server script files to Amazon GameLift Servers. Do this by creating an Amazon GameLift Servers script resource and specifying the location of your script files. You can also update server script files that are already deployed by uploading new files for an existing script resource.

When creating a script, you can choose the Node.js runtime version. Node.js 24.x runs on Amazon Linux 2023, while Node.js 10.x uses Amazon Linux 2. Amazon GameLift Servers automatically manages Node.js minor and patch version updates for new fleets. The runtime will remain unchanged for a fleet once it is created. If you would like to pick a specific minor and patch version of Node.js you can install it using an install script. See Add an install script (optional).

When you create a new script resource, Amazon GameLift Servers assigns it a unique script ID (for example, `script-1111aaaa-22bb-33cc-44dd-5555eeee66ff`) and uploads a copy of the script files. Upload time depends on the size of your script files and on your connection speed.

After you create the script resource, Amazon GameLift Servers deploys the script with a new Amazon GameLift Servers Realtime fleet. Amazon GameLift Servers installs your server script onto each instance in the fleet, placing the script files in `/local/game`.

To troubleshoot fleet activation problems related to the server script, see Debug managed EC2 fleets for Amazon GameLift Servers Realtime.

Package script files

Your server script can include one or more files combined into a single .zip file for uploading. The .zip file must contain all files that your script needs to run.

You can store your zipped script files in either a local file directory or in an Amazon Simple Storage Service (Amazon S3) bucket.

Add an install script (optional)

Similar to Add a build install script for an Amazon GameLift Servers build, you can optionally add an install script to customize the runtime environment for your Realtime script. This feature is only available for scripts using Node.js version 24.x and later.

To add an install script, include a file named `install.sh` in the root directory of your script .zip file. Amazon GameLift Servers will automatically detect and execute this script during fleet deployment.

Note

The install script only executes during fleet creation. If you update an existing script using UpdateScript, the install script will not be executed on fleets that are already running. To apply install script changes, you must create a new fleet with the updated script.

Example Install script example

This example `install.sh` file installs Node.js 24.10.0 to replace the default Node.js 24.x runtime:

```
#!/bin/bash

# Remove existing Node.js installation
sudo rm -rf /local/NodeJS/*

# Install Node.js v24.10.0
curl -L https://nodejs.org/dist/v24.10.0/node-v24.10.0-linux-x64.tar.xz | sudo tar -xJ
 -C /local/NodeJS --strip-components=1
```

Upload script files from a local directory

If you have your script files stored locally, you can upload them to Amazon GameLift Servers from there. To create the script resource, use either the Amazon GameLift Servers console or the Amazon Command Line Interface (Amazon CLI).

Amazon GameLift Servers console

To create a script resource

1. Open the Amazon GameLift Servers console.
2. In the navigation pane, choose **Hosting**, **Scripts**.
3. On the **Scripts** page, choose **Create script**.
4. On the **Create script** page, under **Script settings,** do the following:
   a. For **Name**, enter a script name.
   b. (Optional) For **Version**, enter version information. Because you can update the content of a script, version data can be helpful in tracking updates.
   c. For **Node.js version**, choose the runtime version of Node.js the server will run on.
   d. For **Script source**, choose **Upload a .zip file**.
   e. For **Script files**, choose **Choose file**, browse for the .zip file that contains your script, and then choose that file.
5. (Optional) Under **Tags**, add tags to the script by entering **Key** and **Value** pairs.
6. Choose **Create**.

Amazon GameLift Servers assigns an ID to the new script and uploads the designated .zip file. You can view the new script, including its status, on the **Scripts** page.

Amazon CLI

Use the `create-script` Amazon CLI command to define the new script and upload your server script files.

To create a script resource

1. Place the .zip file into a directory where you can use the Amazon CLI.
2. Open a command line window and switch to the directory where you placed the .zip file.
3. Enter the following **create-script** command and parameters. For the **--zip-file** parameter, be sure to add the string `fileb://` to the name of the .zip file. It identifies the file as binary so that Amazon GameLift Servers processes the compressed content.

```
aws gamelift create-script \
    --name user-defined name of script \
    --script-version user-defined version info \
    --zip-file fileb://name of zip file \
    --node-js-version 10.x or 24.x \
    --region region name
```

Example

```
aws gamelift create-script \
    --name "My_Realtime_Server_Script_1" \
    --script-version "1.0.0" \
    --zip-file fileb://myrealtime_script_1.0.0.zip \
    --node-js-version 24.x \
    --region us-west-2
```

In response to your request, Amazon GameLift Servers returns the new script object.

4. To view the new script, call `describe-script`.

Upload script files from Amazon S3

You can store your script files in an Amazon S3 bucket and upload them to Amazon GameLift Servers from there. When you create your script, you specify the S3 bucket location and Amazon GameLift Servers retrieves your script files from Amazon S3.

To create a script resource

1. **Store your script files in an S3 bucket.** Create a .zip file containing your server script files and upload it to an S3 bucket in an Amazon Web Services account that you control. Take note of the object URI—you need this when creating an Amazon GameLift Servers script.

   Note

   Amazon GameLift Servers doesn't support uploading from S3 buckets with names that contain a period (.).

2. **Give Amazon GameLift Servers access to your script files.** To create an Amazon Identity and Access Management (IAM) role that allows Amazon GameLift Servers to access the S3 bucket containing your server script, follow the instructions in Set up an IAM service role for Amazon GameLift Servers. After you create the new role, take note of its name, which you need when creating a script.
3. **Create a script.** Use the Amazon GameLift Servers console or the Amazon CLI to create a new script record. To make this request, you must have the IAM `PassRole` permission, as described in IAM permission examples for Amazon GameLift Servers.

Console

1. In the Amazon GameLift Servers console, in the navigation pane, choose **Hosting**, **Scripts**.
2. On the **Scripts** page, choose **Create script**.
3. On the **Create script** page, under **Script settings**, do the following:

   a. For **Name**, enter a script name.

   b. (Optional) For **Version**, enter version information. Because you can update the content of a script, version data can be helpful in tracking updates.

   c. For **Node.js version**, choose the runtime version of Node.js the server will run on.

   d. For **Script source**, choose **Amazon S3 URI**.

   e. Enter the **S3 URI** of the script object that you uploaded to Amazon S3, and then choose the **Object version**. If you don't remember the Amazon S3 URI and object version, choose **Browse S3**, and then search for the script object.

4. (Optional) Under **Tags**, add tags to the script by entering **Key** and **Value** pairs.
5. Choose **Create**.

   Amazon GameLift Servers assigns an ID to the new script and uploads the designated .zip file. You can view the new script, including its status, on the **Scripts** page.

Amazon CLI

Use the `create-script` Amazon CLI command to define the new script and upload your server script files.

1. Open a command line window and switch to a directory where you can use the Amazon CLI.
2. Enter the following **create-script** command and parameters. The **--storage-location** parameter specifies the Amazon S3 bucket location of your script files.

```
aws gamelift create-script \
    --name [user-defined name of script] \
    --script-version [user-defined version info] \
    --storage-location "Bucket"=S3 bucket name,"Key"=name of zip file in S3
 bucket,"RoleArn"=Access role ARN \
    --node-js-version 10.x or 24.x \
    --region region name
```

   Example

```
aws gamelift create-script \
    --name "My_Realtime_Server_Script_1" \
    --script-version "1.0.0" \
    --storage-location "Bucket"="gamelift-
script","Key"="myrealtime_script_1.0.0.zip","RoleArn"="arn:aws:iam::123456789012:role/
S3Access" \
    --node-js-version 24.x \
    --region us-west-2
```

   In response to your request, Amazon GameLift Servers returns the new script object.

3. To view the new script, call `describe-script`.

Update an Amazon GameLift Servers Realtime script

You can update the metadata for a script resource using either the Amazon GameLift Servers console or the `update-script` Amazon CLI command.

You can also update the script content for a script resource. Amazon GameLift Servers deploys script content to all fleet instances that use the updated script resource. When the updated script is deployed, instances use it when starting new game sessions. Game sessions that are already running at the time of the update don't use the updated script.

To update script files

- For script files stored locally, to upload the updated script .zip file, use either the Amazon GameLift Servers console or the **update-script** command.
- For script files stored in an Amazon S3 bucket, upload the updated script files to the S3 bucket. Amazon GameLift Servers periodically checks for updated script files and retrieves them directly from the S3 bucket.

Creating a managed EC2 fleet for Amazon GameLift Servers Realtime

This section guides you through how to create an Amazon GameLift Servers managed EC2 fleet for Realtime servers. Managed fleets use Amazon Elastic Compute Cloud (Amazon EC2) compute instances that are optimized for multiplayer game hosting. You can create a fleet that deploys Realtime servers globally, in any Amazon Web Services Region or Local Zone that Amazon GameLift Servers supports. For a list of supported locations, see Amazon GameLift Servers service locations. You can set up multi-location fleets for your Realtime servers.

When you create a managed EC2 fleet, the fleet creation process starts immediately. This process involves several phases as Amazon GameLift Servers prepares your Realtime servers based on your configuration script, provisions EC2 instances and deploys the server code, and prepares to host game servers on each instance. You can monitor a fleet's status in the console or using the Amazon Command Line Interface (Amazon CLI). A fleet is ready to host game sessions when its status reaches `ACTIVE`. For more guidance about creating a managed EC2 fleet, , see the following topics in the *Amazon GameLift Servers Hosting Guide*:

- Amazon GameLift Servers managed EC2 fleets
- Customize your Amazon GameLift Servers managed EC2 fleets
- Debug Amazon GameLift Servers fleet issues

Topics

Create a hosting fleet for Amazon GameLift Servers Realtime

When you're ready to deploy Realtime servers for your game, create an Amazon GameLift Servers managed EC2 fleet. This topic assumes you've completed the following steps:

- Created a Realtime script that's configured for your game. See Customize an Amazon GameLift Servers Realtime script.
- Uploaded your Realtime script to Amazon GameLift Servers. See Upload a script for Amazon GameLift Servers Realtime servers.

To create a managed EC2 fleet

Use either the Amazon GameLift Servers console or the Amazon Command Line Interface (Amazon CLI) to create a managed EC2 fleet. If you're exploring the Realtime servers feature or want to get your servers up and running fast, follow the guidance for a minimal fleet configuration.

Console

To create a managed EC2 fleet

1. In the Amazon GameLift Servers console, in the navigation pane, choose **Fleets**.
2. On the **Fleets** page, choose **Create fleet**.
3. Choose **Managed EC2**.
4. On the **Fleet details** page do the following:
   a. For **Name**, enter a fleet name. We recommend including the fleet type (Spot or On-demand) in your fleet names. This makes it much easier to identify fleet types when viewing a list of fleets.
   b. For **Description**, provide a short description of the fleet.
   c. For **Binary type**, select **Script** to identify the type of game server to deploy to this fleet.
   d. Select a **Script** from the dropdown list of previously uploaded Realtime scripts.
5. (Optional) Under **Additional details** for the following:
   a. For **Instance role**, specify an IAM role that authorizes applications in your game build to access other Amazon resources in your account. For more information, see Communicate with other Amazon resources from your fleets. To create a fleet with an instance role, your account must have the IAM `PassRole` permission. For more information, see IAM permission examples for Amazon GameLift Servers.

      You can't update these settings after fleet creation.
   b. For **Metric group**, Enter the name of a new or existing fleet metric group. You can aggregate the metrics for multiple fleets by adding them to the same metric group.

      You can't update the metric group after fleet creation.
6. Choose **Next**.

7. On the **Select locations** page, select one or more additional remote locations to deploy instances to. The home Region is automatically selected based on the Region you are accessing the console from. If you select additional locations, fleet instances are also deployed in these locations.

   Important

   To use Regions that aren't enabled by default, enable them in your Amazon Web Services account.

   - Fleets with Regions that aren't enabled that you created before February 28, 2022 are not affected by this requirement.
   - To create new multi-location fleets or to update existing multi-location fleets, first enable any Regions or Local Zones that you choose to use.

   For more information about Regions that aren't enabled by default and how to enable them, see Managing Amazon Web Services Regions in the *Amazon Web Services General Reference*. See Getting started with Local Zones in the *Amazon Local Zones User Guide*.

8. Choose **Next**.
9. On the **Define instance details page**, choose

   a. **On-demand** or **Spot** instances for this fleet. For more information about fleet types, see On-Demand Instances versus Spot Instances.

   b. From the **Filter architecture** menu choose **x64** or **Arm**.

      Note

      Graviton Arm instances require a server build for a Linux AMI. Server SDK 5.1.1 or newer is required for C++ and C#. Server SDK 5.0 or newer is required for Go. These instances do not provide out-of-the-box support for Mono installation on Amazon Linux 2023 (AL2023) or Amazon Linux 2 (AL2).

      For information on Amazon EC2 Arm architectures, see Amazon Graviton Processor and Amazon EC2 instance types.

For information on the instance types supported by Amazon GameLift Servers, see the `EC2InstanceType` values under CreateFleet() request parameters.

10. Select an Amazon EC2 **Instance type** from the list. For more information about choosing an instance type, see Instance types.

    After you create the fleet, you can't change the instance type.

11. Choose **Next**.

12. On the **Configure runtime** page, under **Runtime configuration** do the following:

    a. For **Launch path**, enter the path to the server script. Example: **`MyRealtimeLaunchScript.js`**.

    b. For **Concurrent processes**, choose the number of server processes to run concurrently on each instance in the fleet. Review the Amazon GameLift Servers limits on number of concurrent server processes.

       Limits on concurrent server processes per instance apply to the total of concurrent processes for all configurations. If you configure the fleet to exceed the limit, the fleet can't activate.

13. Under **Game session activation**, provide limits for activating new game sessions on the instances in this fleet:

    a. For **Max concurrent game session activation**, enter the number of game sessions on an instance that activate at the same time. This limit is useful when launching multiple new game sessions may have an impact on the performance of other game sessions running on the instance.

    b. For **New activation timeout**, enter how long to wait for a session to activate. If the game session doesn't move to `ACTIVE` status before the timeout, Amazon GameLift Servers terminates the game session activation.

14. (Optional) Under **EC2 port settings**, do the following:

    a. Choose **Add port setting** to define access permissions for inbound traffic connecting to the server process deployed on the fleet.

    b. For **Type**, choose **Custom TCP** or **Custom UDP**.

   c. For **Port range**, Enter a range of port numbers that allow inbound connections. A port range must use the format `nnnnn[-nnnnn]`, with values between 1026 and 60000. Example: **`1500`** or **`1500-20000`**.

   d. For **IP address range**, Enter a range of IP addresses. Use CIDR notation. Example: **`0.0.0.0/0`** (This example allows access to anyone trying to connect.)

15. (Optional) Under **Game session resource settings** do the following:

   a. For **Game scaling protection policy**, Turn on or off scaling protection. Amazon GameLift Servers won't terminate instance with protection during a scale down event if they're hosting an active game session.

   b. For **Resource creation limit**, enter a maximum number of game sessions a player can create during the policy period.

16. Choose **Next**.

17. (Optional) Add tags to the build by entering **Key** and **Value** pairs. Choose **Next** to continue to fleet creation review.

18. Choose **Create**. Amazon GameLift Servers assigns an ID to the new fleet and begins the fleet activation process. You can track the new fleet's status on the **Fleets** page.

You can update the fleet's metadata and configuration at any time, regardless of fleet status. For more information, see Update an Amazon GameLift Servers fleet configuration. You can update fleet capacity after the fleet has reached ACTIVE status. For more information, see Scaling game hosting capacity with Amazon GameLift Servers.

You can also add or remove remote locations.

Amazon CLI

Use the `create-fleet` command to create a fleet of compute type `EC2`. Amazon GameLift Servers creates the fleet resource in your current default Amazon Web Services Region (or you can add a --region tag to specify a different Amazon Web Services Region).

Create a minimal managed fleet

The following example request creates a new fleet with the minimal settings that are required to deploy a fleet with running game servers that game clients can connect to. The new fleet has these characteristics:

- It specifies a Realtime server script, which is uploaded to Amazon GameLift Servers and in READY status.
- Is uses c5.large On-Demand Instances.
- It sets the fleet's home Amazon Web Services Region to `us-west-2` and deploys instances to that Region only.
- Based on the runtime configuration, each instance in the fleet runs one game server process, which means that each instance can host only one game session at a time. By default, game session activation timeout is set to 300 seconds, with no limit on the number of concurrent activations.
- Players can connect to game servers using port 33435.
- All other features are either turned off or use default settings.

```
aws gamelift create-fleet \
    --name MinimalRealtimeFleet123 \
    --description "A basic test fleet" \
    --region us-west-2 \
    --ec2-instance-type c5.large \
    --fleet-type ON_DEMAND \
    --script-id script-1111aaaa-22bb-33cc-44dd-5555eeee66ff \
    --runtime-configuration "ServerProcesses=[{LaunchPath=/local/game/
megafrograce.js, ConcurrentExecutions=1}]" \
    --ec2-inbound-permissions
 "FromPort=33435,ToPort=33435,IpRange=0.0.0.0/0,Protocol=UDP"
```

Create a fully configured managed fleet

The following example request creates a production fleet with settings for all optional features. The new fleet has these characteristics:

- It specifies a Realtime server script, which has been uploaded to Amazon GameLift Servers and is in READY status.
- It uses c5.large On-Demand Instances with the operating system that matches the selected game build.
- It sets the fleet's home Amazon Web Services Region to `us-west-2` and deploys instances to the home Region and one remote location `sa-east-1`.
- Based on the runtime configuration:

- Each compute in the fleet runs 10 game server processes with the same launch parameters, which means that each compute can host up to 10 game sessions simultaneously.
- On each compute, only two game sessions can be activating at the same time. Activating game sessions must be ready to host players within 300 seconds (5 minutes) or be terminated.
- Players can connect to game servers using a port in the following range `33435 to 33535`.
- All game sessions in the fleet have game session protection turned on.
- Individual players are limited to creating three new game sessions within a 15-minute period.
- Metrics for this fleet are included in the metric group `AMERfleets`, which (for this example) aggregates metrics for a group of fleets in North, Central, and South America.

```
aws gamelift create-fleet \
    --name ProdRealtimeFleet123 \
    --description "A fully configured prod fleet" \
    --ec2-instance-type c5.large \
    --region us-west-2 \
    --locations "Location=sa-east-1" \
    --fleet-type ON_DEMAND \
    --script-id script-1111aaaa-22bb-33cc-44dd-5555eeee66ff \
    --certificate-configuration "CertificateType=GENERATED" \
    --runtime-configuration "GameSessionActivationTimeoutSeconds=300,
 MaxConcurrentGameSessionActivations=2, ServerProcesses=[{LaunchPath=/local/game/
megafrograce.js, ConcurrentExecutions=10}]" \
    --new-game-session-protection-policy "FullProtection" \
    --resource-creation-limit-policy "NewGameSessionsPerCreator=3,
 PolicyPeriodInMinutes=15" \
    --ec2-inbound-permissions
 "FromPort=33435,ToPort=33535,IpRange=0.0.0.0/0,Protocol=UDP" \
    --metric-groups  "AMERfleets"
```

If the create-fleet request is successful, Amazon GameLift Servers returns a set of fleet attributes that includes the configuration settings you requested and a new fleet ID. Amazon GameLift Servers then initiates the fleet activation process and sets the fleet status and the location statuses to **New**. You can track the fleet's status and view other fleet information using these CLI commands:

- describe-fleet-events
- describe-fleet-attributes

- describe-fleet-capacity
- describe-fleet-port-settings
- describe-fleet-utilization
- describe-runtime-configuration
- describe-fleet-location-attributes
- describe-fleet-location-capacity
- describe-fleet-location-utilization

You can change the fleet's capacity and other configuration settings as needed using these commands:

- update-fleet-attributes
- update-fleet-capacity
- update-fleet-port-settings
- update-runtime-configuration
- create-fleet-locations
- delete-fleet-locations

Debug managed EC2 fleets for Amazon GameLift Servers Realtime

This topic provides guidance on how to resolve issues with your Amazon GameLift Servers managed EC2 fleets for Realtime servers. For general troubleshooting help with managed EC2 fleets, see also Debug Amazon GameLift Servers fleet issues .

Zombie game sessions: They start and run a game, but they never end.

You might observe this issues as any of the following scenarios:

- Script updates are not picked up by the fleet's Realtime servers.
- The fleet quickly reaches maximum capacity and does not scale down when player activity (such as new game session requests) decreases.

This is almost certainly a result of failing to successfully call `processEnding` in your Realtime script. Although the fleet goes active and game sessions are started, there is no method for stopping them. As a result, the Realtime server that is running the game session is never freed up to start a new one, and new game sessions can only start when new Realtime servers are spun up. In addition, updates to the Realtime script do not impact already- running game sessions, only ones.

To prevent this from happening, scripts need to provide a mechanism to trigger a `processEnding` call. As illustrated in the Amazon GameLift Servers Realtime script example, one way is to program an idle session timeout where, if no player is connected for a certain amount of time, the script will end the current game session.

However, if you do fall into this scenario, there are a couple workarounds to get your Realtime servers unstuck. The trick is to trigger the Realtime server processes—or the underlying fleet instances—to restart. In this event, Amazon GameLift Servers automatically closes the game sessions for you. Once Realtime servers are freed up, they can start new game sessions using the latest version of the Realtime script.

There are a couple of methods to achieve this, depending on how pervasive the problem is:

- Scale the entire fleet down. This method is the simplest to do but has a widespread effect. Scale the fleet down to zero instances, wait for the fleet to fully scale down, and then scale it back up. This will wipe out all existing game sessions, and let you start fresh with the most recently updated Realtime script.
- Remotely access the instance and restart the process. This is a good option if you have only a few processes to fix. If you are already logged onto the instance, such as to tail logs or debug, then this may be the quickest method. See Remotely connect to Amazon GameLift Servers fleet instances.

If you opt not to include way to call `processEnding` in your Realtime script, there are a couple of tricky situations that might occur even when the fleet goes active and game sessions are started. First, a running game session does not end. As a result, the server process that is running that game session is never free to start a new game session. Second, the Realtime server does not pick up any script updates.

Logging server messages for Amazon GameLift Servers Realtime

You can capture a variety of custom server messages from your Amazon GameLift Servers Realtime servers and save them to game session log files. When a game session ends, Amazon GameLift Servers automatically uploads the session logs to Amazon Simple Storage Service (Amazon S3). This topic provides instructions on how to add log messaging to your server scripts and then access game session logs from Amazon S3 storage. You can also configure logging levels to manage the volume of log messages that your servers generate.

Important

There is a limit on the size of a log file per game session (see Amazon GameLift Servers endpoints and quotas in the *Amazon Web Services General Reference*). When a game session ends, Amazon GameLift Servers uploads the server logs to Amazon Simple Storage Service (Amazon S3). Amazon GameLift Servers will not upload logs that exceed the limit. Logs can grow very quickly and exceed the size limit. You should monitor your logs and limit the log output to necessary messages only.

Logging messages in your server script

You can output custom messages in the script for your Amazon GameLift Servers Realtime. Use the following steps to send server messages to a log file:

1. Create a variable to hold the reference to the logger object.

```
var logger;
```

2. In the `init()` function, get the logger from the session object and assign it to your logger variable.

```
function init(rtSession) {
    session = rtSession;
    logger = session.getLogger();
}
```

3. Call the appropriate function on the logger to output a message.

   Debug messages

   ```
   logger.debug("This is my debug message...");
   ```

   Informational messages

   ```
   logger.info("This is my info message...");
   ```

   Warning messages

   ```
   logger.warn("This is my warn message...");
   ```

   Error messages

   ```
   logger.error("This is my error message...");
   ```

   Fatal error messages

   ```
   logger.fatal("This is my fatal error message...");
   ```

   Customer experience fatal error messages

   ```
   logger.cxfatal("This is my customer experience fatal error message...");
   ```

For an example of the logging statements in a script, see Amazon GameLift Servers Realtime script example.

The output in the log files indicates the type of message (`DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`, `CXFATAL`), as shown in the following lines from an example log:

```

09 Sep 2021 11:46:32,970 [INFO] (gamelift.js) 215: Calling GameLiftServerAPI.InitSDK...
09 Sep 2021 11:46:32,993 [INFO] (gamelift.js) 220: GameLiftServerAPI.InitSDK succeeded
09 Sep 2021 11:46:32,993 [INFO] (gamelift.js) 223: Waiting for Realtime server to
 start...
```

```
09 Sep 2021 11:46:33,15 [WARN] (index.js) 204: Connection is INSECURE. Messages will be
 sent/received as plaintext.
```

Accessing server logs

When a game session ends, Amazon GameLift Servers automatically stores the logs in Amazon S3 and retains them for 14 days. You can use the GetGameSessionLogUrl API call to get the location of the logs for a game session. Use URL returned by the API call to download the logs.

Adjusting the logging level

Logs can grow very quickly and exceed the size limit. You should monitor your logs and limit the log output to necessary messages only. For Amazon GameLift Servers Realtime, you can adjust the logging level by providing a parameter in your fleet's runtime configuration in the form `loggingLevel:`*`LOGGING_LEVEL`*, where *`LOGGING_LEVEL`* is one of the following values:

1. `debug`
2. `info` *(default)*
3. `warn`
4. `error`
5. `fatal`
6. `cxfatal`

This list is ordered from least severe (`debug`) to most severe (`cxfatal`). You set a single `loggingLevel` and the server will only log messages at that severity level or a higher severity level. For example, setting `loggingLevel:error` will make all of the servers in your fleet only write `error`, `fatal`, and `cxfatal` messages to the log.

You can set the logging level for your fleet when you create it or after it is running. Changing your fleet's logging level after it is running will only affect logs for game sessions created after the update. Logs for any existing game sessions won't be affected. If you don't set a logging level when you create your fleet, your servers will set the logging level to `info` by default. Refer to the following sections for instructions to set the logging level.

Setting the logging level when creating a Amazon GameLift Servers Realtime fleet (Console)

Follow the instructions at Create a hosting fleet for Amazon GameLift Servers Realtime to create your fleet, with the following addition:

- In the **Process management** step, under **Server process allocation**, provide the logging level key-value pair (such as `loggingLevel:error`) as a value for **Launch parameters**. Use a non-alphanumeric character (except comma) to separate the logging level from any additional parameters (for example, `loggingLevel:error +map Winter444`).

Setting the logging level when creating a Amazon GameLift Servers Realtime fleet (Amazon CLI)

Follow the instructions at Create a hosting fleet for Amazon GameLift Servers Realtime to create your fleet, with the following addition:

- In the argument to the `--runtime-configuration` parameter for `create-fleet`, provide the logging level key-value pair (such as `loggingLevel:error`) as a value for `Parameters`. Use a non-alphanumeric character (except comma) to separate the logging level from any additional parameters. See the following example:

```
--runtime-configuration "GameSessionActivationTimeoutSeconds=60,
                    MaxConcurrentGameSessionActivations=2,
                    ServerProcesses=[{LaunchPath=/local/game/myRealtimeLaunchScript.js,
                                    Parameters=loggingLevel:error +map Winter444,
                                    ConcurrentExecutions=10}]"
```

Setting the logging level for a running Amazon GameLift Servers Realtime fleet (Console)

Follow the instructions at Update an Amazon GameLift Servers fleet configuration to update your fleet using the Amazon GameLift Servers Console, with the following addition:

- On the **Edit fleet** page, under **Server process allocation**, provide the logging level key-value pair (such as `loggingLevel:error`) as a value for **Launch parameters**. Use a non-alphanumeric character (except comma) to separate the logging level from any additional parameters (for example, `loggingLevel:error +map Winter444`).

Setting the logging level for a running Amazon GameLift Servers Realtime fleet (Amazon CLI)

Follow the instructions at Update an Amazon GameLift Servers fleet configuration to update your fleet using the Amazon CLI, with the following addition:

- In the argument to the `--runtime-configuration` parameter for `update-runtime-configuration`, provide the logging level key-value pair (such as `loggingLevel:error`) as a value for `Parameters`. Use a non-alphanumeric character (except comma) to separate the logging level from any additional parameters. See the following example:

```
--runtime-configuration "GameSessionActivationTimeoutSeconds=60,
                    MaxConcurrentGameSessionActivations=2,
                    ServerProcesses=[{LaunchPath=/local/game/myRealtimeLaunchScript.js,
                                    Parameters=loggingLevel:error +map Winter444,
                                    ConcurrentExecutions=10}]"
```

Amazon GameLift Servers Realtime reference guides

This section contains reference documentation for Amazon GameLift Servers Realtime. If you're building a game to use ready-to-use game servers provided by Amazon GameLift Servers for multi-player games, use these resources to help create a configuration script for the Realtime servers and prepare your game client to work with the servers.

For detailed guidance on getting your game ready to use Amazon GameLift Servers Realtime servers, see Preparing your games for hosting with Amazon GameLift Servers Realtime.

Topics

- Amazon GameLift Servers Realtime client API (C#) reference
- Script reference for Amazon GameLift Servers Realtime

Amazon GameLift Servers Realtime client API (C#) reference

Use this reference guide to understand how to implement Realtime client API functionality into your multiplayer game clients. For guidance on how to integrate this API into your game clients, see Integrate a game client for Amazon GameLift Servers Realtime.

The Realtime client API includes a set of synchronous API calls and asynchronous callbacks that enable a game client to connect to a Realtime server and exchange messages and data with other game clients via the server.

This API is defined in the following libraries:

Client.cs

- Synchronous actions
- Asynchronous callbacks
- Data types

To set up the Realtime client API

1. **Download the Amazon GameLift ServersRealtime client SDK.**
2. **Build the C# SDK libraries.** Locate the solution file `GameLiftRealtimeClientSdkNet45.sln`. See the `README.md` file for the C# Server SDK

for minimum requirements and additional build options. In an IDE, load the solution file. To generate the SDK libraries, restore the NuGet packages and build the solution.

3. **Add the Realtime Client libraries to your game client project.**

Amazon GameLift Servers Realtime client API (C#) reference: Actions

This C# Realtime Client API reference can help you prepare your multiplayer game for use with Amazon GameLift Servers Realtime deployed on Amazon GameLift Servers fleets.

- Synchronous Actions
- Asynchronous Callbacks
- Data Types

Client()

Initializes a new client to communicate with the Realtime server and identifies the type of connection to use.

Syntax

```
public Client(ClientConfiguration configuration)
```

Parameters

clientConfiguration

Configuration details specifying the client/server connection type. You can opt to call Client() without this parameter; however, this approach results in an unsecured connection by default.

Type: ClientConfiguration

Required: No

Return value

Returns an instance of the Realtime client for use with communicating with the Realtime server.

Connect()

Requests a connection to a server process that is hosting a game session.

Syntax

```
public ConnectionStatus Connect(string endpoint, int remoteTcpPort, int listenPort,
 ConnectionToken token)
```

Parameters

endpoint

DNS name or IP address of the game session to connect to. The endpoint is specified in a `GameSession` object, which is returned in response to a client call to the *Amazon SDK Amazon GameLift Servers API* actions StartGameSessionPlacement, CreateGameSession, or DescribeGameSessions.

Type: String

Required: Yes

remoteTcpPort

Port number for the TCP connection assigned to the game session. This information is specified in a `GameSession` object, which is returned in response to a StartGameSessionPlacement CreateGameSession, or DescribeGameSession request.

Type: Integer

Valid Values: 1900 to 2000.

Required: Yes

listenPort

Port number that the game client is listening on for messages sent using the UDP channel.

Type: Integer

Valid Values: 33400 to 33500.

Required: Yes

token

Optional information that identifies the requesting game client to the server process.

Type: ConnectionToken

Required: Yes

Return value

Returns a ConnectionStatus enum value indicating the client's connection status.

Disconnect()

When connected to a game session, disconnects the game client from the game session.

Syntax

```
public void Disconnect()
```

Parameters

This action has no parameters.

Return value

This method does not return anything.

NewMessage()

Creates a new message object with a specified operation code. Once a message object is returned, complete the message content by specifying a target, updating the delivery method, and adding a data payload as needed. Once completed, send the message using `SendMessage()`.

Syntax

```
public RTMessage NewMessage(int opCode)
```

Parameters

opCode

Developer-defined operation code that identifies a game event or action, such as a player move or a server notification.

Type: Integer

Required: Yes

Return value

Returns an RTMessage object containing the specified operation code and default delivery method. The delivery intent parameter is set to FAST by default.

SendMessage()

Sends a message to a player or group using the delivery method specified.

Syntax

```
public void SendMessage(RTMessage message)
```

Parameters

message

Message object that specifies the target recipient, delivery method, and message content.

Type: RTMessage

Required: Yes

Return value

This method does not return anything.

JoinGroup()

Adds the player to the membership of a specified group. Groups can contain any of the players that are connected to the game. Once joined, the player receives all future messages sent to the group and can send messages to the entire group.

Syntax

```
public void JoinGroup(int targetGroup)
```

Parameters

targetGroup

Unique ID that identifies the group to add the player to. Group IDs are developer-defined.

Type: Integer

Required: Yes

Return value

This method does not return anything. Because this request is sent using the reliable (TCP) delivery method, a failed request triggers the callback OnError().

LeaveGroup()

Removes the player from the membership of a specified group. Once no longer in the group, the player does not receive messages sent to the group and cannot send messages to the entire group.

Syntax

```
public void LeaveGroup(int targetGroup)
```

Parameters

targetGroup

Unique ID identifying the group to remove the player from. Group IDs are developer-defined.

Type: Integer

Required: Yes

Return value

This method does not return anything. Because this request is sent using the reliable (TCP) delivery method, a failed request triggers the callback OnError().

RequestGroupMembership()

Requests that a list of players in the specified group be sent to the game client. Any player can request this information, regardless of whether they are a member of the group or not. In response to this request, the membership list is sent to the client via an OnGroupMembershipUpdated() callback.

Syntax

```
public void RequestGroupMembership(int targetGroup)
```

Parameters

targetGroup

Unique ID identifying the group to get membership information for. Group IDs are developer-defined.

Type: Integer

Required: Yes

Return value

This method does not return anything.

Amazon GameLift Servers Realtime client API (C#) reference: Asynchronous callbacks

Use this C# Realtime Client API reference to help you prepare your multiplayer game for use with Amazon GameLift Servers Realtime deployed on Amazon GameLift Servers fleets.

- Synchronous Actions

- Asynchronous Callbacks
- Data Types

A game client needs to implement these callback methods to respond to events. The Realtime server invokes these callbacks to send game-related information to the game client. Callbacks for the same events can also be implemented with custom game logic in the Realtime server script. See Script callbacks for Amazon GameLift Servers Realtime.

Callback methods are defined in `ClientEvents.cs`.

OnOpen()

Invoked when the server process accepts the game client's connection request and opens a connection.

Syntax

```
public void OnOpen()
```

Parameters

This method takes no parameters.

Return value

This method does not return anything.

OnClose()

Invoked when the server process terminates the connection with the game client, such as after a game session ends.

Syntax

```
public void OnClose()
```

Parameters

This method takes no parameters.

Return value

This method does not return anything.

OnError()

Invoked when a failure occurs for a Realtime Client API request. This callback can be customized to handle a variety of connection errors.

Syntax

```
private void OnError(byte[] args)
```

Parameters

This method takes no parameters.

Return value

This method does not return anything.

OnDataReceived()

Invoked when the game client receives a message from the Realtime server. This is the primary method by which messages and notifications are received by a game client.

Syntax

```
public void OnDataReceived(DataReceivedEventArgs dataReceivedEventArgs)
```

Parameters

dataReceivedEventArgs

Information related to message activity.

Type: DataReceivedEventArgs

Required: Yes

Return value

This method does not return anything.

OnGroupMembershipUpdated()

Invoked when the membership for a group that the player belongs to has been updated. This callback is also invoked when a client calls `RequestGroupMembership`.

Syntax

```
public void OnGroupMembershipUpdated(GroupMembershipEventArgs groupMembershipEventArgs)
```

Parameters

groupMembershipEventArgs

Information related to group membership activity.

Type: GroupMembershipEventArgs

Required: Yes

Return value

This method does not return anything.

Amazon GameLift Servers Realtime client API (C#) reference: Data types

This C# Realtime Client API reference can help you prepare your multiplayer game for use with Amazon GameLift Servers Realtime deployed on Amazon GameLift Servers fleets.

- Synchronous Actions
- Asynchronous Callbacks
- Data Types

ClientConfiguration

Information about how the game client connects to a Realtime server.

Contents

ConnectionType

Type of client/server connection to use, either secured or unsecured. If you don't specify a connection type, the default is unsecured.

Type: A `ConnectionType` enum value.

Required: No

ConnectionToken

Information about the game client and/or player that is requesting a connection with a Realtime server.

Contents

playerSessionId

Unique ID issued by Amazon GameLift Servers when a new player session is created. A player session ID is specified in a `PlayerSession` object, which is returned in response to a client call to the *GameLift API* actions StartGameSessionPlacement, CreateGameSession, DescribeGameSessionPlacement, or DescribePlayerSessions.

Type: String

Required: Yes

payload

Developer-defined information to be communicated to the Realtime server on connection. This includes any arbitrary data that might be used for a custom sign-in mechanism. For examples, a payload may provide authentication information to be processed by the Realtime server script before allowing a client to connect.

Type: byte array

Required: No

RTMessage

Content and delivery information for a message. A message must specify either a target player or a target group.

Contents

opCode

Developer-defined operation code that identifies a game event or action, such as a player move or a server notification. A message's Op code provides context for the data payload that is being provided. Messages that are created using NewMessage() already have the operation code set, but it can be changed at any time.

Type: Integer

Required: Yes

targetPlayer

Unique ID identifying the player who is the intended recipient of the message being sent. The target may be the server itself (using the server ID) or another player (using a player ID).

Type: Integer

Required: No

targetGroup

Unique ID identifying the group that is the intended recipient of the message being sent. Group IDs are developer defined.

Type: Integer

Required: No

deliveryIntent

Indicates whether to send the message using the reliable TCP connection or using the fast UDP channel. Messages created using NewMessage().

Type: DeliveryIntent enum

Valid values: FAST | RELIABLE

Required: Yes

payload

Message content. This information is structured as needed to be processed by the game client based on the accompanying operation code. It may contain game state data or other information that needs to be communicated between game clients or between a game client and the Realtime server.

Type: Byte array

Required: No

DataReceivedEventArgs

Data provided with an OnDataReceived() callback.

Contents

sender

Unique ID identifying the entity (player ID or server ID) who originated the message.

Type: Integer

Required: Yes

opCode

Developer-defined operation code that identifies a game event or action, such as a player move or a server notification. A message's Op code provides context for the data payload that is being provided.

Type: Integer

Required: Yes

data

Message content. This information is structured as needed to be processed by the game client based on the accompanying operation code. It may contain game state data or other information that needs to be communicated between game clients or between a game client and the Realtime server.

Type: Byte array

Required: No

GroupMembershipEventArgs

Data provided with an OnGroupMembershipUpdated() callback.

Contents

sender

Unique ID identifying the player who requested a group membership update.

Type: Integer

Required: Yes

opCode

Developer-defined operation code that identifies a game event or action.

Type: Integer

Required: Yes

groupId

Unique ID identifying the group that is the intended recipient of the message being sent. Group IDs are developer defined.

Type: Integer

Required: Yes

playerId

List of player IDs who are current members of the specified group.

Type: Integer array

Required: Yes

Enums

Enums defined for the client SDK for Amazon GameLift Servers Realtime are defined as follows:

ConnectionStatus

- CONNECTED – Game client is connected to the Realtime server with a TCP connection only. All messages regardless of delivery intent are sent via TCP.
- CONNECTED_SEND_FAST – Game client is connected to the Realtime server with a TCP and a UDP connection. However, the ability to receive messages via UDP is not yet verified; as a result, all messages sent to the game client use TCP.
- CONNECTED_SEND_AND_RECEIVE_FAST – Game client is connected to the Realtime server with a TCP and a UDP connection. The game client can send and receive messages using either TCP or UDP.
- CONNECTING Game client has sent a connection request and the Realtime server is processing it.
- DISCONNECTED_CLIENT_CALL – Game client was disconnected from the Realtime server in response to a Disconnect()request from the game client.
- DISCONNECTED – Game client was disconnected from the Realtime server for a reason other than a client disconnect call.

ConnectionType

- RT_OVER_WSS_DTLS_TLS12 – Secure connection type.
- RT_OVER_WS_UDP_UNSECURED – Non-secure connection type.
- RT_OVER_WEBSOCKET – Non-secure connection type. This value is no longer preferred.

DeliveryIntent

- FAST – Delivered using a UDP channel.
- RELIABLE – Delivered using a TCP connection.

Script reference for Amazon GameLift Servers Realtime

Use this reference guide to understand the options available when configuring a Realtime script and optionally adding custom game logic for Amazon GameLift Servers Realtime servers. For guidance on how to configure a Realtime script, including required elements, see Customize an Amazon GameLift Servers Realtime script.

Topics

Script callbacks for Amazon GameLift Servers Realtime

You can provide custom logic to respond to events by implementing these callbacks in your Realtime script.

Init

Initializes the Realtime server and receives a Realtime server interface.

Syntax

```
init(rtsession)
```

onMessage

Invoked when a received message is sent to the server.

Syntax

```
onMessage(gameMessage)
```

onHealthCheck

Invoked to set the status of the game session health. By default, health status is healthy (or `true`. This callback can be implemented to perform custom health checks and return a status.

Syntax

```
onHealthCheck()
```

onStartGameSession

Invoked when a new game session starts, with a game session object passed in.

Syntax

```
onStartGameSession(session)
```

onProcessTerminate

Invoked when the server process is being terminated by the Amazon GameLift Servers service. This can act as a trigger to exit cleanly from the game session. There is no need to call `processEnding()`.

Syntax

```
onProcessTerminate()
```

onPlayerConnect

Invoked when a player requests a connection and has passed initial validation.

Syntax

```
onPlayerConnect(connectMessage)
```

onPlayerAccepted

Invoked when a player connection is accepted.

Syntax

```
onPlayerAccepted(player)
```

onPlayerDisconnect

Invoked when a player disconnects from the game session, either by sending a disconnect request or by other means.

Syntax

```
onPlayerDisconnect(peerId)
```

onProcessStarted

Invoked when a server process is started. This callback allows the script to perform any custom tasks needed to prepare to host a game session.

Syntax

```
onProcessStarted(args)
```

onSendToPlayer

Invoked when a message is received on the server from one player to be delivered to another player. This process runs before the message is delivered.

Syntax

```
onSendToPlayer(gameMessage)
```

onSendToGroup

Invoked when a message is received on the server from one player to be delivered to a group. This process runs before the message is delivered.

Syntax

```
onSendToGroup(gameMessage))
```

onPlayerJoinGroup

Invoked when a player sends a request to join a group.

Syntax

```
onPlayerJoinGroup(groupId, peerId)
```

onPlayerLeaveGroup

Invoked when a player sends a request to leave a group.

Syntax

```
onPlayerLeaveGroup(groupId, peerId)
```

Amazon GameLift Servers Realtime interface

When an Amazon GameLift Servers Realtime script initializes, an interface to the Realtime server is returned. This topic describes the properties and methods available through the interface. Learn more about writing Realtime scripts and view a detailed script example in Customize an Amazon GameLift Servers Realtime script.

The Realtime interface provides access to the following objects:

- session
- player
- gameMessage
- configuration

Realtime Session object

Use these methods to access server-related information and perform server-related actions.

getPlayers()

Retrieves a list of peer IDs for players that are currently connected to the game session. Returns an array of player objects.

Syntax

```
rtSession.getPlayers()
```

broadcastGroupMembershipUpdate()

Triggers delivery of an updated group membership list to player group. Specify which membership to broadcast (groupIdToBroadcast) and the group to receive the update (targetGroupId). Group IDs must be a positive integer or "-1" to indicate all groups. See Amazon GameLift Servers Realtime script example for an example of user-defined group IDs.

Syntax

```
rtSession.broadcastGroupMembershipUpdate(groupIdToBroadcast, targetGroupId)
```

getServerId()

Retrieves the server's unique peer ID identifier, which is used to route messages to the server.

Syntax

```
rtSession.getServerId()
```

getAllPlayersGroupId()

Retrieves the group ID for the default group that contains all players currently connected to the game session.

Syntax

```
rtSession.getAllPlayersGroupId()
```

processEnding()

Triggers the Realtime server to terminate the game server. This function must be called from the Realtime script to exit cleanly from a game session.

Syntax

```
rtSession.processEnding()
```

getGameSessionId()

Retrieves the unique ID of the game session currently running.

Syntax

```
rtSession.getGameSessionId()
```

getLogger()

Retrieves the interface for logging. Use this to log statements that will be captured in your game session logs. The logger supports use of "info", "warn", and "error" statements. For example: `logger.info("<string>")`.

Syntax

```
rtSession.getLogger()
```

sendMessage()

Sends a message, created using `newTextGameMessage` or `newBinaryGameMessage`, from the Realtime server to a player recipient using the UDP channel. Identify the recipient using the player's peer ID.

Syntax

```
rtSession.sendMessage(gameMessage, targetPlayer)
```

sendGroupMessage()

Sends a message, created using `newTextGameMessage` or `newBinaryGameMessage`, from the Realtime server to all players in a player group using the UDP channel. Group IDs must be a positive integer or "-1" to indicate all groups. See Amazon GameLift Servers Realtime script example for an example of user-defined group IDs.

Syntax

```
rtSession.sendGroupMessage(gameMessage, targetGroup)
```

sendReliableMessage()

Sends a message, created using `newTextGameMessage` or `newBinaryGameMessage`, from the Realtime server to a player recipient using the TCP channel. Identify the recipient using the player's peer ID.

Syntax

```
rtSession.sendReliableMessage(gameMessage, targetPlayer)
```

sendReliableGroupMessage()

Sends a message, created using `newTextGameMessage` or `newBinaryGameMessage`, from the Realtime server to all players in a player group using the TCP channel. Group IDs which must be a positive integer or "-1" to indicate all groups. See Amazon GameLift Servers Realtime script example for an example of user-defined group IDs.

Syntax

```
rtSession.sendReliableGroupMessage(gameMessage, targetGroup)
```

newTextGameMessage()

Creates a new message containing text, to be sent from the server to player recipients using the SendMessage functions. Message format is similar to the format used in the client SDK for Realtime (see RTMessage). Returns a `gameMessage` object.

Syntax

```
rtSession.newTextGameMessage(opcode, sender, payload)
```

newBinaryGameMessage()

Creates a new message containing binary data, to be sent from the server to player recipients using the SendMessage functions. Message format is similar to the format used in the client SDK for Realtime (see RTMessage). Returns a `gameMessage` object.

Syntax

```
rtSession.newBinaryGameMessage(opcode, sender, binaryPayload)
```

Player object

Access player-related information.

player.peerId

Unique ID that is assigned to a game client when it connects to the Realtime server and joined the game session.

player.playerSessionId

Player session ID that was referenced by the game client when it connected to the Realtime server and joined the game session.

Game message object

Use these methods to access messages that are received by the Realtime server. Messages received from game clients have the RTMessage structure.

getPayloadAsText()

Gets the game message payload as text.

Syntax

```
gameMessage.getPayloadAsText()
```

gameMessage.opcode

Operation code contained in a message.

gameMessage.payload

Payload contained in a message. May be text or binary.

gameMessage.sender

Peer ID of the game client that sent a message.

gameMessage.reliable

Boolean indicating whether the message was sent via TCP (true) or UDP (false).

Configuration object

The configuration object can be used to override default configurations.

configuration.maxPlayers

The maximum number of client / server connections that can be accepted by RealTimeServers.

The default is 32.

configuration.pingIntervalTime

Time interval in milliseconds that server will attempt to send a ping to all connected clients to verify connections are healthy.

The default is 3000ms.