Amazon GameLift

Developer Guide

Version

Amazon GameLift: Developer Guide

Table of Contents

What is Amazon GameLift?

Amazon GameLift is a fully managed service for deploying, operating, and scaling your session-based multiplayer game servers in the cloud. If you're developing games using Amazon Lumberyard, GameLift replaces the work required to host your own game servers, including buying and setting up hardware, and managing ongoing activity, security, storage, and performance tracking. GameLift's autoscaling features provide additional protection from having to pay for more resources than you need, while helping to ensure that your players can find and join games fast.

Why Amazon GameLift?

Here are some of the benefits of using GameLift:

- Provide low-latency player experience to support fast-action game play.
- Release session-based, multiplayer games fast, with little or no back-end experience required.
- Reduce engineering and operational effort to deploy and operate game servers.
- Get started fast and pay as you go, with no upfront costs and no long-term commitments.
- Reduce the risks involved in handling fluctuating player traffic.
- Rely on Amazon Web Services (AWS), including Amazon Elastic Compute Cloud (Amazon EC2) for web-scale cloud computing resources and automatic scaling to manage your hosting capacity.

Key Features

Amazon GameLift includes these features:

- Use the Amazon Lumberyard game engine features to set up GameLift for your game.
- Deploy game servers to run on either Amazon Linux or Windows Server operating systems.
- Provide high-quality game hosting to players around the world by deploying to computing resources in multiple regions.
- Use autoscaling tools to adjust your game hosting capacity to meet actual player usage and balance player experience against cost savings.
- Configure game session characteristics, such as maximum number of players allowed, join rules, and game-specific properties.
- Help players find sessions to join quickly with game session search features.
- Analyze game performance using the GameLift console to track metrics, view game session logs, and review data on individual game sessions and player sessions.

- Set up customized health tracking for server processes to detect problems fast and resolve poor-performing processes.

Integration Steps

If you're using Amazon GameLift for the first time, we recommend that you get familiar with the service by reading the topics in this section. When you are ready to start using Amazon GameLift, follow this integration workflow to prepare and deploy your games.

1. **Get set up to use Amazon GameLift.** Create and configure your AWS account, and install tools and resources. See Setting Up (p. 9).
2. **Prepare your game server for hosting on GameLift.**
   - Add hooks to your game server code to allow it to interact with the GameLift service. Use either the Amazon Lumberyard game engine or the GameLift Server SDK. See Integrating your Game Server for Amazon GameLift (p. 12).
   - Upload your game builds to the GameLift service. See Uploading Your Game to Amazon GameLift (p. 34).
3. **Build a fleet of virtual computing resources to host your game.** Define the type of resource instances to use and configure how you want each instance to deploy your game servers. See Working with Fleets (p. 37).
4. **Prepare your game client to connect to GameLift-hosted game sessions.** Add hooks to your game client code to enable players to find and connect to game sessions. Use either the Amazon Lumberyard game engine or the AWS SDK with GameLift. See Integrating your Game Client for Amazon GameLift (p. 25).
5. **Manage player capacity.** Set the number of game sessions to have available for players. You can adjust capacity manually or use autoscaling to have capacity track player usage patterns. See Working with Fleets (p. 37) topics on updating capacity and autoscaling.
6. **Track performance and usage metrics.** Use collected data to learn: (1) how players are connecting with and playing your game, (2) how well your hosting configuration and capacity settings are meeting player needs, and (3) how healthy your game servers perform over time and usage. See Viewing Your Game Data in the Console (p. 53).

Tip
You don't need to have a game ready to start experimenting with GameLift. The GameLift console offers a quick sample setup that gets you up and running with a sample game server and client in five easy steps. In addition, the GameLift Getting Started tutorials, in text and video format, walk you through each step in the process of creating and uploading a build, setting up a fleet, creating game sessions and connecting a client. The tutorials use a sample multiplayer game, which is included in the Lumberyard download.

How Amazon GameLift Works

This topic provides an overview of Amazon GameLift components and how the service works to deploy your multiplayer game servers and manage player traffic.

Key Components

Setting up Amazon GameLift to host your game involves working with the following components:

- A **game server** is your game's server software running in the cloud. You provide your game servers to GameLift by uploading a **game build**, which includes the server executables, supporting assets,

libraries, and dependencies. GameLift deploys the game server as a set of server processes, each of which hosts a game session for players.

- The **GameLift service** manages the virtual resources needed to host your game server processes, and makes it possible for players to connect to games. It does this by regulating resources to accommodate player activity, handling players' join requests and directing them to appropriate server processes, and enforcing rules that control which game servers can be joined. The service also collects performance data on server process health and player usage.
- A **game client** is your game's software running on a player's device. It enables a player to connect to one of your game server processes on GameLift and play your game.
- Optional **game services** might communicate with the GameLift service for a variety of purposes. For example, you might create a game service to act as an intermediary between game clients and servers, such as to manage matchmaking or player authentication.

See Amazon GameLift and Game Client/Server Interactions (p. 30) for a detailed description of how these components interact.

Configuring Computing Resources

To deploy a game server on GameLift, you create a **fleet** of virtual computing resources and specify how you want GameLift to use those resources to host your game. You configure the fleet to use a specific type of Amazon Elastic Compute Cloud (Amazon EC2) resource, called an **instance**, based on how much computing power you need for your game. When running your game server, GameLift creates and terminates instances as needed to manage capacity. (See GameLift service limits for more information on how many instances can be used with your AWS account.)

Your fleet configuration also specifies the game build to deploy, and describes how many game server processes should be run on each instance in the fleet. You'll need to balance the size and type of instance you're using against the computing requirements of the total number of server processes for an instance.

> **Note**
> GameLift supports multiple **regions** to optimize gaming performance for players around the world. Each fleet you set up is configured to deploy your game to a specific region, To make your game available to players in multiple regions, you must set up a separate fleet for each region. See a list of available regions for GameLift at AWS Regions and Endpoints.

You may want to assign an **alias** to a fleet. An alias is a convenient way to genericize how game clients connect to game servers. Because an alias can be changed to point to any fleet you want, referencing an alias ID in your client instead of a fleet ID helps you gracefully transition players from one fleet to another—without having to deploy game client updates.

Once a fleet is active, GameLift is ready to start accepting requests for new game sessions. From this point on, you can manage fleet configuration, capacity, and utilization as demand requires.

Handling Capacity and Utilization

Here's how GameLift handles capacity and utilization for multiplayer game sessions in real time.

To create a new **game session**, a game client sends a request to the GameLift service. When GameLift receives the request, which specifies the fleet and type of game server, it goes through the following sequence:

1. The GameLift service checks the fleet for any instance with an idle server process of the right type. If one exists, a new game session is launched and the player can connect to it.
2. If no idle server processes are available, the request to create a new game session fails. The game client can continue to request a new game session, and requests will fail until an idle server

process becomes available. This occurs either when an active game session ends and frees up the server process, or when fleet capacity is increased and new instances begin running new server processes.

You can change a fleet's capacity at any time, increasing or decreasing the number of desired instances. When you increase capacity, GameLift immediately begins the process of starting a new instance and a new set of idle server processes, which can then host new game sessions as they are requested. This continues until the new capacity setting is reached.

When you decrease fleet capacity, a scale down event is triggered. GameLift selects instances to terminate in order to meet the reduced capacity setting. Any instance in the fleet may be terminated, regardless of whether or not the instance is hosting active game sessions. You can choose to avoid terminating instances with active game sessions by turning on a feature called game session protection, either for an individual game session or for an entire fleet. An instance can only be terminated when none of its server processes are hosting protected game sessions. If a scale down event is triggered but all instances in the fleet have at least one protected game session, no scale down action takes place.

While the ability to set fleet capacity lets you control overall usage and costs for your GameLift resources, it is also possible to control how players can consume fleet resources (that is, create new game sessions). For games that allow individual players to create game sessions, you can use a resource creation limit policy to a fleet to restrict the number of game sessions that any one player can create over a span of time.

Autoscaling

Autoscaling simply automates the process of increasing or decreasing capacity, enabling the fleet to quickly accommodate changes in player demand. Usage peaks and valleys can fluctuate unpredictably, making it difficult to manually adjust capacity that strikes a balance between maintaining enough fleet capacity to accommodate incoming players and avoiding paying for unused resources during idle times.

The goal of autoscaling is to have GameLift change capacity on the fly in response to player demand. Autoscaling is a set of rules, based on actual player activity, that you define to tell GameLift when to increase or reduce capacity. With the right rules in place, GameLift can maintain a capacity level that always has room for new players without 't running idle servers. Autoscaling rules, called **policies**, are based on certain GameLift metrics that track player utilization. These include the number of current players, the number of available player slots, and the number of idle instances, as well as others.

An autoscaling policy statement takes the following form: "If a specified metric hits or exceeds a specified threshold value for a specified number of minutes, then change fleet capacity a specified amount." So, for example, you might decide to use available player slots as an indicator of when demand is spiking up: "If the number of available player slots falls below 50 for more than 10 consecutive minutes, then increase capacity by 1 instance."

A fleet can have multiple policies in force; each one is evaluated independently. For example, you might have a policy that tells GameLift to decrease capacity by 10% if the number of idle instances is more than 5 for 15 minutes. If you also have a policy to decrease capacity based on number of available player slots, both policies can be triggered as demand decreases. A more common scenario is to have a set of policies, each of which responds to either increasing demand or decreasing demand.

Autoscaling and game session protection work together to ensure that players are not arbitrarily dropped as GameLift automatically adjusts capacity and takes down unneeded game servers to save you money. If autoscaling triggers a scale down but all available instances are hosting active game sessions and are protected, the scale down fails and capacity does not change. However, the policy will continue to be evaluated with each metric report; if it continues to trigger a scale down, one will occur the next time a game session ends. As a result, capacity is automatically adjusted without affecting active players.

Monitoring Fleet Activity and Troubleshooting

Once you have fleets up and running, GameLift collects a variety of information to help you monitor the performance of your deployed game servers. Use this information to optimize your use of resources, troubleshoot issues, and gain insight into how players are active in your games.

- **Fleet, game session, and player session details** – This data includes status, which can help identify health issues, as well as details such as game session length and player connection time.
- **Utilization metrics** – GameLift tracks fleet metrics over time:
  - For instances: network activity and CPU utilization
  - For server processes: number of active processes, new activations, and terminations
  - For games and players: number of active game sessions and player sessions
- **Server process health** – GameLift tracks the health of each server process running on a fleet, including the number of healthy processes, percent of active processes that are healthy, and number of abnormal terminations.
- **Game session logs** – You can have your game servers log session data and set GameLift to collect and store the logs one the game session ends. Logs can then be downloaded from the service.

All of this data is available through the Amazon GameLift console. The console dashboard presents an overview of activity across all you builds and fleets as well as the option to drill down to more detailed information.

How Players Connect to Games

In Amazon GameLift, a game session is a process that is running your game server, which players can connect to and play. Game sessions are created by making calls to the AWS SDK for Amazon GameLift from a game client or game service. You can create game sessions only after you have a game server deployed on a GameLift fleet.

Players connect to a game session by requesting a player session for an existing game session. If the game session can be joined--that is, it has an open player slot and it is accepting new players, GameLift reserves a slot for the player. It then gives the game client the information it needs to connect to the game session and claim the reserved slot.

See more about integrating GameLift into your game client or service in Add Amazon GameLift to Your Game Client (p. 25).

Game and Player Session Features

GameLift provides several features related to game and player sessions:

Session search and sort

Use session search and sort to populate game session browsers and give players information that helps them quickly find a session to join. This can be an effective way to lead players to sessions that are most likely to result in a positive gaming experience. For example, if your game requires a certain number of players before launching gameplay, directing new players to sessions that are closest to that minimum helps fill sessions quickly with a minimum of wait time. For games with very short session lifespans, you might want to hide sessions that are older than a few minutes. Alternatively, session search can eliminate the need for a session browser altogether, instead backing a "join now" feature with a well-formulated search and sort expression that gets players into positive gaming experiences fast. Currently, you can search and/or sort by the following session characteristics:

- **Session age** – how long the session has been running

- **Current player count** – the number of players currently connected to the game
- **Maximum players count** – maximum number of players allowed
- **Open player slots** – whether or not a session has room for a new player to join
- **Session name** – nonunique name given the session on creation
- **Session ID** – unique ID number assigned to the session on creation

Custom game properties

Use custom game properties to configure a game session on launch. Define meaningful game properties and pass them directly to the game server when you create a new game session. Game properties are formatted as a simple collection of key:value string pairs (ex: ["key": "map", "value": "WinterGarden"]). You can also retrieve game properties for a game session and display them in a game session browser to help players find sessions faster and with better results.

Player access control

Set a game session to allow or deny join requests from new players, regardless of the number of players currently connected. You might use this feature to enable private sessions, to limit access for troubleshooting or other problem resolution, etc.

Session logs

Have GameLift automatically store logs for completed game sessions. Set up log storage when integrating GameLift into your game servers. You can access stored logs by downloading them using the GameLift console or programmatically with the AWS SDK for GameLift.

Utilization data and metrics

Use the Amazon GameLift console to view detailed information on game sessions, including session metadata and settings as well as player session data. For each game session, you can view a list of player sessions along with total times played. You can also view metrics data and graphs that track the number of active game sessions and player sessions over time. See more information at View Data on Game and Player Sessions (p. 60) and Metrics (p. 56).

Tools and Resources

Amazon GameLift provides a collection of tools and resources for you to use.

Core Tools

Use these tools to work with Amazon GameLift.

Amazon Lumberyard game engine
: Amazon Lumberyard comes with Amazon GameLift functionality built in, and integration is handled automatically. The Amazon GameLift Server SDK can be found in `\3rdParty\AWS\GameLift`. See the Lumberyard User Guide for more information.

GameLift SDKs
: The GameLift SDKs contain the libraries needed to communicate with the Amazon GameLift service from your game clients, game servers and game services. Versions of these SDKs are available with Lumberyard or you can download the latest versions separately. See details in Amazon GameLift SDKs (p. 7).

Amazon GameLift console
: Use the AWS Management Console for GameLift to manage your game deployments, configure resources, and track player usage and performance metrics. The GameLift console provides a GUI alternative to managing resources programmatically with the AWS SDK.

AWS CLI for GameLift
Use this command line tool to make calls to the AWS SDK, including the GameLift API. Download the AWS Command Line Interface or view the http://docs.aws.amazon.com/cli/latest/reference/gamelift/AWS CLI Command Reference for GameLift.

Additional Resources

Use these resources to learn and experiment using GameLift with your multiplayer games.

Getting Started tutorials
These video and text tutorials walk you through the process of getting a multiplayer game up and running on GameLift. using a sample game bundled with the Lumberyard download. Once you complete the series, you can use the game to explore other GameLift features and tools, such as autoscaling and performance metrics (no charge if you use the Amazon GameLift free tier).

Sample multiplayer project
The sample Lumberyard project MultiplayerProject is bundled with the Lumberyard download. It illustrates how to use Lumberyard to integrate a game with GameLift. See the Lumberyard User Guide for more information.

Other resources
For additional help consult the following resources:

- **GameLift forum** – Use the GameDev forums to exchange ideas and knowledge, pick up tips, and get help with issues related to the GameLift service.
- **Amazon GameDev Blog** – Watch the blog to keep up with new features and get expert tips from the team.
- **GameLift FAQ** – Check the FAQ for answers to your general questions about the service and pricing details.

Amazon GameLift SDKs

Use Amazon GameLift software development kits (SDKs) to develop GameLift-enabled multiplayer game servers, game clients and game services that need to connect to the GameLift service.

For Game Servers

Create and deploy 64-bit game servers with the *GameLift Server SDK*. This SDK enables the GameLift service to deploy and manage game server processes across GameLift virtual resources. Download the Server SDK or view the API reference documentation (p. 15).

The GameLift Server SDK is available in the following languages:

- C++

You can build game servers to run on the following platforms:

- Windows Server 2012 R2
- Amazon Linux

For Game Clients and Game Services

Create 64-bit game clients and services using the AWS SDK with the GameLift API. This SDK enables client apps and services to find and manage game sessions and connect players to games being

hosted on GameLift. You can also use this SDK to programmatically manage your game hosting resources and access performance and utilization statistics. Download the AWS SDK or view the GameLift API reference documentation.

The AWS SDK with GameLift is available in the following languages:

- C++
- Java
- .NET
- Go
- Python
- Ruby
- PHP
- JavaScript/Node.js

SDK Compatibility

If you use the GameLift SDKs bundled inside a version of Amazon Lumberyard, your game clients and servers will be compatible. If you upgrade the GameLift Server SDK independently, however, you need to use a compatible version of the AWS SDK to ensure that your game clients and services can successfully connect to your game servers on GameLift.

If your game server uses this Server SDK version:	It can host game clients built with this AWS SDK for C++ version*:	Server SDK versions are available in:
Versions bundled into Amazon Lumberyard.	Versions bundled into Amazon Lumberyard.	Lumberyard v.1.0 to v.1.3 (beta)
version 3.0.7	version 0.12.16 (commit) or later	• Lumberyard v.1.4 or later • Download from GameLift
version 3.1.0	version 1.10.61 or later	Download from GameLift

* Version information for the AWS SDK for C++ can be found in this file: `aws-sdk-cpp/aws-cpp-sdk-core/include/aws/core/VersionConfig.h`.

Free Tier and Billing Alerts

Amazon GameLift includes a free tier that provides 125 hours of a c3.large instance per month for one year. It is possible for the free tier to expire mid-month; therefore, you may want to set up and configure a billing alert to notify you of billing events, such as when you have reached the free tier threshold. For more information, see Creating a Billing Alarm.

In addition to receiving billing alerts, you can view your current estimated bill for Amazon GameLift on the Billing and Cost Management Dashboard. This will allow you to review your resource consumption and determine if you would like to continue using these resources beyond the free tier allowance and incur charges, or if you would like to scale down your fleet and avoid incurring charges.

To avoid incurring charges in excess of the free tier, you may want to scale down your fleet when not in use.

Setting Up

The topics in this section describe the basic steps needed to begin using Amazon GameLift.

Topics

Set Up Your Amazon Lumberyard Game Project

Get your project ready for Amazon GameLift with the following steps. If you have installed Lumberyard, you may have already completed some of these tasks.

- Install Visual Studio 2013 runtime. Run the installer from the `\3rdParty\Redistributables\Visual Studio 2013` directory or download and run the installer directly from Microsoft.
- Run Setup Assistant to validate that you have installed the appropriate third-party software and SDKs, including the Amazon GameLift client. `LumberyardLauncherBatch.exe` is provided in `\dev\Bin64`.
- Configure your Lumberyard game project to ensure it compiles properly. Follow these guidelines:
  - The server and client executables must link `aws-cpp-sdk-core` and `aws-cpp-sdk-gamelift`.
  - The server executable must be built on a platform supported by Amazon GameLift. See Amazon GameLift SDKs (p. 7) for a list of allowed platforms.
  - Your project must set the `AWS_CUSTOM_MEMORY_MANAGEMENT` pre-processor flag to `0` or `1`, depending on your use of a custom memory manager.

Set up an AWS Account

Amazon GameLift is an AWS service, and you must have an AWS account to use GameLift. Creating an AWS account is free.

For more information on what you can do with an AWS account, see Getting Started with AWS.

Set up your account for GameLift

1. **Get an account.** Open Amazon Web Services and choose **Sign In to the Console**. Follow the prompts to either create a new account or sign into an existing one.

2. **Set up user groups and access permissions.** Open the AWS Identity and Access Management (IAM) service console and follow the steps below to define a set of users or user groups and assign access permissions to them. Permissions are extended to a user or user group by attaching an IAM policy, which specifies the set of AWS services and actions a user should have access to. For detailed instructions on using the Console (or the AWS CLI or other tools) to set up your user groups, see Creating IAM Users.

    1. **Create an administrative user or user group**. Administrative users include anyone who manages core GameLift resources, such as builds and fleets. To set permissions, you must create your own policy from scratch. This example (p. 10) illustrates an administrator policy for GameLift services.
    2. **Create a player user**. A player user represents your game client(s). It enables access to GameLift client functionality, such as acquiring game session information and joining players to games. Your game client must use the player user credentials when communicating with the GameLift service. To set permissions, you must create your own policy from scratch. This example (p. 10) illustrates a player policy for GameLift services.

IAM Policy Examples for Amazon GameLift

You can use the following examples to create policies and add the appropriate permissions to your IAM users or user groups.

Simple Policy for Administrators

This policy provides full administrative access to a user. Attach it to a user or user group to permit all Amazon GameLift actions on all GameLift resources (fleets, aliases, game sessions, player sessions, etc.).

```
{
"Version": "2012-10-17",
"Statement":
  {
    "Effect": "Allow",
    "Action": "gamelift:*",
    "Resource": "*"
  }
}
```

Simple Policy for Players

This policy enables access only to functionality needed by players who are using a game client to connect to a GameLift-hosted game server. This policy allows a user to get game session information, create new game sessions, and join a game session.

```
{
"Version": "2012-10-17",
"Statement":
  {
    "Effect": "Allow",
    "Action": [ "gamelift:CreateGameSession",
 "gamelift:DescribeGameSessions", "gamelift:SearchGameSessions",
 "gamelift:CreatePlayerSession" ],
    "Resource": "*"
  }
}
```

Install the AWS CLI

You can use the AWS command line interface (AWS CLI) tool to make calls to any action in the AWS SDK, including the Amazon GameLift API. At a minimum, you'll need to use this tool to upload your game builds to the GameLift service.

To install the AWS CLI for GameLift

1. **Get the tool.** Download the latest version of the AWS CLI tool and follow the instructions to install it. This tool runs on Windows, Linux, OS X, and Unix.
2. **Verify installation.** Open a command line window or terminal and type `aws gamelift help`. If the CLI is correctly installed, you will see a "Welcome to the Amazon GameLift API Reference" message, followed by a list of GameLift commands.
3. **Configure the tool.** Type `aws configure` and enter the following values at the prompts:

   - **AWS access key ID** – Half of the AWS account user credentials, which are generated using the IAM service. For help, see Get your access key ID and secret access key.
   - **AWS secret access key** – Half of the AWS account user credentials, along with the AWS access key ID.
   - **Default region name** – Name of a region, such as "us-west-2" you want to set as default. If you don't set a default region, every command must specify a region using the `--region` parameter. See a list of available regions for GameLift at AWS Regions and Endpoints.
   - **Default output format** – Format to receive API responses. Options include "json", "text" or "table". If you don't enter a default format, the CLI processes your requests but doesn't display any results.

Integrating the Amazon GameLift SDKs into Your Games

The Amazon GameLift SDKs provide the libraries needed to enable your game clients and servers to communicate with the Amazon GameLift service. The SDKs are available as part of the Amazon Lumberyard game engine download or can be downloaded separately. For more details on the SDKs and where to get them, see Amazon GameLift SDKs (p. 7).

The topics in this section describe how game clients and servers interact with the GameLift service, and provide instruction on adding GameLift functionality to your game clients and servers.

Test Out Amazon GameLift

Before you prepare your own game to use GameLift, you can experiment with the service and console tools using the five-click sample wizard. This wizard uses a sample game to quickly step through the tasks of uploading and deploying a game server and connecting to it with a game client. Once you have the sample game server deployed and are connected, you can browse the GameLift console tools, view or edit configuration settings, and see GameLift's metrics and other tools in action. To access the sample wizard, sign into the Amazon GameLift console and select Sample Game from the Amazon GameLift menu.

Topics

Integrating your Game Server for Amazon GameLift

The topics in this section describe how to integrate Amazon GameLift into a multiplayer game server.

Adding Amazon GameLift to your game server is Step 2 on the Integration Steps (p. 2) roadmap. The integration topics in this section assume that you've created an AWS account and have an existing game server project.

Topics

- Amazon GameLift Server API (C++) Reference (p. 15)

Add Amazon GameLift to Your Game Server

Your game server needs to interact with the Amazon GameLift service once it is deployed and running as multiple server processes on a GameLift fleet. The code you add enables each server process to communicate with the GameLift service. Server processes must be able to respond to certain events triggered by the GameLift service and keep GameLift informed about player activity and server process status. See this complete description of GameLift interactions (p. 30).

Use the GameLift Server SDK for C++ to add GameLift functionality to your game server. For complete information on the Server SDK, see the Amazon GameLift Server API (C++) Reference (p. 15). For details on how to get the AWS SDK and version information, see Amazon GameLift SDKs (p. 7).

To integrate GameLift into your game server, add the GameLift Server SDK for C++ to your game server project and build the following functionality:

- Set up a new server process
- Report server process health
- Start a game session
- Validate a new player
- Report a dropped connection/player session ending
- Stop a game session
- Shut down a server process

Prepare a Server Process

Add code to initialize a GameLift client and notify the GameLift service that the server is ready to host a game session. This code should run automatically before any GameLift-dependent code, such as on launch.

1. Initialize the Server SDK. Call `InitSDK() (p. 16)`.
2. Notify GameLift that a game server process is ready to host a game session. Each server process started on a GameLift instance must call `ProcessReady() (p. 17)`; each call may include the following information: (1) a port number allowing access to the server process, (2) location of the log paths for game sessions hosted by the server process, and (3) the references to three callback functions used by GameLift to trigger certain actions on the server process.

   GameLift invokes these callbacks in the following circumstances:

   - `onHealthCheck` is called regularly to request a health status report from the server process.
   - `onStartGameSession` is called when the GameLift service receives request to start a new game session (`CreateGameSession()`).
   - `onProcessTerminate` is called when GameLift service needs to force the server process to terminate, allowing the server process to shut down gracefully.

   Your game server code must implement these callback functions. The names are assigned for convenience; reference the actual names in this `ProcessReady()` call. For more information on implementing these functions, see the sections that follow.

You can have multiple server processes running on a fleet instance. Once one server process on an instance calls `ProcessReady()` successfully, the GameLift service sets the instance's status to ACTIVE.

Report Server Process Health

Add code to implement the callback function `onHealthCheck()`. This function is invoked by the GameLift service regularly to collect health metrics from the server process. The server process's response to a health check is a binary: healthy or unhealthy. When implementing this callback function, do the following:

- Evaluate the status of the server process using whatever measures make sense for your game. For example, you might report the server process as unhealthy if any external dependencies have failed or if metrics such as memory capacity fall outside a defined limit.
- Complete the health evaluation and respond to the callback within 60 seconds. If the GameLift service does not receive a response in that time, it will automatically consider the server process to be unhealthy.
- Return a boolean value: true for healthy, false for unhealthy.

If you do not implement a health check callback, the GameLift service considers the server process to be healthy unless the process is not responding, in which case it is considered unhealthy.

Server process health is used by GameLift to efficiently end unhealthy processes and free up resources. If a server process is reported unhealthy or does not respond for three consecutive minutes, the GameLift service can shut down the process and start a new one. Metrics on a fleet's server process health is collected and viewable on the GameLift console.

Start a Game Session

Add code to implement the callback function `onStartGameSession`. This function is invoked by the GameLift service in response to receiving a `CreateGameSession()` request from a game client.

The `onStartGameSession` function takes a GameSession object, provided by the GameLift service, as an input parameter. This object contains the game session ID and other information that defines the requested game session. The function should accomplish the following tasks:

- Perform whatever actions are needed to create a new game session. The new game session should reflect the `GameSession` object, including creating slots for the specified maximum number of players and referencing the game session name and ID. The GameLift service provides the same game session information to the game client.
- Process the game property values specified by the game client in its request. The game properties are contained in the `GameSession` object.
- At some point after the new game session is ready to accept players, the server process must call the server API action `ActivateGameSession() (p. 20)`. In response to a successful call, the GameLift service changes the game session status to ACTIVE.

Validate a New Player

Add code to verify a player connection request with the GameLift service. This code should run whenever a new player attempts to connect to the server process and before accepting the connection.

Connection requests from a game client must reference a player session ID. This ID was issued by the GameLift service and used to reserve a player slot in the game session (in response to a game client call to `CreatePlayerSession()`). The game server must call `AcceptPlayerSession() (p. 22)` with the referenced player session ID to verify that the connection request is coming from the same game client that reserved the slot.

Once the player session ID is validated by the GameLift service, the server process can accept the connection and allow the player to join the game session. If the player session ID is not validated by the GameLift service, the server process should deny the connection.

Report a Player Session Ending

Add code to notify the GameLift service when a player disconnects from the game session. This code should run whenever the server process detects a dropped connection.

In the code handling the dropped connection, add a call to the server API action `RemovePlayerSession() (p. 23)` with the player session ID associated with the dropped connection. This notification enables the GameLift service to accurately track the number of current players and available slots in the game session.

Stop a Game Session

Add code to notify the GameLift service when a game session is ending. The notification enables the GameLift service to accurately track a server process's availability for new game sessions. This code should be added to the normal game session ending process.

At the end of the code to stop a game session, add a call to the server API action `TerminateGameSession() (p. 20)`. On successful receipt of this notification, the GameLift service changes the game session status to TERMINATED and may immediately start a new game session.

> **Note**
> If stopping a game session will be immediately followed by shutting down the server process, you can call the server API action `ProcessEnding() (p. 19)`, which terminates both the game session and the server process.

Shut Down a Server Process

Add code to notify the GameLift service when a server process will shut down. Once called, the server process can safely end. Shut down may be initiated by the server process or in response to the GameLift service invoking the `onProcessTerminate()` callback function.

Add the following code:

- At the end of the game server code that shuts down a server process, add a call to the server API action `ProcessEnding() (p. 19)`. On receipt of this notification, the GameLift service changes the server process status to TERMINATED and recycles the instance's resources as needed.

- Implement the callback function `onProcessTerminate()`. This function simply needs to call the game server termination code, which now includes the call to `ProcessEnding()`. This callback function is invoked by the GameLift service prior to terminating the instance hosting the server process. A common reason for GameLift service to invoke this call is to shut down an unhealthy server process. After receiving this call, the server process has five minutes to gracefully disconnect players, preserve game state data, and perform other cleanup tasks. If the server process calls `ProcessEnding()` before five minutes has elapsed, the GameLift service may immediately shut down the process.

Amazon GameLift Server API (C++) Reference

This GameLift C++ Server API reference can help you prepare your multiplayer game for use with Amazon GameLift. For details on the integration process, see Add Amazon GameLift to Your Game Server (p. 13).

This API is defined in `GameLiftServerAPI.h`, `LogParameters.h`, and `ProcessParameters.h`.

GetSdkVersion()

Returns the current version number of the SDK built into the server process.

Syntax

```
AwsStringOutcome GetSdkVersion();
```

Parameters

This action has no parameters.

Return Value

If successful, returns the current SDK version as an `AwsStringOutcome` object. The returned string includes the version number only (ex. "3.0.0"). If not successful, returns an error message.

Example

```
Aws::GameLift::AwsStringOutcome SdkVersionOutcome =
    Aws::GameLift::Server::GetSdkVersion();
```

InitSDK()

Initializes the GameLift SDK. This method should be called on launch, before any other GameLift-related initialization occurs.

Syntax

```
InitSDKOutcome InitSDK();
```

Parameters

This action has no parameters.

Return Value

If successful, returns an `InitSDKOutcome` object containing either a pointer to an internal server state, for use with `InitSDKWithExisting() (p. 16)`, or an error code.

Example

```
Aws::GameLift::Server::InitSDKOutcome initOutcome =
    Aws::GameLift::Server::InitSDK();
```

InitSDKWithExisting()

Initializes the GameLift SDK using an existing server state. Use this function if you're invoking the GameLift API across DLL boundaries. This method should be called on launch, before any other GameLift-related initialization occurs.

Syntax

```
InitSDKOutcome InitSDKWithExisting(
     Aws::GameLift::Internal::GameLiftServerState* existingState);
```

Parameters

existingState
Pointer to an internal server state, returned when initializing GameLift.

Required: Yes

Return Value

If successful, returns an `InitSDKOutcome` object containing either a pointer to an internal server state or an error code.

Example

This example illustrates initializing the SDK using a server state returned from a previous call to either this action or `InitSDK() (p. 16)`.

```
initOutcome = Aws::GameLift::Server::InitSDKWithExisting(initOutcome);
```

ProcessReady()

Notifies the GameLift service that the server process is ready to host game sessions. This method should be called once the server process is ready to host a game session. The parameters specify the names of callback functions for GameLift to call in certain circumstances. Game server code must implement these functions.

This call is synchronous. To make an asynchronous call, use `ProcessReadyAsync() (p. 18)`.

Syntax

```
GenericOutcome ProcessReady(
                const Aws::GameLift::Server::ProcessParameters
 &processParameters);
```

Parameters

processParameters
A ProcessParameters (p. 23) object containing information related to the server process.

Required: Yes

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

This example illustrates both the `ProcessReady() (p. 17)` call and callback function implementations.

```
// Set parameters and call ProcessReady
Aws::String serverLog("serverOut.log");        //Example of a log file
Aws::Vector<Aws::String> logPaths;
logPaths.push_back(serverLog);

int listenPort = 9339;
```

```
Aws::GameLift::Server::ProcessParameters processReadyParameter =
 Aws::GameLift::Server::ProcessParameters(
    std::bind(&Server::onStartGameSession, this, std::placeholders::_1),
    std::bind(&Server::onProcessTerminate, this),
    std::bind(&Server::OnHealthCheck, this),
    listenPort,
    Aws::GameLift::Server::LogParameters(logPaths));

Aws::GameLift::GenericOutcome outcome =
   Aws::GameLift::Server::ProcessReady(processReadyParameter);

// Implement callback functions
void Server::onStartGameSession(Aws::GameLift::Model::GameSession
 myGameSession)
{
   // game-specific tasks when starting a new game session, such as loading
 map
   GenericOutcome outcome =
       Aws::GameLift::Server::ActivateGameSession (maxPlayers);
}

void Server::onProcessTerminate()
{
   // game-specific tasks required to gracefully shut down a game session,
   // such as notifying players, preserving game state data, and other
 cleanup
   GenericOutcome outcome = Aws::GameLift::Server::ProcessEnding();
}

bool Server::onHealthCheck()
{
    bool health;
    // complete health evaluation within 60 seconds and set health
    return health;
}
```

ProcessReadyAsync()

Notifies the GameLift service that the server process is ready to host game sessions. This method should be called once the server process is ready to host a game session. The parameters specify the names of callback functions for GameLift to call in certain circumstances. Game server code must implement these functions.

This call is asynchronous. To make a synchronous call, use `ProcessReady() (p. 17)`.

Syntax

```
GenericOutcomeCallable ProcessReadyAsync(
                const Aws::GameLift::Server::ProcessParameters
 &processParameters);
```

Parameters

processParameters

A ProcessParameters (p. 23) object containing information related to the server process.

Required: Yes

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

```
// Set parameters and call ProcessReady
Aws::String serverLog("serverOut.log");        //Example of a log file
Aws::Vector<Aws::String> logPaths;
logPaths.push_back(serverLog);

int listenPort = 9339;

Aws::GameLift::Server::ProcessParameters processReadyParameter =
 Aws::GameLift::Server::ProcessParameters(
    std::bind(&Server::onStartGameSession, this, std::placeholders::_1),
    std::bind(&Server::onProcessTerminate, this),
    std::bind(&Server::OnHealthCheck, this),
    listenPort,
    Aws::GameLift::Server::LogParameters(logPaths));

Aws::GameLift::GenericOutcomeCallable outcome =
   Aws::GameLift::Server::ProcessReadyAsync(processReadyParameter);

// Implement callback functions
void onStartGameSession(Aws::GameLift::Model::GameSession myGameSession)
{
   // game-specific tasks when starting a new game session, such as loading
 map
   GenericOutcome outcome = Aws::GameLift::Server::ActivateGameSession
 (maxPlayers);
}

void onProcessTerminate()
{
   // game-specific tasks required to gracefully shut down a game session,
   // such as notifying players, preserving game state data, and other
 cleanup
   GenericOutcome outcome = Aws::GameLift::Server::ProcessEnding();
}

bool onHealthCheck()
{
    // perform health evaluation and complete within 60 seconds
    return health;
}
```

ProcessEnding()

Notifies the GameLift service that the server process is shutting down. The call triggers GameLift to change the instance's status from ACTIVE to TERMINATING. This method should be called after all other cleanup tasks, including shutting down all active game sessions. GameLift waits at least 30 seconds after receiving this call before terminating the instance.

Syntax

```
GenericOutcome ProcessEnding();
```

Parameters

This action has no parameters.

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

```
Aws::GameLift::GenericOutcome outcome =
 Aws::GameLift::Server::ProcessEnding();
```

ActivateGameSession()

Notifies the GameLift service that the server process has activated a game session and is now ready to receive player connections. This action should be called as part of the `onStartGameSession()` callback function, after all game session initialization has been completed.

Syntax

```
GenericOutcome ActivateGameSession();
```

Parameters

This action has no parameters.

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

This example shows `ActivateGameSession()` being called as part of the `onStartGameSession()` callback function.

```
void onStartGameSession (Aws::GameLift::Model::GameSession myGameSession)
{
   // game-specific tasks when starting a new game session, such as loading
 map
   GenericOutcome outcome =
    Aws::GameLift::Server::ActivateGameSession ();
}
```

TerminateGameSession()

Notifies the GameLift service that the server process has shut down the game session. (Currently, each server process hosts only one game session at a time, so there's no need to specify which session.) This action should be called at the end of the game session shutdown process. After calling this action, the server process should either call `ProcessReady() (p. 17)` to signal availability to host a new game session, or call `ProcessEnding() (p. 19)` to shut down the server process and terminate the instance.

Syntax

```
GenericOutcome TerminateGameSession();
```

Parameters

This action has no parameters.

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

This example illustrates the server process shutting down the current game session, notifying GameLift that the game session has ended, and signalling the server's availability to host a new game session.

```
// game-specific tasks required to gracefully shut down a game session,
// such as notifying players, preserving game state data, and other cleanup

Aws::GameLift::GenericOutcome outcome =
    Aws::GameLift::Server::TerminateGameSession();
Aws::GameLift::GenericOutcome outcome =
    Aws::GameLift::Server::ProcessReady(onStartGameSession,
 onProcessTerminate);
```

UpdatePlayerSessionCreationPolicy()

Updates the current game session's ability to accept new player sessions. A game session can have its player session creation policy set to either accept or deny all new player sessions (see the `UpdateGameSession()` action in the *Amazon GameLift Service API Reference*).

Syntax

```
GenericOutcome UpdatePlayerSessionCreationPolicy(
     Aws::GameLift::Model::PlayerSessionCreationPolicy
 newPlayerSessionPolicy);
```

Parameters

newPlayerSessionPolicy
String value indicating whether or not the game session accepts new players.

Type: Aws::GameLift::Model::PlayerSessionCreationPolicy enum. Valid values include:

- **ACCEPT_ALL** – Accept all new player sessions.
- **DENY_ALL** – Deny all new player sessions.

Required: Yes

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

```
Aws::GameLift::GenericOutcome outcome =
```

```
Aws::GameLift::Server::UpdatePlayerSessionCreationPolicy(Aws::GameLift::Model::PlayerSessi
```

GetGameSessionId()

Retrieves the ID of the game session currently being hosted by the server process, if the server process is active.

Syntax

```
AwsStringOutcome GetGameSessionId();
```

Parameters

This action has no parameters.

Return Value

If successful, returns the game session ID as an `AwsStringOutcome` object. If not successful, returns an error message.

Example

```
Aws::GameLift::AwsStringOutcome sessionIdOutcome =
    Aws::GameLift::Server::GetGameSessionId();
```

AcceptPlayerSession()

Notifies the GameLift service that a player with the specified player session ID has connected to the server process and requests validation. GameLift verifies that the player session ID is valid—that is, that the ID was issued to a player in response to a join request. Once validated, GameLift changes the status of the player slot from RESERVED to ACTIVE.

Syntax

```
GenericOutcome AcceptPlayerSession(
     const Aws::String& playerSessionId);
```

Parameters

playerSessionId
: Unique ID issued by the GameLift service to a game client in response to a call to the AWS SDK GameLift API action `CreatePlayerSession`. The client passes this ID when connecting to the server process.

  Type: Aws::String

  Required: Yes

Return Value

Returns a generic outcome consisting of success or failure with an error message. The error `UNEXPECTED_PLAYER_SESSION` indicates that the player session ID is invalid.

Example

This example illustrates a function for handling a connection request, including validating and rejecting invalid player session IDs.

```
void ReceiveConnectingPlayerSessionID (Connection& connection, const
 Aws::String& playerSessionId){
    Aws::GameLift::GenericOutcome connectOutcome =
        Aws::GameLift::Server::AcceptPlayerSession(playerSessionId);
    if(connectOutcome.IsSuccess())
    {
        connectionToSessionMap.emplace(connection, playerSessionId);
        connection.Accept();
    }
    else
    {
        connection.Reject(connectOutcome.GetError().GetMessage();
    }
}
```

RemovePlayerSession()

Notifies the GameLift service that a player with the specified player session ID has disconnected from the server process. In response, GameLift changes the player slot to available, which allows it to be assigned to a new player.

Syntax

```
GenericOutcome RemovePlayerSession(
     const Aws::String& playerSessionId);
```

Parameters

playerSessionId

Unique ID issued by the GameLift service to a game client in response to a call to the client API action `CreatePlayerSession`. The client passes this ID when connecting to the server process.

Type: Aws::String

Required: Yes

Return Value

Returns a generic outcome consisting of success or failure with an error message.

Example

```
Aws::GameLift::GenericOutcome connectOutcome =
    Aws::GameLift::Server::RemovePlayerSession(playerSessionId);
```

ProcessParameters

This data type contains the set of parameters sent to the GameLift service in a `ProcessReady()` call.

Contents

port
Port number this server process is listening. This value must fall into one of the port ranges configured for the fleet in use. This port setting is provided to game clients and other applications in the GameSession and PlayerSession objects.

Type: int

Required: Yes

logParameters
Object containing a list of directory paths indicating where game session logs are saved.

Type: Aws::GameLift::Server::LogParameters (p. 24)

Required: No

onStartGameSession
Name of callback function that the GameLift service calls to activate a new game session. GameLift calls this function in response to the client request `CreateGameSession`. The callback function takes a `GameSession` object (defined in the *Amazon GameLift Service API Reference*).

Type: const std::function<void(Aws::GameLift::Model::GameSession)> onStartGameSession

Required: Yes

onProcessTerminate
Name of callback function that the GameLift service calls to force the server process to shut down. After calling this function, GameLift waits five minutes for the server process to shut down and respond with a `ProcessEnding() (p. 19)` call before it shuts down the server process.

Type: std::function<void()> onProcessTerminate

Required: No

onHealthCheck
Name of callback function that the GameLift service calls to request a health status report from the server process. GameLift calls this function every 60 seconds. After calling this function GameLift waits 60 seconds for a response, and if none is received. records the server process as unhealthy.

Type: std::function<bool()> onHealthCheck

Required: No

LogParameters

This data type is used to identify where GameLift should look for game session log files to upload and stor at the end of a game session. This information is communicated to the GameLift service in a `ProcessReady()` call.

Contents

logPaths
Array of strings identifying directory paths to game server log files, including game session logs.

Log files are stored on the GameLift instance running the server process, in the directory and using file names designated by your game server. (Given that instances run multiple server processes, often simultaneously, you should consider whether or not to assign unique log file names for each game session.) Each log path must point to a location in your root game build directory. For example, if your game build is packaged into a directory called "MyGame", your log path might be `MyGame\sessionlogs\`. When the game build is deployed to a GameLift

instance, the log path would be `c:\game\MyGame\sessionLogs` (on Windows) or `/local/game/MyGame/sessionLogs`.

Type: Aws::Vector<Aws::String>

Required: No

Integrating your Game Client for Amazon GameLift

The topics in this section describe the Amazon GameLift functionality you can add to a game client or game service that handles the following tasks:

- Requests information about active game sessions from the GameLift service.
- Joins a player to an existing game session.
- Creates a new game session.
- Changes metadata about an existing game session.

Adding Amazon GameLift to your multiplayer game client is Step 5 in the Integration Steps (p. 2). The following instructions assume that you've created an AWS account, generated a GameLift-enabled game server and uploaded it to GameLift, and used GameLift tools (such as the Amazon GameLift console) to create and configure a virtual fleet to host your game sessions. When adding GameLift to your game client, you must be able to provide AWS account credentials and specify a fleet to be used with the client.

For more information on how game clients interact with the Amazon GameLift service and game servers running on GameLift, see Amazon GameLift and Game Client/Server Interactions (p. 30).

Topics

Add Amazon GameLift to Your Game Client

Use the AWS SDK with Amazon GameLift API to add functionality to your game client. For details on how to get the AWS SDK and version information, see Amazon GameLift SDKs (p. 7). The following reference information is available:

- *Amazon GameLiftService API Reference* – This guide describes the low-level service API calls for GameLift-related actions.
- *AWS SDK for C++ Reference* – This guide documents C++ calls to the AWS SDK. Documentation for other language versions of the AWS SDK with GameLift are also available (see Amazon GameLift SDKs (p. 7).

To integrate GameLift into your game client, add the AWS SDK to your game client project and build the following functionality:

- Initialize and set up a GameLift client.
- Find a game session to join or create a new one.

- Join a game session.

Initialize and Set Up a GameLift Client

Add code to initialize a GameLift client and store some key settings for use with the game client. This code needs to be located so that it runs before any GameLift-dependent code, such as on launch.

1. Decide whether or not to use the default client configuration or create custom settings.
   For custom settings, you must create a custom `ClientConfiguration` object. See `AWS ClientConfiguration` (C++) for object structure and the default settings.

   Set a region and endpoint. It is important that you configure your game client to use the correct region and endpoint. GameLift deploys fleets by region, and game clients can only access fleets in the region specified in the client configuration. See this list of AWS regions supported by GameLift for names and endpoints. The default region is US East (N. Virginia), so if you're using any other region you must create a custom configuration.
2. Initialize a GameLift client. Call Aws::GameLift::GameLiftClient() (C++) using either a client configuration with the default settings or a custom configuration.
3. Add a mechanism to generate a unique identifier for each player. GameLift requires a unique player ID to connect to a game session. For more details, see Generating Identifiers for Your Players.
4. Collect and store the following information to use when contacting GameLift:

   - **Target fleet** – Each game client must specify one fleet to connect to. The client can create new game sessions or access existing game sessions on any instance in the specified fleet. (The target fleet must be deployed in the region specified in the client configuration.) You have two options for identifying a target fleet: (1) Specify the fleet's ID, or (2) specify the ID of an alias pointing to the fleet. Fleet aliases are highly useful in that you can use them to change fleets without having to issue a game client update.
   - **Player ID** – This is the unique identifier generated in step 3.
   - **AWS credentials** – All calls to the GameLift service must provide credentials for the AWS account that hosts the game. This is the account you used to set up your GameLift fleets, and you should have created an IAM user or user group for players with a permissions policy. You need to create an Aws::Auth::AWSCredentials (C++) object containing an IAM access key and secret key for the player user group. For help finding the keys, see Managing Access Keys for IAM Users.

Find or Create a Game Session

Add code to discover existing game sessions, create a new game session, or both. See How Players Connect to Games (p. 5) for more information on game sessions.

1. **Find existing game sessions.** Create a mechanism to retrieve information on game sessions currently running on the target fleet.

   To find active game sessions, call `SearchGameSessions`. You can specify search criteria or leave empty to retrieve information on all active sessions. This call returns a `GameSession` object, with game session characteristics, for each active game session that matches your search request.

   Filter returned game sessions and present options for players to join. For example, you may want to filter sessions based on the following characteristics:

   - Exclude game sessions that are not accepting new players: `PlayerSessionCreationPolicy = DENY_ALL`

- Exclude game sessions that are full: `CurrentPlayerSessionCount = MaximumPlayerSessionCount`
- Choose game sessions based on length of time the session has been running: evaluate `CreationTime`
- Evaluate game session properties

2. **Create new game sessions.** Create a mechanism to allow players to start a new game sessions to join.

   To create a new session, call `CreateGameSession`. When creating a new game session, specify a session name and the maximum number of concurrent players to allow.

   When creating a new session, you also have the option to pass a set of properties to the server process. The `GameProperty` object is simply an array of key–value pairs in which you define both the keys, or types, and a set of allowed values that are meaningful to your game. This assumes you have set up your game server to recognize and act on these properties when received. For example, you might use game properties to load a certain game map or apply a set of rules to the game session.

Whether you locate an existing game session to join or create a new one, you'll have a `GameSession` object. Use the following API actions to manage game sessions:

- `CreateGameSession()` – Create new game sessions with custom properties and maximum player counts.
- `DescribeGameSessionDetails()` – Get game session metadata for one or multiple game sessions, including connection information, status, and current player count.
- `UpdateGameSession()` – Change a game session's metadata and settings as needed.
- `SearchGameSessions()` – Find game sessions that match certain characteristics.
- `GetGameSessionLogUrl` – Access stored game session logs.

Join a Game Session

Add code to request access to a game session and connect to the session. Once the client successfully requests a player session, the client can connect to the server process IP address and port.

1. **Claim a player slot.** Create a mechanism to reserve a player slot in a specified game session.

   Ask to join a game session by calling `CreatePlayerSession` and specifying a game session ID (included in a `GameSession` object). The GameLift service verifies that the game session can be joined (that is, it accepts new players and has not reached its maximum number of players), and reserves a player slot for the requester. If successful, a `PlayerSession` object is created and returned, with information the game client needs to connect to the server process that is hosting the game session.
2. **Connect to a game session.** Create a mechanism to connect a player to the server process that is hosting the game session.

   Connect to the server process using the IP address and port from either the `GameSession` object or the `PlayerSession` object. Specify the player session ID value to claim the reserved slot. The game server verifies with the GameLift service that the player trying to connect is entitled to the reserved slot. If successful, the player joins the game session.

   Once connected, the game client and server communicate directly with each other, without involving GameLift. This framework minimizes the amount of latency in gameplay. When the client disconnects from the game session, the server process automatically detects the dropped connection and informs the GameLift service, which tracks the availability of player slots.

Once you've created a player session you'll have a `PlayerSession` object. Use the following API actions to manage player sessions:

- `CreatePlayerSession()` – Create one new player session and reserve a player slot.
- `CreatePlayerSessions()` – Create multiple player sessions and reserve player slots for each.
- `DescribePlayerSessions()` – Get metadata for one or multiple player sessions.

Generate Player IDs

Amazon GameLift uses a player session to represent a player connected to a game session. A player session must be created each time a player connects to a game session. When a player leaves a game, the player session ends and is not reused.

Amazon GameLift fees are partly based on the number of your end users that connect to Amazon GameLift. This is the daily active user (DAU) portion of your bill. For more information, see the Amazon GameLift pricing page.

To allow us to accurately calculate your fees and in order to receive an accurate daily active user count from Amazon GameLift, you must provide a unique, non-personally identifiable player ID for each player. For example, you must not use email addresses, phone numbers, real names, or public screen names for the player ID.

Amazon GameLift provides a file called `Lobby.cpp` in the MultiplayerProject sample that demonstrates how to generate a new, random ID number for every player in every new game session. You are not required to use the sample code; we provide it as an example.

The following sample code in `Lobby.cpp` will randomly generate unique player IDs:

```
bool includeBrackets = false;
bool includeDashes = true;
string playerId = AZ::Uuid::CreateRandom().ToString<string>(includeBrackets,
 includeDashes);
```

If you use this sample code to generate unique player IDs for your game, a player that starts their game client and logs into the game server three times in one day will appear as three DAU for billing purposes.

You can also rewrite the code to persist your own unique, non-personally identifiable player IDs.

Prepare Your Game Client in Amazon Lumberyard

All game clients must be configured to enable communication with the Amazon GameLift service, including specifics on which fleets to use, access credentials, how to connect, etc. The simplest method is to create a batch file that sets the console variables listed below.

> **Tip**
> You don't need to have a game ready to start experimenting with GameLift. The GameLift console offers a quick sample setup that gets you up and running with a sample game server and client in five easy steps. In addition, the GameLift Getting Started tutorials, in text and video format, walk you through each step in the process of creating and uploading a build, setting up a fleet, creating game sessions and connecting a client. The tutorials use a sample multiplayer game, which is included in the Lumberyard download.

To prepare the game client

1. In your batch file, set the following console variables to launch the game client. These variables have been added to `\dev\Code\CryEngine\CryNetwork\Lobby\LobbyCvars`

- `gamelift_aws_access_key` = part of the IAM security credentials (p. 9) for a user with "player" access in your AWS account
- `gamelift_aws_secret_key` = part of the IAM security credentials (p. 9) for a user with "player" access in your AWS account
- `gamelift_fleet_id` = Unique ID of an active fleet to connect to
- `gamelift_alias_id` = Unique ID of an alias pointing to a fleet to connect to
- (Optional) `gamelift_endpoint` = Amazon GameLift server endpoint; the default value is `gamelift.us-west-2.amazonaws.com`
- (Optional) `gamelift_aws_region` = AWS region name; default value is `us-west-2`
- (Optional) `gamelift_player_id` = ID that you generate to uniquely identify a player (p. 28)

2. Add the following command to launch the server browser:

   Follow this pattern when using a GameLift fleet ID (`gamelift_fleet_id`):

```
.\Bin64\[your game executable] +gamelift_fleet_id [your fleet ID]
 +gamelift_aws_region us-west-2 +gamelift_aws_access_key [your AWS access
 key] +gamelift_aws_secret_key [your AWS secret key] +sv_port 64091 +map
 [map name]
```

   Follow this pattern when using a GameLift alias ID (`gamelift_alias_id`):

```
.\Bin64\[your game executable] +gamelift_alias_id  [your alias ID]
 +gamelift_aws_region us-west-2 +gamelift_aws_access_key [your AWS access
 key] +gamelift_aws_secret_key [your AWS secret key] +sv_port 64091 +map
 [map name]
```

Customize the Amazon GameLift SDK

You can customize the Amazon GameLift SDK to work with your own tools and integrations.

Custom Memory Management

The AWS native SDK supports custom memory managers.

To use a custom memory manager with the Amazon GameLift SDK

1. Implement `Aws::Utils::Memory::MemorySystemInterface` with hooks into your memory manager. This interface has four methods: `Begin`, `End`, `AllocateMemory`, and `FreeMemory`.
2. Before you begin using AWS native SDK or Amazon GameLift SDK features, call `Aws::Utils::Memory::InitializeMemorySystem(myMemoryInterface)`.
3. Set the `AWS_CUSTOM_MEMORY_MANAGEMENT` preprocessor flag to `1`.

Logging Integration

The AWS native SDK supports integration with your preferred log system.

To use your own log system

1. Implement `Aws::Utils::Logging::LogSystemInterface`.
2. Call `Aws::Utils::Logging::InitializeAWSLogging(myLogInterface)`.

GameLift Client Configuration

You can preconfigure a GameLift client to use with the Amazon GameLift SDK.

To create a custom GameLift client

1. Create an `Aws::Client::ClientConfiguration` object and set the properties with your preferred values.
2. Use the `ClientConfiguration` object when calling `Aws::GameLift::Client::Initialize`.

Direct Calls to GameLift Client

You can extend the Amazon GameLift SDK functionality by making calls directly to the GameLift client.

To make a custom call to the GameLift client

1. Call `Client::GetGameLiftClient()`.
2. Create a request object that corresponds to the type of request you want to make.
3. Use the request object to call the method that you want to execute on the client.

You will receive a response object that corresponds to the type of request you made.

Amazon GameLift and Game Client/Server Interactions

This topic describes the interactions between a client app, a game server, and the Amazon GameLift service. See also the Amazon GameLift–Game Server/Client Interactions (p. 33) diagram.

Setting up a new server process

1. The **GameLift service** launches a new server process on an Amazon Elastic Compute Cloud (Amazon EC2) instance.
2. The **server process**, as part of the launch process, calls these server API actions:
   - `InitSDK()` to initialize the server SDK.
   - `ProcessReady()` to communicate readiness to accept a game session and specify connection port and location of game session log files.

   It then waits for a callback from the GameLift service.
3. The **GameLift service** changes the status of the EC2 instance to ACTIVE, with 0 game sessions and 0 players.
4. The **GameLift service** begins calling the `onHealthCheck` callback regularly while the server process is active. The server process must report either healthy or not healthy within one minute.

Creating a game session

1. The **Client app** calls the client API action `CreateGameSession()`.
2. The **GameLift service** searches for an active server with 0 game sessions. When found, it does the following:
   - Creates a new `GameSession` object, using the port setting reported by the server process in `ProcessReady()`, and sets its status to ACTIVATING.

   - Responds to the client app request with the `GameSession` object.
   - Invokes the `onStartGameSession` callback on the server process, passing the `GameSession` object.
3. The **server process** runs the `onStartGameSession` callback function. When ready to accept player connections, the server process calls the server API action `ActivateGameSession()` and waits for player connections.
4. The **GameLift service** changes the `GameSession` status to ACTIVE.

Adding a player to a game session:

1. The **Client app** calls the client API action `CreatePlayerSession()` with a game session ID.
2. The **GameLift service** checks the game session status (must be ACTIVE), and looks for an open player slot in the game session. If a slot is available, it does the following:
   - Creates a new `PlayerSession` object and sets its status to RESERVED.
   - Responds to the client app request with the `PlayerSession` object.
3. The **Client app** connects directly to the server process with the player session ID.
4. The **server process** calls the Server API action `AcceptPlayerSession()` to validate the player session ID and either accepts or rejects the connection. If accepted, the server process notifies the GameLift service.
5. The **GameLift service** does one of the following:
   - If the connection is accepted, sets the `PlayerSession` status to ACTIVE.
   - If no response is received within 60 seconds of the client app's original `CreatePlayerSession()` call, changes the `PlayerSession` status to TIMEDOUT and reopens the player slot in the game session.

Removing a player from a game session

1. The **Client app** disconnects from the server process.
2. The **server process** detects the lost connection and calls the server API action `RemovePlayerSession()`.
3. The **GameLift service** changes the `PlayerSession` status to COMPLETED and reopens the player slot in the game session.

Shutting down a game session

1. The **server process** calls the server API action `TerminateGameSession()`.
2. The **GameLift service** does the following:
   - Changes the `GameSession` status to TERMINATED.
   - Uploads game session logs to Amazon Simple Storage Service (Amazon S3).
   - Updates fleet utilization to indicate the server is idle(0 game sessions, 0 players).

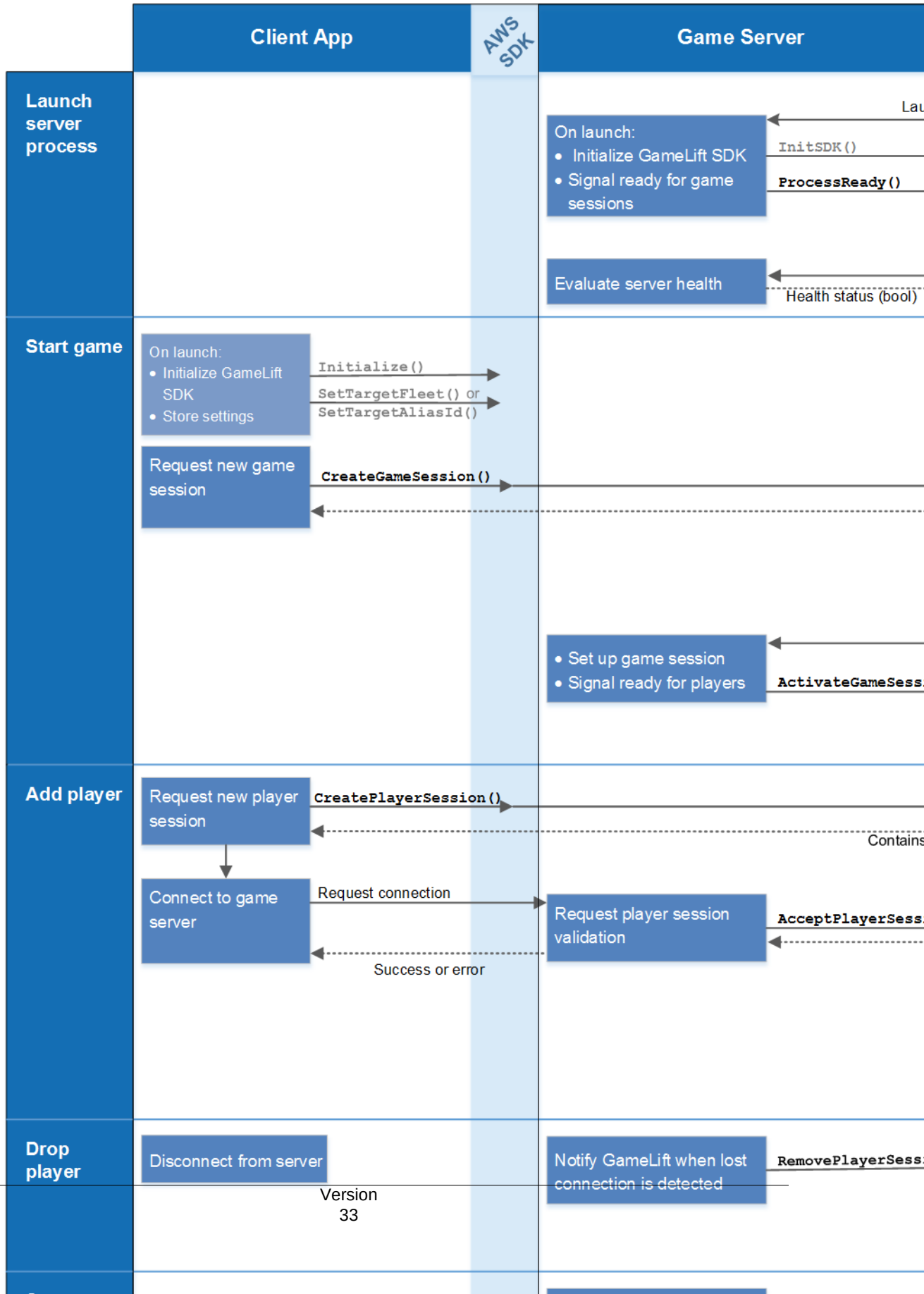
Terminating a server process

1. The **server process** does the following:
   - Runs process to gracefully terminate the server process.
   - Calls the server API action `ProcessEnding()` to inform the GameLift service.
2. The **GameLift service** does the following:
   - Uploads game session logs (if any) to Amazon S3.
   - Changes the server process status to TERMINATED.
   - Recycles instance resources based on the fleet's runtime configuration.

Responding to a shutdown request

1. The **GameLift service** invokes the server process's `onProcessTerminate` callback. This call is used to shut down a server process that has reported unhealthy or not responded with health status for three consecutive minutes.
2. The **server process** runs the `onProcessTerminate` callback function, which triggers the server's termination process, ending with a call to `ProcessEnding()`.
3. The **GameLift service** does the following, either in response to receiving the `ProcessEnding()` call or after five minutes:
   - Uploads game session logs (if any) to Amazon S3.
   - Changes the server process status to TERMINATED.
   - Recycles instance resources based on the fleet's runtime configuration.

Amazon GameLift–Game Server/Client Interactions



Uploading Your Game to Amazon GameLift

Before setting up computing resources to host your Amazon GameLift-enabled multiplayer game, you first need to create a game build and upload it to the GameLift service. A game build includes all the server executables and dependent files needed to run server processes and host game sessions. Once you've uploaded a build to GameLift, you can then create a fleet of computing resources to operate your game.

The topics in this section describe how to package your game build and how to use the AWS CLI tool to upload it to GameLift.

> **Tip**
> You don't need to have a game ready to start experimenting with GameLift. The GameLift console offers a quick sample setup that gets you up and running with a sample game server and client in five easy steps. In addition, the GameLift Getting Started tutorials, in text and video format, walk you through each step in the process of creating and uploading a build, setting up a fleet, creating game sessions and connecting a client. The tutorials use a sample multiplayer game, which is included in the Lumberyard download.

Topics

Package a Build

To prepare your Amazon GameLift-enabled game server for uploading to the Amazon GameLift service for deployment, you need to package all game server files into a build folder. This folder must include all components required to run your game servers. When deployed, the build will be copied to each instance in a fleet and used to run your game.

- **Game server binaries** – The binary files required to run a game server. A build can include binaries for multiple game servers, as long as they are built to run on the same platform (see supported platforms (p. 7)).

- **Dependencies** – Any dependent files required by your game server executables to run. Examples include assets, configuration files, and dependent libraries.
- **Installation instructions** – A script file to perform tasks required to configure a fleet instance for the game. This script is run once, immediately after the build files are copied to the instance. Installation tasks cannot require any user input. For Windows-based games, create an install batch file named "`install.bat`". For Linux games, create a shell script file named "`install.sh`". These files must be located at the root of the build folder.

You can use the Amazon Lumberyard Multiplayer project to experiment with creating a build package. For more information, see the Amazon GameDev tutorials for GameLift.

Upload a Build

Once your game build is packaged (p. 34), use the AWS Command Line Interface (AWS CLI) to upload the build to the Amazon GameLift service.

To upload your game build

1. **Send an upload request.** In a command line window, type the following command. Valid values for the `operating-system` parameter are `WINDOWS_2012` and `AMAZON_LINUX`. The build name and version parameters are optional, and can be changed later using the `update-build` command.

```
aws gamelift upload-build --operating-system [supported OS] --build-root
 [build path] --name [user-defined name of build] --build-version [user-
defined build number] --region [region name]
```

   Examples:

```
aws gamelift upload-build --operating-system AMAZON_LINUX --build-root ~/
mygame --name "My Game Nightly Build" --build-version "build 255" --region
 us-west-2
```

```
aws gamelift upload-build --operating-system WINDOWS_2012 --build-root
 "C:\mygame" --name "My Game Nightly Build" --build-version "build 255" --
region us-west-2
```

   Note
   Use the `--region` parameter only if you haven't configured the AWS CLI with a default region or if you want to specify a different region.

   In response to your upload request, the GameLift service returns a unique build ID (such as `build-75bf99cd-2dd8-2039-8074-ab24da1f80e4`). Upload time depends on the size of your game files and the connection speed.

2. **Check upload status.** Track the progress of your build upload request using either the GameLift console or the AWS CLI. The AWS CLI command is as follows:

```
aws gamelift describe-build --build-id [build ID returned with the upload
 request] --region [region name]
```

   Example:

```
aws gamelift describe-build --build-id "build-75bf99cd-2dd8-2039-8074-
ab24da1f80e4" --region us-west-2
```

In response to your request, the GameLift service returns a set of metadata about the build, including status, size of the uploaded files, and a creation time stamp. A build upload in progress has a status of **Initialized** and a build size of zero. Once an upload is completed successfully, it moves to a **Ready** status. For more information, see `DescribeBuild()` in the GameLift API reference.

Once a build reaches a **Ready** status, you can deploy it with a new GameLift fleet. When you create a new fleet for the build, GameLift sets up new fleet instances, copies the build files to each instance, and runs the installation script file (if you included one). Build files are copied to the following location on the instance:

- For Windows fleets: `C:\game`
- For Linux fleets: `/local/game`

To troubleshoot fleet activation problems that might be related to build installation, you can remotely access a fleet instance for debugging. See Remotely Access Fleet Instances (p. 48).

Working with Fleets

Amazon GameLift uses the concept of fleets to represent the deployed state of a single server build across zero or more Amazon Elastic Compute Cloud (Amazon EC2) instances. You can have multiple fleets per account with the same or different configurations (see AWS service limits on resources per account). You can delete fleets when you no longer need them. Use the AWS Command Line Interface (CLI) or the GameLift console to create a fleet.

> **Tip**
> You don't need to have a game ready to start experimenting with GameLift. The GameLift console offers a quick sample setup that gets you up and running with a sample game server and client in five easy steps. In addition, the GameLift Getting Started tutorials, in text and video format, walk you through each step in the process of creating and uploading a build, setting up a fleet, creating game sessions and connecting a client. The tutorials use a sample multiplayer game, which is included in the Lumberyard download.

Topics

Choose Computing Resources

Amazon GameLift uses Amazon Elastic Compute Cloud (Amazon EC2) resources to deploy your game servers and host game sessions for your players. When creating a fleet, you decide what type of resources to use and how many instances to maintain in the fleet. The more Amazon EC2 instances you have in your fleet, the more game sessions you can run concurrently. Keep in mind that each instance type has a cost associated with it, and you pay for each instance dedicated to your fleet. You can increase or decrease the number of instances in a fleet at any time (see Change Fleet Capacity (p. 41)) but the instance type is fixed once the fleet is created.

When you choose an instance type, you're determining the hardware that will be dedicated to each instance in the fleet, including computing power, memory, storage, and networking capacity. All

instances in a fleet use the same instance type, so if you have game servers with different computing requirements, you need to set up separate fleets.

The platform for a fleet's instances depends on the operating system of the fleet's build (see supported game server platforms (p. 7)). Learn more about instance types for:

- Microsoft Windows
- Amazon Linux

AWS Service Limits

AWS places limits how many Amazon EC2 instances can be use by an AWS account. Each instance type has a maximum number allowed per account, and this limit varies by AWS region. Each account also is limited in the total number of instances used regardless of type of instance. You can access information on limits in several ways:

- Find general limits for GameLift, as well as all other AWS services on the AWS Service Limits page.
- See limits for a specific region in the GameLift console: Select a region and choose **Service limits** from the Amazon GameLift menu. You can also view the total number of instances currently in use in the region.
- Retrieve the maximum number of instances per AWS account (by region) by using the GameLift API action DescribeEC2InstanceLimits. This action also returns the number for instances currently active in the region.

If you need more instances than allowed by AWS service limits, you can request an increase on the Amazon GameLift Service limits page of the AWS Management Console.

Create a Fleet

You can create a new fleet to host game servers for any game build that has been uploaded to the Amazon GameLift service and is in a **Ready** status. Use either the Amazon GameLift console or the AWS Command Line Interface (p. 11) (CLI) to create a fleet.

Create a Fleet (Console)

To create a fleet with the GameLift console:

1. Open the GameLift console at https://console.aws.amazon.com/gamelift/.
2. On the **Builds** page, find the build that you want to create a fleet for and verify that its status is **Ready**.
3. Select the build (use the option button to the left of the build status) and click **Create fleet from build**.
4. On the **Create fleet** page, complete the **Fleet Details** section:
   - **Name** – Create a meaningful fleet name so you can easily identify it in a list and in metrics.
   - **Description** – (Optional) Add a short description for your fleet to further aid identification.
   - **Build** – Note that the build information, including name, ID and operating system, is automatically filled in.
5. Under **Instance type**, select an Amazon EC2 instance type from the list. Only instance types matching the selected build's operating system are listed. For help choosing an instance type, see Choose Computing Resources (p. 37). You cannot change a fleet's instance type once the fleet is created.

Amazon GameLift provides a free tier with 125 hours of a c3.large instance per month for one year. To avoid incurring charges in excess of the free tier, you may want to set up billing alerts or turn on safe scaling for the fleet. For more information, see Free Tier and Billing Alerts (p. 8).

6. Under **Capacity allocation**, configure the type and number of game server processes to run on each instance. Each fleet must have one server process configuration defined. Once the fleet is created, you can edit the fleet to change a fleet's server process configuration.

   - **Launch path** – Type the path to the game executable in your build. All launch paths must start with the game server location, which varies based on the operating system in use. On Windows instances, game servers are built to the path `C:\game`. On Linux instances, game servers are built to `/local/game`, so all launch paths must start with this location. Examples: **`C:\game\MyGame\server.exe`** or **`/local/game/MyGame/server.exe`**.
   - **Launch parameters** – (Optional) You can pass information to your game executable at launch time. Type the information as a set of command line parameters here. Example: **`+sv_port 33435 +start_lobby`**.
   - **Concurrent processes** – Indicate how many server processes with this configuration to run concurrently on each instance in the fleet. Check GameLift's limits on number of concurrent server processes; they depend on which SDK your game server uses.

   Click **Add configuration** to configure another server process. Limits on concurrent server processes per instance apply to the total of concurrent processes for all configurations. If you're limited to one process, you can have only one configuration, and concurrent processes must be set to **`1`**. If you configure the fleet to exceed the limit, the fleet will not be activated.
7. Under **Safe scaling policy**, turn on this autoscaling policy to ensure that your fleet scales down to zero instances when there is no activity. Once the fleet is created, you can turn this policy on or off or set custom autoscaling policies on the fleet's detail page (see the **Scaling** tab).
8. Under **EC2 port settings** click **Add port settings** to define access permissions for inbound traffic connecting to server processes deployed on this fleet. You can create multiple port settings for a fleet. At least one port setting must be set for the fleet before any access is allowed. Once the fleet is created, you can edit the fleet to change a fleet's port settings.

   - **Port range** – Specify a range of port numbers that your game servers can use to allow inbound connections. A port range must use the format `nnnnn[-nnnnn]`, with values between 1025 and 60000. Example: **`1500`** or **`1500-20000`**.
   - **Protocol** – Select the type of communication protocol for the fleet to use.
   - **IP address range** – Specify a range of IP addresses valid for instances in this fleet. Use CIDR notation. Example: **`0.0.0.0/0`** (This example allows access to anyone trying to connect.)
9. Under **Protection policy**, choose whether or not to apply game session protection to instances in this fleet. Instances with protection are not terminated during a scale down event if they are hosting an active game session with connected players. You can also set protection for individual game sessions. Once the fleet is created, you can edit the fleet to change the fleet-wide protection policy.
10. Once you've finished setting the configuration for your new fleet, click **Initialize fleet**. GameLift assigns an ID to the new fleet and begins the fleet activation process. You can view the new fleet's status on the **Fleets** page. Once the fleet is active, you can change the fleet's capacity (p. 41) and other configuration settings as needed.

Create a Fleet (AWS CLI)

To create a fleet with the AWS CLI, open a command line window and use the `create-fleet` command to define a new fleet. See complete documentation on this command in the AWS CLI Command Reference. If you haven't yet installed the AWS CLI, see the topic Install the AWS CLI (p. 11).

This example creates a new fleet for an uploaded game build in a **Ready** status. The new fleet uses The operating system used for the fleet depends on which OS was defined for the game build.

```
$ aws gamelift create-fleet
--name "Sample test fleet"
--description "The sample test fleet"
--build-id "build-92f061ed-27c9-4a02-b1f4-6f85b2385620"
--ec2-instance-type "c3.large"
--runtime-configuration "ServerProcesses=[{LaunchPath=C:\game\Bin64.dedicated
\MultiplayerProjectLauncher_Server.exe,Parameters=+sv_port 33435
 +start_lobby,ConcurrentExecutions=1}]"
--new-game-session-protection-policy "FullProtection"
--ec2-inbound-permissions
 "FromPort=33435,ToPort=33435,IpRange=0.0.0.0/0,Protocol=UDP"
 "FromPort=33235,ToPort=33235,IpRange=0.0.0.0/0,Protocol=UDP"
```

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

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

Once the fleet is active, 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

Debug Fleet Creation Issues

Understanding how GameLift sets up a new fleet can help you resolve fleet activation errors.

When a new fleet is created, the GameLift service prepares to build the fleet, passing through a series of statuses. Problems occurring during this stage will cause the fleet status to move to **Error** with meaningful error messaging (for example, an incorrect build path or a service error). Once the preparation stage is complete, GameLift moves the fleet status to **Activating**.

To activate the new fleet, GameLift starts a new instance and attempts to deploy the build to it: copying build files, running an installation script, and launching server processes (as defined in the run-time configuration). Each server process launched on an instance must report to the GameLift service when it is ready to host a game session. Once a server process notifies GameLift that it is ready, the instance status moves to **Active**, which in turn moves the fleet status to **Active**.

The most common problem that game developers experience when creating a new fleet is having a fleet get stuck in an **Activating** status. Common reasons that a fleet might fail to activate are described below.

- **Server processes start but fail quickly or report poor health.** Other than issues with your game build, this outcome can happen when trying to run too many server processes simultaneously on the instance. The optimum number of concurrent processes depends on both the instance type and your game server's resource requirements. Try reducing the number of concurrent processes, which is set in the fleet's runtime configuration, to see if performance improves. You can change a fleet's runtime configuration using either the GameLift console (edit the fleet's capacity allocation settings) or by calling the AWS CLI command `update-runtime-configuration`.

  Server processes fail to start. Assuming there are no issues with your game build and installation script (if you provided one), check that you've correctly set the launch path and optional parameters in the fleet's runtime configuration. You can view the fleet's current run-time configuration using either the GameLift console (see the Fleet detail page, Capacity Allocation tab (p. 59)) or by calling the AWS CLI command `describe-runtime-configuration`.
- **Server processes start but fleet fails to activate.** If server processes start and continue to run successfully, but the fleet does not move to **Active** status, a likely cause is that the server process is failing to notify GameLift that it is ready to host game sessions. Check that your game server is correctly calling the Server API action `ProcessReady()` (see Prepare a Server Process (p. 13)).

For additional troubleshooting, you can also remotely access a fleet instance. See Remotely Access Fleet Instances (p. 48).

Change Fleet Capacity

You can change a fleet's capacity as needed (within the service limits) as long as the fleet's status is **Active**. When you create a new fleet, GameLift automatically sets the capacity to 1, which allows it to start one new instance of your game server. You can make these changes from either the Amazon GameLift console or the AWS CLI.

Once you change the fleet's desired capacity, GameLift immediately takes action to make the number of active instances match desired instances by creating new instances (scaling up) or terminating existing ones (scaling down).

To change capacity (console)

1. Open the GameLift console at https://console.aws.amazon.com/gamelift/.
2. On the **Fleets** page, click the name of an active fleet you want to change capacity for. This opens the fleet's detail page. You can also access a fleet's detail page via the **Dashboard**.
3. Click **Scaling**. This tab contains information about current and historical fleet capacity. Under **Scaling Limits**, there are several editable values, including **Desired**. This value tells GameLift how many active instances to maintain in the fleet and make available to host game sessions. To change the fleet's capacity, click on the Desired value, edit it, and then click the check mark to save your change.

   Once you change the **Desired** value, you should start to see the number of **Active** instances (shown on the **Scaling** tab under **Instance Counts**), change to match the new desired value. This process can take time, depending on a number of circumstances.

To change capacity (AWS CLI)

- In a command line window, type the `update-fleet-capacity` command with the following parameters:

```
--fleet-id <unique fleet identifier>
--desired-instances <fleet capacity as an integer>
```

```
--max-size <maximum capacity for autoscaling>    [Optional]
--min-size <minimum capacity for autoscaling>    [Optional]
```

Example:

```
aws gamelift update-fleet-capacity
--fleet-id fleet-eead767f-acb4-4c2a-9280-a3c523cbe50f
--desired-capacity 5
--max-size 10
--min-size 1
```

Edit a Fleet

Use the **Edit fleet** page in the Amazon GameLift console to update the name, description, server process configuration, and inbound access permissions for your fleet.

Note
If your build has been deleted or an error has occurred while attempting to retrieve your build, you may see one of the following **Build** statuses:

- **Deleted** – The build for this fleet was deleted. Your fleet will still run properly despite the build having been deleted.
- **Error** – An error occurred while attempting to retrieve build information for the fleet.

To edit a fleet

1. Open the GameLift console at https://console.aws.amazon.com/gamelift/.
2. Choose **Fleets** from the menu bar.
3. On the **Fleets** page, click the name of the fleet you want to edit.

   Note
   You can only edit an active fleet.
4. On the selected fleet page, for **Actions**, choose **Edit fleet**.
5. On the **Edit fleet** page, you can edit the following:

   - **Name** – Friendly name for your fleet
   - **Description** – (Optional) Short description for your fleet
   - **Capacity allocation** – Configuration of server process(es) to be run on each instance in the fleet
   - **Amazon EC2 port settings** – Port range, protocol, and IP address ranges to allow inbound port settings
6. Click **Submit**.

Delete a Fleet

In the process of testing or running multiple versions of your game, you might create dozens of fleets that eventually become unnecessary to maintain. You can delete fleets that you no longer need. Deleting a fleet also permanently removes associated game sessions and collected data.

Note
You can delete a fleet only after you have scaled it down to 0. Set the desired number of EC2 instances to 0 and then wait for the scaled down state to take effect before you delete the fleet.

To delete a fleet

1. In the Amazon GameLift console, choose **Fleets** from the menu bar.
2. On the **Fleets** page, select the fleet you want to delete.
3. On the selected fleet page, for **Actions**, choose **Terminate fleet**.
4. In the **Terminate fleet** dialog box, confirm the deletion by typing the name of the fleet.
5. Click **Delete**.

Set Up Fleet Autoscaling

Use autoscaling to have Amazon GameLift automatically scale your fleet capacity in response to activity on your game servers. This topic provides help with creating an autoscaling policy and offers tips on configuring your fleet to optimize the benefits of autoscaling. For more information on how autoscaling works, see the Autoscaling (p. 4) section in "How Amazon GameLift Works" overview.

To activate Amazon GameLift's autoscaling feature for a fleet, define one or more autoscaling policies. The Amazon GameLift console offers a simple tool for creating, updating, and viewing your autoscaling policies. You can also manage your policies using the AWS Command Line Interface (CLI).

Set Autoscaling with the Console

1. Sign in to the AWS Management Console and open the GameLift console at https://console.aws.amazon.com/gamelift/.
2. On the **Fleets** page, click the name of the fleet you want to set an autoscaling policy for. This opens the fleet's detail page.
3. Click **Scaling**. This tab contains information about current and historical fleet capacity as well as options for creating and updating fleet autoscaling policies.
4. Click **Add Policy** to start a new policy statement. Type a policy name that is unique to this fleet.
5. Create your policy statement by setting the policy parameters. For help setting parameters, see Create an Autoscaling Policy Statement (p. 44). When done, click the check mark icon to save the policy. Once saved, GameLift starts evaluating metric data for against the policy within about ten minutes.

   For example, the following policy statement ensures that fleet capacity will always be increased if the number of idle instances (instances waiting to host a game session) drops below two for longer than 10 minutes: "If **`Idle Instances`** are **`< 2`** for **`10`** minutes, then **`scale up by 1 instance`**".

6. When using autoscaling, you should set minimum and maximum capacity limits. In the fleet's Scaling tab, under **Scaling Limits**, edit the **Minimum** and **Maximum** values.

Set Autoscaling with the AWS CLI

1. In a command line window, type the `put-scaling-policy` command with the following parameters. For help setting parameters, see Create an Autoscaling Policy Statement (p. 44).

```
--fleet-id <unique fleet identifier>
--name "<unique policy name>"
--metric-name <name of metric>
--comparison-operator <comparison operator>
--threshold <threshold integer value>
--evaluation-periods <number of minutes>
--scaling-adjustment-type <adjustment type>
--scaling-adjustment <adjustment amount>
```

   Example:

```
aws gamelift put-scaling-policy
--fleet-id fleet-eead767f-acb4-4c2a-9280-a3c523cbe50f
--name "scale up when available player sessions is low"
--metric-name AvailablePlayerSessions
--comparison-operator LessThanThreshold
--threshold 50
--evaluation-periods 10
--scaling-adjustment-type ChangeInCapacity
--scaling-adjustment 1
```

2. When using autoscaling, you should set minimum and maximum capacity limits. Type the `update-fleet-capacity` command with the following parameters:

```
--fleet-id <unique fleet identifier>
--min-size <minimum capacity for autoscaling>
--max-size <maximum capacity for autoscaling>
```

   Example:

```
aws gamelift update-fleet-capacity
--fleet-id fleet-eead767f-acb4-4c2a-9280-a3c523cbe50f
--min-size 1
--max-size 10
```

Create an Autoscaling Policy Statement

An autoscaling policy for a fleet makes the following statement: "If a fleet metric meets or crosses a threshold value for a certain length of time, then change the fleet's capacity by a certain amount."

To construct an autoscaling policy statement, you must specify six key variables:

> If *<metric name>* remains at *<comparison operator>* *<threshold value>* for *<evaluation period>*, then change fleet capacity using *<adjustment type>* to/by *<adjustment value>*.

For example:

> If `AvailablePlayerSessions` remains at `less than 50` for `60 minutes`, then change fleet capacity using `ChangeInCapacity` by `1 instance`.

Metric name

Metrics track some type of activity in your game servers. Each minute, GameLift collects a heartbeat from every instance, game session, and player session in the fleet. This data, in aggregate, represents the metric data that autoscaling uses to trigger an increase or decrease in capacity. An autoscaling policy can be linked to one of six possible metrics:

- **AvailablePlayerSessions** – Number of player session slots currently available in active game sessions across the fleet (including player slots in game sessions that are not currently accepting players). This metric is useful for measuring how close the fleet is to maximum or minimum capacity.
- **IdleInstances** – Number of instances in the fleet ready and waiting to host new game sessions. This metric helps track how quickly a fleet can get new players connected to a game if demand spikes.
- **ActiveGameSessions** – Number of active game sessions currently running.
- **CurrentPlayerSessions** – Number of active or reserved player sessions in the fleet.
- **ActiveInstances** – Number of instances currently running a game session (this is currently the same as ActiveGameSessions).
- **ActivatingGameSessions** – Number of game sessions in the process of being created.

Comparison operator

This variable tells GameLift how to compare the metric data against the threshold. Valid comparison operators include greater than (>), less than (<), greater than or equal (>=) or less than or equal (<=).

Threshold value

Threshold value is used to trigger scaling events when the metric value meets or crosses it. This value is always a positive integer. Depending on the metric selected, it indicates an amount of player sessions, game sessions, or instances.

Evaluation period

To trigger a scaling event, the metric must meet or cross the threshold for the entire length of an evaluation period. If the metric retreats from the threshold, the evaluation period starts over again.

Adjustment type & value

This set of variables works together to specify how you want GameLift to adjust fleet capacity when this policy triggers a scaling event. An autoscaling policy can make any of three types of adjustments:

- **Change in capacity** – Increase or decrease current capacity by a specified number of instances. Set the adjustment value to the number of instances to add or subtract from the fleet.
- **Percent change in capacity** – Increase or decrease current capacity by a specified percentage. Set the adjustment value to the percentage you want to increase or decrease the fleet by. For example, with an adjustment value of **15** percent, a 20-instance fleet is reduced by three instances.
- **Exact capacity** – Set capacity to a specified size. Set the adjustment value to the exact number of instances you want in the fleet.

> **Note**
>
> If you're using the AWS CLI or the AWS SDK for GameLift, use positive adjustment values to scale up and negative values to scale down. If you're using the Amazon GameLift console, choose the **scale up by** or **scale down by** options.

Tips on Autoscaling

The following suggestions can help you get the most out of autoscaling.

Use multiple policies

You can have multiple autoscaling policies in force for a fleet at the same time. The most common scenario is to have one policy to manage scaling up and one to manage scaling down. However, there are no limits on combining policies.

Multiple policies behave independently. Keep in mind that there is no way to control the sequence of scaling events. For example, if you have multiple policies driving scaling up, it is possible that player activity could trigger multiple scaling events simultaneously. For example, the effects of two scale up policies can easily be compounded if it is possible for player activity to trigger both metrics. Also watch for policies that trigger each other. For example, you can create an infinite loop if you create scale up and scale down policies that sets capacity beyond the threshold of each other.

Set maximum and minimum capacity

Each fleet has a maximum and minimum capacity setting. This feature is particularly important when using autoscaling. Autoscaling will never set capacity to a value outside of this range. By default, newly created fleets have a minimum of 0 and a maximum of 1. For your autoscaling policy to affect capacity as intended, you must increase the maximum value.

Fleet capacity is also constrained by limits on the fleet's instance type and on your AWS account. You cannot set a minimum and maximum that is outside the service and account limits.

Track metrics after a change in capacity

After changing capacity in response to an autoscaling policy, GameLift waits ten minutes before responding to triggers from that policy. This wait allows GameLift time to add the new instances, launch the game servers, connect players, and start collecting data from the new instances. During this time, GameLift continues to evaluate the policy against the metric and track the evaluation period, which restarts once a scaling event is triggered. This means that a scaling policy could trigger another scaling event immediately after the wait time is over.

There is no wait time between scaling events triggered by different autoscaling policies.

Run Multiple Processes on a Fleet

This topic provides additional information on how multiple processes per instance are managed on a fleet and how you can make use of this feature for your games. Depending on how you configure your fleet, running multiple processes gives you greater control over how efficiently you use your Amazon GameLift resources, which can potentially reduce overall operating costs for your game.

Optimizing for Multiple Processes

At a minimum, you must do the following to enable multiple processes:

- Create a build (p. 34) that contains all of the game server executables that you want to deploy to a fleet and upload it to GameLift. All game servers in a build must run on the same platform and be integrated with GameLift using the GameLift Server SDK for C++, version 3.0.7 or later.

- Create a run-time configuration with one or more server process configurations and multiple concurrent processes.
- Game clients connecting to games hosted on this fleet must be integrated using the AWS SDK, version 2016-08-04 or later.

In addition, implement the following in your game servers to optimize fleet performance:

- Handle server process shutdown scenarios to ensure that GameLift can recycle processes efficiently. If you don't do this, server processes can't be shut down until they fail.
  - Add a shutdown procedure to your game server code, ending with the server API call to `ProcessEnding()`.
  - Implement the callback function `OnProcessTerminate()` in your game server code to gracefully handle termination requests from GameLift.
- Make sure that "unhealthy" server processes are shut down and relaunched quickly by defining what "healthy" and "unhealthy" mean and reporting this status back to GameLift. You do this by implementing the `OnHealthCheck()` callback function in your game server code. GameLift automatically shuts down server processes that are reported unhealthy for three consecutive minutes. If you don't implement `OnHealthCheck()`, GameLift assumes a server process is healthy unless it fails to respond. As a result, poorly performing server processes can continue to exist, using up resources until they finally fail.

How a Fleet Manages Multiple Processes

Amazon GameLift uses a fleet run-time configuration to manage what processes to maintain on each instance in the fleet. A run-time configuration is actually made up of one or multiple server process configurations, each of which identifies the following:

- The path and file name of a server executable in the game build deployed on the fleet
- (Optional) Parameters to pass to the server process on launch
- The number of this server process to maintain concurrently on the instance

When an instance is started in the fleet, the instance immediately begins launching server processes called for in the run-time configuration. Server process launches on an instance are staggered by a few seconds, so depending on the total number of server processes configured for an instance, it may take a few minutes to achieve full capacity.

Over time, server processes end, either by self-terminating (calling the Server SDK `ProcessEnding()`) or by being terminated by GameLift. An instance regularly checks that it is running the number and type of server processes specified in the run-time configuration. If not, the instance automatically launches server processes as needed to meet the run-time configuration requirements. As a result, as server processes end, their resources are continually recycled to support new server processes, and instances generally maintain the expected complement of active server processes.

You can change a fleet's run-time configuration at any time by adding, changing, or removing server process configurations. Here's how GameLift adopts run-time configuration changes.

1. Before an instance checks that it is running the correct type and number of server processes, it first requests the latest version of the fleet's run-time configuration from the GameLift service. If you've changed the run-time configuration, the instance acquires the new version and implements it.
2. The instance checks its active processes against the current run-time configuration and handles discrepancies as follows:
   - The updated run-time configuration removes a server process type. Active server processes that no longer match the run-time configuration continue to run until they end.

- The updated run-time configuration decreases the number of concurrent processes for a server process type. Excess server processes of that type continue to run until they end.
- The updated run-time configuration adds a new server process type or increases the concurrent processes setting for an existing type. New server processes are launched immediately to match the updated run-time configuration, unless the instance is already running the maximum number of server processes. In this case, new server processes are launched only when existing processes end.

Choosing the Number of Processes per Instance

There are effectively three limits to keep in mind when determining the number of concurrent processes:

- GameLift limits each instance to a maximum number of concurrent processes. Whether your run-time configuration has one or multiple server process configurations specified , the sum of all concurrent processes for each server process configuration can't exceed this limit.
- The Amazon EC2 instance type that you choose may limit the number of processes that can be effectively run concurrently. You need to balance the number of processes against the instance type size and test different configurations. Key factors include the resource requirements of the game servers, the number of players to be hosted in each game session, and your performance expectations.
- The total number of concurrent processes never exceeds the total number specified in the current run-time configuration. This limit comes into play when you change your run-time configuration, and ensures that an instance doesn't launch a new set of server processes while the old processes are still running, potentially overtaxing the instance's resources. Instead, GameLift only starts new processes when old processes end. Here's an example: An instance is running 10 concurrent processes of the server executable `mygamev1.exe`. You update the fleet's run-time configuration to run 10 concurrent processes of the server executable `mygamev2.exe`. This means that the maximum number of concurrent processes on an instance remains at 10. As a result, no `mygamev2.exe` processes are launched until a `mygamev1.exe` process ends.

Remotely Access Fleet Instances

You can remotely access any fleet instance that is currently running in an Amazon GameLift fleet. This capability is useful for troubleshooting fleet activation issues. You can also use this feature to get real-time game server activity, such as track log updates or run benchmarking tools using actual player traffic.

When remotely accessing individual Amazon GameLift instances, keep the following in mind:

- The GameLift service continues to manage fleet activity and capacity. Establishing a remote connection to an instance does not affect how GameLift manages it in any way. As a result, the instance continues to execute the fleet runtime configuration, stop and start server processes, create and terminate game sessions, and allow player connections. In addition, the GameLift service may terminate the instance at any time as part of a scale down event.
- Making local changes to an instance that is hosting active game sessions and has live players connected may significantly affect player experience. For example, your local changes have the potential to drop individual players, crash game sessions or even shut down the entire instance with multiple game sessions and players affected.

For more information on how games are deployed and managed on Amazon GameLift instances, see the following topics:

- How Amazon GameLift Works (p. 2)
- Debug Fleet Creation Issues (p. 40)
- How a Fleet Manages Multiple Processes (p. 47)

Connect to an Instance

You can access remote instances that are running either Windows or Linux. To connect to a Windows instance, use a remote desktop protocol (RDP) client. To connect to a Linux instance, use an SSH client.

Use the AWS CLI get the information you need to access a remote instance. For help, see the AWS CLI Command Reference. If you haven't yet installed the AWS CLI, see the topic Install the AWS CLI (p. 11).

1. **Find the ID of the instance you want to connect to.** When requesting access, you must specify an instance ID. You can get information on all fleet instances using the command `describe-instances` with a fleet ID. This example retrieves the first three instances in a fleet:

   ```
   $ aws gamelift describe-instances
       --fleet-id "fleet-a7abc071-5537-4f0f-b5ee-1b5c1187565f"
       --limit 3
   ```

2. **Request access credentials for the instance.** Once you have an instance ID, use the command `get-instance-access` to request access credentials and other information. If successful, GameLift returns the instance's operating system, IP address, and a set of credentials (user name and secret key). The credentials format depends on the instance operating system. Use the following instructions to retrieve credentials for either RDP or SSH.

   For Windows instances – To connect to a Windows instance, RDP requires a user name and password. The `get-instance-access` request returns these values as simple strings, so you can use the returned values as is. Example:

   ```
   $ aws gamelift get-instance-access
       --fleet-id "fleet-a7abc071-5537-4f0f-b5ee-1b5c1187565f"
       --instance-id "i-01463992e435d836c"
   ```

   For Linux instances – To connect to a Linux instance, SSH requires a user name and private key. GameLift issues RSA private keys and returns them as a single string, with the newline character (`\n`) indicating line breaks. To make the private key usable, you must (1) convert the string to a `.pem` file, and (2) set permissions for the new file.

   - To convert the string to a properly formatted `.pem` file, add special parameters to your `get-instance-access` request, as shown in the example below. This example automatically outputs the returned value of `Secret` to a text file named `MyPrivateKey.pem`, and replaces all the `\n` characters with line breaks.

     ```
     $ aws gamelift get-instance-access
         --fleet-id  "fleet-a7abc071-5537-4f0f-b5ee-1b5c1187565f"
         --instance-id  "i-01463992e435d836c"
         --query  'InstanceAccess.Credentials.Secret'
         --output text > MyPrivateKey.pem
     ```

   - To set permissions on the new file, run the following command

     ```
     $ chmod 400 MyPrivateKey.pem
     ```

3. **Open a port for the remote connection.** Instances in GameLift fleets can only be accessed through ports authorized in the fleet configuration. You can view a fleet's port settings using the command `describe-fleet-port-settings`.

   As a best practice, we recommend opening ports for remote access only when you need them and closing them when you're finished. Use the command `update-fleet-port-settings` to add a port setting for the remote connection (such as 22 for SSH or 3389 for RDP). For the IP range value, specify the IP addresses for the devices you plan to use to connect (converted to CIDR format). Example:

```
$ aws gamelift update-fleet-port-settings
    --fleet-id  "fleet-a7abc071-5537-4f0f-b5ee-1b5c1187565f"
    --inbound-permission-authorizations
 "FromPort=22,ToPort=22,IpRange=54.186.139.221/32,Protocol=TCP"
```

4. **Open a remote connection client.** Use Remote Desktop for Windows or SSH for Linux instances. Connect to the instance using the IP address, port setting, and access credentials.

View and Update Remote Instances

When connected to an instance remotely, you have full user and administrative access. This means you also have the ability to cause errors and failures in game hosting. If the instance is hosting games with active players, you run the risk of crashing game sessions and dropping players, as well as disrupting game shutdown processes and causing errors in saved game data and logs.

Hosting resources on an instance can be found in the following locations:

- **Game build files.** These are the files included in the game build you uploaded to GameLift. They include one or more game server executables, assets and dependencies. These files are located in a root directory called `game`:
  - On Windows: `c:\game`
  - On Linux: `/local/game`
- **Game log files.** Any log files your game server generates are stored in the `game` root directory at whatever directory path you designated.
- **GameLift hosting resources.** Files used by the GameLift service to manage game hosting are located in a root directory called `Whitewater`. These files should not be changed for any reason.

- **Runtime configuration.** The fleet runtime configuration is not accessible for individual instances. To test changes to a runtime configuration (launch path, launch parameters, maximum number of concurrent processes), you must update the fleet-wide runtime configuration (see the AWS SDK action UpdateRuntimeConfiguration or the AWS CLI update-runtime-configuration).

Working with Aliases

Amazon GameLift uses the term *alias* to refer to the redirecting of player sessions to a fleet that you specify. You can use aliases to direct player sessions to fleets that you're testing for launch or to support multiple game client sessions.

Amazon GameLift uses the term alias to refer to the redirect connection between a fleet and players. You can decide which fleet you want an alias to resolve to, and you can embed the alias ID into your game client to control the flow of players to the designated fleet.

There are two types of routing strategies for aliases:

- **Simple** – A simple alias routes player traffic to the associated fleet. You can update the fleet to which the alias resolves at any time.
- **Terminal** – A terminal alias does not resolve to a fleet. Instead, it passes a message back to the client. For example, you may want to notify players to upgrade their game client versions if they are attempting to connect to a fleet that is no longer supported.

Create an Alias

You can create a new alias to resolve to a fleet.

To create a new alias

1. Open the GameLift console at https://console.aws.amazon.com/gamelift/.
2. Choose **Aliases** from the menu bar.
3. On the **Aliases** page, click **Create alias**.
4. On the **Create alias** page, in the **Alias details** section, do the following:
   - **Alias name** – Type a friendly name so you can easily identify the alias in the catalog.
   - **Description** – (Optional) Type a short description for your alias to add further identification.
5. In the **Routing options** section, for **Type**, choose **Simple** or **Terminal**:
   - If you choose **Simple**, select an available fleet to associate with your alias. A simple alias routes player traffic to the associated fleet.
   - If you select **Terminal**, type a message that will be displayed to players. A terminal alias does not resolve to a fleet but only passes your message to the client.
6. Click **Configure alias**.

Edit an Alias

Use the **Edit alias** page in the Amazon GameLift console to update the information for your alias.

To edit an alias

1. Open the GameLift console at https://console.aws.amazon.com/gamelift/.
2. Choose **Aliases** from the menu bar.
3. On the **Aliases** page, click the name of the alias you want to edit.
4. On the selected alias page, for **Actions**, choose **Edit alias**.
5. On the **Edit alias** page, you can edit the following:

   - **Alias name** – Friendly name for your alias.
   - **Description** – Short description for your alias.
   - **Type** – Routing strategy for player traffic. Select **Simple** to change the associated fleet or select **Terminal** to edit the termination message.
6. Click **Submit**.

Viewing Your Game Data in the Console

Amazon GameLift continually collects data for active games to help you understand player behavior and performance. With the Amazon GameLift console, you can view, manage, and analyze this information for your builds, fleets, game sessions, and player sessions.

Topics

View Your Current Amazon GameLift Status

The **Dashboard** provides a grid view showing the following:

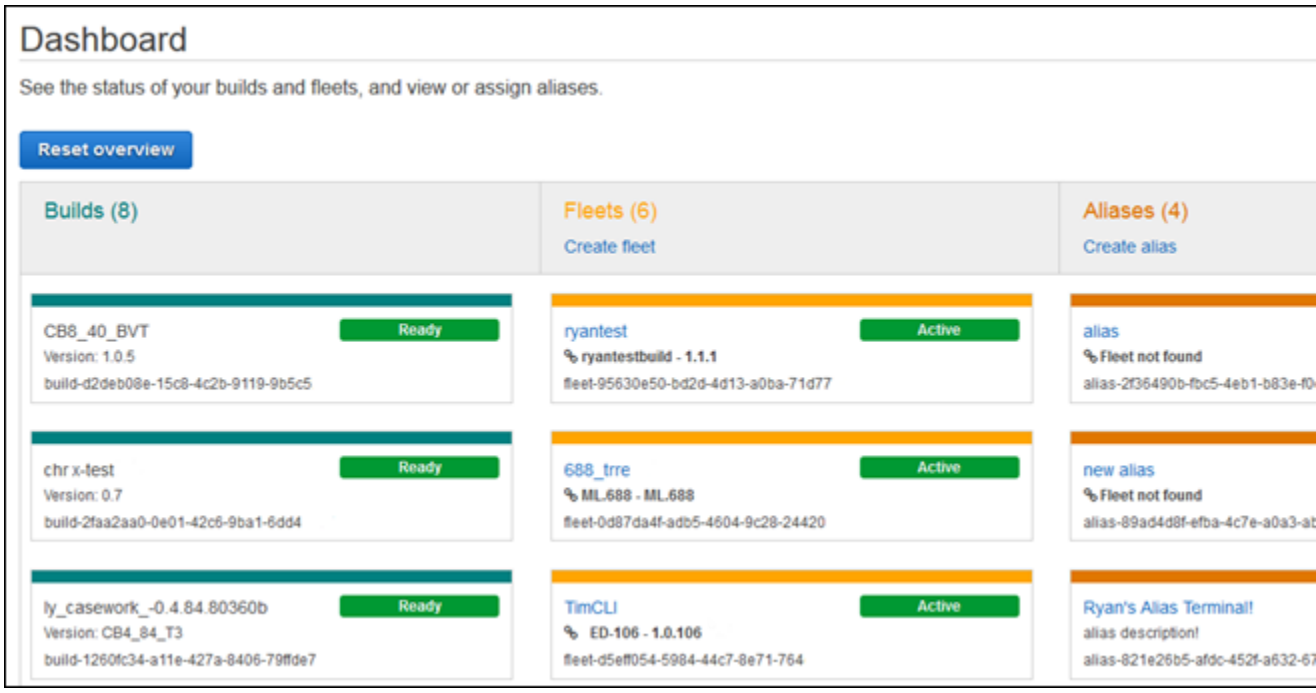
- Uploaded builds
- Created fleets in all statuses
- Created aliases and the fleets they point to (if any)

To open the GameLift dashboard

- In the GameLift console, choose **Dashboard** from the menu bar.

From the dashboard you can take the following actions:

- Create a new fleet or alias.
- View relationships among items. Click anywhere inside an item box to show the relationships between that item and others on the dashboard. For example, click a build to display all fleets that were created with that build. Click a fleet to see the build it was created with and the alias it points to. To reset the dashboard, click the **Reset overview** button.
- View details on a build, fleet, or alias. Click the ID number for a item to open the details page.



View Your Builds

You can view information about all the game server builds you have uploaded to Amazon GameLift and take actions on them. Builds shown include only those uploaded for the selected region.

Build Catalog

Uploaded builds are shown on the **Builds** page. To view this page, choose **Builds** from the GameLift console menu bar.

The **Builds** page provides the following summary information for all builds:

- **Status** – Displays one of three possible status messages:
  - **Initialized** – The build has been created, but the upload has not yet started or the upload is still in progress.
  - **Ready** – The build has been successfully received and is ready for fleet creation.
  - **Failed** – The build timed out before the binaries were received.
- **Build name** – Name associated with the uploaded build. A build name is provided when uploading the build to GameLift, and can be changed using the AWS SDK action `UpdateBuild`.
- **Build ID** – Unique ID assigned to the build on upload.
- **Version** – Version label associated with the uploaded build. A build name is provided when uploading the build to GameLift, and can be changed using the AWS SDK action `UpdateBuild`.
- **OS** – Operating system that the build runs on. The build OS determines what operating system is installed on a fleet's instances.
- **Size** – Size, in megabytes (MB) of the build file uploaded to GameLift.
- **Date created** – Date and time that the build was uploaded to GameLift.
- **Fleets** – Number of fleets currently deployed with this build.

From this page you can do any of the following:

- Create a new fleet from a build. Select a build and click **Create fleet from build**.
- Delete a fleet. Select a build and click **Delete build**.
- Filter and sort the build list. Use the controls at the top of the table.
- View build details. Click a build name to open the build detail page.

Build Detail

Access a build's detail page from either the console dashboard or the **Builds** page by clicking the build name. The **Build** detail page displays the same build summary information as the Builds page. It also shows a list of fleets created with the build. This list is essentially the fleets catalog, filtered by build. It includes the same summary information as the **Fleets** page (p. 55).

View Your Fleets

You can view information on all the fleets created to host your games on Amazon GameLift under your AWS account. Fleets shown include only those created in the selected region. From the **Fleets** page, you can create a new fleet or view additional detail on one selected fleet. A **Fleet** detail page contains usage information and metrics; it also lets you view and edit fleet configuration settings, create or remove the fleet, and access the fleet's game session and player session data.

To view the **Fleets** page, choose **Fleets** from the GameLift console's menu bar.

The **Fleets** page displays the following summary information:

- **Status** – The status of the fleet, which can be one of these states: **New**, **Downloading**, **Building**, and **Active**. A fleet must be in Active status to be able to host game sessions and allow player connections.
- **Fleet name** – Friendly name given to the fleet.
- **EC2 type** – The Amazon EC2 instance type, which determines the computing capacity of fleet's instances.
- **OS** – Operating system on each instances in the fleet. A fleet's OS is determined by the build deployed to it.
- **Active** – The number of EC2 instances in use for the fleet.
- **Desired** – The number of EC2 instances GameLift should maintain in the fleet. This value is configurable (within service limits). GameLift starts or terminates instances as needed to maintain the desired number of instances.
- **Active game sessions** – The number of game sessions currently running in the fleet. The data is delayed five minutes.
- **Player sessions** – The number of players connected to game sessions in the fleet. The data is delayed five minutes.
- **Uptime** – The total length of time the fleet has been running.
- **Date created** – The date and time the fleet was created.

View Fleet Details

You can access detailed information on any fleet, including configuration settings, scaling settings, metrics, and game and player data. Access a **Fleet** detail page from either the console dashboard or the **Fleets** page by clicking the fleet name.

The fleet detail page displays a summary table and tabs containing additional information. On this page you can do the following:

- Update the fleet's metadata and run-time configuration. Choose **Actions: Edit fleet**.
- Change fleet capacity settings. On the **Scaling** page, edit values from **Minimum**, **Maximum**, and **Desired** instances.
- Set or change autoscaling policies. On the **Scaling** page, add or edit a policy.
- Shut down a fleet. Choose **Actions: Terminate fleet**.

Summary

The summary table includes the following information:

- **Status** – Current status of the fleet, which may be **New**, **Downloading**, **Building**, and **Active**. A fleet must be active before it can host game sessions or accept player connections.
- **Fleet ID** – Unique identifier assigned to the fleet.
- **EC2 type** – Amazon EC2 instance type selected for the fleet when it was created. A fleet's instance type specifies the computing hardware and capacity used for each instance in the fleet and determines the instance limits for the fleet.
- **OS** – Operating system on each instances in the fleet. A fleet's OS is determined by the build deployed to it.
- **Active instances** – Number of instances currently in an **Active** status in the fleet.
- **Active servers** – Number of server processes currently in an **Active** status in the fleet. The data has a five-minute delay.
- **Active game sessions** – Number of game sessions currently running on instances in the fleet. The data has a five-minute delay.
- **Current player sessions** – Number of players currently connected along with the total number of player slots in active game sessions across the fleet. For example: 25 (connected players) of 100 (possible players) means the fleet can support 75 additional players. The data has a five-minute delay.
- **Protection** – Current setting for game session protection (p. 3) for the fleet.

  Uptime – Total length of time the fleet has been active.
- **Date created** – Date and time indicating when the fleet was created.

Metrics

The **Metrics** tab shows a graphical representation of fleet metrics over time.

To display metrics information in the graph

1. Click one or more metric name to the left of the graph area to add it to the graph display. Metric names that are turned off are gray. Use the color key to identify which graphed line matches a selected metric. The following metrics are available:

   - **Game** – These metrics show utilization of the fleet's capacity over time.
     - **Available player sessions** – Number of unused player slots in active game sessions across the fleet. This number includes open slots in all game sessions regardless of whether or not the game is currently accepting new players.
     - **Current player sessions** – Number of players currently connected to active game sessions across the fleet.

- **Player session activations** – Number of players joining an active game session. This metric is useful for tracking how the influx of new players changes over time.
- **Active game sessions** – Number of game sessions currently running across the fleet.
- **Activating game sessions** – Number of new game sessions in an ACTIVATING status across the fleet. Game sessions cannot accept player connections until they are active.

- **Server processes** – These metrics track the status and health of server processes across the fleet. The GameLift service regularly polls each active server process for its health.
  - **Active** – Number of server processes in an ACTIVE status. Active server processes are able to host game sessions.
  - **Healthy** – Number of active server processes that reported healthy in the last health check.
  - **Percent healthy** – Percentage of active server processes that reported healthy in the last health check.
  - **Activations** – Number of new server processes in an ACTIVATING status across the fleet. Server processes cannot host a game session until they are active.
  - **Terminations** – Number of server processes that were shut down. This metric includes all process terminations regardless of reason.
  - **Abnormal Terminations** – Number of server processes did not terminate cleanly; that is, the server process did not call `ProcessEnding() (p. 19)` and terminate with a zero exit code.
- **Hardware** – These metrics reflect utilization of the fleet's computing resources. CPU utilization is expressed as a percentage.

2. Use the following filters, shown above the graph area, to change how metric data is displayed:

   - **Data & Period** – Offers two options for selecting a date range:
     - Use **Relative** to select a period of time relative to the current time, such as **Last hour**, **Last day**, **Last week**.
     - Use **Absolute** to specify a period with an arbitrary start and end date/time.
   - **Granularity** – Select a length of time to aggregate data points.
   - **Refresh rate** – Select how often you want the graph display to be updated.
   - **Format** – Select which time format to use in the graph display: **UTC** (universal coordinated time) or **Local**.
   - **Show Points** – Toggle on or off to display discrete data points (as circles) or display lines only.

Events

The **Events** tab provides a log of all events that have occurred on the fleet, including the event code, message, and time stamp.

Scaling

The **Scaling** tab contains information related to fleet capacity, including the current status and a graphical representation of capacity changes over time. It also provides tools to update capacity limits and manage autoscaling.

To view current and historical scaling information

1. Go to the top of the **Scaling** tab. To view the current capacity status for this fleet, look at the scaling values at the left. These values are defined as follows:

   - **Scaling Limits** – These metrics track the history of changes to capacity limits.
     - **Minimum** – Hard lower limit on the number of instances to maintain in the fleet. Fleet capacity will not drop below the current minimum during autoscaling or even if desired capacity is set below the minimum.

- **Desired** – The number of active instances currently *wanted* in the fleet. GameLift's goal is to make the number of **Active** instances (explained later) match the number of desired instances; it achieves this by creating or terminating instances as needed.
- **Maximum** – Hard upper limit on the number of instances to maintain in the fleet. Fleet capacity will not exceed the current maximum during autoscaling or if desired capacity is set above the maximum.

- **Instance Counts** – These metrics track actual changes in capacity over time.
  - **Active** – Number of instances in the fleet that are running a game server. This number should match the number of **Desired** instances; if it does not, then GameLift may be in the process of scaling up or down, or at the fleet's minimum or maximum limit.
  - **Idle** – Number of active instances in the fleet that are not currently hosting game sessions. This metric indicates available capacity that is not being utilized.
  - **Pending** – Number of instances GameLift is currently starting up to host game sessions (scaling up).
  - **Terminating** – Number of instances GameLift is currently in the process of shutting down (scaling down).

2. To view how fleet capacity has changed over time, display in the graph any or all of the scaling metrics listed on the left. Click the metric name to add it to the graph. (Metric names are gray when not in use.) Use the color key to identify which graphed line matches a selected metric.
3. (Optional) Use the following filters, shown above the graph area, to specify how metric data is displayed in the graph:

   - **Data & Period** – Offers two options for selecting a date range:
     - Use **Relative** to select a period of time relative to the current time, such as **Last hour**, **Last day**, **Last week**.
     - Use **Absolute** to specify a period with an arbitrary start and end date/time.
   - **Granularity** – Select a length of time to aggregate data points.
   - **Refresh rate** – Select how often you want the graph display to be updated.
   - **Format** – Select which time format to use in the graph display: **UTC** (universal coordinated time) or **Local**.
   - **Show Points** – Toggle on or off to display discrete data points (as circles) or display lines only.

To change fleet capacity

1. Go to the top of the **Scaling** tab. You can manually set the fleet's capacity by changing the current values of the fleet's scaling limits. You can edit these values, which are shown along the left side of the **Scaling** tab.
2. To specify the number of instances that you want available to host game sessions, set the value of **Desired**. GameLift immediately attempts to scale instances up or down to meet the new value. If the new value you set is above the maximum or below the minimum limits, then GameLift attempts to scale up or down to that limit.
3. Set the maximum and minimum scaling limits as needed. If the value of **Desired** instances was or is now outside the range of your limits, changing these limits causes GameLift to immediately try to scale up or down to get closer to the **Desired** value while staying inside the new limit range.

To manage automatic scaling

- Go to the end of the **Scaling** tab to find tools for setting up autoscaling policies. See Set Up Fleet Autoscaling (p. 43) for more details on autoscaling and how to manage policies.

Game sessions

The **Game sessions** tab lists past and present contains game sessions hosted on the fleet, including some detail information. Click a game session ID to access additional game session information, including player sessions.

- **Status** – Game session status. Valid statuses are:
  - **Activating** – A game session has been initiated and is preparing to run.
  - **Active** – A game session is running and available to receive players (depending on the session's player creation policy).
  - **Terminated** – Game session has ended.
- **Name** – Game generated for the game session.
- **ID** – Unique identifier assigned by GameLift to the game session.
- **IP address** – IP address specified for the game session.
- **Port** – Port number used to connect to the game session.
- **Player sessions** – Number of players connected to the game sessions along with total possible players the game session can support. For example: 2 (connected players) of 10 (possible players) means the fleet can support 8 additional players.
- **Uptime** – Total length of time the game session has been running.
- **Date created** – Date and time stamp indicating when the fleet was created.

Build

The **Build** tab displays the fleet's build-related configuration, which was set when the fleet was created. Select the build ID to see the full **build** detail page.

If your build has been deleted or an error has occurred while attempting to retrieve your build, you may see one of the following status messages:

- **Deleted** – The build for this fleet was deleted. Your fleet will still run properly despite the build having been deleted.
- **Error** – An error occurred while attempting to retrieve build information for the fleet.

Capacity allocation

The **Capacity allocation** tab displays the run-time configuration for the fleet, which specifies what server processes to launch on each instance and how. It includes the path for the game server executable and optional launch parameters. You can change the fleet's capacity allocation either by editing the fleet in the console or by using the AWS CLI to update the run-time configuration.

Ports

The **Ports** tab displays the fleet's connection permissions, including IP address and port setting ranges. You can change connection permissions by either editing the fleet in the console or using the AWS CLI to update the fleet's port settings.

Logs

The **Logs** tab lists the locations of log files that GameLift uploads at the end of a game session. Log paths can only be specified when creating a new fleet from the AWS CLI.

View Data on Game and Player Sessions

You can view information about the games running on a fleet and as well as individual players. For more information about game sessions and player sessions, see How Players Connect to Games (p. 5).

To view game session data

1. In the GameLift console, open the detail page for the fleet you want to study. (Choose **Fleets** in the menu bar and click on a fleet name.)
2. Open the **Game sessions** tab. This tab lists all game sessions hosted on the fleet along with summary information.
3. Click a game session to view additional information about the game session as well as a list of players that were connected to the game.

Game sessions

A summary of your game session information is displayed at the top of the page and includes:

- **Status** – The status of the game session. Valid statuses include:
  - **Activating** – GameLift has created a game session and passed your game properties to the game server process. The game server interprets the game properties and calls back to GameLift when it is ready for potential player sessions to connect.
  - **Active** – The game session can support game play with zero or more player sessions connected to it.
  - **Terminated** – The game session has ended, and player sessions are no longer permitted to connect to the terminated game session.
- **Name** – The name automatically generated for the game session.
- **IP address** – For game sessions with a status of **Activating** or **Active**, the IP address used to connect to the game.
- **Port** – The port number used to connect to the game session.
- **Player sessions** – The number of players currently connected to the game session along with the total number of player slots in the game session. For example, the value **10 of 15** indicates that of the 15 available slots in the game, 10 are filled and 5 are open.
- **Player session creation policy** – The policy that determines whether new players can connect to the game. Values are **Accept all** or **Deny all**. For more information, see the `GameSession` object.
- **Uptime** – The total length of time the game session has been running.
- **Date created** – The date and time the game session was created.

Player sessions

The following player session data is collected for each game session:

- **Status** – The status of the player session. Options include:
  - **Reserved** – Player session has been reserved, but the player has not yet connected.
  - **Active** – Player session is currently connected to the game server.
  - **Completed** – Player session has ended; player is no longer connected.
  - **Timed Out** – Player session was reserved, but the player failed to connect.
- **ID** – The identifier assigned to the player session.
- **Player ID** – A unique identifier for the player. Click this ID to get additional player information.

- **Start time** – The time the player connected to the game session.
- **End time** – The time the player disconnected from the game session.
- **Total time** – The total length of time the player has been active in the player session.

Player information

View additional information for a selected player, including a list of all games the player connected to across all fleets in the current region. This information includes the status, start and end times, and total connected time for each player session. You can click to view data for the relevant game sessions and fleets.

View Your Aliases

You can view information on all of the fleet aliases you have created and take actions on them on the Aliases page. Aliases shown include only those created for the selected region.

Alias Catalog

All created aliases are shown on the Aliases catalog page. To view the Aliases page, choose **Aliases** from the GameLift console's menu bar.

The Aliases page provides summary information on all builds, including type. From this page you can:

- Create a new alias. click **Create alias**.
- Filter and sort the aliases list. Use the controls at the top of the table.
- View alias details. Click an alias name to open the alias detail page.

Alias Detail

Access an alias's detail page from either the console dashboard or the Aliases catalog page by clicking the alias name. The Alias detail page displays a summary of information on the alias.

From this page you can:

- Edit an alias, including changing the name, description, and the fleet ID the alias is associated with. Click **Actions: Edit alias**.
- View information on the fleet the alias is currently associated with. This includes the fleet's status and current utilization (active game sessions and players).
- Delete an alias. Click **Actions: Delete alias**.

Alias detail information includes:

- **Type** – The routing option for the alias, which can be one of these:
  - **Simple** – A simple alias routes a player to games on an associated fleet. You can update the alias to point to a different fleet at any time.
  - **Terminal** – A terminal alias does not point to a fleet. Instead it passes a message back to the client. This alias type is useful for gracefully notifying players when a set of game servers is no longer available. For example, a terminal alias might inform players that their game clients are out of date and provide upgrade options.
- **Alias ID** – The unique number used to identify the alias.

- **Description** – The description of the alias.
- **Date created** – The date and time the alias was created.

Logging Amazon GameLift API Calls with AWS CloudTrail

Amazon GameLift is integrated with AWS CloudTrail, a service that captures all of the API calls made by or on behalf of GameLift in your AWS account and delivers the log files to an Amazon Simple Storage Service (Amazon S3) bucket that you specify. CloudTrail captures API calls from the GameLift console or from the GameLift API. Using the information collected by CloudTrail, you can determine what request was made to GameLift, the source IP address from which the request was made, who made the request, when it was made, and so on. To learn more about CloudTrail, including how to configure and enable it, see the *AWS CloudTrail User Guide*.

GameLift Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, API calls made to GameLift actions are tracked in log files. GameLift records are written together with other AWS service records in a log file. CloudTrail determines when to create and write to a new file based on a time period and file size.

All GameLift actions are logged by CloudTrail. For example, calls to `CreateGameSession`, `CreatePlayerSession` and `UpdateGameSession` generate entries in the CloudTrail log files. For the complete list of actions, see the *Amazon GameLift API Reference*.

Every log entry contains information about who generated the request. The user identity information in the log helps you determine whether the request was made with root or IAM user credentials, with temporary security credentials for a role or federated user, or by another AWS service. For more information, see the `userIdentity` field in the CloudTrail Event Reference.

You can store your log files in your S3 bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted with Amazon S3 server-side encryption (SSE).

You can choose to have CloudTrail publish Amazon Simple Notification Service (Amazon SNS) notifications when new log files are delivered if you want to take quick action upon log file delivery. For more information, see Configuring Amazon SNS Notifications.

You can also aggregate GameLift log files from multiple AWS regions and multiple AWS accounts into a single S3 bucket. For more information, see Aggregating CloudTrail Log Files to a Single Amazon S3 Bucket.

Understanding GameLift Log File Entries

CloudTrail log files can contain one or more log entries where each entry is made up of multiple JSON-formatted events. A log entry represents a single request from any source and includes information about the requested action, any parameters, the date and time of the action, and so on. The log entries are not guaranteed to be in any particular order. That is, they are not an ordered stack trace of the public API calls.

The following example shows a CloudTrail log entry that demonstrates the `CreateFleet` and `DescribeFleetAttributes` actions.


```
{
    "Records": [
        {
            "eventVersion": "1.04",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "AIDACKCEVSQ6C2EXAMPLE",
                "arn": "arn:aws:iam::111122223333:user/myUserName",
                "accountId": "111122223333",
                "accessKeyId": AKIAIOSFODNN7EXAMPLE",
                "userName": "myUserName"
            },
            "eventTime": "2015-12-29T23:40:15Z",
            "eventSource": "gamelift.amazonaws.com",
            "eventName": "CreateFleet",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "[]",
            "requestParameters": {
                "buildId": "build-92b6e8af-37a2-4c10-93bd-4698ea23de8d",
                "eC2InboundPermissions": [
                    {
                        "ipRange": "10.24.34.0/23",
                        "fromPort": 1935,
                        "protocol": "TCP",
                        "toPort": 1935
                    }
                ],
                "logPaths": [
                    "C:\\game\\serverErr.log",
                    "C:\\game\\serverOut.log"
                ],
                "eC2InstanceType": "c4.large",
                "serverLaunchPath": "C:\\game\\MyServer.exe",
                "description": "Test fleet",
                "serverLaunchParameters": "-paramX=baz",
                "name": "My_Test_Server_Fleet"
            },
            "responseElements": {
                "fleetAttributes": {
                    "fleetId": "fleet-0bb84136-4f69-4bb2-bfec-a9b9a7c3d52e",
                    "serverLaunchPath": "C:\\game\\MyServer.exe",
                    "status": "NEW",
                    "logPaths": [
                        "C:\\game\\serverErr.log",
                        "C:\\game\\serverOut.log"
                    ],
```

```
                    "description": "Test fleet",
                    "serverLaunchParameters": "-paramX=baz",
                    "creationTime": "Dec 29, 2015 11:40:14 PM",
                    "name": "My_Test_Server_Fleet",
                    "buildId": "build-92b6e8af-37a2-4c10-93bd-4698ea23de8d"
                }
            },
            "requestID": "824a2a4b-ae85-11e5-a8d6-61d5cafb25f2",
            "eventID": "c8fbea01-fbf9-4c4e-a0fe-ad7dc205ce11",
            "eventType": "AwsApiCall",
            "recipientAccountId": "111122223333"
        },
        {
            "eventVersion": "1.04",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "AIDACKCEVSQ6C2EXAMPLE",
                "arn": "arn:aws:iam::111122223333:user/myUserName",
                "accountId": "111122223333",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "userName": "myUserName"
            },
            "eventTime": "2015-12-29T23:40:15Z",
            "eventSource": "gamelift.amazonaws.com",
            "eventName": "DescribeFleetAttributes",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "192.0.2.0",
            "userAgent": "[]",
            "requestParameters": {
                "fleetIds": [
                    "fleet-0bb84136-4f69-4bb2-bfec-a9b9a7c3d52e"
                ]
            },
            "responseElements": null,
            "requestID": "82e7f0ec-ae85-11e5-a8d6-61d5cafb25f2",
            "eventID": "11daabcb-0094-49f2-8b3d-3a63c8bad86f",
            "eventType": "AwsApiCall",
            "recipientAccountId": "111122223333"
        },
    ]
}
```

Document History for Amazon GameLift

The following table describes important changes to the Amazon GameLift documentation.

Change	API Version	Description	Date

New features:
• Remote access to GameLift fleet instances
	AWS SDK: 2016-11-18	See full release notes.
		Developer Guide:
		• New topics:
		  • Remotely Access Fleet Instances (p. 48) – How to get access and remotely connect to GameLift instances.
		  • Debug Fleet Creation Issues (p. 40) – Troubleshooting tips for new fleets that fail to activate.
		Service API Reference:
		• For remote access:
		  • GetInstanceAccess (new)
		  • InstanceAccess (new)
		  • InstanceCredentials (new)
			November 18, 2016

New features: • Resource creation protection • Access to instance data Updates and corrections: • Additional help for Linux.	AWS SDK: 2016-10-13	See full release notes for full description of this release, as well as changes to the AWS SDK and the *Service API Reference*. **Developer Guide:** • Revised topics: • How Amazon GameLift Works (p. 2) – Added description of resource	October 13, 2016

Change	API Version	Description	Date
		protection, and improved description of capacity handling. • Added Linux-specific help: Package a Build (p. 34) – Install scripts for Linux. Upload a Build (p. 35) – New Linux examples. Create a Fleet (Console) (p. 38) – New launch path example for Linux. **Service API Reference:** • CreateFleet and UpdateFleetAttributes – New ResourceCreationLimitPolicy parameter. • ResourceCreationLimitPolicy (new) • CreateGameSession – New CreatorId parameter. • DescribeInstances (new)	
New features: • Game servers can now run on Linux	AWS SDK: 2016-09-01 AWS SDK for C++: 1.10.61 Server SDK for C++: 3.1.0	See full release notes. **Developer Guide:** • New topics: • Amazon GameLift SDKs (p. 7) – Reference topic describing all GameLift SDKs, including supported languages and operating systems. **Service API Reference:** • New OS parameters were added to the following: • `upload-build (p. 35)` (CLI only) • `CreateBuild()` • `Build` • `FleetAttributes`	September 1, 2016

Change	API Version	Description	Date

New features:
• Game session search
• Customized health checks
 	
Updates:
• Expanded support for capacity allocation (multiple processes per fleet instance)
• GameLift Server SDK for C++ now available for download
• All APIs for game client integration is now included in the AWS SDK.
	AWS SDK: 2016-08-04
	
	AWS SDK for C++: 0.12.16
	
	Server SDK for C++: 3.0.7
		See full release notes.
		
		Developer Guide:
		• New topics:
		  • Amazon GameLift Server API (C++) Reference (p. 15) – Complete reference documentation.
		  • Run Multiple Processes on a Fleet (p. 46) – Technical overview of capacity allocation and how to configure a fleet to run multiple processes.
		  • Tools and Resources (p. 6) – Comprehensive list of tools & resources, including SDK version compatibility.
		• Revised topics:
		  • How Players Connect to Games (p. 5) – Expanded topic describes features related to game sessions, including the new search feature.
		  • Add Amazon GameLift to Your Game Server (p. 13) – Integration steps have been revised for use with version 3.0.7 Server SDK for C++.
		  • Add Amazon GameLift to Your Game Client (p. 25) – Integration steps have been revised for use with the AWS SDK for C++.
		
		Service API Reference:
		• `SearchGameSessions()` (new)
			August 4, 2016

Change	API Version	Description	Date

Updates:
• New server process health metrics
• Revised processes for fleet capacity allocation and game server launch settings
• Revised build packaging instructions
	AWS SDK: 2016-06-28	**Developer Guide:**
		• Revised topics:
		  • Package a Build (p. 34) – Description now reflects how GameLift handles an `install.bat` file in a game build.
		  • Create a Fleet (Console) (p. 38) and Create a Fleet (AWS CLI) (p. 39) – Instructions for creating a fleet now cover capacity allocation using a runtime configuration.
		  • View Fleet Details (p. 55) and View Data on Game and Player Sessions (p. 60) – Console page descriptions now reflect current metrics and scaling tabs.
		  • Amazon GameLift and Game Client/Server Interactions (p. 30) – Descriptions and diagram (p. 33) have been corrected to use callback function names from the samples, and to clarify that the `onProcessTerminate()`callback refers to shutting down a game server, not a game session.
		
		Service API Reference:
		• For new capacity allocation:
		  • `CreateFleet()` – Runtime configuration added.
		  • `DescribeRuntimeConfiguration` (new)
		  • `UpdateRuntimeConfiguration` (new)
		• For game server launch process:
		  • `GameSession` – Port number added. `PlayerSession` – Port number added.
		• For health metrics:
		  • `FleetUtilization` – New count added for active server processes.
			June 28, 2016

Change	API Version	Description	Date

New features:
• Autoscaling
• Game session protection
• Fleet capacity limits
	2016-03-10	**Developer Guide:**
		• New topics:
		  • Set Up Fleet Autoscaling (p. 43) – How to set up and manage autoscaling policies.
		  • Change Fleet Capacity (p. 41) – How to change the number of instances in a fleet and set limits.
		  • How Amazon GameLift Works (p. 2) – A technical overview of how GameLift manages game deployment across virtual resources.
		• Revised topics:
		  • Create a Fleet (Console) (p. 38) – Revised to include settings for game session protection and safe scaling.
		• Other changes:
		  • Lumberyard-GameLift tutorial was moved to the GameDev Tutorials repository.
		
		Service API Reference:
		• For autoscaling:
		  • `PutScalingPolicy`
		  • `DescribeScalingPolicies`
		  • `DeleteScalingPolicy`
		• For game session protection:
		  • `DescribeGameSessionDetails`
		  • `CreateFleet` (revised)
		  • `UpdateFleetAttributes` (revised)
		  • `DescribeFleetAttributes` (revised)
		  • `UpdateGameSession` (revised)
		• For fleet capacity limits:
		  • `UpdateFleetCapacity` (revised)
		  • `DescribeFleetCapacity` (revised)
			March 10, 2016

Service launch	2016-02-09	Developer Guide and API Reference for the GameLift service released on AWS.	February 9, 2016