

# Contents

[Azure SignalR Service documentation](#)

[Overview](#)

[About Azure SignalR Service](#)

[Quickstarts](#)

[ASP.NET Core - C#](#)

[ASP.NET - C#](#)

[Azure Functions - JavaScript](#)

[Azure Functions - C#](#)

[Azure Functions - Java](#)

[Azure Functions - Python](#)

[Azure SignalR Service deployment - ARM template](#)

[REST API](#)

[Tutorials](#)

[Build a serverless real-time app with authentication](#)

[Build a Blazor Server chat app](#)

[Samples](#)

[Code samples](#)

[Azure CLI](#)

[Concepts](#)

[Azure SignalR Service internals](#)

[Service mode](#)

[Scale ASP.NET Core SignalR](#)

[Serverless](#)

[Build real-time apps with Azure Functions](#)

[Develop and configure serverless SignalR Service apps](#)

[SignalR Service bindings for Azure Functions reference](#)

[Upstream settings](#)

[Availability](#)

[Gracefully shutdown your app server.](#)

## Security

### Authorization

Authorize access with Azure Active Directory

### Authentication

Authenticate from an application

Authenticate with a managed identity

## High availability

Resiliency and disaster recovery

## Messages and connections

## Performance considerations

### Authentication

### Event handling

### Security

Security baseline

Security controls by Azure Policy

## How-to guides

### Scale

Single instance

Multiple instances

### Secure

Access key rotation

Use Azure Service Tags

Use Azure Private Endpoints

Manage network access control

Secure outbound traffic through Shared Private Endpoints

Use managed identity

### Integrate

Azure Functions bindings

Event Grid integration

### Monitor

Use diagnostic logs to monitor SignalR Service

Audit compliance using Azure Policy

## Troubleshooting

- [Troubleshooting guides](#)
- [Troubleshooting methods](#)
- [Troubleshooting with live trace tool](#)

## References

- [Azure CLI](#)
- [REST API](#)
- [ASP.NET Core SignalR](#)
- [Azure SignalR Service Protocol](#)
- [Azure Policy built-ins](#)

## Resources

- [Solutions](#)
- [Customer stories](#)
- [SignalR roadmap](#)
- [Service updates](#)
- [Region availability](#)
- [FAQs](#)
- [SignalR Service subscription level limits](#)
- [Pricing](#)
- [Videos](#)
- [StackOverflow](#)
- [Twitter](#)
- [ASP.NET forums](#)

# What is Azure SignalR Service?

11/2/2020 • 3 minutes to read • [Edit Online](#)

Azure SignalR Service simplifies the process of adding real-time web functionality to applications over HTTP. This real-time functionality allows the service to push content updates to connected clients, such as a single page web or mobile application. As a result, clients are updated without the need to poll the server, or submit new HTTP requests for updates.

This article provides an overview of Azure SignalR Service.

## What is Azure SignalR Service used for?

Any scenario that requires pushing data from server to client in real time, can use Azure SignalR Service.

Traditional real-time features that often require polling from server, can also use Azure SignalR Service.

Azure SignalR Service has been used in a wide variety of industries, for any application type that requires real-time content updates. We list some examples that are good to use Azure SignalR Service:

- **High frequency data updates:** gaming, voting, polling, auction.
- **Dashboards and monitoring:** company dashboard, financial market data, instant sales update, multi-player game leader board, and IoT monitoring.
- **Chat:** live chat room, chat bot, on-line customer support, real-time shopping assistant, messenger, in-game chat, and so on.
- **Real-time location on map:** logistic tracking, delivery status tracking, transportation status updates, GPS apps.
- **Real time targeted ads:** personalized real time push ads and offers, interactive ads.
- **Collaborative apps:** coauthoring, whiteboard apps and team meeting software.
- **Push notifications:** social network, email, game, travel alert.
- **Real-time broadcasting:** live audio/video broadcasting, live captioning, translating, events/news broadcasting.
- **IoT and connected devices:** real-time IoT metrics, remote control, real-time status, and location tracking.
- **Automation:** real-time trigger from upstream events.

## What are the benefits using Azure SignalR Service?

### Standard based:

SignalR provides an abstraction over a number of techniques used for building real-time web applications.

[WebSockets](#) is the optimal transport, but other techniques like [Server-Sent Events \(SSE\)](#) and Long Polling are used when other options aren't available. SignalR automatically detects and initializes the appropriate transport based on the features supported on the server and client.

### Native ASP.NET Core support:

SignalR Service provides native programming experience with ASP.NET Core and ASP.NET. Developing new SignalR application with SignalR Service, or migrating from existing SignalR based application to SignalR Service requires minimal efforts. SignalR Service also supports ASP.NET Core's new feature, Server-side Blazor.

### Broad client support:

SignalR Service works with a broad range of clients, such as web and mobile browsers, desktop apps, mobile

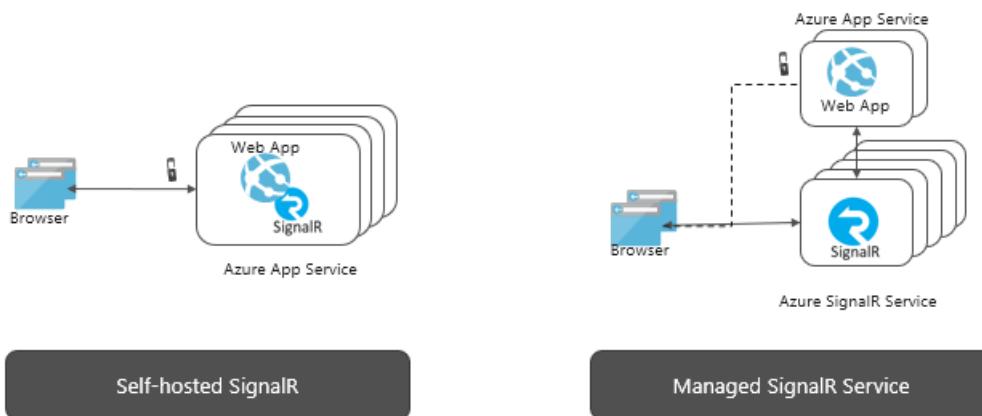
apps, server process, IoT devices, and game consoles. SignalR Service offers SDKs in different languages. In addition to native ASP.NET Core or ASP.NET C# SDKs, SignalR Service also provides JavaScript client SDK, to enable web clients, and many JavaScript frameworks. Java client SDK is also supported for Java applications, including Android native apps. SignalR Service supports REST API, and serverless through integrations with Azure Functions and Event Grid.

#### Handle large-scale client connections:

SignalR Service is designed for large-scale real-time applications. SignalR Service allows multiple instances to work together to scale to millions of client connections. The service also supports multiple global regions for sharding, high availability, or disaster recovery purposes.

#### Remove the burden to self-host SignalR:

Compared to self-hosted SignalR applications, switching to SignalR Service will remove the need to manage back planes that handle the scales and client connections. The fully managed service also simplifies web applications and saves hosting cost. SignalR Service offers global reach and world-class data center and network, scales to millions of connections, guarantees SLA, while providing all the compliance and security at Azure standard.



#### Offer rich APIs for different messaging patterns:

SignalR Service allows the server to send messages to a particular connection, all connections, or a subset of connections that belong to a specific user, or have been placed in an arbitrary group.

## How to use Azure SignalR Service

There are many different ways to program with Azure SignalR Service, as some of the samples listed here:

- [Scale an ASP.NET Core SignalR App](#) - Integrate Azure SignalR Service with an ASP.NET Core SignalR application to scale out to hundreds of thousands of connections.
- [Build serverless real-time apps](#) - Use Azure Functions' integration with Azure SignalR Service to build serverless real-time applications in languages such as JavaScript, C#, and Java.
- [Send messages from server to clients via REST API](#) - Azure SignalR Service provides REST API to enable applications to post messages to clients connected with SignalR Service, in any REST capable programming languages.

# Quickstart: Create a chat room by using SignalR Service

4/22/2021 • 10 minutes to read • [Edit Online](#)

Azure SignalR Service is an Azure service that helps developers easily build web applications with real-time features. This service was originally based on [SignalR for ASP.NET Core 2.1](#), but now supports later versions.

This article shows you how to get started with the Azure SignalR Service. In this quickstart, you'll create a chat application by using an ASP.NET Core MVC web app. This app will make a connection with your Azure SignalR Service resource to enable real-time content updates. You'll host the web application locally and connect with multiple browser clients. Each client will be able to push content updates to all other clients.

You can use any code editor to complete the steps in this quickstart. One option is [Visual Studio Code](#), which is available on the Windows, macOS, and Linux platforms.

The code for this tutorial is available for download in the [AzureSignalR-samples GitHub repository](#). Also, you can create the Azure resources used in this quickstart by following [Create a SignalR Service script](#).

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

## Prerequisites

- Install the [.NET Core SDK](#).
- Download or clone the [AzureSignalR-sample GitHub repository](#).

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Create an Azure SignalR resource

1. To create an Azure SignalR Service resource, first sign in to the [Azure portal](#). In the upper-left side of the page, select **+ Create a resource**. In the **Search the Marketplace** text box, enter **SignalR Service**.
2. Select **SignalR Service** in the results, and select **Create**.
3. On the new **SignalR** settings page, add the following settings for your new SignalR resource:

NAME	RECOMMENDED VALUE	DESCRIPTION
Resource name	<i>testsignalr</i>	Enter a unique resource name to use for the SignalR resource. The name must be a string of 1 to 63 characters and contain only numbers, letters, and the hyphen (-) character. The name cannot start or end with the hyphen character, and consecutive hyphen characters are not valid.

Name	Recommended Value	Description
Subscription	Choose your subscription	Select the Azure subscription that you want to use to test SignalR. If your account has only one subscription, it's automatically selected and the <b>Subscription</b> drop-down isn't displayed.
Resource group	Create a resource group named <i>SignalRTTestResources</i>	Select or create a resource group for your SignalR resource. This group is useful for organizing multiple resources that you might want to delete at the same time by deleting the resource group. For more information, see <a href="#">Using resource groups to manage your Azure resources</a> .
Location	<i>East US</i>	Use <b>Location</b> to specify the geographic location in which your SignalR resource is hosted. For the best performance, we recommend that you create the resource in the same region as other components of your application.
Pricing tier	<i>Free</i>	Currently, <b>Free</b> and <b>Standard</b> options are available.
Pin to dashboard	<input checked="" type="checkbox"/>	Select this box to have the resource pinned to your dashboard so it's easier to find.

4. Select **Review + create**. Wait for the validation to complete.
5. Select **Create**. The deployment might take a few minutes to complete.
6. After the deployment is complete, select **Keys** under **SETTINGS**. Copy your connection string for the primary key. You'll use this string later to configure your app to use the Azure SignalR Service resource.

The connection string will have the following form:

```
Endpoint=<service_endpoint>;AccessKey=<access_key>;
```

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Create an ASP.NET Core web app

In this section, you use the [.NET Core command-line interface \(CLI\)](#) to create an ASP.NET Core MVC web app project. The advantage of using the .NET Core CLI over Visual Studio is that it's available across the Windows, macOS, and Linux platforms.

1. Create a folder for your project. This quickstart uses the *E:\Testing\chattest* folder.
2. In the new folder, run the following command to create the project:

```
dotnet new mvc
```

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Add Secret Manager to the project

In this section, you'll add the [Secret Manager tool](#) to your project. The Secret Manager tool stores sensitive data for development work outside your project tree. This approach helps prevent the accidental sharing of app secrets in source code.

1. Open your `.csproj` file. Add a `DotNetCliToolReference` element to include

`Microsoft.Extensions.SecretManager.Tools`. Also add a `UserSecretsId` element as shown in the following code for `chattest.csproj`, and save the file.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <UserSecretsId>SignalRChatRoomEx</UserSecretsId>
</PropertyGroup>

<ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.4" />
    <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="2.0.2" />
</ItemGroup>

</Project>
```

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Add Azure SignalR to the web app

1. Add a reference to the `Microsoft.Azure.SignalR` NuGet package by running the following command:

```
dotnet add package Microsoft.Azure.SignalR
```

2. Run the following command to restore packages for your project:

```
dotnet restore
```

3. Add a secret named `Azure:SignalR:ConnectionString` to Secret Manager.

This secret will contain the connection string to access your SignalR Service resource.

`Azure:SignalR:ConnectionString` is the default configuration key that SignalR looks for to establish a connection. Replace the value in the following command with the connection string for your SignalR Service resource.

You must run this command in the same directory as the `.csproj` file.

```
dotnet user-secrets set Azure:SignalR:ConnectionString "<Your connection string>"
```

Secret Manager will be used only for testing the web app while it's hosted locally. In a later tutorial, you'll deploy the chat web app to Azure. After the web app is deployed to Azure, you'll use an application setting instead of storing the connection string with Secret Manager.

This secret is accessed with the Configuration API. A colon (:) works in the configuration name with the

Configuration API on all supported platforms. See [Configuration by environment](#).

4. Open *Startup.cs* and update the `ConfigureServices` method to use Azure SignalR Service by calling the `AddSignalR()` and `AddAzureSignalR()` methods:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR()
        .AddAzureSignalR();
}
```

By not passing a parameter to `AddAzureSignalR()`, this code uses the default configuration key for the SignalR Service resource connection string. The default configuration key is `Azure:SignalR:ConnectionString`.

5. In *Startup.cs*, update the `Configure` method by replacing it with the following code.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();
    app.UseFileServer();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHub<ChatHub>("/chat");
    });
}
```

## Add a hub class

In SignalR, a hub is a core component that exposes a set of methods that can be called from the client. In this section, you define a hub class with two methods:

- `Broadcast` : This method broadcasts a message to all clients.
- `Echo` : This method sends a message back to the caller.

Both methods use the `Clients` interface that the ASP.NET Core SignalR SDK provides. This interface gives you access to all connected clients, so you can push content to your clients.

1. In your project directory, add a new folder named *Hub*. Add a new hub code file named *ChatHub.cs* to the new folder.
2. Add the following code to *ChatHub.cs* to define your hub class and save the file.

Update the namespace for this class if you used a project name that differs from *SignalR.Mvc*.

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalR.Mvc
{
    public class ChatHub : Hub
    {
        public Task BroadcastMessage(string name, string message) =>
            Clients.All.SendAsync("broadcastMessage", name, message);

        public Task Echo(string name, string message) =>
            Clients.Client(Context.ConnectionId)
                .SendAsync("echo", name, $"{message} (echo from server)");
    }
}
```

## Add the client interface for the web app

The client user interface for this chat room app will consist of HTML and JavaScript in a file named *index.html* in the *wwwroot* directory.

Copy the *css/site.css* file from the *wwwroot* folder of the [samples repository](#). Replace your project's *css/site.css* with the one you copied.

Here's the main code of *index.html*:

```
<!DOCTYPE html>
<html>
<head>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap.min.css" rel="stylesheet" />
    <link href="css/site.css" rel="stylesheet" />
    <title>Azure SignalR Group Chat</title>
</head>
<body>
    <h2 class="text-center" style="margin-top: 0; padding-top: 30px; padding-bottom: 30px;">Azure SignalR
    Group Chat</h2>
    <div class="container" style="height: calc(100% - 110px);">
        <div id="messages" style="background-color: whitesmoke; "></div>
        <div style="width: 100%; border-left-style: ridge; border-right-style: ridge;">
            <textarea id="message"
                style="width: 100%; padding: 5px 10px; border-style: hidden;"
                placeholder="Type message and press Enter to send..."></textarea>
        </div>
        <div style="overflow: auto; border-style: ridge; border-top-style: hidden;">
            <button class="btn-warning pull-right" id="echo">Echo</button>
            <button class="btn-success pull-right" id="sendmessage">Send</button>
        </div>
    </div>
    <div class="modal alert alert-danger fade" id="myModal" tabindex="-1" role="dialog" aria-
labelledby="myModalLabel">
        <div class="modal-dialog" role="document">
            <div class="modal-content">
                <div class="modal-header">
                    <div>Connection Error...</div>
                    <div><strong style="font-size: 1.5em;">Hit Refresh/F5</strong> to rejoin. ;)</div>
                </div>
            </div>
        </div>
    </div>
</div>

<!--Reference the SignalR library. -->
<script src="https://cdn.jsdelivr.net/npm/@microsoft/signalr@3.1.8/dist/browser/signalr.min.js">
</script>

<!--Add script to update the page and send messages.-->
<script type="text/javascript">
    document.addEventListener('DOMContentLoaded', function () {

        const generateRandomName = () =>
            Math.random().toString(36).substring(2, 10);

        let username = generateRandomName();
        const promptMessage = 'Enter your name:';
        do {
            username = prompt(promptMessage, username);
            if (!username || username.startsWith('_') || username.indexOf('<') > -1 || username.indexOf('>') > -1) {
                username = '';
                promptMessage = 'Invalid input. Enter your name:';
            }
        }
    })
</script>
```

```

        } while (!username)

        const messageInput = document.getElementById('message');
        messageInput.focus();

        function createMessageEntry(encodedName, encodedMsg) {
            var entry = document.createElement('div');
            entry.classList.add("message-entry");
            if (encodedName === "_SYSTEM_") {
                entry.innerHTML = encodedMsg;
                entry.classList.add("text-center");
                entry.classList.add("system-message");
            } else if (encodedName === "_BROADCAST_") {
                entry.classList.add("text-center");
                entry.innerHTML = `<div class="text-center broadcast-message">${encodedMsg}</div>`;
            } else if (encodedName === username) {
                entry.innerHTML = `<div class="message-avatar pull-right">${encodedName}</div>` +
                    `<div class="message-content pull-right">${encodedMsg}</div>`;
            } else {
                entry.innerHTML = `<div class="message-avatar pull-left">${encodedName}</div>` +
                    `<div class="message-content pull-left">${encodedMsg}</div>`;
            }
            return entry;
        }

        function bindConnectionMessage(connection) {
            var messageCallback = function (name, message) {
                if (!message) return;
                var encodedName = name;
                var encodedMsg = message.replace(/&/g, "&").replace(/</g, "<").replace(/>/g,
                ">");
                var messageEntry = createMessageEntry(encodedName, encodedMsg);

                var messageBox = document.getElementById('messages');
                messageBox.appendChild(messageEntry);
                messageBox.scrollTop = messageBox.scrollHeight;
            };
            connection.on('broadcastMessage', messageCallback);
            connection.on('echo', messageCallback);
            connection.onclose(onConnectionError);
        }

        function onConnected(connection) {
            console.log('connection started');
            connection.send('broadcastMessage', '_SYSTEM_', username + ' JOINED');
            document.getElementById('sendmessage').addEventListener('click', function (event) {
                if (messageInput.value) {
                    connection.send('broadcastMessage', username, messageInput.value);
                }

                messageInput.value = '';
                messageInput.focus();
                event.preventDefault();
            });

            document.getElementById('message').addEventListener('keypress', function (event) {
                if (event.keyCode === 13) {
                    event.preventDefault();
                    document.getElementById('sendmessage').click();
                    return false;
                }
            });
            document.getElementById('echo').addEventListener('click', function (event) {
                connection.send('echo', username, messageInput.value);

                messageInput.value = '';
                messageInput.focus();
                event.preventDefault();
            });
        }
    }
}

```

```

        function onConnectionError(error) {
            if (error && error.message) {
                console.error(error.message);
            }
            var modal = document.getElementById('myModal');
            modal.classList.add('in');
            modal.style = 'display: block;';
        }

        const connection = new signalR.HubConnectionBuilder()
            .withUrl('/chat')
            .build();
        bindConnectionMessage(connection);
        connection.start()
            .then(() => onConnected(connection))
            .catch(error => console.error(error.message));
    });
</script>
</body>
</html>

```

The code in *index.htm* calls `HubConnectionBuilder.build()` to make an HTTP connection to the Azure SignalR resource.

If the connection is successful, that connection is passed to `bindConnectionMessage`, which adds event handlers for incoming content pushes to the client.

`HubConnection.start()` starts communication with the hub. Then, `onConnected()` adds the button event handlers. These handlers use the connection to allow this client to push content updates to all connected clients.

## Add a development runtime profile

In this section, you'll add a development runtime environment for ASP.NET Core. For more information, see [Work with multiple environments in ASP.NET Core](#).

1. Create a folder named *Properties* in your project.
2. Add a new file named *launchSettings.json* to the folder, with the following content, and save the file.

```
{
  "profiles": {
    "ChatRoom": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000/"
    }
  }
}
```

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Build and run the app locally

1. To build the app by using the .NET Core CLI, run the following command in the command shell:

```
dotnet build
```

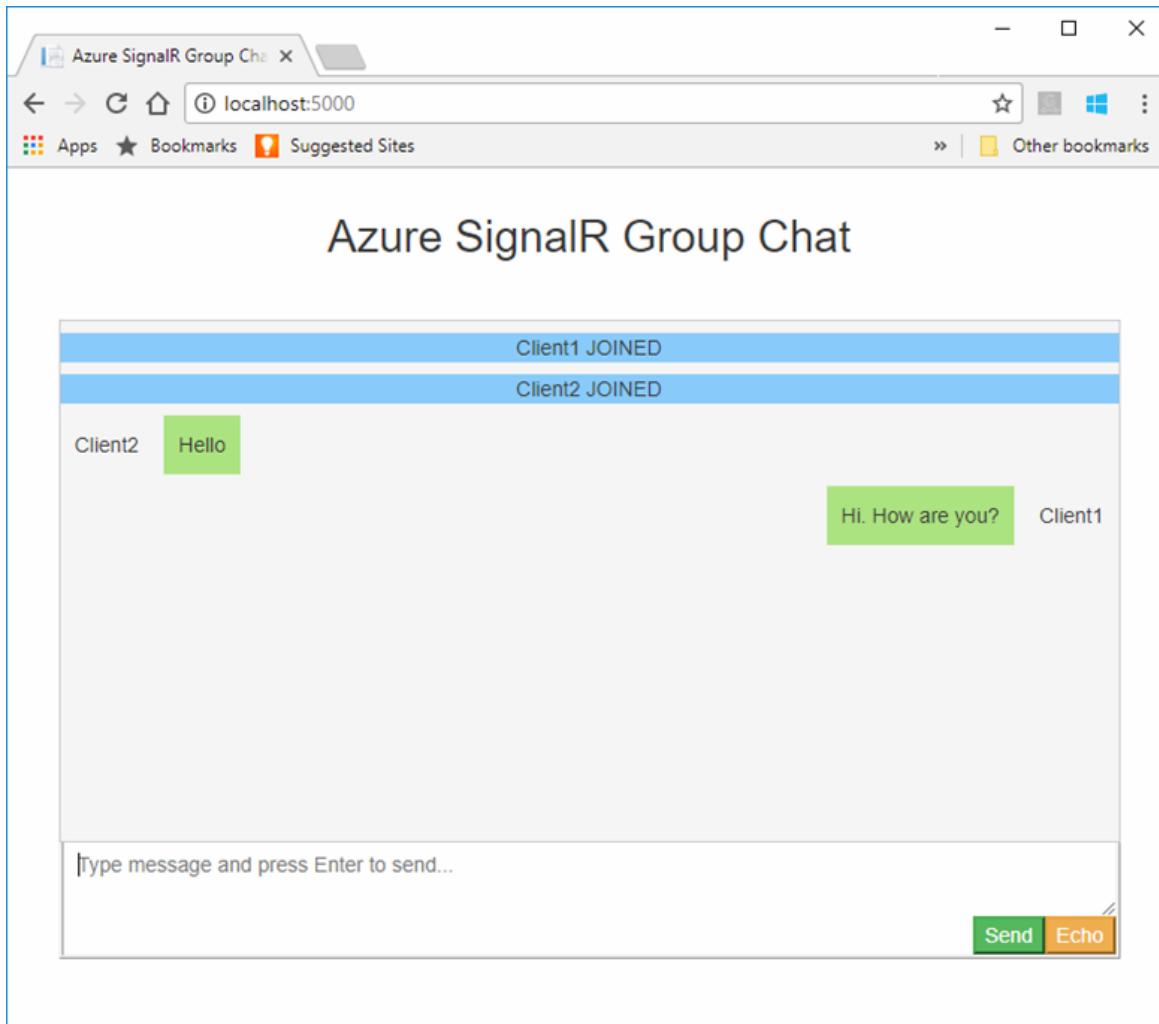
2. After the build successfully finishes, run the following command to run the web app locally:

```
dotnet run
```

The app will be hosted locally on port 5000, as configured in our development runtime profile:

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: E:\Testing\chattest
```

3. Open two browser windows. In each browser, go to <http://localhost:5000>. You're prompted to enter your name. Enter a client name for both clients and test pushing message content between both clients by using the **Send** button.



Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Clean up resources

If you'll continue to the next tutorial, you can keep the resources created in this quickstart and reuse them.

If you're finished with the quickstart sample application, you can delete the Azure resources created in this

quickstart to avoid charges.

#### IMPORTANT

Deleting a resource group is irreversible and includes all the resources in that group. Make sure that you don't accidentally delete the wrong resource group or resources. If you created the resources for hosting this sample in an existing resource group that contains resources you want to keep, you can delete each resource individually from its blade instead of deleting the resource group.

Sign in to the [Azure portal](#) and select **Resource groups**.

In the **Filter by name** text box, type the name of your resource group. The instructions for this quickstart used a resource group named *SignalRTestResources*. On your resource group in the result list, select the ellipsis (...) > **Delete resource group**.

The screenshot shows the Microsoft Azure portal's Resource groups blade. The left sidebar has a 'Resource groups' item highlighted with a red box. The main area shows a table with one item: 'SignalRTestResources'. The 'NAME' column has a checkbox checked. To the right of the table are 'SUBSCRIPTION' and 'LOCATION' filters. Below the table, there are two buttons: 'Delete resource group' and '...'. Both buttons are highlighted with red boxes. The top navigation bar includes a search bar, notification icons, and other standard portal controls.

You're asked to confirm the deletion of the resource group. Enter the name of your resource group to confirm, and select **Delete**.

After a few moments, the resource group and all of its resources are deleted.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Next steps

In this quickstart, you created a new Azure SignalR Service resource. You then used it with an ASP.NET Core web app to push content updates in real time to multiple connected clients. To learn more about using Azure SignalR Service, continue to the tutorial that demonstrates authentication.

[Azure SignalR Service authentication](#)

# Quickstart: Create a chat room with ASP.NET and SignalR Service

4/22/2021 • 5 minutes to read • [Edit Online](#)

Azure SignalR Service is based on [SignalR for ASP.NET Core 2.1](#), which is **not** 100% compatible with ASP.NET SignalR. Azure SignalR Service re-implemented ASP.NET SignalR data protocol based on the latest ASP.NET Core technologies. When using Azure SignalR Service for ASP.NET SignalR, some ASP.NET SignalR features are no longer supported, for example, Azure SignalR does not replay messages when the client reconnects. Also, the Forever Frame transport and JSONP are not supported. Some code changes and proper version of dependent libraries are needed to make ASP.NET SignalR application work with SignalR Service.

Refer to the [version differences doc](#) for a complete list of feature comparison between ASP.NET SignalR and ASP.NET Core SignalR.

In this quickstart, you will learn how to get started with the ASP.NET and Azure SignalR Service for a similar [Chat Room application](#).

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

## Prerequisites

- [Visual Studio 2019](#)
- [.NET Framework 4.6.1](#)
- [ASP.NET SignalR 2.4.1](#)

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Sign in to Azure

Sign in to the [Azure portal](#) with your Azure account.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Create an Azure SignalR Service instance

Your application will connect to a SignalR Service instance in Azure.

1. Select the New button found on the upper left-hand corner of the Azure portal. In the New screen, type *SignalR Service* in the search box and press enter.

The screenshot shows the Azure Marketplace search results for 'signalr service'. The search bar at the top contains the text 'signalr service'. Below the search bar, there is a 'Filter' button and a 'Results' section. The results table has columns for NAME, PUBLISHER, and CATEGORY. There are five items listed:

NAME	PUBLISHER	CATEGORY
SignalR Service	Microsoft	Web
Signal Sciences - BYOL	Signal Sciences	Compute
MX Public Store Edition Server (with SQL)	NDL Software	Compute
SmartMessage-Connect	ODC Business Solutions	Compute
MX Public Store Edition Server (without SQL)	NDL Software	Compute

2. Select **SignalR Service** from the search results, then select **Create**.

3. Enter the following settings.

SETTING	SUGGESTED VALUE	DESCRIPTION
Resource name	Globally unique name	Name that identifies your new SignalR Service instance. Valid characters are <code>a-z</code> , <code>0-9</code> , and <code>-</code> .
Subscription	Your subscription	The subscription under which this new SignalR Service instance is created.
Resource Group	myResourceGroup	Name for the new resource group in which to create your SignalR Service instance.
Location	West US	Choose a <a href="#">region</a> near you.
Pricing tier	Free	Try Azure SignalR Service for free.
Unit count	Not applicable	Unit count specifies how many connections your SignalR Service instance can accept. It is only configurable in the Standard tier.
Service mode	Serverless	For use with Azure Functions or REST API.

The screenshot shows the Microsoft Azure portal interface for creating a new SignalR Service. The top navigation bar includes 'Microsoft Azure', a search bar, and various account and service icons. The main title is 'SignalR' with a 'SignalR' logo. The left sidebar has a vertical list of service icons. The main content area is titled 'SignalR' and shows the 'Basics' tab selected. It displays fields for 'Project Details' (Subscription: Ignite Demo, Resource group: (New) serverlesschat), 'Service Details' (Resource Name: serverlesschat, Region: South Central US, Pricing tier: Free, Unit count: 1, ServiceMode: Serverless), and a summary of the deployment. At the bottom are buttons for 'Review + create', 'Next : Tags >', and 'Download a template for automation'.

4. Select **Create** to start deploying the SignalR Service instance.

5. After the instance is deployed, open it in the portal and locate its Settings page. Change the Service Mode

setting to *Serverless* only if you are using Azure SignalR Service through Azure Functions binding or REST API. Leave it in *Classic* or *Default* otherwise.

*Serverless* mode is not supported for ASP.NET SignalR applications. Always use *Default* or *Classic* for the Azure SignalR Service instance.

You can also create Azure resources used in this quickstart with [Create a SignalR Service script](#).

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Clone the sample application

While the service is deploying, let's switch to working with code. Clone the [sample app from GitHub](#), set the SignalR Service connection string, and run the application locally.

1. Open a git terminal window. Change to a folder where you want to clone the sample project.
2. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/aspnet/AzureSignalR-samples.git
```

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Configure and run Chat Room web app

1. Start Visual Studio and open the solution in the *aspnet-samples/ChatRoom*/folder of the cloned repository.
2. In the browser where the Azure portal is opened, find and select the instance you created.
3. Select **Keys** to view the connection strings for the SignalR Service instance.
4. Select and copy the primary connection string.
5. Now set the connection string in the *web.config* file.

```
<configuration>
<connectionStrings>
    <add name="Azure:SignalR:ConnectionString" connectionString="<Replace By Your Connection String>"/>
</connectionStrings>
...
</configuration>
```

6. In *Startup.cs*, instead of calling `MapSignalR()`, you need to call `MapAzureSignalR({YourApplicationName})` and pass in connection string to make the application connect to the service instead of hosting SignalR by itself. Replace `{YourApplicationName}` to the name of your application. This name is a unique name to distinguish this application from your other applications. You can use `this.GetType().FullName` as the value.

```
public void Configuration(IAppBuilder app)
{
    // Any connection or hub wire up and configuration should go here
    app.MapAzureSignalR(this.GetType().FullName);
}
```

You also need to reference the service SDK before using these APIs. Open the [Tools | NuGet Package Manager | Package Manager Console](#) and run command:

```
Install-Package Microsoft.Azure.SignalR.AspNet
```

Other than these changes, everything else remains the same, you can still use the hub interface you're already familiar with to write business logic.

#### NOTE

In the implementation an endpoint `/signalr/negotiate` is exposed for negotiation by Azure SignalR Service SDK. It will return a special negotiation response when clients try to connect and redirect clients to service endpoint defined in the connection string.

7. Press F5 to run the project in debug mode. You can see the application runs locally. Instead of hosting a SignalR runtime by application itself, it now connects to the Azure SignalR Service.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart with the following steps so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left, and then select the resource group you created. Alternatively, you may use the search box to find the resource group by its name.
2. In the window that opens, select the resource group, and then click **Delete resource group**.
3. In the new window, type the name of the resource group to delete, and then click **Delete**.

#### IMPORTANT

Deleting a resource group is irreversible and that the resource group and all the resources in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the resources for hosting this sample inside an existing resource group that contains resources you want to keep, you can delete each resource individually from their respective blades instead of deleting the resource group.

Sign in to the [Azure portal](#) and click **Resource groups**.

In the **Filter by name...** textbox, type the name of your resource group. The instructions for this quickstart used a resource group named *SignalRTTestResources*. On your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Microsoft Azure portal interface. On the left, the navigation bar includes 'Create a resource', 'All services', 'FAVORITES' (with 'Resource groups' highlighted by a red box), 'Dashboard', 'All resources', 'Recent', and 'App Services'. The main content area is titled 'Resource groups' and shows a table with one item selected: 'SignalRTTestResources'. The table has columns for NAME, SUBSCRIPTION, and LOCATION. A red box highlights the 'NAME' column header and the row for 'SignalRTTestResources'. To the right of the row, there is a 'Delete resource group' button and a three-dot menu icon, both also highlighted with red boxes.

After a few moments, the resource group and all of its contained resources are deleted.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Next steps

In this quickstart, you created a new Azure SignalR Service resource and used it with an ASP.NET web app. Next, learn how to develop real-time applications using Azure SignalR Service with ASP.NET Core.

[Azure SignalR Service with ASP.NET Core](#)

# Quickstart: Use JavaScript to create an App showing GitHub star count with Azure Functions and SignalR Service

6/22/2021 • 5 minutes to read • [Edit Online](#)

Azure SignalR Service lets you easily add real-time functionality to your application and Azure Functions is a serverless platform that lets you run your code without managing any infrastructure. In this quickstart, learn how to use SignalR Service and Azure Functions to build a serverless application with JavaScript to broadcast messages to clients.

## NOTE

You can get all codes mentioned in the article from [GitHub](#)

## Prerequisites

- A code editor, such as [Visual Studio Code](#)
- An Azure account with an active subscription. [Create an account for free.](#)
- [Azure Functions Core Tools](#), version 2 or above. Used to run Azure Function apps locally.
- [Node.js](#), version 10.x

## NOTE

The examples should work with other versions of Node.js, see [Azure Functions runtime versions documentation](#) for more information.

## NOTE

This quickstart can be run on macOS, Windows, or Linux.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Log in to Azure

Sign in to the Azure portal at <https://portal.azure.com/> with your Azure account.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Create an Azure SignalR Service instance

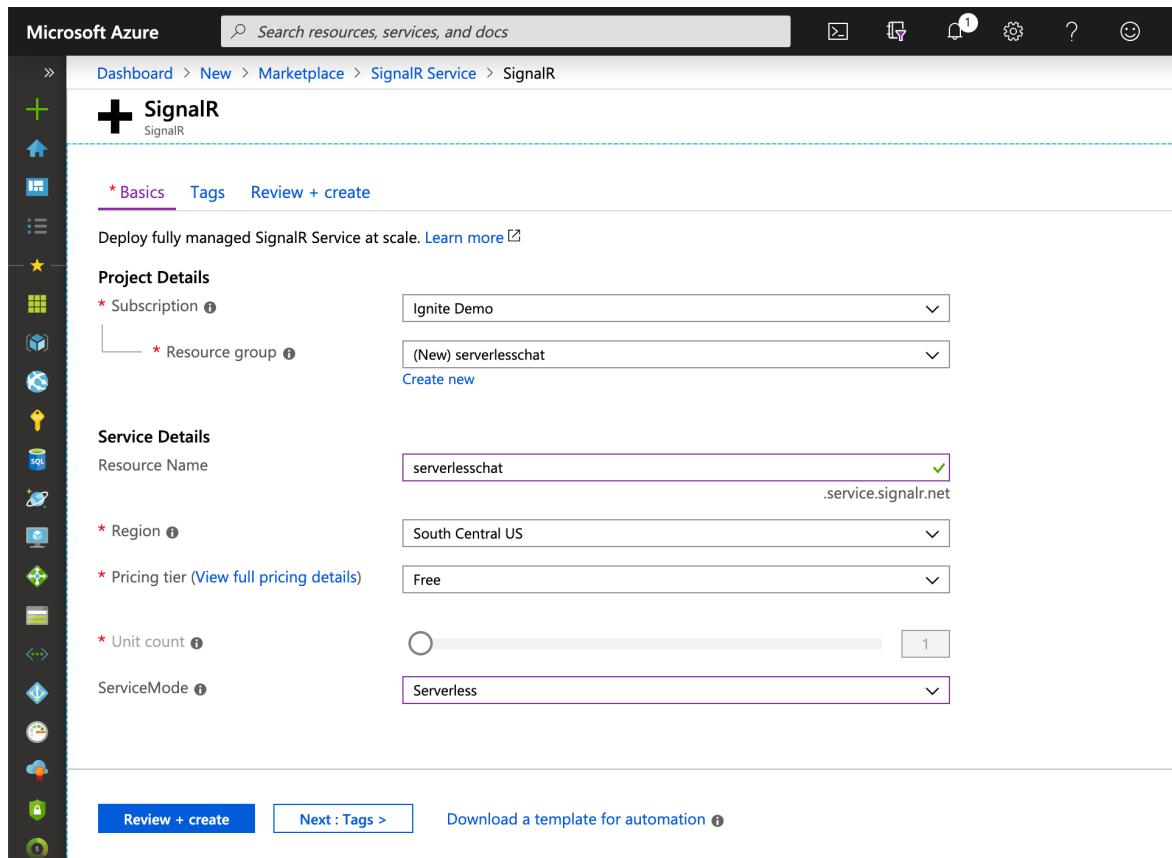
Your application will connect to a SignalR Service instance in Azure.

1. Select the New button found on the upper left-hand corner of the Azure portal. In the New screen, type *SignalR Service* in the search box and press enter.

2. Select **SignalR Service** from the search results, then select **Create**.

3. Enter the following settings.

SETTING	SUGGESTED VALUE	DESCRIPTION
<b>Resource name</b>	Globally unique name	Name that identifies your new SignalR Service instance. Valid characters are <code>a-z</code> , <code>0-9</code> , and <code>-</code> .
<b>Subscription</b>	Your subscription	The subscription under which this new SignalR Service instance is created.
<b>Resource Group</b>	myResourceGroup	Name for the new resource group in which to create your SignalR Service instance.
<b>Location</b>	West US	Choose a <a href="#">region</a> near you.
<b>Pricing tier</b>	Free	Try Azure SignalR Service for free.
<b>Unit count</b>	Not applicable	Unit count specifies how many connections your SignalR Service instance can accept. It is only configurable in the Standard tier.
<b>Service mode</b>	Serverless	For use with Azure Functions or REST API.



4. Select **Create** to start deploying the SignalR Service instance.
5. After the instance is deployed, open it in the portal and locate its Settings page. Change the Service Mode setting to *Serverless* only if you are using Azure SignalR Service through Azure Functions binding or REST API. Leave it in *Classic* or *Default* otherwise.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Setup and run the Azure Function locally

1. Make sure you have Azure Function Core Tools installed. And create an empty directory and navigate to the directory with command line.

```
# Initialize a function project
func init --worker-runtime javascript
```

2. After you initialize a project, you need to create functions. In this sample, we need to create 3 functions.

- a. Run the following command to create a `index` function, which will host a web page for client.

```
func new -n index -t HttpTrigger
```

Open `index/index.js` and copy the following codes.

```

var fs = require('fs').promises

module.exports = async function (context, req) {
    const path = context.executionContext.functionDirectory + '/../content/index.html'
    try {
        var data = await fs.readFile(path);
        context.res = {
            headers: {
                'Content-Type': 'text/html'
            },
            body: data
        }
        context.done()
    } catch (error) {
        context.log.error(err);
        context.done(err);
    }
}

```

- b. Create a `negotiate` function for clients to get access token.

```
func new -n negotiate -t SignalRNegotiateHTTPTrigger
```

Open `negotiate/function.json` and copy the following json codes:

```
{
    "disabled": false,
    "bindings": [
        {
            "authLevel": "anonymous",
            "type": "httpTrigger",
            "direction": "in",
            "methods": [
                "post"
            ],
            "name": "req",
            "route": "negotiate"
        },
        {
            "type": "http",
            "direction": "out",
            "name": "res"
        },
        {
            "type": "signalRConnectionInfo",
            "name": "connectionInfo",
            "hubName": "serverless",
            "connectionStringSetting": "AzureSignalRConnectionString",
            "direction": "in"
        }
    ]
}
```

- c. Create a `broadcast` function to broadcast messages to all clients. In the sample, we use time trigger to broadcast messages periodically.

```
func new -n broadcast -t TimerTrigger
```

Open `broadcast/function.json` and copy the following codes.

```
{
  "bindings": [
    {
      "name": "myTimer",
      "type": "timerTrigger",
      "direction": "in",
      "schedule": "*/* * * * *"
    },
    {
      "type": "signalR",
      "name": "signalRMessages",
      "hubName": "serverless",
      "connectionStringSetting": "AzureSignalRConnectionString",
      "direction": "out"
    }
  ]
}
```

Open `broadcast/index.js` and copy the following codes.

```
var https = require('https');

module.exports = function (context) {
  var req = https.request("https://api.github.com/repos/azure/azure-signalr", {
    method: 'GET',
    headers: {'User-Agent': 'serverless'}
  }, res => {
    var body = "";

    res.on('data', data => {
      body += data;
    });
    res.on("end", () => {
      var jbody = JSON.parse(body);
      context.bindings.signalRMessages = [{
        "target": "newMessage",
        "arguments": [ `Current star count of https://github.com/Azure/azure-signalr
is: ${jbody['stargazers_count']}` ]
      }];
      context.done();
    });
  }).on("error", (error) => {
    context.log(error);
    context.res = {
      status: 500,
      body: error
    };
    context.done();
  });
  req.end();
}
```

3. The client interface of this sample is a web page. Considered we read HTML content from `content/index.html` in `index` function, create a new file `index.html` in `content` directory. And copy the following content.

```

<html>

<body>
    <h1>Azure SignalR Serverless Sample</h1>
    <div id="messages"></div>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/3.1.7/signalr.min.js">
    </script>
    <script>
        let messages = document.querySelector('#messages');
        const apiBaseUrl = window.location.origin;
        const connection = new signalR.HubConnectionBuilder()
            .withUrl(apiBaseUrl + '/api')
            .configureLogging(signalR.LogLevel.Information)
            .build();
        connection.on('newMessage', (message) => {
            document.getElementById("messages").innerHTML = message;
        });

        connection.start()
            .catch(console.error);
    </script>
</body>

</html>

```

4. It's almost done now. The last step is to set a connection string of the SignalR Service to Azure Function settings.

- In the browser where the Azure portal is opened, confirm the SignalR Service instance you deployed earlier was successfully created by searching for its name in the search box at the top of the portal. Select the instance to open it.

RESOURCES		All 5 results
	serverlesschat	Azure Cosmos DB account
	serverlesschat	SignalR
	serverlesschat	Storage account
	serverlesschat	App Service
	serverlesschat2	Storage account
RESOURCE GROUPS		All 1 results
	serverlesschat	Resource group
SERVICES		0 results
MARKETPLACE		0 results
DOCUMENTATION		0 results
Searching all subscriptions. <a href="#">Change</a>		

- Select **Keys** to view the connection strings for the SignalR Service instance.

The screenshot shows the Azure portal interface for a SignalR service named 'serverlesschat'. The left sidebar has a 'Keys' section selected. The main pane displays information about access keys, including a host name ('serverlesschat.service.signalr.net') and a primary key ('PRIMARY-KEY'). The 'PRIMARY-CONNECTION-STRING' is shown below and is highlighted with a red box.

c. Copy the primary connection string. And execute the command below.

```
func settings add AzureSignalRConnectionString '<signalr-connection-string>'
```

5. Run the Azure Function in local:

```
func start
```

After Azure Function running locally. Use your browser to visit <http://localhost:7071/api/index> and you can see the current start count. And if you star or unstar in the GitHub, you will get a start count refreshing every few seconds.

#### NOTE

SignalR binding needs Azure Storage, but you can use local storage emulator when the Function is running locally. If you got some error like

```
There was an error performing a read operation on the Blob Storage Secret Repository. Please ensure the 'AzureWebJobsStorage' connection string is valid.
```

You need to download and enable [Storage Emulator](#)

Having issues? Try the [troubleshooting guide](#) or [let us know](#)

## Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart with the following steps so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left, and then select the resource group you created. Alternatively, you may use the search box to find the resource group by its name.
2. In the window that opens, select the resource group, and then click **Delete resource group**.
3. In the new window, type the name of the resource group to delete, and then click **Delete**.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Next steps

In this quickstart, you built and ran a real-time serverless application in local. Learn more how to use SignalR Service bindings for Azure Functions. Next, learn more about how to bi-directional communicating between clients and Azure Function with SignalR Service.

[SignalR Service bindings for Azure Functions](#)

[Bi-directional communicating in Serverless](#)

[Deploy Azure Functions with VS Code](#)

# Quickstart: Create an App showing GitHub star count with Azure Functions and SignalR Service using C#

6/22/2021 • 5 minutes to read • [Edit Online](#)

Azure SignalR Service lets you easily add real-time functionality to your application. Azure Functions is a serverless platform that lets you run your code without managing any infrastructure. In this quickstart, learn how to use SignalR Service and Azure Functions to build a serverless application with C# to broadcast messages to clients.

## NOTE

You can get all codes mentioned in the article from [GitHub](#)

## Prerequisites

If you don't already have Visual Studio Code installed, you can download and use it for free(<https://code.visualstudio.com/Download>).

You may also run this tutorial on the command line (macOS, Windows, or Linux) using the [Azure Functions Core Tools](#)). Also the [.NET Core SDK](#), and your favorite code editor.

If you don't have an Azure subscription, [create one for free](#) before you begin.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Log in to Azure and create SignalR Service instance

Sign in to the Azure portal at <https://portal.azure.com/> with your Azure account.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Create an Azure SignalR Service instance

Your application will connect to a SignalR Service instance in Azure.

1. Select the New button found on the upper left-hand corner of the Azure portal. In the New screen, type *SignalR Service* in the search box and press enter.

2. Select **SignalR Service** from the search results, then select **Create**.

3. Enter the following settings.

SETTING	SUGGESTED VALUE	DESCRIPTION
<b>Resource name</b>	Globally unique name	Name that identifies your new SignalR Service instance. Valid characters are <code>a-z</code> , <code>0-9</code> , and <code>-</code> .
<b>Subscription</b>	Your subscription	The subscription under which this new SignalR Service instance is created.
<b>Resource Group</b>	myResourceGroup	Name for the new resource group in which to create your SignalR Service instance.
<b>Location</b>	West US	Choose a <a href="#">region</a> near you.
<b>Pricing tier</b>	Free	Try Azure SignalR Service for free.
<b>Unit count</b>	Not applicable	Unit count specifies how many connections your SignalR Service instance can accept. It is only configurable in the Standard tier.
<b>Service mode</b>	Serverless	For use with Azure Functions or REST API.

The screenshot shows the Microsoft Azure portal interface for creating a new SignalR Service instance. The left sidebar has a 'SignalR' icon selected. The main area shows the 'SignalR' blade with 'Project Details' and 'Service Details' sections. In the 'Service Details' section, the 'Resource Name' is 'serverlesschat', which generates a '.service.signalr.net' URL. The 'Region' is 'South Central US' and the 'Pricing tier' is 'Free'. The 'Unit count' is set to 1. The 'ServiceMode' is set to 'Serverless'. At the bottom, there are buttons for 'Review + create', 'Next : Tags >', and 'Download a template for automation'.

4. Select **Create** to start deploying the SignalR Service instance.
5. After the instance is deployed, open it in the portal and locate its Settings page. Change the Service Mode setting to *Serverless* only if you are using Azure SignalR Service through Azure Functions binding or REST API. Leave it in *Classic* or *Default* otherwise.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Setup and run the Azure Function locally

1. Make sure you have Azure Function Core Tools installed. And create an empty directory and navigate to the directory with command line.

```
# Initialize a function project
func init --worker-runtime dotnet

# Add SignalR Service package reference to the project
dotnet add package Microsoft.Azure.WebJobs.Extensions.SignalRService
```

2. After you initialize a project. Create a new file with name *Function.cs*. Add the following code to *Function.cs*.

```

using System;
using System.IO;
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Azure.WebJobs.Extensions.SignalRService;
using Newtonsoft.Json;

namespace CSharp
{
    public static class Function
    {
        private static HttpClient httpClient = new HttpClient();

        [FunctionName("index")]
        public static IActionResult Index([HttpTrigger(AuthorizationLevel.Anonymous)]HttpRequest req,
        ExecutionContext context)
        {
            var path = Path.Combine(context.FunctionAppDirectory, "content", "index.html");
            return new ContentResult
            {
                Content = File.ReadAllText(path),
                ContentType = "text/html",
            };
        }

        [FunctionName("negotiate")]
        public static SignalRConnectionInfo Negotiate(
            [HttpTrigger(AuthorizationLevel.Anonymous)] HttpRequest req,
            [SignalRConnectionInfo(HubName = "serverlessSample")] SignalRConnectionInfo
connectionInfo)
        {
            return connectionInfo;
        }

        [FunctionName("broadcast")]
        public static async Task Broadcast([TimerTrigger("*/5 * * * *")] TimerInfo myTimer,
        [SignalR(HubName = "serverlessSample")] IAsyncCollector<SignalRMessage> signalRMessages)
        {
            var request = new HttpRequestMessage(HttpMethod.Get,
"https://api.github.com/repos/azure/azure-signalr");
            request.Headers.UserAgent.ParseAdd("Serverless");
            var response = await httpClient.SendAsync(request);
            var result = JsonConvert.DeserializeObject<GitResult>(await
response.Content.ReadAsStringAsync());
            await signalRMessages.AddAsync(
                new SignalRMessage
                {
                    Target = "newMessage",
                    Arguments = new[] { $"Current start count of https://github.com/Azure/azure-
signalr is: {result.StartCount}" }
                });
        }

        private class GitResult
        {
            [JsonProperty]
            [JsonProperty("stargazers_count")]
            public string StartCount { get; set; }
        }
    }
}

```

These codes have three functions. The `Index` is used to get a website as client. The `Negotiate` is used for client to get access token. The `Broadcast` is periodically get start count from GitHub and broadcast messages to all clients.

3. The client interface of this sample is a web page. Considered we read HTML content from `content/index.html` in `GetHomePage` function, create a new file `index.html` in `content` directory. And copy the following content.

```
<html>

<body>
    <h1>Azure SignalR Serverless Sample</h1>
    <div id="messages"></div>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/3.1.7/signalr.min.js">
    </script>
    <script>
        let messages = document.querySelector('#messages');
        const apiBaseUrl = window.location.origin;
        const connection = new signalR.HubConnectionBuilder()
            .withUrl(apiBaseUrl + '/api')
            .configureLogging(signalR.LogLevel.Information)
            .build();
        connection.on('newMessage', (message) => {
            document.getElementById("messages").innerHTML = message;
        });

        connection.start()
            .catch(console.error);
    </script>
</body>

</html>
```

4. It's almost done now. The last step is to set a connection string of the SignalR Service to Azure Function settings.

- a. In the browser where the Azure portal is opened, confirm the SignalR Service instance you deployed earlier was successfully created by searching for its name in the search box at the top of the portal. Select the instance to open it.

The screenshot shows the Azure portal search results for the term "serverlesschat". The search bar at the top contains "serverlesschat". Below the search bar, there are three main sections: "RESOURCES", "RESOURCE GROUPS", and "SERVICES".

- RESOURCES:** Contains five items:
  - serverlesschat** (Azure Cosmos DB account)
  - serverlesschat** (SignalR)
  - serverlesschat** (Storage account)
  - serverlesschat** (App Service)
  - serverlesschat2** (Storage account)
- RESOURCE GROUPS:** Contains one item:
  - serverlesschat** (Resource group)
- SERVICES:** Contains zero items: "0 results".

At the bottom of the search results, there is a footer message: "Searching all subscriptions. [Change](#)".

- b. Select **Keys** to view the connection strings for the SignalR Service instance.

The screenshot shows the Azure portal interface for a SignalR service named 'serverlesschat'. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings, Keys (which is selected and highlighted in blue), Quickstart, and Scale. The main content area is titled 'serverlesschat - Keys' under 'SignalR'. It contains a search bar, a summary about access keys, and a table for primary and secondary keys. The 'Primary' key section shows the host name 'serverlesschat.service.signalr.net' and the key value 'PRIMARY-KEY'. Below it, the 'CONNECTION STRING' row shows 'PRIMARY-CONNECTION-STRING', which is also highlighted with a red box.

c. Copy the primary connection string. And execute the command below.

```
func settings add AzureSignalRConnectionString '<signalr-connection-string>'
```

5. Run the Azure Function in local:

```
func start
```

After Azure Function running locally. Use your browser to visit <http://localhost:7071/api/index> and you can see the current start count. And if you star or unstar in the GitHub, you will get a start count refreshing every few seconds.

#### NOTE

SignalR binding needs Azure Storage, but you can use local storage emulator when the Function is running locally. If you got some error like

```
There was an error performing a read operation on the Blob Storage Secret Repository. Please ensure the 'AzureWebJobsStorage' connection string is valid.
```

You need to download and enable [Storage Emulator](#)

Having issues? Try the [troubleshooting guide](#) or [let us know](#)

## Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart with the following steps so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left, and then select the resource group you created. Alternatively, you may use the search box to find the resource group by its name.
2. In the window that opens, select the resource group, and then click **Delete resource group**.
3. In the new window, type the name of the resource group to delete, and then click **Delete**.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Next steps

In this quickstart, you built and ran a real-time serverless application in local. Learn more how to use SignalR Service bindings for Azure Functions. Next, learn more about how to bi-directional communicating between clients and Azure Function with SignalR Service.

[SignalR Service bindings for Azure Functions](#)

[Bi-directional communicating in Serverless](#)

[Develop Azure Functions using Visual Studio](#)

# Quickstart: Use Java to create an App showing GitHub star count with Azure Functions and SignalR Service

6/22/2021 • 6 minutes to read • [Edit Online](#)

Azure SignalR Service lets you easily add real-time functionality to your application and Azure Functions is a serverless platform that lets you run your code without managing any infrastructure. In this quickstart, learn how to use SignalR Service and Azure Functions to build a serverless application with Java to broadcast messages to clients.

## NOTE

You can get all codes mentioned in the article from [GitHub](#)

## Prerequisites

- A code editor, such as [Visual Studio Code](#)
- An Azure account with an active subscription. [Create an account for free.](#)
- [Azure Functions Core Tools](#). Used to run Azure Function apps locally.

## NOTE

The required SignalR Service bindings in Java are only supported in Azure Function Core Tools version 2.4.419 (host version 2.0.12332) or above.

## NOTE

To install extensions, Azure Functions Core Tools requires the [.NET Core SDK](#) installed. However, no knowledge of .NET is required to build JavaScript Azure Function apps.

- [Java Developer Kit](#), version 11
- [Apache Maven](#), version 3.0 or above

## NOTE

This quickstart can be run on macOS, Windows, or Linux.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Log in to Azure

Sign in to the Azure portal at <https://portal.azure.com/> with your Azure account.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

# Create an Azure SignalR Service instance

Your application will connect to a SignalR Service instance in Azure.

1. Select the New button found on the upper left-hand corner of the Azure portal. In the New screen, type *SignalR Service* in the search box and press enter.

The screenshot shows the Azure Marketplace search interface. The search bar at the top contains the text "signalr service". Below the search bar, the results section is titled "Results". It displays a list of items with columns for "NAME", "PUBLISHER", and "CATEGORY". The first item in the list is "SignalR Service" by Microsoft, which is categorized under "Web". Other items listed include "Signal Sciences -BYOL" by Signal Sciences, "MX Public Store Edition Server (with SQL)" by NDL Software, "SmartMessage-Connect" by ODC Business Solutions, and "MX Public Store Edition Server (without SQL)" by NDL Software. The "Compute" category is also visible on the left sidebar.

2. Select **SignalR Service** from the search results, then select **Create**.

3. Enter the following settings.

SETTING	SUGGESTED VALUE	DESCRIPTION
Resource name	Globally unique name	Name that identifies your new SignalR Service instance. Valid characters are <code>a-z</code> , <code>0-9</code> , and <code>-</code> .
Subscription	Your subscription	The subscription under which this new SignalR Service instance is created.
Resource Group	myResourceGroup	Name for the new resource group in which to create your SignalR Service instance.
Location	West US	Choose a <a href="#">region</a> near you.
Pricing tier	Free	Try Azure SignalR Service for free.
Unit count	Not applicable	Unit count specifies how many connections your SignalR Service instance can accept. It is only configurable in the Standard tier.
Service mode	Serverless	For use with Azure Functions or REST API.

The screenshot shows the Microsoft Azure portal interface for creating a new SignalR Service. The left sidebar has a 'SignalR' icon. The main area has a title bar with 'Microsoft Azure' and a search bar. The breadcrumb navigation shows 'Dashboard > New > Marketplace > SignalR Service > SignalR'. The page title is 'SignalR'. Below the title, there are tabs for 'Basics', 'Tags', and 'Review + create'. A note says 'Deploy fully managed SignalR Service at scale.' with a 'Learn more' link. The 'Project Details' section includes 'Subscription' (Ignite Demo), 'Resource group' ((New) serverlesschat, with a 'Create new' link), and a 'Service Details' section with 'Resource Name' (serverlesschat), 'Region' (South Central US), 'Pricing tier' (Free), 'Unit count' (1), and 'ServiceMode' (Serverless). At the bottom are 'Review + create', 'Next : Tags >', and 'Download a template for automation'.

4. Select **Create** to start deploying the SignalR Service instance.
5. After the instance is deployed, open it in the portal and locate its Settings page. Change the Service Mode setting to *Serverless* only if you are using Azure SignalR Service through Azure Functions binding or REST API. Leave it in *Classic* or *Default* otherwise.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Configure and run the Azure Function app

1. Make sure you have Azure Function Core Tools, java (version 11 in the sample) and maven installed.

```
mvn archetype:generate -DarchetypeGroupId=com.microsoft.azure -DarchetypeArtifactId=azure-functions-archetype -DjavaVersion=11
```

Maven asks you for values needed to finish generating the project. You can provide the following values.

PROMPT	VALUE	DESCRIPTION
groupId	com.signalr	A value that uniquely identifies your project across all projects, following the <a href="#">package naming rules</a> for Java.
artifactId	java	A value that is the name of the jar, without a version number.
version	1.0-SNAPSHOT	Choose the default value.
package	com.signalr	A value that is the Java package for the generated function code. Use the default.

2. After you initialize a project. Go to the folder `src/main/java/com/signalr` and copy the following codes to

`Function.java`

```
package com.signalr;

import com.google.gson.Gson;
import com.microsoft.azure.functions.ExecutionContext;
import com.microsoft.azure.functions.HttpMethod;
import com.microsoft.azure.functions.HttpRequestMessage;
import com.microsoft.azure.functions.HttpResponseMessage;
import com.microsoft.azure.functions.HttpStatus;
import com.microsoft.azure.functions.annotation.AuthorizationLevel;
import com.microsoft.azure.functions.annotation.FunctionName;
import com.microsoft.azure.functions.annotation.HttpTrigger;
import com.microsoft.azure.functions.annotation.TimerTrigger;
import com.microsoft.azure.functions.signalr.*;
import com.microsoft.azure.functions.signalr.annotation.*;

import org.apache.commons.io.IOUtils;

import java.io.IOException;
import java.io.InputStream;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.net.http.HttpResponse.BodyHandlers;
import java.nio.charset.StandardCharsets;
import java.util.Optional;

public class Function {
    @FunctionName("index")
    public HttpResponseMessage run(
        @HttpTrigger(
            name = "req",
            methods = { HttpMethod.GET },
            authLevel = AuthorizationLevel.ANONYMOUS) HttpRequestMessage<Optional<String>>
    request,
        final ExecutionContext context) throws IOException {

        InputStream inputStream =
getClass().getClassLoader().getResourceAsStream("content/index.html");
        String text = IOUtils.toString(inputStream, StandardCharsets.UTF_8.name());
        return request.createResponseBuilder(HttpStatus.OK).header("Content-Type",
"text/html").body(text).build();
    }

    @FunctionName("negotiate")
    public SignalRConnectionInfo negotiate(
        @HttpTrigger(
            name = "req",
            methods = { HttpMethod.POST },
            authLevel = AuthorizationLevel.ANONYMOUS) HttpRequestMessage<Optional<String>> req,
        @SignalRConnectionInfoInput(
            name = "connectionInfo",
            hubName = "serverless") SignalRConnectionInfo connectionInfo) {

        return connectionInfo;
    }

    @FunctionName("broadcast")
    @SignalROutput(name = "$return", hubName = "serverless")
    public SignalRMessage broadcast(
        @TimerTrigger(name = "timeTrigger", schedule = "*/5 * * * *") String timerInfo) throws
IOException, InterruptedException {
}
```

```

HttpClient client = HttpClient.newHttpClient();
HttpRequest req =
HttpRequest.newBuilder().uri(URI.create("https://api.github.com/repos/azure/azure-signalr")).header("User-Agent", "serverless").build();
HttpResponse<String> res = client.send(req, BodyHandlers.ofString());
Gson gson = new Gson();
GitResult result = gson.fromJson(res.body(), GitResult.class);
return new SignalRMessage("newMessage", "Current start count of
https://github.com/Azure/azure-signalr is:".concat(result.stargazers_count));
}

class GitResult {
    public String stargazers_count;
}
}

```

3. Some dependencies need to be added. So open the `pom.xml` and add some dependency that used in codes.

```

<dependency>
    <groupId>com.microsoft.azure.functions</groupId>
    <artifactId>azure-functions-java-library-signalr</artifactId>
    <version>1.0.0</version>
</dependency>
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.4</version>
</dependency>
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.7</version>
</dependency>

```

4. The client interface of this sample is a web page. Considered we read HTML content from `content/index.html` in `index` function, create a new file `content/index.html` in `resources` directory. Your directory tree should look like this.

```

FunctionsProject
| - src
| | - main
| | | - java
| | | | - com
| | | | | - signalr
| | | | | | - Function.java
| | | | - resources
| | | | | - content
| | | | | | - index.html
| - pom.xml
| - host.json
| - local.settings.json

```

Open the `index.html` and copy the following content.

```

<html>

<body>
    <h1>Azure SignalR Serverless Sample</h1>
    <div id="messages"></div>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/3.1.7/signalr.min.js">
    </script>
    <script>
        let messages = document.querySelector('#messages');
        const apiBaseUrl = window.location.origin;
        const connection = new signalR.HubConnectionBuilder()
            .withUrl(apiBaseUrl + '/api')
            .configureLogging(signalR.LogLevel.Information)
            .build();
        connection.on('newMessage', (message) => {
            document.getElementById("messages").innerHTML = message;
        });

        connection.start()
            .catch(console.error);
    </script>
</body>

</html>

```

5. It's almost done now. The last step is to set a connection string of the SignalR Service to Azure Function settings.

- In the browser where the Azure portal is opened, confirm the SignalR Service instance you deployed earlier was successfully created by searching for its name in the search box at the top of the portal. Select the instance to open it.

RESOURCES		All 5 results
	serverlesschat	Azure Cosmos DB account
	serverlesschat	SignalR
	serverlesschat	Storage account
	serverlesschat	App Service
	serverlesschat2	Storage account
RESOURCE GROUPS		All 1 results
	serverlesschat	Resource group
SERVICES		0 results
MARKETPLACE		0 results
DOCUMENTATION		0 results
Searching all subscriptions. <a href="#">Change</a>		

- Select **Keys** to view the connection strings for the SignalR Service instance.

The screenshot shows the 'Keys' page of an Azure SignalR service. On the left, there's a sidebar with links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings, Keys (which is selected and highlighted in blue), Quickstart, and Scale. The main area has a search bar at the top. It contains sections for Host name (serverlesschat.service.signalr.net), Primary key (PRIMARY-KEY), and Connection string (PRIMARY-CONNECTION-STRING). The 'PRIMARY-CONNECTION-STRING' field is highlighted with a red border.

c. Copy the primary connection string. And execute the command below.

```
func settings add AzureSignalRConnectionString '<signalr-connection-string>'  
# Also we need to set AzureWebJobsStorage as Azure Function's requirement  
func settings add AzureWebJobsStorage 'UseDevelopmentStorage=true'
```

6. Run the Azure Function in local:

```
mvn clean package  
mvn azure-functions:run
```

After Azure Function running locally. Use your browser to visit <http://localhost:7071/api/index> and you can see the current start count. And if you star or unstar in the GitHub, you will get a start count refreshing every few seconds.

#### NOTE

SignalR binding needs Azure Storage, but you can use local storage emulator when the Function is running locally.

If you got some error like

```
There was an error performing a read operation on the Blob Storage Secret Repository. Please  
ensure the 'AzureWebJobsStorage' connection string is valid.
```

You need to download and enable [Storage Emulator](#)

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart with the following steps so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left, and then select the resource group you created. Alternatively, you may use the search box to find the resource group by its name.
2. In the window that opens, select the resource group, and then click **Delete resource group**.
3. In the new window, type the name of the resource group to delete, and then click **Delete**.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Next steps

In this quickstart, you built and ran a real-time serverless application in local. Learn more how to use SignalR

Service bindings for Azure Functions. Next, learn more about how to bi-directional communicating between clients and Azure Function with SignalR Service.

[SignalR Service bindings for Azure Functions](#)

[Bi-directional communicating in Serverless](#)

[Create your first function with Java and Maven](#)

# Quickstart: Create an App showing GitHub star count with Azure Functions and SignalR Service using Python

6/22/2021 • 5 minutes to read • [Edit Online](#)

Azure SignalR Service lets you easily add real-time functionality to your application. Azure Functions is a serverless platform that lets you run your code without managing any infrastructure. In this quickstart, learn how to use SignalR Service and Azure Functions to build a serverless application with Python to broadcast messages to clients.

## NOTE

You can get all codes mentioned in the article from [GitHub](#)

## Prerequisites

This quickstart can be run on macOS, Windows, or Linux.

Make sure you have a code editor such as [Visual Studio Code](#) installed.

Install the [Azure Functions Core Tools](#) (version 2.7.1505 or higher) to run Python Azure Function apps locally.

Azure Functions requires [Python 3.6+](#). (See [Supported Python versions](#))

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Log in to Azure

Sign in to the Azure portal at <https://portal.azure.com/> with your Azure account.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Create an Azure SignalR Service instance

Your application will connect to a SignalR Service instance in Azure.

1. Select the New button found on the upper left-hand corner of the Azure portal. In the New screen, type *SignalR Service* in the search box and press enter.

2. Select **SignalR Service** from the search results, then select **Create**.

3. Enter the following settings.

SETTING	SUGGESTED VALUE	DESCRIPTION
<b>Resource name</b>	Globally unique name	Name that identifies your new SignalR Service instance. Valid characters are <code>a-z</code> , <code>0-9</code> , and <code>-</code> .
<b>Subscription</b>	Your subscription	The subscription under which this new SignalR Service instance is created.
<b>Resource Group</b>	myResourceGroup	Name for the new resource group in which to create your SignalR Service instance.
<b>Location</b>	West US	Choose a <a href="#">region</a> near you.
<b>Pricing tier</b>	Free	Try Azure SignalR Service for free.
<b>Unit count</b>	Not applicable	Unit count specifies how many connections your SignalR Service instance can accept. It is only configurable in the Standard tier.
<b>Service mode</b>	Serverless	For use with Azure Functions or REST API.

The screenshot shows the Microsoft Azure portal interface for creating a new SignalR Service. The top navigation bar includes 'Microsoft Azure', a search bar, and various icons for notifications and settings. The main page title is 'SignalR' under 'SignalR'. The left sidebar has a 'SignalR' icon and a list of other service categories like Storage, Functions, and Logic Apps. The main content area is titled 'SignalR' and shows the 'Basics' tab selected. It prompts to 'Deploy fully managed SignalR Service at scale' with a 'Learn more' link. The 'Project Details' section requires selecting a 'Subscription' (Ignite Demo) and a 'Resource group' ((New) serverlesschat). The 'Service Details' section includes fields for 'Resource Name' (serverlesschat), 'Region' (South Central US), 'Pricing tier' (Free), 'Unit count' (1), and 'ServiceMode' (Serverless). At the bottom, there are 'Review + create', 'Next : Tags >', and 'Download a template for automation' buttons.

4. Select **Create** to start deploying the SignalR Service instance.
5. After the instance is deployed, open it in the portal and locate its Settings page. Change the Service Mode setting to *Serverless* only if you are using Azure SignalR Service through Azure Functions binding or REST API. Leave it in *Classic* or *Default* otherwise.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Setup and run the Azure Function locally

1. Make sure you have Azure Function Core Tools installed. And create an empty directory and navigate to the directory with command line.

```
# Initialize a function project
func init --worker-runtime python
```

2. After you initialize a project, you need to create functions. In this sample, we need to create 3 functions.

- a. Run the following command to create a `index` function, which will host a web page for client.

```
func new -n index -t HttpTrigger
```

Open `index/__init__.py` and copy the following codes.

```

import os

import azure.functions as func


def main(req: func.HttpRequest) -> func.HttpResponse:
    f = open(os.path.dirname(os.path.realpath(__file__)) + '/../content/index.html')
    return func.HttpResponse(f.read(), mimetype='text/html')

```

- b. Create a `negotiate` function for clients to get access token.

```
func new -n negotiate -t SignalRNegotiateHTTPTrigger
```

Open `negotiate/function.json` and copy the following json codes:

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "post"
      ],
      "type": "http",
      "direction": "out",
      "name": "$return"
    },
    {
      "type": "signalRConnectionInfo",
      "name": "connectionInfo",
      "hubName": "serverless",
      "connectionStringSetting": "AzureSignalRConnectionString",
      "direction": "in"
    }
  ]
}
```

And open the `negotiate/__init__.py` and copy the following codes:

```

import azure.functions as func


def main(req: func.HttpRequest, connectionInfo) -> func.HttpResponse:
    return func.HttpResponse(connectionInfo)

```

- c. Create a `broadcast` function to broadcast messages to all clients. In the sample, we use time trigger to broadcast messages periodically.

```

func new -n broadcast -t TimerTrigger
# install requests
pip install requests

```

Open `broadcast/function.json` and copy the following codes.

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "name": "myTimer",
      "type": "timerTrigger",
      "direction": "in",
      "schedule": "*/* * * * *"
    },
    {
      "type": "signalR",
      "name": "signalRMessages",
      "hubName": "serverless",
      "connectionStringSetting": "AzureSignalRConnectionString",
      "direction": "out"
    }
  ]
}
```

Open `broadcast/__init__.py` and copy the following codes.

```
import requests
import json

import azure.functions as func


def main(myTimer: func.TimerRequest, signalRMessages: func.Out[str]) -> None:
    headers = {'User-Agent': 'serverless'}
    res = requests.get('https://api.github.com/repos/azure/azure-signalr', headers=headers)
    jres = res.json()

    signalRMessages.set(json.dumps({
        'target': 'newMessage',
        'arguments': [ 'Current star count of https://github.com/Azure/azure-signalr is: ' +
                      str(jres['stargazers_count']) ]
    }))
```

3. The client interface of this sample is a web page. Considered we read HTML content from

`content/index.html` in `index` function, create a new file `index.html` in `content` directory. And copy the following content.

```

<html>

<body>
    <h1>Azure SignalR Serverless Sample</h1>
    <div id="messages"></div>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/3.1.7/signalr.min.js">
    </script>
    <script>
        let messages = document.querySelector('#messages');
        const apiBaseUrl = window.location.origin;
        const connection = new signalR.HubConnectionBuilder()
            .withUrl(apiBaseUrl + '/api')
            .configureLogging(signalR.LogLevel.Information)
            .build();
        connection.on('newMessage', (message) => {
            document.getElementById("messages").innerHTML = message;
        });

        connection.start()
            .catch(console.error);
    </script>
</body>

</html>

```

4. It's almost done now. The last step is to set a connection string of the SignalR Service to Azure Function settings.

- In the browser where the Azure portal is opened, confirm the SignalR Service instance you deployed earlier was successfully created by searching for its name in the search box at the top of the portal. Select the instance to open it.

RESOURCES		All 5 results
	serverlesschat	Azure Cosmos DB account
	serverlesschat	SignalR
	serverlesschat	Storage account
	serverlesschat	App Service
	serverlesschat2	Storage account
RESOURCE GROUPS		All 1 results
	serverlesschat	Resource group
SERVICES		0 results
MARKETPLACE		0 results
DOCUMENTATION		0 results
Searching all subscriptions. <a href="#">Change</a>		

- Select **Keys** to view the connection strings for the SignalR Service instance.

The screenshot shows the 'serverlesschat - Keys' page in the Azure portal. On the left, there's a sidebar with links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings (which is selected), Keys (selected), Quickstart, and Scale. The main area has a search bar at the top. It explains the importance of access keys for SignalR clients. Below that, it shows the host name as 'serverlesschat.service.signalr.net'. Under 'Primary', it lists the key as 'PRIMARY-KEY' and the connection string as 'PRIMARY-CONNECTION-STRING'. The 'PRIMARY-CONNECTION-STRING' field is highlighted with a red border.

c. Copy the primary connection string. And execute the command below.

```
func settings add AzureSignalRConnectionString '<signalr-connection-string>'
```

5. Run the Azure Function in local:

```
func start
```

After Azure Function running locally. Use your browser to visit <http://localhost:7071/api/index> and you can see the current start count. And if you star or unstar in the GitHub, you will get a start count refreshing every few seconds.

#### NOTE

SignalR binding needs Azure Storage, but you can use local storage emulator when the Function is running locally. If you got some error like

```
There was an error performing a read operation on the Blob Storage Secret Repository. Please ensure the 'AzureWebJobsStorage' connection string is valid.
```

You need to download and enable [Storage Emulator](#)

## Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart with the following steps so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left, and then select the resource group you created. Alternatively, you may use the search box to find the resource group by its name.
2. In the window that opens, select the resource group, and then click **Delete resource group**.
3. In the new window, type the name of the resource group to delete, and then click **Delete**.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Next steps

In this quickstart, you built and ran a real-time serverless application in local. Learn more how to use SignalR Service bindings for Azure Functions. Next, learn more about how to bi-directional communicating between clients and Azure Function with SignalR Service.

[SignalR Service bindings for Azure Functions](#)

Bi-directional communicating in Serverless

Deploy Azure Functions with VS Code

# Quickstart: Use an ARM template to deploy Azure SignalR Service

6/9/2021 • 8 minutes to read • [Edit Online](#)

This quickstart describes how to use an Azure Resource Manager template (ARM template) to create an Azure SignalR Service. You can deploy the Azure SignalR Service through the Azure portal, PowerShell, or CLI.

An [ARM template](#) is a JavaScript Object Notation (JSON) file that defines the infrastructure and configuration for your project. The template uses declarative syntax. In declarative syntax, you describe your intended deployment without writing the sequence of programming commands to create the deployment.

If your environment meets the prerequisites and you're familiar with using ARM templates, select the **Deploy to Azure** button. The template will open in the Azure portal once you sign in.

 Deploy to Azure

## Prerequisites

- [Portal](#)
- [PowerShell](#)
- [CLI](#)

An Azure account with an active subscription. [Create one for free](#).

## Review the template

The template used in this quickstart is from [Azure Quickstart Templates](#).

```
{  
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",  
  "contentVersion": "1.0.0.0",  
  "parameters": {  
    "name": {  
      "type": "string",  
      "defaultValue": "[uniqueString(resourceGroup().id)]",  
      "metadata": {  
        "description": "The globally unique name of the SignalR resource to create."  
      }  
    },  
    "location": {  
      "type": "string",  
      "defaultValue": "[resourceGroup().location]",  
      "metadata": {  
        "description": "Location for the SignalR resource."  
      }  
    },  
    "pricingTier": {  
      "type": "string",  
      "defaultValue": "Standard_S1",  
      "allowedValues": [  
        "Free_F1",  
        "Standard_S1"  
      ],  
      "metadata": {  
        "description": "The pricing tier of the SignalR resource."  
      }  
    }  
  }  
}
```

```
        }
    },
    "capacity": {
        "type": "int",
        "defaultValue": 1,
        "allowedValues": [
            1,
            2,
            5,
            10,
            20,
            50,
            100
        ],
        "metadata": {
            "description": "The number of SignalR Unit."
        }
    },
    "serviceMode": {
        "type": "string",
        "defaultValue": "Default",
        "allowedValues": [
            "Default",
            "Serverless",
            "Classic"
        ],
        "metadata": {
            "description": "Visit https://github.com/Azure/azure-signalr/blob/dev/docs/faq.md#service-mode to understand SignalR Service Mode."
        }
    },
    "enableConnectivityLogs": {
        "type": "string",
        "defaultValue": "true",
        "allowedValues": [
            "true",
            "false"
        ]
    },
    "enableMessagingLogs": {
        "type": "string",
        "defaultValue": "true",
        "allowedValues": [
            "true",
            "false"
        ]
    },
    "enableLiveTrace": {
        "type": "string",
        "defaultValue": "true",
        "allowedValues": [
            "true",
            "false"
        ]
    },
    "allowedOrigins": {
        "type": "array",
        "defaultValue": [
            "https://foo.com",
            "https://bar.com"
        ],
        "metadata": {
            "description": "Set the list of origins that should be allowed to make cross-origin calls."
        }
    },
    "resources": [
    {
        "type": "Microsoft.SignalRService/SignalR",
```

```
"apiVersion": "2020-07-01-preview",
"name": "[parameters('name')]",
"location": "[parameters('location')]",
"sku": {
    "capacity": "[parameters('capacity')]",
    "name": "[parameters('pricingTier')]"
},
"kind": "SignalR",
"identity": {
    "type": "SystemAssigned"
},
"properties": {
    "tls": {
        "clientCertEnabled": false
    },
    "features": [
        {
            "flag": "ServiceMode",
            "value": "[parameters('serviceMode')]"
        },
        {
            "flag": "EnableConnectivityLogs",
            "value": "[parameters('enableConnectivityLogs')]"
        },
        {
            "flag": "EnableMessagingLogs",
            "value": "[parameters('enableMessagingLogs')]"
        },
        {
            "flag": "EnableLiveTrace",
            "value": "[parameters('enableLiveTrace')]"
        }
    ],
    "cors": {
        "allowedOrigins": "[parameters('allowedOrigins')]"
    },
    "networkACLs": {
        "defaultAction": "deny",
        "publicNetwork": {
            "allow": [
                "ClientConnection"
            ]
        },
        "privateEndpoints": [
            {
                "name": "mySignalRService.1fa229cd-bf3f-47f0-8c49-afb36723997e",
                "allow": [
                    "ServerConnection"
                ]
            }
        ]
    },
    "upstream": {
        "templates": [
            {
                "categoryPattern": "*",
                "eventPattern": "connect,disconnect",
                "hubPattern": "*",
                "urlTemplate": "https://example.com/chat/api/connect"
            }
        ]
    }
}
]
```

The template defines one Azure resource:

- [Microsoft.SignalRService/SignalR](#)

## Deploy the template

- [Portal](#)
- [PowerShell](#)
- [CLI](#)

Select the following link to deploy the Azure SignalR Service using the ARM template in the Azure portal:



On the Deploy an Azure SignalR Service page:

1. If you want, change the **Subscription** from the default.
2. For **Resource group**, select **Create new**, enter a name for the new resource group, and select **OK**.
3. If you created a new resource group, select a **Region** for the resource group.
4. If you want, enter a new **Name** and the **Location** (such as **eastus2**) of the Azure SignalR Service. If you don't specify a name, it's automatically generated. The location for the Azure SignalR Service can be the same as or different from the region of the resource group. If you don't specify a location, it's set to the same region as the resource group.
5. Choose the **Pricing Tier** (**Free\_F1** or **Standard\_S1**), enter the **Capacity** (number of SignalR units), and choose a **Service Mode** of **Default** (requires hub server), **Serverless** (doesn't allow any server connection), or **Classic** (routed to hub server only if hub has server connection). Then choose whether to **Enable Connectivity Logs** or **Enable Messaging Logs**.

### NOTE

For the **Free\_F1** pricing tier, the capacity is limited to 1 unit.

The screenshot shows the Azure portal interface for deploying an Azure SignalR service. At the top, it says 'Deploy an Azure SignalR service' and 'Azure quickstart template'. Below that, there are tabs for 'Basics' (which is selected) and 'Review + create'. Under 'Template', there's a section for '101-signalr' which contains '1 resource'. There are 'Edit template' and 'Edit parameters' buttons. The 'Deployment scope' section asks to select a subscription and resource group. The 'Subscription' dropdown is set to 'Contoso'. The 'Resource group' dropdown is set to '(New) contoso-test-group' and is highlighted with a red box. The 'Parameters' section includes fields for Region (East US), Name (contoso-signalr-service), Location (westus2), Pricing Tier (Free\_F1), Capacity (1), Service Mode (Default), Enable Connectivity Logs (true), Enable Messaging Logs (true), and Allowed Origins (["https://foo.com", "https://bar.com"]). The 'Allowed Origins' field is also highlighted with a red box. At the bottom, there are buttons for 'Review + create' (which is blue and highlighted with a red box), '< Previous', and 'Next : Review + create >'.

6. Select **Review + create**.

7. Read the terms and conditions, and then select **Create**.

#### NOTE

The deployment may take a few minutes to complete. Note the names for the Azure SignalR Service and the resource group, which you use to review the deployed resources later.

## Review deployed resources

- [Portal](#)
- [PowerShell](#)
- [CLI](#)

Follow these steps to see an overview of your new Azure SignalR Service:

1. In the [Azure portal](#), search for and select **SignalR**.
2. In the SignalR list, select your new service. The **Overview** page for the new Azure SignalR Service

appears.

## Clean up resources

When it's no longer needed, delete the resource group, which deletes the resources in the resource group.

- [Portal](#)
- [PowerShell](#)
- [CLI](#)

1. In the [Azure portal](#), search for and select **Resource groups**.
2. In the resource group list, choose the name of your resource group.
3. In the **Overview** page of your resource group, select **Delete resource group**.
4. In the confirmation dialog box, type the name of your resource group, and then select **Delete**.

## Next steps

For a step-by-step tutorial that guides you through the process of creating an ARM template, see:

[Tutorial: Create and deploy your first ARM template](#)

# Quickstart: Broadcast real-time messages from console app

4/22/2021 • 6 minutes to read • [Edit Online](#)

Azure SignalR Service provides [REST API](#) to support server to client communication scenarios, such as broadcasting. You can choose any programming language that can make REST API call. You can post messages to all connected clients, a specific client by name, or a group of clients.

In this quickstart, you will learn how to send messages from a command-line app to connected client apps in C#.

## Prerequisites

This quickstart can be run on macOS, Windows, or Linux.

- [.NET Core SDK](#)
- A text editor or code editor of your choice.

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Sign in to Azure

Sign in to the Azure portal at <https://portal.azure.com/> with your Azure account.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Create an Azure SignalR Service instance

Your application will connect to a SignalR Service instance in Azure.

1. Select the New button found on the upper left-hand corner of the Azure portal. In the New screen, type *SignalR Service* in the search box and press enter.

The screenshot shows the Azure Marketplace search interface. On the left, there's a sidebar with categories like Compute, Networking, Storage, Web, Mobile, Containers, Databases, and Analytics. The main area has a search bar with 'signalr service' typed in. Below the search bar is a 'Filter' dropdown. The results table has columns for NAME, PUBLISHER, and CATEGORY. There are five items listed:

NAME	PUBLISHER	CATEGORY
SignalR Service	Microsoft	Web
Signal Sciences -BYOL	Signal Sciences	Compute
MX Public Store Edition Server (with SQL)	NDL Software	Compute
SmartMessage-Connect	ODC Business Solutions	Compute
MX Public Store Edition Server (without SQL)	NDL Software	Compute

2. Select **SignalR Service** from the search results, then select **Create**.

3. Enter the following settings.

SETTING	SUGGESTED VALUE	DESCRIPTION
Resource name	Globally unique name	Name that identifies your new SignalR Service instance. Valid characters are <code>a-z</code> , <code>0-9</code> , and <code>-</code> .
Subscription	Your subscription	The subscription under which this new SignalR Service instance is created.
Resource Group	myResourceGroup	Name for the new resource group in which to create your SignalR Service instance.
Location	West US	Choose a <a href="#">region</a> near you.
Pricing tier	Free	Try Azure SignalR Service for free.
Unit count	Not applicable	Unit count specifies how many connections your SignalR Service instance can accept. It is only configurable in the Standard tier.
Service mode	Serverless	For use with Azure Functions or REST API.

Microsoft Azure Search resources, services, and docs Dashboard > New > Marketplace > SignalR Service > SignalR

**Basics** Tags Review + create

Deploy fully managed SignalR Service at scale. [Learn more](#)

**Project Details**

- \* Subscription: Ignite Demo
- \* Resource group: (New) serverlesschat [Create new](#)

**Service Details**

- Resource Name: serverlesschat [.service.signalr.net](#)
- \* Region: South Central US
- \* Pricing tier ([View full pricing details](#)): Free
- \* Unit count: 1
- ServiceMode: Serverless

[Review + create](#) [Next : Tags >](#) [Download a template for automation](#)

- Select **Create** to start deploying the SignalR Service instance.
- After the instance is deployed, open it in the portal and locate its Settings page. Change the Service Mode setting to *Serverless* only if you are using Azure SignalR Service through Azure Functions binding or REST API. Leave it in *Classic* or *Default* otherwise.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Clone the sample application

While the service is deploying, let's switch to prepare the code. Clone the [sample app from GitHub](#), set the SignalR Service connection string, and run the application locally.

1. Open a git terminal window. Change to a folder where you want to clone the sample project.
2. Run the following command to clone the sample repository. This command creates a copy of the sample app on your computer.

```
git clone https://github.com/aspnet/AzureSignalR-samples.git
```

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Build and run the sample

This sample is a console app showing the use of Azure SignalR Service. It provides two modes:

- Server Mode: use simple commands to call Azure SignalR Service REST API.
- Client Mode: connect to Azure SignalR Service and receive messages from server.

Also you can find how to generate an access token to authenticate with Azure SignalR Service.

### Build the executable file

We use macOS osx.10.13-x64 as example. You can find [reference](#) on how to build on other platforms.

```
cd AzureSignalR-samples/samples/Serverless/  
dotnet publish -c Release -r osx.10.13-x64
```

### Start a client

```
cd bin/Release/netcoreapp2.1/osx.10.13-x64/  
Serverless client <ClientName> -c "<ConnectionString>" -h <HubName>
```

### Start a server

```
cd bin/Release/netcoreapp2.1/osx.10.13-x64/  
Serverless server -c "<ConnectionString>" -h <HubName>
```

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Run the sample without publishing

You can also run the command below to start a server or client

```
# Start a server  
dotnet run -- server -c "<ConnectionString>" -h <HubName>  
  
# Start a client  
dotnet run -- client <ClientName> -c "<ConnectionString>" -h <HubName>
```

### Use user-secrets to specify Connection String

You can run `dotnet user-secrets set Azure:SignalR:ConnectionString "<ConnectionString>"` in the root directory of the sample. After that, you don't need the option `-c "<ConnectionString>"` anymore.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Usage

After the server started, use the command to send message:

```
send user <User Id>
send users <User List>
send group <Group Name>
send groups <Group List>
broadcast
```

You can start multiple clients with different client names.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Integration with third-party services

The Azure SignalR service allows third-party services to integrate with the system.

### Definition of technical specifications

The following table shows all the versions of the REST APIs supported to date. You can also find the definition file for each specific version

VERSION	API STATE	DOOR	SPECIFIC
1.0-preview	Available	5002	<a href="#">Swagger</a>
1.0	Available	Standard	<a href="#">Swagger</a>

The list of available APIs for each specific version is available in the following list.

API	1.0-PREVIEW	1.0
Broadcast to all	✓	✓
Broadcast to a group	✓	✓
Broadcast to some groups	✓ (Deprecated)	N / A
Send to a user	✓	✓
Send to some users	✓ (Deprecated)	N / A
Adding a user to a group	N / A	✓
Removing a user from a group	N / A	✓
Check user existence	N / A	✓
Remove a user from all groups	N / A	✓

API	1.0-PREVIEW	1.0
Send to a connection	N / A	✓
Add a connection to a group	N / A	✓
Remove a connection from a group	N / A	✓
Close a client connection	N / A	✓
Service Health	N / A	✓

## Broadcast to everyone

VERSION	API HTTP METHOD	REQUEST URL	REQUEST BODY
1.0-preview	POST	<a href="https://&lt;instance-name&gt;.service.signalr.net:50">https://&lt;instance-name&gt;.service.signalr.net:50</a>	{"target": "<method-name>", "arguments": "preview/hub/<hub-name> [...]}}
1.0	POST	<a href="https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;">https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;</a>	Same as above

## Broadcast to a group

VERSION	API HTTP METHOD	REQUEST URL	REQUEST BODY
1.0-preview	POST	<a href="https://&lt;instance-name&gt;.service.signalr.net:50">https://&lt;instance-name&gt;.service.signalr.net:50</a>	{"target": "<method-name>", "arguments": "preview/hub/<hub-name>/group [...]}name"}
1.0	POST	<a href="https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/groups/&lt;group-name&gt;">https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/groups/&lt;group-name&gt;</a>	Same as above

## Sending to a user

VERSION	API HTTP METHOD	REQUEST URL	REQUEST BODY
1.0-preview	POST	<a href="https://&lt;instance-name&gt;.service.signalr.net:50">https://&lt;instance-name&gt;.service.signalr.net:50</a>	{"target": "<method-name>", "arguments": "preview/hub/<hub-name>/user/ [...]}id"}
1.0	POST	<a href="https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/users/&lt;user-id&gt;">https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/users/&lt;user-id&gt;</a>	Same as above

## Adding a user to a group

VERSION	API HTTP METHOD	REQUEST URL
1.0	PUT	<a href="https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/groups/&lt;group-name&gt;/users/&lt;user-id&gt;">https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/groups/&lt;group-name&gt;/users/&lt;user-id&gt;</a>

## Removing a user from a group

VERSION	API HTTP METHOD	REQUEST URL
1.0	DELETE	<code>https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/groups/&lt;group-name&gt;/users/&lt;user-id&gt;</code>

## Check user existence in a group

API VERSION	API HTTP METHOD	REQUEST URL
1.0	GET	<code>https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/users/&lt;user-id&gt;/groups/&lt;group-name&gt;</code>
1.0	GET	<code>https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/groups/&lt;group-name&gt;/users/&lt;user-id&gt;</code>
RESPONSE STATUS CODE		DESCRIPTION
200		User exists
404		User not exists

## Remove a user from all groups

API VERSION	API HTTP METHOD	REQUEST URL
1.0	DELETE	<code>https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/users/&lt;user-id&gt;/groups</code>

## Send message to a connection

API VERSION	API HTTP METHOD	REQUEST URL	REQUEST BODY
1.0	POST	<code>https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/connections/&lt;connection-id&gt;</code>	<code>{ "target": "&lt;method-name&gt;", "arguments": [ "name" ] }</code>

## Add a connection to a group

API VERSION	API HTTP METHOD	REQUEST URL
1.0	PUT	<code>https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/groups/&lt;group-name&gt;/connections/&lt;connection-id&gt;</code>
1.0	PUT	<code>https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/connections/&lt;connection-id&gt;/groups/&lt;group-name&gt;</code>

## Remove a connection from a group

API VERSION	API HTTP METHOD	REQUEST URL
1.0	DELETE	<code>https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/groups/&lt;group-name&gt;/connections/&lt;connection-id&gt;</code>
1.0	DELETE	<code>https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/connections/&lt;connection-id&gt;/groups/&lt;group-name&gt;</code>

## Close a client connection

API VERSION	API HTTP METHOD	REQUEST URL
1.0	DELETE	<code>https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/connections/&lt;connection-id&gt;</code>
1.0	DELETE	<code>https://&lt;instance-name&gt;.service.signalr.net/api/v1/hubs/&lt;hub-name&gt;/connections/&lt;connection-id&gt;?reason=&lt;close-reason&gt;</code>

## Service Health

API VERSION	API HTTP METHOD	REQUEST URL
1.0	GET	<code>https://&lt;instance-name&gt;.service.signalr.net/api/v1/health</code>
RESPONSE STATUS CODE	DESCRIPTION	
200	Service Good	
5xx	Service Error	

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Clean up resources

If you're not going to continue to use this app, delete all resources created by this quickstart with the following steps so you don't incur any charges:

1. In the Azure portal, select **Resource groups** on the far left, and then select the resource group you created. Alternatively, you may use the search box to find the resource group by its name.
2. In the window that opens, select the resource group, and then click **Delete resource group**.
3. In the new window, type the name of the resource group to delete, and then click **Delete**.

Having issues? Try the [troubleshooting guide](#) or [let us know](#).

## Next steps

In this quickstart, you learned how to use REST API to broadcast real-time message from SignalR Service to clients. Next, learn more about how to develop and deploy Azure Functions with SignalR Service binding, which

is built on top of REST API.

[Develop Azure Functions using Azure SignalR Service bindings](#)

# Tutorial: Azure SignalR Service authentication with Azure Functions

11/2/2020 • 12 minutes to read • [Edit Online](#)

A step by step tutorial to build a chat room with authentication and private messaging using Azure Functions, App Service Authentication, and SignalR Service.

## Introduction

### Technologies used

- [Azure Functions](#) - Backend API for authenticating users and sending chat messages
- [Azure SignalR Service](#) - Broadcast new messages to connected chat clients
- [Azure Storage](#) - Host the static website for the chat client UI

### Prerequisites

The following software is required to build this tutorial.

- [Git](#)
- [Node.js](#) (Version 10.x)
- [.NET SDK](#) (Version 2.x, required for Functions extensions)
- [Azure Functions Core Tools](#) (Version 2)
- [Visual Studio Code](#) (VS Code) with the following extensions
  - [Azure Functions](#) - work with Azure Functions in VS Code
  - [Live Server](#) - serve web pages locally for testing

[Having issues? Let us know.](#)

## Sign into the Azure portal

Go to the [Azure portal](#) and sign in with your credentials.

[Having issues? Let us know.](#)

## Create an Azure SignalR Service instance

You will build and test the Azure Functions app locally. The app will access a SignalR Service instance in Azure that needs to be created ahead of time.

1. Click on the **Create a resource** (+) button for creating a new Azure resource.
2. Search for **SignalR Service** and select it. Click **Create**.

NAME	PUBLISHER	CATEGORY
SignalR Service	Microsoft	Web
Signal Sciences -BYOL	Signal Sciences	Compute
MX Public Store Edition Server (with SQL)	NDL Software	Compute
SmartMessage-Connect	ODC Business Solutions	Compute
MX Public Store Edition Server (without SQL)	NDL Software	Compute

3. Enter the following information.

NAME	VALUE
Resource name	A unique name for the SignalR Service instance
Resource group	Create a new resource group with a unique name
Location	Select a location close to you
Pricing Tier	Free

4. Click **Create**.

5. After the instance is deployed, open it in the portal and locate its Settings page. Change the Service Mode setting to *Serverless*.

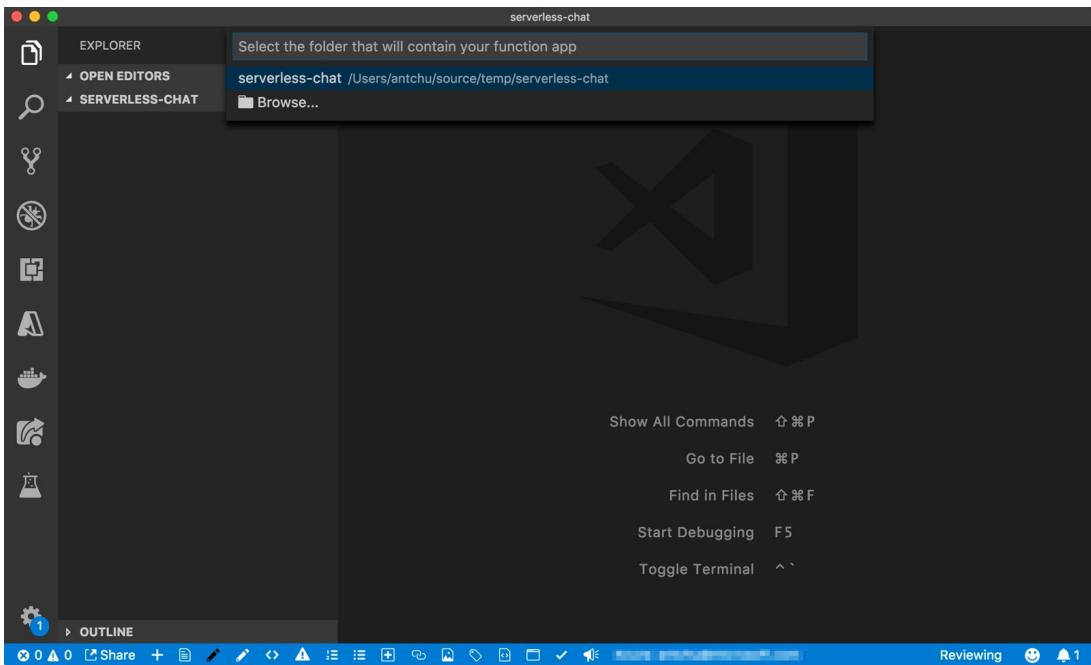


[Having issues? Let us know.](#)

## Initialize the function app

### Create a new Azure Functions project

- In a new VS Code window, use `File > Open Folder` in the menu to create and open an empty folder in an appropriate location. This will be the main project folder for the application that you will build.
- Using the Azure Functions extension in VS Code, initialize a Function app in the main project folder.
  - Open the Command Palette in VS Code by selecting `View > Command Palette` from the menu (shortcut `Ctrl-Shift-P`, macOS: `Cmd-Shift-P`).
  - Search for the **Azure Functions: Create New Project** command and select it.
  - The main project folder should appear. Select it (or use "Browse" to locate it).
  - In the prompt to choose a language, select **JavaScript**.



## Install function app extensions

This tutorial uses Azure Functions bindings to interact with Azure SignalR Service. Like most other bindings, the SignalR Service bindings are available as an extension that needs to be installed using the Azure Functions Core Tools CLI before they can be used.

1. Open a terminal in VS Code by selecting **View > Terminal** from the menu (Ctrl-`).
2. Ensure the main project folder is the current directory.
3. Install the SignalR Service function app extension.

```
func extensions install -p Microsoft.Azure.WebJobs.Extensions.SignalRService -v 1.0.0
```

## Configure application settings

When running and debugging the Azure Functions runtime locally, application settings are read from **local.settings.json**. Update this file with the connection string of the SignalR Service instance that you created earlier.

1. In VS Code, select **local.settings.json** in the Explorer pane to open it.
2. Replace the file's contents with the following.

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureSignalRConnectionString": "<signalr-connection-string>",
    "WEBSITE_NODE_DEFAULT_VERSION": "10.14.1",
    "FUNCTIONS_WORKER_RUNTIME": "node"
  },
  "Host": {
    "LocalHttpPort": 7071,
    "CORS": "http://127.0.0.1:5500",
    "CORS Credentials": true
  }
}
```

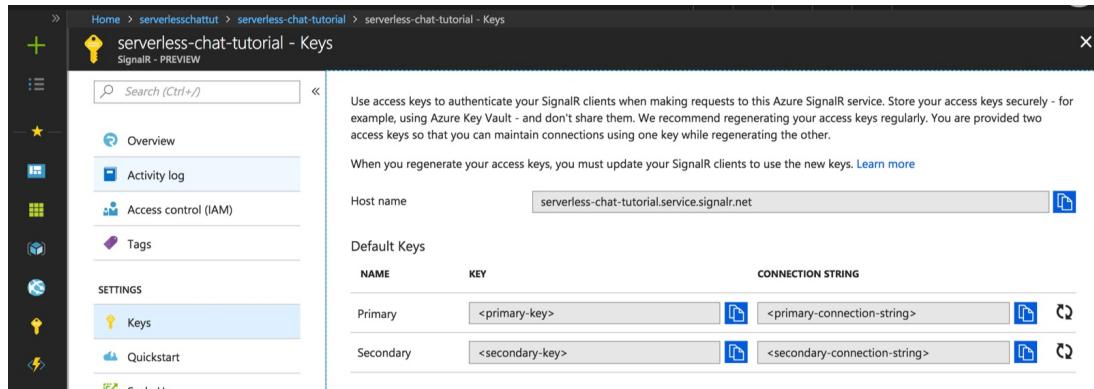
- Enter the Azure SignalR Service connection string into a setting named `AzureSignalRConnectionString`. Obtain the value from the **Keys** page in the Azure SignalR Service

resource in the Azure portal; either the primary or secondary connection string can be used.

- The `WEBSITE_NODE_DEFAULT_VERSION` setting is not used locally, but is required when deployed to Azure.
- The `Host` section configures the port and CORS settings for the local Functions host (this setting has no effect when running in Azure).

**NOTE**

Live Server is typically configured to serve content from `http://127.0.0.1:5500`. If you find that it is using a different URL or you are using a different HTTP server, change the `cors` setting to reflect the correct origin.



3. Save the file.

[Having issues? Let us know.](#)

## Create a function to authenticate users to SignalR Service

When the chat app first opens in the browser, it requires valid connection credentials to connect to Azure SignalR Service. You'll create an HTTP triggered function named `negotiate` in your function app to return this connection information.

**NOTE**

This function must be named `negotiate` as the SignalR client requires an endpoint that ends in `/negotiate`.

1. Open the VS Code command palette (`Ctrl+Shift+P`, macOS: `Cmd+Shift+P`).
2. Search for and select the **Azure Functions: Create Function** command.
3. When prompted, provide the following information.

NAME	VALUE
Function app folder	Select the main project folder
Template	HTTP Trigger
Name	negotiate
Authorization level	Anonymous

A folder named **negotiate** is created that contains the new function.

4. Open **negotiate/function.json** to configure bindings for the function. Modify the content of the file to the following. This adds an input binding that generates valid credentials for a client to connect to an Azure SignalR Service hub named `chat`.

```
{  
    "disabled": false,  
    "bindings": [  
        {  
            "authLevel": "anonymous",  
            "type": "httpTrigger",  
            "direction": "in",  
            "name": "req"  
        },  
        {  
            "type": "http",  
            "direction": "out",  
            "name": "res"  
        },  
        {  
            "type": "signalRConnectionInfo",  
            "name": "connectionInfo",  
            "userId": "",  
            "hubName": "chat",  
            "direction": "in"  
        }  
    ]  
}
```

The `userId` property in the `signalRConnectionInfo` binding is used to create an authenticated SignalR Service connection. Leave the property blank for local development. You will use it when the function app is deployed to Azure.

5. Open **negotiate/index.js** to view the body of the function. Modify the content of the file to the following.

```
module.exports = async function (context, req, connectionInfo) {  
    context.res.body = connectionInfo;  
};
```

This function takes the SignalR connection information from the input binding and returns it to the client in the HTTP response body. The SignalR client will use this information to connect to the SignalR Service instance.

[Having issues? Let us know.](#)

## Create a function to send chat messages

The web app also requires an HTTP API to send chat messages. You will create an HTTP triggered function named *SendMessage* that sends messages to all connected clients using SignalR Service.

1. Open the VS Code command palette (`Ctrl-Shift-P`, macOS: `Cmd-Shift-P`).
2. Search for and select the **Azure Functions: Create Function** command.
3. When prompted, provide the following information.

NAME	VALUE
Function app folder	select the main project folder
Template	HTTP Trigger
Name	SendMessage
Authorization level	Anonymous

A folder named **SendMessage** is created that contains the new function.

4. Open **SendMessage/function.json** to configure bindings for the function. Modify the content of the file to the following.

```
{
  "disabled": false,
  "bindings": [
    {
      "authLevel": "anonymous",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "route": "messages",
      "methods": [
        "post"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "res"
    },
    {
      "type": "signalR",
      "name": "$return",
      "hubName": "chat",
      "direction": "out"
    }
  ]
}
```

This makes two changes to the original function:

- Changes the route to `messages` and restricts the HTTP trigger to the **POST** HTTP method.
- Adds a SignalR Service output binding that sends a message returned by the function to all clients connected to a SignalR Service hub named `chat`.

5. Save the file.
6. Open **SendMessage/index.js** to view the body of the function. Modify the content of the file to the following.

```

module.exports = async function (context, req) {
    const message = req.body;
    message.sender = req.headers && req.headers['x-ms-client-principal-name'] || '';

    let recipientUserId = '';
    if (message.recipient) {
        recipientUserId = message.recipient;
        message.isPrivate = true;
    }

    return {
        'userId': recipientUserId,
        'target': 'newMessage',
        'arguments': [ message ]
    };
};

```

This function takes the body from the HTTP request and sends it to clients connected to SignalR Service, invoking a function named `newMessage` on each client.

The function can read the sender's identity and can accept a *recipient* value in the message body to allow for a message to be sent privately to a single user. These functionalities will be used later in the tutorial.

7. Save the file.

[Having issues? Let us know.](#)

## Create and run the chat client web user interface

The chat application's UI is a simple single page application (SPA) created with the Vue JavaScript framework. It will be hosted separately from the function app. Locally, you will run the web interface using the Live Server VS Code extension.

1. In VS Code, create a new folder named **content** at the root of the main project folder.
2. In the **content** folder, create a new file named **index.html**.
3. Copy and paste the content of [index.html](#).
4. Save the file.
5. Press **F5** to run the function app locally and attach a debugger.
6. With **index.html** open, start Live Server by opening the VS Code command palette (`Ctrl+Shift+P`, macOS: `Cmd+Shift+P`) and selecting **Live Server: Open with Live Server**. Live Server will open the application in a browser.
7. The application opens. Enter a message in the chat box and press enter. Refresh the application to see new messages. Because no authentication was configured, all messages will be sent as "anonymous".

[Having issues? Let us know.](#)

## Deploy to Azure and enable authentication

You have been running the function app and chat application locally. You will now deploy them to Azure and enable authentication and private messaging in the application.

### Log into Azure with VS Code

1. Open the VS Code command palette (`Ctrl+Shift+P`, macOS: `Cmd+Shift+P`).

2. Search for and select the **Azure: Sign in** command.
3. Follow the instructions to complete the sign in process in your browser.

### Create a Storage account

An Azure Storage account is required by a function app running in Azure. You will also host the web page for the chat UI using the static websites feature of Azure Storage.

1. In the Azure portal, click on the **Create a resource (+)** button for creating a new Azure resource.
2. Select the **Storage** category, then select **Storage account**.
3. Enter the following information.

NAME	VALUE
Subscription	Select the subscription containing the SignalR Service instance
Resource group	Select the same resource group
Resource name	A unique name for the Storage account
Location	Select the same location as your other resources
Performance	Standard
Account kind	StorageV2 (general purpose V2)
Replication	Locally-redundant storage (LRS)
Access Tier	Hot

4. Click **Review + create**, then **Create**.

### Configure static websites

1. After the Storage account is created, open it in the Azure portal.
2. Select **Static website**.
3. Select **Enabled** to enable the static website feature.
4. In **Index document name**, enter *index.html*.
5. Click **Save**.
6. A **Primary endpoint** appears. Note this value. It will be required to configure the function app.

### Configure function app for authentication

So far, the chat app works anonymously. In Azure, you will use [App Service Authentication](#) to authenticate the user. The user ID or username of the authenticated user can be passed to the *SignalRConnectionInfo* binding to generate connection information that is authenticated as the user.

When a sending message, the app can decide whether to send it to all connected clients, or only to the clients that have been authenticated to a given user.

1. In VS Code, open **negotiate/function.json**.
2. Insert a [binding expression](#) into the *userId* property of the *SignalRConnectionInfo* binding:

{headers.x-ms-client-principal-name} . This sets the value to the username of the authenticated user. The attribute should now look like this.

```
{  
    "type": "signalRConnectionInfo",  
    "name": "connectionInfo",  
    "userId": "{headers.x-ms-client-principal-name}",  
    "hubName": "chat",  
    "direction": "in"  
}
```

3. Save the file.

### Deploy function app to Azure

1. Open the VS Code command palette ( `Ctrl+Shift+P` , macOS: `Cmd+Shift+P` ) and select **Azure Functions: Deploy to Function App**.
2. When prompted, provide the following information.

NAME	VALUE
Folder to deploy	Select the main project folder
Subscription	Select your subscription
Function app	Select <b>Create New Function App</b>
Function app name	Enter a unique name
Resource group	Select the same resource group as the SignalR Service instance
Storage account	Select the storage account you created earlier

A new function app is created in Azure and the deployment begins. Wait for the deployment to complete.

### Upload function app local settings

1. Open the VS Code command palette ( `Ctrl+Shift+P` , macOS: `Cmd+Shift+P` ).
2. Search for and select the **Azure Functions: Upload local settings** command.
3. When prompted, provide the following information.

NAME	VALUE
Local settings file	local.settings.json
Subscription	Select your subscription
Function app	Select the previously deployed function app

Local settings are uploaded to the function app in Azure. If prompted to overwrite existing settings, select **Yes to all**.

### Enable App Service Authentication

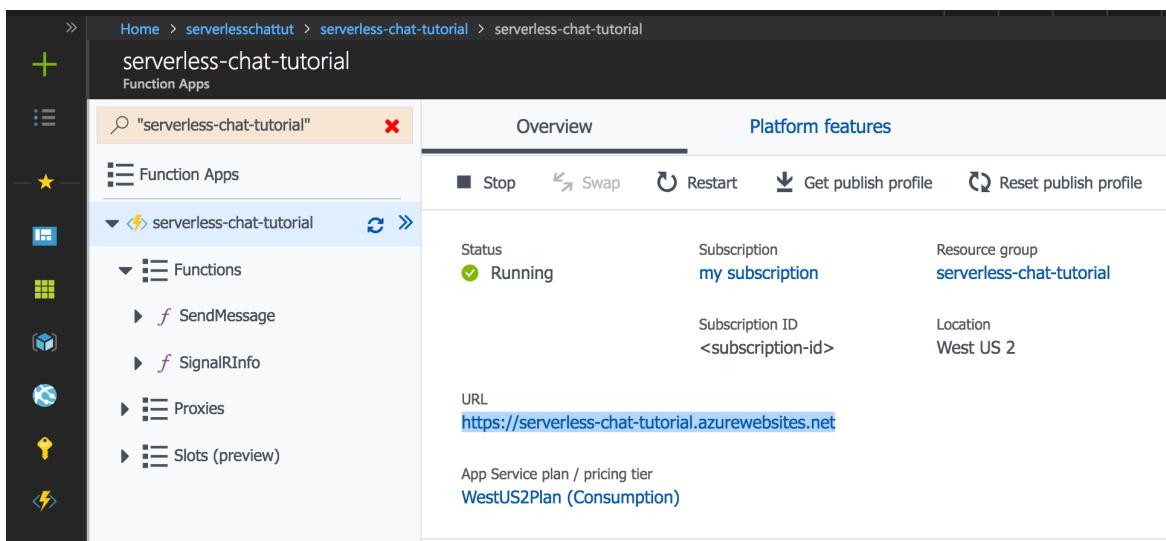
App Service Authentication supports authentication with Azure Active Directory, Facebook, Twitter, Microsoft

account, and Google.

1. Open the VS Code command palette (`Ctrl+Shift+P`, macOS: `Cmd+Shift+P`).
2. Search for and select the **Azure Functions: Open in portal** command.
3. Select the subscription and function app name to open the function app in the Azure portal.
4. In the function app that was opened in the portal, locate the **Platform features** tab, select **Authentication/Authorization**.
5. Turn On App Service Authentication.
6. In **Action to take when request is not authenticated**, select "Log in with {authentication provider you selected earlier}".
7. In **Allowed External Redirect URLs**, enter the URL of your storage account primary web endpoint that you previously noted.
8. Follow the documentation for the login provider of your choice to complete the configuration.
  - [Azure Active Directory](#)
  - [Facebook](#)
  - [Twitter](#)
  - [Microsoft account](#)
  - [Google](#)

## Update the web app

1. In the Azure portal, navigate to the function app's overview page.
2. Copy the function app's URL.



3. In VS Code, open `index.html` and replace the value of `apiBaseUrl` with the function app's URL.
4. The application can be configured with authentication using Azure Active Directory, Facebook, Twitter, Microsoft account, or Google. Select the authentication provider that you will use by setting the value of `authProvider`.
5. Save the file.

## Deploy the web application to blob storage

The web application will be hosted using Azure Blob Storage's static websites feature.

1. Open the VS Code command palette (`Ctrl+Shift+P`, macOS: `Cmd+Shift+P`).

2. Search for and select the **Azure Storage: Deploy to Static Website** command.

3. Enter the following values:

NAME	VALUE
Subscription	Select your subscription
Storage account	Select the storage account you created earlier
Folder to deploy	Select <b>Browse</b> and select the <i>content</i> folder

The files in the *content* folder should now be deployed to the static website.

### Enable function app cross origin resource sharing (CORS)

Although there is a CORS setting in `local.settings.json`, it is not propagated to the function app in Azure. You need to set it separately.

1. Open the function app in the Azure portal.

2. Under the **Platform features** tab, select CORS.

The screenshot shows the Azure portal interface. On the left, there's a sidebar with icons for Home, Function Apps, Functions, and more. The main area shows a search bar with 'serverless-chat-tutorial' and a dropdown menu with 'antchu'. Below that, under 'Function Apps', there's a list item 'serverless-chat-tutorial'. The 'Platform features' tab is selected. In the center, there are several sections: 'GENERAL SETTINGS' (Function app settings, Application settings, Properties, Backups, All settings), 'NETWORKING' (Networking, SSL, Custom domains, Authentication / Authorization, Managed service identity, Push notifications), 'API' (API definition, CORS - which is highlighted with a blue background), 'APP SERVICE PLAN' (App Service plan, Scale up, Quotas), and 'RESOURCE MANAGEMENT' (Diagnose and solve problems, Activity log). A search bar labeled 'Search features' is also present.

3. In the *Allowed origins* section, add an entry with the static website *primary endpoint* as the value (remove the trailing `/`).

4. In order for the SignalR JavaScript SDK call your function app from a browser, support for credentials in CORS must be enabled. Select the "Enable Access-Control-Allow-Credentials" checkbox.

## Request Credentials

Enable Access-Control-Allow-Credentials i

5. Click **Save** to persist the CORS settings.

### Try the application

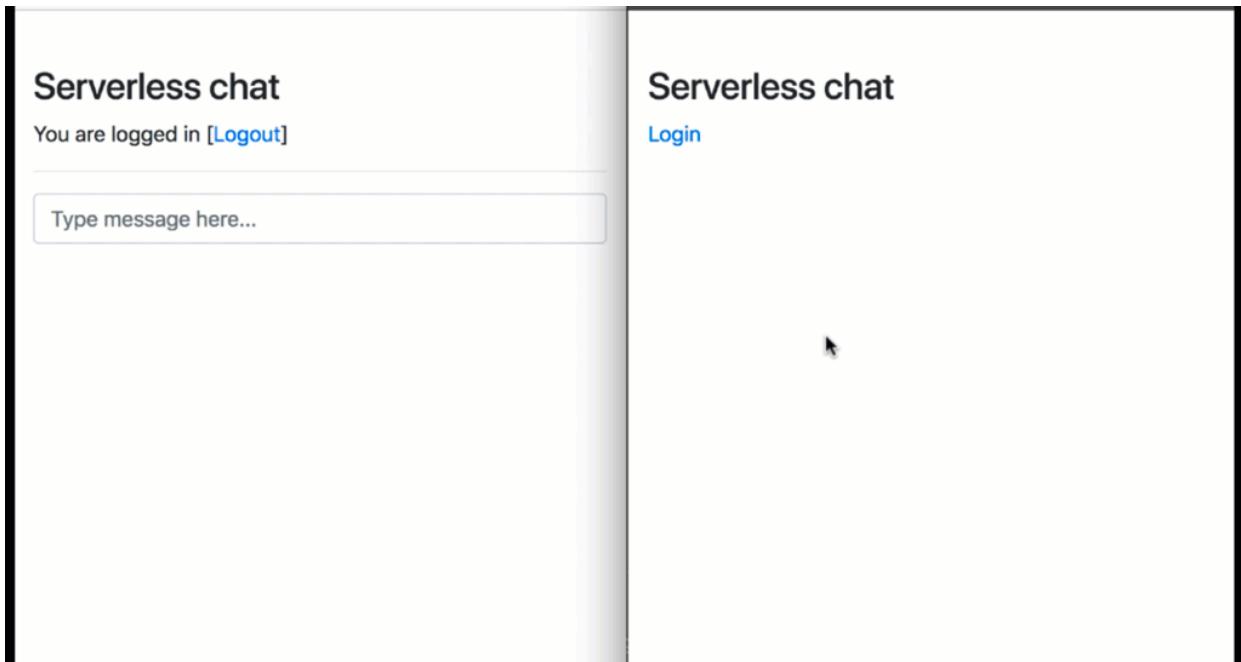
1. In a browser, navigate to the storage account's primary web endpoint.

2. Select **Login** to authenticate with your chosen authentication provider.

3. Send public messages by entering them into the main chat box.

4. Send private messages by clicking on a username in the chat history. Only the selected recipient will receive these messages.

Congratulations! You have deployed a real-time, serverless chat app!



[Having issues? Let us know.](#)

## Clean up resources

To clean up the resources created in this tutorial, delete the resource group using the Azure portal.

[Having issues? Let us know.](#)

## Next steps

In this tutorial, you learned how to use Azure Functions with Azure SignalR Service. Read more about building real-time serverless applications with SignalR Service bindings for Azure Functions.

[Build Real-time Apps with Azure Functions](#)

[Having issues? Let us know.](#)

# Tutorial: Build a Blazor Server chat app

5/2/2021 • 7 minutes to read • [Edit Online](#)

This tutorial shows you how to build and modify a Blazor Server app. You'll learn how to:

- Build a simple chat room with the Blazor Server app template.
- Work with Razor components.
- Use event handling and data binding in Razor components.
- Quick-deploy to Azure App Service in Visual Studio.
- Migrate from local SignalR to Azure SignalR Service.

## Prerequisites

- Install [.NET Core 3.0 SDK](#) (Version >= 3.0.100)
- Install [Visual Studio 2019](#) (Version >= 16.3)

[Having issues? Let us know.](#)

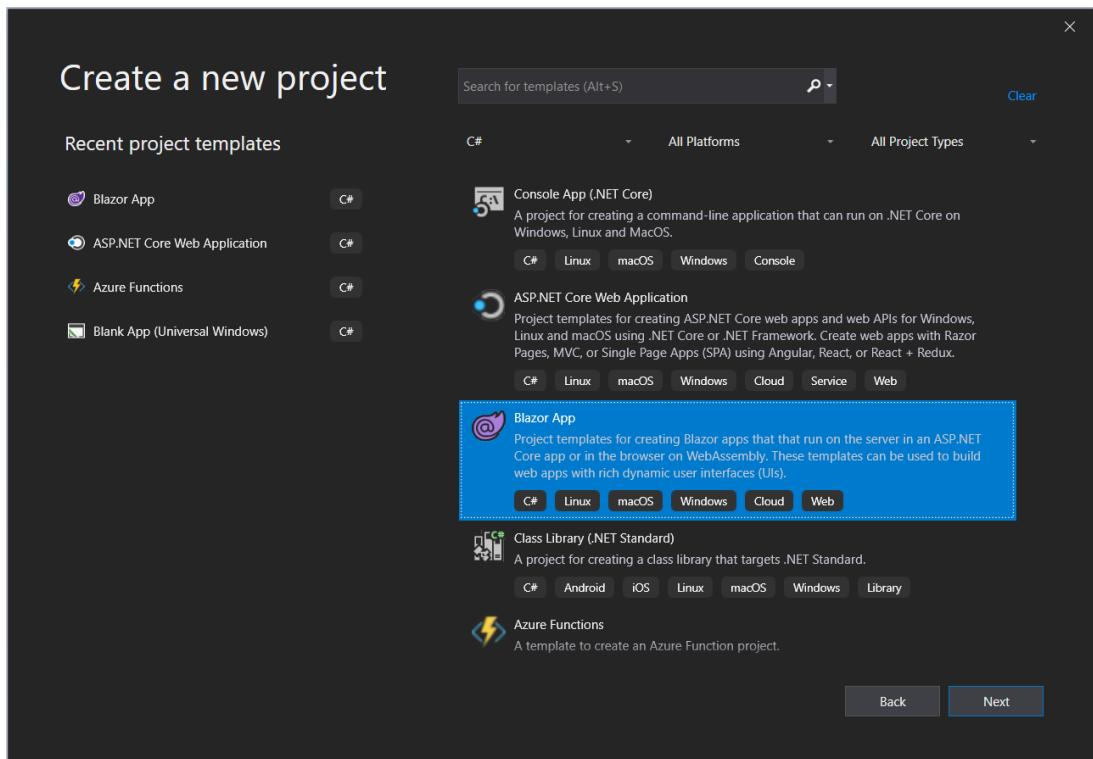
## Build a local chat room in Blazor Server app

Beginning in Visual Studio 2019 version 16.2.0, Azure SignalR Service is built into the web application publish process to make managing the dependencies between the web app and SignalR service much more convenient. You can work in a local SignalR instance in a local development environment and work in Azure SignalR Service for Azure App Service at the same time without any code changes.

1. Create a Blazor chat app:
  - a. In Visual Studio, choose **Create a new project**.
  - b. Select **Blazor App**.
  - c. Name the application and choose a folder.
  - d. Select the **Blazor Server App** template.

### NOTE

Make sure that you've already installed .NET Core SDK 3.0+ to enable Visual Studio to correctly recognize the target framework.



e. You also can create a project by running the `dotnet new` command in the .NET CLI:

```
dotnet new blazorserver -o BlazorChat
```

2. Add a new C# file called `BlazorChatSampleHub.cs` and create a new class `BlazorSampleHub` deriving from the `Hub` class for the chat app. For more information on creating hubs, see [Create and Use Hubs](#).

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.SignalR;

namespace BlazorChat
{
    public class BlazorChatSampleHub : Hub
    {
        public const string HubUrl = "/chat";

        public async Task Broadcast(string username, string message)
        {
            await Clients.All.SendAsync("Broadcast", username, message);
        }

        public override Task OnConnectedAsync()
        {
            Console.WriteLine($"{Context.ConnectionId} connected");
            return base.OnConnectedAsync();
        }

        public override async Task OnDisconnectedAsync(Exception e)
        {
            Console.WriteLine($"Disconnected {e?.Message} {Context.ConnectionId}");
            await base.OnDisconnectedAsync(e);
        }
    }
}
```

3. Add an endpoint for the hub in the `Startup.Configure()` method.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapBlazorHub();
    endpoints.MapFallbackToPage("/_Host");
    endpoints.MapHub<BlazorChatSampleHub>(BlazorChatSampleHub.HubUrl);
});
```

4. Install the `Microsoft.AspNetCore.SignalR.Client` package to use the SignalR client.

```
dotnet add package Microsoft.AspNetCore.SignalR.Client --version 3.1.7
```

5. Create a new **Razor component** called `ChatRoom.razor` under the `Pages` folder to implement the SignalR client. Follow the steps below or use the `ChatRoom.razor` file.

- a. Add the `@page` directive and the using statements. Use the `@inject` directive to inject the `NavigationManager` service.

```
@page "/chatroom"
@inject NavigationManager navigationManager
@using Microsoft.AspNetCore.SignalR.Client;
```

- b. In the `@code` section, add the following members to the new SignalR client to send and receive messages.

```
@code {
    // flag to indicate chat status
    private bool _isChatting = false;

    // name of the user who will be chatting
    private string _username;

    // on-screen message
    private string _message;

    // new message input
    private string _newMessage;

    // list of messages in chat
    private List<Message> _messages = new List<Message>();

    private string _hubUrl;
    private HubConnection _hubConnection;

    public async Task Chat()
    {
        // check username is valid
        if (string.IsNullOrWhiteSpace(_username))
        {
            _message = "Please enter a name";
            return;
        };

        try
        {
            // Start chatting and force refresh UI.
            _isChatting = true;
            await Task.Delay(1);

            // remove old messages if any
            _messages.Clear();
        }
    }
}
```

```

// Create the chat client
string baseUrl = navigationManager.BaseUri;

_hubUrl = baseUrl.TrimEnd('/') + BlazorChatSampleHub.HubUrl;

_hubConnection = new HubConnectionBuilder()
    .WithUrl(_hubUrl)
    .Build();

_hubConnection.On<string, string>("Broadcast", BroadcastMessage);

await _hubConnection.StartAsync();

await SendAsync($"[Notice] {_username} joined chat room.");
}

catch (Exception e)
{
    _message = $"ERROR: Failed to start chat client: {e.Message}";
    _isChatting = false;
}

private void BroadcastMessage(string name, string message)
{
    bool isMine = name.Equals(_username, StringComparison.OrdinalIgnoreCase);

    _messages.Add(new Message(name, message, isMine));

    // Inform blazor the UI needs updating
    StateHasChanged();
}

private async Task DisconnectAsync()
{
    if (_isChatting)
    {
        await SendAsync($"[Notice] {_username} left chat room.");

        await _hubConnection.StopAsync();
        await _hubConnection.DisposeAsync();

        _hubConnection = null;
        _isChatting = false;
    }
}

private async Task SendAsync(string message)
{
    if (_isChatting && !string.IsNullOrWhiteSpace(message))
    {
        await _hubConnection.SendAsync("Broadcast", _username, message);

        _newMessage = string.Empty;
    }
}

private class Message
{
    public Message(string username, string body, bool mine)
    {
        Username = username;
        Body = body;
        Mine = mine;
    }

    public string Username { get; set; }
    public string Body { get; set; }
    public bool Mine { get; set; }
}

```

```

        public bool IsNotice => Body.StartsWith("[Notice]");

        public string CSS => Mine ? "sent" : "received";
    }
}

```

- c. Add the UI markup before the `@code` section to interact with the SignalR client.

```

<h1>Blazor SignalR Chat Sample</h1>
<hr />

@if (!_isChatting)
{
    <p>
        Enter your name to start chatting:
    </p>

    <input type="text" maxlength="32" @bind="@_username" />
    <button type="button" @onclick="@Chat"><span class="oi oi-chat" aria-hidden="true"></span>
    Chat!</button>

    // Error messages
    @if (_message != null)
    {
        <div class="invalid-feedback"> @_message</div>
        <small id="emailHelp" class="form-text text-muted"> @_message</small>
    }
}
else
{
    // banner to show current user
    <div class="alert alert-secondary mt-4" role="alert">
        <span class="oi oi-person mr-2" aria-hidden="true"></span>
        <span>You are connected as <b> @_username </b></span>
        <button class="btn btn-sm btn-warning ml-md-auto"
    @onclick="@DisconnectAsync">Disconnect</button>
    </div>
    // display messages
    <div id="scrollbox">
        @foreach (var item in _messages)
        {
            @if (item.IsNotice)
            {
                <div class="alert alert-info"> @item.Body</div>
            }
            else
            {
                <div class="@item.CSS">
                    <div class="user"> @_item.Username</div>
                    <div class="msg"> @_item.Body</div>
                </div>
            }
        }
        <hr />
        <textarea class="input-lg" placeholder="enter your comment" @bind="@_newMessage">
    </textarea>
        <button class="btn btn-default" @onclick="@(() =>
SendAsync(_newMessage))">Send</button>
    </div>
}

```

6. Update the `NavMenu.razor` component to insert a new `NavLink` component to link to the chat room under `NavMenuCssClass`.

```
<li class="nav-item px-3">
    <NavLink class="nav-link" href="chatroom">
        <span class="oi oi-chat" aria-hidden="true"></span> Chat room
    </NavLink>
</li>
```

7. Add a few CSS classes to the `site.css` file to style the UI elements in the chat page.

```

/* improved for chat text box */
textarea {
    border: 1px dashed #888;
    border-radius: 5px;
    width: 80%;
    overflow: auto;
    background: #f7f7f7
}

/* improved for speech bubbles */
.received, .sent {
    position: relative;
    font-family: arial;
    font-size: 1.1em;
    border-radius: 10px;
    padding: 20px;
    margin-bottom: 20px;
}

.received:after, .sent:after {
    content: '';
    border: 20px solid transparent;
    position: absolute;
    margin-top: -30px;
}

.sent {
    background: #03a9f4;
    color: #fff;
    margin-left: 10%;
    top: 50%;
    text-align: right;
}

.received {
    background: #4CAF50;
    color: #fff;
    margin-left: 10px;
    margin-right: 10%;
}

.sent:after {
    border-left-color: #03a9f4;
    border-right: 0;
    right: -20px;
}

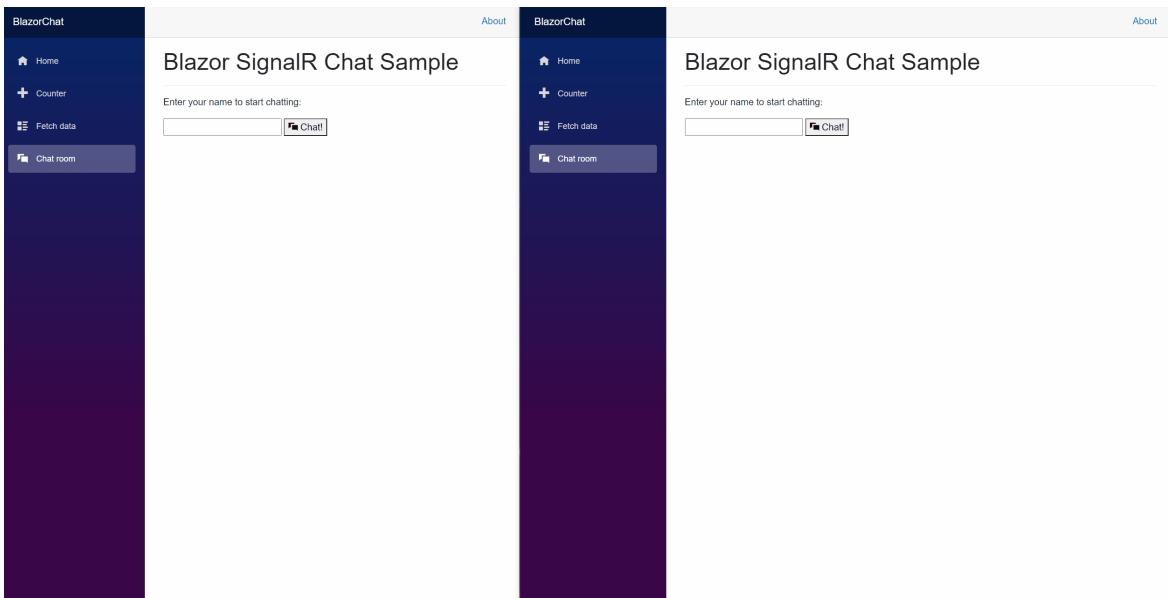
.received:after {
    border-right-color: #4CAF50;
    border-left: 0;
    left: -20px;
}

/* div within bubble for name */
.user {
    font-size: 0.8em;
    font-weight: bold;
    color: #000;
}

.msg {
    /*display: inline;*/
}

```

8. Press F5 to run the app. Now, you can initiate the chat:



[Having issues? Let us know.](#)

## Publish to Azure

When you deploy the Blazor app to Azure App Service, we recommend that you use [Azure SignalR Service](#). Azure SignalR Service allows for scaling up a Blazor Server app to a large number of concurrent SignalR connections. In addition, the SignalR service's global reach and high-performance datacenters significantly aid in reducing latency due to geography.

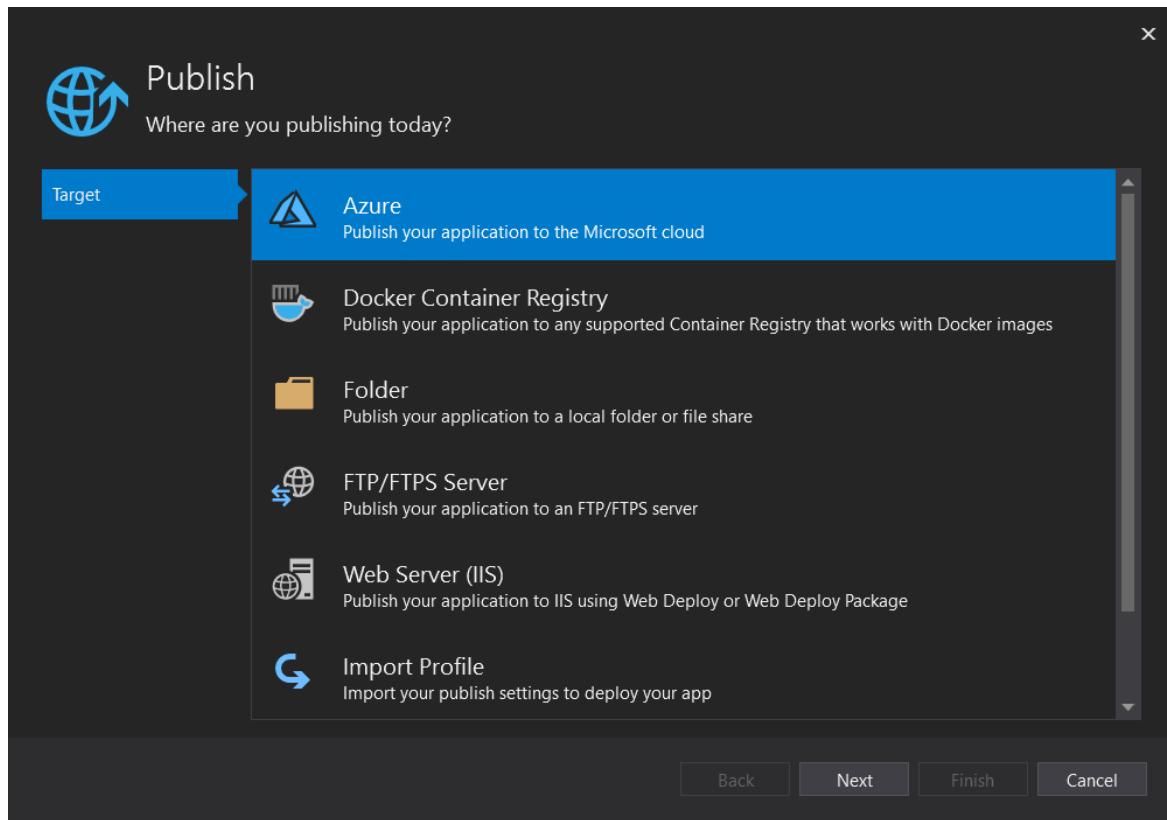
### IMPORTANT

In a Blazor Server app, UI states are maintained on the server side, which means a sticky server session is required to preserve state. If there is a single app server, sticky sessions are ensured by design. However, if there are multiple app servers, there are chances that the client negotiation and connection may go to different servers which may lead to an inconsistent UI state management in a Blazor app. Hence, it is recommended to enable sticky server sessions as shown below in `appsettings.json`:

```
"Azure:SignalR:ServerStickyMode": "Required"
```

1. Right-click the project and go to **Publish**. Use the following settings:

- **Target:** Azure
- **Specific target:** All types of **Azure App Service** are supported.
- **App Service:** Create or select the App Service instance.



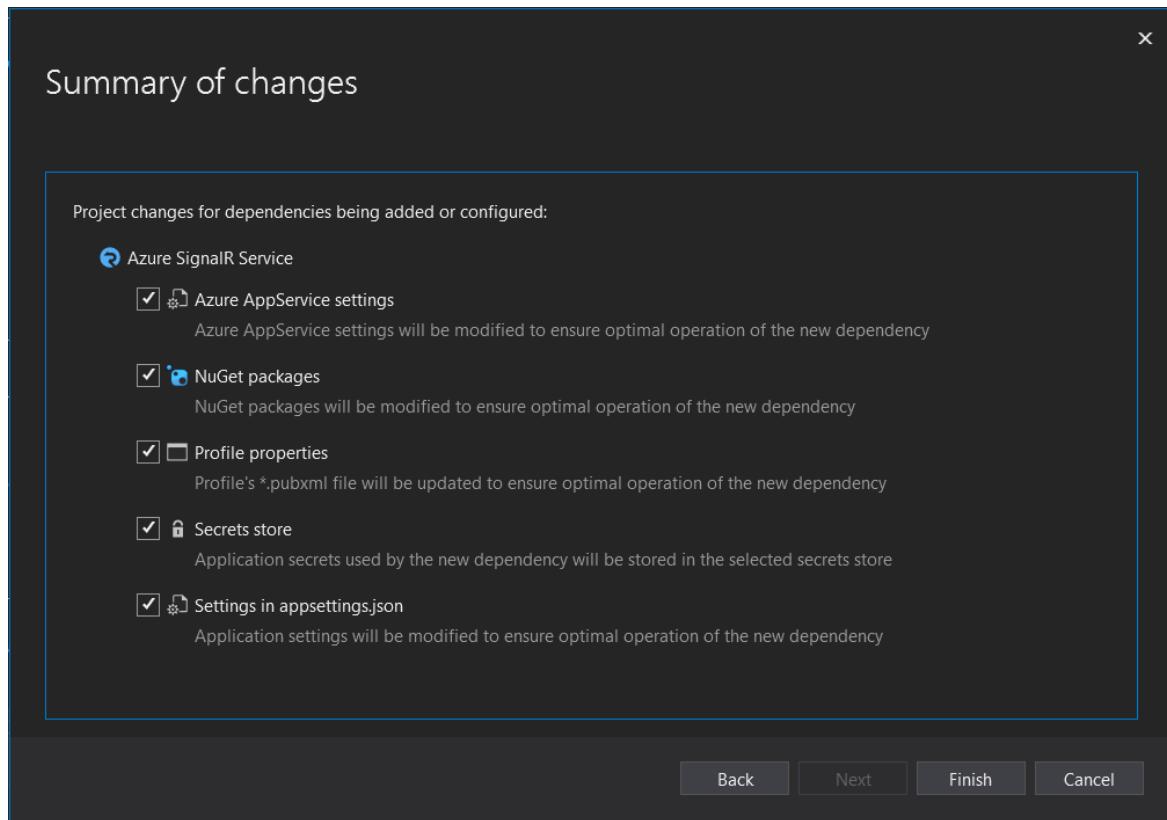
## 2. Add the Azure SignalR Service dependency.

After the creation of the publish profile, you can see a recommendation message to add Azure SignalR service under **Service Dependencies**. Select **Configure** to create a new or select an existing Azure SignalR Service in the pane.

The screenshot shows the Azure portal's Publish blade for a web application. The 'Publish' tab is active. In the 'Service Dependencies' section, there is a recommendation for 'SignalR'. A red box highlights the 'Configure' button next to the recommendation text.

The service dependency will carry out the following activities to enable your app to automatically switch to Azure SignalR Service when on Azure:

- Update `HostingStartupAssembly` to use Azure SignalR Service.
- Add the Azure SignalR Service NuGet package reference.
- Update the profile properties to save the dependency settings.
- Configure the secrets store as per your choice.
- Add the configuration in `appsettings.json` to make your app target Azure SignalR Service.



### 3. Publish the app.

Now the app is ready to be published. Upon the completion of the publishing process, the app automatically launches in a browser.

#### NOTE

The app may require some time to start due to the Azure App Service deployment start latency. You can use the browser debugger tools (usually by pressing F12) to ensure that the traffic has been redirected to Azure SignalR Service.

```
[2020-09-08T08:23:16.286Z] Information: Normalizing '_blazor' to 'https://<REDACTED>.azurewebsites.net/\_blazor'. blazor.server.js:1
[2020-09-08T08:23:17.813Z] Information: WebSocket connected to wss://<REDACTED>.service.signalr.net/client/?hub=componenthub&asr blazor.server.js:1
s.op=%2F_bla_W50Lz9odWI9Y29tG9uZW50ahVlIn0.qz3Srd6Hgh618uFXUU_KnkwoLoPrvBaH4KyR4r_Ozz8.
```

[Having issues? Let us know.](#)

## Enable Azure SignalR Service for local development

1. Add a reference to the Azure SignalR SDK using the following command.

```
dotnet add package Microsoft.Azure.SignalR --version 1.5.1
```

2. Add a call to `AddAzureSignalR()` in `Startup.ConfigureServices()` as demonstrated below.

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddSignalR().AddAzureSignalR();
    ...
}
```

3. Configure the Azure SignalR Service connection string either in `appsettings.json` or by using the [Secret Manager](#) tool.

#### NOTE

Step 2 can be replaced with configuring [Hosting Startup Assemblies](#) to use the SignalR SDK.

1. Add the configuration to turn on Azure SignalR Service in `appsettings.json`:

```
"Azure": {
    "SignalR": {
        "Enabled": true,
        "ConnectionString": <your-connection-string>
    }
}
```

2. Configure the hosting startup assembly to use the Azure SignalR SDK. Edit `launchSettings.json` and add a configuration like the following example inside `environmentVariables`:

```
"environmentVariables": {
    ...
    "ASPNETCORE_HOSTINGSTARTUPASSEMBLIES": "Microsoft.Azure.SignalR"
}
```

[Having issues? Let us know.](#)

## Clean up resources

To clean up the resources created in this tutorial, delete the resource group using the Azure portal.

## Additional resources

- [ASP.NET Core Blazor](#)

## Next steps

In this tutorial, you learned how to:

- Build a simple chat room with the Blazor Server app template.
- Work with Razor components.
- Use event handling and data binding in Razor components.
- Quick-deploy to Azure App Service in Visual Studio.

- Migrate from local SignalR to Azure SignalR Service.

Read more about high availability:

[Resiliency and disaster recovery](#)

# Azure CLI reference

11/2/2020 • 2 minutes to read • [Edit Online](#)

The following table includes links to bash scripts for the Azure SignalR Service using the Azure CLI.

SCRIPT	DESCRIPTIONS
<b>Create</b>	
<a href="#">Create a new SignalR Service and resource group</a>	Creates a new Azure SignalR Service resource in a new resource group with a random name.
<b>Integrate</b>	
<a href="#">Create a new SignalR Service and Web App configured to use SignalR</a>	Creates a new Azure SignalR Service resource in a new resource group with a random name. Also adds a new Web App and App Service plan to host an ASP.NET Core Web App that uses the SignalR Service. The web app is configured with an App Setting to connect to the new SignalR service resource.
<a href="#">Create a new SignalR Service and Web App configured to use SignalR, and GitHub OAuth</a>	Creates a new Azure SignalR Service resource in a new resource group with a random name. Also adds a new Azure Web App and hosting plan to host an ASP.NET Core Web App that uses the SignalR Service. The web app is configured with app settings for the connection string to the new SignalR service resource, and client secrets to support <a href="#">GitHub authentication</a> as demonstrated in the <a href="#">authentication tutorial</a> . The web app is also configured to use a local git repository deployment source.

# Azure SignalR Service internals

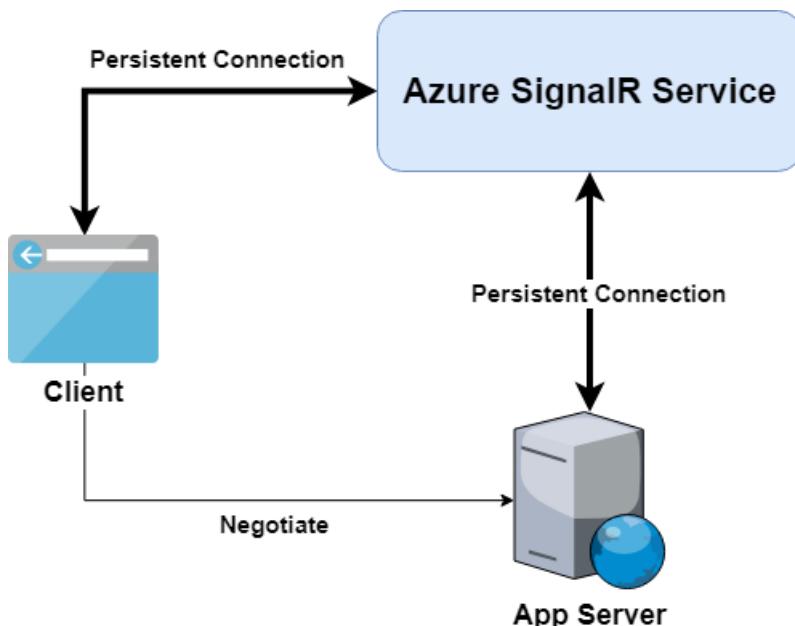
5/25/2021 • 2 minutes to read • [Edit Online](#)

Azure SignalR Service is built on top of ASP.NET Core SignalR framework. It also supports ASP.NET SignalR by reimplementing ASP.NET SignalR's data protocol on top of the ASP.NET Core framework.

You can easily migrate a local ASP.NET Core SignalR application or ASP.NET SignalR application to work with SignalR Service, with a few lines of code change.

The diagram below describes the typical architecture when you use the SignalR Service with your application server.

The differences from self-hosted ASP.NET Core SignalR application are discussed as well.



## Server connections

Self-hosted ASP.NET Core SignalR application server listens to and connects clients directly.

With SignalR Service, the application server is no longer accepting persistent client connections, instead:

1. A `negotiate` endpoint is exposed by Azure SignalR Service SDK for each hub.
2. This endpoint will respond to client's negotiation requests and redirect clients to SignalR Service.
3. Eventually, clients will be connected to SignalR Service.

For more information, see [Client connections](#).

Once the application server is started,

- For ASP.NET Core SignalR, Azure SignalR Service SDK opens 5 WebSocket connections per hub to SignalR Service.
- For ASP.NET SignalR, Azure SignalR Service SDK opens 5 WebSocket connections per hub to SignalR Service, and one per application WebSocket connection.

5 WebSocket connections is the default value that can be changed in [configuration](#).

Messages to and from clients will be multiplexed into these connections.

These connections will remain connected to the SignalR Service all the time. If a server connection is disconnected for network issue,

- all clients that are served by this server connection disconnect (for more information about it, see [Data transmit between client and server](#));
- the server connection starts reconnecting automatically.

## Client connections

When you use the SignalR Service, clients connect to SignalR Service instead of application server. There are two steps to establish persistent connections between the client and the SignalR Service.

1. Client sends a negotiate request to the application server. With Azure SignalR Service SDK, application server returns a redirect response with SignalR Service's URL and access token.

- For ASP.NET Core SignalR, a typical redirect response looks like:

```
{  
  "url": "https://test.service.signalr.net/client/?hub=chat&... ",  
  "accessToken": "<a typical JWT token>"  
}
```

- For ASP.NET SignalR, a typical redirect response looks like:

```
{  
  "ProtocolVersion": "2.0",  
  "RedirectUrl": "https://test.service.signalr.net/aspnetclient",  
  "AccessToken": "<a typical JWT token>"  
}
```

2. After receiving the redirect response, client uses the new URL and access token to start the normal process to connect to SignalR Service.

Learn more about ASP.NET Core SignalR's [transport protocols](#).

## Data transmit between client and server

When a client is connected to the SignalR Service, service runtime will find a server connection to serve this client

- This step happens only once, and is a one-to-one mapping between the client and server connections.
- The mapping is maintained in SignalR Service until the client or server disconnects.

At this point, the application server receives an event with information from the new client. A logical connection to the client is created in the application server. The data channel is established from client to application server, via SignalR Service.

SignalR Service transmits data from the client to the pairing application server. And data from the application server will be sent to the mapped clients.

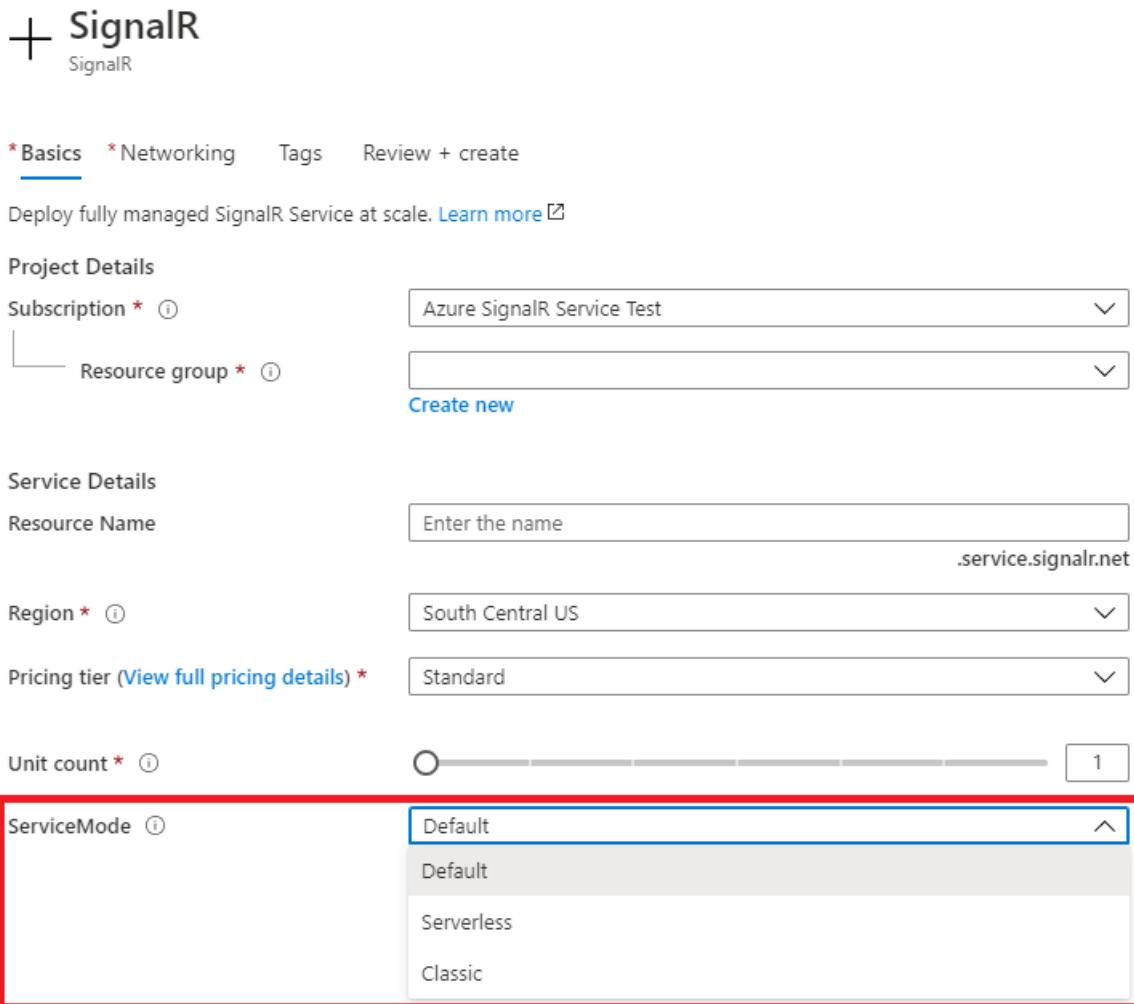
SignalR Service does not save or store customer data, all customer data received is transmitted to target server or clients in real-time.

As you can see, the Azure SignalR Service is essentially a logical transport layer between application server and clients. All persistent connections are offloaded to SignalR Service. Application server only needs to handle the business logic in hub class, without worrying about client connections.

# Service mode in Azure SignalR Service

11/2/2020 • 6 minutes to read • [Edit Online](#)

Service mode is an important concept in Azure SignalR Service. When you create a new SignalR resource, you will be asked to specify a service mode:



The screenshot shows the 'SignalR' creation wizard in the Azure portal. The 'Basics' tab is selected. In the 'Project Details' section, 'Subscription' is set to 'Azure SignalR Service Test' and 'Resource group' is set to 'Create new'. In the 'Service Details' section, 'Resource Name' is 'Enter the name .service.signalr.net', 'Region' is 'South Central US', 'Pricing tier' is 'Standard', and 'Unit count' is set to 1. The 'ServiceMode' dropdown is open, showing options: Default (selected), Default, Serverless, and Classic. A red box highlights the 'ServiceMode' dropdown.

\* Basics \* Networking Tags Review + create

Deploy fully managed SignalR Service at scale. [Learn more](#)

**Project Details**

Subscription \* Azure SignalR Service Test

Resource group \* Create new

[Create new](#)

**Service Details**

Resource Name  Enter the name .service.signalr.net

Region \* South Central US

Pricing tier ([View full pricing details](#)) \* Standard

Unit count \* 1

ServiceMode Default

Default  
Serverless  
Classic

You can also change it later in the settings menu:

Search (Ctrl+ /) Save Discard

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Events

Settings

- Keys
- Quickstart
- Scale
- Settings**
- CORS

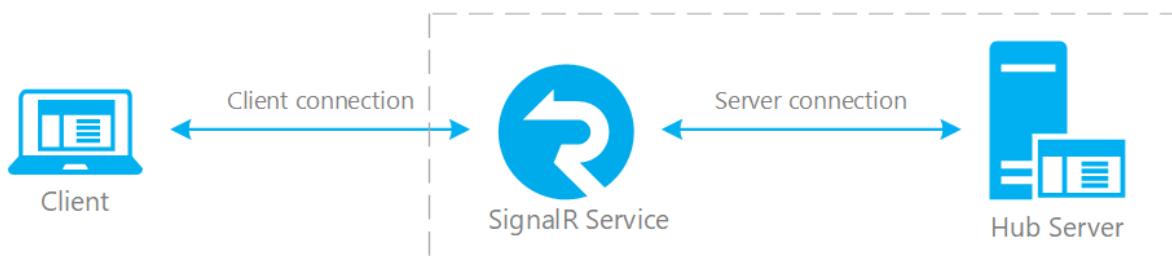
Service Mode

**Default** Serverless Classic

Azure SignalR Service currently supports three service modes: **default**, **serverless** and **classic**. Your SignalR resource will behave differently in different modes. In this article, you'll learn their differences and how to choose the right service mode based on your scenario.

## Default mode

Default mode is the default value for service mode when you create a new SignalR resource. In this mode, your application works as a typical ASP.NET Core (or ASP.NET) SignalR application, where you have a web server that hosts a hub (called hub server hereinafter) and clients can have duplex real-time communication with the hub server. The only difference is instead of connecting client and server directly, client and server both connect to SignalR service and use the service as a proxy. Below is a diagram that illustrates the typical application structure in default mode:



So if you have a SignalR application and want to integrate with SignalR service, default mode should be the right choice for most cases.

### Connection routing in default mode

In default mode, there will be websocket connections between hub server and SignalR service (called server connections). These connections are used to transfer messages between server and client. When a new client is connected, SignalR service will route the client to one hub server (assume you have more than one server) through existing server connections. Then the client connection will stick to the same hub server during its lifetime. When client sends messages, they always go to the same hub server. With this behavior, you can safely maintain some states for individual connections on your hub server. For example, if you want to stream something between server and client, you don't need to consider the case that data packets go to different servers.

#### IMPORTANT

This also means in default mode client cannot connect without server being connected first. If all your hub servers are disconnected due to network interruption or server reboot, your client connect will get an error telling you no server is connected. So it's your responsibility to make sure at any time there is at least one hub server connected to SignalR service (for example, have multiple hub servers and make sure they won't go offline at the same time for things like maintenance).

This routing model also means when a hub server goes offline, the connections routed that server will be dropped. So you should expect connection drop when your hub server is offline for maintenance and handle reconnect properly so that it won't have negative impact to your application.

## Serverless mode

Serverless mode, as its name implies, is a mode that you cannot have any hub server. Comparing to default mode, in this mode client doesn't require hub server to get connected. All connections are connected to service in a "serverless" mode and service is responsible for maintaining client connections like handling client pings (in default mode this is handled by hub servers).

Also there is no server connection in this mode (if you try to use service SDK to establish server connection, you will get an error). Therefore there is also no connection routing and server-client stickiness (as described in the default mode section). But you can still have server-side application to push messages to clients. This can be done in two ways, use [REST APIs](#) for one-time send, or through a websocket connection so that you can send multiple messages more efficiently (note this websocket connection is different than server connection).

#### NOTE

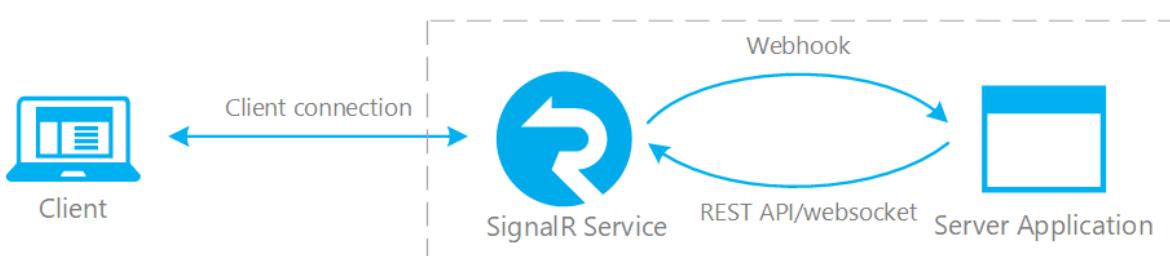
Both REST API and websocket way are supported in SignalR service [management SDK](#). If you're using a language other than .NET, you can also manually invoke the REST APIs following this [spec](#).

If you're using Azure Functions, you can use [SignalR service bindings for Azure Functions](#) (hereinafter called function binding) to send messages as an output binding.

It's also possible for your server application to receive messages and connection events from clients. Service will deliver messages and connection events to preconfigured endpoints (called Upstream) using webhooks. Comparing to default mode, there is no guarantee of stickiness and HTTP requests may be less efficient than websocket connections.

For more information about how to configure upstream, see this [doc](#).

Below is a diagram that illustrates how serverless mode works:



#### **NOTE**

Please note in default mode you can also use REST API/management SDK/function binding to directly send messages to client if you don't want to go through hub server. But in default mode client connections are still handled by hub servers and upstream won't work in that mode.

## Classic mode

Classic is a mixed mode of default and serverless mode. In this mode, connection mode is decided by whether there is hub server connected when client connection is established. If there is hub server, client connection will be routed to a hub server. Otherwise it will enter a serverless mode where client to server message cannot be delivered to hub server. This will cause some discrepancies, for example if all hub servers are unavailable for a short time, all client connections created during that time will be in serverless mode and cannot send messages to hub server.

#### **NOTE**

Classic mode is mainly for backward compatibility for those applications created before there is default and serverless mode. It's strongly recommended to not use this mode anymore. For new applications, please choose default or serverless based on your scenario. For existing applications, it's also recommended to review your use cases and choose a proper service mode.

Classic mode also doesn't support some new features like upstream in serverless mode.

## Choose the right service mode

Now you should understand the differences between service modes and know how to choose between them. As you already learned in the previous section, classic mode is not encouraged and you should only choose between default and serverless. Here are some more tips that can help you make the right choice for new applications and retire classic mode for existing applications.

- If you're already familiar with how SignalR library works and want to move from a self-hosted SignalR to use Azure SignalR Service, choose default mode. Default mode works exactly the same way as self-hosted SignalR (and you can use the same programming model in SignalR library), SignalR service just acts as a proxy between clients and hub servers.
- If you're creating a new application and don't want to maintain hub server and server connections, choose serverless mode. This mode usually works together with Azure Functions so you don't need to maintain any server at all. You can still have duplex communications (with REST API/management SDK/function binding + upstream) but the programming model will be different than SignalR library.
- If you have both hub servers to serve client connections and backend application to directly push messages to clients (for example through REST API), you should still choose default mode. Keep in mind that the key difference between default and serverless mode is whether you have hub servers and how client connections are routed. REST API/management SDK/function binding can be used in both modes.
- If you really have a mixed scenario, for example, you have two different hubs on the same SignalR resource, one used as a traditional SignalR hub and the other one used with Azure Functions and doesn't have hub server, you should really consider to separate them into two SignalR resources, one in default mode and one in serverless mode.

## Next steps

To learn more about how to use default and serverless mode, read the following articles:

- Azure SignalR Service internals
- Azure Functions development and configuration with Azure SignalR Service

# Scale ASP.NET Core SignalR applications with Azure SignalR Service

11/2/2020 • 2 minutes to read • [Edit Online](#)

## Developing SignalR apps

Currently, there are [two versions](#) of SignalR you can use with your web applications: SignalR for ASP.NET, and ASP.NET Core SignalR, which is the newest version. The Azure SignalR Service is an Azure-managed service built on ASP.NET Core SignalR.

ASP.NET Core SignalR is a rewrite of the previous version. As a result, ASP.NET Core SignalR is not backward compatible with the earlier SignalR version. The APIs and behaviors are different. The ASP.NET Core SignalR SDK targets .NET Standard so you can still use it with the .NET Framework. However, you must use the new APIs instead of old ones. If you're using SignalR and want to move to ASP.NET Core SignalR, or Azure SignalR Service, you'll need to change your code to handle differences in the APIs.

With Azure SignalR Service, the server-side component of ASP.NET Core SignalR is hosted in Azure. However, since the technology is built on top of ASP.NET Core, you have the ability to run your actual web application on multiple platforms (Windows, Linux, and MacOS) while hosting with [Azure App Service](#), [IIS](#), [Nginx](#), [Apache](#), [Docker](#). You can also use self-hosting in your own process.

If the goals for your application include: supporting the latest functionality for updating web clients with real-time content updates, running across multiple platforms (Azure, Windows, Linux, and macOS), and hosting in different environments, then the best choice could be leveraging the Azure SignalR Service.

## Why not deploy SignalR myself?

It is still a valid approach to deploy your own Azure web app supporting ASP.NET Core SignalR as a backend component to your overall web application.

One of the key reasons to use the Azure SignalR Service is simplicity. With Azure SignalR Service, you don't need to handle problems like performance, scalability, availability. These issues are handled for you with a 99.9% service-level agreement.

Also, WebSockets are typically the preferred technique to support real-time content updates. However, load balancing a large number of persistent WebSocket connections becomes a complicated problem to solve as you scale. Common solutions leverage: DNS load balancing, hardware load balancers, and software load balancing. Azure SignalR Service handles this problem for you.

Another reason may be you have no requirements to actually host a web application at all. The logic of your web application may leverage [Serverless computing](#). For example, maybe your code is only hosted and executed on demand with [Azure Functions](#) triggers. This scenario can be tricky because your code only runs on-demand and doesn't maintain long connections with clients. Azure SignalR Service can handle this situation since the service already manages connections for you. See the [overview on how to use SignalR Service with Azure Functions](#) for more details.

## How does it scale?

It is common to scale SignalR with SQL Server, Azure Service Bus, or Azure Cache for Redis. Azure SignalR Service handles the scaling approach for you. The performance and cost is comparable to these approaches without the complexity of dealing with these other services. All you have to do is update the unit count for your

service. Each unit supports up to 1000 client connections.

## Next steps

- [Quickstart: Create a chat room with Azure SignalR](#)

# Build real-time Apps with Azure Functions and Azure SignalR Service

11/2/2020 • 2 minutes to read • [Edit Online](#)

Because Azure SignalR Service and Azure Functions are both fully managed, highly scalable services that allow you to focus on building applications instead of managing infrastructure, it's common to use the two services together to provide real-time communications in a [serverless](#) environment.

## NOTE

Learn to use SignalR and Azure Functions together in the interactive tutorial [Enable automatic updates in a web application using Azure Functions and SignalR Service](#).

## Integrate real-time communications with Azure services

Azure Functions allow you to write code in [several languages](#), including JavaScript, Python, C#, and Java, that triggers whenever events occur in the cloud. Examples of these events include:

- HTTP and webhook requests
- Periodic timers
- Events from Azure services, such as:
  - Event Grid
  - Event Hubs
  - Service Bus
  - Cosmos DB change feed
  - Storage - blobs and queues
  - Logic Apps connectors such as Salesforce and SQL Server

By using Azure Functions to integrate these events with Azure SignalR Service, you have the ability to notify thousands of clients whenever events occur.

Some common scenarios for real-time serverless messaging that you can implement with Azure Functions and SignalR Service include:

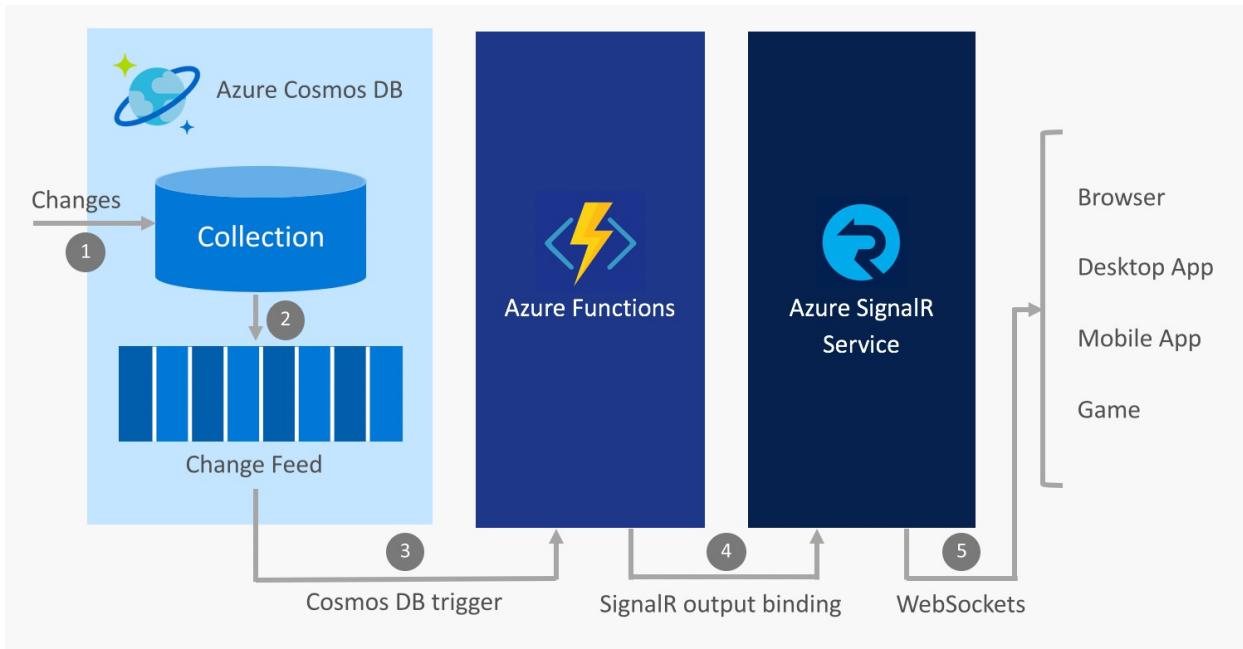
- Visualize IoT device telemetry on a real-time dashboard or map
- Update data in an application when documents update in Cosmos DB
- Send in-app notifications when new orders are created in Salesforce

## SignalR Service bindings for Azure Functions

The SignalR Service bindings for Azure Functions allow an Azure Function app to publish messages to clients connected to SignalR Service. Clients can connect to the service using a SignalR client SDK that is available in .NET, JavaScript, and Java, with more languages coming soon.

### An example scenario

An example of how to use the SignalR Service bindings is using Azure Functions to integrate with Azure Cosmos DB and SignalR Service to send real-time messages when new events appear on a Cosmos DB change feed.



1. A change is made in a Cosmos DB collection
2. The change event is propagated to the Cosmos DB change feed
3. An Azure Functions is triggered by the change event using the Cosmos DB trigger
4. The SignalR Service output binding publishes a message to SignalR Service
5. SignalR Service publishes the message to all connected clients

#### Authentication and users

SignalR Service allows you to broadcast messages to all clients or only to a subset of clients, such as those belonging to a single user. The SignalR Service bindings for Azure Functions can be combined with App Service Authentication to authenticate users with providers such as Azure Active Directory, Facebook, and Twitter. You can then send messages directly to these authenticated users.

## Next steps

In this article, you got an overview of how to use Azure Functions with SignalR Service to enable a wide array of serverless real-time messaging scenarios.

For full details on how to use Azure Functions and SignalR Service together visit the following resources:

- [Azure Functions development and configuration with SignalR Service](#)
- [Enable automatic updates in a web application using Azure Functions and SignalR Service](#)

Follow one of these quickstarts to learn more.

- [Azure SignalR Service Serverless Quickstart - C#](#)
- [Azure SignalR Service Serverless Quickstart - JavaScript](#)

# Azure Functions development and configuration with Azure SignalR Service

3/5/2021 • 9 minutes to read • [Edit Online](#)

Azure Functions applications can leverage the [Azure SignalR Service bindings](#) to add real-time capabilities. Client applications use client SDKs available in several languages to connect to Azure SignalR Service and receive real-time messages.

This article describes the concepts for developing and configuring an Azure Function app that is integrated with SignalR Service.

## SignalR Service configuration

Azure SignalR Service can be configured in different modes. When used with Azure Functions, the service must be configured in *Serverless* mode.

In the Azure portal, locate the *Settings* page of your SignalR Service resource. Set the *Service mode* to *Serverless*.



## Azure Functions development

A serverless real-time application built with Azure Functions and Azure SignalR Service typically requires two Azure Functions:

- A "negotiate" function that the client calls to obtain a valid SignalR Service access token and service endpoint URL
- One or more functions that handle messages from SignalR Service and send messages or manage group membership

### **negotiate function**

A client application requires a valid access token to connect to Azure SignalR Service. An access token can be anonymous or authenticated to a given user ID. Serverless SignalR Service applications require an HTTP endpoint named "negotiate" to obtain a token and other connection information, such as the SignalR Service endpoint URL.

Use an HTTP triggered Azure Function and the *SignalRConnectionInfo* input binding to generate the connection information object. The function must have an HTTP route that ends in `/negotiate`.

With [class based model](#) in C#, you don't need *SignalRConnectionInfo* input binding and can add custom claims much easier. See [Negotiate experience in class based model](#)

For more information on how to create the negotiate function, see the [SignalRConnectionInfo input binding reference](#).

To learn about how to create an authenticated token, refer to [Using App Service Authentication](#).

### **Handle messages sent from SignalR Service**

Use the *SignalR Trigger* binding to handle messages sent from SignalR Service. You can get notified when clients send messages or clients get connected or disconnected.

For more information, see the [SignalR trigger binding reference](#).

You also need to configure your function endpoint as an upstream so that service will trigger the function when there is message from client. For more information about how to configure upstream, please refer to this [doc](#).

### Sending messages and managing group membership

Use the *SignalR* output binding to send messages to clients connected to Azure SignalR Service. You can broadcast messages to all clients, or you can send them to a subset of clients that are authenticated with a specific user ID or have been added to a specific group.

Users can be added to one or more groups. You can also use the *SignalR* output binding to add or remove users to/from groups.

For more information, see the [SignalR output binding reference](#).

### SignalR Hubs

SignalR has a concept of "hubs". Each client connection and each message sent from Azure Functions is scoped to a specific hub. You can use hubs as a way to separate your connections and messages into logical namespaces.

## Class based model

The class based model is dedicated for C#. With class based model can have a consistent SignalR server-side programming experience. It has the following features.

- Less configuration work: The class name is used as `HubName`, the method name is used as `Event` and the `Category` is decided automatically according to method name.
- Auto parameter binding: Neither `ParameterNames` nor attribute `[SignalRParameter]` is needed. Parameters are auto bound to arguments of Azure Function method in order.
- Convenient output and negotiate experience.

The following codes demonstrate these features:

```

public class SignalRTTestHub : ServerlessHub
{
    [FunctionName("negotiate")]
    public SignalRConnectionInfo Negotiate([HttpTrigger(AuthorizationLevel.Anonymous)]HttpRequest req)
    {
        return Negotiate(req.Headers["x-ms-signalr-user-id"], GetClaims(req.Headers["Authorization"]));
    }

    [FunctionName(nameof(OnConnected))]
    public async Task OnConnected([SignalRTTrigger]InvocationContext invocationContext, ILogger logger)
    {
        await Clients.All.SendAsync(NewConnectionTarget, new NewConnection(invocationContext.ConnectionId));
        logger.LogInformation($"{invocationContext.ConnectionId} has connected");
    }

    [FunctionName(nameof(Broadcast))]
    public async Task Broadcast([SignalRTTrigger]InvocationContext invocationContext, string message, ILogger logger)
    {
        await Clients.All.SendAsync(NewMessageTarget, new NewMessage(invocationContext, message));
        logger.LogInformation($"{invocationContext.ConnectionId} broadcast {message}");
    }

    [FunctionName(nameof(OnDisconnected))]
    public void OnDisconnected([SignalRTTrigger]InvocationContext invocationContext)
    {
    }
}

```

All functions that want to leverage class based model need to be the method of class that inherits from `ServerlessHub`. The class name `SignalRTTestHub` in the sample is the hub name.

### Define hub method

All the hub methods **must** have an argument of `InvocationContext` decorated by `[SignalRTTrigger]` attribute and use parameterless constructor. Then the **method name** is treated as parameter **event**.

By default, `category=messages` except the method name is one of the following names:

- **OnConnected:** Treated as `category=connections, event=connected`
- **OnDisconnected:** Treated as `category=connections, event=disconnected`

### Parameter binding experience

In class based model, `[SignalRParameter]` is unnecessary because all the arguments are marked as `[SignalRParameter]` by default except it is one of the following situations:

- The argument is decorated by a binding attribute.
- The argument's type is `ILogger` or `CancellationToken`
- The argument is decorated by attribute `[SignalRIgnore]`

### Negotiate experience in class based model

Instead of using SignalR input binding `[SignalR]`, negotiation in class based model can be more flexible. Base class `ServerlessHub` has a method

```
SignalRConnectionInfo Negotiate(string userId = null, IList<Claim> claims = null, TimeSpan? lifeTime = null)
```

This features user customizes `userId` or `claims` during the function execution.

### Use `SignalRFilterAttribute`

User can inherit and implement the abstract class `SignalRFilterAttribute`. If exceptions are thrown in `FilterAsync`, `403 Forbidden` will be sent back to clients.

The following sample demonstrates how to implement a customer filter that only allows the `admin` to invoke `broadcast`.

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = true, Inherited = true)]
internal class FunctionAuthorizeAttribute: SignalRFilterAttribute
{
    private const string AdminKey = "admin";

    public override Task FilterAsync(InvocationContext invocationContext, CancellationToken cancellationToken)
    {
        if (invocationContext.Claims.TryGetValue(AdminKey, out var value) &&
            bool.TryParse(value, out var isAdmin) &&
            isAdmin)
        {
            return Task.CompletedTask;
        }

        throw new Exception($"'{invocationContext.ConnectionId}' doesn't have admin role");
    }
}
```

Leverage the attribute to authorize the function.

```
[FunctionAuthorize]
[FunctionName(nameof(Broadcast))]
public async Task Broadcast([SignalRTrigger]InvocationContext invocationContext, string message, ILogger logger)
{}
```

## Client development

SignalR client applications can leverage the SignalR client SDK in one of several languages to easily connect to and receive messages from Azure SignalR Service.

### Configuring a client connection

To connect to SignalR Service, a client must complete a successful connection negotiation that consists of these steps:

1. Make a request to the `negotiate` HTTP endpoint discussed above to obtain valid connection information
2. Connect to SignalR Service using the service endpoint URL and access token obtained from the `negotiate` endpoint

SignalR client SDKs already contain the logic required to perform the negotiation handshake. Pass the negotiate endpoint's URL, minus the `negotiate` segment, to the SDK's `HubConnectionBuilder`. Here is an example in JavaScript:

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl('https://my-signalr-function-app.azurewebsites.net/api')
    .build()
```

By convention, the SDK automatically appends `/negotiate` to the URL and uses it to begin the negotiation.

#### NOTE

If you are using the JavaScript/TypeScript SDK in a browser, you need to [enable cross-origin resource sharing \(CORS\)](#) on your Function App.

For more information on how to use the SignalR client SDK, refer to the documentation for your language:

- [.NET Standard](#)
- [JavaScript](#)
- [Java](#)

#### Sending messages from a client to the service

If you have [upstream](#) configured for your SignalR resource, you can send messages from client to your Azure Functions using any SignalR client. Here is an example in JavaScript:

```
connection.send('method1', 'arg1', 'arg2');
```

## Azure Functions configuration

Azure Function apps that integrate with Azure SignalR Service can be deployed like any typical Azure Function app, using techniques such as [continuously deployment](#), [zip deployment](#), and [run from package](#).

However, there are a couple of special considerations for apps that use the SignalR Service bindings. If the client runs in a browser, CORS must be enabled. And if the app requires authentication, you can integrate the negotiate endpoint with App Service Authentication.

#### Enabling CORS

The JavaScript/TypeScript client makes HTTP requests to the negotiate function to initiate the connection negotiation. When the client application is hosted on a different domain than the Azure Function app, cross-origin resource sharing (CORS) must be enabled on the Function app or the browser will block the requests.

#### localhost

When running the Function app on your local computer, you can add a `Host` section to `local.settings.json` to enable CORS. In the `Host` section, add two properties:

- `cors` - enter the base URL that is the origin the client application
- `CORScredentials` - set it to `true` to allow "withCredentials" requests

Example:

```
{
  "IsEncrypted": false,
  "Values": {
    // values
  },
  "Host": {
    "CORS": "http://localhost:8080",
    "CORScredentials": true
  }
}
```

#### Cloud - Azure Functions CORS

To enable CORS on an Azure Function app, go to the CORS configuration screen under the *Platform features* tab of your Function app in the Azure portal.

#### NOTE

CORS configuration is not yet available in Azure Functions Linux Consumption plan. Use [Azure API Management](#) to enable CORS.

CORS with Access-Control-Allow-Credentials must be enabled for the SignalR client to call the negotiate function. Select the checkbox to enable it.

In the *Allowed origins* section, add an entry with the origin base URL of your web application.

The screenshot shows the Azure portal interface. On the left, there's a sidebar with various icons and a search bar. The main area shows a 'Function Apps' blade for a 'serverless-flight-prices' app. In the center, there's an 'Overview' tab and a 'Platform F' tab. Under 'General Settings', there's a 'Request Credentials' section with a checked checkbox labeled 'Enable Access-Control-Allow-Credentials'. Below that is a 'Allowed Origins' section containing three entries: 'https://functions.azure.com', 'https://functions-staging.azure.com', and 'https://functions-next.azure.com'. A red box highlights the 'Request Credentials' checkbox and another red box highlights the 'Allowed Origins' input field for the custom origin URL.

#### Cloud - Azure API Management

Azure API Management provides an API gateway that adds capabilities to existing back-end services. You can use it to add CORS to your function app. It offers a consumption tier with pay-per-action pricing and a monthly free grant.

Refer to the API Management documentation for information on how to [import an Azure Function app](#). Once imported, you can add an inbound policy to enable CORS with Access-Control-Allow-Credentials support.

```
<cors allow-credentials="true">
  <allowed-origins>
    <origin>https://azure-samples.github.io</origin>
  </allowed-origins>
  <allowed-methods>
    <method>GET</method>
    <method>POST</method>
  </allowed-methods>
  <allowed-headers>
    <header>*</header>
  </allowed-headers>
  <expose-headers>
    <header>*</header>
  </expose-headers>
</cors>
```

Configure your SignalR clients to use the API Management URL.

#### Using App Service Authentication

Azure Functions has built-in authentication, supporting popular providers such as Facebook, Twitter, Microsoft Account, Google, and Azure Active Directory. This feature can be integrated with the *SignalRConnectionInfo* binding to create connections to Azure SignalR Service that have been authenticated to a user ID. Your application can send messages using the *SignalR* output binding that are targeted to that user ID.

In the Azure portal, in your Function app's *Platform features* tab, open the *Authentication/authorization* settings window. Follow the documentation for [App Service Authentication](#) to configure authentication using an identity provider of your choice.

Once configured, authenticated HTTP requests will include `x-ms-client-principal-name` and `x-ms-client-principal-id` headers containing the authenticated identity's username and user ID, respectively.

You can use these headers in your *SignalRConnectionInfo* binding configuration to create authenticated connections. Here is an example C# negotiate function that uses the `x-ms-client-principal-id` header.

```
[FunctionName("negotiate")]
public static SignalRConnectionInfo Negotiate(
    [HttpTrigger(AuthorizationLevel.Anonymous)]HttpRequest req,
    [SignalRConnectionInfo
        (HubName = "chat", UserId = "{headers.x-ms-client-principal-id}")]
    SignalRConnectionInfo connectionInfo)
{
    // connectionInfo contains an access key token with a name identifier claim set to the authenticated user
    return connectionInfo;
}
```

You can then send messages to that user by setting the `UserId` property of a SignalR message.

```
[FunctionName("SendMessage")]
public static Task SendMessage(
    [HttpTrigger(AuthorizationLevel.Anonymous, "post")]object message,
    [SignalR(HubName = "chat")]IAsyncCollector<SignalRMessage> signalRMessages)
{
    return signalRMessages.AddAsync(
        new SignalRMessage
        {
            // the message will only be sent to these user IDs
            UserId = "userId1",
            Target = "newMessage",
            Arguments = new [] { message }
        });
}
```

For information on other languages, see the [Azure SignalR Service bindings](#) for Azure Functions reference.

## Next steps

In this article, you have learned how to develop and configure serverless SignalR Service applications using Azure Functions. Try creating an application yourself using one of the quick starts or tutorials on the [SignalR Service overview page](#).

# SignalR Service bindings for Azure Functions

6/30/2021 • 2 minutes to read • [Edit Online](#)

This set of articles explains how to authenticate and send real-time messages to clients connected to [Azure SignalR Service](#) by using SignalR Service bindings in Azure Functions. Azure Functions supports input and output bindings for SignalR Service.

ACTION	TYPE
Handle messages from SignalR Service	<a href="#">Trigger binding</a>
Return the service endpoint URL and access token	<a href="#">Input binding</a>
Send SignalR Service messages	<a href="#">Output binding</a>

## Add to your Functions app

### Functions 2.x and higher

Working with the trigger and bindings requires that you reference the appropriate package. The NuGet package is used for .NET class libraries while the extension bundle is used for all other application types.

LANGUAGE	ADD BY...	REMARKS
C#	Installing the <a href="#">NuGet package</a> , version 3.x	
C# Script, Java, JavaScript, Python, PowerShell	Registering the <a href="#">extension bundle</a>	The <a href="#">Azure Tools extension</a> is recommended to use with Visual Studio Code.
C# Script (online-only in Azure portal)	Adding a binding	To update existing binding extensions without having to republish your function app, see <a href="#">Update your extensions</a> .

For details on how to configure and use SignalR Service and Azure Functions together, refer to [Azure Functions development and configuration with Azure SignalR Service](#).

### Annotations library (Java only)

To use the SignalR Service annotations in Java functions, you need to add a dependency to the *azure-functions-java-library-signalr* artifact (version 1.0 or higher) to your *pom.xml* file.

```
<dependency>
  <groupId>com.microsoft.azure.functions</groupId>
  <artifactId>azure-functions-java-library-signalr</artifactId>
  <version>1.0.0</version>
</dependency>
```

## Connection string settings

Add the `AzureSignalRConnectionString` key to the `host.json` file that points to the application setting with your connection string. For local development, this value may exist in the `local.settings.json` file.

## Next steps

- [Handle messages from SignalR Service \(Trigger binding\)](#)
- [Return the service endpoint URL and access token \(Input binding\)](#)
- [Send SignalR Service messages \(Output binding\)](#)

# Upstream settings

3/5/2021 • 5 minutes to read • [Edit Online](#)

Upstream is a preview feature that allows Azure SignalR Service to send messages and connection events to a set of endpoints in serverless mode. You can use upstream to invoke a hub method from clients in serverless mode and let endpoints get notified when client connections are connected or disconnected.

## NOTE

Only serverless mode can configure upstream settings.

## Details of upstream settings

Upstream settings consist of a list of order-sensitive items. Each item consists of:

- A URL template, which specifies where messages send to.
- A set of rules.
- Authentication configurations.

When the specified event happens, an item's rules are checked one by one in order. Messages will be sent to the first matching item's upstream URL.

### URL template settings

You can parameterize the URL to support various patterns. There are three predefined parameters:

PREDEFINED PARAMETER	DESCRIPTION
{hub}	A hub is a concept of Azure SignalR Service. A hub is a unit of isolation. The scope of users and message delivery is constrained to a hub.
{category}	A category can be one of the following values: <ul style="list-style-type: none"><li>• <b>connections</b>: Connection lifetime events. It's fired when a client connection is connected or disconnected. It includes connected and disconnected events.</li><li>• <b>messages</b>: Fired when clients invoke a hub method. It includes all other events, except those in the <b>connections</b> category.</li></ul>
{event}	For the <b>messages</b> category, an event is the target in an <b>invocation message</b> that clients send. For the <b>connections</b> category, only <i>connected</i> and <i>disconnected</i> are used.

These predefined parameters can be used in the URL pattern. Parameters will be replaced with a specified value when you're evaluating the upstream URL. For example:

```
http://host.com/{hub}/api/{category}/{event}
```

When a client connection in the "chat" hub is connected, a message will be sent to this URL:

```
http://host.com/chat/api/connections/connected
```

When a client in the "chat" hub invokes the hub method `broadcast`, a message will be sent to this URL:

```
http://host.com/chat/api/messages/broadcast
```

### Key Vault secret reference in URL template settings

The URL of upstream is not encryption at rest. If you have any sensitive information, it's suggested to use Key Vault to save them where access control has better insurance. Basically, you can enable the managed identity of Azure SignalR Service and then grant read permission on a Key Vault instance and use Key Vault reference instead of plaintext in Upstream URL Pattern.

1. Add a system-assigned identity or user-assigned identity. See [How to add managed identity in Azure Portal](#)
2. Grant secret read permission for the managed identity in the Access policies in the Key Vault. See [Assign a Key Vault access policy using the Azure portal](#)
3. Replace your sensitive text with the syntax `{@Microsoft.KeyVault(SecretUri=<secret-identity>)}` in the Upstream URL Pattern.

#### NOTE

The secret content only rereads when you change the Upstream settings or change the managed identity. Make sure you have granted secret read permission to the managed identity before using the Key Vault secret reference.

### Rule settings

You can set rules for *hub rules*, *category rules*, and *event rules* separately. The matching rule supports three formats. Take event rules as an example:

- Use an asterisk(\*) to match any events.
- Use a comma (,) to join multiple events. For example, `connected, disconnected` matches the connected and disconnected events.
- Use the full event name to match the event. For example, `connected` matches the connected event.

#### NOTE

If you're using Azure Functions and [SignalR trigger](#), SignalR trigger will expose a single endpoint in the following format: `<Function_App_URL>/runtime/webhooks/signalr?code=<API_KEY>`. You can just configure **URL template settings** to this url and keep **Rule settings** default. See [SignalR Service integration](#) for details about how to find `<Function_App_URL>` and `<API_KEY>`.

### Authentication settings

You can configure authentication for each upstream setting item separately. When you configure authentication, a token is set in the `Authentication` header of the upstream message. Currently, Azure SignalR Service supports the following authentication types:

- `None`
- `ManagedIdentity`

When you select `ManagedIdentity`, you must enable a managed identity in Azure SignalR Service in advance and optionally specify a resource. See [Managed identities for Azure SignalR Service](#) for details.

# Create upstream settings via the Azure portal

1. Go to Azure SignalR Service.
2. Select **Settings** and switch **Service Mode** to **Serverless**. The upstream settings will appear:

The screenshot shows the Azure SignalR Service settings page. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Keys, Quickstart, Scale, and Settings. The Settings option is selected. In the main area, there's a note about choosing Default or Serverless mode. Below that, the Service Mode is set to Serverless. Under Upstream Settings, it says: "Upstreams allow external services to be notified when certain events for certain hubs happen. When the specified events happen, we'll send a POST request to the first matching URL pattern with the parameters evaluated. There are three supported parameters: {hub}, {event}, and {category}. The matching rule supports three formats. Take the Event Rules as an example:" followed by a bulleted list. At the bottom, there are buttons for Upstream URL Pattern, Hub Rules, Event Rules, and Category Rules, along with a text input field "Add an upstream URL pattern".

3. Add URLs under **Upstream URL Pattern**. Then settings such as **Hub Rules** will show the default value.
4. To set settings for **Hub Rules**, **Event Rules**, **Category Rules**, and **Upstream Authentication**, select the value of **Hub Rules**. A page that allows you to edit settings appears:

## Upstream Settings

Configure Upstream Settings

Upstream URL Pattern \* ⓘ

Hub Rules \* ⓘ

Event Rules \* ⓘ

Category Rules \* ⓘ

Upstream Authentication ⓘ  
 No Authentication  
 Use Managed Identity

5. To set **Upstream Authentication**, make sure you've enabled a managed identity first. Then select **Use Managed Identity**. According to your needs, you can choose any options under **Auth Resource ID**. See [Managed identities for Azure SignalR Service](#) for details.

## Create upstream settings via Resource Manager template

To create upstream settings by using an [Azure Resource Manager template](#), set the `upstream` property in the `properties` property. The following snippet shows how to set the `upstream` property for creating and updating upstream settings.

```
{
  "properties": {
    "upstream": {
      "templates": [
        {
          "UrlTemplate": "http://host.com/{hub}/api/{category}/{event}",
          "EventPattern": "*",
          "HubPattern": "*",
          "CategoryPattern": "*",
          "Auth": {
            "Type": "ManagedIdentity",
            "ManagedIdentity": {
              "Resource": "<resource>"
            }
          }
        }
      ]
    }
  }
}
```

## Serverless protocols

Azure SignalR Service sends messages to endpoints that follow the following protocols. You can use [SignalR Service trigger binding](#) with Function App, which handles these protocols for you.

### Method

POST

### Request header

NAME	DESCRIPTION
X-ASRS-Connection-Id	The connection ID for the client connection.
X-ASRS-Hub	The hub that the client connection belongs to.
X-ASRS-Category	The category that the message belongs to.
X-ASRS-Event	The event that the message belongs to.
X-ASRS-Signature	A hash-based message authentication code (HMAC) that's used for validation. See <a href="#">Signature</a> for details.
X-ASRS-User-Claims	A group of claims of the client connection.
X-ASRS-User-Id	The user identity of the client that sends the message.
X-ASRS-Client-Query	The query of the request when clients connect to the service.
Authentication	An optional token when you're using <a href="#">ManagedIdentity</a> .

### Request body

#### Connected

Content-Type: application/json

#### Disconnected

Content-Type: application/json

NAME	TYPE	DESCRIPTION
Error	string	The error message of a closed connection. Empty when connections close with no error.

#### Invocation message

Content-Type: application/json or application/x-msgpack

NAME	TYPE	DESCRIPTION
InvocationId	string	An optional string that represents an invocation message. Find details in <a href="#">Invocations</a> .
Target	string	The same as the event and the same as the target in an <a href="#">invocation message</a> .
Arguments	Array of object	An array that contains arguments to apply to the method referred to in <a href="#">Target</a> .

#### Signature

The service will calculate SHA256 code for the `X-ASRS-Connection-Id` value by using both the primary access key and the secondary access key as the `HMAC` key. The service will set it in the `X-ASRS-Signature` header when making HTTP requests to upstream:

```
Hex_encoded(HMAC_SHA256(accessKey, connection-id))
```

## Next steps

- [Managed identities for Azure SignalR Service](#)
- [Azure Functions development and configuration with Azure SignalR Service](#)
- [Handle messages from SignalR Service \(Trigger binding\)](#)
- [SignalR Service Trigger binding sample](#)

# Server graceful shutdown

3/5/2021 • 2 minutes to read • [Edit Online](#)

Microsoft Azure SignalR Service provides two modes for gracefully shutdown a server.

The key advantage of using this feature is to prevent your customer from experiencing unexpectedly connection drops.

Instead, you could either wait your client connections to close themselves with respect to your business logic, or even migrate the client connection to another server without losing data.

## How it works

In general, there will be four stages in a graceful shutdown process:

### 1. Set the server offline

It means no more client connections will be routed to this server.

### 2. Trigger `OnShutdown` hooks

You could register shutdown hooks for each hub you have owned in your server. They will be called with respect to the registered order right after we got an FINACK response from our Azure SignalR Service, which means this server has been set offline in the Azure SignalR Service.

You can broadcast messages or do some cleaning jobs in this stage, once all shutdown hooks has been executed, we will proceed to the next stage.

### 3. Wait until all client connections finished

, depends on the mode you choose, it could be:

#### Mode set to `WaitForClientsToClose`

Azure SignalR Service will hold existing clients.

You may have to design a way, like broadcast a closing message to all clients, and then let your clients to decide when to close/reconnect itself.

Read [ChatSample](#) for sample usage, which we broadcast a 'exit' message to trigger client close in shutdown hook.

#### Mode set to `MigrateClients`

Azure SignalR Service will try to reroute the client connection on this server to another valid server.

In this scenario, `OnConnectedAsync` and `OnDisconnectedAsync` will be triggered on the new server and the old server respectively with an `IConnectionMigrationFeature` set in the `HttpContext`, which can be used to identify if the client connection was being migrated-in or migrated-out. It could be useful especially for stateful scenarios.

The client connection will be immediately migrated after the current message has been delivered, which means the next message will be routed to the new server.

### 4. Stop server connections

After all client connections have been closed/migrated, or timeout (30s by default) exceeded,

SignalR Server SDK will proceed the shutdown process to this stage, and close all server connections.

Client connections will still be dropped if it failed to be closed/migrated. For example, no suitable target server / current client-to-server message hasn't finished.

## Sample codes.

Add following options when `AddAzureSignalR` :

```
services.AddSignalR().AddAzureSignalR(option =>
{
    option.GracefulShutdown.Mode = GracefulShutdownMode.WaitForClientsClose;
    // option.GracefulShutdown.Mode = GracefulShutdownMode.MigrateClients;
    option.GracefulShutdown.Timeout = TimeSpan.FromSeconds(30);

    option.GracefulShutdown.Add<Chat>(async (c) =>
    {
        await c.Clients.All.SendAsync("exit");
    });
});
```

**configure** `OnConnected` **and** `OnDisconnected` **while setting graceful shutdown mode to** `MigrateClients`.

We have introduced an "IConnectionMigrationFeature" to indicate if a connection was being migrated-in/out.

```
public class Chat : Hub {

    public override async Task OnConnectedAsync()
    {
        Console.WriteLine($"{Context.ConnectionId} connected.");

        var feature = Context.GetHttpContext().Features.Get<IConnectionMigrationFeature>();
        if (feature != null)
        {
            Console.WriteLine($"[{feature.MigrateTo}] {Context.ConnectionId} is migrated from {feature.MigrateFrom}.");
            // Your business logic.
        }

        await base.OnConnectedAsync();
    }

    public override async Task OnDisconnectedAsync(Exception e)
    {
        Console.WriteLine($"{Context.ConnectionId} disconnected.");

        var feature = Context.GetHttpContext().Features.Get<IConnectionMigrationFeature>();
        if (feature != null)
        {
            Console.WriteLine($"[{feature.MigrateFrom}] {Context.ConnectionId} will be migrated to {feature.MigrateTo}.");
            // Your business logic.
        }

        await base.OnDisconnectedAsync(e);
    }
}
```

# Authorize access to Azure SignalR Service resources using Azure Active Directory

3/5/2021 • 2 minutes to read • [Edit Online](#)

Azure SignalR Service supports using Azure Active Directory (Azure AD) to authorize requests to Azure SignalR Service resources. With Azure AD, you can use role-based access control (RBAC) to grant permissions to a security principal, which may be a user, or an application service principal. To learn more about roles and role assignments, see [Understanding the different roles](#).

## Overview

When a security principal (an application) attempts to access a Azure SignalR Service resource, the request must be authorized. With Azure AD, access to a resource is a two-step process.

1. First, the security principal's identity is authenticated, and an OAuth 2.0 token is returned. The resource name to request a token is `https://signalr.azure.com/`.
2. Next, the token is passed as part of a request to the Azure SignalR Service to authorize access to the specified resource.

The authentication step requires that an application request contains an OAuth 2.0 access token at runtime. If your hub server is running within an Azure entity such as an Azure VM, a virtual machine scale set, or an Azure Function app, it can use a managed identity to access the resources. To learn how to authenticate requests made by a managed identity to Azure SignalR Service, see [Authenticate access to Azure SignalR Service resources with Azure Active Directory and managed identities for Azure Resources](#).

The authorization step requires that one or more RBAC roles be assigned to the security principal. Azure SignalR Service provides RBAC roles that encompass sets of permissions for Azure SignalR resources. The roles that are assigned to a security principal determine the permissions that the principal will have. For more information about RBAC roles, see [Azure built-in roles for Azure SignalR Service](#).

SignalR Hub Server that isn't running within an Azure entity can also authorize with Azure AD. To learn how to request an access token and use it to authorize requests for Azure SignalR Service resources, see [Authenticate access to Azure SignalR Service with Azure AD from an application](#).

## Azure Built-in roles for Azure SignalR Service

- [SignalR App Server]
- [SignalR Service Reader]
- [SignalR Service Owner]

## Assign RBAC roles for access rights

Azure Active Directory (Azure AD) authorizes access rights to secured resources through [role-based access control \(RBAC\)](#). Azure SignalR Service defines a set of Azure built-in roles that encompass common sets of permissions used to access Azure SignalR Service and you can also define custom roles for accessing the resource.

When an RBAC role is assigned to an Azure AD security principal, Azure grants access to those resources for that security principal. Access can be scoped to the level of subscription, the resource group, or any Azure SignalR Service resource. An Azure AD security principal may be a user, or an application, or a [managed identity for](#)

[Azure resources](#).

## Built-in roles for Azure SignalR Service

Azure provides the following Azure built-in roles for authorizing access to Azure SignalR Service resource using Azure AD and OAuth:

### **SignalR App Server**

Use this role to give the access to Get a temporary access key for signing client tokens.

### **SignalR Serverless Contributor**

Use this role to give the access to Get a client token from Azure SignalR Service directly.

## Next steps

See the following related articles:

- [Authenticate an application with Azure AD to access Azure SignalR Service](#)
- [Authenticate a managed identity with Azure AD to access Azure SignalR Service](#)

# Authenticate an application with Azure Active Directory to access Azure SignalR Service

3/5/2021 • 4 minutes to read • [Edit Online](#)

Microsoft Azure provides integrated access control management for resources and applications based on Azure Active Directory (Azure AD). A key advantage of using Azure AD with Azure SignalR Service is that you don't need to store your credentials in the code anymore. Instead, you can request an OAuth 2.0 access token from the Microsoft Identity platform. The resource name to request a token is <https://signalr.azure.com/>. Azure AD authenticates the security principal (an application, resource group, or service principal) running the application. If the authentication succeeds, Azure AD returns an access token to the application, and the application can then use the access token to authorize requests to Azure SignalR Service resources.

When a role is assigned to an Azure AD security principal, Azure grants access to those resources for that security principal. Access can be scoped to the level of subscription, the resource group, or the Azure SignalR resource. An Azure AD security principal can assign roles to a user, a group, an application service principal, or a [managed identity for Azure resources](#).

## NOTE

A role definition is a collection of permissions. Role-based access control (RBAC) controls how these permissions are enforced through role assignment. A role assignment consists of three elements: security principal, role definition, and scope. For more information, see [Understanding the different roles](#).

## Register your application with an Azure AD tenant

The first step in using Azure AD to authorize Azure SignalR Service resources is registering your application with an Azure AD tenant from the [Azure portal](#). When you register your application, you supply information about the application to AD. Azure AD then provides a client ID (also called an application ID) that you can use to associate your application with Azure AD runtime. To learn more about the client ID, see [Application and service principal objects in Azure Active Directory](#).

The following images show steps for registering a web application:

Dashboard > App registrations > Register an application

## Register an application

\* Name

The user-facing display name for this application (this can be changed later).

### Supported account types

Who can use this application or access this API?

- Accounts in this organizational directory only (Default Directory only - Single tenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant)
- Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)

[Help me choose...](#)

### Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web  e.g. <https://myapp.com/auth>

[By proceeding, you agree to the Microsoft Platform Policies](#)

[Register](#)

### NOTE

If you register your application as a native application, you can specify any valid URI for the Redirect URI. For native applications, this value does not have to be a real URL. For web applications, the redirect URI must be a valid URI, because it specifies the URL to which tokens are provided.

After you've registered your application, you'll see the **Application (client) ID** under **Settings**:

Dashboard > App registrations > mywebappregistration

### mywebappregistration

Search (Ctrl+*Space*)

Welcome to the new and improved App registrations. Looking to learn how it's changed from App registrations (Legacy)? [→](#)

Display name mywebappregistration	Supported account types <a href="#">My organization only</a>
Application (client) ID <b>&lt;Application ID&gt;</b>	Redirect URIs 1 web, 0 public client
Directory (tenant) ID <Directory ID>	Managed application in local directory <a href="#">mywebappregistration</a>
Object ID <Object ID>	

**Call APIs**

Build more powerful apps with rich user and business data from Microsoft services and your own company's data sources.

[View API Permissions](#)

**Documentation**

[Microsoft identity platform](#)  
[Authentication scenarios](#)  
[Authentication libraries](#)  
[Code samples](#)  
[Microsoft Graph](#)  
[Glossary](#)  
[Help and Support](#)

For more information about registering an application with Azure AD, see [Integrating applications with Azure Active Directory](#).

### Create a client secret

The application needs a client secret to prove its identity when requesting a token. To add the client secret, follow these steps.

1. Navigate to your app registration in the Azure portal.
2. Select the **Certificates & secrets** setting.
3. Under **Client secrets**, select **New client secret** to create a new secret.
4. Provide a description for the secret, and choose the wanted expiration interval.
5. Immediately copy the value of the new secret to a secure location. The fill value is displayed to you only once.

#### Client secrets

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

Description	Expires	Value	
Password uploaded on Mon Apr 27 2020	4/27/2021	Y29*****	

### Upload a certificate

You could also upload a certification instead of creating a client secret.

#### Certificates

Certificates can be used as secrets to prove the application's identity when requesting a token. Also can be referred to as public keys.

Thumbprint	Start date	Expires	
E73DBBC99345AC4ED40BB721E20BEA12A689A403	4/13/2020	4/13/2021	

## Assign Azure roles using the Azure portal

To learn more on managing access to Azure resources using Azure RBAC and the Azure portal, see [this article](#).

After you've determined the appropriate scope for a role assignment, navigate to that resource in the Azure portal. Display the access control (IAM) settings for the resource, and follow these instructions to manage role assignments:

1. In the [Azure portal](#), navigate to your SignalR resource.
2. Select **Access Control (IAM)** to display access control settings for the Azure SignalR.
3. Select the **Role assignments** tab to see the list of role assignments. Select the **Add** button on the toolbar and then select **Add role assignment**.

The screenshot shows the Azure portal interface for managing access control (IAM) on a SignalR resource named 'tests158'. The 'Role assignments' tab is active, and the '+ Add' button is highlighted with a red box. The page displays a summary of role assignments for the subscription, with a count of 90 assigned to 2000 scopes. The toolbar includes buttons for 'Add', 'Manage view', 'Search (Ctrl+)', 'Download role assignments', 'Edit columns', 'Refresh', 'Remove', and 'Got feedback?'. The left sidebar lists other resources like 'azure-signalr-sdk-e2e-test', 'azure-signalr-serverless-event-grid', 'chenyinetwork', and 'hamiltart'.

4. On the **Add role assignment** page, do the following steps:
  - a. Select the **Azure SignalR** role that you want to assign.
  - b. Search to locate the **security principal** (user, group, service principal) to which you want to

assign the role.

- c. Select **Save** to save the role assignment.

The screenshot shows two windows side-by-side. On the left is the 'Access control (IAM)' blade for a resource named 'test335'. It has sections for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings, Keys, Quickstart, Scale, Settings, CORS, Identity, Private endpoint connections, Network access control, Properties, Locks, Monitoring, Alerts, Metrics, Diagnostic Settings, Automation, Tasks (preview), Export template, Support + troubleshooting, and New support request. On the right is the 'Add role assignment' dialog. In the 'Role' dropdown, 'SignalR App Server' is selected. In the 'Assign access to' dropdown, 'User, group, or service principal' is selected. In the 'Select' input field, 'signalr' is typed, and a list of results appears, including 'SignalR Smoke Test', 'signalr1', and 'signalr-service-test (Guest) signalrservice-test@outlook.com'. A red box highlights the 'Role' dropdown. In the 'Selected members:' section, 'signalrdev' is listed with a 'Remove' button next to it, also highlighted with a red box. At the bottom of the dialog are 'Save' and 'Discard' buttons.

- d. The identity to whom you assigned the role appears listed under that role. For example, the following image shows that application `signalr-dev` and `signalr-service` are in the SignalR App Server role.

SignalR App Server (Preview)			
<input type="checkbox"/>	signalr-dev	App	SignalR App Server (Preview) ⓘ This resource
<input type="checkbox"/>	signalr-service	App	SignalR App Server (Preview) ⓘ This resource

You can follow similar steps to assign a role scoped to resource group, or subscription. Once you define the role and its scope, you can test this behavior with samples [in this GitHub location](#).

## Sample codes while configuring your app server.

Add following options when `AddAzureSignalR` :

```
services.AddSignalR().AddAzureSignalR(option =>
{
    option.ConnectionString = "Endpoint=https://<name>.signalr.net;AuthType=aad;clientId=<clientId>;clientSecret=<clientSecret>;tenantId=<tenantId>";
});
```

Or simply configure the `ConnectionString` in your `appsettings.json` file.

```
{
"Azure": {
    "SignalR": {
        "Enabled": true,
        "ConnectionString": "Endpoint=https://<name>.signalr.net;AuthType=aad;clientId=<clientId>;clientSecret=<clientSecret>;tenantId=<tenantId>"
    }
}}
```

When using `certificate`, change the `clientSecret` to `clientCert` like this:

```
option.ConnectionString = "Endpoint=https://<name>.signalr.net;AuthType=aad;clientId=<clientId>;clientCert=<clientCertFilepath>;tenantId=<tenantId>";
```

## Next steps

- To learn more about RBAC, see [What is Azure role-based access control \(Azure RBAC\)?](#)
- To learn how to assign and manage Azure role assignments with Azure PowerShell, Azure CLI, or the REST API, see these articles:
  - [Manage role-based access control \(RBAC\) with Azure PowerShell](#)
  - [Manage role-based access control \(RBAC\) with Azure CLI](#)
  - [Manage role-based access control \(RBAC\) with the REST API](#)
  - [Manage role-based access control \(RBAC\) with Azure Resource Manager Templates](#)

See the following related articles:

- [Authenticate a managed identity with Azure Active Directory to access Azure SignalR Service](#)
- [Authorize access to Azure SignalR Service using Azure Active Directory](#)

# Authenticate a managed identity with Azure Active Directory to access Azure SignalR Resources

5/25/2021 • 3 minutes to read • [Edit Online](#)

Azure SignalR Service supports Azure Active Directory (Azure AD) authentication with [managed identities for Azure resources](#). Managed identities for Azure resources can authorize access to Azure SignalR Service resources using Azure AD credentials from applications running in Azure Virtual Machines (VMs), Function apps, Virtual Machine Scale Sets, and other services. By using managed identities for Azure resources together with Azure AD authentication, you can avoid storing credentials with your applications that run in the cloud.

This article shows how to authorize access to an Azure SignalR Service by using a managed identity.

## Enable managed identities

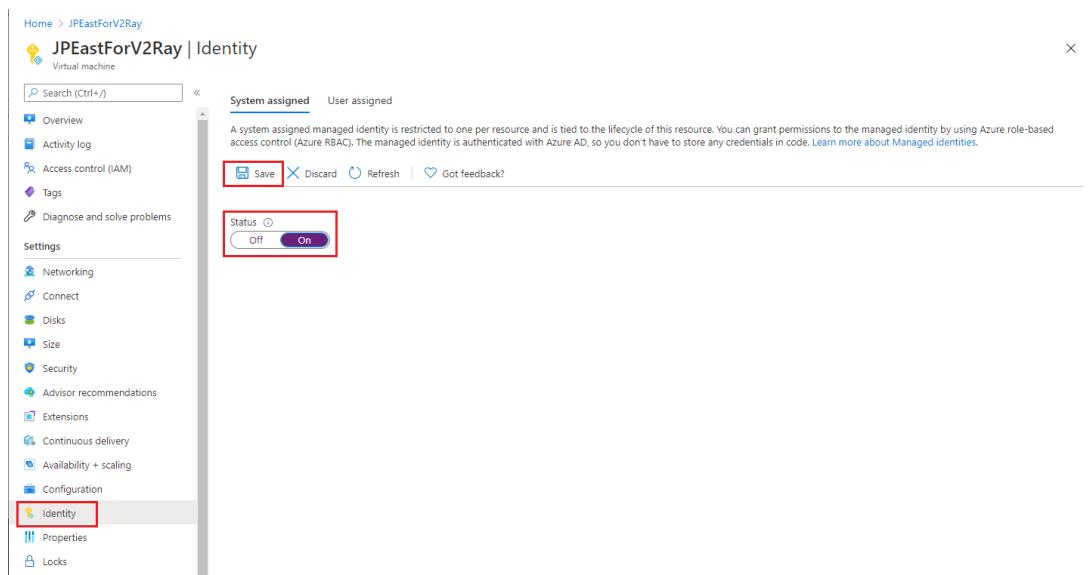
Before you can use managed identities for Azure Resources to authorize access to Azure SignalR Service resources, you must first enable managed identities for Azure Resources.

### For Azure resources on a VM

To learn how to enable managed identities for Azure Resources on a VM, see one of these articles:

- [Azure portal](#)

1. Go to **Settings** on Azure portal and select **Identity**.
2. Select the **Status** to be **On**.
3. Select **Save** to save the setting.



- [Azure PowerShell](#)
- [Azure CLI](#)
- [Azure Resource Manager template](#)
- [Azure Resource Manager client libraries](#)

### For Azure Functions

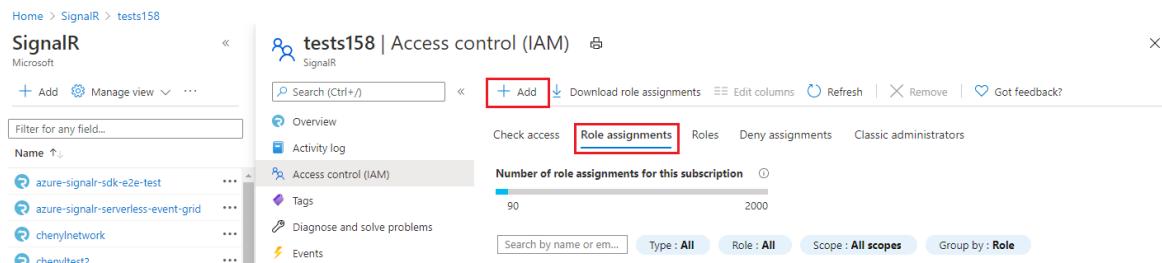
See [How to use managed identities for App Service and Azure Functions](#).

# Grant permissions to a managed identity in Azure AD

To authorize a request to Azure SignalR Service from a managed identity in your application, first configure role-based access control (RBAC) settings for that managed identity. Azure SignalR Service defines RBAC roles that encompass permissions for acquiring `AccessKey` or `ClientToken`. When the RBAC role is assigned to a managed identity, the managed identity is granted access to Azure SignalR Service at the appropriate scope.

Follow these instructions to manage role assignments:

1. In the [Azure portal](#), navigate to your SignalR resource.
2. Select **Access Control (IAM)** to display access control settings for the Azure SignalR.
3. Select the **Role assignments** tab to see the list of role assignments. Select the **Add** button on the toolbar and then select **Add role assignment**.



The screenshot shows the Azure portal interface for managing access control. The left sidebar lists resources under 'SignalR' (Microsoft). The main area is titled 'tests158 | Access control (IAM)' and shows the 'Role assignments' tab selected. A red box highlights the '+ Add' button. Other tabs include 'Overview', 'Activity log', 'Check access', 'Roles', 'Deny assignments', and 'Classic administrators'. A progress bar indicates 'Number of role assignments for this subscription' at 90 out of 2000. Filter and search options are available at the bottom.

4. On the **Add role assignment** page, do the following steps:
  - a. Select the **SignalR App Server** as the role. Note that this also applies to **Azure Functions App**.
  - b. Search to locate the **security principal** (user, group, service principal) to which you want to assign the role.
  - c. Select **Save** to save the role assignment.

- d. The identity to whom you assigned the role appears listed under that role. For example, the following image shows that application `signalr-dev` and `signalr-service` are in the SignalR App Server role.

SignalR App Server (Preview)			
		SignalR App Server (Preview) ⓘ	This resource
<input type="checkbox"/>	signalr-dev	App	
<input type="checkbox"/>	signalr-service	App	

You can follow similar steps to assign a role scoped to resource group, or subscription. Once you define the role and its scope, you can test this behavior with samples [in this GitHub location](#).

For more information about assigning RBAC roles, see [Authenticate with Azure Active Directory for access to Azure SignalR Service resources](#).

To learn more on managing access to Azure resources using Azure RBAC and the Azure portal, see [this article](#).

## Configure your app

### App server

Add following options when `AddAzureSignalR` :

```
services.AddSignalR().AddAzureSignalR(option =>
{
    option.ConnectionString = "Endpoint=https://<name>.signalr.net;AuthType=aad;Version=1.0;";
});
```

## Azure Functions App

On Azure portal, add an application setting with name `AzureSignalRConnectionString` and value `Endpoint=https://<name>.signalr.net;AuthType=aad;`.

On local, in your `local.appsettings.json` file, add in the `Values` section:

```
{
  "Values": {
    "AzureSignalRConnectionString": "Endpoint=https://<name>.signalr.net;AuthType=aad;Version=1.0;"
  }
}
```

## Next steps

- To learn more about RBAC, see [What is Azure role-based access control \(Azure RBAC\)?](#)
- To learn how to assign and manage Azure role assignments with Azure PowerShell, Azure CLI, or the REST API, see these articles:
  - [Manage role-based access control \(RBAC\) with Azure PowerShell](#)
  - [Manage role-based access control \(RBAC\) with Azure CLI](#)
  - [Manage role-based access control \(RBAC\) with the REST API](#)
  - [Manage role-based access control \(RBAC\) with Azure Resource Manager Templates](#)

See the following related articles:

- [Authenticate an application with Azure Active Directory to access Azure SignalR Service](#)
- [Authorize access to Azure SignalR Service using Azure Active Directory](#)

# Resiliency and disaster recovery in Azure SignalR Service

3/25/2021 • 6 minutes to read • [Edit Online](#)

Resiliency and disaster recovery is a common need for online systems. Azure SignalR Service already guarantees 99.9% availability, but it's still a regional service. Your service instance is always running in one region and won't fail-over to another region when there is a region-wide outage.

Instead, our service SDK provides a functionality to support multiple SignalR service instances and automatically switch to other instances when some of them are not available. With this feature, you'll be able to recover when a disaster takes place, but you will need to set up the right system topology by yourself. You'll learn how to do so in this document.

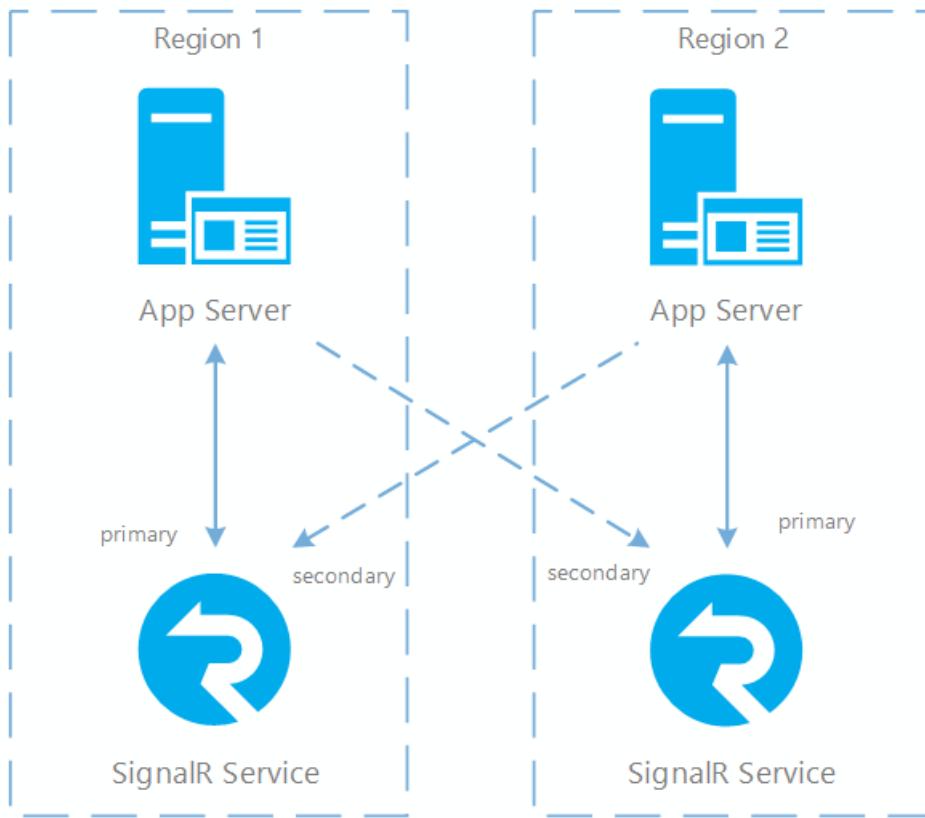
## High available architecture for SignalR service

In order to have cross region resiliency for SignalR service, you need to set up multiple service instances in different regions. So when one region is down, the others can be used as backup. When connecting multiple service instances to app server, there are two roles, primary and secondary. Primary is an instance who is taking online traffic and secondary is a fully functional but backup instance for primary. In our SDK implementation, negotiate will only return primary endpoints so in normal case clients only connect to primary endpoints. But when primary instance is down, negotiate will return secondary endpoints so client can still make connections. Primary instance and app server are connected through normal server connections but secondary instance and app server are connected through a special type of connection called weak connection. The main difference of a weak connection is that it doesn't accept client connection routing, because secondary instance is located in another region. Routing a client to another region is not an optimal choice (increases latency).

One service instance can have different roles when connecting to multiple app servers. One typical setup for cross region scenario is to have two (or more) pairs of SignalR service instances and app servers. Inside each pair app server and SignalR service are located in the same region, and SignalR service is connected to the app server as a primary role. Between each pairs app server and SignalR service are also connected, but SignalR becomes a secondary when connecting to server in another region.

With this topology, message from one server can still be delivered to all clients as all app servers and SignalR service instances are interconnected. But when a client is connected, it's always routed to the app server in the same region to achieve optimal network latency.

Below is a diagram that illustrates such topology:



## Configure multiple SignalR service instances

Multiple SignalR service instances are supported on both app servers and Azure Functions.

Once you have SignalR service and app servers/Azure Functions created in each region, you can configure your app servers/Azure Functions to connect to all SignalR service instances.

### Configure on app servers

There are two ways you can do it:

#### Through config

You should already know how to set SignalR service connection string through environment variables/app settings/web.config, in a config entry named `Azure:SignalR:ConnectionString`. If you have multiple endpoints, you can set them in multiple config entries, each in the following format:

```
Azure:SignalR:ConnectionString:<name>:<role>
```

Here `<name>` is the name of the endpoint and `<role>` is its role (primary or secondary). Name is optional but it will be useful if you want to further customize the routing behavior among multiple endpoints.

#### Through code

If you prefer to store the connection strings somewhere else, you can also read them in your code and use them as parameters when calling `AddAzureSignalR()` (in ASP.NET Core) or `MapAzureSignalR()` (in ASP.NET).

Here is the sample code:

ASP.NET Core:

```

services.AddSignalR()
    .AddAzureSignalR(options => options.Endpoints = new ServiceEndpoint[]
    {
        new ServiceEndpoint("<connection_string1>", EndpointType.Primary, "region1"),
        new ServiceEndpoint("<connection_string2>", EndpointType.Secondary, "region2"),
    });

```

ASP.NET:

```

app.MapAzureSignalR(GetType().FullName, hub, options => options.Endpoints = new ServiceEndpoint[]
{
    new ServiceEndpoint("<connection_string1>", EndpointType.Primary, "region1"),
    new ServiceEndpoint("<connection_string2>", EndpointType.Secondary, "region2"),
});

```

You can configure multiple primary or secondary instances. If there're multiple primary and/or secondary instances, negotiate will return an endpoint in the following order:

1. If there is at least one primary instance online, return a random primary online instance.
2. If all primary instances are down, return a random secondary online instance.

### Configure on Azure Functions

See [this article](#).

## Failover sequence and best practice

Now you have the right system topology setup. Whenever one SignalR service instance is down, online traffic will be routed to other instances. Here is what happens when a primary instance is down (and recovers after some time):

1. Primary service instance is down, all server connections on this instance will be dropped.
2. All servers connected to this instance will mark it as offline, and negotiate will stop returning this endpoint and start returning secondary endpoint.
3. All client connections on this instance will also be closed, clients will reconnect. Since app servers now return secondary endpoint, clients will connect to secondary instance.
4. Now secondary instance takes all online traffic. All messages from server to clients can still be delivered as secondary is connected to all app servers. But client to server messages are only routed to the app server in the same region.
5. After primary instance is recovered and back online, app server will reestablish connections to it and mark it as online. Negotiate will now return primary endpoint again so new clients are connected back to primary. But existing clients won't be dropped and will continue being routed to secondary until they disconnect themselves.

Below diagrams illustrate how failover is done in SignalR service:

Fig.1 Before failover

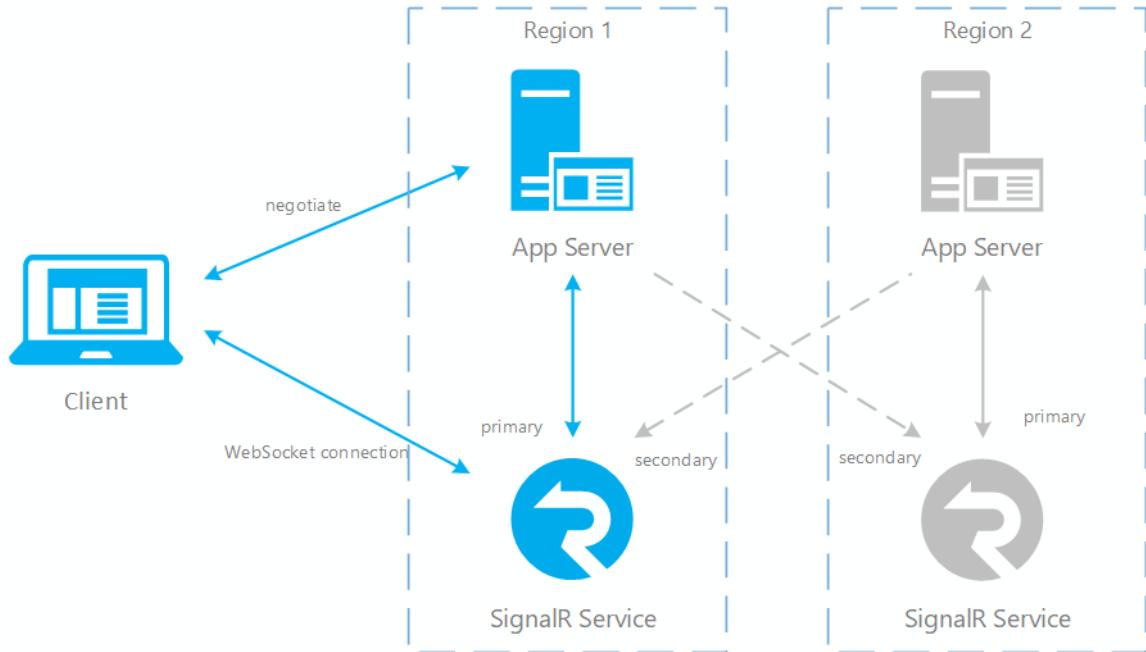


Fig.2 After failover

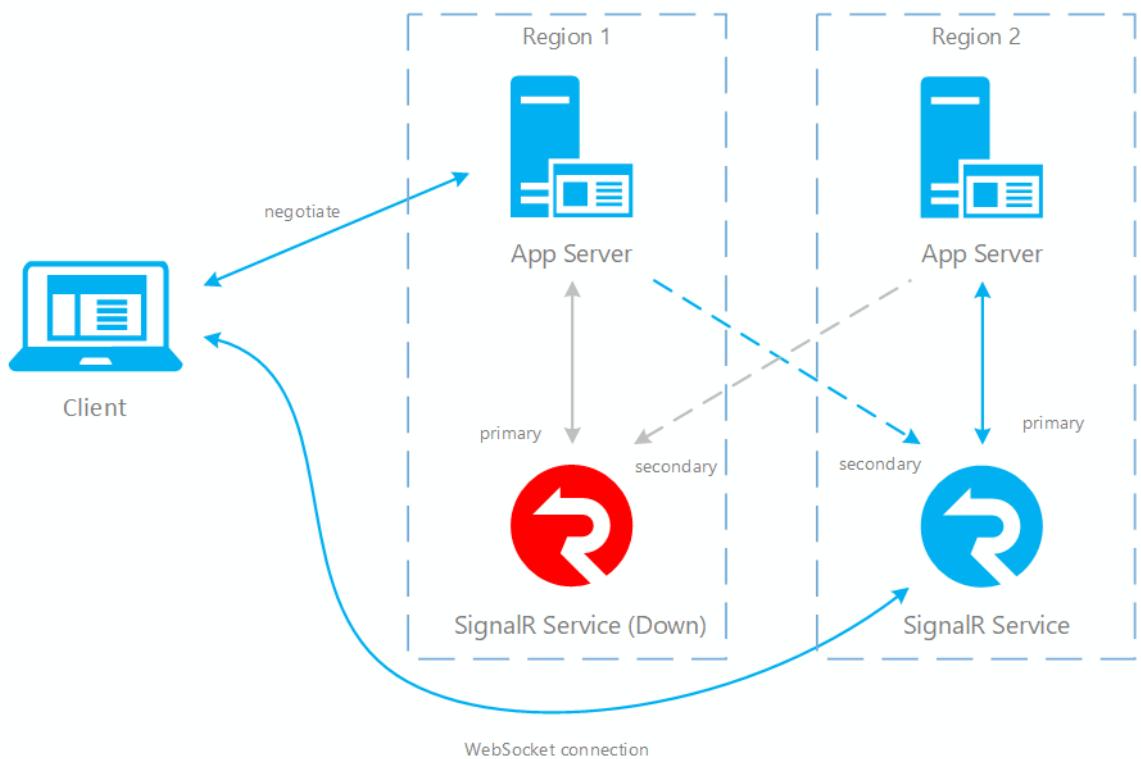
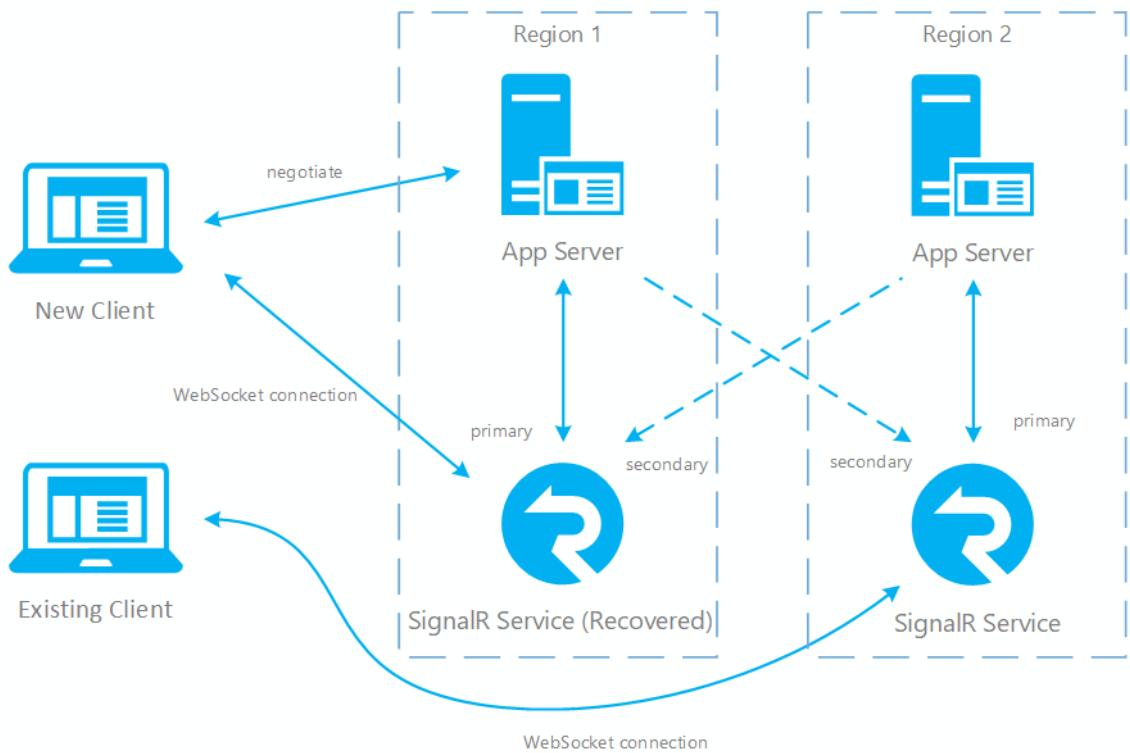


Fig.3 Short time after primary recovers



You can see in normal case only primary app server and SignalR service have online traffic (in blue). After failover, secondary app server and SignalR service also become active. After primary SignalR service is back online, new clients will connect to primary SignalR. But existing clients still connect to secondary so both instances have traffic. After all existing clients disconnect, your system will be back to normal (Fig.1).

There are two main patterns for implementing a cross region high available architecture:

1. The first one is to have a pair of app server and SignalR service instance taking all online traffic, and have another pair as a backup (called active/passive, illustrated in Fig.1).
2. The other one is to have two (or more) pairs of app servers and SignalR service instances, each one taking part of the online traffic and serves as backup for other pairs (called active/active, similar to Fig.3).

SignalR service can support both patterns, the main difference is how you implement app servers. If app servers are active/passive, SignalR service will also be active/passive (as the primary app server only returns its primary SignalR service instance). If app servers are active/active, SignalR service will also be active/active (as all app servers will return their own primary SignalR instances, so all of them can get traffic).

Be noted no matter which patterns you choose to use, you'll need to connect each SignalR service instance to an app server as primary.

Also due to the nature of SignalR connection (it's a long connection), clients will experience connection drops when there is a disaster and failover take place. You'll need to handle such cases at client side to make it transparent to your end customers. For example, do reconnect after a connection is closed.

## Next steps

In this article, you have learned how to configure your application to achieve resiliency for SignalR service. To understand more details about server/client connection and connection routing in SignalR service, you can read [this article](#) for SignalR service internals.

For scaling scenarios such as sharding, that use multiple instances together to handle large number of connections, read [how to scale multiple instances](#).

For details on how to configure Azure Functions with multiple SignalR service instances, read [multiple Azure SignalR Service instances support in Azure Functions](#).

# Messages and connections in Azure SignalR Service

3/17/2021 • 3 minutes to read • [Edit Online](#)

The billing model for Azure SignalR Service is based on the number of connections and the number of messages. This article explains how messages and connections are defined and counted for billing.

## Message formats

Azure SignalR Service supports the same formats as ASP.NET Core SignalR: [JSON](#) and [MessagePack](#).

## Message size

Azure SignalR Service has no size limit for messages.

Large messages are split into smaller messages that are no more than 2 KB each and transmitted separately. SDKs handle message splitting and assembling. No developer efforts are needed.

Large messages do negatively affect messaging performance. Use smaller messages whenever possible, and test to determine the optimal message size for each use-case scenario.

## How messages are counted for billing

For billing, only outbound messages from Azure SignalR Service are counted. Ping messages between clients and servers are ignored.

Messages larger than 2 KB are counted as multiple messages of 2 KB each. The message count chart in the Azure portal is updated every 100 messages per hub.

For example, imagine you have one application server, and three clients:

App server broadcasts a 1-KB message to all connected clients, the message from app server to the service is considered free inbound message. Only the three messages sending from service to each of the client are billed as outbound messages.

Client A sends a 1-KB message to another client B, without going through app server. The message from client A to service is free inbound message. The message from service to client B is billed as outbound message.

If you have three clients and one application server. One client sends a 4-KB message to let the server broadcast to all clients. The billed message count is eight: one message from the service to the application server and three messages from the service to the clients. Each message is counted as two 2-KB messages.

## How connections are counted

There are server connections and client connections with Azure SignalR Service. By default, each application server starts with five initial connections per hub, and each client has one client connection.

For example, assume that you have two application servers and you define five hubs in code. The server connection count will be 50: 2 app servers \* 5 hubs \* 5 connections per hub.

The connection count shown in the Azure portal includes server connections, client connections, diagnostic connections, and live trace connections. The connection types are defined in the following list:

- **Server connection:** Connects Azure SignalR Service and the app server.

- **Client connection:** Connects Azure SignalR Service and the client app.
- **Diagnostic connection:** A special kind of client connection that can produce a more detailed log, which might affect performance. This kind of client is designed for troubleshooting.
- **Live trace connection:** Connects to the live trace endpoint and receives live traces of Azure SignalR Service.

Note that a live trace connection isn't counted as a client connection or as a server connection.

ASP.NET SignalR calculates server connections in a different way. It includes one default hub in addition to hubs that you define. By default, each application server needs five more initial server connections. The initial connection count for the default hub stays consistent with other hubs.

The service and the application server keep syncing connection status and making adjustment to server connections to get better performance and service stability. So you might see server connection number changes from time to time.

## How inbound/outbound traffic is counted

Message sent into the service is inbound message. Message sent out of the service is outbound message. Traffic is calculated in bytes.

## Related resources

- [Aggregation types in Azure Monitor](#)
- [ASP.NET Core SignalR configuration](#)
- [JSON](#)
- [MessagePack](#)

# Performance guide for Azure SignalR Service

11/18/2019 • 19 minutes to read • [Edit Online](#)

One of the key benefits of using Azure SignalR Service is the ease of scaling SignalR applications. In a large-scale scenario, performance is an important factor.

In this guide, we'll introduce the factors that affect SignalR application performance. We'll describe typical performance in different use-case scenarios. In the end, we'll introduce the environment and tools that you can use to generate a performance report.

## Term definitions

*Inbound*: The incoming message to Azure SignalR Service.

*Outbound*: The outgoing message from Azure SignalR Service.

*Bandwidth*: The total size of all messages in 1 second.

*Default mode*: The default working mode when an Azure SignalR Service instance was created. Azure SignalR Service expects the app server to establish a connection with it before it accepts any client connections.

*Serverless mode*: A mode in which Azure SignalR Service accepts only client connections. No server connection is allowed.

## Overview

Azure SignalR Service defines seven Standard tiers for different performance capacities. This guide answers the following questions:

- What is the typical Azure SignalR Service performance for each tier?
- Does Azure SignalR Service meet my requirements for message throughput (for example, sending 100,000 messages per second)?
- For my specific scenario, which tier is suitable for me? Or how can I select the proper tier?
- What kind of app server (VM size) is suitable for me? How many of them should I deploy?

To answer these questions, this guide first gives a high-level explanation of the factors that affect performance. It then illustrates the maximum inbound and outbound messages for every tier for typical use cases: **echo**, **broadcast**, **send to group**, and **send to connection** (peer-to-peer chatting).

This guide can't cover all scenarios (and different use cases, message sizes, message sending patterns, and so on). But it provides some methods to help you:

- Evaluate your approximate requirement for the inbound or outbound messages.
- Find the proper tiers by checking the performance table.

## Performance insight

This section describes the performance evaluation methodologies, and then lists all factors that affect performance. In the end, it provides methods to help you evaluate performance requirements.

### Methodology

*Throughput* and *latency* are two typical aspects of performance checking. For Azure SignalR Service, each SKU tier has its own throughput throttling policy. The policy defines *the maximum allowed throughput (inbound and outbound bandwidth)* as the maximum achieved throughput when 99 percent of messages have latency that's less than 1 second.

Latency is the time span from the connection sending the message to receiving the response message from Azure SignalR Service. Let's take **echo** as an example. Every client connection adds a time stamp in the message. The app server's hub sends the original message back to the client. So the propagation delay is easily calculated by every client connection. The time stamp is attached for every message in **broadcast**, **send to group**, and **send to connection**.

To simulate thousands of concurrent client connections, multiple VMs are created in a virtual private network in Azure. All of these VMs connect to the same Azure SignalR Service instance.

In the default mode of Azure SignalR Service, app server VMs are deployed in the same virtual private network as client VMs. All client VMs and app server VMs are deployed in the same network of the same region to avoid cross-region latency.

## Performance factors

Theoretically, Azure SignalR Service capacity is limited by computation resources: CPU, memory, and network. For example, more connections to Azure SignalR Service cause the service to use more memory. For larger message traffic (for example, every message is larger than 2,048 bytes), Azure SignalR Service needs to spend more CPU cycles to process traffic. Meanwhile, Azure network bandwidth also imposes a limit for maximum traffic.

The transport type is another factor that affects performance. The three types are [WebSocket](#), [Server-Sent-Event](#), and [Long-Polling](#).

WebSocket is a bidirectional and full-duplex communication protocol over a single TCP connection. Server-Sent-Event is a unidirectional protocol to push messages from server to client. Long-Polling requires the clients to periodically poll information from the server through an HTTP request. For the same API under the same conditions, WebSocket has the best performance, Server-Sent-Event is slower, and Long-Polling is the slowest. Azure SignalR Service recommends WebSocket by default.

The message routing cost also limits performance. Azure SignalR Service plays a role as a message router, which routes the message from a set of clients or servers to other clients or servers. A different scenario or API requires a different routing policy.

For **echo**, the client sends a message to itself, and the routing destination is also itself. This pattern has the lowest routing cost. But for **broadcast**, **send to group**, and **send to connection**, Azure SignalR Service needs to look up the target connections through the internal distributed data structure. This extra processing uses more CPU, memory, and network bandwidth. As a result, performance is slower.

In the default mode, the app server might also become a bottleneck for certain scenarios. The Azure SignalR SDK has to invoke the hub, while it maintains a live connection with every client through heartbeat signals.

In serverless mode, the client sends a message by HTTP post, which is not as efficient as WebSocket.

Another factor is protocol: JSON and [MessagePack](#). MessagePack is smaller in size and delivered faster than JSON. MessagePack might not improve performance, though. The performance of Azure SignalR Service is not sensitive to protocols because it doesn't decode the message payload during message forwarding from clients to servers or vice versa.

In summary, the following factors affect the inbound and outbound capacity:

- SKU tier (CPU/memory)
- Number of connections

- Message size
- Message send rate
- Transport type (WebSocket, Server-Sent-Event, or Long-Polling)
- Use-case scenario (routing cost)
- App server and service connections (in server mode)

## Finding a proper SKU

How can you evaluate the inbound/outbound capacity or find which tier is suitable for a specific use case?

Assume that the app server is powerful enough and is not the performance bottleneck. Then, check the maximum inbound and outbound bandwidth for every tier.

### Quick evaluation

Let's simplify the evaluation first by assuming some default settings:

- The transport type is WebSocket.
- The message size is 2,048 bytes.
- A message is sent every 1 second.
- Azure SignalR Service is in the default mode.

Every tier has its own maximum inbound bandwidth and outbound bandwidth. A smooth user experience is not guaranteed after the inbound or outbound connection exceeds the limit.

**Echo** gives the maximum inbound bandwidth because it has the lowest routing cost. **Broadcast** defines the maximum outbound message bandwidth.

Do *not* exceed the highlighted values in the following two tables.

ECHO	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
Inbound bandwidth	2 MBps	4 MBps	10 MBps	20 MBps	40 MBps	100 MBps	200 MBps
Outbound bandwidth	2 MBps	4 MBps	10 MBps	20 MBps	40 MBps	100 MBps	200 MBps

BROADCAST	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
Inbound bandwidth	4 KBps	4 KBps	4 KBps	4 KBps	4 KBps	4 KBps	4 KBps
Outbound bandwidth	4 MBps	8 MBps	20 MBps	40 MBps	80 MBps	200 MBps	400 MBps

*Inbound bandwidth* and *outbound bandwidth* are the total message size per second. Here are the formulas for

them:

```
inboundBandwidth = inboundConnections * messageSize / sendInterval  
outboundBandwidth = outboundConnections * messageSize / sendInterval
```

- *inboundConnections*: The number of connections sending the message.
- *outboundConnections*: The number of connections receiving the message.
- *messageSize*: The size of a single message (average value). A small message that's less than 1,024 bytes has a performance impact that's similar to a 1,024-byte message.
- *sendInterval*: The time of sending one message. Typically it's 1 second per message, which means sending one message every second. A smaller interval means sending more message in a time period. For example, 0.5 seconds per message means sending two messages every second.
- *Connections*: The committed maximum threshold for Azure SignalR Service for every tier. If the connection number is increased further, it will suffer from connection throttling.

#### Evaluation for complex use cases

##### Bigger message size or different sending rate

The real use case is more complicated. It might send a message larger than 2,048 bytes, or the sending message rate is not one message per second. Let's take Unit100's broadcast as an example to find how to evaluate its performance.

The following table shows a real use case of **broadcast**. But the message size, connection count, and message sending rate are different from what we assumed in the previous section. The question is how we can deduce any of those items (message size, connection count, or message sending rate) if we know only two of them.

BROADCAST	MESSAGE SIZE	INBOUND MESSAGES PER SECOND	CONNECTIONS	SEND INTERVALS
1	20 KB	1	100,000	5 sec
2	256 KB	1	8,000	5 sec

The following formula is easy to infer based on the previous formula:

```
outboundConnections = outboundBandwidth * sendInterval / messageSize
```

For Unit100, the maximum outbound bandwidth is 400 MB from the previous table. For a 20-KB message size, the maximum outbound connections should be  $400 \text{ MB} * 5 / 20 \text{ KB} = 100,000$ , which matches the real value.

##### Mixed use cases

The real use case typically mixes the four basic use cases together: **echo**, **broadcast**, **send to group**, and **send to connection**. The methodology that you use to evaluate the capacity is to:

1. Divide the mixed use cases into four basic use cases.
2. Calculate the maximum inbound and outbound message bandwidth by using the preceding formulas separately.
3. Sum the bandwidth calculations to get the total maximum inbound/outbound bandwidth.

Then pick up the proper tier from the maximum inbound/outbound bandwidth tables.

#### **NOTE**

For sending a message to hundreds or thousands of small groups, or for thousands of clients sending a message to each other, the routing cost will become dominant. Take this impact into account.

For the use case of sending a message to clients, make sure that the app server is *not* the bottleneck. The following "Case study" section gives guidelines about how many app servers you need and how many server connections you should configure.

## Case study

The following sections go through four typical use cases for WebSocket transport: **echo**, **broadcast**, **send to group**, and **send to connection**. For each scenario, the section lists the current inbound and outbound capacity for Azure SignalR Service. It also explains the main factors that affect performance.

In the default mode, the app server creates five server connections with Azure SignalR Service. The app server uses the Azure SignalR Service SDK by default. In the following performance test results, server connections are increased to 15 (or more for broadcasting and sending a message to a big group).

Different use cases have different requirements for app servers. **Broadcast** needs small number of app servers. **Echo** or **send to connection** needs many app servers.

In all use cases, the default message size is 2,048 bytes, and the message send interval is 1 second.

### **Default mode**

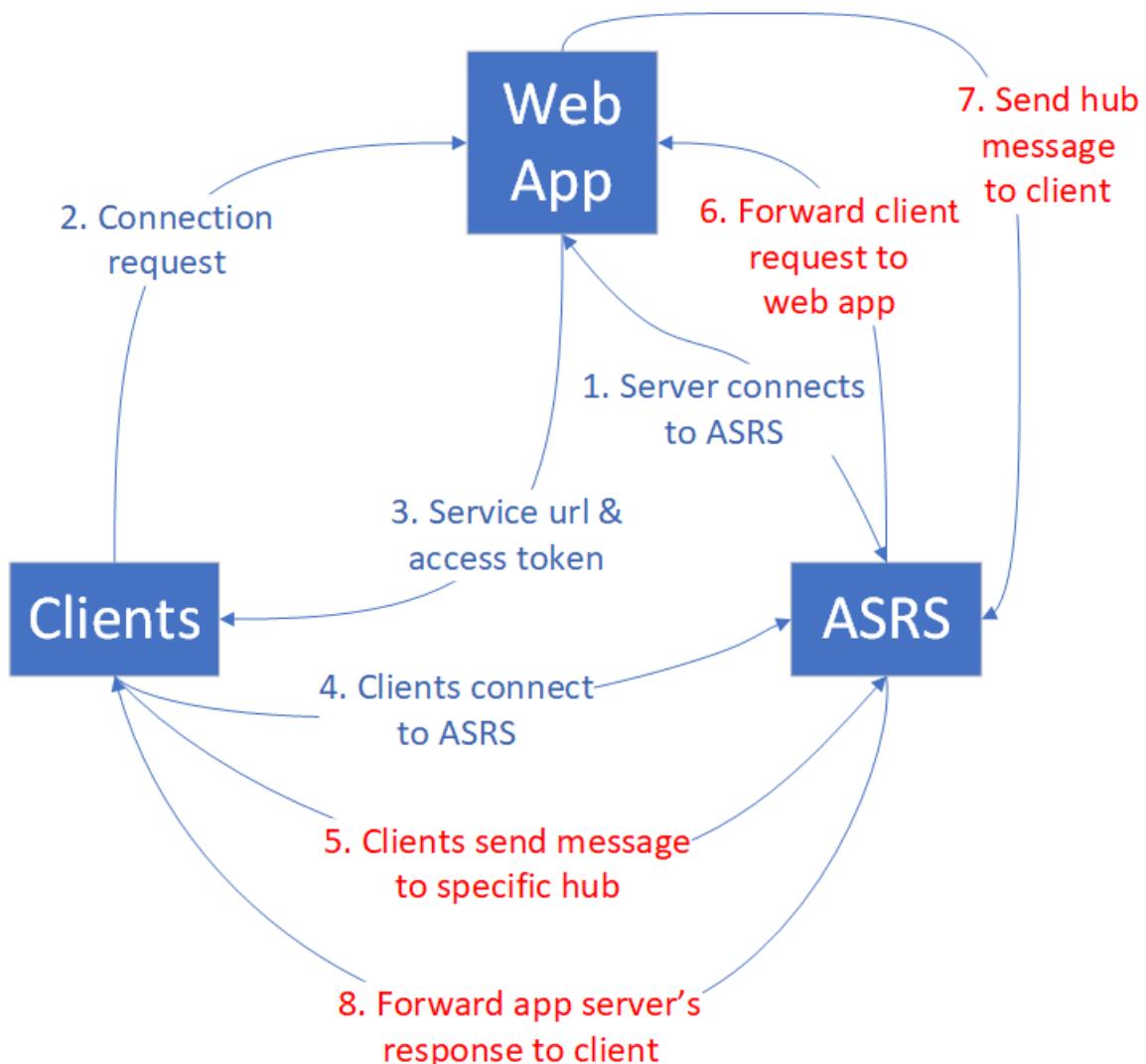
Clients, web app servers, and Azure SignalR Service are involved in the default mode. Every client stands for a single connection.

#### **Echo**

First, a web app connects to Azure SignalR Service. Second, many clients connect to the web app, which redirects the clients to Azure SignalR Service with the access token and endpoint. Then, the clients establish WebSocket connections with Azure SignalR Service.

After all clients establish connections, they start sending a message that contains a time stamp to the specific hub every second. The hub echoes the message back to its original client. Every client calculates the latency when it receives the echo message back.

In the following diagram, 5 through 8 (red highlighted traffic) are in a loop. The loop runs for a default duration (5 minutes) and gets the statistic of all message latency.



The behavior of **echo** determines that the maximum inbound bandwidth is equal to the maximum outbound bandwidth. For details, see the following table.

ECHO	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
Inbound/outbound messages per second	1,000	2,000	5,000	10,000	20,000	50,000	100,000
Inbound/outbound bandwidth	2 MBps	4 MBps	10 MBps	20 MBps	40 MBps	100 MBps	200 MBps

In this use case, every client invokes the hub defined in the app server. The hub just calls the method defined in the original client side. This hub is the most lightweight hub for **echo**.

```

public void Echo(IDictionary<string, object> data)
{
    Clients.Client(Context.ConnectionId).SendAsync("RecordLatency", data);
}
    
```

Even for this simple hub, the traffic pressure on the app server is prominent as the **echo** inbound message load

increases. This traffic pressure requires many app servers for large SKU tiers. The following table lists the app server count for every tier.

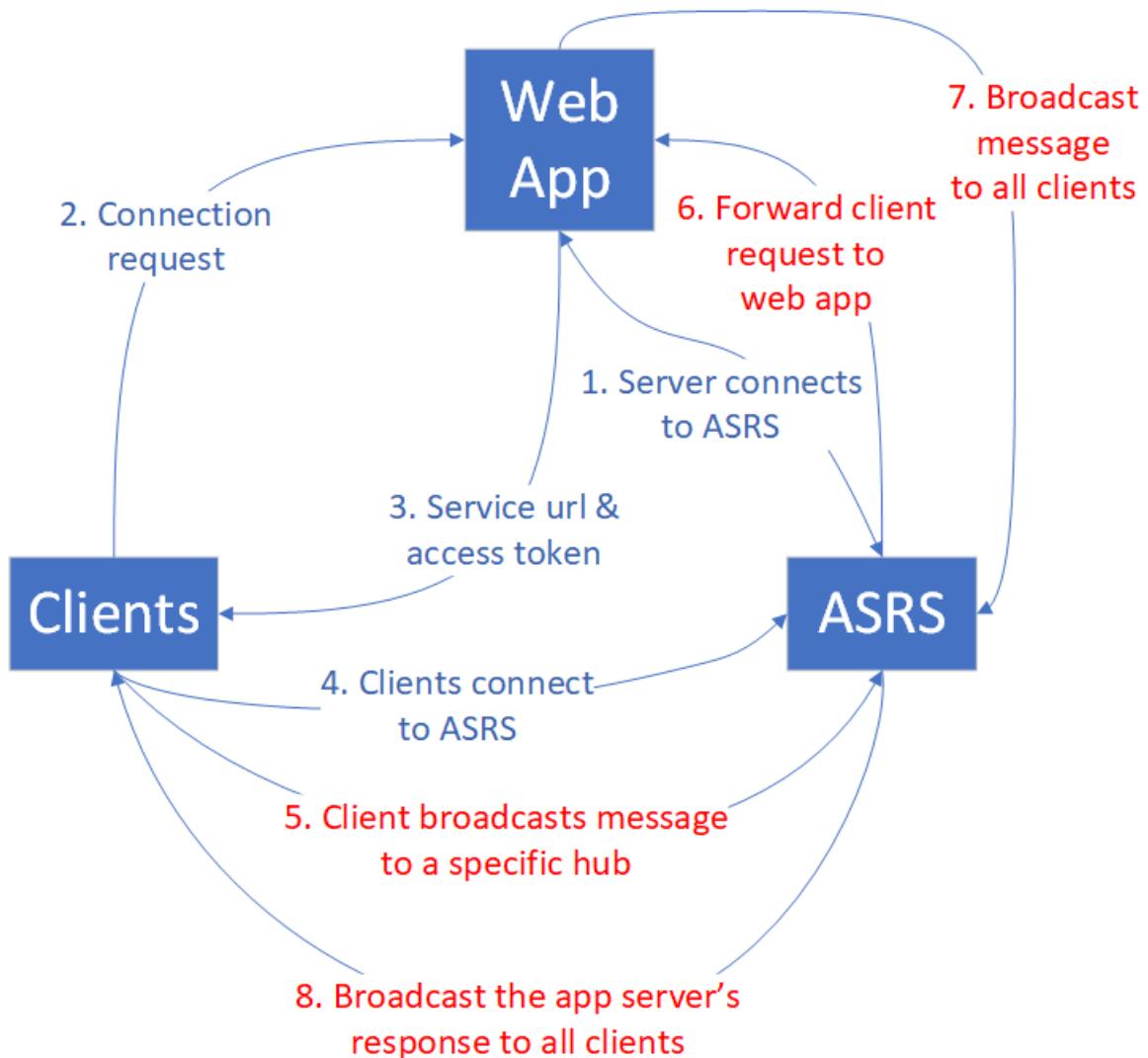
ECHO	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
App server count	2	2	2	3	3	10	20

#### NOTE

The client connection number, message size, message sending rate, SKU tier, and CPU/memory of the app server affect the overall performance of echo.

#### Broadcast

For broadcast, when the web app receives the message, it broadcasts to all clients. The more clients there are to broadcast, the more message traffic there is to all clients. See the following diagram.



A small number of clients are broadcasting. The inbound message bandwidth is small, but the outbound bandwidth is huge. The outbound message bandwidth increases as the client connection or broadcast rate increases.

The following table summarizes maximum client connections, inbound/outbound message count, and

bandwidth.

BROADCAST	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
Inbound messages per second	2	2	2	2	2	2	2
Outbound messages per second	2,000	4,000	10,000	20,000	40,000	100,000	200,000
Inbound bandwidth	4 KBps	4 KBps	4 KBps	4 KBps	4 KBps	4 KBps	4 KBps
Outbound bandwidth	4 MBps	8 MBps	20 MBps	40 MBps	80 MBps	200 MBps	400 MBps

The broadcasting clients that post messages are no more than four. They need fewer app servers compared with **echo** because the inbound message amount is small. Two app servers are enough for both SLA and performance considerations. But you should increase the default server connections to avoid imbalance, especially for Unit50 and Unit100.

BROADCAST	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
App server count	2	2	2	2	2	2	2

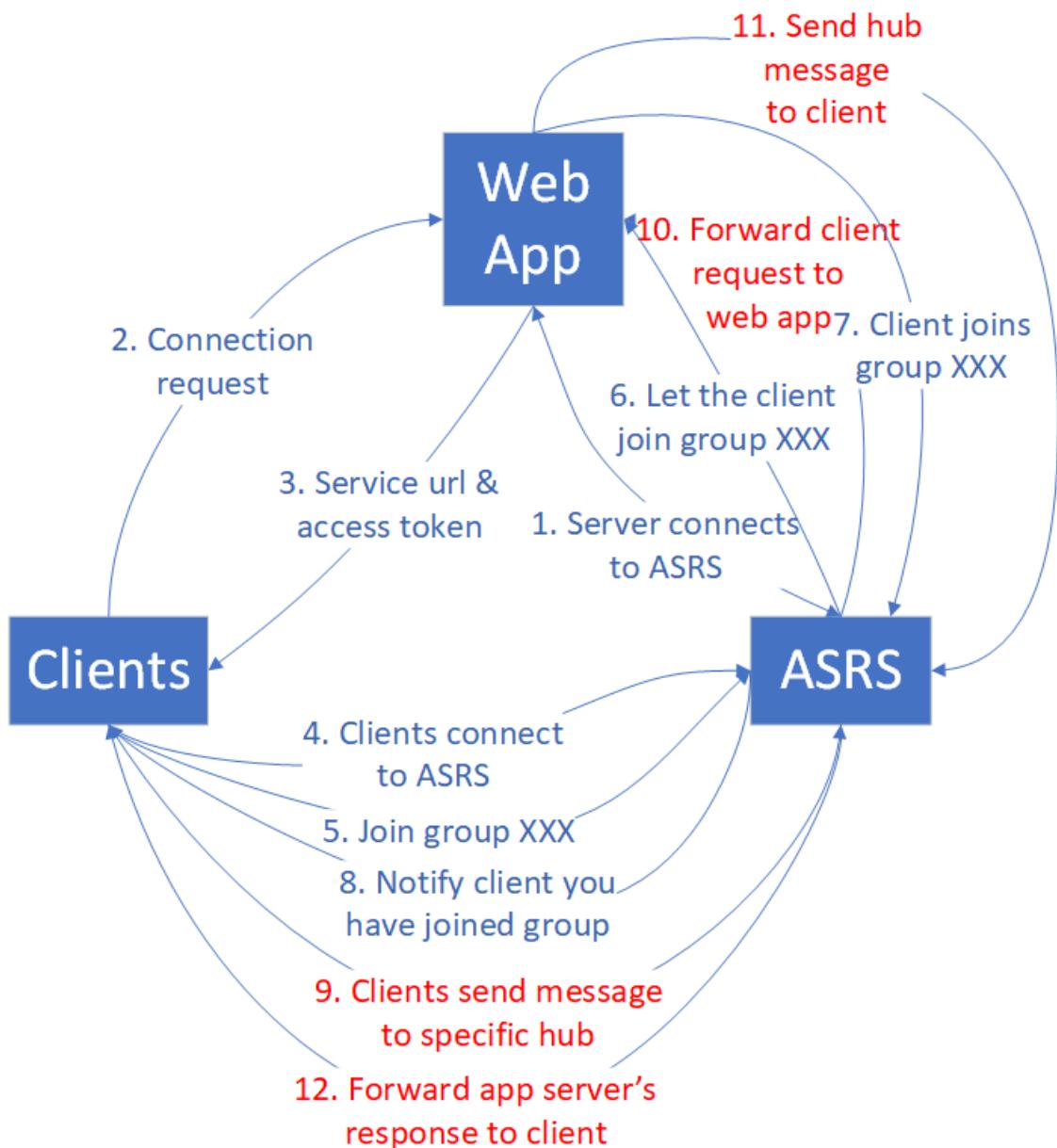
#### NOTE

Increase the default server connections from 5 to 40 on every app server to avoid possible unbalanced server connections to Azure SignalR Service.

The client connection number, message size, message sending rate, and SKU tier affect the overall performance for **broadcast**.

#### Send to group

The **send to group** use case has a similar traffic pattern to **broadcast**. The difference is that after clients establish WebSocket connections with Azure SignalR Service, they must join groups before they can send a message to a specific group. The following diagram illustrates the traffic flow.



Group member and group count are two factors that affect performance. To simplify the analysis, we define two kinds of groups:

- **Small group:** Every group has 10 connections. The group number is equal to  $(\text{max connection count}) / 10$ . For example, for Unit1, if there are 1,000 connection counts, then we have  $1000 / 10 = 100$  groups.
- **Big group:** The group number is always 10. The group member count is equal to  $(\text{max connection count}) / 10$ . For example, for Unit1, if there are 1,000 connection counts, then every group has  $1000 / 10 = 100$  members.

**Send to group** brings a routing cost to Azure SignalR Service because it has to find the target connections through a distributed data structure. As the sending connections increase, the cost increases.

#### Small group

The routing cost is significant for sending message to many small groups. Currently, the Azure SignalR Service implementation hits the routing cost limit at Unit50. Adding more CPU and memory doesn't help, so Unit100 can't improve further by design. If you need more inbound bandwidth, contact customer support.

SEND TO SMALL GROUP	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
Group member count	10	10	10	10	10	10	10
Group count	100	200	500	1,000	2,000	5,000	10,000
Inbound messages per second	200	400	1,000	2,500	4,000	7,000	7,000
Inbound bandwidth	400 KBps	800 KBps	2 MBps	5 MBps	8 MBps	14 MBps	14 MBps
Outbound messages per second	2,000	4,000	10,000	25,000	40,000	70,000	70,000
Outbound bandwidth	4 MBps	8 MBps	20 MBps	50 MBps	80 MBps	140 MBps	140 MBps

Many client connections are calling the hub, so the app server number is also critical for performance. The following table lists the suggested app server counts.

SEND TO SMALL GROUP	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
App server count	2	2	2	3	3	10	20

#### NOTE

The client connection number, message size, message sending rate, routing cost, SKU tier, and CPU/memory of the app server affect the overall performance of **send to small group**.

#### Big group

For **send to big group**, the outbound bandwidth becomes the bottleneck before hitting the routing cost limit. The following table lists the maximum outbound bandwidth, which is almost the same as that for **broadcast**.

SEND TO BIG GROUP	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000

SEND TO BIG GROUP	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Group member count	100	200	500	1,000	2,000	5,000	10,000
Group count	10	10	10	10	10	10	10
Inbound messages per second	20	20	20	20	20	20	20
Inbound bandwidth	80 KBps	40 KBps	40 KBps	20 KBps	40 KBps	40 KBps	40 KBps
Outbound messages per second	2,000	4,000	10,000	20,000	40,000	100,000	200,000
Outbound bandwidth	8 MBps	8 MBps	20 MBps	40 MBps	80 MBps	200 MBps	400 MBps

The sending connection count is no more than 40. The burden on the app server is small, so the suggested number of web apps is small.

SEND TO BIG GROUP	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
App server count	2	2	2	2	2	2	2

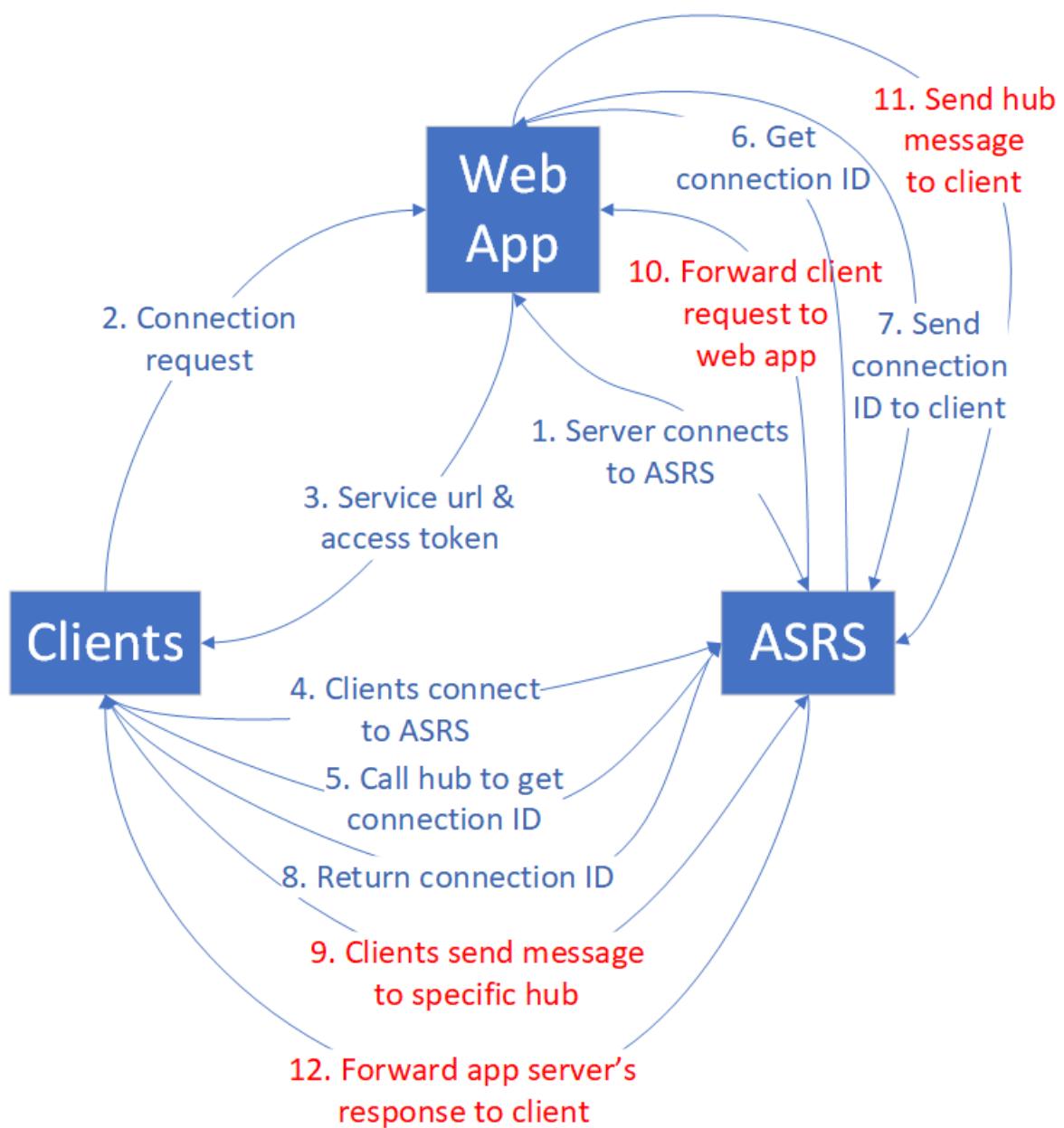
#### NOTE

Increase the default server connections from 5 to 40 on every app server to avoid possible unbalanced server connections to Azure SignalR Service.

The client connection number, message size, message sending rate, routing cost, and SKU tier affect the overall performance of **send to big group**.

#### Send to connection

In the **send to connection** use case, when clients establish the connections to Azure SignalR Service, every client calls a special hub to get their own connection ID. The performance benchmark collects all connection IDs, shuffles them, and reassigns them to all clients as a sending target. The clients keep sending the message to the target connection until the performance test finishes.



The routing cost for **send to connection** is similar to the cost for **send to small group**.

As the connection count increases, the routing cost limits overall performance. Unit50 has reached the limit. As a result, Unit100 can't improve further.

The following table is a statistical summary after many rounds of running the **send to connection** benchmark.

SEND TO CONNECTION	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
Inbound/outbound messages per second	1,000	2,000	5,000	8,000	9,000	20,000	20,000
Inbound/outbound bandwidth	2 MBps	4 MBps	10 MBps	16 MBps	18 MBps	40 MBps	40 MBps

This use case requires high load on the app server side. See the suggested app server count in the following table.

SEND TO CONNECTION	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
App server count	2	2	2	3	3	10	20

#### NOTE

The client connection number, message size, message sending rate, routing cost, SKU tier, and CPU/memory for the app server affect the overall performance of **send to connection**.

#### ASP.NET SignalR echo, broadcast, and send to small group

Azure SignalR Service provides the same performance capacity for ASP.NET SignalR.

The performance test uses Azure Web Apps from [Standard Service Plan S3](#) for ASP.NET SignalR.

The following table gives the suggested web app count for ASP.NET SignalR echo.

ECHO	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
App server count	2	2	4	4	8	32	40

The following table gives the suggested web app count for ASP.NET SignalR broadcast.

BROADCAST	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
App server count	2	2	2	2	2	2	2

The following table gives the suggested web app count for ASP.NET SignalR send to small group.

SEND TO SMALL GROUP	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
App server count	2	2	4	4	8	32	40

#### Serverless mode

Clients and Azure SignalR Service are involved in serverless mode. Every client stands for a single connection. The client sends messages through the REST API to another client or broadcast messages to all.

Sending high-density messages through the REST API is not as efficient as using WebSocket. It requires you to build a new HTTP connection every time, and that's an extra cost in serverless mode.

#### Broadcast through REST API

All clients establish WebSocket connections with Azure SignalR Service. Then some clients start broadcasting through the REST API. The message sending (inbound) is all through HTTP Post, which is not efficient compared with WebSocket.

BROADCAST THROUGH REST API	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
Inbound messages per second	2	2	2	2	2	2	2
Outbound messages per second	2,000	4,000	10,000	20,000	40,000	100,000	200,000
Inbound bandwidth	4 KBps	4 KBps	4 KBps	4 KBps	4 KBps	4 KBps	4 KBps
Outbound bandwidth	4 MBps	8 MBps	20 MBps	40 MBps	80 MBps	200 MBps	400 MBps

#### Send to user through REST API

The benchmark assigns usernames to all of the clients before they start connecting to Azure SignalR Service. After the clients establish WebSocket connections with Azure SignalR Service, they start sending messages to others through HTTP Post.

SEND TO USER THROUGH REST API	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Connections	1,000	2,000	5,000	10,000	20,000	50,000	100,000
Inbound messages per second	300	600	900	1,300	2,000	10,000	18,000
Outbound messages per second	300	600	900	1,300	2,000	10,000	18,000
Inbound bandwidth	600 KBps	1.2 MBps	1.8 MBps	2.6 MBps	4 MBps	10 MBps	36 MBps

SEND TO USER THROUGH REST API	UNIT1	UNIT2	UNIT5	UNIT10	UNIT20	UNIT50	UNIT100
Outbound bandwidth	600 KBps	1.2 MBps	1.8 MBps	2.6 MBps	4 MBps	10 MBps	36 MBps

## Performance test environments

For all use cases listed earlier, we conducted the performance tests in an Azure environment. At most, we used 50 client VMs and 20 app server VMs. Here are some details:

- Client VM size: StandardDS2V2 (2 vCPU, 7G memory)
- App server VM size: StandardF4sV2 (4 vCPU, 8G memory)
- Azure SignalR SDK server connections: 15

## Performance tools

You can find performance tools for Azure SignalR Service on [GitHub](#).

## Next steps

In this article, you got an overview of Azure SignalR Service performance in typical use-case scenarios.

To get details on the internals of the service and scaling for it, read the following guides:

- [Azure SignalR Service internals](#)
- [Azure SignalR Service scaling](#)

# Azure SignalR Service authentication

4/22/2021 • 14 minutes to read • [Edit Online](#)

This tutorial builds on the chat room application introduced in the quickstart. If you have not completed [Create a chat room with SignalR Service](#), complete that exercise first.

In this tutorial, you'll learn how to implement your own authentication and integrate it with the Microsoft Azure SignalR Service.

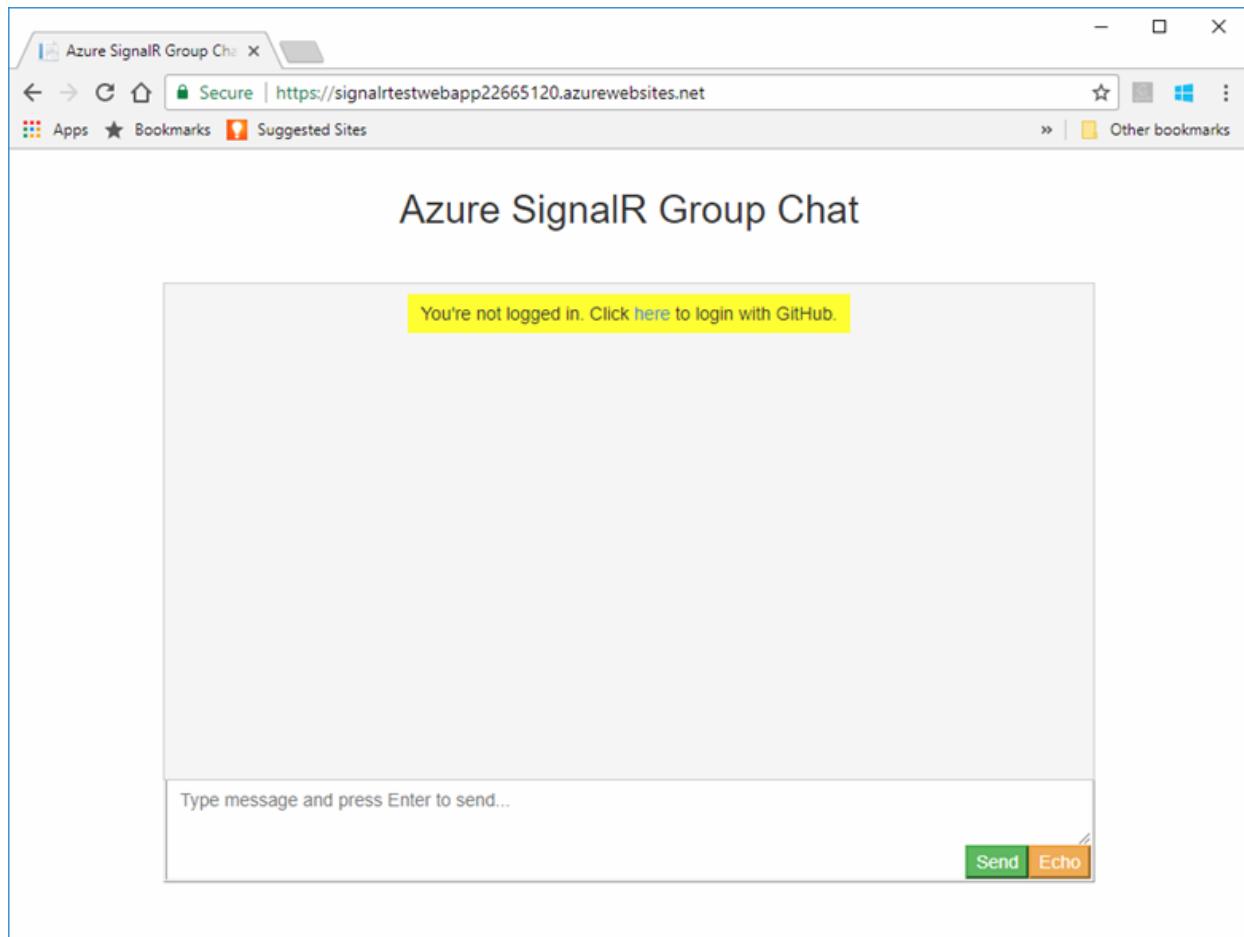
The authentication initially used in the quickstart's chat room application is too simple for real-world scenarios. The application allows each client to claim who they are, and the server simply accepts that. This approach is not very useful in real-world applications where a rogue user would impersonate others to access sensitive data.

[GitHub](#) provides authentication APIs based on a popular industry-standard protocol called [OAuth](#). These APIs allow third-party applications to authenticate GitHub accounts. In this tutorial, you will use these APIs to implement authentication through a GitHub account before allowing client logins to the chat room application. After authenticating a GitHub account, the account information will be added as a cookie to be used by the web client to authenticate.

For more information on the OAuth authentication APIs provided through GitHub, see [Basics of Authentication](#).

You can use any code editor to complete the steps in this quickstart. However, [Visual Studio Code](#) is an excellent option available on the Windows, macOS, and Linux platforms.

The code for this tutorial is available for download in the [AzureSignalR-samples GitHub repository](#).



In this tutorial, you learn how to:

- Register a new OAuth app with your GitHub account
- Add an authentication controller to support GitHub authentication
- Deploy your ASP.NET Core web app to Azure

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

## Prerequisites

To complete this tutorial, you must have the following prerequisites:

- An account created on [GitHub](#)
- [Git](#)
- [.NET Core SDK](#)
- [Azure Cloud Shell](#) configured for the bash environment.
- Download or clone the [AzureSignalR-sample](#) GitHub repository.

## Create an OAuth app

1. Open a web browser and navigate to <https://github.com> and sign into your account.
2. For your account, navigate to **Settings > Developer settings** and click **Register a new application**, or **New OAuth App** under *OAuth Apps*.
3. Use the following settings for the new OAuth App, then click **Register application**:

SETTING NAME	SUGGESTED VALUE	DESCRIPTION
Application name	<i>Azure SignalR Chat</i>	The GitHub user should be able to recognize and trust the app they are authenticating with.
Homepage URL	<a href="http://localhost:5000/home">http://localhost:5000/home</a>	
Application description	<i>A chat room sample using the Azure SignalR Service with GitHub authentication</i>	A useful description of the application that will help your application users understand the context of the authentication being used.
Authorization callback URL	<a href="http://localhost:5000/signin-github">http://localhost:5000/signin-github</a>	This setting is the most important setting for your OAuth application. It's the callback URL that GitHub returns the user to after successful authentication. In this tutorial, you must use the default callback URL for the <i>AspNet.Security.OAuth.GitHub</i> package, <i>/signin-github</i> .

4. Once the new OAuth app registration is complete, add the *Client ID* and *Client Secret* to Secret Manager using the following commands. Replace *Your\_GitHub\_Client\_Id* and *Your\_GitHub\_Client\_Secret* with the values for your OAuth app.

```
dotnet user-secrets set GitHubClientId Your_GitHub_Client_Id
dotnet user-secrets set GitHubClientSecret Your_GitHub_Client_Secret
```

# Implement the OAuth flow

## Update the Startup class to support GitHub authentication

1. Add a reference to the latest *Microsoft.AspNetCore.Authentication.Cookies* and *AspNet.Security.OAuth.GitHub* packages and restore all packages.

```
dotnet add package Microsoft.AspNetCore.Authentication.Cookies -v 2.1.0-rc1-30656
dotnet add package AspNet.Security.OAuth.GitHub -v 2.0.0-rc2-final
dotnet restore
```

2. Open *Startup.cs*, and add `using` statements for the following namespaces:

```
using System.Net.Http;
using System.Net.Http.Headers;
using System.Security.Claims;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OAuth;
using Newtonsoft.Json.Linq;
```

3. At the top of the `Startup` class, add constants for the Secret Manager keys that hold the GitHub OAuth app secrets.

```
private const string GitHubClientId = "GitHubClientId";
private const string GitHubClientSecret = "GitHubClientSecret";
```

4. Add the following code to the `ConfigureServices` method to support authentication with the GitHub OAuth app:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie()
    .AddGitHub(options =>
{
    options.ClientId = Configuration[GitHubClientId];
    options.ClientSecret = Configuration[GitHubClientSecret];
    options.Scope.Add("user:email");
    options.Events = new OAuthEvents
    {
        OnCreatingTicket = GetUserCompanyInfoAsync
    };
});
```

5. Add the  `GetUserCompanyInfoAsync` helper method to the `Startup` class.

```

private static async Task GetUserCompanyInfoAsync(OAuthCreatingTicketContext context)
{
    var request = new HttpRequestMessage(HttpMethod.Get, context.Options.UserInformationEndpoint);
    request.Headers.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));
    request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", context.AccessToken);

    var response = await context.Backchannel.SendAsync(request,
        HttpCompletionOption.ResponseHeadersRead, context.HttpContext.RequestAborted);

    var user = JObject.Parse(await response.Content.ReadAsStringAsync());
    if (user.ContainsKey("company"))
    {
        var company = user["company"].ToString();
        var companyIdentity = new ClaimsIdentity(new[]
        {
            new Claim("Company", company)
        });
        context.Principal.AddIdentity(companyIdentity);
    }
}

```

6. Update the `Configure` method of the Startup class with the following line of code, and save the file.

```
app.UseAuthentication();
```

## Add an authentication controller

In this section, you will implement a `Login` API that authenticates clients using the GitHub OAuth app. Once authenticated, the API will add a cookie to the web client response before redirecting the client back to the chat app. That cookie will then be used to identify the client.

1. Add a new controller code file to the `chattest\Controllers` directory. Name the file `AuthController.cs`.
2. Add the following code for the authentication controller. Make sure to update the namespace, if your project directory was not `chattest`

```

using AspNet.Security.OAuth.GitHub;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Mvc;

namespace chattest.Controllers
{
    [Route("/")]
    public class AuthController : Controller
    {
        [HttpGet("login")]
        public IActionResult Login()
        {
            if (!User.Identity.IsAuthenticated)
            {
                return Challenge(GitHubAuthenticationDefaults.AuthenticationScheme);
            }

            HttpContext.Response.Cookies.Append("githubchat_username", User.Identity.Name);
            HttpContext.SignInAsync(User);
            return Redirect("/");
        }
    }
}

```

3. Save your changes.

## Update the Hub class

By default when a web client attempts to connect to SignalR Service, the connection is granted based on an access token that is provided internally. This access token is not associated with an authenticated identity. This access is actually anonymous access.

In this section, you will turn on real authentication by adding the `[Authorize]` attribute to the hub class, and updating the hub methods to read the username from the authenticated user's claim.

1. Open `Hub\Chat.cs` and add references to these namespaces:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
```

2. Update the hub code as shown below. This code adds the `[Authorize]` attribute to the `Chat` class, and uses the user's authenticated identity in the hub methods. Also, the `OnConnectedAsync` method is added, which will log a system message to the chat room each time a new client connects.

```
[Authorize]
public class Chat : Hub
{
    public override Task OnConnectedAsync()
    {
        return Clients.All.SendAsync("broadcastMessage", "_SYSTEM_", $"{Context.User.Identity.Name} JOINED");
    }

    // Uncomment this line to only allow user in Microsoft to send message
    // [Authorize(Policy = "Microsoft_Only")]
    public void BroadcastMessage(string message)
    {
        Clients.All.SendAsync("broadcastMessage", Context.User.Identity.Name, message);
    }

    public void Echo(string message)
    {
        var echoMessage = $"{message} (echo from server)";
        Clients.Client(Context.ConnectionId).SendAsync("echo", Context.User.Identity.Name,
echoMessage);
    }
}
```

3. Save your changes.

## Update the web client code

1. Open `wwwroot\index.html` and replace the code that prompts for the username with code to use the cookie returned by the authentication controller.

Remove the following code from `index.html`:

```
// Get the user name and store it to prepend to messages.  
var username = generateRandomName();  
var promptMessage = 'Enter your name:';  
do {  
    username = prompt(promptMessage, username);  
    if (!username || username.startsWith('_') || username.indexOf('<') > -1 || username.indexOf('>')  
        > -1) {  
        username = '';  
        promptMessage = 'Invalid input. Enter your name:';  
    }  
} while(!username)
```

Add the following code in place of the code above to use the cookie:

```
// Get the user name cookie.  
function getCookie(key) {  
    var cookies = document.cookie.split(';').map(c => c.trim());  
    for (var i = 0; i < cookies.length; i++) {  
        if (cookies[i].startsWith(key + '=')) return unescape(cookies[i].slice(key.length + 1));  
    }  
    return '';  
}  
var username = getCookie('githubchat_username');
```

- Just beneath the line of code you added to use the cookie, add the following definition for the `appendMessage` function:

```
function appendMessage(encodedName, encodedMsg) {  
    var messageEntry = createMessageEntry(encodedName, encodedMsg);  
    var messageBox = document.getElementById('messages');  
    messageBox.appendChild(messageEntry);  
    messageBox.scrollTop = messageBox.scrollHeight;  
}
```

- Update the `bindConnectionMessage` and `onConnected` functions with the following code to use `appendMessage`.

```

function bindConnectionMessage(connection) {
    var messageCallback = function(name, message) {
        if (!message) return;
        // Html encode display name and message.
        var encodedName = name;
        var encodedMsg = message.replace(/&/g, "&").replace(/</g, "<").replace(/>/g, ">");
        appendMessage(encodedName, encodedMsg);
    };
    // Create a function that the hub can call to broadcast messages.
    connection.on('broadcastMessage', messageCallback);
    connection.on('echo', messageCallback);
    connection.onclose(onConnectionError);
}

function onConnected(connection) {
    console.log('connection started');
    document.getElementById('sendmessage').addEventListener('click', function (event) {
        // Call the broadcastMessage method on the hub.
        if (messageInput.value) {
            connection
                .invoke('broadcastMessage', messageInput.value)
                .catch(e => appendMessage('_BROADCAST_', e.message));
        }

        // Clear text box and reset focus for next comment.
        messageInput.value = '';
        messageInput.focus();
        event.preventDefault();
    });
    document.getElementById('message').addEventListener('keypress', function (event) {
        if (event.keyCode === 13) {
            event.preventDefault();
            document.getElementById('sendmessage').click();
            return false;
        }
    });
    document.getElementById('echo').addEventListener('click', function (event) {
        // Call the echo method on the hub.
        connection.send('echo', messageInput.value);

        // Clear text box and reset focus for next comment.
        messageInput.value = '';
        messageInput.focus();
        event.preventDefault();
    });
}

```

- At the bottom of *index.html*, update the error handler for `connection.start()` as shown below to prompt the user to log in.

```
connection.start()
    .then(function () {
        onConnected(connection);
    })
    .catch(function (error) {
        if (error) {
            if (error.message) {
                console.error(error.message);
            }
            if (error.statusCode && error.statusCode === 401) {
                appendMessage('_BROADCAST_', 'You\'re not logged in. Click <a href="/login">here</a> to login with GitHub.');
            }
        }
    });
});
```

5. Save your changes.

## Build and Run the app locally

1. Save changes to all files.
2. Build the app using the .NET Core CLI, execute the following command in the command shell:

```
dotnet build
```

3. Once the build successfully completes, execute the following command to run the web app locally:

```
dotnet run
```

By default, the app will be hosted locally on port 5000:

```
E:\Testing\chattest>dotnet run
Hosting environment: Production
Content root path: E:\Testing\chattest
Now listening on: http://localhost:5000
      Application started. Press Ctrl+C to shut down.
```

4. Launch a browser window and navigate to `http://localhost:5000`. Click the **here** link at the top to log in with GitHub.

Azure SignalR Group Chat

You're not logged in. Click [here](#) to login with GitHub.

Type message and press Enter to send...

Send Echo

You will be prompted to authorize the chat app's access to your GitHub account. Click the **Authorize** button.

Authorize Azure SignalR Chat

Azure SignalR Chat by [wesmc7777](#)  
wants to access your [wesmc7777](#) account

Personal user data  
Email addresses (read-only)

Authorize wesmc7777

Authorizing will redirect to  
<http://localhost:5000>

Not owned or operated by GitHub

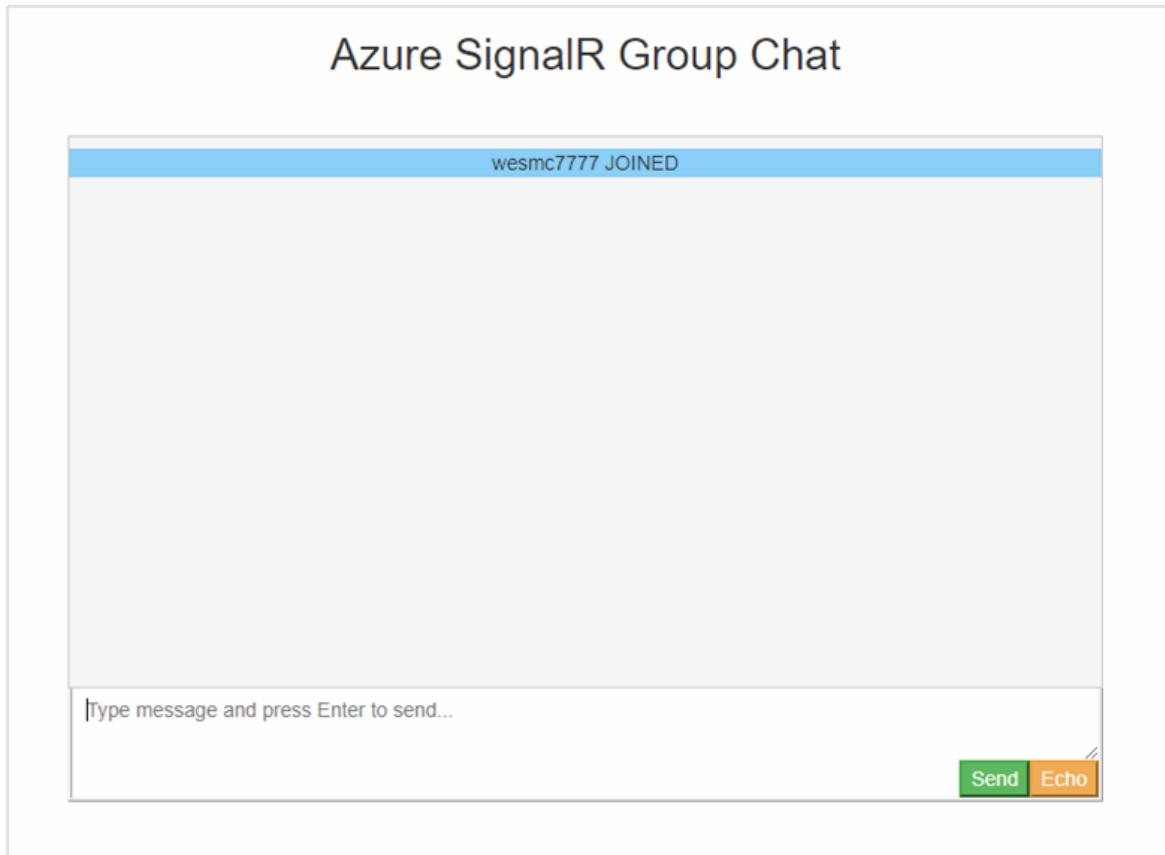
Created 2 weeks ago

Fewer than 10 GitHub users

[Learn more about OAuth](#)

You will be redirected back to the chat application and logged in with your GitHub account name. The web application determined your account name by authenticating you using the new authentication you

added.



Now that the chat app performs authentication with GitHub and stores the authentication information as cookies, you should deploy it to Azure so other users can authenticate with their accounts and communicate from other workstations.

## Deploy the app to Azure

Prepare your environment for the Azure CLI:

- Use the Bash environment in [Azure Cloud Shell](#).  
[Launch Cloud Shell](#)
- If you prefer, [install](#) the Azure CLI to run CLI reference commands.
  - If you're using a local installation, sign in to the Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. For additional sign-in options, see [Sign in with the Azure CLI](#).
  - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
  - Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).

In this section, you will use the Azure command-line interface (CLI) to create a new web app in [Azure App Service](#) to host your ASP.NET application in Azure. The web app will be configured to use local Git deployment. The web app will also be configured with your SignalR connection string, GitHub OAuth app secrets, and a deployment user.

When creating the following resources, make sure to use the same resource group that your SignalR Service resource resides in. This approach will make clean up a lot easier later when you want to remove all the resources. The examples given assume you used the group name recommended in previous tutorials,

## Create the web app and plan

Copy the text for the commands below and update the parameters. Paste the updated script into the Azure Cloud Shell, and press **Enter** to create a new App Service plan and web app.

```
#=====
#== Update these variable for your resource group name. ===
#=====
ResourceGroupName=SignalRTTestResources

#=====
#== Update these variable for your web app. ===
#=====
WebAppName=myWebAppName
WebAppPlan=myAppServicePlanName

# Create an App Service plan.
az appservice plan create --name $WebAppPlan --resource-group $ResourceGroupName \
--sku FREE

# Create the new Web App
az webapp create --name $WebAppName --resource-group $ResourceGroupName \
--plan $WebAppPlan
```

PARAMETER	DESCRIPTION
ResourceGroupName	This resource group name was suggested in previous tutorials. It is a good idea to keep all tutorial resources grouped together. Use the same resource group you used in the previous tutorials.
WebAppPlan	Enter a new, unique, App Service Plan name.
WebAppName	This will be the name of the new web app and part of the URL. Use a unique name. For example, signalrtestwebapp22665120.

## Add app settings to the web app

In this section, you will add app settings for the following components:

- SignalR Service resource connection string
- GitHub OAuth app client ID
- GitHub OAuth app client secret

Copy the text for the commands below and update the parameters. Paste the updated script into the Azure Cloud Shell, and press **Enter** to add the app settings:

```

=====
===== Update these variables for your GitHub OAuth App. =====
GitHubClientId=1234567890
GitHubClientSecret=1234567890

=====
===== Update these variables for your resources. =====
ResourceGroupName=SignalRTTestResources
SignalRServiceResource=mySignalRresourcename
WebAppName=myWebAppName

# Get the SignalR primary connection string
primaryConnectionString=$(az signalr key list --name $SignalRServiceResource \
--resource-group $ResourceGroupName --query primaryConnectionString -o tsv)

#Add an app setting to the web app for the SignalR connection
az webapp config appsettings set --name $WebAppName \
--resource-group $ResourceGroupName \
--settings "Azure__SignalR__ConnectionString=$primaryConnectionString"

#Add the app settings to use with GitHub authentication
az webapp config appsettings set --name $WebAppName \
--resource-group $ResourceGroupName \
--settings "GitHubClientId=$GitHubClientId"
az webapp config appsettings set --name $WebAppName \
--resource-group $ResourceGroupName \
--settings "GitHubClientSecret=$GitHubClientSecret"

```

PARAMETER	DESCRIPTION
GitHubClientId	Assign this variable the secret Client Id for your GitHub OAuth App.
GitHubClientSecret	Assign this variable the secret password for your GitHub OAuth App.
ResourceGroupName	Update this variable to be the same resource group name you used in the previous section.
SignalRServiceResource	Update this variable with the name of the SignalR Service resource you created in the quickstart. For example, signalrtestsvc48778624.
WebAppName	Update this variable with the name of the new web app you created in the previous section.

## Configure the web app for local Git deployment

In the Azure Cloud Shell, paste the following script. This script creates a new deployment user name and password that you will use when deploying your code to the web app with Git. The script also configures the web app for deployment with a local Git repository, and returns the Git deployment URL.

```

=====
===== Update these variables for your resources. =====
ResourceGroupName=SignalRTTestResources
WebAppName=myWebAppName

=====
===== Update these variables for your deployment user. =====
DeploymentUserName=myUserName
DeploymentUserPassword=myPassword

# Add the desired deployment user name and password
az webapp deployment user set --user-name $DeploymentUserName \
--password $DeploymentUserPassword

# Configure Git deployment and note the deployment URL in the output
az webapp deployment source config-local-git --name $WebAppName \
--resource-group $ResourceGroupName \
--query [url] -o tsv

```

PARAMETER	DESCRIPTION
DeploymentUserName	Choose a new deployment user name.
DeploymentUserPassword	Choose a password for the new deployment user.
ResourceGroupName	Use the same resource group name you used in the previous section.
WebAppName	This will be the name of the new web app you created previously.

Make a note the Git deployment URL returned from this command. You will use this URL later.

### Deploy your code to the Azure web app

To deploy your code, execute the following commands in a Git shell.

1. Navigate to the root of your project directory. If you don't have the project initialized with a Git repository, execute following command:

```
git init
```

2. Add a remote for the Git deployment URL you noted earlier:

```
git remote add Azure <your git deployment url>
```

3. Stage all files in the initialized repository and add a commit.

```
git add -A
git commit -m "init commit"
```

4. Deploy your code to the web app in Azure.

```
git push Azure main
```

You will be prompted to authenticate in order to deploy the code to Azure. Enter the user name and password of the deployment user you created above.

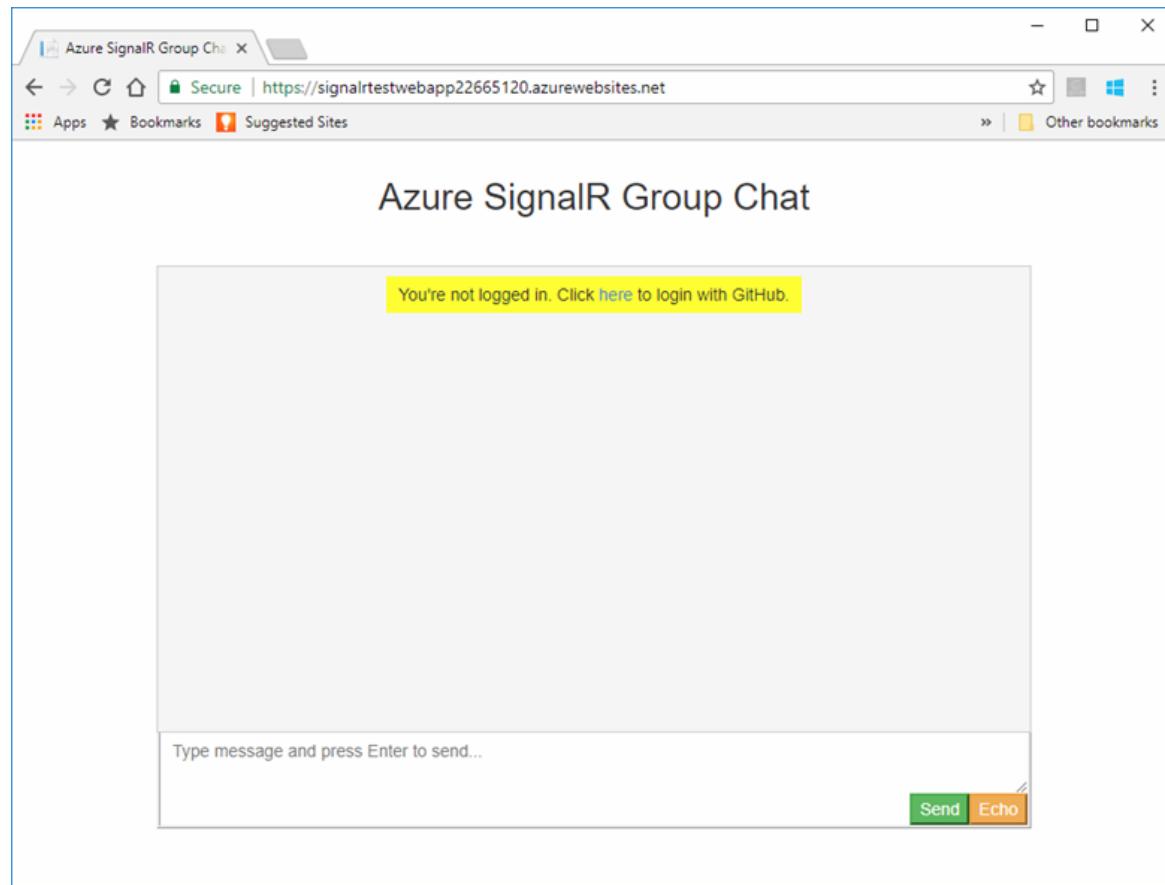
### Update the GitHub OAuth app

The last thing you need to do is update the **Homepage URL** and **Authorization callback URL** of the GitHub OAuth app to point to the new hosted app.

1. Open <https://github.com> in a browser and navigate to your account's **Settings > Developer settings > Oauth Apps**.
2. Click on your authentication app and update the **Homepage URL** and **Authorization callback URL** as shown below:

SETTING	EXAMPLE
Homepage URL	<a href="https://signalrtestwebapp22665120.azurewebsites.net/home">https://signalrtestwebapp22665120.azurewebsites.net/home</a>
Authorization callback URL	<a href="https://signalrtestwebapp22665120.azurewebsites.net/signin-github">https://signalrtestwebapp22665120.azurewebsites.net/signin-github</a>

3. Navigate to your web app URL and test the application.



### Clean up resources

If you will be continuing to the next tutorial, you can keep the resources created in this quickstart and reuse them with the next tutorial.

Otherwise, if you are finished with the quickstart sample application, you can delete the Azure resources created in this quickstart to avoid charges.

## IMPORTANT

Deleting a resource group is irreversible and that the resource group and all the resources in it are permanently deleted. Make sure that you do not accidentally delete the wrong resource group or resources. If you created the resources for hosting this sample inside an existing resource group that contains resources you want to keep, you can delete each resource individually from their respective blades instead of deleting the resource group.

Sign in to the [Azure portal](#) and click **Resource groups**.

In the **Filter by name...** textbox, type the name of your resource group. The instructions for this article used a resource group named *SignalRTTestResources*. On your resource group in the result list, click ... then **Delete resource group**.

The screenshot shows the Microsoft Azure Resource Groups blade. The left sidebar has a red box around the 'Resource groups' item. The main area shows a table with one item: 'SignalRTTestResources'. A red box highlights the 'NAME' column entry 'SignalRTTestResources'. To the right of the table, there is a 'Delete resource group' button with a red box around it, and a three-dot ellipsis button.

You will be asked to confirm the deletion of the resource group. Type the name of your resource group to confirm, and click **Delete**.

After a few moments, the resource group and all of its contained resources are deleted.

## Next steps

In this tutorial, you added authentication with OAuth to provide a better approach to authentication with Azure SignalR Service. To learn more about using Azure SignalR Server, continue to the Azure CLI samples for SignalR Service.

[Azure SignalR CLI Samples](#)

# Reacting to Azure SignalR Service events

11/2/2020 • 2 minutes to read • [Edit Online](#)

Azure SignalR Service events allow applications to react to client connections connected or disconnected using modern serverless architectures. It does so without the need for complicated code or expensive and inefficient polling services. Instead, events are pushed through [Azure Event Grid](#) to subscribers such as [Azure Functions](#), [Azure Logic Apps](#), or even to your own custom http listener. With Azure SignalR, you only pay for what you consume.

Azure SignalR Service events are reliably sent to the Event Grid service which provides reliable delivery services to your applications through rich retry policies and dead-letter delivery. To learn more, see [Event Grid message delivery and retry](#).



## Serverless state

Azure SignalR Service events are only active when client connections are in a serverless state. If a client does not route to a hub server, it goes into the serverless state. Classic mode only works when the hub that client connections connect to doesn't have a hub server. Serverless mode is recommended as a best practice. To learn more details about service mode, see [How to choose Service Mode](#).

## Available Azure SignalR Service events

Event grid uses [event subscriptions](#) to route event messages to subscribers. Azure SignalR Service event subscriptions support two types of events:

EVENT NAME	DESCRIPTION
<code>Microsoft.SignalRService.ClientConnectionConnected</code>	Raised when a client connection is connected.
<code>Microsoft.SignalRService.ClientConnectionDisconnected</code>	Raised when a client connection is disconnected.

## Event schema

Azure SignalR Service events contain all the information you need to respond to the changes in your data. You can identify an Azure SignalR Service event with the `eventType` property starts with "Microsoft.SignalRService". Additional information about the usage of Event Grid event properties is documented at [Event Grid event schema](#).

Here is an example of a client connection connected event:

```
[{
  "topic": "/subscriptions/{subscription-id}/resourceGroups/signalr-
    rg/providers/Microsoft.SignalRService/SignalR/signalr-resource",
  "subject": "/hub/chat",
  "eventType": "Microsoft.SignalRService.ClientConnectionConnected",
  "eventTime": "2019-06-10T18:41:00.9584103Z",
  "id": "831e1650-001e-001b-66ab-eeb76e069631",
  "data": {
    "timestamp": "2019-06-10T18:41:00.9584103Z",
    "hubName": "chat",
    "connectionId": "crH0uxVSvP61p5wkFY1x1A",
    "userId": "user-eymwo23"
  },
  "dataVersion": "1.0",
  "metadataVersion": "1"
}]
```

For more information, see [SignalR Service events schema](#).

## Next steps

Learn more about Event Grid and give Azure SignalR Service events a try:

[Try a sample Event Grid integration with Azure SignalR Service About Event Grid](#)

# Azure Policy Regulatory Compliance controls for Azure SignalR

8/13/2021 • 6 minutes to read • [Edit Online](#)

Regulatory Compliance in Azure Policy provides Microsoft created and managed initiative definitions, known as **built-ins**, for the **compliance domains** and **security controls** related to different compliance standards. This page lists the **compliance domains** and **security controls** for Azure SignalR. You can assign the built-ins for a **security control** individually to help make your Azure resources compliant with the specific standard.

The title of each built-in policy definition links to the policy definition in the Azure portal. Use the link in the **Policy Version** column to view the source on the [Azure Policy GitHub repo](#).

## IMPORTANT

Each control below is associated with one or more Azure Policy definitions. These policies may help you [assess compliance](#) with the control; however, there often is not a one-to-one or complete match between a control and one or more policies. As such, **Compliant** in Azure Policy refers only to the policies themselves; this doesn't ensure you're fully compliant with all requirements of a control. In addition, the compliance standard includes controls that aren't addressed by any Azure Policy definitions at this time. Therefore, compliance in Azure Policy is only a partial view of your overall compliance status. The associations between controls and Azure Policy Regulatory Compliance definitions for these compliance standards may change over time.

## Azure Security Benchmark

The [Azure Security Benchmark](#) provides recommendations on how you can secure your cloud solutions on Azure. To see how this service completely maps to the Azure Security Benchmark, see the [Azure Security Benchmark mapping files](#).

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - Azure Security Benchmark](#).

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Network Security	NS-2	Connect private networks together	<a href="#">Azure SignalR Service should use private link</a>	1.0.1
Network Security	NS-3	Establish private network access to Azure services	<a href="#">Azure SignalR Service should use private link</a>	1.0.1

## FedRAMP High

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - FedRAMP High](#). For more information about this compliance standard, see [FedRAMP High](#).

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Access Control	AC-4	Information Flow Enforcement	Azure SignalR Service should use private link	1.0.1
Access Control	AC-4	Information Flow Enforcement	Azure Web PubSub Service should use private link	1.0.0
Access Control	AC-17	Remote Access	Azure SignalR Service should use private link	1.0.1
Access Control	AC-17	Remote Access	Azure Web PubSub Service should use private link	1.0.0
Access Control	AC-17 (1)	Automated Monitoring / Control	Azure SignalR Service should use private link	1.0.1
Access Control	AC-17 (1)	Automated Monitoring / Control	Azure Web PubSub Service should use private link	1.0.0
System and Communications Protection	SC-7	Boundary Protection	Azure SignalR Service should use private link	1.0.1
System and Communications Protection	SC-7	Boundary Protection	Azure Web PubSub Service should use private link	1.0.0
System and Communications Protection	SC-7 (3)	Access Points	Azure SignalR Service should use private link	1.0.1
System and Communications Protection	SC-7 (3)	Access Points	Azure Web PubSub Service should use private link	1.0.0

## FedRAMP Moderate

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - FedRAMP Moderate](#). For more information about this compliance standard, see [FedRAMP Moderate](#).

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Access Control	AC-4	Information Flow Enforcement	Azure SignalR Service should use private link	1.0.1

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY	POLICY VERSION
Access Control	AC-4	Information Flow Enforcement	Azure Web PubSub Service should use private link	1.0.0
Access Control	AC-17	Remote Access	Azure SignalR Service should use private link	1.0.1
Access Control	AC-17	Remote Access	Azure Web PubSub Service should use private link	1.0.0
Access Control	AC-17 (1)	Automated Monitoring / Control	Azure SignalR Service should use private link	1.0.1
Access Control	AC-17 (1)	Automated Monitoring / Control	Azure Web PubSub Service should use private link	1.0.0
System and Communications Protection	SC-7	Boundary Protection	Azure SignalR Service should use private link	1.0.1
System and Communications Protection	SC-7	Boundary Protection	Azure Web PubSub Service should use private link	1.0.0
System and Communications Protection	SC-7 (3)	Access Points	Azure SignalR Service should use private link	1.0.1
System and Communications Protection	SC-7 (3)	Access Points	Azure Web PubSub Service should use private link	1.0.0

## New Zealand ISM Restricted

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - New Zealand ISM Restricted](#). For more information about this compliance standard, see [New Zealand ISM Restricted](#).

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Infrastructure	INF-9	10.8.35 Security Architecture	Azure SignalR Service should use private link	1.0.1

## NIST SP 800-53 Rev. 4

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - NIST SP 800-53 Rev. 4](#). For more information about this compliance

standard, see [NIST SP 800-53 Rev. 4](#).

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Access Control	AC-4	Information Flow Enforcement	Azure SignalR Service should use private link	1.0.1
Access Control	AC-4	Information Flow Enforcement	Azure Web PubSub Service should use private link	1.0.0
Access Control	AC-17	Remote Access	Azure SignalR Service should use private link	1.0.1
Access Control	AC-17	Remote Access	Azure Web PubSub Service should use private link	1.0.0
Access Control	AC-17 (1)	Automated Monitoring / Control	Azure SignalR Service should use private link	1.0.1
Access Control	AC-17 (1)	Automated Monitoring / Control	Azure Web PubSub Service should use private link	1.0.0
System and Communications Protection	SC-7	Boundary Protection	Azure SignalR Service should use private link	1.0.1
System and Communications Protection	SC-7	Boundary Protection	Azure Web PubSub Service should use private link	1.0.0
System and Communications Protection	SC-7 (3)	Access Points	Azure SignalR Service should use private link	1.0.1
System and Communications Protection	SC-7 (3)	Access Points	Azure Web PubSub Service should use private link	1.0.0

## NIST SP 800-53 Rev. 5

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see [Azure Policy Regulatory Compliance - NIST SP 800-53 Rev. 5](#). For more information about this compliance standard, see [NIST SP 800-53 Rev. 5](#).

DOMAIN	CONTROL ID	CONTROL TITLE	POLICY (AZURE PORTAL)	POLICY VERSION (GITHUB)
Access Control	AC-4	Information Flow Enforcement	Azure SignalR Service should use private link	1.0.1

Domain	Control ID	Control Title	Policy	Policy Version
Access Control	AC-4	Information Flow Enforcement	Azure Web PubSub Service should use private link	1.0.0
Access Control	AC-17	Remote Access	Azure SignalR Service should use private link	1.0.1
Access Control	AC-17	Remote Access	Azure Web PubSub Service should use private link	1.0.0
Access Control	AC-17 (1)	Monitoring and Control	Azure SignalR Service should use private link	1.0.1
Access Control	AC-17 (1)	Monitoring and Control	Azure Web PubSub Service should use private link	1.0.0
System and Communications Protection	SC-7	Boundary Protection	Azure SignalR Service should use private link	1.0.1
System and Communications Protection	SC-7	Boundary Protection	Azure Web PubSub Service should use private link	1.0.0
System and Communications Protection	SC-7 (3)	Access Points	Azure SignalR Service should use private link	1.0.1
System and Communications Protection	SC-7 (3)	Access Points	Azure Web PubSub Service should use private link	1.0.0

## Next steps

- Learn more about [Azure Policy Regulatory Compliance](#).
- See the built-ins on the [Azure Policy GitHub repo](#).

# How to scale an Azure SignalR Service instance?

11/2/2020 • 2 minutes to read • [Edit Online](#)

This article shows you how to scale your instance of Azure SignalR Service. There are two scenarios for scaling, scale up and scale out.

- **Scale up:** Get more units, connections, messages, and more. You scale up by changing the pricing tier from Free to Standard.
- **Scale out:** Increase the number of SignalR units. You can scale out to as many as 100 units. There are limited unit options to select for the scaling: 1, 2, 5, 10, 20, 50 and 100 units for a single SignalR Service instance.

The scale settings take a few minutes to apply. In rare cases, it may take around 30 minutes to apply. They don't require you to change your code or redeploy your server application.

For information about the pricing and capacities of individual SignalR Service, see [Azure SignalR Service Pricing Details](#).

## NOTE

Changing SignalR Service from **Free** tier to **Standard** tier or vice versa, the public service IP will be changed and it usually takes 30-60 minutes to propagate the change to DNS servers across the entire internet. Your service might be unreachable before DNS gets updated. Generally it's not recommended to change your pricing tier too often.

## Scale on Azure portal

1. In your browser, open the [Azure portal](#).
2. In your SignalR Service page, from the left menu, select **Scale**.
3. Choose your pricing tier, and then click **Select**. Set the unit count for **Standard Tier**.

The screenshot shows the Microsoft Azure portal interface for managing a SignalR service named "scaledemo - Scale". The left sidebar contains a navigation menu with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Keys, Quickstart, Scale (which is currently selected), Settings, CORS, Properties, Locks, Export template, Monitoring, Alerts, Metrics, Diagnostic settings, Support + troubleshooting, and New support request. The main content area displays the current configuration for the "Scale" tab, showing a "Pricing Tier" set to "Standard" and a "Unit count" of "2". A prominent warning message box is present, cautioning users about the potential impact of changing the pricing tier from Free to Standard, including a note that it can take up to 60 minutes for DNS propagation and that the service might be unreachable during this period.

4. Click Save.

## Scale using Azure CLI

This script creates a new SignalR Service resource of **Free** Tier and a new resource group, and scales it up to **Standard** Tier.

```
#!/bin/bash

# Generate a unique suffix for the service name
let randomNum=$RANDOM*$RANDOM

# Generate a unique service and group name with the suffix
SignalRName=SignalRTTestSvc$randomNum
#resource name must be lowercase
mySignalRSvcName=${SignalRName,,}
myResourceGroupName=$SignalRName"Group"

# Create resource group
az group create --name $myResourceGroupName --location eastus

# Create the Azure SignalR Service resource
az signalr create \
--name $mySignalRSvcName \
--resource-group $myResourceGroupName \
--sku Free_F1 \
--service-mode Default

# Scale up to Standard Tier, and scale out to 50 units
az signalr update \
--name $mySignalRSvcName \
--resource-group $myResourceGroupName \
--sku Standard_S1 \
--unit-count 50
```

Make a note of the actual name generated for the new resource group. You will use that resource group name when you want to delete all group resources.

## Clean up deployment

After the sample script has been run, the following command can be used to remove the resource group and all resources associated with it.

```
az group delete --name myResourceGroup
```

## Compare pricing tiers

For detailed information, such as included messages and connections for each pricing tier, see [SignalR Service Pricing Details](#).

For a table of service limits, quotas, and constraints in each tier, see [SignalR Service limits](#).

## Next steps

In this guide, you learned about how to scale single SignalR Service instance.

Multiple endpoints are also supported for scaling, sharding, and cross-region scenarios.

[scale SignalR Service with multiple instances](#)

# How to scale SignalR Service with multiple instances?

11/2/2020 • 5 minutes to read • [Edit Online](#)

The latest SignalR Service SDK supports multiple endpoints for SignalR Service instances. You can use this feature to scale the concurrent connections, or use it for cross-region messaging.

## For ASP.NET Core

### How to add multiple endpoints from config?

Config with key `Azure:SignalR:ConnectionString` or `Azure:SignalR:ConnectionString:` for SignalR Service connection string.

If the key starts with `Azure:SignalR:ConnectionString:`, it should be in format

`Azure:SignalR:ConnectionString:{Name}:{EndpointType}`, where `Name` and `EndpointType` are properties of the `ServiceEndpoint` object, and are accessible from code.

You can add multiple instance connection strings using the following `dotnet` commands:

```
dotnet user-secrets set Azure:SignalR:ConnectionString:east-region-a <ConnectionString1>
dotnet user-secrets set Azure:SignalR:ConnectionString:east-region-b:primary <ConnectionString2>
dotnet user-secrets set Azure:SignalR:ConnectionString:backup:secondary <ConnectionString3>
```

### How to add multiple endpoints from code?

A `ServiceEndpoint` class is introduced to describe the properties of an Azure SignalR Service endpoint. You can configure multiple instance endpoints when using Azure SignalR Service SDK through:

```
services.AddSignalR()
    .AddAzureSignalR(options =>
{
    options.Endpoints = new ServiceEndpoint[]
    {
        // Note: this is just a demonstration of how to set options.Endpoints
        // Having ConnectionStrings explicitly set inside the code is not encouraged
        // You can fetch it from a safe place such as Azure KeyVault
        new ServiceEndpoint("<ConnectionString0>"),
        new ServiceEndpoint("<ConnectionString1>", type: EndpointType.Primary, name: "east-region-a"),
        new ServiceEndpoint("<ConnectionString2>", type: EndpointType.Primary, name: "east-region-b"),
        new ServiceEndpoint("<ConnectionString3>", type: EndpointType.Secondary, name: "backup"),
    };
});
```

### How to customize endpoint router?

By default, the SDK uses the `DefaultEndpointRouter` to pick up endpoints.

#### Default behavior

1. Client request routing

When client `/negotiate` with the app server. By default, SDK **randomly selects** one endpoint from the set of available service endpoints.

## 2. Server message routing

When *sending message to a specific connection*, and the target connection is routed to current server, the message goes directly to that connected endpoint. Otherwise, the messages are broadcasted to every Azure SignalR endpoint.

### Customize routing algorithm

You can create your own router when you have special knowledge to identify which endpoints the messages should go to.

A custom router is defined below as an example when groups starting with `east-` always go to the endpoint named `east`:

```
private class CustomRouter : EndpointRouterDecorator
{
    public override IEnumerable<ServiceEndpoint> GetEndpointsForGroup(string groupName,
    IEnumerable<ServiceEndpoint> endpoints)
    {
        // Override the group broadcast behavior, if the group name starts with "east-", only send messages
        to endpoints inside east
        if (groupName.StartsWith("east-"))
        {
            return endpoints.Where(e => e.Name.StartsWith("east-"));
        }

        return base.GetEndpointsForGroup(groupName, endpoints);
    }
}
```

Another example below, that overrides the default negotiate behavior, to select the endpoints depends on where the app server is located.

```
private class CustomRouter : EndpointRouterDecorator
{
    public override ServiceEndpoint GetNegotiateEndpoint(HttpContext context, IEnumerable<ServiceEndpoint>
    endpoints)
    {
        // Override the negotiate behavior to get the endpoint from query string
        var endpointName = context.Request.Query["endpoint"];
        if (endpointName.Count == 0)
        {
            context.Response.StatusCode = 400;
            var response = Encoding.UTF8.GetBytes("Invalid request");
            context.Response.Body.Write(response, 0, response.Length);
            return null;
        }

        return endpoints.FirstOrDefault(s => s.Name == endpointName && s.Online) // Get the endpoint with
        name matching the incoming request
        ?? base.GetNegotiateEndpoint(context, endpoints); // Or fallback to the default behavior to
        randomly select one from primary endpoints, or fallback to secondary when no primary ones are online
    }
}
```

Don't forget to register the router to DI container using:

```

services.AddSingleton(typeof(IEndpointRouter), typeof(CustomRouter));
services.AddSignalR()
    .AddAzureSignalR(
        options =>
    {
        options.Endpoints = new ServiceEndpoint[]
        {
            new ServiceEndpoint(name: "east", connectionString: "<connectionString1>"),
            new ServiceEndpoint(name: "west", connectionString: "<connectionString2>"),
            new ServiceEndpoint("<connectionString3>")
        };
    });

```

## For ASP.NET

### How to add multiple endpoints from config?

Config with key `Azure:SignalR:ConnectionString` or `Azure:SignalR:ConnectionString:` for SignalR Service connection string.

If the key starts with `Azure:SignalR:ConnectionString:`, it should be in format

`Azure:SignalR:ConnectionString:{Name}:{EndpointType}`, where `Name` and `EndpointType` are properties of the `ServiceEndpoint` object, and are accessible from code.

You can add multiple instance connection strings to `web.config`:

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <connectionStrings>
        <add name="Azure:SignalR:ConnectionString" connectionString="<ConnectionString1>"/>
        <add name="Azure:SignalR:ConnectionString:en-us" connectionString="<ConnectionString2>"/>
        <add name="Azure:SignalR:ConnectionString:zh-cn:secondary" connectionString="<ConnectionString3>"/>
        <add name="Azure:SignalR:ConnectionString:Backup:secondary" connectionString="<ConnectionString4>"/>
    </connectionStrings>
    ...
</configuration>

```

### How to add multiple endpoints from code?

A `ServiceEndpoint` class is introduced to describe the properties of an Azure SignalR Service endpoint. You can configure multiple instance endpoints when using Azure SignalR Service SDK through:

```

app.MapAzureSignalR(
    this.GetType().FullName,
    options => {
        options.Endpoints = new ServiceEndpoint[]
        {
            // Note: this is just a demonstration of how to set options.Endpoints
            // Having ConnectionStrings explicitly set inside the code is not encouraged
            // You can fetch it from a safe place such as Azure KeyVault
            new ServiceEndpoint("<ConnectionString1>"),
            new ServiceEndpoint("<ConnectionString2>"),
            new ServiceEndpoint("<ConnectionString3>"),
        }
    });

```

### How to customize router?

The only difference between ASP.NET SignalR and ASP.NET Core SignalR is the http context type for `GetNegotiateEndpoint`. For ASP.NET SignalR, it is of `IOwinContext` type.

Below is the custom negotiate example for ASP.NET SignalR:

```
private class CustomRouter : EndpointRouterDecorator
{
    public override ServiceEndpoint GetNegotiateEndpoint(IOwinContext context, IEnumerable<ServiceEndpoint> endpoints)
    {
        // Override the negotiate behavior to get the endpoint from query string
        var endpointName = context.Request.Query["endpoint"];
        if (string.IsNullOrEmpty(endpointName))
        {
            context.Response.StatusCode = 400;
            context.Response.Write("Invalid request.");
            return null;
        }

        return endpoints.FirstOrDefault(s => s.Name == endpointName && s.Online) // Get the endpoint with
name matching the incoming request
        ?? base.GetNegotiateEndpoint(context, endpoints); // Or fallback to the default behavior to
randomly select one from primary endpoints, or fallback to secondary when no primary ones are online
    }
}
```

Don't forget to register the router to DI container using:

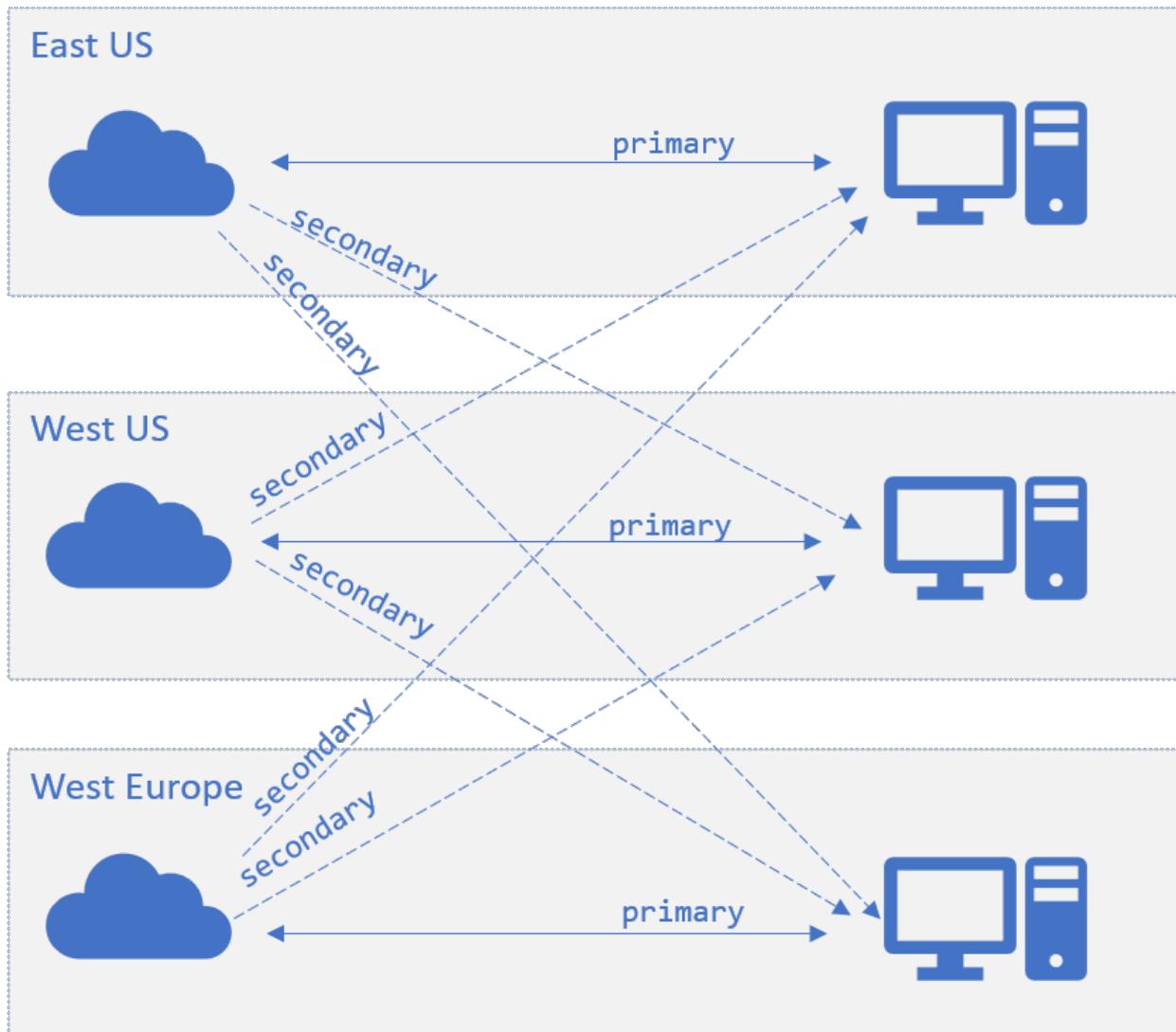
```
var hub = new HubConfiguration();
var router = new CustomRouter();
hub.Resolver.Register(typeof(IEndpointRouter), () => router);
app.MapAzureSignalR(GetType().FullName, hub, options => {
    options.Endpoints = new ServiceEndpoint[]
    {
        new ServiceEndpoint(name: "east", connectionString: "<connectionString1>"),
        new ServiceEndpoint(name: "west", connectionString: "<connectionString2>"),
        new ServiceEndpoint("<connectionString3>")
    };
});
```

## Configuration in cross-region scenarios

The `ServiceEndpoint` object has an `EndpointType` property with value `primary` or `secondary`.

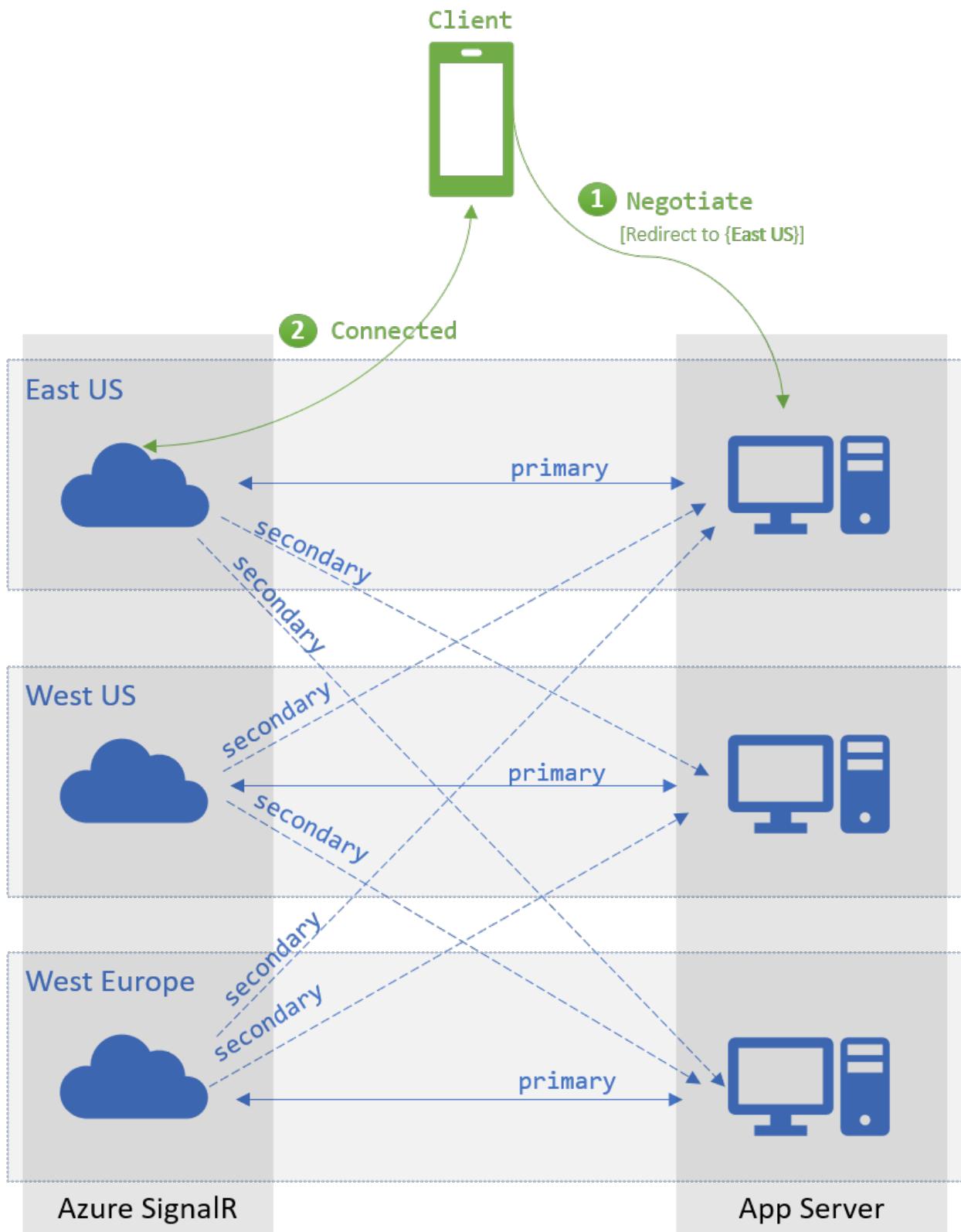
`primary` endpoints are preferred endpoints to receive client traffic, and are considered to have more reliable network connections; `secondary` endpoints are considered to have less reliable network connections and are used only for taking server to client traffic, for example, broadcasting messages, not for taking client to server traffic.

In cross-region cases, network can be unstable. For one app server located in *East US*, the SignalR Service endpoint located in the same *East US* region can be configured as `primary` and endpoints in other regions marked as `secondary`. In this configuration, service endpoints in other regions can **receive** messages from this *East US* app server, but there will be no **cross-region** clients routed to this app server. The architecture is shown in the diagram below:



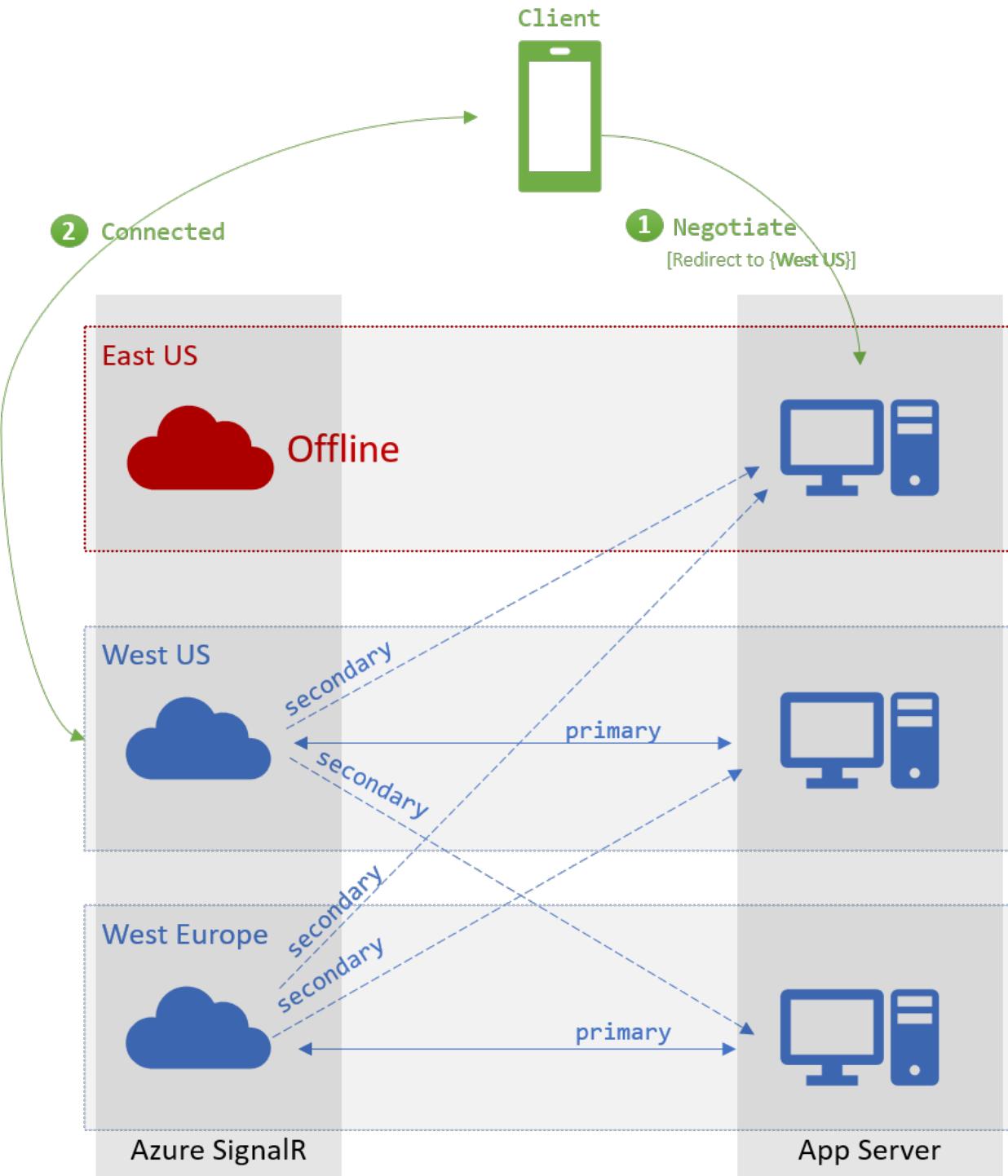
When a client tries `/negotiate` with the app server, with the default router, SDK **randomly selects** one endpoint from the set of available `primary` endpoints. When the primary endpoint is not available, SDK then **randomly selects** from all available `secondary` endpoints. The endpoint is marked as **available** when the connection between server and the service endpoint is alive.

In cross-region scenario, when a client tries `/negotiate` with the app server hosted in *East US*, by default it always returns the `primary` endpoint located in the same region. When all *East US* endpoints are not available, the client is redirected to endpoints in other regions. Fail-over section below describes the scenario in detail.



## Fail-over

When all `primary` endpoints are not available, client's `/negotiate` picks from the available `secondary` endpoints. This fail-over mechanism requires that each endpoint should serve as `primary` endpoint to at least one app server.



## Next steps

In this guide, you learned about how to configure multiple instances in the same application for scaling, sharding, and cross-region scenarios.

Multiple endpoints supports can also be used in high availability and disaster recovery scenarios.

[Setup SignalR Service for disaster recovery and high availability](#)

# How to rotate access key for Azure SignalR Service

4/21/2021 • 2 minutes to read • [Edit Online](#)

Each Azure SignalR Service instance has a pair of access keys called Primary and Secondary keys. They're used to authenticate SignalR clients when requests are made to the service. The keys are associated with the instance endpoint url. Keep your keys secure, and rotate them regularly. You're provided with two access keys, so you can maintain connections by using one key while regenerating the other.

## Why rotate access keys?

For security reasons and compliance requirements, routinely rotate your access keys.

## Regenerate access keys

1. Go to the [Azure portal](#), and sign in with your credentials.
2. Find the **Keys** section in the Azure SignalR Service instance with the keys that you want to regenerate.
3. Select **Keys** on the navigation menu.
4. Select **Regenerate Primary Key** or **Regenerate Secondary Key**.

A new key and corresponding connection string are created and displayed.

The screenshot shows the Azure portal interface for managing keys. The left sidebar shows various service categories like Home, Dashboard, and App Services. The main content area is titled 'azuresignalrdemos - Keys' under the 'SignalR' provider. The 'Keys' section is highlighted. It contains fields for 'Host name' (azuresignalrdemos.service.signalr.net), 'Primary KEY' (<Primary Key>), and 'Secondary KEY' (<Secondary Key>). Below each key are 'CONNECTION STRING' fields: 'Endpoint=https://azuresignalrdemos.service.signalr.net?AccessKey=<Primary Key>;Version=1.0' for the primary and 'Endpoint=https://azuresignalrdemos.service.signalr.net?AccessKey=<Secondary Key>;Version=1.0' for the secondary. Red boxes highlight the 'Regenerate Primary Key' and 'Regenerate Secondary Key' buttons.

You also can regenerate keys by using the [Azure CLI](#).

## Update configurations with new connection strings

1. Copy the newly generated connection string.
2. Update all configurations to use the new connection string.

3. Restart the application as needed.

## Forced access key regeneration

Azure SignalR Service might enforce a mandatory access key regeneration under certain situations. The service notifies customers via email and portal notification. If you receive this communication or encounter service failure due to an access key, rotate the keys by following the instructions in this guide.

## Next steps

Rotate your access keys regularly as a good security practice.

In this guide, you learned how to regenerate access keys. Continue to the next tutorials about authentication with OAuth or with Azure Functions.

[Integrate with ASP.NET core identity](#)

[Build a serverless real-time app with authentication](#)

# Use service tags for Azure SignalR Service

8/2/2021 • 2 minutes to read • [Edit Online](#)

You can use [Service Tags](#) for Azure SignalR Service when configuring [Network Security Group](#). It allows you to define inbound/outbound network security rule for Azure SignalR Service endpoints without need to hardcode IP addresses.

Azure SignalR Service manages these service tags. You can't create your own service tag or modify an existing one. Microsoft manages these address prefixes that match to the service tag and automatically updates the service tag as addresses change.

## NOTE

Starting from 15 August 2021, Azure SignalR Service supports bidirectional Service Tag for both inbound and outbound traffic.

## Use service tag on portal

### Configure outbound traffic

You can allow outbound traffic to Azure SignalR Service by adding a new outbound network security rule:

1. Go to the network security group.
2. Click on the settings menu called **Outbound security rules**.
3. Click the button **+ Add** on the top.
4. Choose **Service Tag** under **Destination**.
5. Choose **AzureSignalR** under **Destination service tag**.
6. Fill in **443** in **Destination port ranges**.

 Add outbound security rule X

Basic

**Source \*** ⓘ ▼  
Any

**Source port ranges \*** ⓘ ▼  
\*

**Destination \*** ⓘ ▼  
Service Tag

**Destination service tag** ⓘ ▼  
AzureSignalR

**Destination port ranges \*** ⓘ ▼  
443

**Protocol \***  
 Any  TCP  UDP  ICMP

**Action \***  
 Allow  Deny

**Priority \*** ⓘ ▼  
100

**Name \*** ✓  
Port\_443

**Description**

**Add**

7. Adjust other fields according to your needs.

8. Click Add.

### Configure inbound traffic

If you have upstreams, you can also allow inbound traffic from Azure SignalR Service by adding a new inbound network security rule:

1. Go to the network security group.
2. Click on the settings menu called **Inbound security rules**.
3. Click the button **+ Add** on the top.
4. Choose **Service Tag** under **Source**.

5. Choose AzureSignalR under Source service tag.

6. Fill in \* in Source port ranges.

 Add inbound security rule X

---

Source ⓘ

Service Tag AzureSignalR

Source service tag \* ⓘ

AzureSignalR

Source port ranges \* ⓘ

\* \*

Destination ⓘ

Any

Service ⓘ

HTTPS

Destination port ranges ⓘ

443

Protocol

Any  
 TCP  
 UDP  
 ICMP

Action

Allow  
 Deny

Priority \* ⓘ

100 ✓

Name \*

Port\_443 ✓

Description

---

Add Cancel

7. Adjust other fields according to your needs.

8. Click Add.

## Next steps

- Network security groups: service tags

# Use private endpoints for Azure SignalR Service

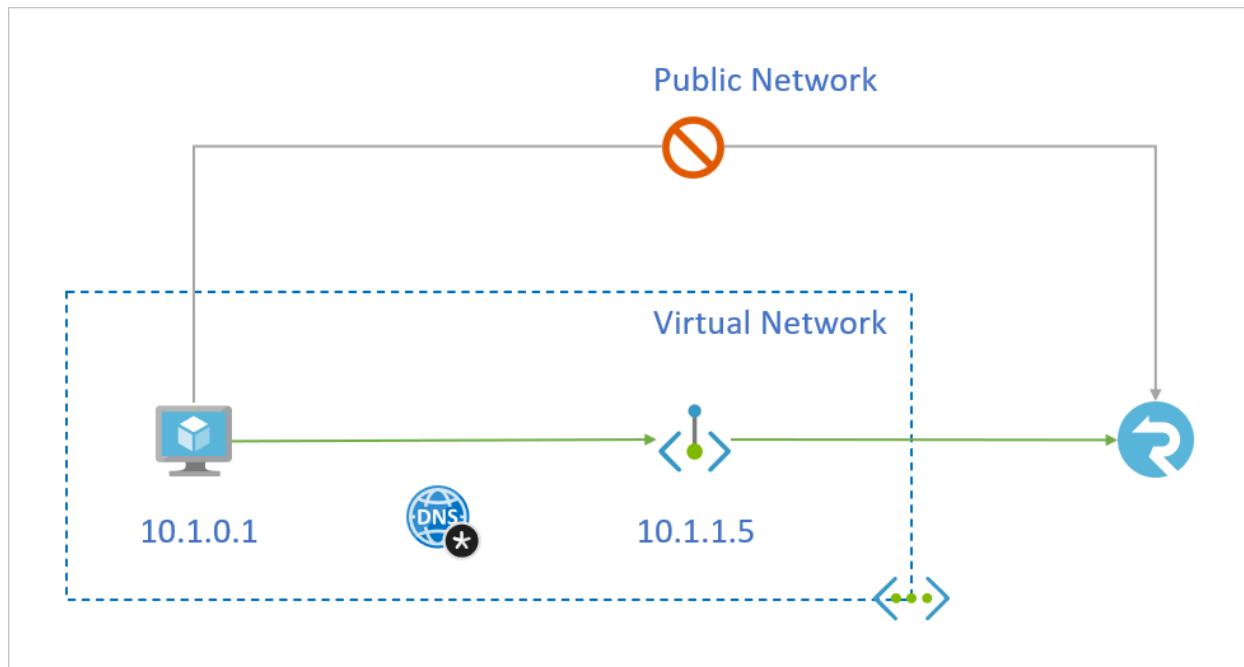
3/5/2021 • 7 minutes to read • [Edit Online](#)

You can use [private endpoints](#) for your Azure SignalR Service to allow clients on a virtual network (VNet) to securely access data over a [Private Link](#). The private endpoint uses an IP address from the VNet address space for your Azure SignalR Service. Network traffic between the clients on the VNet and Azure SignalR Service traverses over a private link on the Microsoft backbone network, eliminating exposure from the public internet.

Using private endpoints for your Azure SignalR Service enables you to:

- Secure your Azure SignalR Service using the network access control to block all connections on the public endpoint for Azure SignalR Service.
- Increase security for the virtual network (VNet), by enabling you to block exfiltration of data from the VNet.
- Securely connect to Azure SignalR Services from on-premises networks that connect to the VNet using [VPN](#) or [ExpressRoutes](#) with private-peering.

## Conceptual overview



A private endpoint is a special network interface for an Azure service in your [Virtual Network](#) (VNet). When you create a private endpoint for your Azure SignalR Service, it provides secure connectivity between clients on your VNet and your service. The private endpoint is assigned an IP address from the IP address range of your VNet. The connection between the private endpoint and Azure SignalR Service uses a secure private link.

Applications in the VNet can connect to Azure SignalR Service over the private endpoint seamlessly, **using the same connection strings and authorization mechanisms that they would use otherwise**. Private endpoints can be used with all protocols supported by the Azure SignalR Service, including REST API.

When you create a private endpoint for an Azure SignalR Service in your VNet, a consent request is sent for approval to the Azure SignalR Service owner. If the user requesting the creation of the private endpoint is also an owner of the Azure SignalR Service, this consent request is automatically approved.

Azure SignalR Service owners can manage consent requests and the private endpoints, through the '*Private endpoints*' tab for the Azure SignalR Service in the [Azure portal](#).

**TIP**

If you want to restrict access to your Azure SignalR Service through the private endpoint only, [configure the Network Access Control](#) to deny or control access through the public endpoint.

## Connecting to private endpoints

Clients on a VNet using the private endpoint should use the same connection string for the Azure SignalR Service, as clients connecting to the public endpoint. We rely upon DNS resolution to automatically route the connections from the VNet to Azure SignalR Service over a private link.

**IMPORTANT**

Use the same connection string to connect to Azure SignalR Service using private endpoints, as you'd use otherwise. Please don't connect to Azure SignalR Service using its `privatelink` subdomain URL.

We create a [private DNS zone](#) attached to the VNet with the necessary updates for the private endpoints, by default. However, if you're using your own DNS server, you may need to make additional changes to your DNS configuration. The section on [DNS changes](#) below describes the updates required for private endpoints.

## DNS changes for private endpoints

When you create a private endpoint, the DNS CNAME resource record for your Azure SignalR Service is updated to an alias in a subdomain with the prefix `privatelink`. By default, we also create a [private DNS zone](#), corresponding to the `privatelink` subdomain, with the DNS A resource records for the private endpoints.

When you resolve your Azure SignalR Service domain name from outside the VNet with the private endpoint, it resolves to the public endpoint of the Azure SignalR Service. When resolved from the VNet hosting the private endpoint, the domain name resolves to the private endpoint's IP address.

For the illustrated example above, the DNS resource records for the Azure SignalR Service 'foobar', when resolved from outside the VNet hosting the private endpoint, will be:

NAME	TYPE	VALUE
<code>foobar.service.signalr.net</code>	CNAME	<code>foobar.privatelink.service.signalr.net</code>
<code>foobar.privatelink.service.signalr.net</code>	A	<Azure SignalR Service public IP address>

As previously mentioned, you can deny or control access for clients outside the VNet through the public endpoint using the network access control.

The DNS resource records for 'foobar', when resolved by a client in the VNet hosting the private endpoint, will be:

NAME	TYPE	VALUE
<code>foobar.service.signalr.net</code>	CNAME	<code>foobar.privatelink.service.signalr.net</code>
<code>foobar.privatelink.service.signalr.net</code>	A	10.1.1.5

This approach enables access to Azure SignalR Service **using the same connection string** for clients on the VNet hosting the private endpoints, as well as clients outside the VNet.

If you are using a custom DNS server on your network, clients must be able to resolve the FQDN for the Azure SignalR Service endpoint to the private endpoint IP address. You should configure your DNS server to delegate your private link subdomain to the private DNS zone for the VNet, or configure the A records for `foobar.privatelink.service.signalr.net` with the private endpoint IP address.

**TIP**

When using a custom or on-premises DNS server, you should configure your DNS server to resolve the Azure SignalR Service name in the `privatelink` subdomain to the private endpoint IP address. You can do this by delegating the `privatelink` subdomain to the private DNS zone of the VNet, or configuring the DNS zone on your DNS server and adding the DNS A records.

The recommended DNS zone name for private endpoints for Azure SignalR Service is:

`privatelink.service.signalr.net`.

For more information on configuring your own DNS server to support private endpoints, refer to the following articles:

- [Name resolution for resources in Azure virtual networks](#)
- [DNS configuration for private endpoints](#)

## Create a private endpoint

### **Create a private endpoint along with a new Azure SignalR Service in the Azure portal**

1. When creating a new Azure SignalR Service, select **Networking** tab. Choose **Private endpoint** as connectivity method.

Home >

# + SignalR

SignalR

[X](#)

\* Basics \* Networking Tags Review + create

**Network connectivity**

You can connect to your SignalR service either publicly, via public IP addresses or privately, using a private endpoint.

**Connectivity method \***

Public endpoint  
 Private endpoint (preview)

**Private endpoint**

Name	Subscription	Resource group	Region	Target resource type	Subnet
Click on add to create a private endpoint					
<a href="#">+ Add</a>					

[Review + create](#) [< Previous : Basics](#) [Next : Tags >](#) [Download a template for automation](#)

2. Click **Add**. Fill in subscription, resource group, location, name for the new private endpoint. Choose a virtual network and subnet.

Home >

# + SignalR

SignalR

[X](#)

\* Basics \* Networking Tags Review + create

**Network connectivity**

You can connect to your SignalR service either publicly, via public IP addresses or privately, using a private endpoint.

**Connectivity method \***

Public endpoint  
 Private endpoint (preview)

**Private endpoint**

Name	Subscription	Resource group	Region
Click on add to create a private endpoint			
<a href="#">+ Add</a>			

**Create private endpoint**

Subscription \*

Resource group \*   
[Create new](#)

Location \* (US East US)

Name \*

SignalR subresource \*  SignalR

**Networking**

To deploy the private endpoint, select a virtual network subnet. [Learn more about private endpoint networking](#)

Virtual network \*

Subnet \*  default (10.0.0.0/24)  
If you have a network security group (NSG) enabled for the subnet above, it will be disabled for private endpoints on this subnet only. Other resources on the subnet will still have NSG enforcement.

**Private DNS integration**

To connect privately with your private endpoint, you need a DNS record. We recommend that you integrate your private endpoint with a private DNS zone. You can also utilize your own DNS servers or create DNS records using the host files on your virtual machines. [Learn more about private DNS integration](#)

Integrate with private DNS zone  Yes  No

Private DNS Zone \* (New) privatelink.service.signalr.net

[Review + create](#) [< Previous : Basics](#) [Next : Tags >](#) [Download a template for automation](#) [OK](#) [Discard](#)

3. Click **Review + create**.

## Create a private endpoint for an existing Azure SignalR Service in the Azure portal

1. Go to the Azure SignalR Service.
2. Click on the settings menu called **Private endpoint connections**.
3. Click the button **+ Private endpoint** on the top.

The screenshot shows the Azure SignalR Service settings page for a resource named 'test'. The left sidebar lists various settings options: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings (Keys, Quickstart, Scale, Settings, CORS, Private endpoint connections, Network access control, Properties, Locks, Export template), and a 'No results' message under Private endpoint connections. The 'Private endpoint connections' option is highlighted. The main area displays a table with columns: Connection name, Connection state, Private Endpoint, and Description. A search bar at the top allows filtering by name and connection state (All connection states). Buttons for creating a new private endpoint (+ Private endpoint), approving (✓ Approve), rejecting (✗ Reject), removing (Remove), and refreshing (Refresh) are also present.

4. Fill in subscription, resource group, resource name and region for the new private endpoint.

## Create a private endpoint

X

**1 Basics**   **2 Resource**   **3 Configuration**   **4 Tags**   **5 Review + create**

Use private endpoints to privately connect to a service or resource. Your private endpoint must be in the same region as your virtual network, but can be in a different region from the private link resource that you are connecting to. [Learn more](#)

### Project details

Subscription \* ⓘ

Resource group \* ⓘ

[Create new](#)

### Instance details

Name \*

Region \*

< Previous

Next : Resource >

5. Choose target Azure SignalR Service resource.

## Create a private endpoint

X

✓ Basics    2 Resource    3 Configuration    4 Tags    5 Review + create

Private Link offers options to create private endpoints for different Azure resources, like your private link service, a SQL server, or an Azure storage account. Select which resource you would like to connect to using this private endpoint. [Learn more](#)

Connection method ⓘ

Connect to an Azure resource in my directory.

Connect to an Azure resource by resource ID or alias.

Subscription \* ⓘ

Resource type \* ⓘ

Microsoft.SignalRService/SignalR

Resource \* ⓘ

Target sub-resource \* ⓘ

signalr

< Previous

Next : Configuration >

6. Choose target virtual network

## Create a private endpoint

X

✓ Basics ✓ Resource 3 Configuration 4 Tags 5 Review + create

### Networking

To deploy the private endpoint, select a virtual network subnet. [Learn more](#)

Virtual network \* ⓘ

Subnet \* ⓘ

ⓘ If you have a network security group (NSG) enabled for the subnet above, it will be disabled for private endpoints on this subnet only. Other resources on the subnet will still have NSG enforcement.

### Private DNS integration

To connect privately with your private endpoint, you need a DNS record. We recommend that you integrate your private endpoint with a private DNS zone. You can also utilize your own DNS servers or create DNS records using the host files on your virtual machines. [Learn more](#)

Integrate with private DNS zone

Yes  No

Configuration name	Subscription	Private DNS zones
privatelink-service-sig...	<input type="text"/>	(New) privatelink.service.signalr.net <input type="text"/>

**Review + create**

< Previous

Next : Tags >

7. Click **Review + create**.

## Create a private endpoint using Azure CLI

1. Login to Azure CLI

```
az login
```

2. Select your Azure Subscription

```
az account set --subscription {AZURE SUBSCRIPTION ID}
```

3. Create a new Resource Group

```
az group create -n {RG} -l {AZURE REGION}
```

4. Register Microsoft.SignalRService as a provider

```
az provider register -n Microsoft.SignalRService
```

5. Create a new Azure SignalR Service

```
az signalr create --name {NAME} --resource-group {RG} --location {AZURE REGION} --sku Standard_S1
```

## 6. Create a Virtual Network

```
az network vnet create --resource-group {RG} --name {vNet NAME} --location {AZURE REGION}
```

## 7. Add a subnet

```
az network vnet subnet create --resource-group {RG} --vnet-name {vNet NAME} --name {subnet NAME} --address-prefixes {addressPrefix}
```

## 8. Disable Virtual Network Policies

```
az network vnet subnet update --name {subnet NAME} --resource-group {RG} --vnet-name {vNet NAME} --disable-private-endpoint-network-policies true
```

## 9. Add a Private DNS Zone

```
az network private-dns zone create --resource-group {RG} --name privatelink.service.signalr.net
```

## 10. Link Private DNS Zone to Virtual Network

```
az network private-dns link vnet create --resource-group {RG} --virtual-network {vNet NAME} --zone-name privatelink.service.signalr.net --name {dnsZoneLinkName} --registration-enabled true
```

## 11. Create a Private Endpoint (Automatically Approve)

```
az network private-endpoint create --resource-group {RG} --vnet-name {vNet NAME} --subnet {subnet NAME} --name {Private Endpoint Name} --private-connection-resource-id "/subscriptions/{AZURE SUBSCRIPTION ID}/resourceGroups/{RG}/providers/Microsoft.SignalRService/SignalR/{NAME}" --group-ids signalr --connection-name {Private Link Connection Name} --location {AZURE REGION}
```

## 12. Create a Private Endpoint (Manually Request Approval)

```
az network private-endpoint create --resource-group {RG} --vnet-name {vNet NAME} --subnet {subnet NAME} --name {Private Endpoint Name} --private-connection-resource-id "/subscriptions/{AZURE SUBSCRIPTION ID}/resourceGroups/{RG}/providers/Microsoft.SignalRService/SignalR/{NAME}" --group-ids signalr --connection-name {Private Link Connection Name} --location {AZURE REGION} --manual-request
```

## 13. Show Connection Status

```
az network private-endpoint show --resource-group {RG} --name {Private Endpoint Name}
```

# Pricing

For pricing details, see [Azure Private Link pricing](#).

# Known Issues

Keep in mind the following known issues about private endpoints for Azure SignalR Service

## **Free tier**

You cannot create any private endpoint for free tier Azure SignalR Service.

## **Access constraints for clients in VNets with private endpoints**

Clients in VNets with existing private endpoints face constraints when accessing other Azure SignalR Service instances that have private endpoints. For instance, suppose a VNet N1 has a private endpoint for an Azure SignalR Service instance S1. If Azure SignalR Service S2 has a private endpoint in a VNet N2, then clients in VNet N1 must also access Azure SignalR Service S2 using a private endpoint. If Azure SignalR Service S2 does not have any private endpoints, then clients in VNet N1 can access Azure SignalR Service in that account without a private endpoint.

This constraint is a result of the DNS changes made when Azure SignalR Service S2 creates a private endpoint.

## **Network Security Group rules for subnets with private endpoints**

Currently, you can't configure [Network Security Group](#) (NSG) rules and user-defined routes for private endpoints. NSG rules applied to the subnet hosting the private endpoint are applied to the private endpoint. A limited workaround for this issue is to implement your access rules for private endpoints on the source subnets, though this approach may require a higher management overhead.

## Next steps

- [Configure Network Access Control](#)

# Configure network access control

11/2/2020 • 2 minutes to read • [Edit Online](#)

Azure SignalR Service enables you to secure and control the level of access to your service endpoint, based on the request type and subset of networks used. When network rules are configured, only applications requesting data over the specified set of networks can access your Azure SignalR Service.

Azure SignalR Service has a public endpoint that is accessible through the internet. You can also create [Private Endpoints for your Azure SignalR Service](#). Private Endpoint assigns a private IP address from your VNet to the Azure SignalR Service, and secures all traffic between your VNet and the Azure SignalR Service over a private link. The Azure SignalR Service network access control provides access control for both public endpoint and private endpoints.

Optionally, you can choose to allow or deny certain types of requests for public endpoint and each private endpoint. For example, you can block all [Server Connections](#) from public endpoint and make sure they only originate from a specific VNet.

An application that accesses an Azure SignalR Service when network access control rules are in effect still requires proper authorization for the request.

## Scenario A - No public traffic

To completely deny all public traffic, you should first configure the public network rule to allow no request type. Then, you should configure rules that grant access to traffic from specific VNets. This configuration enables you to build a secure network boundary for your applications.

## Scenario B - Only client connections from public network

In this scenario, you can configure the public network rule to only allow [Client Connections](#) from public network. You can then configure private network rules to allow other types of requests originating from a specific VNet. This configuration hides your app servers from public network and establishes secure connections between your app servers and Azure SignalR Service.

## Managing network access control

You can manage network access control for Azure SignalR Service through the Azure portal.

### Azure portal

1. Go to the Azure SignalR Service you want to secure.
2. Click on the settings menu called **Network access control**.

Home >

## test | Network access control

SignalR

Search (Ctrl+ /) Save Discard

Default action ⓘ

Allow Deny

Public network

Allow ⓘ

None ▾

Private endpoint connections

Connection name	Allow
test.3f54d095-c826-4991-9e18-8c800b6c1bb5	Server connection, Client connection, REST API ▾

Settings

- Keys
- Quickstart
- Scale
- Settings
- CORS
- Private endpoint connections
- Network access control**
- Properties
- Locks
- Export template

3. To edit default action, toggle the Allow/Deny button.

**TIP**

Default action is the action we take when there is no ACL rule matches. For example, if the default action is **Deny**, then request types that are not explicitly approved below will be denied.

4. To edit public network rule, select allowed types of requests under **Public network**.

Home >

## test | Network access control

SignalR

Search (Ctrl+ /) Save Discard

Default action ⓘ

Allow Deny

Public network

Allow ⓘ

None ▾

Select all

Client connection

Server connection

REST API

Allow

9e18-8c800b6c1bb5	Server connection, Client connection, REST API ▾
-------------------	--

Settings

- Keys
- Quickstart
- Scale
- Settings
- CORS
- Private endpoint connections
- Network access control**
- Properties
- Locks
- Export template

5. To edit private endpoint network rules, select allowed types of requests in each row under **Private endpoint connections**.

The screenshot shows the Azure portal interface for managing network access control. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Keys, Quickstart, Scale, Settings, CORS, Private endpoint connections, Network access control (which is selected and highlighted in grey), Properties, Locks, and Export template. The main area has tabs for Default action (Allow is selected) and Public network. Under Private endpoint connections, there's a table with one row. The row has columns for Connection name (test.3f54d095-c826-4991-9e18-8c800b6c1bb5), Allow, and a dropdown menu. The Allow column contains the text "Server connection, Client connection, REST API". The dropdown menu is open, showing four items: "Select all" (checked), "Client connection" (checked), "Server connection" (checked), and "REST API" (checked). There are also "Save" and "Discard" buttons at the top.

6. Click **Save** to apply your changes.

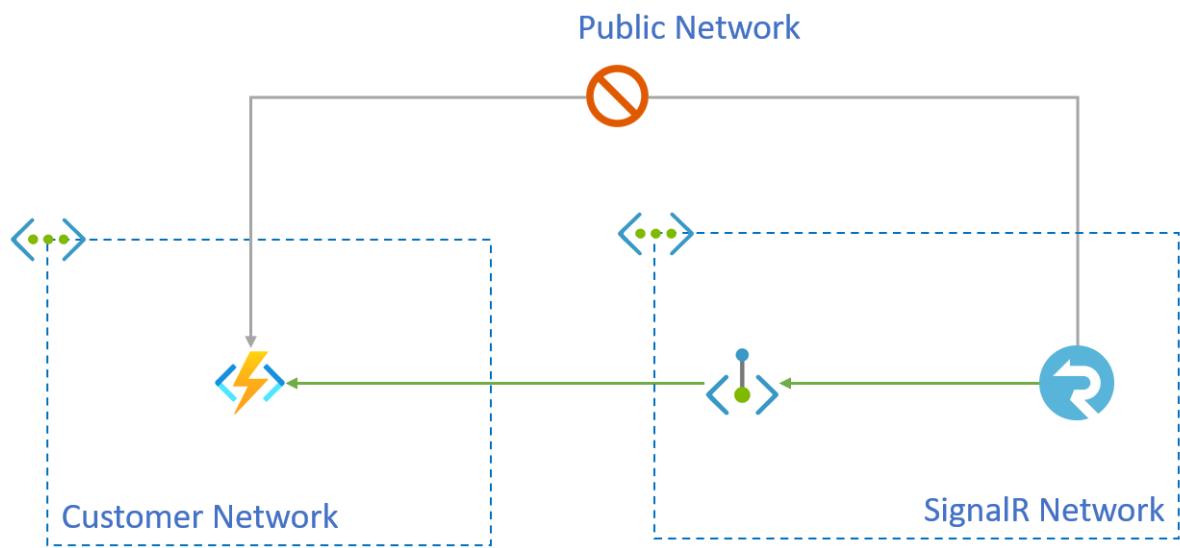
## Next steps

Learn more about [Azure Private Link](#).

# Secure Azure SignalR outbound traffic through Shared Private Endpoints

7/15/2021 • 3 minutes to read • [Edit Online](#)

If you're using [serverless mode](#) in Azure SignalR Service, you might have outbound traffic to upstream. Upstream such as Azure Web App and Azure Functions, can be configured to accept connections from a list of virtual networks and refuse outside connections that originate from a public network. You can create an outbound [private endpoint connection](#) to reach these endpoints.



This outbound method is subject to the following requirements:

- The upstream must be Azure Web App or Azure Function.
- The Azure SignalR Service service must be on the Standard tier.
- The Azure Web App or Azure Function must be on certain SKUs. See [Use Private Endpoints for Azure Web App](#).

## Shared Private Link Resources Management APIs

Private endpoints of secured resources that are created through Azure SignalR Service APIs are referred to as *shared private link resources*. This is because you're "sharing" access to a resource, such as an Azure Function, that has been integrated with the [Azure Private Link service](#). These private endpoints are created inside Azure SignalR Service execution environment and are not directly visible to you.

At this moment, you can use Management REST API to create or delete *shared private link resources*. In the remainder of this article, we will use [Azure CLI](#) to demonstrate the REST API calls.

## NOTE

The examples in this article are based on the following assumptions:

- The resource ID of this Azure SignalR Service is `/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/contoso/providers/Microsoft.SignalRService/signalr/contoso-signalr`.
- The resource ID of upstream Azure Function is `/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/contoso/providers/Microsoft.Web/sites/contoso-func`.

The rest of the examples show how the `contoso-signalr` service can be configured so that its upstream calls to function go through a private endpoint rather than public network.

### Step 1: Create a shared private link resource to the function

You can make the following API call with the [Azure CLI](#) to create a shared private link resource:

```
az rest --method put --uri https://management.azure.com/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/contoso/providers/Microsoft.SignalRService/signalr/contoso-signalr/sharedPrivateLinkResources/func-pe?api-version=2021-06-01-preview --body @create-pe.json
```

The contents of the `create-pe.json` file, which represent the request body to the API, are as follows:

```
{
  "name": "func-pe",
  "properties": {
    "privateLinkResourceId": "/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/contoso/providers/Microsoft.Web/sites/contoso-func",
    "groupId": "sites",
    "requestMessage": "please approve"
  }
}
```

The process of creating an outbound private endpoint is a long-running (asynchronous) operation. As in all asynchronous Azure operations, the `PUT` call returns an `Azure-AsyncOperation` header value that looks like the following:

```
"Azure-AsyncOperation": "https://management.azure.com/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/contoso/providers/Microsoft.SignalRService/signalr/contoso-signalr/operationStatuses/c0786383-8d5f-4554-8d17-f16fcf482fb2?api-version=2021-06-01-preview"
```

You can poll this URI periodically to obtain the status of the operation.

If you are using the CLI, you can poll for the status by manually querying the `Azure-AsyncOperationHeader` value,

```
az rest --method get --uri https://management.azure.com/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/contoso/providers/Microsoft.SignalRService/signalr/contoso-signalr/operationStatuses/c0786383-8d5f-4554-8d17-f16fcf482fb2?api-version=2021-06-01-preview
```

Wait until the status changes to "Succeeded" before proceeding to the next steps.

### Step 2a: Approve the private endpoint connection for the function

## NOTE

In this section, you use the Azure portal to walk through the approval flow for a private endpoint to Azure Function. Alternately, you could use the [REST API](#) that's available via the App Service provider.

## IMPORTANT

After you approved the private endpoint connection, the Function is no longer accessible from public network. You may need to create other private endpoints in your own virtual network to access the Function endpoint.

1. In the Azure portal, select the **Networking** tab of your Function App and navigate to **Private endpoint connections**. Click **Configure your private endpoint connections**. After the asynchronous operation has succeeded, there should be a request for a private endpoint connection with the request message from the previous API call.

Connection name ↑	Connection state ↑↓	Private endpoint ↑↓	Description
contoso-signalr-pe-e48be3cd-ce29-4035-89cd-5...	Pending	contoso-signalr-pe-5be807ad-8b6f-45bc-a1fb-2c...	please approve

2. Select the private endpoint that Azure SignalR Service created. In the **Private endpoint** column, identify the private endpoint connection by the name that's specified in the previous API, select **Approve**.

Make sure that the private endpoint connection appears as shown in the following screenshot. It could take one to two minutes for the status to be updated in the portal.

Connection name ↑	Connection state ↑↓	Private endpoint ↑↓	Description
contoso-signalr-pe-e48be3cd-ce29-4035-89cd-5...	Approved	contoso-signalr-pe-5be807ad-8b6f-45bc-a1fb-2c...	please approve

## Step 2b: Query the status of the shared private link resource

It takes minutes for the approval to be propagated to Azure SignalR Service. To confirm that the shared private link resource has been updated after approval, you can also obtain the "Connection state" by using the GET API.

```
az rest --method get --uri https://management.azure.com/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/contoso/providers/Microsoft.SignalRService/signalr/contoso-signalr/sharedPrivateLinkResources/func-pe?api-version=2021-06-01-preview
```

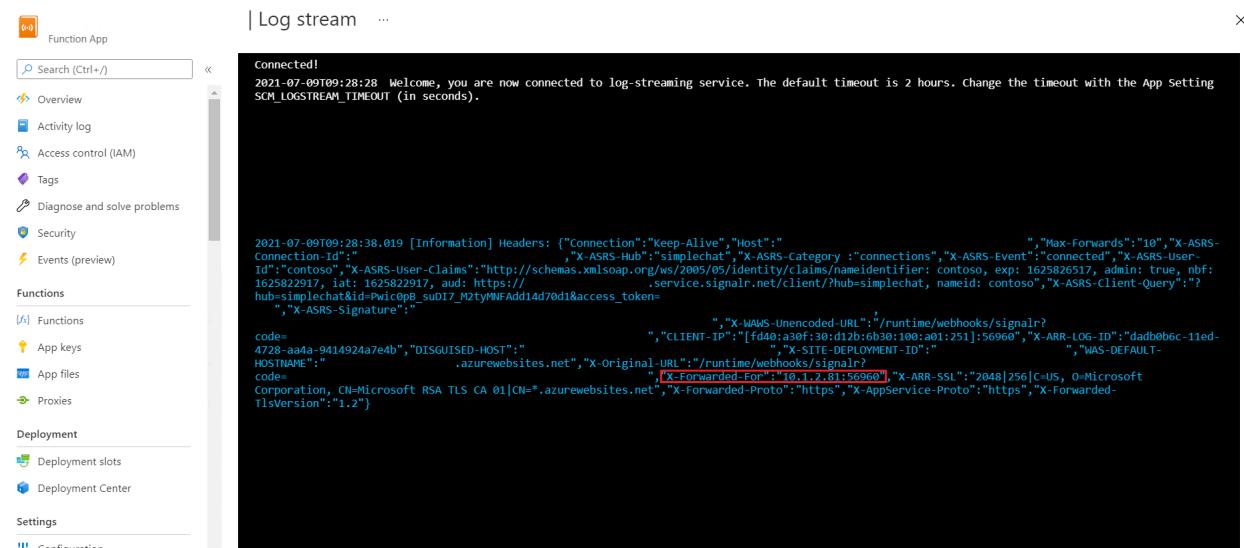
This would return a JSON, where the connection state would show up as "status" under the "properties" section.

```
{
  "name": "func-pe",
  "properties": {
    "privateLinkResourceId": "/subscriptions/00000000-0000-0000-0000-
000000000000/resourceGroups/contoso/providers/Microsoft.Web/sites/contoso-func",
    "groupId": "sites",
    "requestMessage": "please approve",
    "status": "Approved",
    "provisioningState": "Succeeded"
  }
}
```

If the "Provisioning State" (`properties.provisioningState`) of the resource is `Succeeded` and "Connection State" (`properties.status`) is `Approved`, it means that the shared private link resource is functional and Azure SignalR Service can communicate over the private endpoint.

### Step 3: Verify upstream calls are from a private IP

Once the private endpoint is set up, you can verify incoming calls are from a private IP by checking the `X-Forwarded-For` header at upstream side.



## Next steps

Learn more about private endpoints:

- [What are private endpoints?](#)

# Managed identities for Azure SignalR Service

5/17/2021 • 4 minutes to read • [Edit Online](#)

This article shows you how to create a managed identity for Azure SignalR Service and how to use it in serverless scenarios.

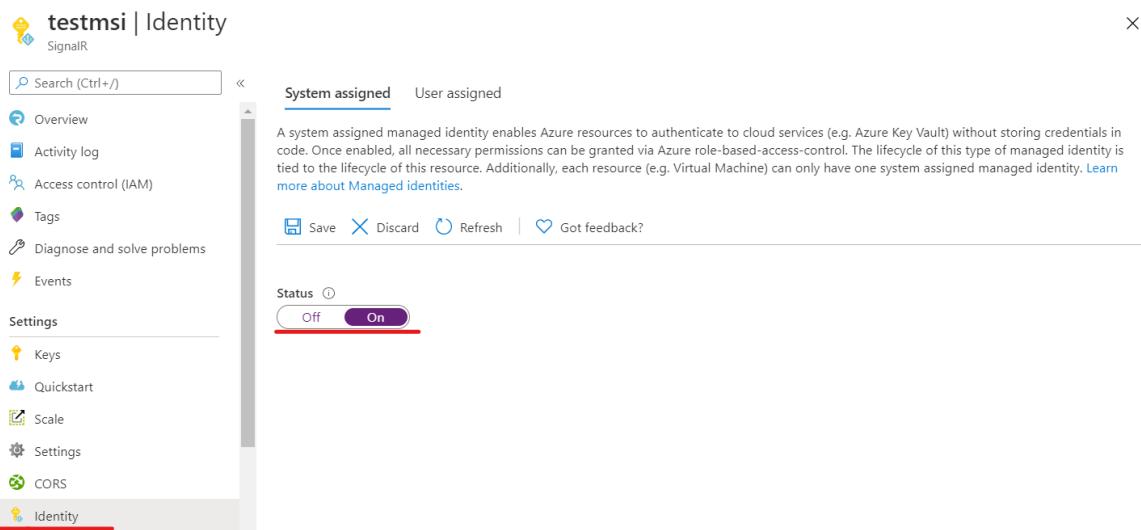
## IMPORTANT

Azure SignalR Service can support only one managed identity. That means you can add either a system-assigned identity or a user-assigned identity.

## Add a system-assigned identity

To set up a managed identity in the Azure portal, you'll first create an Azure SignalR Service instance and then enable the feature.

1. Create an Azure SignalR Service instance in the portal as you normally would. Browse to it in the portal.
2. Select **Identity**.
3. On the **System assigned** tab, switch **Status** to **On**. Select **Save**.



## Add a user-assigned identity

Creating an Azure SignalR Service instance with a user-assigned identity requires that you create the identity and then add its resource identifier to your service.

1. Create a user-assigned managed identity resource according to [these instructions](#).
2. Create an Azure SignalR Service instance in the portal as you normally would. Browse to it in the portal.
3. Select **Identity**.
4. On the **User assigned** tab, select **Add**.
5. Search for the identity that you created earlier and selects it. Select **Add**.

## Use a managed identity in serverless scenarios

Azure SignalR Service is a fully managed service, so you can't use a managed identity to get tokens manually. Instead, Azure SignalR Service uses the managed identity that you set to obtain an access token. The service then sets the access token into an `Authorization` header in an upstream request in serverless scenarios.

### Enable managed identity authentication in upstream settings

1. Add a system-assigned identity or user-assigned identity.
2. Add one Upstream Setting and click any asterisk to get into a detailed page as shown below.

**Upstream Settings (Preview)**  

Configure Upstream Settings

Upstream URL Pattern \* ⓘ  
https://[REDACTED]/runtime/webhooks/signalr?code=[REDACTED]

Hub Rules \* ⓘ  
\*

Event Rules \* ⓘ  
\*

Category Rules \* ⓘ  
\*

Upstream Authentication ⓘ  
 No Authentication  
 Use Managed Identity

Auth Resource ID  
 Use default value  
 Select from existing Applications  
 Specify

https://[REDACTED].azurewebsites.net

 Here lists the Resource ID for the Azure Services supporting Azure AD Authentication

3. In the managed identity authentication settings, for **Resource**, you can specify the target resource. The resource will become an `aud` claim in the obtained access token, which can be used as a part of validation in your upstream endpoints. The resource can be one of the following:

- Empty
- Application (client) ID of the service principal
- Application ID URI of the service principal
- [Resource ID of an Azure service](#)

**NOTE**

If you validate an access token by yourself in your service, you can choose any one of the resource formats. Just make sure that the **Resource** value in **Auth** settings and the validation are consistent. If you use Azure role-based access control (Azure RBAC) for a data plane, you must use the resource that the service provider requests.

## Validate access tokens

The token in the `Authorization` header is a [Microsoft identity platform access token](#).

To validate access tokens, your app should also validate the audience and the signing tokens. These need to be validated against the values in the OpenID discovery document. For example, see the [tenant-independent version of the document](#).

The Azure Active Directory (Azure AD) middleware has built-in capabilities for validating access tokens. You can browse through our [samples](#) to find one in the language of your choice.

We provide libraries and code samples that show how to handle token validation. There are also several open-source partner libraries available for JSON Web Token (JWT) validation. There's at least one option for almost every platform and language out there. For more information about Azure AD authentication libraries and code samples, see [Microsoft identity platform authentication libraries](#).

### Authentication in Function App

Setting access token validation in Function App is easy and efficient without code works.

1. In the **Authentication (classic)** page, switch **App Service Authentication** to **On**.

2. Select Log in with Azure Active Directory in Action to take when request is not authenticated.
3. In the Authentication Provider, click into Azure Active Directory
4. In the new page. Select Express and Create New AD App and then click OK

**Azure Active Directory Settings**

### Active Directory Authentication

These settings allow users to sign in with Azure Active Directory. Click here to learn more. [Learn more](#)

Management mode  Off  Express  Advanced

**i** Express mode allows user to create an AD Application or select an existing AD application in your current Active Dir

Current Active Directory	Microsoft
Management mode	<input checked="" type="button"/> Create New AD App <input type="button"/> Select Existing AD App
Create App *	myFunctionApp
Grant Common Data Services Permissions	<input type="radio"/> On <input checked="" type="radio"/> Off

**OK**

5. Navigate to SignalR Service and follow [steps](#) to add a system-assigned identity or user-assigned identity.
6. Get into **Upstream settings** in SignalR Service and choose **Use Managed Identity** and **Select from existing Applications**. Select the application you created previously.

After these settings, the Function App will reject requests without an access token in the header.

#### IMPORTANT

To pass the authentication, the *Issuer Url* must match the *iss* claim in token. Currently, we only support v1 endpoint (see [v1.0 and v2.0](#)), so the *Issuer Url* should look like <https://sts.windows.net/<tenant-id>/>. Check the *Issuer Url* configured in Azure Function. For **Authentication**, go to *Identity provider* -> *Edit* -> *Issuer Url* and for **Authentication (classic)**, go to *Azure Active Directory* -> *Advanced* -> *Issuer Url*

## Use a managed identity for Key Vault reference

SignalR Service can access Key Vault to get secret using the managed identity.

1. Add a system-assigned identity or user-assigned identity for Azure SignalR Service.

2. Grant secret read permission for the managed identity in the Access policies in the Key Vault. See [Assign a Key Vault access policy using the Azure portal](#)

Currently, this feature can be used in the following scenarios:

- [Reference secret in Upstream URL Pattern](#)
- [Azure Functions development and configuration with Azure SignalR Service](#)

## Next steps

# Build real-time Apps with Azure Functions and Azure SignalR Service

11/2/2020 • 2 minutes to read • [Edit Online](#)

Because Azure SignalR Service and Azure Functions are both fully managed, highly scalable services that allow you to focus on building applications instead of managing infrastructure, it's common to use the two services together to provide real-time communications in a [serverless](#) environment.

## NOTE

Learn to use SignalR and Azure Functions together in the interactive tutorial [Enable automatic updates in a web application using Azure Functions and SignalR Service](#).

## Integrate real-time communications with Azure services

Azure Functions allow you to write code in [several languages](#), including JavaScript, Python, C#, and Java, that triggers whenever events occur in the cloud. Examples of these events include:

- HTTP and webhook requests
- Periodic timers
- Events from Azure services, such as:
  - Event Grid
  - Event Hubs
  - Service Bus
  - Cosmos DB change feed
  - Storage - blobs and queues
  - Logic Apps connectors such as Salesforce and SQL Server

By using Azure Functions to integrate these events with Azure SignalR Service, you have the ability to notify thousands of clients whenever events occur.

Some common scenarios for real-time serverless messaging that you can implement with Azure Functions and SignalR Service include:

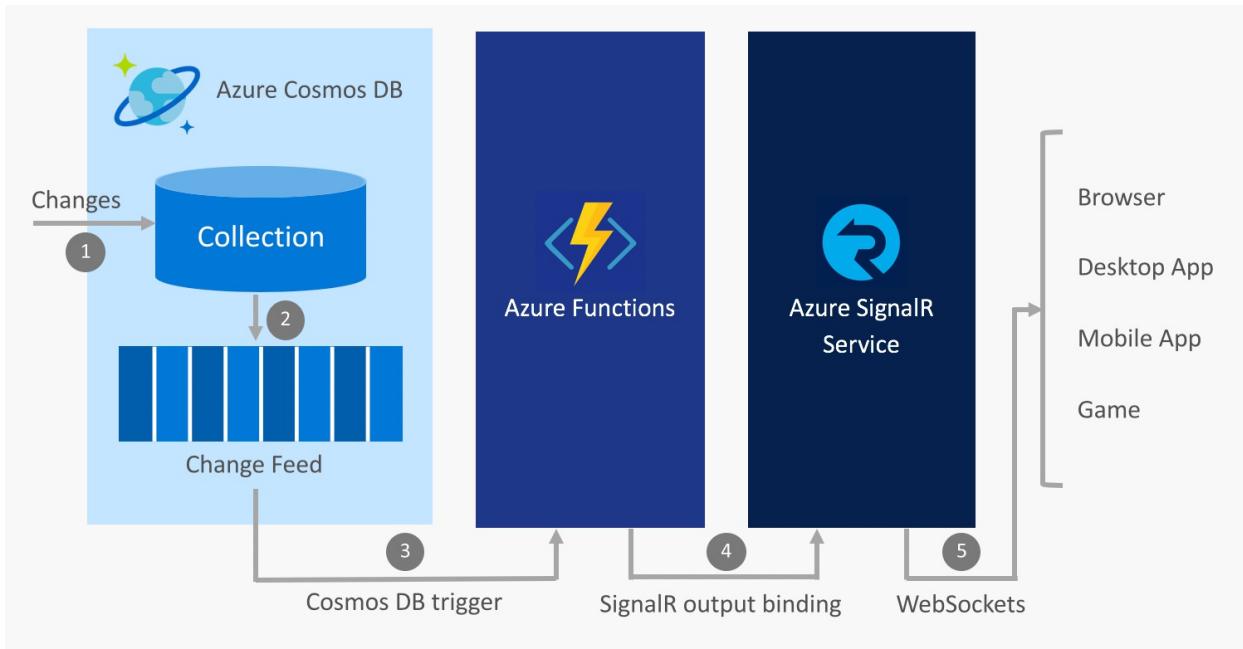
- Visualize IoT device telemetry on a real-time dashboard or map
- Update data in an application when documents update in Cosmos DB
- Send in-app notifications when new orders are created in Salesforce

## SignalR Service bindings for Azure Functions

The SignalR Service bindings for Azure Functions allow an Azure Function app to publish messages to clients connected to SignalR Service. Clients can connect to the service using a SignalR client SDK that is available in .NET, JavaScript, and Java, with more languages coming soon.

### An example scenario

An example of how to use the SignalR Service bindings is using Azure Functions to integrate with Azure Cosmos DB and SignalR Service to send real-time messages when new events appear on a Cosmos DB change feed.



1. A change is made in a Cosmos DB collection
2. The change event is propagated to the Cosmos DB change feed
3. An Azure Functions is triggered by the change event using the Cosmos DB trigger
4. The SignalR Service output binding publishes a message to SignalR Service
5. SignalR Service publishes the message to all connected clients

#### Authentication and users

SignalR Service allows you to broadcast messages to all clients or only to a subset of clients, such as those belonging to a single user. The SignalR Service bindings for Azure Functions can be combined with App Service Authentication to authenticate users with providers such as Azure Active Directory, Facebook, and Twitter. You can then send messages directly to these authenticated users.

## Next steps

In this article, you got an overview of how to use Azure Functions with SignalR Service to enable a wide array of serverless real-time messaging scenarios.

For full details on how to use Azure Functions and SignalR Service together visit the following resources:

- [Azure Functions development and configuration with SignalR Service](#)
- [Enable automatic updates in a web application using Azure Functions and SignalR Service](#)

Follow one of these quickstarts to learn more.

- [Azure SignalR Service Serverless Quickstart - C#](#)
- [Azure SignalR Service Serverless Quickstart - JavaScript](#)

# How to send events from Azure SignalR Service to Event Grid

4/21/2021 • 4 minutes to read • [Edit Online](#)

Azure Event Grid is a fully managed event routing service that provides uniform event consumption using a pub-sub model. In this guide, you use the Azure CLI to create an Azure SignalR Service, subscribe to connection events, then deploy a sample web application to receive the events. Finally, you can connect and disconnect and see the event payload in the sample application.

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

## Prerequisites

- Use the Bash environment in [Azure Cloud Shell](#).

 [Launch Cloud Shell](#)

- If you prefer, [install](#) the Azure CLI to run CLI reference commands.
  - If you're using a local installation, sign in to the Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. For additional sign-in options, see [Sign in with the Azure CLI](#).
  - When you're prompted, install Azure CLI extensions on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
  - Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).
- The Azure CLI commands in this article are formatted for the **Bash** shell. If you're using a different shell like PowerShell or Command Prompt, you may need to adjust line continuation characters or variable assignment lines accordingly. This article uses variables to minimize the amount of command editing required.

## Create a resource group

An Azure resource group is a logical container in which you deploy and manage your Azure resources. The following [az group create](#) command creates a resource group named *myResourceGroup* in the *eastus* region. If you want to use a different name for your resource group, set `RESOURCE_GROUP_NAME` to a different value.

```
RESOURCE_GROUP_NAME=myResourceGroup

az group create --name $RESOURCE_GROUP_NAME --location eastus
```

## Create a SignalR Service

Next, deploy an Azure SignalR Service into the resource group with the following commands.

```
SIGNALR_NAME=SignalRTTestSvc

az signalr create --resource-group $RESOURCE_GROUP_NAME --name $SIGNALR_NAME --sku Free_F1
```

Once the SignalR Service has been created, the Azure CLI returns output similar to the following:

```
{
  "externalIp": "13.76.156.152",
  "hostName": "clitest.servicedev.signalr.net",
  "hostNamePrefix": "clitest",
  "id": "/subscriptions/28cf13e2-c598-4aa9-b8c8-098441f0827a/resourceGroups/clitest1/providers/Microsoft.SignalRService/SignalR/clitest",
  "location": "southeastasia",
  "name": "clitest",
  "provisioningState": "Succeeded",
  "publicPort": 443,
  "resourceGroup": "clitest1",
  "serverPort": 443,
  "sku": {
    "capacity": 1,
    "family": null,
    "name": "Free_F1",
    "size": "F1",
    "tier": "Free"
  },
  "tags": null,
  "type": "Microsoft.SignalRService/SignalR",
  "version": "1.0"
}
```

## Create an event endpoint

In this section, you use a Resource Manager template located in a GitHub repository to deploy a pre-built sample web application to Azure App Service. Later, you subscribe to your registry's Event Grid events and specify this app as the endpoint to which the events are sent.

To deploy the sample app, set `SITE_NAME` to a unique name for your web app, and execute the following commands. The site name must be unique within Azure because it forms part of the fully qualified domain name (FQDN) of the web app. In a later section, you navigate to the app's FQDN in a web browser to view your registry's events.

```
SITE_NAME=<your-site-name>

az deployment group create \
  --resource-group $RESOURCE_GROUP_NAME \
  --template-uri "https://raw.githubusercontent.com/Azure-Samples/azure-event-grid-viewer/master/azuredeploy.json" \
  --parameters siteName=$SITE_NAME hostingPlanName=$SITE_NAME-plan
```

Once the deployment succeeds (it might take a few minutes), open a browser and navigate to your web app to make sure it's running:

```
http://<your-site-name>.azurewebsites.net
```

## Enable the Event Grid resource provider

If you haven't previously used Event Grid in your Azure subscription, you might need to register the Event Grid

resource provider. Run the following command to register the provider:

```
az provider register --namespace Microsoft.EventGrid
```

It might take a moment for the registration to finish. To check the status, run:

```
az provider show --namespace Microsoft.EventGrid --query "registrationState"
```

When `registrationState` is `Registered`, you're ready to continue.

## Subscribe to registry events

In Event Grid, you subscribe to a *topic* to tell it which events you want to track, and where to send them. The following [az eventgrid event-subscription create](#) command subscribes to the Azure SignalR Service you created, and specifies your web app's URL as the endpoint to which it should send events. The environment variables you populated in earlier sections are reused here, so no edits are required.

```
SIGNALR_SERVICE_ID=$(az signalr show --resource-group $RESOURCE_GROUP_NAME --name $SIGNALR_NAME --query id -o tsv)
APP_ENDPOINT=https://$SITE_NAME.azurewebsites.net/api/updates

az eventgrid event-subscription create \
--name event-sub-signalr \
--source-resource-id $SIGNALR_SERVICE_ID \
--endpoint $APP_ENDPOINT
```

When the subscription is completed, you should see output similar to the following:

```
{
  "deadLetterDestination": null,
  "destination": {
    "endpointBaseUrl": "https://$SITE_NAME.azurewebsites.net/api/updates",
    "endpointType": "WebHook",
    "endpointUrl": null
  },
  "filter": {
    "includedEventTypes": [
      "Microsoft.SignalRService.ClientConnectionConnected",
      "Microsoft.SignalRService.ClientConnectionDisconnected"
    ],
    "isSubjectCaseSensitive": null,
    "subjectBeginsWith": "",
    "subjectEndsWith": ""
  },
  "id": "/subscriptions/28cf13e2-c598-4aa9-b8c8-098441f0827a/resourceGroups/myResourceGroup/providers/Microsoft.SignalRService/SignalR/SignalRTTestSvc/providers/Microsoft.EventGrid/eventSubscriptions/event-sub-signalr",
  "labels": null,
  "name": "event-sub-signalr",
  "provisioningState": "Succeeded",
  "resourceGroup": "myResourceGroup",
  "retryPolicy": {
    "eventTimeToLiveInMinutes": 1440,
    "maxDeliveryAttempts": 30
  },
  "topic": "/subscriptions/28cf13e2-c598-4aa9-b8c8-098441f0827a/resourceGroups/myResourceGroup/providers/microsoft.signalrservice/signalr/SignalRTTestSvc",
  "type": "Microsoft.EventGrid/eventSubscriptions"
}
```

## Trigger registry events

Switch to the service mode to [Serverless Mode](#) and setup a client connection to the SignalR Service. You can take [Serverless Sample](#) as a reference.

```
git clone git@github.com:aspnet/AzureSignalR-samples.git

cd samples/Management

# Start the negotiation server
# Negotiation server is responsible for generating access token for clients
cd NegotiationServer
dotnet user-secrets set Azure:SignalR:ConnectionString "<Connection String>"
dotnet run

# Use a separate command line
# Start a client
cd SignalRClient
dotnet run
```

## View registry events

You have now connected a client to the SignalR Service. Navigate to your Event Grid Viewer web app, and you should see a [ClientConnectionConnected](#) event. If you terminate the client, you will also see a [ClientConnectionDisconnected](#) event.

# Resource logs for Azure SignalR Service

3/5/2021 • 6 minutes to read • [Edit Online](#)

This tutorial discusses what resource logs for Azure SignalR Service are, how to set them up, and how to troubleshoot with them.

## Prerequisites

To enable resource logs, you'll need somewhere to store your log data. This tutorial uses Azure Storage and Log Analytics.

- [Azure storage](#) - Retains resource logs for policy audit, static analysis, or backup.
- [Log Analytics](#) - A flexible log search and analytics tool that allows for analysis of raw logs generated by an Azure resource.

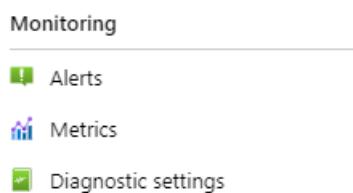
## Set up resource logs for an Azure SignalR Service

You can view resource logs for Azure SignalR Service. These logs provide richer view of connectivity to your Azure SignalR Service instance. The resource logs provide detailed information of every connection. For example, basic information (user ID, connection ID and transport type and so on) and event information (connect, disconnect and abort event and so on) of the connection. resource logs can be used for issue identification, connection tracking and analysis.

### Enable resource logs

Resource logs are disabled by default. To enable resource logs, follow these steps:

1. In the [Azure portal](#), under **Monitoring**, click **Diagnostic settings**.



2. Then click **Add diagnostic setting**.

[+ Add diagnostic setting](#)

3. Set the archive target that you want. Currently, we support **Archive to a storage account** and **Send to Log Analytics**.

4. Select the logs you want to archive.

## Diagnostics settings

Save Discard Delete

Name  
wanl

Archive to a storage account

Storage account



Stream to an event hub

Send to Log Analytics

Subscription



Log Analytics Workspace



log

AllLogs

Retention (days)



0

metric

Traffic

Retention (days)



0

Errors

Retention (days)



0



Retention only applies to storage account.

### 5. Save the new diagnostics settings.

New settings take effect in about 10 minutes. After that, logs appear in the configured archival target, in the **Diagnostics logs** pane.

For more information about configuring diagnostics, see the [overview of Azure resource logs](#).

### Resource logs categories

Azure SignalR Service captures resource logs in one category:

- **All Logs:** Track connections that connect to Azure SignalR Service. The logs provide information about the connect/disconnect, authentication and throttling. For more information, see the next section.

### Archive to a storage account

Logs are stored in the storage account that configured in **Diagnostics logs** pane. A container named `insights-logs-alllogs` is created automatically to store resource logs. Inside the container, logs are stored in the file

```
resourceId=/SUBSCRIPTIONS/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXX/RESOURCEGROUPS/XXXX/PROVIDERS/MICROSOFT.SIGNALRSERVICE/SIGNALR/XXX/y=YYYY/m=MM/d=DD/h=HH/m=00/PT1H.json
```

. Basically, the path is combined by `resource ID` and `Date Time`. The log files are split by `hour`. Therefore, the minutes always be `m=00`.

All logs are stored in JavaScript Object Notation (JSON) format. Each entry has string fields that use the format described in the following sections.

Archive log JSON strings include elements listed in the following tables:

### Format

NAME	DESCRIPTION
time	Log event time
level	Log event level
resourceId	Resource ID of your Azure SignalR Service
location	Location of your Azure SignalR Service
category	Category of the log event
operationName	Operation name of the event
callerIpAddress	IP address of your server/client
properties	Detailed properties related to this log event. For more detail, see the properties table below

### Properties Table

NAME	DESCRIPTION
type	Type of the log event. Currently, we provide information about connectivity to the Azure SignalR Service. Only <code>ConnectivityLogs</code> type is available
collection	Collection of the log event. Allowed values are: <code>Connection</code> , <code>Authorization</code> and <code>Throttling</code>
connectionId	Identity of the connection
transportType	Transport type of the connection. Allowed values are: <code>Websockets</code>   <code>ServerSentEvents</code>   <code>LongPolling</code>
connectionType	Type of the connection. Allowed values are: <code>Server</code>   <code>Client</code> . <code>Server</code> : connection from server side; <code>Client</code> : connection from client side
userId	Identity of the user
message	Detailed message of log event

The following code is an example of an archive log JSON string:

```
{
  "properties": {
    "message": "Entered Serverless mode.",
    "type": "ConnectivityLogs",
    "collection": "Connection",
    "connectionId": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
    "userId": "User",
    "transportType": "WebSockets",
    "connectionType": "Client"
  },
  "operationName": "ServerlessModeEntered",
  "category": "AllLogs",
  "level": "Informational",
  "callerIpAddress": "xxx.xxx.xxx.xxx",
  "time": "2019-01-01T00:00:00Z",
  "resourceId": "/SUBSCRIPTIONS/XXXXXX-XXXX-XXXX-XXXX-
XXXXXXXXXX/RESOURCEGROUPS/XXXX/PROVIDERS/MICROSOFT.SIGNALRSERVICE/SIGNALR/XXX",
  "location": "xxxx"
}
}
```

## Archive logs schema for Log Analytics

To view resource logs, follow these steps:

1. Click **Logs** in your target Log Analytics.

### General

- Quick Start
- Workspace summary
- View Designer
- Workbooks
- Logs

2. Enter **SignalRServiceDiagnosticLogs** and select time range to query resource logs. For advanced query, see [Get started with Log Analytics in Azure Monitor](#)

The screenshot shows the Azure Log Analytics interface with the following details:

- Run** button and **Last 7 days** time range selector.
- Save**, **Copy**, **Export**, **New alert rule**, and **Pin to dashboard** buttons.
- SignalRServiceDiagnosticLogs** table results table.
- Table columns: TimeGenerated (UTC), Collection, OperationName, Location, Level, CallerIpAddress, Message, UserId, ConnectionId, ConnectionType, TransportType, Type, \_ResourceId.
- Single row displayed:
 

TimeGenerated (UTC)	2019-11-22T10:05:15Z	Collection	Connection	Location	westcentralus	Level	Informational	CallerIpAddress	Message	UserId	ConnectionId	ConnectionType	TransportType	Type	_ResourceId
...									Connection started	User		Client	WebSockets	SignalRServiceDiagnosticLogs	/subscriptions/...
TenantId	[redacted]														
SourceSystem	Azure														
...															
TimeGenerated (UTC)	2019-11-22T10:05:15Z														
Collection	Connection														
OperationName	ConnectionStarted														
Location	westcentralus														
Level	Informational														
CallerIpAddress	[redacted]														
Message	Connection started														
UserId	User														
ConnectionId	[redacted]														
ConnectionType	Client														
TransportType	WebSockets														
Type	SignalRServiceDiagnosticLogs														
_ResourceId	[redacted]														

Archive log columns include elements listed in the following table:

NAME	DESCRIPTION
TimeGenerated	Log event time

NAME	DESCRIPTION
Collection	Collection of the log event. Allowed values are: <code>Connection</code> , <code>Authorization</code> and <code>Throttling</code>
OperationName	Operation name of the event
Location	Location of your Azure SignalR Service
Level	Log event level
CallerIpAddress	IP address of your server/client
Message	Detailed message of log event
UserId	Identity of the user
ConnectionId	Identity of the connection
ConnectionType	Type of the connection. Allowed values are: <code>Server</code>   <code>Client</code> . <code>Server</code> : connection from server side; <code>Client</code> : connection from client side
TransportType	Transport type of the connection. Allowed values are: <code>Websockets</code>   <code>ServerSentEvents</code>   <code>LongPolling</code>

## Troubleshooting with resource logs

To troubleshoot for Azure SignalR Service, you can enable server/client side logs to capture failures. At present, Azure SignalR Service exposes resource logs, you can also enable logs for service side.

When encountering connection unexpected growing or dropping situation, you can take advantage of resource logs to troubleshoot.

Typical issues are often about connections' unexpected quantity changes, connections reach connection limits and authorization failure. See the next sections about how to troubleshoot.

### Unexpected connection number changes

#### Unexpected connection dropping

If you encounter unexpected connections drop, firstly enable logs in service, server and client sides.

If a connection disconnects, the resource logs will record this disconnecting event, you will see

`ConnectionAborted` or `ConnectionEnded` in `operationName`.

The difference between `ConnectionAborted` and `ConnectionEnded` is that `ConnectionEnded` is an expected disconnecting which is triggered by client or server side. While the `ConnectionAborted` is usually an unexpected connection dropping event, and aborting reason will be provided in `message`.

The abort reasons are listed in the following table:

REASON	DESCRIPTION
Connection count reaches limit	Connection count reaches limit of your current price tier. Consider scale up service unit

REASON	DESCRIPTION
Application server closed the connection	App server triggers the abortion. It can be considered as an expected abortion
Connection ping timeout	Usually it is caused by network issue. Consider checking your app server's availability from the internet
Service reloading, reconnect	Azure SignalR Service is reloading. Azure SignalR supports auto-reconnecting, you can wait until reconnected or manually reconnect to Azure SignalR Service
Internal server transient error	Transient error occurs in Azure SignalR Service, should be auto-recovered
Server connection dropped	Server connection drops with unknown error, consider self-troubleshooting with service/server/client side log first. Try to exclude basic issues (e.g Network issue, app server side issue and so on). If the issue isn't resolved, contact us for further help. For more information, see <a href="#">Get help</a> section.

#### **Unexpected connection growing**

To troubleshoot about unexpected connection growing, the first thing you need to do is to filter out the extra connections. You can add unique test user ID to your test client connection. Then verify it in with resource logs, if you see more than one client connections have the same test user ID or IP, then it is likely the client side create and establish more connections than expectation. Check your client side.

#### **Authorization failure**

If you get 401 Unauthorized returned for client requests, check your resource logs. If you encounter

`Failed to validate audience. Expected Audiences: <valid audience>. Actual Audiences: <actual audience>`, it means all audiences in your access token are invalid. Try to use the valid audiences suggested in the log.

#### **Throttling**

If you find that you cannot establish SignalR client connections to Azure SignalR Service, check your resource logs. If you encounter `Connection count reaches limit` in resource log, you establish too many connections to SignalR Service, which reach the connection count limit. Consider scaling up your SignalR Service. If you encounter `Message count reaches limit` in resource log, it means you use free tier, and you use up the quota of messages. If you want to send more messages, consider changing your SignalR Service to standard tier to send additional messages. For more information, see [Azure SignalR Service Pricing](#).

#### **Get help**

We recommend you troubleshoot by yourself first. Most issues are caused by app server or network issues. Follow [troubleshooting guide with resource log](#) and [basic trouble shooting guide](#) to find the root cause. If the issue still can't be resolved, then consider open an issue in GitHub or create ticket in Azure Portal. Provide:

1. Time range about 30 minutes when the issue occurs
2. Azure SignalR Service's resource ID
3. Issue details, as specific as possible: For example, appserver doesn't send messages, client connection drops and so on
4. Logs collected from server/client side, and other material that might be useful
5. [Optional] Repro code

**NOTE**

If you open issue in GitHub, keep your sensitive information (For example, resource ID, server/client logs) private, only send to members in Microsoft organization privately.

# Audit compliance of Azure SignalR Service resources using Azure Policy

4/21/2021 • 4 minutes to read • [Edit Online](#)

[Azure Policy](#) is a service in Azure that you use to create, assign, and manage policies. These policies enforce different rules and effects over your resources, so those resources stay compliant with your corporate standards and service level agreements.

This article introduces built-in policies (preview) for Azure SignalR Service. Use these policies to audit new and existing SignalR resources for compliance.

There are no charges for using Azure Policy.

## Built-in policy definitions

The following built-in policy definitions are specific to Azure SignalR Service:

NAME (AZURE PORTAL)	DESCRIPTION	EFFECT(S)	VERSION (GITHUB)
<a href="#">Azure SignalR Service should disable public network access</a>	To improve the security of Azure SignalR Service resource, ensure that it isn't exposed to the public internet and can only be accessed from a private endpoint. Disable the public network access property as described in <a href="https://aka.ms/asrs/network_acls">https://aka.ms/asrs/network acls</a> . This option disables access from any public address space outside the Azure IP range, and denies all logins that match IP or virtual network-based firewall rules. This reduces data leakage risks.	Audit, Deny, Disabled	<a href="#">1.0.0</a>
<a href="#">Azure SignalR Service should have local authentication methods disabled</a>	Disabling local authentication methods improves security by ensuring that Azure SignalR Service exclusively require Azure Active Directory identities for authentication.	Audit, Deny, Disabled	<a href="#">1.0.0</a>

NAME	DESCRIPTION	EFFECT(S)	VERSION
<a href="#">Azure SignalR Service should use a Private Link enabled SKU</a>	Azure Private Link lets you connect your virtual network to Azure services without a public IP address at the source or destination which protect your resources against public data leakage risks. The policy limits you to Private Link enabled SKUs for Azure SignalR Service. Learn more about private link at: <a href="https://aka.ms/asrs/privatelink">https://aka.ms/asrs/privatelink</a> .	Audit, Deny, Disabled	1.0.0
<a href="#">Azure SignalR Service should use private link</a>	Azure Private Link lets you connect your virtual network to Azure services without a public IP address at the source or destination. The private link platform handles the connectivity between the consumer and services over the Azure backbone network. By mapping private endpoints to your Azure SignalR Service resource instead of the entire service, you'll reduce your data leakage risks. Learn more about private links at: <a href="https://aka.ms/asrs/privatelink">https://aka.ms/asrs/privatelink</a> .	Audit, Deny, Disabled	1.0.1
<a href="#">Configure private endpoints to Azure SignalR Service</a>	Private endpoints connect your virtual network to Azure services without a public IP address at the source or destination. By mapping private endpoints to Azure SignalR Service resources, you can reduce data leakage risks. Learn more at <a href="https://aka.ms/asrs/privatelink">https://aka.ms/asrs/privatelink</a> .	DeployIfNotExists, Disabled	1.0.0
<a href="#">Deploy - Configure private DNS zones for private endpoints connect to Azure SignalR Service</a>	Use private DNS zones to override the DNS resolution for a private endpoint. A private DNS zone links to your virtual network to resolve to Azure SignalR Service resource. Learn more at: <a href="https://aka.ms/asrs/privatelink">https://aka.ms/asrs/privatelink</a> .	DeployIfNotExists, Disabled	1.0.0

NAME	DESCRIPTION	EFFECT(S)	VERSION
<a href="#">Modify Azure SignalR Service resources to disable public network access</a>	To improve the security of Azure SignalR Service resource, ensure that it isn't exposed to the public internet and can only be accessed from a private endpoint. Disable the public network access property as described in <a href="https://aka.ms/asrs/network_acls">https://aka.ms/asrs/network acls</a> . This option disables access from any public address space outside the Azure IP range, and denies all logins that match IP or virtual network-based firewall rules. This reduces data leakage risks.	Modify, Disabled	1.0.0

## Assign policy definitions

- Assign policy definitions using the [Azure portal](#), [Azure CLI](#), a [Resource Manager template](#), or the Azure Policy SDKs.
- Scope a policy assignment to a resource group, a subscription, or an [Azure management group](#). SignalR policy assignments apply to existing and new SignalR resources within the scope.
- Enable or disable [policy enforcement](#) at any time.

### NOTE

After you assign or update a policy, it takes some time for the assignment to be applied to resources in the defined scope. See information about [policy evaluation triggers](#).

## Review policy compliance

Access compliance information generated by your policy assignments using the Azure portal, Azure command-line tools, or the Azure Policy SDKs. For details, see [Get compliance data of Azure resources](#).

When a resource is non-compliant, there are many possible reasons. To determine the reason or to find the change responsible, see [Determine non-compliance](#).

### Policy compliance in the portal:

- Select All services, and search for Policy.
- Select Compliance.
- Use the filters to limit compliance states or to search for policies

The screenshot shows the Azure Policy | Compliance blade. The 'Compliance' tab is selected. Key metrics displayed include Overall resource compliance at 38% (2123 out of 5530), Non-compliant initiatives at 8 (out of 10), Non-compliant policies at 221 (out of 649), and Non-compliant resources at 3407 (out of 5530). A search bar at the top right is set to 'signalr'. Below the metrics, a table lists a single policy assignment: '[Preview]: Azure SignalR Service should use private links'.

- Select a policy to review aggregate compliance details and events. If desired, then select a specific SignalR for resource compliance.

## Policy compliance in the Azure CLI

You can also use the Azure CLI to get compliance data. For example, use the [az policy assignment list](#) command in the CLI to get the policy IDs of the Azure SignalR Service policies that are applied:

```
az policy assignment list --query "[?contains(displayName,'SignalR')].{name:displayName, ID:id}" --output table
```

Sample output:

Name	ID
[Preview]: Azure SignalR Service should use private links	/subscriptions/<subscriptionId>/resourceGroups/<resourceGroup>/providers/Microsoft.Authorization/policyAssignments/<assignmentId>

Then run [az policy state list](#) to return the JSON-formatted compliance state for all resources under a specific resource group:

```
az policy state list --g <resourceGroup>
```

Or run [az policy state list](#) to return the JSON-formatted compliance state of a specific SignalR resource:

```
az policy state list \
--resource
/subscriptions/<subscriptionId>/resourceGroups/<resourceGroup>/providers/Microsoft.SignalRService/SignalR/<resourceName> \
--namespace Microsoft.SignalRService \
--resource-group <resourceGroup>
```

## Next steps

- Learn more about Azure Policy [definitions](#) and [effects](#)
- Create a [custom policy definition](#)
- Learn more about [governance capabilities](#) in Azure

# Troubleshooting guide for Azure SignalR Service common issues

4/27/2021 • 13 minutes to read • [Edit Online](#)

This guidance is to provide useful troubleshooting guide based on the common issues customers met and resolved in the past years.

## Access token too long

### Possible errors

- Client-side `ERR_CONNECTION_`
- 414 URI Too Long
- 413 Payload Too Large
- Access Token must not be longer than 4K. 413 Request Entity Too Large

### Root cause

For HTTP/2, the max length for a single header is **4 K**, so if using browser to access Azure service, there will be an error `ERR_CONNECTION_` for this limitation.

For HTTP/1.1, or C# clients, the max URI length is **12 K**, the max header length is **16 K**.

With SDK version **1.0.6** or higher, `/negotiate` will throw `413 Payload Too Large` when the generated access token is larger than **4 K**.

### Solution

By default, claims from `context.User.Claims` are included when generating JWT access token to **ASRS**(Azure SignalR Service), so that the claims are preserved and can be passed from **ASRS** to the `Hub` when the client connects to the `Hub`.

In some cases, `context.User.Claims` are used to store lots of information for app server, most of which are not used by `Hub`s but by other components.

The generated access token is passed through the network, and for WebSocket/SSE connections, access tokens are passed through query strings. So as the best practice, we suggest only passing **necessary** claims from the client through **ASRS** to your app server when the Hub needs.

There is a `ClaimsProvider` for you to customize the claims passing to **ASRS** inside the access token.

For ASP.NET Core:

```
services.AddSignalR()
    .AddAzureSignalR(options =>
{
    // pick up necessary claims
    options.ClaimsProvider = context => context.User.Claims.Where(...);
});
```

For ASP.NET:

```

services.MapAzureSignalR(GetType().FullName, options =>
{
    // pick up necessary claims
    options.ClaimsProvider = context.Authentication?.User.Claims.Where(...);
});

```

Having issues or feedback about the troubleshooting? Let us know.

## TLS 1.2 required

### Possible errors

- ASP.NET "No server available" error [#279](#)
- ASP.NET "The connection is not active, data cannot be sent to the service." error [#324](#)
- "An error occurred while making the HTTP request to https://. This error could be because the server certificate is not configured properly with HTTPS in the HTTPS case. This error could also be caused by a mismatch of the security binding between the client and the server."

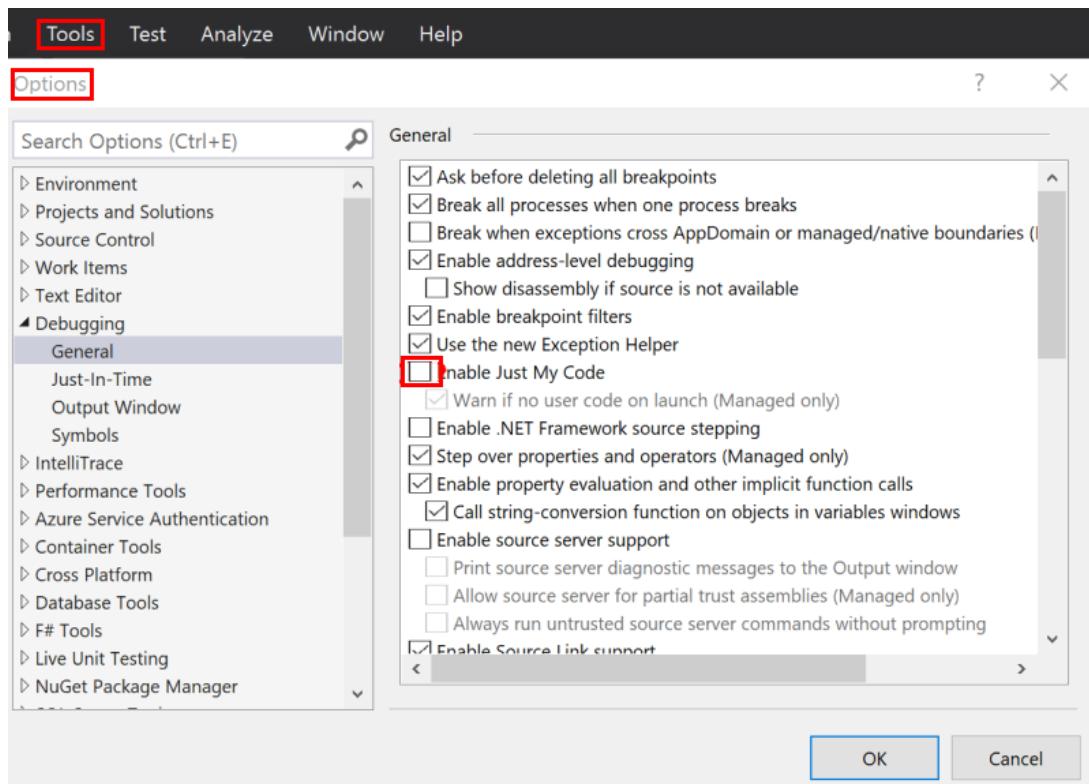
### Root cause

Azure Service only supports TLS1.2 for security concerns. With .NET framework, it is possible that TLS1.2 is not the default protocol. As a result, the server connections to ASRS cannot be successfully established.

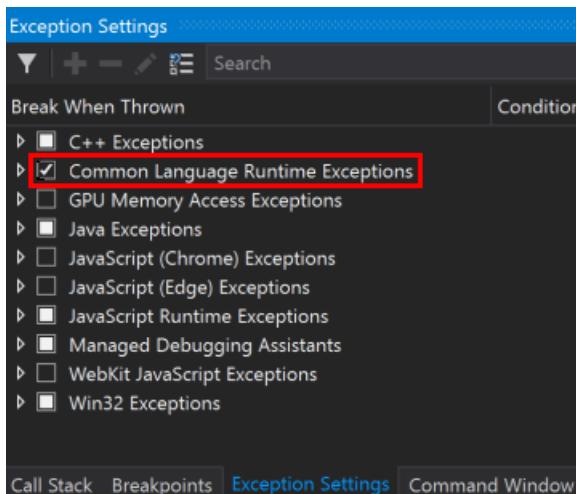
### Troubleshooting guide

- If this error can be reproduced locally, uncheck *Just My Code* and throw all CLR exceptions and debug the app server locally to see what exception throws.

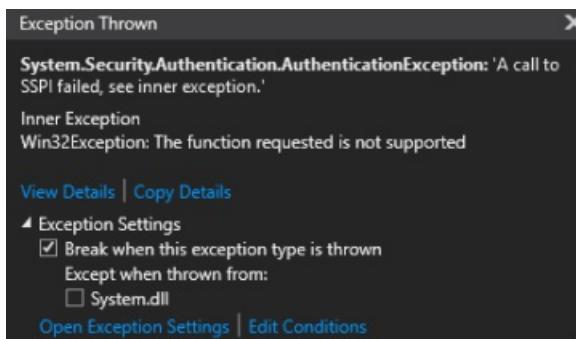
- Uncheck *Just My Code*



- Throw CLR exceptions



- See the exceptions throw when debugging the app server-side code:



2. For ASP.NET ones, you can also add following code to your `Startup.cs` to enable detailed trace and see the errors from the log.

```
app.MapAzureSignalR(this.GetType().FullName);
// Make sure this switch is called after MapAzureSignalR
GlobalHost.TraceManager.Switch.Level = SourceLevels.Information;
```

## Solution

Add following code to your Startup:

```
ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
```

[Having issues or feedback about the troubleshooting? Let us know.](#)

## 400 Bad Request returned for client requests

### Root cause

Check if your client request has multiple `hub` query strings. `hub` is a preserved query parameter and 400 will throw if the service detects more than one `hub` in the query.

[Having issues or feedback about the troubleshooting? Let us know.](#)

## 401 Unauthorized returned for client requests

### Root cause

Currently the default value of JWT token's lifetime is 1 hour.

For ASP.NET Core SignalR, when it is using WebSocket transport type, it is OK.

For ASP.NET Core SignalR's other transport type, SSE and long-polling, this means by default the connection can at most persist for 1 hour.

For ASP.NET SignalR, the client sends a `/ping` KeepAlive request to the service from time to time, when the `/ping` fails, the client **aborts** the connection and never reconnect. This means, for ASP.NET SignalR, the default token lifetime makes the connection lasts for **at most** 1 hour for all the transport type.

## Solution

For security concerns, extend TTL is not encouraged. We suggest adding reconnect logic from the client to restart the connection when such 401 occurs. When the client restarts the connection, it will negotiate with app server to get the JWT token again and get a renewed token.

Check [here](#) for how to restart client connections.

[Having issues or feedback about the troubleshooting? Let us know.](#)

## 404 returned for client requests

For a SignalR persistent connection, it first `/negotiate` to Azure SignalR service and then establishes the real connection to Azure SignalR service.

### Troubleshooting guide

- Following [How to view outgoing requests](#) to get the request from the client to the service.
- Check the URL of the request when 404 occurs. If the URL is targeting to your web app, and similar to `{your_web_app}/hubs/{hubName}`, check if the client `skipNegotiation` is `true`. When using Azure SignalR, the client receives redirect URL when it first negotiates with the app server. The client should **NOT** skip negotiation when using Azure SignalR.
- Another 404 can happen when the connect request is handled more than 5 seconds after `/negotiate` is called. Check the timestamp of the client request, and open an issue to us if the request to the service has a slow response.

[Having issues or feedback about the troubleshooting? Let us know.](#)

## 404 returned for ASP.NET SignalR's reconnect request

For ASP.NET SignalR, when the [client connection drops](#), it reconnects using the same `connectionId` for three times before stopping the connection. `/reconnect` can help if the connection is dropped due to network intermittent issues that `/reconnect` can reestablish the persistent connection successfully. Under other circumstances, for example, the client connection is dropped due to the routed server connection is dropped, or SignalR Service has some internal errors like instance restart/failover/deployment, the connection no longer exists, thus `/reconnect` returns `404`. It is the expected behavior for `/reconnect` and after three times retry the connection stops. We suggest having [connection restart](#) logic when connection stops.

[Having issues or feedback about the troubleshooting? Let us know.](#)

## 429 (Too Many Requests) returned for client requests

There are two cases.

### Concurrent connection count exceeds limit

For **Free** instances, Concurrent connection count limit is 20 For **Standard** instances, concurrent connection count limit **per unit** is 1 K, which means Unit100 allows 100-K concurrent connections.

The connections include both client and server connections. check [here](#) for how connections are counted.

### Too many negotiate requests at the same time

We suggest having a random delay before reconnecting, check [here](#) for retry samples.

Having issues or feedback about the troubleshooting? Let us know.

## 500 Error when negotiate: Azure SignalR Service is not connected yet, please try again later

### Root cause

This error is reported when there is no server connection to Azure SignalR Service connected.

### Troubleshooting guide

Enable server-side trace to find out the error details when the server tries to connect to Azure SignalR Service.

### Enable server-side logging for ASP.NET Core SignalR

Server-side logging for ASP.NET Core SignalR integrates with the `ILogger` based [logging](#) provided in the ASP.NET Core framework. You can enable server-side logging by using `ConfigureLogging`, a sample usage as follows:

```
.ConfigureLogging((hostingContext, logging) =>
{
    logging.AddConsole();
    logging.AddDebug();
})
```

Logger categories for Azure SignalR always start with `Microsoft.Azure.SignalR`. To enable detailed logs from Azure SignalR, configure the preceding prefixes to `Debug` level in your `appsettings.json` file like below:

```
{
  "Logging": {
    "LogLevel": {
      ...
      "Microsoft.Azure.SignalR": "Debug",
      ...
    }
  }
}
```

### Enable server-side traces for ASP.NET SignalR

When using SDK version >= `1.0.0`, you can enable traces by adding the following to `web.config`: ([Details](#))

```
<system.diagnostics>
  <sources>
    <source name="Microsoft.Azure.SignalR" switchName="SignalRSwitch">
      <listeners>
        <add name="ASRS" />
      </listeners>
    </source>
  </sources>
  <!-- Sets the trace verbosity level -->
  <switches>
    <add name="SignalRSwitch" value="Information" />
  </switches>
  <!-- Specifies the trace writer for output -->
  <sharedListeners>
    <add name="ASRS" type="System.Diagnostics.TextWriterTraceListener" initializeData="asrs.log.txt" />
  </sharedListeners>
  <trace autoflush="true" />
</system.diagnostics>
```

[Having issues or feedback about the troubleshooting? Let us know.](#)

## Client connection drops

When the client is connected to the Azure SignalR, the persistent connection between the client and Azure SignalR can sometimes drop for different reasons. This section describes several possibilities causing such connection drop and provides some guidance on how to identify the root cause.

### Possible errors seen from the client side

- The remote party closed the WebSocket connection without completing the close handshake
- Service timeout. 30.00ms elapsed without receiving a message from service.
- {"type":7,"error":"Connection closed with an error."}
- {"type":7,"error":"Internal server error."}

### Root cause

Client connections can drop under various circumstances:

- When `Hub` throws exceptions with the incoming request.
- When the server connection, which the client routed to, drops, see below section for details on [server connection drops](#).
- When a network connectivity issue happens between client and SignalR Service.
- When SignalR Service has some internal errors like instance restart, failover, deployment, and so on.

### Troubleshooting guide

1. Open app server-side log to see if anything abnormal took place
2. Check app server-side event log to see if the app server restarted
3. Create an issue to us providing the time frame, and email the resource name to us

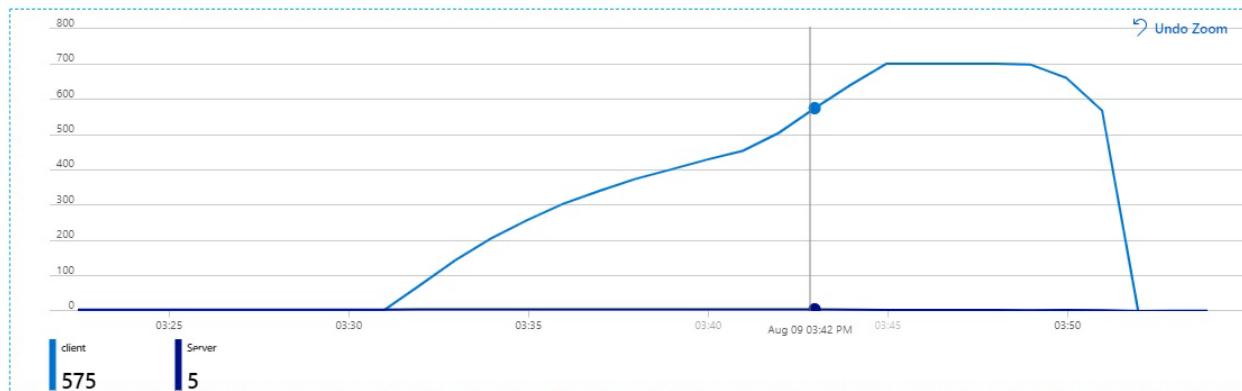
[Having issues or feedback about the troubleshooting? Let us know.](#)

## Client connection increases constantly

It might be caused by improper usage of client connection. If someone forgets to stop/dispose SignalR client, the connection remains open.

### Possible errors seen from the SignalR's metrics that is in Monitoring section of Azure portal resource menu

Client connections rise constantly for a long time in Azure SignalR's Metrics.



### Root cause

SignalR client connection's `DisposeAsync` never be called, the connection keeps open.

### Troubleshooting guide

Check if the SignalR client never closes.

## Solution

Check if you close connection. Manually call `HubConnection.DisposeAsync()` to stop the connection after using it.

For example:

```
var connection = new HubConnectionBuilder()
    .WithUrl(...)
    .Build();
try
{
    await connection.StartAsync();
    // Do your stuff
    await connection.StopAsync();
}
finally
{
    await connection.DisposeAsync();
}
```

## Common improper client connection usage

### Azure Function example

This issue often occurs when someone establishes SignalR client connection in Azure Function method instead of making it a static member to your Function class. You might expect only one client connection is established, but you see client connection count increases constantly in Metrics that is in Monitoring section of Azure portal resource menu, all these connections drop only after the Azure Function or Azure SignalR service restarts. This is because for **each** request, Azure Function creates **one** client connection, if you don't stop client connection in Function method, the client keeps the connections alive to Azure SignalR service.

### Solution

- Remember to close client connection if you use SignalR clients in Azure function or use SignalR client as a singleton.
- Instead of using SignalR clients in Azure function, you can create SignalR clients anywhere else and use [Azure Functions Bindings for Azure SignalR Service](#) to **negotiate** the client to Azure SignalR. And you can also utilize the binding to **send messages**. Samples to negotiate client and send messages can be found [here](#). Further information can be found [here](#).
- When you use SignalR clients in Azure function, there might be a better architecture to your scenario. Check if you design a proper serverless architecture. You can refer to [Real-time serverless applications with the SignalR Service bindings in Azure Functions](#).

[Having issues or feedback about the troubleshooting? Let us know.](#)

## Server connection drops

When the app server starts, in the background, the Azure SDK starts to initiate server connections to the remote Azure SignalR. As described in [Internals of Azure SignalR Service](#), Azure SignalR routes incoming client traffics to these server connections. Once a server connection is dropped, all the client connections it serves will be closed too.

As the connections between the app server and SignalR Service are persistent connections, they may experience network connectivity issues. In the Server SDK, we have **Always Reconnect** strategy to server connections. As the best practice, we also encourage users to add continuous reconnect logic to the clients with a random delay time to avoid massive simultaneous requests to the server.

On a regular basis, there are new version releases for the Azure SignalR Service, and sometimes the Azure-wide OS patching or upgrades or occasionally interruption from our dependent services. These may bring in a short period of service disruption, but as long as client-side has the disconnect/reconnect mechanism, the impact is

minimal like any client-side caused disconnect-reconnect.

This section describes several possibilities leading to server connection drop, and provides some guidance on how to identify the root cause.

### Possible errors seen from the server side

- [Error]Connection "..." to the service was dropped
- The remote party closed the WebSocket connection without completing the close handshake
- Service timeout. 30.00ms elapsed without receiving a message from service.

### Root cause

Server-service connection is closed by ASRS(Azure SignalR Service).

For ping timeout, it might be caused by high CPU usage or thread pool starvation on the server side.

For ASP.NET SignalR, a known issue was fixed in SDK 1.6.0. Upgrade your SDK to newest version.

## Thread pool starvation

If your server is starving, that means no threads are working on message processing. All threads are not responding in a certain method.

Normally, this scenario is caused by async over sync or by `Task.Result` / `Task.Wait()` in async methods.

See [ASP.NET Core performance best practices](#).

See more about [thread pool starvation](#).

### How to detect thread pool starvation

Check your thread count. If there are no spikes at that time, take these steps:

- If you're using Azure App Service, check the thread count in metrics. Check the `Max` aggregation:

Max Thread Count for Example

Add metric Add filter Apply splitting

Scope	Metric Namespace	Metric	Aggregation
example-scope	App Service standard m...	Thread Count	Max

- If you're using the .NET Framework, you can find [metrics](#) in the performance monitor in your server VM.
- If you're using .NET Core in a container, see [Collect diagnostics in containers](#).

You also can use code to detect thread pool starvation:

```

public class ThreadPoolStarvationDetector : EventListener
{
    private const int EventIdForThreadPoolWorkerThreadAdjustmentAdjustment = 55;
    private const uint ReasonForStarvation = 6;

    private readonly ILogger<ThreadPoolStarvationDetector> _logger;

    public ThreadPoolStarvationDetector(ILogger<ThreadPoolStarvationDetector> logger)
    {
        _logger = logger;
    }

    protected override void OnEventSourceCreated(EventSource eventSource)
    {
        if (eventSource.Name == "Microsoft-Windows-DotNETRuntime")
        {
            EnableEvents(eventSource, EventLevel.Informational, EventKeywords.All);
        }
    }

    protected override void OnEventWritten(EventWrittenEventArgs eventData)
    {
        // See: https://docs.microsoft.com/en-us/dotnet/framework/performance/thread-pool-etw-events#threadpoolworkerthreadadjustmentadjustment
        if (eventData.EventId == EventIdForThreadPoolWorkerThreadAdjustmentAdjustment &&
            eventData.Payload[3] as uint? == ReasonForStarvation)
        {
            _logger.LogWarning("Thread pool starvation detected!");
        }
    }
}

```

Add it to your service:

```
service.AddSingleton<ThreadPoolStarvationDetector>();
```

Then, check your log when the server connection is disconnected by ping timeout.

## How to find the root cause of thread pool starvation

To find the root cause of thread pool starvation:

- Dump the memory, and then analyze the call stack. For more information, see [Collect and analyze memory dumps](#).
- Use `clrmd` to dump the memory when thread pool starvation is detected. Then, log the call stack.

## Troubleshooting guide

1. Open the app server-side log to see if anything abnormal took place.
2. Check the app server-side event log to see if the app server restarted.
3. Create an issue. Provide the time frame, and email the resource name to us.

[Having issues or feedback about the troubleshooting? Let us know.](#)

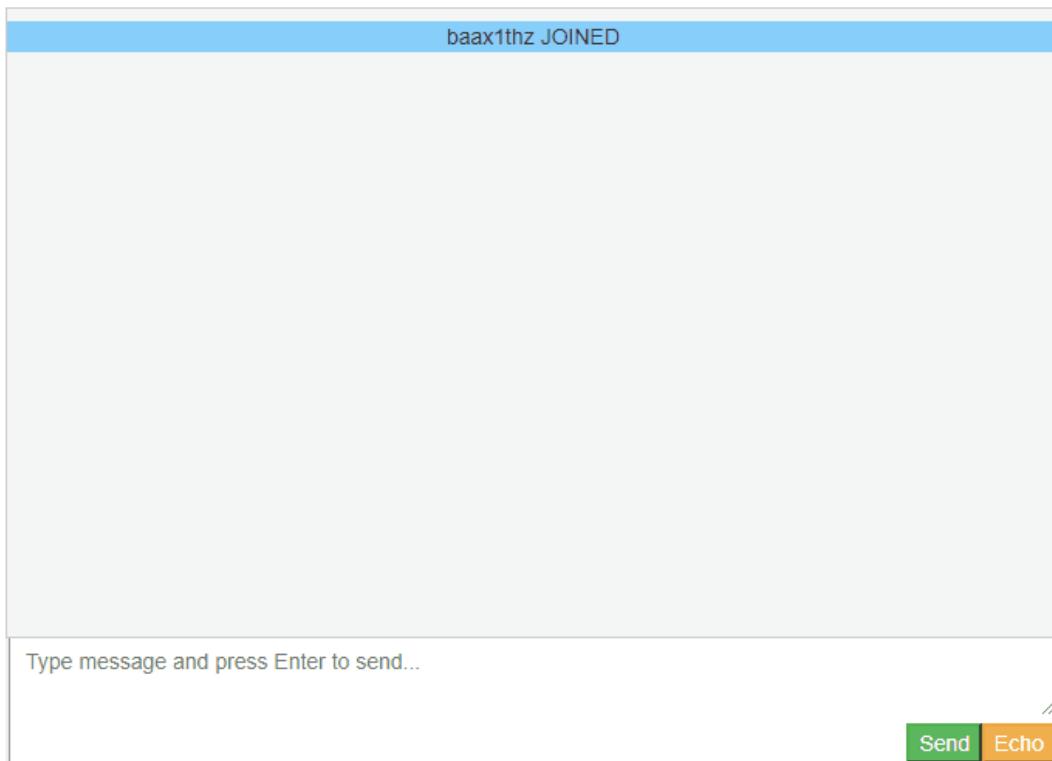
## Tips

- How to view the outgoing request from client? Take ASP.NET Core one for example (ASP.NET one is similar):
  - From browser:

Take Chrome as an example, you can use F12 to open the console window, and switch to **Network** tab. You might need to refresh the page using F5 to capture the network from the very

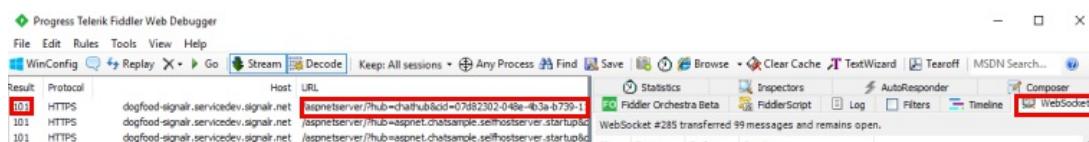
beginning.

## Azure SignalR Group Chat



- From C# client:

You can view local web traffics using [Fiddler](#). WebSocket traffics are supported since Fiddler 4.5.



- How to restart client connection?

Here are the [Sample codes](#) containing restarting connection logic with *ALWAYS RETRY* strategy:

- [ASP.NET Core C# Client](#)
- [ASP.NET Core JavaScript Client](#)
- [ASP.NET C# Client](#)
- [ASP.NET JavaScript Client](#)

[Having issues or feedback about the troubleshooting? Let us know.](#)

## Next steps

In this guide, you learned about how to handle the common issues. You could also learn more generic troubleshooting methods.

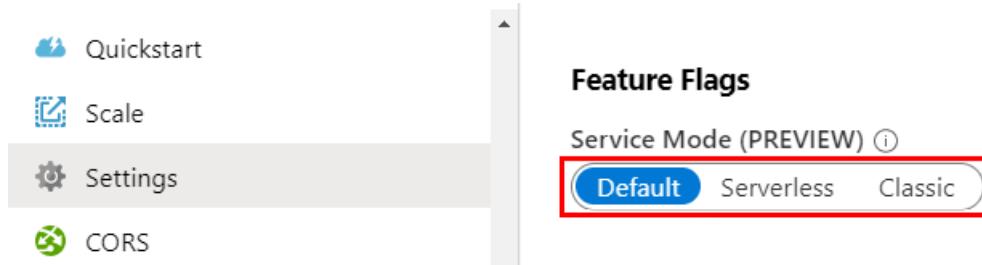
[How to troubleshoot connectivity and message delivery issues](#)

# How to troubleshoot connectivity and message delivery issues

7/1/2021 • 6 minutes to read • [Edit Online](#)

This guidance introduces several ways to help do self-diagnosis to find the root cause directly or narrow down the issue. The self-diagnosis result is also useful when reporting it to us for further investigation.

First, you need to check from the Azure portal which [ServiceMode](#) is the Azure SignalR Service (also known as ASRS) configured to.



- For `Default` mode, refer to [default mode troubleshooting](#)
- For `Serverless` mode, refer to [serverless mode troubleshooting](#)
- For `Classic` mode, refer to [classic mode troubleshooting](#)

Second, you need to capture service traces to troubleshoot. For how to capture traces, refer to [How to capture service traces](#).

[Having issues or feedback about the troubleshooting? Let us know.](#)

## How to capture service traces

To simplify troubleshooting process, Azure SignalR service provides [live trace tool](#) to expose service traces on **connectivity** and **messaging** categories. The traces includes but not limited to connection connected/disconnected events, message received/left events. With [live trace tool](#), you can capture, view, sort, filter and export live traces. For more details, refer to [How to use live trace tool](#).

[Having issues or feedback about the troubleshooting? Let us know.](#)

## Default mode troubleshooting

When ASRS is in `Default` mode, there are **three** roles: *Client*, *Server*, and *Service*.

- *Client*: *Client* stands for the clients connected to ASRS. The persistent connections connecting client and ASRS are called *Client Connections* in this guidance.
- *Server*: *Server* stands for the server that serves client negotiation and hosts SignalR `Hub` logic. And the persistent connections between *Server* and ASRS are called *Server Connections* in this guidance.
- *Service*: *Service* is the short name for ASRS in this guidance.

Refer to [Internals of Azure SignalR Service](#) for the detailed introduction of the whole architecture and workflow.

There are several ways that can help you narrow down the issue.

- If the issue happens right in the way or is repro-able, the straight-forward way is to view the on-going traffic.
- If the issue is hard to repro, traces and logs can help.

### How to view the traffic and narrow down the issue

Capturing the on-going traffic is the most straight-forward way to narrow down the issue. You can capture the [Network traces](#) using the options described below:

- [Collect a network trace with Fiddler](#)
- [Collect a network trace with tcpdump](#)
- [Collect a network trace in the browser](#)

#### Client requests

For a SignalR persistent connection, it first `/negotiate` to your hosted app server and then redirected to the Azure SignalR service and then establishes the real persistent connection to Azure SignalR service. Refer to [Internals of Azure SignalR Service](#) for the detailed steps.

With the client-side network trace in hand, check which request fails with what status code and what response, and look for solutions inside [Troubleshooting Guide](#).

#### Server requests

SignalR *Server* maintains the *Server Connection* between *Server* and *Service*. When the app server starts, it starts the **WebSocket** connection to Azure SignalR service. All the client traffics are routed through Azure SignalR service to these *Server Connections* and then dispatched to the `Hub`. When a *Server Connection* drops, the clients routed to this *Server Connection* will be impacted. Our Azure SignalR SDK has a logic "Always Retry" to reconnect the *Server Connection* with at most 1-minute delay to minimize the impact.

*Server Connections* can drop because of network instability or regular maintenance of Azure SignalR Service, or your hosted app server updates/maintainance. As long as client-side has the disconnect/reconnect mechanism, the impact is minimal like any client-side caused disconnect-reconnect.

View server-side network trace to find out the status code and error detail why *Server Connection* drops or is rejected by the *Service*, and look for the root cause inside [Troubleshooting Guide](#).

[Having issues or feedback about the troubleshooting? Let us know.](#)

### How to add logs

Logs can be useful to diagnose issues and monitor the running status.

#### How to enable client-side log

Client side logging experience is exactly the same as when using self-hosted SignalR.



- [JavaScript client logging](#)
- [.NET client logging](#)



- [.NET client](#)
- [Enabling tracing in Windows Phone 8 clients](#)
- [Enabling tracing in the JavaScript client](#)

#### How to enable server-side log

#### Enable server-side logging for ASP.NET Core SignalR

Server-side logging for ASP.NET Core SignalR integrates with the ILogger based **logging** provided in the ASP.NET Core framework. You can enable server-side logging by using ConfigureLogging, a sample usage as follows:

```
.ConfigureLogging((hostingContext, logging) =>
{
    logging.AddConsole();
    logging.AddDebug();
})
```

Logger categories for Azure SignalR always start with Microsoft.Azure.SignalR. To enable detailed logs from Azure SignalR, configure the preceding prefixes to Information level in your **appsettings.json** file like below:

```
{
  "Logging": {
    "LogLevel": {
      ...
      "Microsoft.Azure.SignalR": "Information",
      ...
    }
  }
}
```

Check if there are any abnormal warning/error logs recorded.

#### Enable server-side traces for ASP.NET SignalR

When using SDK version >= 1.0.0, you can enable traces by adding the following to **web.config**: ([Details](#))

```
<system.diagnostics>
  <sources>
    <source name="Microsoft.Azure.SignalR" switchName="SignalRSwitch">
      <listeners>
        <add name="ASRS" />
      </listeners>
    </source>
  </sources>
  <!-- Sets the trace verbosity level -->
  <switches>
    <add name="SignalRSwitch" value="Information" />
  </switches>
  <!-- Specifies the trace writer for output -->
  <sharedListeners>
    <add name="ASRS" type="System.Diagnostics.TextWriterTraceListener" initializeData="asrs.log.txt" />
  </sharedListeners>
  <trace autoflush="true" />
</system.diagnostics>
```

Check if there are any abnormal warning/error logs recorded.

#### How to enable logs inside Azure SignalR service

You can also [enable diagnostic logs](#) for Azure SignalR service, these logs provide detailed information of every connection connected to the Azure SignalR service.

[Having issues or feedback about the troubleshooting? Let us know.](#)

## Serverless mode troubleshooting

When ASRS is in *Serverless* mode, only ASP.NET Core SignalR supports `Serverless` mode, and ASP.NET SignalR does NOT support this mode.

To diagnose connectivity issues in `Serverless` mode, the most straight forward way is to [view client side traffic](#). Enable [client-side logs](#) and [service-side logs](#) can also be helpful.

[Having issues or feedback about the troubleshooting? Let us know.](#)

## Classic mode troubleshooting

`Classic` mode is obsoleted and is not encouraged to use. When in this mode, Azure SignalR service uses the connected *Server Connections* to determine if current service is in `default` mode or `serverless` mode. This can lead to some intermediate client connectivity issues because, when there is a sudden drop of all the connected *Server Connection*, for example due to network instability, Azure SignalR believes it is now switched to `serverless` mode, and clients connected during this period will never be routed to the hosted app server. Enable [service-side logs](#) and check if there are any clients recorded as `ServerlessModeEntered` if you have hosted app server however some clients never reach the app server side. If there is any, [abort these client connections](#) and let the clients restart can help.

Troubleshooting `classic` mode connectivity and message delivery issues are similar to [troubleshooting default mode issues](#).

[Having issues or feedback about the troubleshooting? Let us know.](#)

## Service health

You can check the health api for service health.

- Request: GET `https://{{instance_name}}.service.signalr.net/api/v1/health`
- Response status code:
  - 200: healthy.
  - 503: your service is unhealthy. You can:
    - Wait several minutes for autorecover.
    - Check the ip address is same as the ip from portal.
    - Or restart instance.
    - If all above options do not work, contact us by adding new support request in Azure portal.

More about [disaster recovery](#).

[Having issues or feedback about the troubleshooting? Let us know.](#)

## Next steps

In this guide, you learned about how to troubleshoot connectivity and message delivery issues. You could also learn how to handle the common issues.

[Troubleshooting guide](#)

# How to use live trace tool for Azure SignalR service

7/15/2021 • 2 minutes to read • [Edit Online](#)

Live trace tool is a single web application for capturing and displaying live traces in Azure SignalR service. The live traces can be collected in real time without any dependency on other services. You can enable and disable the live trace feature with a single click. You can also choose any log category that you're interested.

## NOTE

Please note that the live traces will be counted as outbound messages.

## Launch the live trace tool

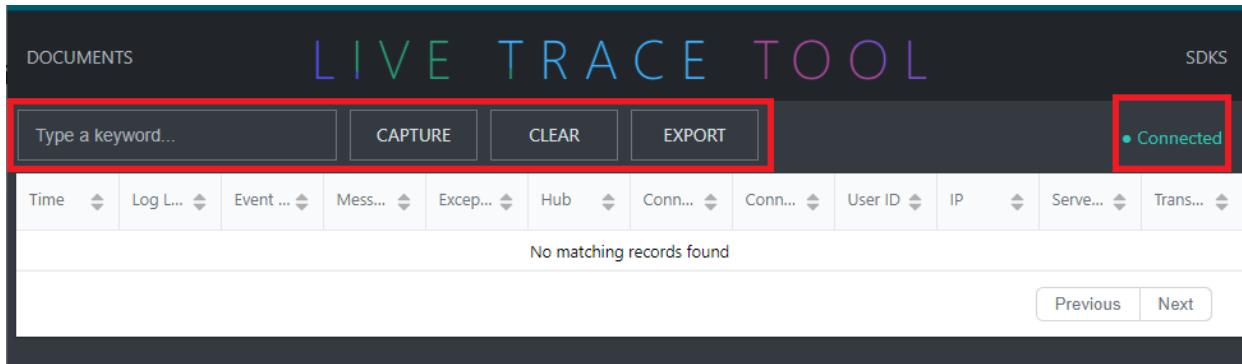
1. Go to the Azure portal.
2. Check **Enable Live Trace**.
3. click **Save** button in tool bar and wait for the changes take effect.
4. On the **Diagnostic Settings** page of your Azure Web PubSub service instance, select **Open Live Trace Tool**.

The screenshot shows the Azure Diagnostic Settings page for an Azure SignalR service. The 'Log Source Settings' section includes instructions for collecting logs from all or partial connections. The 'Log Destination Settings' section shows a connection to a storage account and log analytics workspace. The 'Live Trace Settings' section contains a note about live traces being counted as outbound messages and features an 'Enable Live Trace' checkbox, which is checked. The 'Diagnostic Settings' menu item at the bottom left is highlighted with a red box.

## Capture live traces

The live trace tool provides some fundamental functionalities to help you capture the live traces for troubleshooting.

- **Capture:** Begin to capture the real time live traces from Azure Web PubSub instance with live trace tool.
- **Clear:** Clear the captured real time live traces.
- **Export:** Export live traces to a file. The current supported file format is CSV file.
- **Log filter:** The live trace tool allows you filtering the captured real time live traces with one specific key word. The common separator (for example, space, comma, semicolon, and so on) will be treated as part of the key word.
- **Status:** The status shows whether the live trace tool is connected or disconnected with the specific instance.



The real time live traces captured by live trace tool contain detailed information for troubleshooting.

NAME	DESCRIPTION
Time	Log event time
Log Level	Log event level (Trace/Debug/Informational/Warning/Error)
Event Name	Operation name of the event
Message	Detailed message of log event
Exception	The run-time exception of Azure Web PubSub service
Hub	User-defined Hub Name
Connection ID	Identity of the connection
Connection ID	Type of the connection. Allowed values are <code>Server</code> (connections between server and service) and <code>Client</code> (connections between client and service)
User ID	Identity of the user
IP	The IP address of client
Server Sticky	Routing mode of client. Allowed values are <code>Disabled</code> , <code>Preferred</code> and <code>Required</code> . For more information, see <a href="#">ServerStickyMode</a>

NAME	DESCRIPTION
Transport	The transport that the client can use to send HTTP requests. Allowed values are <code>WebSockets</code> , <code>ServerSentEvents</code> and <code>LongPolling</code> . For more information, see <a href="#">HttpTransportType</a>

## Next Steps

In this guide, you learned about how to use live trace tool. You could also learn how to handle the common issues:

- Troubleshooting guides: For how to troubleshoot typical issues based on live traces, see our [troubleshooting guide](#).
- Troubleshooting methods: For self-diagnosis to find the root cause directly or narrow down the issue, see our [troubleshooting methods introduction](#).

# Azure Policy built-in definitions for Azure SignalR

8/13/2021 • 3 minutes to read • [Edit Online](#)

This page is an index of [Azure Policy](#) built-in policy definitions for Azure SignalR. For additional Azure Policy built-ins for other services, see [Azure Policy built-in definitions](#).

The name of each built-in policy definition links to the policy definition in the Azure portal. Use the link in the **Version** column to view the source on the [Azure Policy GitHub repo](#).

## Azure SignalR

Name (Azure Portal)	Description	Effect(s)	Version (GitHub)
<a href="#">Azure SignalR Service should disable public network access</a>	To improve the security of Azure SignalR Service resource, ensure that it isn't exposed to the public internet and can only be accessed from a private endpoint. Disable the public network access property as described in <a href="https://aka.ms/asrs/network_acls">https://aka.ms/asrs/network_acls</a> . This option disables access from any public address space outside the Azure IP range, and denies all logins that match IP or virtual network-based firewall rules. This reduces data leakage risks.	Audit, Deny, Disabled	1.0.0
<a href="#">Azure SignalR Service should have local authentication methods disabled</a>	Disabling local authentication methods improves security by ensuring that Azure SignalR Service exclusively require Azure Active Directory identities for authentication.	Audit, Deny, Disabled	1.0.0
<a href="#">Azure SignalR Service should use a Private Link enabled SKU</a>	Azure Private Link lets you connect your virtual network to Azure services without a public IP address at the source or destination which protect your resources against public data leakage risks. The policy limits you to Private Link enabled SKUs for Azure SignalR Service. Learn more about private link at: <a href="https://aka.ms/asrs/privatelink">https://aka.ms/asrs/privatelink</a> .	Audit, Deny, Disabled	1.0.0

NAME	DESCRIPTION	EFFECT(S)	VERSION
<a href="#">Azure SignalR Service should use private link</a>	<p>Azure Private Link lets you connect your virtual network to Azure services without a public IP address at the source or destination. The private link platform handles the connectivity between the consumer and services over the Azure backbone network. By mapping private endpoints to your Azure SignalR Service resource instead of the entire service, you'll reduce your data leakage risks. Learn more about private links at: <a href="https://aka.ms/asrs/privatelink">https://aka.ms/asrs/privatelink</a>.</p>	Audit, Deny, Disabled	1.0.1
<a href="#">Azure Web PubSub Service should disable public network access</a>	<p>Disabling public network access improves security by ensuring that Azure Web PubSub service isn't exposed on the public internet. Creating private endpoints can limit exposure of Azure Web PubSub service. Learn more at: <a href="https://aka.ms/awps/networkacl">https://aka.ms/awps/networkacl</a>.</p>	Audit, Deny, Disabled	1.0.0
<a href="#">Azure Web PubSub Service should use a SKU that supports private link</a>	<p>With supported SKU, Azure Private Link lets you connect your virtual network to Azure services without a public IP address at the source or destination. The Private Link platform handles the connectivity between the consumer and services over the Azure backbone network. By mapping private endpoints to Azure Web PubSub service, you can reduce data leakage risks. Learn more about private links at: <a href="https://aka.ms/awps/private-link">https://aka.ms/awps/private-link</a>.</p>	Audit, Deny, Disabled	1.0.0

NAME	DESCRIPTION	EFFECT(S)	VERSION
<a href="#">Azure Web PubSub Service should use private link</a>	Azure Private Link lets you connect your virtual networks to Azure services without a public IP address at the source or destination. The private link platform handles the connectivity between the consumer and services over the Azure backbone network. By mapping private endpoints to your Azure Web PubSub Service, you can reduce data leakage risks. Learn more about private links at: <a href="https://aka.ms/awps/private-link">https://aka.ms/awps/private-link</a> .	Audit, Deny, Disabled	1.0.0
<a href="#">Configure Azure Web PubSub Service to disable public network access</a>	Disable public network access for your Azure Web PubSub resource so that it's not accessible over the public internet. This can reduce data leakage risks. Learn more at: <a href="https://aka.ms/awps/networkacl">https://aka.ms/awps/networkacl</a> .	Modify, Disabled	1.0.0
<a href="#">Configure Azure Web PubSub Service with private endpoints</a>	Private endpoints connect your virtual networks to Azure services without a public IP address at the source or destination. By mapping private endpoints to Azure Web PubSub service, you can reduce data leakage risks. Learn more about private links at: <a href="https://aka.ms/awps/private-link">https://aka.ms/awps/private-link</a> .	DeployIfNotExists, Disabled	1.0.0
<a href="#">Configure private endpoints to Azure SignalR Service</a>	Private endpoints connect your virtual network to Azure services without a public IP address at the source or destination. By mapping private endpoints to Azure SignalR Service resources, you can reduce data leakage risks. Learn more at <a href="https://aka.ms/asrs/private-link">https://aka.ms/asrs/private-link</a> .	DeployIfNotExists, Disabled	1.0.0

NAME	DESCRIPTION	EFFECT(S)	VERSION
<a href="#">Modify Azure SignalR Service resources to disable public network access</a>	To improve the security of Azure SignalR Service resource, ensure that it isn't exposed to the public internet and can only be accessed from a private endpoint. Disable the public network access property as described in <a href="https://aka.ms/asrs/network_acls">https://aka.ms/asrs/network acls</a> . This option disables access from any public address space outside the Azure IP range, and denies all logins that match IP or virtual network-based firewall rules. This reduces data leakage risks.	Modify, Disabled	1.0.0

## Next steps

- See the built-ins on the [Azure Policy GitHub repo](#).
- Review the [Azure Policy definition structure](#).
- Review [Understanding policy effects](#).

# Azure subscription and service limits, quotas, and constraints

8/2/2021 • 114 minutes to read • [Edit Online](#)

This document lists some of the most common Microsoft Azure limits, which are also sometimes called quotas.

To learn more about Azure pricing, see [Azure pricing overview](#). There, you can estimate your costs by using the [pricing calculator](#). You also can go to the pricing details page for a particular service, for example, [Windows VMs](#). For tips to help manage your costs, see [Prevent unexpected costs with Azure billing and cost management](#).

## Managing limits

### NOTE

Some services have adjustable limits.

When a service doesn't have adjustable limits, the following tables use the header **Limit**. In those cases, the default and the maximum limits are the same.

When the limit can be adjusted, the tables include **Default limit** and **Maximum limit** headers. The limit can be raised above the default limit but not above the maximum limit.

If you want to raise the limit or quota above the default limit, [open an online customer support request at no charge](#).

The terms *soft limit* and *hard limit* often are used informally to describe the current, adjustable limit (soft limit) and the maximum limit (hard limit). If a limit isn't adjustable, there won't be a soft limit, only a hard limit.

[Free Trial subscriptions](#) aren't eligible for limit or quota increases. If you have a [Free Trial subscription](#), you can upgrade to a [Pay-As-You-Go](#) subscription. For more information, see [Upgrade your Azure Free Trial subscription to a Pay-As-You-Go subscription](#) and the [Free Trial subscription FAQ](#).

Some limits are managed at a regional level.

Let's use vCPU quotas as an example. To request a quota increase with support for vCPUs, you must decide how many vCPUs you want to use in which regions. You then request an increase in vCPU quotas for the amounts and regions that you want. If you need to use 30 vCPUs in West Europe to run your application there, you specifically request 30 vCPUs in West Europe. Your vCPU quota isn't increased in any other region--only West Europe has the 30-vCPU quota.

As a result, decide what your quotas must be for your workload in any one region. Then request that amount in each region into which you want to deploy. For help in how to determine your current quotas for specific regions, see [Resolve errors for resource quotas](#).

## General limits

For limits on resource names, see [Naming rules and restrictions for Azure resources](#).

For information about Resource Manager API read and write limits, see [Throttling Resource Manager requests](#).

### Management group limits

The following limits apply to [management groups](#).

RESOURCE	LIMIT
Management groups per Azure AD tenant	10,000
Subscriptions per management group	Unlimited.
Levels of management group hierarchy	Root level plus 6 levels <sup>1</sup>
Direct parent management group per management group	One
<a href="#">Management group level deployments</a> per location	800 <sup>2</sup>
Locations of <a href="#">Management group level deployments</a>	10

<sup>1</sup>The 6 levels don't include the subscription level.

<sup>2</sup>If you reach the limit of 800 deployments, delete deployments from the history that are no longer needed. To delete management group level deployments, use [Remove-AzManagementGroupDeployment](#) or [az deployment mg delete](#).

## Subscription limits

The following limits apply when you use Azure Resource Manager and Azure resource groups.

RESOURCE	LIMIT
Subscriptions <a href="#">associated with an Azure Active Directory tenant</a>	Unlimited
<a href="#">Coadministrators</a> per subscription	Unlimited
<a href="#">Resource groups</a> per subscription	980
Azure Resource Manager API request size	4,194,304 bytes
Tags per subscription <sup>1</sup>	50
Unique tag calculations per subscription <sup>1</sup>	80,000
<a href="#">Subscription-level deployments</a> per location	800 <sup>2</sup>
Locations of <a href="#">Subscription-level deployments</a>	10

<sup>1</sup>You can apply up to 50 tags directly to a subscription. However, the subscription can contain an unlimited number of tags that are applied to resource groups and resources within the subscription. The number of tags per resource or resource group is limited to 50. Resource Manager returns a [list of unique tag name and values](#) in the subscription only when the number of tags is 80,000 or less. You still can find a resource by tag when the number exceeds 80,000.

<sup>2</sup>Deployments are automatically deleted from the history as you near the limit. For more information, see [Automatic deletions from deployment history](#).

## Resource group limits

RESOURCE	LIMIT
Resources per <a href="#">resource group</a>	Resources aren't limited by resource group. Instead, they're limited by resource type in a resource group. See next row.
Resources per resource group, per resource type	800 - Some resource types can exceed the 800 limit. See <a href="#">Resources not limited to 800 instances per resource group</a> .
Deployments per resource group in the deployment history	800 <sup>1</sup>
Resources per deployment	800
Management locks per unique <a href="#">scope</a>	20
Number of tags per resource or resource group	50
Tag key length	512
Tag value length	256

<sup>1</sup>Deployments are automatically deleted from the history as you near the limit. Deleting an entry from the deployment history doesn't affect the deployed resources. For more information, see [Automatic deletions from deployment history](#).

#### Template limits

VALUE	LIMIT
Parameters	256
Variables	256
Resources (including copy count)	800
Outputs	64
Template expression	24,576 chars
Resources in exported templates	200
Template size	4 MB
Parameter file size	4 MB

You can exceed some template limits by using a nested template. For more information, see [Use linked templates when you deploy Azure resources](#). To reduce the number of parameters, variables, or outputs, you can combine several values into an object. For more information, see [Objects as parameters](#).

You may get an error with a template or parameter file of less than 4 MB, if the total size of the request is too large. For more information about how to simplify your template to avoid a large request, see [Resolve errors for job size exceeded](#).

## Active Directory limits

Here are the usage constraints and other service limits for the Azure Active Directory (Azure AD) service.

CATEGORY	LIMIT
Tenants	A single user can belong to a maximum of 500 Azure AD tenants as a member or a guest. A single user can create a maximum of 200 directories.
Domains	You can add no more than 5000 managed domain names. If you set up all of your domains for federation with on-premises Active Directory, you can add no more than 2500 domain names in each tenant.
Resources	<ul style="list-style-type: none"><li>A maximum of 50,000 Azure AD resources can be created in a single tenant by users of the Free edition of Azure Active Directory by default. If you have at least one verified domain, the default Azure AD service quota for your organization is extended to 300,000 Azure AD resources. Azure AD service quota for organizations created by self-service sign-up remains 50,000 Azure AD resources even after you performed an internal admin takeover and the organization is converted to a managed tenant with at least one verified domain. This service limit is unrelated to the pricing tier limit of 500,000 resources on the Azure AD pricing page. To go beyond the default quota, you must contact Microsoft Support.</li><li>A non-admin user can create no more than 250 Azure AD resources. Both active resources and deleted resources that are available to restore count toward this quota. Only deleted Azure AD resources that were deleted fewer than 30 days ago are available to restore. Deleted Azure AD resources that are no longer available to restore count toward this quota at a value of one-quarter for 30 days. If you have developers who are likely to repeatedly exceed this quota in the course of their regular duties, you can <a href="#">create and assign a custom role</a> with permission to create a limitless number of app registrations.</li></ul>
Schema extensions	<ul style="list-style-type: none"><li>String-type extensions can have a maximum of 256 characters.</li><li>Binary-type extensions are limited to 256 bytes.</li><li>Only 100 extension values, across <i>all</i> types and <i>all</i> applications, can be written to any single Azure AD resource.</li><li>Only User, Group, TenantDetail, Device, Application, and ServicePrincipal entities can be extended with string-type or binary-type single-valued attributes.</li></ul>

CATEGORY	LIMIT
Applications	<ul style="list-style-type: none"> <li>• A maximum of 100 users can be owners of a single application.</li> <li>• A user, group, or service principal can have a maximum of 1,500 app role assignments. The limitation is on the service principal, user, or group across all app roles and not on a limit on the number of assignments on a single app role.</li> <li>• Password-based single sign-on (SSO) app has a limit of 48 users, which means that there is a limit of 48 keys for username/password pairs per app. If you want to add additional users, see the troubleshooting instructions in <a href="#">Troubleshoot password-based single sign-on in Azure AD</a>.</li> <li>• A user can only have a maximum of 48 apps where they have username and password credentials configured.</li> </ul>
Application Manifest	A maximum of 1200 entries can be added in the Application Manifest.

CATEGORY	LIMIT
Groups	<ul style="list-style-type: none"> <li>A non-admin user can create a maximum of 250 groups in an Azure AD organization. Any Azure AD admin who can manage groups in the organization can also create unlimited number of groups (up to the Azure AD object limit). If you assign a role to remove the limit for a user, assign them to a less privileged built-in role such as User Administrator or Groups Administrator.</li> <li>An Azure AD organization can have a maximum of 5000 dynamic groups.</li> <li>A maximum of 300 role-assignable groups can be created in a single Azure AD organization (tenant).</li> <li>A maximum of 100 users can be owners of a single group.</li> <li>Any number of Azure AD resources can be members of a single group.</li> <li>A user can be a member of any number of groups.</li> <li>By default, the number of members in a group that you can synchronize from your on-premises Active Directory to Azure Active Directory by using Azure AD Connect is limited to 50,000 members. If you need to sync a group membership that's over this limit, you must onboard the <a href="#">Azure AD Connect Sync V2 endpoint API</a>.</li> <li>Nested Groups in Azure AD are not supported within all scenarios</li> <li>Group expiration policy can be assigned to a maximum of 500 Microsoft 365 groups, when selecting a list of groups. There is no limit when the policy is applied to all Microsoft 365 groups.</li> </ul> <p>At this time the following are the supported scenarios with nested groups.</p> <ul style="list-style-type: none"> <li>One group can be added as a member of another group and you can achieve group nesting.</li> <li>Group membership claims (when an app is configured to receive group membership claims in the token, nested groups in which the signed-in user is a member are included)</li> <li>Conditional access (when a conditional access policy has a group scope)</li> <li>Restricting access to self-serve password reset</li> <li>Restricting which users can do Azure AD Join and device registration</li> </ul> <p>The following scenarios DO NOT support nested groups:</p> <ul style="list-style-type: none"> <li>App role assignment (assigning groups to an app is supported, but groups nested within the directly assigned group will not have access), both for access and for provisioning</li> <li>Group-based licensing (assigning a license automatically to all members of a group)</li> <li>Microsoft 365 Groups.</li> </ul>

CATEGORY	LIMIT
Application Proxy	<ul style="list-style-type: none"> <li>A maximum of 500 transactions per second per App Proxy application</li> <li>A maximum of 750 transactions per second for the Azure AD organization</li> </ul> <p>A transaction is defined as a single http request and response for a unique resource. When throttled, clients will receive a 429 response (too many requests).</p>
Access Panel	There's no limit to the number of applications that can be seen in the Access Panel per user regardless of assigned licenses.
Reports	A maximum of 1,000 rows can be viewed or downloaded in any report. Any additional data is truncated.
Administrative units	An Azure AD resource can be a member of no more than 30 administrative units.
Azure AD roles and permissions	<ul style="list-style-type: none"> <li>A maximum of 30 <a href="#">Azure AD custom roles</a> can be created in an Azure AD organization.</li> <li>A maximum of 100 Azure AD custom role assignments for a single principal at tenant scope.</li> <li>A maximum of 100 Azure AD built-in role assignments for a single principal at non-tenant scope (such as administrative unit or Azure AD object). There is no limit for Azure AD built-in role assignments at tenant scope.</li> <li>A group can't be added as a <a href="#">group owner</a>.</li> <li>A user's ability to read other users' tenant information can be restricted only by the Azure AD organization-wide switch to disable all non-admin users' access to all tenant information (not recommended). For more information, see <a href="#">To restrict the default permissions for member users</a>.</li> <li>It may take up to 15 minutes or signing out/signing in before admin role membership additions and revocations take effect.</li> </ul>

## API Management limits

RESOURCE	LIMIT
Maximum number of scale units	12 per region <sup>1</sup>
Cache size	5 GiB per unit <sup>2</sup>
Concurrent back-end connections <sup>3</sup> per HTTP authority	2,048 per unit <sup>4</sup>
Maximum cached response size	2 MiB
Maximum policy document size	256 KiB <sup>5</sup>

RESOURCE	LIMIT
Maximum custom gateway domains per service instance <sup>6</sup>	20
Maximum number of CA certificates per service instance <sup>7</sup>	10
Maximum number of service instances per subscription <sup>8</sup>	20
Maximum number of subscriptions per service instance <sup>8</sup>	500
Maximum number of client certificates per service instance <sup>8</sup>	50
Maximum number of APIs per service instance <sup>8</sup>	50
Maximum number of API management operations per service instance <sup>8</sup>	1,000
Maximum total request duration <sup>8</sup>	30 seconds
Maximum request payload size <sup>8</sup>	1 GiB
Maximum buffered payload size <sup>8</sup>	2 MiB
Maximum request URL size <sup>9</sup>	4096 bytes
Maximum length of URL path segment <sup>10</sup>	260 characters
Maximum size of API schema used by <a href="#">validation policy</a> <sup>10</sup>	4 MB
Maximum size of request or response body in <a href="#">validate-content policy</a>	100 KB
Maximum number of self-hosted gateways <sup>11</sup>	25

<sup>1</sup>Scaling limits depend on the pricing tier. For details on the pricing tiers and their scaling limits, see [API Management pricing](#).

<sup>2</sup>Per unit cache size depends on the pricing tier. To see the pricing tiers and their scaling limits, see [API Management pricing](#).

<sup>3</sup>Connections are pooled and reused unless explicitly closed by the back end.

<sup>4</sup>This limit is per unit of the Basic, Standard, and Premium tiers. The Developer tier is limited to 1,024. This limit doesn't apply to the Consumption tier.

<sup>5</sup>This limit applies to the Basic, Standard, and Premium tiers. In the Consumption tier, policy document size is limited to 16 KiB.

<sup>6</sup>Multiple custom domains are supported in the Developer and Premium tiers only.

<sup>7</sup>CA certificates are not supported in the Consumption tier.

<sup>8</sup>This limit applies to the Consumption tier only. There are no limits in these categories for other tiers.

<sup>9</sup>Applies to the Consumption tier only. Includes an up to 2048 bytes long query string.

<sup>10</sup>To increase this limit, please contact [support](#).

<sup>11</sup>Self-hosted gateways are supported in the Developer and Premium tiers only. The limit applies to the number of [self-hosted gateway resources](#). To raise this limit please contact [support](#). Note, that the number of nodes (or replicas) associated with a self-hosted gateway resource is unlimited in the Premium tier and capped at a single node in the Developer tier.

## App Service limits

RESOURCE	FREE	SHARED	BASIC	STANDARD	PREMIUM (V1-V3)	ISOLATED
Web, mobile, or API apps per Azure App Service plan <sup>1</sup>	10	100	Unlimited <sup>2</sup>	Unlimited <sup>2</sup>	Unlimited <sup>2</sup>	Unlimited <sup>2</sup>
App Service plan	10 per region	10 per resource group	100 per resource group	100 per resource group	100 per resource group	100 per resource group
Compute instance type	Shared	Shared	Dedicated <sup>3</sup>	Dedicated <sup>3</sup>	Dedicated <sup>3</sup>	Dedicated <sup>3</sup>
Scale out (maximum instances)	1 shared	1 shared	3 dedicated <sup>3</sup>	10 dedicated <sup>3</sup>	20 dedicated for v1; 30 dedicated for v2 and v3. <sup>3</sup>	100 dedicated <sup>4</sup>
Storage <sup>5</sup>	1 GB <sup>5</sup>	1 GB <sup>5</sup>	10 GB <sup>5</sup>	50 GB <sup>5</sup>	250 GB <sup>5</sup>	1 TB <sup>5</sup> The available storage quota is 999 GB.
CPU time (5 minutes) <sup>6</sup>	3 minutes	3 minutes	Unlimited, pay at standard rates	Unlimited, pay at standard rates	Unlimited, pay at standard rates	Unlimited, pay at standard rates
CPU time (day) <sup>6</sup>	60 minutes	240 minutes	Unlimited, pay at standard rates	Unlimited, pay at standard rates	Unlimited, pay at standard rates	Unlimited, pay at standard rates
Memory (1 hour)	1,024 MB per App Service plan	1,024 MB per app	N/A	N/A	N/A	N/A
Bandwidth	165 MB	Unlimited, data transfer rates apply	Unlimited, data transfer rates apply	Unlimited, data transfer rates apply	Unlimited, data transfer rates apply	Unlimited, data transfer rates apply
Application architecture	32-bit	32-bit	32-bit/64-bit	32-bit/64-bit	32-bit/64-bit	32-bit/64-bit
Web sockets per instance <sup>7</sup>	5	35	350	Unlimited	Unlimited	Unlimited
Outbound IP connections per instance	600	600	Depends on instance size <sup>8</sup>	Depends on instance size <sup>8</sup>	Depends on instance size <sup>8</sup>	16,000

RESOURCE	FREE	SHARED	BASIC	STANDARD	PREMIUM (V1-V3)	ISOLATED
Concurrent debugger connections per application	1	1	1	5	5	5
App Service Certificates per subscription <sup>9</sup>	Not supported	Not supported	10	10	10	10
Custom domains per app	0 (azurewebsites.net subdomain only)	500	500	500	500	500
Custom domain SSL support	Not supported, wildcard certificate for *.azurewebsites.net available by default	Not supported, wildcard certificate for *.azurewebsites.net available by default	Unlimited SNI SSL connections	Unlimited SNI SSL and 1 IP SSL connections included	Unlimited SNI SSL and 1 IP SSL connections included	Unlimited SNI SSL and 1 IP SSL connections included
Hybrid connections			5 per plan	25 per plan	220 per app	220 per app
Virtual Network Integration				X	X	X
Private Endpoints					100 per app	
Integrated load balancer		X	X	X	X	X <sup>10</sup>
Access restrictions	512 rules per app	512 rules per app	512 rules per app	512 rules per app	512 rules per app	512 rules per app
Always On			X	X	X	X
Scheduled backups				Scheduled backups every 2 hours, a maximum of 12 backups per day (manual + scheduled)	Scheduled backups every hour, a maximum of 50 backups per day (manual + scheduled)	Scheduled backups every hour, a maximum of 50 backups per day (manual + scheduled)
Autoscale				X	X	X

RESOURCE	FREE	SHARED	BASIC	STANDARD	PREMIUM (V1-V3)	ISOLATED
WebJobs <sup>11</sup>	X	X	X	X	X	X
Endpoint monitoring			X	X	X	X
Staging slots per app				5	20	20
Testing in Production				X	X	X
Diagnostic Logs	X	X	X	X	X	X
Kudu	X	X	X	X	X	X
Authentication and Authorization	X	X	X	X	X	X
App Service Managed Certificates (Public Preview) <sup>12</sup>			X	X	X	X
SLA			99.95%	99.95%	99.95%	99.95%

<sup>1</sup> Apps and storage quotas are per App Service plan unless noted otherwise.

<sup>2</sup> The actual number of apps that you can host on these machines depends on the activity of the apps, the size of the machine instances, and the corresponding resource utilization.

<sup>3</sup> Dedicated instances can be of different sizes. For more information, see [App Service pricing](#).

<sup>4</sup> More are allowed upon request.

<sup>5</sup> The storage limit is the total content size across all apps in the same App service plan. The total content size of all apps across all App service plans in a single resource group and region cannot exceed 500 GB. The file system quota for App Service hosted apps is determined by the aggregate of App Service plans created in a region and resource group.

<sup>6</sup> These resources are constrained by physical resources on the dedicated instances (the instance size and the number of instances).

<sup>7</sup> If you scale an app in the Basic tier to two instances, you have 350 concurrent connections for each of the two instances. For Standard tier and above, there are no theoretical limits to web sockets, but other factors can limit the number of web sockets. For example, maximum concurrent requests allowed (defined by `maxConcurrentRequestsPerCpu`) are: 7,500 per small VM, 15,000 per medium VM (7,500 x 2 cores), and 75,000 per large VM (18,750 x 4 cores).

<sup>8</sup> The maximum IP connections are per instance and depend on the instance size: 1,920 per B1/S1/P1V3 instance, 3,968 per B2/S2/P2V3 instance, 8,064 per B3/S3/P3V3 instance.

<sup>9</sup> The App Service Certificate quota limit per subscription can be increased via a support request to a maximum

limit of 200.

<sup>10</sup> App Service Isolated SKUs can be internally load balanced (ILB) with Azure Load Balancer, so there's no public connectivity from the internet. As a result, some features of an ILB Isolated App Service must be used from machines that have direct access to the ILB network endpoint.

<sup>11</sup> Run custom executables and/or scripts on demand, on a schedule, or continuously as a background task within your App Service instance. Always On is required for continuous WebJobs execution. There's no predefined limit on the number of WebJobs that can run in an App Service instance. There are practical limits that depend on what the application code is trying to do.

<sup>12</sup> Naked domains aren't supported. Only issuing standard certificates (wildcard certificates aren't available). Limited to only one free certificate per custom domain.

## Automation limits

### Process automation

RESOURCE	LIMIT	NOTES
Maximum number of new jobs that can be submitted every 30 seconds per Azure Automation account (nonscheduled jobs)	100	When this limit is reached, the subsequent requests to create a job fail. The client receives an error response.
Maximum number of concurrent running jobs at the same instance of time per Automation account (nonscheduled jobs)	200	When this limit is reached, the subsequent requests to create a job fail. The client receives an error response.
Maximum storage size of job metadata for a 30-day rolling period	10 GB (approximately 4 million jobs)	When this limit is reached, the subsequent requests to create a job fail.
Maximum job stream limit	1 MiB	A single stream cannot be larger than 1 MiB.
Maximum number of modules that can be imported every 30 seconds per Automation account	5	
Maximum size of a module	100 MB	
Maximum size of a node configuration file	1 MB	Applies to state configuration
Job run time, Free tier	500 minutes per subscription per calendar month	
Maximum amount of disk space allowed per sandbox <sup>1</sup>	1 GB	Applies to Azure sandboxes only.
Maximum amount of memory given to a sandbox <sup>1</sup>	400 MB	Applies to Azure sandboxes only.
Maximum number of network sockets allowed per sandbox <sup>1</sup>	1,000	Applies to Azure sandboxes only.

RESOURCE	LIMIT	NOTES
Maximum runtime allowed per runbook <sup>1</sup>	3 hours	Applies to Azure sandboxes only.
Maximum number of Automation accounts in a subscription	No limit	
Maximum number of system hybrid runbook workers per Automation Account	4,000	
Maximum number of user hybrid runbook workers per Automation Account	4,000	
Maximum number of concurrent jobs that can be run on a single Hybrid Runbook Worker	50	
Maximum runbook job parameter size	512 kilobytes	
Maximum runbook parameters	50	If you reach the 50-parameter limit, you can pass a JSON or XML string to a parameter and parse it with the runbook.
Maximum webhook payload size	512 kilobytes	
Maximum days that job data is retained	30 days	
Maximum PowerShell workflow state size	5 MB	Applies to PowerShell workflow runbooks when checkpointing workflow.
Maximum number of tags supported by an Automation account	15	

<sup>1</sup>A sandbox is a shared environment that can be used by multiple jobs. Jobs that use the same sandbox are bound by the resource limitations of the sandbox.

#### Change Tracking and Inventory

The following table shows the tracked item limits per machine for change tracking.

RESOURCE	LIMIT	NOTES
File	500	
File size	5 MB	
Registry	250	
Windows software	250	Doesn't include software updates.

RESOURCE	LIMIT	NOTES
Linux packages	1,250	
Services	250	
Daemon	250	

#### Update Management

The following table shows the limits for Update Management.

RESOURCE	LIMIT	NOTES
Number of machines per update deployment	1000	
Number of dynamic groups per update deployment	500	

## Azure App Configuration

RESOURCE	LIMIT
Configuration stores - Free tier	1 per subscription
Configuration stores - Standard tier	unlimited per subscription
Configuration store requests - Free tier	1,000 requests per day
Configuration store requests - Standard tier	Throttling starts at 20,000 requests per hour
Storage - Free tier	10 MB
Storage - Standard tier	1 GB
keys and values	10 KB for a single key-value item

## Azure API for FHIR service limits

Azure API for FHIR is a managed, standards-based, compliant API for clinical health data that enables solutions for actionable analytics and machine learning.

QUOTA NAME	DEFAULT LIMIT	MAXIMUM LIMIT	NOTES
Request Units (RUs)	10,000 RUs	<a href="#">Contact support</a> Maximum available is 1,000,000.	You need a minimum of 400 RUs or 40 RUs/GB, whichever is larger.
Concurrent connections	15 concurrent connections on two instances (for a total of 30 concurrent requests)	<a href="#">Contact support</a>	

Quota name	Default limit	Maximum limit	Notes
Azure API for FHIR Service Instances per Subscription	10	Contact support	

## Azure Cache for Redis limits

Resource	Limit
Cache size	1.2 TB
Databases	64
Maximum connected clients	40,000
Azure Cache for Redis replicas, for high availability	1
Shards in a premium cache with clustering	10

Azure Cache for Redis limits and sizes are different for each pricing tier. To see the pricing tiers and their associated sizes, see [Azure Cache for Redis pricing](#).

For more information on Azure Cache for Redis configuration limits, see [Default Redis server configuration](#).

Because configuration and management of Azure Cache for Redis instances is done by Microsoft, not all Redis commands are supported in Azure Cache for Redis. For more information, see [Redis commands not supported in Azure Cache for Redis](#).

## Azure Cloud Services limits

Resource	Limit
<a href="#">Web or worker roles per deployment</a> <sup>1</sup>	25
<a href="#">Instance input endpoints</a> per deployment	25
<a href="#">Input endpoints</a> per deployment	25
<a href="#">Internal endpoints</a> per deployment	25
<a href="#">Hosted service certificates</a> per deployment	199

<sup>1</sup>Each Azure Cloud Service with web or worker roles can have two deployments, one for production and one for staging. This limit refers to the number of distinct roles, that is, configuration. This limit doesn't refer to the number of instances per role, that is, scaling.

## Azure Cognitive Search limits

Pricing tiers determine the capacity and limits of your search service. Tiers include:

- **Free** multi-tenant service, shared with other Azure subscribers, is intended for evaluation and small development projects.
- **Basic** provides dedicated computing resources for production workloads at a smaller scale, with up to three

replicas for highly available query workloads.

- **Standard**, which includes S1, S2, S3, and S3 High Density, is for larger production workloads. Multiple levels exist within the Standard tier so that you can choose a resource configuration that best matches your workload profile.

## Limits per subscription

You can create multiple services within a subscription. Each one can be provisioned at a specific tier. You're limited only by the number of services allowed at each tier. For example, you could create up to 12 services at the Basic tier and another 12 services at the S1 tier within the same subscription. For more information about tiers, see [Choose an SKU or tier for Azure Cognitive Search](#).

Maximum service limits can be raised upon request. If you need more services within the same subscription, contact Azure Support.

RESOURCE	FREE <sup>1</sup>	BASIC	S1	S2	S3	S3 HD	L1	L2
Maximum services	1	16	16	8	6	6	6	6
Maximum scale in search units (SU) <sup>2</sup>	N/A	3 SU	36 SU	36 SU	36 SU	36 SU	36 SU	36 SU

<sup>1</sup> Free is based on shared, not dedicated, resources. Scale-up is not supported on shared resources.

<sup>2</sup> Search units are billing units, allocated as either a *replica* or a *partition*. You need both resources for storage, indexing, and query operations. To learn more about SU computations, see [Scale resource levels for query and index workloads](#).

## Limits per search service

A search service is constrained by disk space or by a hard limit on the maximum number of indexes or indexers, whichever comes first. The following table documents storage limits. For maximum object limits, see [Limits by resource](#).

RESOURCE	FREE	BASIC <sup>1</sup>	S1	S2	S3	S3 HD	L1	L2
Service level agreement (SLA) <sup>2</sup>	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Storage per partition	50 MB	2 GB	25 GB	100 GB	200 GB	200 GB	1 TB	2 TB
Partitions per service	N/A	1	12	12	12	3	12	12
Partition size	N/A	2 GB	25 GB	100 GB	200 GB	200 GB	1 TB	2 TB

RESOURCE	FREE	BASIC	S1	S2	S3	S3 HD	L1	L2
Replicas	N/A	3	12	12	12	12	12	12

<sup>1</sup> Basic has one fixed partition. Additional search units can be used to add replicas for larger query volumes.

<sup>2</sup> Service level agreements are in effect for billable services on dedicated resources. Free services and preview features have no SLA. For billable services, SLAs take effect when you provision sufficient redundancy for your service. Two or more replicas are required for query (read) SLAs. Three or more replicas are required for query and indexing (read-write) SLAs. The number of partitions isn't an SLA consideration.

To learn more about limits on a more granular level, such as document size, queries per second, keys, requests, and responses, see [Service limits in Azure Cognitive Search](#).

## Azure Cognitive Services limits

The following limits are for the number of Cognitive Services resources per Azure subscription. Each of the Cognitive Services may have other limitations, for more information, see [Azure Cognitive Services](#).

TYPE	LIMIT	EXAMPLE
A mixture of Cognitive Services resources	Maximum of 200 total Cognitive Services resources per region.	100 Computer Vision resources in West US, 50 Speech Service resources in West US, and 50 Text Analytics resources in West US.
A single type of Cognitive Services resources.	Maximum of 100 resources per region	100 Computer Vision resources in West US 2, and 100 Computer Vision resources in East US.

## Azure Cosmos DB limits

For Azure Cosmos DB limits, see [Limits in Azure Cosmos DB](#).

## Azure Data Explorer limits

The following table describes the maximum limits for Azure Data Explorer clusters.

RESOURCE	LIMIT
Clusters per region per subscription	20
Instances per cluster	1000
Number of databases in a cluster	10,000
Number of follower clusters (data share consumers) per leader cluster (data share producer)	70

The following table describes the limits on management operations performed on Azure Data Explorer clusters.

SCOPE	OPERATION	LIMIT
Cluster	read (for example, get a cluster)	500 per 5 minutes

SCOPE	OPERATION	LIMIT
Cluster	write (for example, create a database)	1000 per hour

## Azure Database for MySQL

For Azure Database for MySQL limits, see [Limitations in Azure Database for MySQL](#).

## Azure Database for PostgreSQL

For Azure Database for PostgreSQL limits, see [Limitations in Azure Database for PostgreSQL](#).

## Azure Functions limits

RESOURCE	CONSUMPTION PLAN	PREMIUM PLAN	DEDICATED PLAN	ASE	KUBERNETES
Default <a href="#">timeout duration</a> (min)	5	30	30 <sup>1</sup>	30	30
Max <a href="#">timeout duration</a> (min)	10	unbounded <sup>7</sup>	unbounded <sup>2</sup>	unbounded	unbounded
Max outbound connections (per instance)	600 active (1200 total)	unbounded	unbounded	unbounded	unbounded
Max request size (MB) <sup>3</sup>	100	100	100	100	Depends on cluster
Max query string length <sup>3</sup>	4096	4096	4096	4096	Depends on cluster
Max request URL length <sup>3</sup>	8192	8192	8192	8192	Depends on cluster
ACU per instance	100	210-840	100-840	210-250 <sup>8</sup>	<a href="#">AKS pricing</a>
Max memory (GB per instance)	1.5	3.5-14	1.75-14	3.5 - 14	Any node is supported
Max instance count	200	100 <sup>9</sup>	varies by SKU <sup>10</sup>	100 <sup>10</sup>	Depends on cluster
Function apps per plan	100	100	unbounded <sup>4</sup>	unbounded	unbounded
<a href="#">App Service plans</a>	100 per <a href="#">region</a>	100 per resource group	100 per resource group	-	-
Storage <sup>5</sup>	5 TB	250 GB	50-1000 GB	1 TB	n/a

RESOURCE	CONSUMPTION PLAN	PREMIUM PLAN	DEDICATED PLAN	ASE	KUBERNETES
Custom domains per app	500 <sup>6</sup>	500	500	500	n/a
Custom domain SSL support	unbounded SNI SSL connection included	unbounded SNI SSL and 1 IP SSL connections included	unbounded SNI SSL and 1 IP SSL connections included	unbounded SNI SSL and 1 IP SSL connections included	n/a

<sup>1</sup> By default, the timeout for the Functions 1.x runtime in an App Service plan is unbounded.

<sup>2</sup> Requires the App Service plan be set to [Always On](#). Pay at standard [rates](#).

<sup>3</sup> These limits are [set in the host](#).

<sup>4</sup> The actual number of function apps that you can host depends on the activity of the apps, the size of the machine instances, and the corresponding resource utilization.

<sup>5</sup> The storage limit is the total content size in temporary storage across all apps in the same App Service plan. Consumption plan uses Azure Files for temporary storage.

<sup>6</sup> When your function app is hosted in a [Consumption plan](#), only the CNAME option is supported. For function apps in a [Premium plan](#) or an [App Service plan](#), you can map a custom domain using either a CNAME or an A record.

<sup>7</sup> Guaranteed for up to 60 minutes.

<sup>8</sup> Workers are roles that host customer apps. Workers are available in three fixed sizes: One vCPU/3.5 GB RAM; Two vCPU/7 GB RAM; Four vCPU/14 GB RAM.

<sup>9</sup> When running on Linux in a Premium plan, you're currently limited to 20 instances.

<sup>10</sup> See [App Service limits](#) for details.

For more information, see [Functions Hosting plans comparison](#).

## Azure Kubernetes Service limits

RESOURCE	LIMIT
Maximum clusters per subscription	5000
Maximum nodes per cluster with Virtual Machine Availability Sets and Basic Load Balancer SKU	100
Maximum nodes per cluster with Virtual Machine Scale Sets and <a href="#">Standard Load Balancer SKU</a>	1000 (across all <a href="#">node pools</a> )
Maximum node pools per cluster	100
Maximum pods per node: <a href="#">Basic networking</a> with Kubelet	Maximum: 250 Azure CLI default: 110 Azure Resource Manager template default: 110 Azure portal deployment default: 30
Maximum pods per node: <a href="#">Advanced networking</a> with Azure Container Networking Interface	Maximum: 250 Default: 30
Open Service Mesh (OSM) AKS addon preview	Kubernetes Cluster Version: 1.19+ <sup>1</sup> OSM controllers per cluster: 1 <sup>1</sup> Pods per OSM controller: 500 <sup>1</sup> Kubernetes service accounts managed by OSM: 50 <sup>1</sup>

<sup>1</sup>The OSM add-on for AKS is in a preview state and will undergo additional enhancements before general availability (GA). During the preview phase, it's recommended to not surpass the limits shown.

## Azure Machine Learning limits

The latest values for Azure Machine Learning Compute quotas can be found in the [Azure Machine Learning quota page](#)

## Azure Maps limits

The following table shows the usage limit for the Azure Maps S0 pricing tier. Usage limit depends on the pricing tier.

RESOURCE	S0 PRICING TIER LIMIT
Maximum request rate per subscription	50 requests per second

The following table shows the cumulative data size limit for Azure Maps accounts in an Azure subscription. The Azure Maps Data service is available only at the S1 pricing tier.

RESOURCE	LIMIT
Maximum storage per Azure subscription	1 GB
Maximum size per file upload	100 MB

For more information on the Azure Maps pricing tiers, see [Azure Maps pricing](#).

## Azure Monitor limits

### Alerts

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Metric alerts (classic)	100 active alert rules per subscription.	Call support
Metric alerts	5,000 active alert rules per subscription in Azure public, Azure China 21Vianet and Azure Government clouds. If you are hitting this limit, explore if you can use <a href="#">same type multi-resource alerts</a> . 5,000 metric time-series per alert rule.	Call support.
Activity log alerts	100 active alert rules per subscription (cannot be increased).	Same as default
Log alerts	1000 active alert rules per subscription. 1000 active alert rules per resource.	Call support
Alert rules and Action rules description length	Log search alerts 4096 characters All other 2048 characters	Same as default

### Alerts API

Azure Monitor Alerts have several throttling limits to protect against users making an excessive number of calls. Such behavior can potentially overload the system backend resources and jeopardize service responsiveness. The following limits are designed to protect customers from interruptions and ensure consistent service level. The user throttling and limits are designed to impact only extreme usage scenario and should not be relevant for typical usage.

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
GET alertsSummary	50 calls per minute per subscription	Same as default
GET alerts (without specifying an alert ID)	100 calls per minute per subscription	Same as default
All other calls	1000 calls per minute per subscription	Same as default

## Action groups

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Azure app push	10 Azure app actions per action group.	Same as Default
Email	1,000 email actions in an action group. No more than 100 emails in an hour. Also see the <a href="#">rate limiting information</a> .	Same as Default
ITSM	10 ITSM actions in an action group.	Same as Default
Logic app	10 logic app actions in an action group.	Same as Default
Runbook	10 runbook actions in an action group.	Same as Default
SMS	10 SMS actions in an action group. No more than 1 SMS message every 5 minutes. Also see the <a href="#">rate limiting information</a> .	Same as Default
Voice	10 voice actions in an action group. No more than 1 voice call every 5 minutes. Also see the <a href="#">rate limiting information</a> .	Same as Default
Webhook	10 webhook actions in an action group. Maximum number of webhook calls is 1500 per minute per subscription. Other limits are available at <a href="#">action-specific information</a> .	Same as Default

## Autoscale

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Autoscale settings	100 per region per subscription.	Same as default
Autoscale profiles	20 profiles per autoscale setting.	Same as default

## Log queries and language

### General query limits

LIMIT	DESCRIPTION
Query language	Azure Monitor uses the same <a href="#">Kusto query language</a> as Azure Data Explorer. See <a href="#">Azure Monitor log query language differences</a> for KQL language elements not supported in Azure Monitor.
Azure regions	Log queries can experience excessive overhead when data spans Log Analytics workspaces in multiple Azure regions. See <a href="#">Query limits</a> for details.
Cross resource queries	Maximum number of Application Insights resources and Log Analytics workspaces in a single query limited to 100. Cross-resource query is not supported in View Designer. Cross-resource query in log alerts is supported in the new scheduledQueryRules API. See <a href="#">Cross-resource query limits</a> for details.

### User query throttling

Azure Monitor has several throttling limits to protect against users sending an excessive number of queries. Such behavior can potentially overload the system backend resources and jeopardize service responsiveness. The following limits are designed to protect customers from interruptions and ensure consistent service level. The user throttling and limits are designed to impact only extreme usage scenario and should not be relevant for typical usage.

MEASURE	LIMIT PER USER	DESCRIPTION
Concurrent queries	5	If there are already 5 queries running for the user, any new queries are placed in a per-user concurrency queue. When one of the running queries ends, the next query will be pulled from the queue and started. This does not include queries from alert rules.
Time in concurrency queue	3 minutes	If a query sits in the queue for more than 3 minutes without being started, it will be terminated with an HTTP error response with code 429.
Total queries in concurrency queue	200	Once the number of queries in the queue reaches 200, any additional queries will be rejected with an HTTP error code 429. This number is in addition to the 5 queries that can be running simultaneously.
Query rate	200 queries per 30 seconds	This is the overall rate that queries can be submitted by a single user to all workspaces. This limit applies to programmatic queries or queries initiated by visualization parts such as Azure dashboards and the Log Analytics workspace summary page.

- Optimize your queries as described in [Optimize log queries in Azure Monitor](#).
- Dashboards and workbooks can contain multiple queries in a single view that generate a burst of queries every time they load or refresh. Consider breaking them up into multiple views that load on demand.
- In Power BI, consider extracting only aggregated results rather than raw logs.

## Log Analytics workspaces

### Data collection volume and retention

TIER	LIMIT PER DAY	DATA RETENTION	COMMENT
Current Per GB pricing tier (introduced April 2018)	No limit	30 - 730 days	Data retention beyond 31 days is available for additional charges. Learn more about Azure Monitor pricing.
Legacy Free tiers (introduced April 2016)	500 MB	7 days	When your workspace reaches the 500 MB per day limit, data ingestion stops and resumes at the start of the next day. A day is based on UTC. Note that data collected by Azure Security Center is not included in this 500 MB per day limit and will continue to be collected above this limit.
Legacy Standalone Per GB tier (introduced April 2016)	No limit	30 to 730 days	Data retention beyond 31 days is available for additional charges. Learn more about Azure Monitor pricing.
Legacy Per Node (OMS) (introduced April 2016)	No limit	30 to 730 days	Data retention beyond 31 days is available for additional charges. Learn more about Azure Monitor pricing.
Legacy Standard tier	No limit	30 days	Retention can't be adjusted
Legacy Premium tier	No limit	365 days	Retention can't be adjusted

### Number of workspaces per subscription.

PRICING TIER	WORKSPACE LIMIT	COMMENTS
Free tier	10	This limit can't be increased.
All other tiers	No limit	You're limited by the number of resources within a resource group and the number of resource groups per subscription.

### Azure portal

CATEGORY	LIMIT	COMMENTS
Maximum records returned by a log query	30,000	Reduce results using query scope, time range, and filters in the query.

## Data Collector API

CATEGORY	LIMIT	COMMENTS
Maximum size for a single post	30 MB	Split larger volumes into multiple posts.
Maximum size for field values	32 KB	Fields longer than 32 KB are truncated.

## Query API

CATEGORY	LIMIT	COMMENTS
Maximum records returned in a single query	500,000	
Maximum size of data returned	~104 MB (~100 MiB)	
Maximum query running time	10 minutes	See <a href="#">Timeouts</a> for details.
Maximum request rate	200 requests per 30 seconds per Azure AD user or client IP address	See <a href="#">Rate limits</a> for details.

## Azure Monitor Logs connector

CATEGORY	LIMIT	COMMENTS
Max size of data	~16.7 MB (~16 MiB)	Connector infrastructure dictates that limit is set lower than query API limit
Max number of records	500,000	
Max query timeout	110 second	
Charts		Visualization in Logs page and the connector are using different charting libraries and some functionality isn't available in the connector currently.

## General workspace limits

CATEGORY	LIMIT	COMMENTS
Maximum columns in a table	500	
Maximum characters for column name	500	

## Data ingestion volume rate

Azure Monitor is a high scale data service that serves thousands of customers sending terabytes of data each

month at a growing pace. The volume rate limit intends to isolate Azure Monitor customers from sudden ingestion spikes in multitenancy environment. A default ingestion volume rate threshold of 500 MB (compressed) is defined in workspaces, this is translated to approximately **6 GB/min** uncompressed -- the actual size can vary between data types depending on the log length and its compression ratio. The volume rate limit applies to data ingested from Azure resources via [Diagnostic settings](#). When volume rate limit is reached, a retry mechanism attempts to ingest the data 4 times in a period of 30 minutes and drop it if operation fails. It doesn't apply to data ingested from [agents](#) or [Data Collector API](#).

When data sent to your workspace is at a volume rate higher than 80% of the threshold configured in your workspace, an event is sent to the *Operation* table in your workspace every 6 hours while the threshold continues to be exceeded. When ingested volume rate is higher than threshold, some data is dropped and an event is sent to the *Operation* table in your workspace every 6 hours while the threshold continues to be exceeded. If your ingestion volume rate continues to exceed the threshold or you are expecting to reach it sometime soon, you can request to increase it in by opening a support request.

See [Monitor health of Log Analytics workspace in Azure Monitor](#) to create alert rules to be proactively notified when you reach any ingestion limits.

**NOTE**

Depending on how long you've been using Log Analytics, you might have access to legacy pricing tiers. Learn more about [Log Analytics legacy pricing tiers](#).

## Application Insights

There are some limits on the number of metrics and events per application, that is, per instrumentation key. Limits depend on the [pricing plan](#) that you choose.

RESOURCE	DEFAULT LIMIT	NOTE
Total data per day	100 GB	You can reduce data by setting a cap. If you need more data, you can increase the limit in the portal, up to 1,000 GB. For capacities greater than 1,000 GB, send email to <a href="mailto:AIDataCap@microsoft.com">AIDataCap@microsoft.com</a> .
Throttling	32,000 events/second	The limit is measured over a minute.
Data retention Logs	<a href="#">30 - 730 days</a>	This resource is for <a href="#">Logs</a> .
Data retention Metrics	90 days	This resource is for <a href="#">Metrics Explorer</a> .
<a href="#">Availability multi-step test</a> detailed results retention	90 days	This resource provides detailed results of each step.
Maximum telemetry item size	64 kB	
Maximum telemetry items per batch	64 K	
Property and metric name length	150	See <a href="#">type schemas</a> .
Property value string length	8,192	See <a href="#">type schemas</a> .
Trace and exception message length	32,768	See <a href="#">type schemas</a> .

RESOURCE	DEFAULT LIMIT	NOTE
Availability tests count per app	100	
Profiler data retention	5 days	
Profiler data sent per day	10 GB	

For more information, see [About pricing and quotas in Application Insights](#).

## Azure NetApp Files

Azure NetApp Files has a regional limit for capacity. The standard capacity limit for each subscription is 25 TiB, per region, across all service levels. To increase the capacity, use the [Service and subscription limits \(quotas\)](#) support request.

To learn more about the limits for Azure NetApp Files, see [Resource limits for Azure NetApp Files](#).

## Azure Policy limits

There's a maximum count for each object type for Azure Policy. For definitions, an entry of *Scope* means the [management group](#) or subscription. For assignments and exemptions, an entry of *Scope* means the [management group](#), subscription, resource group, or individual resource.

WHERE	WHAT	MAXIMUM COUNT
Scope	Policy definitions	500
Scope	Initiative definitions	200
Tenant	Initiative definitions	2,500
Scope	Policy or initiative assignments	200
Scope	Exemptions	1000
Policy definition	Parameters	20
Initiative definition	Policies	1000
Initiative definition	Parameters	250
Policy or initiative assignments	Exclusions (notScopes)	400
Policy rule	Nested conditionals	512
Remediation task	Resources	500

## Azure Quantum limits

### Provider Limits & Quota

The Azure Quantum Service supports both first and third-party service providers. Third-party providers own their limits and quotas. Users can view offers and limits in the Azure portal when configuring third-party

providers in the provider blade.

You can find the published quota limits for Microsoft's first party Optimization Solutions provider below.

#### Learn & Develop SKU

RESOURCE	LIMIT
CPU-based concurrent jobs	up to 5 concurrent jobs
FPGA-based concurrent jobs	up to 2 concurrent jobs
CPU-based solver hours	20 hours per month
FPGA-based solver hours	1 hour per month

If you are using the Learn & Develop SKU, you **cannot** request an increase on your quota limits. Instead you should switch to the Performance at Scale SKU.

#### Performance at Scale SKU

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
CPU-based concurrent jobs	up to 100 concurrent jobs	same as default limit
FPGA-based concurrent jobs	up to 10 concurrent jobs	same as default limit
Solver hours	1,000 hours per month	up to 50,000 hours per month

If you need to request a limit increase, please reach out to Azure Support.

For more information, please review the [Azure Quantum pricing page](#). For information on third-party offerings, please review the relevant provider page in the Azure portal.

## Azure RBAC limits

The following limits apply to [Azure role-based access control \(Azure RBAC\)](#).

RESOURCE	LIMIT
<a href="#">Azure role assignments per Azure subscription</a>	2,000
<a href="#">Azure role assignments per management group</a>	500
<a href="#">Size of description for Azure role assignments</a>	2 KB
<a href="#">Size of condition for Azure role assignments</a>	8 KB
<a href="#">Azure custom roles per tenant</a>	5,000
<a href="#">Azure custom roles per tenant</a> (for Azure Germany and Azure China 21Vianet)	2,000

## Azure SignalR Service limits

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Azure SignalR Service units per instance for Free tier	1	1
Azure SignalR Service units per instance for Standard tier	100	100
Azure SignalR Service units per subscription per region for Free tier	5	5
Total Azure SignalR Service unit counts per subscription per region	150	Unlimited
Connections per unit per day for Free tier	20	20
Connections per unit per day for Standard tier	1,000	1,000
Included messages per unit per day for Free tier	20,000	20,000
Additional messages per unit per day for Free tier	0	0
Included messages per unit per day for Standard tier	1,000,000	1,000,000
Additional messages per unit per day for Standard tier	Unlimited	Unlimited

To request an update to your subscription's default limits, open a support ticket.

For more information about how connections and messages are counted, see [Messages and connections in Azure SignalR Service](#).

## Azure VMware Solution limits

The following table describes the maximum limits for Azure VMware Solution.

RESOURCE	LIMIT
Clusters per private cloud	12
Minimum number of hosts per cluster	3
Maximum number of hosts per cluster	16
hosts per private cloud	96
vCenter per private cloud	1
HCX site pairings	3 with Advanced edition, 10 with Enterprise edition

RESOURCE	LIMIT
AVS ExpressRoute max linked private clouds	4 The virtual network gateway used determines the actual max linked private clouds. For more details, see <a href="#">About ExpressRoute virtual network gateways</a>
AVS ExpressRoute portspeed	10 Gbps The virtual network gateway used determines the actual bandwidth. For more details, see <a href="#">About ExpressRoute virtual network gateways</a>
Public IPs exposed via vWAN	100
vSAN capacity limits	75% of total usable (keep 25% available for SLA)

For other VMware-specific limits, use the [VMware configuration maximum tool!](#).

## Backup limits

For a summary of Azure Backup support settings and limitations, see [Azure Backup Support Matrices](#).

## Batch limits

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Azure Batch accounts per region per subscription	1-3	50
Dedicated cores per Batch account	90-900	Contact support
Low-priority cores per Batch account	10-100	Contact support
<b>Active</b> jobs and job schedules per Batch account ( <b>completed</b> jobs have no limit)	100-300	1,000 <sup>1</sup>
Pools per Batch account	20-100	500 <sup>1</sup>

<sup>1</sup>To request an increase beyond this limit, contact Azure Support.

### NOTE

Default limits vary depending on the type of subscription you use to create a Batch account. Cores quotas shown are for Batch accounts in Batch service mode. [View the quotas in your Batch account](#).

### IMPORTANT

To help us better manage capacity during the global health pandemic, the default core quotas for new Batch accounts in some regions and for some types of subscription have been reduced from the above range of values, in some cases to zero cores. When you create a new Batch account, [check your core quota](#) and [request a core quota increase](#), if required. Alternatively, consider reusing Batch accounts that already have sufficient quota.

## Classic deployment model limits

If you use classic deployment model instead of the Azure Resource Manager deployment model, the following limits apply.

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
vCPUs per <a href="#">subscription</a> <sup>1</sup>	20	10,000
<a href="#">Co-administrators</a> per subscription	200	200
<a href="#">Storage accounts</a> per subscription <sup>2</sup>	100	100
<a href="#">Cloud services</a> per subscription	20	200
<a href="#">Local networks</a> per subscription	10	500
DNS servers per subscription	9	100
Reserved IPs per subscription	20	100
<a href="#">Affinity groups</a> per subscription	256	256
Subscription name length (characters)	64	64

<sup>1</sup>Extra small instances count as one vCPU toward the vCPU limit despite using a partial CPU core.

<sup>2</sup>The storage account limit includes both Standard and Premium storage accounts.

## Container Instances limits

RESOURCE	LIMIT
Standard sku container groups per region per subscription	100 <sup>1</sup>
Dedicated sku container groups per region per subscription	0 <sup>1</sup>
Number of containers per container group	60
Number of volumes per container group	20
Standard sku cores (CPUs) per region per subscription	10 <sup>1,2</sup>
Standard sku cores (CPUs) for K80 GPU per region per subscription	18 <sup>1,2</sup>
Standard sku cores (CPUs) for P100 or V100 GPU per region per subscription	0 <sup>1,2</sup>
Ports per IP	5
Container instance log size - running instance	4 MB

RESOURCE	LIMIT
Container instance log size - stopped instance	16 KB or 1,000 lines
Container group creates per hour	300 <sup>1</sup>
Container group creates per 5 minutes	100 <sup>1</sup>
Container group deletes per hour	300 <sup>1</sup>
Container group deletes per 5 minutes	100 <sup>1</sup>

<sup>1</sup>To request a limit increase, create an [Azure Support request](#). Free subscriptions including [Azure Free Account](#) and [Azure for Students](#) aren't eligible for limit or quota increases. If you have a free subscription, you can [upgrade](#) to a Pay-As-You-Go subscription.

<sup>2</sup>Default limit for [Pay-As-You-Go](#) subscription. Limit may differ for other category types.

## Container Registry limits

The following table details the features and limits of the Basic, Standard, and Premium [service tiers](#).

RESOURCE	BASIC	STANDARD	PREMIUM
Included storage <sup>1</sup> (GiB)	10	100	500
Storage limit (TiB)	20	20	20
Maximum image layer size (GiB)	200	200	200
Maximum manifest size (MiB)	4	4	4
ReadOps per minute <sup>2, 3</sup>	1,000	3,000	10,000
WriteOps per minute <sup>2, 4</sup>	100	500	2,000
Download bandwidth <sup>2</sup> (Mbps)	30	60	100
Upload bandwidth <sup>2</sup> (Mbps)	10	20	50
Webhooks	2	10	500
Geo-replication	N/A	N/A	<a href="#">Supported</a>
Availability zones	N/A	N/A	<a href="#">Preview</a>
Content trust	N/A	N/A	<a href="#">Supported</a>
Private link with private endpoints	N/A	N/A	<a href="#">Supported</a>

RESOURCE	BASIC	STANDARD	PREMIUM
• Private endpoints	N/A	N/A	10
Public IP network rules	N/A	N/A	100
Service endpoint VNet access	N/A	N/A	Preview
Customer-managed keys	N/A	N/A	Supported
Repository-scoped permissions	N/A	N/A	Preview
• Tokens	N/A	N/A	20,000
• Scope maps	N/A	N/A	20,000
• Repositories per scope map	N/A	N/A	500

<sup>1</sup> Storage included in the daily rate for each tier. Additional storage may be used, up to the registry storage limit, at an additional daily rate per GiB. For rate information, see [Azure Container Registry pricing](#). If you need storage beyond the registry storage limit, please contact Azure Support.

<sup>2</sup> *ReadOps*, *WriteOps*, and *Bandwidth* are minimum estimates. Azure Container Registry strives to improve performance as usage requires.

<sup>3</sup>A `docker pull` translates to multiple read operations based on the number of layers in the image, plus the manifest retrieval.

<sup>4</sup>A `docker push` translates to multiple write operations, based on the number of layers that must be pushed. A `docker push` includes *ReadOps* to retrieve a manifest for an existing image.

## Content Delivery Network limits

RESOURCE	LIMIT
Azure Content Delivery Network profiles	25
Content Delivery Network endpoints per profile	25
Custom domains per endpoint	25
Maximum origin group per profile	10
Maximum origin per origin group	10
Maximum number of rules per CDN endpoint	25
Maximum number of match conditions per rule	10
Maximum number of actions per rule	5

A Content Delivery Network subscription can contain one or more Content Delivery Network profiles. A Content Delivery Network profile can contain one or more Content Delivery Network endpoints. You might want to use multiple profiles to organize your Content Delivery Network endpoints by internet domain, web application, or some other criteria.

## Data Factory limits

Azure Data Factory is a multitenant service that has the following default limits in place to make sure customer subscriptions are protected from each other's workloads. To raise the limits up to the maximum for your subscription, contact support.

### Version 2

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Total number of entities, such as pipelines, data sets, triggers, linked services, Private Endpoints, and integration runtimes, within a data factory	5,000	Contact support.
Total CPU cores for Azure-SSIS Integration Runtimes under one subscription	256	Contact support.
Concurrent pipeline runs per data factory that's shared among all pipelines in the factory	10,000	10,000
Concurrent External activity runs per subscription per <a href="#">Azure Integration Runtime region</a>  External activities are managed on integration runtime but execute on linked services, including Databricks, stored procedure, Web, and others. This limit does not apply to Self-hosted IR.	3,000	3,000
Concurrent Pipeline activity runs per subscription per <a href="#">Azure Integration Runtime region</a>  Pipeline activities execute on integration runtime, including Lookup, GetMetadata, and Delete. This limit does not apply to Self-hosted IR.	1,000	1,000
Concurrent authoring operations per subscription per <a href="#">Azure Integration Runtime region</a>  Including test connection, browse folder list and table list, preview data. This limit does not apply to Self-hosted IR.	200	200
Concurrent Data Integration Units <sup>1</sup> consumption per subscription per <a href="#">Azure Integration Runtime region</a>	Region group 1 <sup>2</sup> : 6,000 Region group 2 <sup>2</sup> : 3,000 Region group 3 <sup>2</sup> : 1,500	Region group 1 <sup>2</sup> : 6,000 Region group 2 <sup>2</sup> : 3,000 Region group 3 <sup>2</sup> : 1,500

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Maximum activities per pipeline, which includes inner activities for containers	40	40
Maximum number of linked integration runtimes that can be created against a single self-hosted integration runtime	100	<a href="#">Contact support.</a>
Maximum parameters per pipeline	50	50
ForEach items	100,000	100,000
ForEach parallelism	20	50
Maximum queued runs per pipeline	100	100
Characters per expression	8,192	8,192
Minimum tumbling window trigger interval	15 min	15 min
Maximum timeout for pipeline activity runs	7 days	7 days
Bytes per object for pipeline objects <sup>3</sup>	200 KB	200 KB
Bytes per object for dataset and linked service objects <sup>3</sup>	100 KB	2,000 KB
Bytes per payload for each activity run <sup>4</sup>	896 KB	896 KB
Data Integration Units <sup>1</sup> per copy activity run	256	256
Write API calls	1,200/h	1,200/h  This limit is imposed by Azure Resource Manager, not Azure Data Factory.
Read API calls	12,500/h	12,500/h  This limit is imposed by Azure Resource Manager, not Azure Data Factory.
Monitoring queries per minute	1,000	1,000
Maximum time of data flow debug session	8 hrs	8 hrs
Concurrent number of data flows per integration runtime	50	<a href="#">Contact support.</a>

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Concurrent number of data flow debug sessions per user per factory	3	3
Data Flow Azure IR TTL limit	4 hrs	4 hrs
Meta Data Entity Size limit in a factory	2 GB	<a href="#">Contact support.</a>

<sup>1</sup> The data integration unit (DIU) is used in a cloud-to-cloud copy operation, learn more from [Data integration units \(version 2\)](#). For information on billing, see [Azure Data Factory pricing](#).

<sup>2</sup> [Azure Integration Runtime](#) is [globally available](#) to ensure data compliance, efficiency, and reduced network egress costs.

REGION GROUP	REGIONS
Region group 1	Central US, East US, East US 2, North Europe, West Europe, West US, West US 2
Region group 2	Australia East, Australia Southeast, Brazil South, Central India, Japan East, North Central US, South Central US, Southeast Asia, West Central US
Region group 3	Other regions

<sup>3</sup> Pipeline, data set, and linked service objects represent a logical grouping of your workload. Limits for these objects don't relate to the amount of data you can move and process with Azure Data Factory. Data Factory is designed to scale to handle petabytes of data.

<sup>4</sup> The payload for each activity run includes the activity configuration, the associated dataset(s) and linked service(s) configurations if any, and a small portion of system properties generated per activity type. Limit for this payload size doesn't relate to the amount of data you can move and process with Azure Data Factory. Learn about the [symptoms and recommendation](#) if you hit this limit.

## Version 1

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Pipelines within a data factory	2,500	<a href="#">Contact support.</a>
Data sets within a data factory	5,000	<a href="#">Contact support.</a>
Concurrent slices per data set	10	10
Bytes per object for pipeline objects <sup>1</sup>	200 KB	200 KB
Bytes per object for data set and linked service objects <sup>1</sup>	100 KB	2,000 KB
Azure HDInsight on-demand cluster cores within a subscription <sup>2</sup>	60	<a href="#">Contact support.</a>
Cloud data movement units per copy activity run <sup>3</sup>	32	32

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Retry count for pipeline activity runs	1,000	MaxInt (32 bit)

<sup>1</sup> Pipeline, data set, and linked service objects represent a logical grouping of your workload. Limits for these objects don't relate to the amount of data you can move and process with Azure Data Factory. Data Factory is designed to scale to handle petabytes of data.

<sup>2</sup> On-demand HDInsight cores are allocated out of the subscription that contains the data factory. As a result, the previous limit is the Data Factory-enforced core limit for on-demand HDInsight cores. It's different from the core limit that's associated with your Azure subscription.

<sup>3</sup> The cloud data movement unit (DMU) for version 1 is used in a cloud-to-cloud copy operation, learn more from [Cloud data movement units \(version 1\)](#). For information on billing, see [Azure Data Factory pricing](#).

RESOURCE	DEFAULT LOWER LIMIT	MINIMUM LIMIT
Scheduling interval	15 minutes	15 minutes
Interval between retry attempts	1 second	1 second
Retry timeout value	1 second	1 second

#### Web service call limits

Azure Resource Manager has limits for API calls. You can make API calls at a rate within the [Azure Resource Manager API limits](#).

## Data Lake Analytics limits

Azure Data Lake Analytics makes the complex task of managing distributed infrastructure and complex code easy. It dynamically provisions resources, and you can use it to do analytics on exabytes of data. When the job completes, it winds down resources automatically. You pay only for the processing power that was used. As you increase or decrease the size of data stored or the amount of compute used, you don't have to rewrite code. To raise the default limits for your subscription, contact support.

RESOURCE	LIMIT	COMMENTS
Maximum number of concurrent jobs	20	
Maximum number of analytics units (AUs) per account	250	Use any combination of up to a maximum of 250 AUs across 20 jobs. To increase this limit, contact Microsoft Support.
Maximum script size for job submission	3 MB	
Maximum number of Data Lake Analytics accounts per region per subscription	5	To increase this limit, contact Microsoft Support.

## Data Lake Storage limits

Azure Data Lake Storage Gen2 is not a dedicated service or storage account type. It is the latest release of

capabilities that are dedicated to big data analytics. These capabilities are available in a general-purpose v2 or BlockBlobStorage storage account, and you can obtain them by enabling the **Hierarchical namespace** feature of the account. For scale targets, see these articles.

- [Scale targets for Blob storage](#).
- [Scale targets for standard storage accounts](#).

**Azure Data Lake Storage Gen1** is a dedicated service. It's an enterprise-wide hyper-scale repository for big data analytic workloads. You can use Data Lake Storage Gen1 to capture data of any size, type, and ingestion speed in one single place for operational and exploratory analytics. There's no limit to the amount of data you can store in a Data Lake Storage Gen1 account.

RESOURCE	LIMIT	COMMENTS
Maximum number of Data Lake Storage Gen1 accounts, per subscription, per region	10	To request an increase for this limit, contact support.
Maximum number of access ACLs, per file or folder	32	This is a hard limit. Use groups to manage access with fewer entries.
Maximum number of default ACLs, per file or folder	32	This is a hard limit. Use groups to manage access with fewer entries.

## Data Share limits

Azure Data Share enables organizations to simply and securely share data with their customers and partners.

RESOURCE	LIMIT
Maximum number of Data Share resources per Azure subscription	100
Maximum number of sent shares per Data Share resource	200
Maximum number of received shares per Data Share resource	100
Maximum number of invitations per sent share	200
Maximum number of share subscriptions per sent share	200
Maximum number of datasets per share	200
Maximum number of snapshot schedules per share	1

## Database Migration Service Limits

Azure Database Migration Service is a fully managed service designed to enable seamless migrations from multiple database sources to Azure data platforms with minimal downtime.

RESOURCE	LIMIT	COMMENTS
Maximum number of services per subscription, per region	10	To request an increase for this limit, contact support.

## Device Update for IoT Hub limits

### NOTE

When a given resource or operation doesn't have adjustable limits, the default and the maximum limits are the same. When the limit can be adjusted, the table includes different values for Default limit and Maximum limit headers. The limit can be raised above the default limit but not above the maximum limit. If you want to raise the limit or quota above the default limit, open an [online customer support request](#).

This table provides the limits for the Device Update for IoT Hub resource in Azure Resource Manager:

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT	ADJUSTABLE?
Accounts per subscription	2	25	Yes
Instances per account	2	25	Yes
Length of account name	Minimum: 3 Maximum: 24	Minimum: 3 Maximum: 24	No
Length of instance name	Minimum: 3 Maximum: 36	Minimum: 3 Maximum: 36	No

This table provides the various limits associated with the operations within Device Update for IoT Hub:

OPERATION	DEFAULT LIMIT	MAXIMUM LIMIT	ADJUSTABLE?
Number of devices per instance	10,000	10,000	No
Number of update providers per instance	25	25	No
Number of update names per provider per instance	25	25	No
Number of update versions per update provider and name per instance	100	100	No
Total number of updates per instance	100	100	No
Maximum single update file size	800 MB	800 MB	No
Maximum combined size of all files in a single import action	800 MB	800 MB	No

OPERATION	DEFAULT LIMIT	MAXIMUM LIMIT	ADJUSTABLE?
Number of device groups per instance	75	75	No

## Digital Twins limits

### NOTE

Some areas of this service have adjustable limits, and others do not. This is represented in the tables below with the *Adjustable?* column. When the limit can be adjusted, the *Adjustable?* value is *Yes*.

### Functional limits

The following table lists the functional limits of Azure Digital Twins.

### TIP

For modeling recommendations to operate within these functional limits, see [Modeling best practices](#).

AREA	CAPABILITY	DEFAULT LIMIT	ADJUSTABLE?
Azure resource	Number of Azure Digital Twins instances in a region, per subscription	10	Yes
Digital twins	Number of twins in an Azure Digital Twins instance	200,000	Yes
Digital twins	Number of incoming relationships to a single twin	5,000	No
Digital twins	Number of outgoing relationships from a single twin	5,000	No
Digital twins	Maximum size (of JSON body in a PUT or PATCH request) of a single twin	32 KB	No
Digital twins	Maximum request payload size	32 KB	No
Routing	Number of endpoints for a single Azure Digital Twins instance	6	No
Routing	Number of routes for a single Azure Digital Twins instance	6	Yes
Models	Number of models within a single Azure Digital Twins instance	10,000	Yes

Area	Capability	Default Limit	Adjustable?
------	------------	---------------	-------------

Models	Number of models that can be uploaded in a single API call	250	No
Models	Maximum size (of JSON body in a PUT or PATCH request) of a single model	1 MB	No
Models	Number of items returned in a single page	100	No
Query	Number of items returned in a single page	100	Yes
Query	Number of <code>AND</code> / <code>OR</code> expressions in a query	50	Yes
Query	Number of array items in an <code>IN</code> / <code>NOT IN</code> clause	50	Yes
Query	Number of characters in a query	8,000	Yes
Query	Number of <code>JOINS</code> in a query	5	Yes

## Rate limits

The following table reflects the rate limits of different APIs.

API	Capability	Default Limit	Adjustable?
Models API	Number of requests per second	100	Yes
Digital Twins API	Number of read requests per second	1,000	Yes
Digital Twins API	Number of patch requests per second	1,000	Yes
Digital Twins API	Number of create/delete operations per second across all twins and relationships	50	Yes
Digital Twins API	Number of create/update/delete operations per second on a single twin or its relationships	10	No

API	Capability	Default Limit	Adjustable?
Query API	Number of requests per second	500	Yes
Query API	Query Units per second	4,000	Yes
Event Routes API	Number of requests per second	100	Yes

## Other limits

Limits on data types and fields within DTDL documents for Azure Digital Twins models can be found within its spec documentation in GitHub: [Digital Twins Definition Language \(DTDL\) - version 2](#).

Query latency details are described in [Query language](#). Limitations of particular query language features can be found in the [query reference documentation](#).

## Event Grid limits

The following limits apply to Azure Event Grid **topics** (system, custom, and partner topics).

Resource	Limit
Custom topics per Azure subscription	100
Event subscriptions per topic	500
Publish rate for a custom or a partner topic (ingress)	5,000 events/sec or 5 MB/sec (whichever is met first)
Event size	1 MB
Private endpoint connections per topic	64
IP Firewall rules per topic	16

The following limits apply to Azure Event Grid **domains**.

Resource	Limit
Topics per event domain	100,000
Event subscriptions per topic within a domain	500
Domain scope event subscriptions	50
Publish rate for an event domain (ingress)	5,000 events/sec or 5 MB/sec (whichever is met first)
Event Domains per Azure Subscription	100
Private endpoint connections per domain	64
IP Firewall rules per domain	16

# Event Hubs limits

The following tables provide quotas and limits specific to [Azure Event Hubs](#). For information about Event Hubs pricing, see [Event Hubs pricing](#).

## Common limits for all tiers

The following limits are common across all tiers.

LIMIT	NOTES	VALUE
Size of an event hub name	-	256 characters
Size of a consumer group name	Kafka protocol doesn't require the creation of a consumer group.	Kafka: 256 characters AMQP: 50 characters
Number of non-epoch receivers per consumer group	-	5
Number of authorization rules per namespace	Subsequent requests for authorization rule creation are rejected.	12
Number of calls to the GetRuntimeInformation method	-	50 per second
Number of virtual networks (VNet)	-	128
Number of IP Config rules	-	128
Maximum length of a schema group name		50
Maximum length of a schema name		100
Size in bytes per schema		1 MB
Number of properties per schema group		1024
Size in bytes per schema group property key		256
Size in bytes per schema group property value		1024

## Basic vs. standard vs. premium vs. dedicated tiers

The following table shows limits that may be different for basic, standard, and dedicated tiers. In the table CU is capacity unit, PU is processing unit, TU is throughput unit.

LIMIT	BASIC	STANDARD	PREMIUM	DEDICATED
Maximum size of Event Hubs publication	256 KB	1 MB	1 MB	1 MB

LIMIT	BASIC	STANDARD	PREMIUM	DEDICATED
Number of consumer groups per event hub	1	20	100	1000 No limit per CU
Number of brokered connections per namespace	100	5,000	10000 per processing unit per PU	100,000 per CU
Maximum retention period of event data	1 day	7 days	90 days 1 TB per PU	90 days 10 TB per CU
Maximum TUs or PUs or CUs	20 TUs	40 TUs	16 PUs	20 CUs
Number of partitions per event hub	32	32	100 (fixed)	1024 per event hub 2000 per CU
Number of namespaces per subscription	1000	1000	1000	1000 (50 per CU)
Number of event hubs per namespace	10	10	100 per PU	1000
Capture	N/A	Pay per hour	Included	Included
Size of the schema registry (namespace) in mega bytes	N/A	25	100	1024
Number of schema groups in a schema registry or namespace	N/A	1 - excluding the default group	100 1 MB per schema	1000 1 MB per schema
Number of schema versions across all schema groups	N/A	25	1000	10000
Throughput per unit	Ingress - 1 MB/s or 1000 events per second Egress – 2 Mb/s or 4096 events per second	Ingress - 1 MB/s or 1000 events per second Egress – 2 Mb/s or 4096 events per second	No limits per PU *	No limits per CU *

\* Depends on various factors such as resource allocation, number of partitions, storage, and so on.

#### NOTE

You can publish events individually or batched. The publication limit (according to SKU) applies regardless of whether it is a single event or a batch. Publishing events larger than the maximum threshold will be rejected.

## IoT Central limits

IoT Central limits the number of applications you can deploy in a subscription to 10. If you need to increase this limit, contact [Microsoft support](#).

## IoT Hub limits

The following table lists the limits associated with the different service tiers S1, S2, S3, and F1. For information about the cost of each *unit* in each tier, see [Azure IoT Hub pricing](#).

RESOURCE	S1 STANDARD	S2 STANDARD	S3 STANDARD	F1 FREE
Messages/day	400,000	6,000,000	300,000,000	8,000
Maximum units	200	200	10	1

### NOTE

If you anticipate using more than 200 units with an S1 or S2 tier hub or 10 units with an S3 tier hub, contact Microsoft Support.

The following table lists the limits that apply to IoT Hub resources.

RESOURCE	LIMIT
Maximum paid IoT hubs per Azure subscription	50
Maximum free IoT hubs per Azure subscription	1
Maximum number of characters in a device ID	128
Maximum number of device identities returned in a single call	1,000
IoT Hub message maximum retention for device-to-cloud messages	7 days
Maximum size of device-to-cloud message	256 KB
Maximum size of device-to-cloud batch	AMQP and HTTP: 256 KB for the entire batch MQTT: 256 KB for each message
Maximum messages in device-to-cloud batch	500
Maximum size of cloud-to-device message	64 KB
Maximum TTL for cloud-to-device messages	2 days
Maximum delivery count for cloud-to-device messages	100
Maximum cloud-to-device queue depth per device	50
Maximum delivery count for feedback messages in response to a cloud-to-device message	100

RESOURCE	LIMIT
Maximum TTL for feedback messages in response to a cloud-to-device message	2 days
<a href="#">Maximum size of device twin</a>	8 KB for tags section, and 32 KB for desired and reported properties sections each
Maximum length of device twin string key	1 KB
Maximum length of device twin string value	4 KB
<a href="#">Maximum depth of object in device twin</a>	10
Maximum size of direct method payload	128 KB
Job history maximum retention	30 days
Maximum concurrent jobs	10 (for S3), 5 for (S2), 1 (for S1)
Maximum additional endpoints	10 (for S1, S2, and S3)
Maximum message routing rules	100 (for S1, S2, and S3)
Maximum number of concurrently connected device streams	50 (for S1, S2, S3, and F1 only)
Maximum device stream data transfer	300 MB per day (for S1, S2, S3, and F1 only)

**NOTE**

If you need more than 50 paid IoT hubs in an Azure subscription, contact Microsoft Support.

**NOTE**

Currently, the total number of devices plus modules that can be registered to a single IoT hub is capped at 1,000,000. If you want to increase this limit, contact [Microsoft Support](#).

IoT Hub throttles requests when the following quotas are exceeded.

THROTTLE	PER-HUB VALUE
Identity registry operations (create, retrieve, list, update, and delete), individual or bulk import/export	83.33/sec/unit (5,000/min/unit) (for S3). 1.67/sec/unit (100/min/unit) (for S1 and S2).
Device connections	6,000/sec/unit (for S3), 120/sec/unit (for S2), 12/sec/unit (for S1). Minimum of 100/sec.
Device-to-cloud sends	6,000/sec/unit (for S3), 120/sec/unit (for S2), 12/sec/unit (for S1). Minimum of 100/sec.

THROTTLE	PER-HUB VALUE
Cloud-to-device sends	83.33/sec/unit (5,000/min/unit) (for S3), 1.67/sec/unit (100/min/unit) (for S1 and S2).
Cloud-to-device receives	833.33/sec/unit (50,000/min/unit) (for S3), 16.67/sec/unit (1,000/min/unit) (for S1 and S2).
File upload operations	83.33 file upload initiations/sec/unit (5,000/min/unit) (for S3), 1.67 file upload initiations/sec/unit (100/min/unit) (for S1 and S2). 10,000 SAS URIs can be out for an Azure Storage account at one time. 10 SAS URIs/device can be out at one time.
Direct methods	24 MB/sec/unit (for S3), 480 KB/sec/unit (for S2), 160 KB/sec/unit (for S1). Based on 8-KB throttling meter size.
Device twin reads	500/sec/unit (for S3), Maximum of 100/sec or 10/sec/unit (for S2), 100/sec (for S1)
Device twin updates	250/sec/unit (for S3), Maximum of 50/sec or 5/sec/unit (for S2), 50/sec (for S1)
Jobs operations (create, update, list, and delete)	83.33/sec/unit (5,000/min/unit) (for S3), 1.67/sec/unit (100/min/unit) (for S2), 1.67/sec/unit (100/min/unit) (for S1).
Jobs per-device operation throughput	50/sec/unit (for S3), maximum of 10/sec or 1/sec/unit (for S2), 10/sec (for S1).
Device stream initiation rate	5 new streams/sec (for S1, S2, S3, and F1 only).

## IoT Hub Device Provisioning Service limits

### NOTE

Some areas of this service have adjustable limits. This is represented in the tables below with the *Adjustable?* column. When the limit can be adjusted, the *Adjustable?* value is *Yes*.

If your business requires raising an adjustable limit or quota above the default limit, you can request additional resources by [opening a support ticket](#).

The following table lists the limits that apply to Azure IoT Hub Device Provisioning Service resources.

RESOURCE	LIMIT	ADJUSTABLE?
Maximum device provisioning services per Azure subscription	10	Yes
Maximum number of registrations	1,000,000	Yes
Maximum number of individual enrollments	1,000,000	Yes

RESOURCE	LIMIT	ADJUSTABLE?
Maximum number of enrollment groups ( <i>X.509 certificate</i> )	100	Yes
Maximum number of enrollment groups ( <i>symmetric key</i> )	100	No
Maximum number of CAs	25	No
Maximum number of linked IoT hubs	50	No
Maximum size of message	96 KB	No

#### TIP

If the hard limit on symmetric key enrollment groups is a blocking issue, it is recommended to use individual enrollments as a workaround.

The Device Provisioning Service has the following rate limits.

RATE	PER-UNIT VALUE	ADJUSTABLE?
Operations	200/min/service	Yes
Device registrations	200/min/service	Yes
Device polling operation	5/10 sec/device	No

Each API call on DPS is billable as one *Operation*. This includes all the service APIs and the device registration API. The device registration polling operation is not billed.

## Key Vault limits

Azure Key Vault service supports two resource types: Vaults and Managed HSMs. The following two sections describe the service limits for each of them respectively.

### Resource type: vault

This section describes service limits for resource type `vaults`.

#### Key transactions (maximum transactions allowed in 10 seconds, per vault per region<sup>1</sup>):

KEY TYPE	HSM KEY CREATE KEY	HSM KEY ALL OTHER TRANSACTIONS	SOFTWARE KEY CREATE KEY	SOFTWARE KEY ALL OTHER TRANSACTIONS
RSA 2,048-bit	5	1,000	10	2,000
RSA 3,072-bit	5	250	10	500
RSA 4,096-bit	5	125	10	250
ECC P-256	5	1,000	10	2,000

KEY TYPE	HSM KEY CREATE KEY	HSM KEY ALL OTHER TRANSACTIONS	SOFTWARE KEY CREATE KEY	SOFTWARE KEY ALL OTHER TRANSACTIONS
ECC P-384	5	1,000	10	2,000
ECC P-521	5	1,000	10	2,000
ECC SECP256K1	5	1,000	10	2,000

#### NOTE

In the previous table, we see that for RSA 2,048-bit software keys, 2,000 GET transactions per 10 seconds are allowed. For RSA 2,048-bit HSM-keys, 1,000 GET transactions per 10 seconds are allowed.

The throttling thresholds are weighted, and enforcement is on their sum. For example, as shown in the previous table, when you perform GET operations on RSA HSM-keys, it's eight times more expensive to use 4,096-bit keys compared to 2,048-bit keys. That's because  $1,000/125 = 8$ .

In a given 10-second interval, an Azure Key Vault client can do *only one* of the following operations before it encounters a 429 throttling HTTP status code:

- 2,000 RSA 2,048-bit software-key GET transactions
- 1,000 RSA 2,048-bit HSM-key GET transactions
- 125 RSA 4,096-bit HSM-key GET transactions
- 124 RSA 4,096-bit HSM-key GET transactions and 8 RSA 2,048-bit HSM-key GET transactions

#### Secrets, managed storage account keys, and vault transactions:

TRANSACTIONS TYPE	MAXIMUM TRANSACTIONS ALLOWED IN 10 SECONDS, PER VAULT PER REGION <sup>1</sup>
All transactions	2,000

For information on how to handle throttling when these limits are exceeded, see [Azure Key Vault throttling guidance](#).

<sup>1</sup> A subscription-wide limit for all transaction types is five times per key vault limit. For example, HSM-other transactions per subscription are limited to 5,000 transactions in 10 seconds per subscription.

#### Backup keys, secrets, certificates

When you back up a key vault object, such as a secret, key, or certificate, the backup operation will download the object as an encrypted blob. This blob can't be decrypted outside of Azure. To get usable data from this blob, you must restore the blob into a key vault within the same Azure subscription and Azure geography.

TRANSACTIONS TYPE	MAXIMUM KEY VAULT OBJECT VERSIONS ALLOWED
Backup individual key, secret, certificate	500

#### NOTE

Attempting to backup a key, secret, or certificate object with more versions than above limit will result in an error. It is not possible to delete previous versions of a key, secret, or certificate.

#### Limits on count of keys, secrets and certificates:

Key Vault does not restrict the number of keys, secrets or certificates that can be stored in a vault. The transaction limits on the vault should be taken into account to ensure that operations are not throttled.

Key Vault does not restrict the number of versions on a secret, key or certificate, but storing a large number of versions (500+) can impact the performance of backup operations. See [Azure Key Vault Backup](#).

#### Azure Private Link integration

##### NOTE

The number of key vaults with private endpoints enabled per subscription is an adjustable limit. The limit shown below is the default limit. If you would like to request a limit increase for your service, please create a support request and it will be assessed on a case by case basis.

RESOURCE	LIMIT
Private endpoints per key vault	64
Key vaults with private endpoints per subscription	400

#### Resource type: Managed HSM

This section describes service limits for resource type `managed HSM`.

##### Object limits

ITEM	LIMITS
Number of HSM instances per subscription per region	1
Number of keys per HSM Pool	5000
Number of versions per key	100
Number of custom role definitions per HSM	50
Number of role assignments at HSM scope	50
Number of role assignment at each individual key scope	10

##### Transaction limits for administrative operations (number of operations per second per HSM instance)

OPERATION	NUMBER OF OPERATIONS PER SECOND
All RBAC operations (includes all CRUD operations for role definitions and role assignments)	5
Full HSM Backup/Restore (only one concurrent backup or restore operation per HSM instance supported)	1

##### Transaction limits for cryptographic operations (number of operations per second per HSM instance)

- Each Managed HSM instance constitutes 3 load balanced HSM partitions. The throughput limits are a function of underlying hardware capacity allocated for each partition. The tables below show maximum throughput with at least one partition available. Actual throughput may be up to 3x higher if all 3 partitions

are available.

- Throughput limits noted assume that one single key is being used to achieve maximum throughput. For example, if a single RSA-2048 key is used the maximum throughput will be 1100 sign operations. If you use 1100 different keys with 1 transaction per second each, they will not be able to achieve the same throughput.

**RSA key operations (number of operations per second per HSM instance)**

OPERATION	2048-BIT	3072-BIT	4096-BIT
Create Key	1	1	1
Delete Key (soft-delete)	10	10	10
Purge Key	10	10	10
Backup Key	10	10	10
Restore Key	10	10	10
Get Key Information	1100	1100	1100
Encrypt	10000	10000	6000
Decrypt	1100	360	160
Wrap	10000	10000	6000
Unwrap	1100	360	160
Sign	1100	360	160
Verify	10000	10000	6000

**EC key operations (number of operations per second per HSM instance)**

This table describes number of operations per second for each curve type.

OPERATION	P-256	P-256K	P-384	P-521
Create Key	1	1	1	1
Delete Key (soft-delete)	10	10	10	10
Purge Key	10	10	10	10
Backup Key	10	10	10	10
Restore Key	10	10	10	10
Get Key Information	1100	1100	1100	1100
Sign	260	260	165	56

OPERATION	P-256	P-256K	P-384	P-521
Verify	130	130	82	28

AES key operations (number of operations per second per HSM instance)

- Encrypt and Decrypt operations assume a 4KB packet size.
- Throughput limits for Encrypt/Decrypt apply to AES-CBC and AES-GCM algorithms.
- Throughput limits for Wrap/Unwrap apply to AES-KW algorithm.

OPERATION	128-BIT	192-BIT	256-BIT
Create Key	1	1	1
Delete Key (soft-delete)	10	10	10
Purge Key	10	10	10
Backup Key	10	10	10
Restore Key	10	10	10
Get Key Information	1100	1100	1100
Encrypt	8000	8000	8000
Decrypt	8000	8000	8000
Wrap	9000	9000	9000
Unwrap	9000	9000	9000

## Managed identity limits

- Each managed identity counts towards the object quota limit in an Azure AD tenant as described in [Azure AD service limits and restrictions](#).
- The rate at which managed identities can be created have the following limits:
  - Per Azure AD Tenant per Azure region: 400 create operations per 20 seconds.
  - Per Azure Subscription per Azure region : 80 create operations per 20 seconds.
- The rate at which a user-assigned managed identity can be assigned with an Azure resource :
  - Per Azure AD Tenant per Azure region: 400 assignment operations per 20 seconds.
  - Per Azure Subscription per Azure region : 300 assignment operations per 20 seconds.

## Media Services limits

### NOTE

For resources that aren't fixed, open a support ticket to ask for an increase in the quotas. Don't create additional Azure Media Services accounts in an attempt to obtain higher limits.

## Account limits

RESOURCE	DEFAULT LIMIT
Media Services accounts in a single subscription	100 (fixed)

## Asset limits

RESOURCE	DEFAULT LIMIT
Assets per Media Services account	1,000,000

## Storage (media) limits

RESOURCE	DEFAULT LIMIT
File size	In some scenarios, there is a limit on the maximum file size supported for processing in Media Services. <sup>(1)</sup>
Storage accounts	100 <sup>(2)</sup> (fixed)

<sup>1</sup> The maximum size supported for a single blob is currently up to 5 TB in Azure Blob Storage. Additional limits apply in Media Services based on the VM sizes that are used by the service. The size limit applies to the files that you upload and also the files that get generated as a result of Media Services processing (encoding or analyzing). If your source file is larger than 260-GB, your Job will likely fail.

The following table shows the limits on the media reserved units S1, S2, and S3. If your source file is larger than the limits defined in the table, your encoding job fails. If you encode 4K resolution sources of long duration, you're required to use S3 media reserved units to achieve the performance needed. If you have 4K content that's larger than the 260-GB limit on the S3 media reserved units, open a support ticket.

MEDIA RESERVED UNIT TYPE	MAXIMUM INPUT SIZE (GB)
S1	26
S2	60
S3	260

<sup>2</sup> The storage accounts must be from the same Azure subscription.

## Jobs (encoding & analyzing) limits

RESOURCE	DEFAULT LIMIT
Jobs per Media Services account	500,000 <sup>(3)</sup> (fixed)
Job inputs per Job	50 (fixed)
Job outputs per Job	20 (fixed)
Transforms per Media Services account	100 (fixed)
Transform outputs in a Transform	20 (fixed)

RESOURCE	DEFAULT LIMIT
Files per job input	10 (fixed)

<sup>3</sup> This number includes queued, finished, active, and canceled Jobs. It does not include deleted Jobs.

Any Job record in your account older than 90 days will be automatically deleted, even if the total number of records is below the maximum quota.

### Live streaming limits

RESOURCE	DEFAULT LIMIT
Live Events <sup>(4)</sup> per Media Services account	5
Live Outputs per Live Event	3 <sup>(5)</sup>
Max Live Output duration	Size of the DVR window

<sup>4</sup> For detailed information about Live Event limitations, see [Live Event types comparison and limitations](#).

<sup>5</sup> Live Outputs start on creation and stop when deleted.

### Packaging & delivery limits

RESOURCE	DEFAULT LIMIT
Streaming Endpoints (stopped or running) per Media Services account	2
Dynamic Manifest Filters	100
Streaming Policies	100 <sup>(6)</sup>
Unique Streaming Locators associated with an Asset at one time	100 <sup>(7)</sup> (fixed)

<sup>6</sup> When using a custom [Streaming Policy](#), you should design a limited set of such policies for your Media Service account, and re-use them for your StreamingLocators whenever the same encryption options and protocols are needed. You should not be creating a new Streaming Policy for each Streaming Locator.

<sup>7</sup> Streaming Locators are not designed for managing per-user access control. To give different access rights to individual users, use Digital Rights Management (DRM) solutions.

### Protection limits

RESOURCE	DEFAULT LIMIT
Options per Content Key Policy	30
Licenses per month for each of the DRM types on Media Services key delivery service per account	1,000,000

### Support ticket

For resources that are not fixed, you may ask for the quotas to be raised, by opening a [support ticket](#). Include detailed information in the request on the desired quota changes, use-case scenarios, and regions required.

Do not create additional Azure Media Services accounts in an attempt to obtain higher limits.

### Media Services v2 (legacy)

For limits specific to Media Services v2 (legacy), see [Media Services v2 \(legacy\)](#)

## Mobile Services limits

TIER	FREE	BASIC	STANDARD
API calls	500,000	1.5 million per unit	15 million per unit
Active devices	500	Unlimited	Unlimited
Scale	N/A	Up to 6 units	Unlimited units
Push notifications	Azure Notification Hubs Free tier included, up to 1 million pushes	Notification Hubs Basic tier included, up to 10 million pushes	Notification Hubs Standard tier included, up to 10 million pushes
Real-time messaging/ Web Sockets	Limited	350 per mobile service	Unlimited
Offline synchronizations	Limited	Included	Included
Scheduled jobs	Limited	Included	Included
Azure SQL Database (required) Standard rates apply for additional capacity	20 MB included	20 MB included	20 MB included
CPU capacity	60 minutes per day	Unlimited	Unlimited
Outbound data transfer	165 MB per day (daily rollover)	Included	Included

For more information on limits and pricing, see [Azure Mobile Services pricing](#).

## Multi-Factor Authentication limits

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Maximum number of trusted IP addresses or ranges per subscription	0	50
Remember my devices, number of days	14	60
Maximum number of app passwords	0	No limit
Allow X attempts during MFA call	1	99
Two-way text message timeout seconds	60	600

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Default one-time bypass seconds	300	1,800
Lock user account after X consecutive MFA denials	Not set	99
Reset account lockout counter after X minutes	Not set	9,999
Unlock account after X minutes	Not set	9,999

## Networking limits

### Networking limits - Azure Resource Manager

The following limits apply only for networking resources managed through **Azure Resource Manager** per region per subscription. Learn how to [view your current resource usage against your subscription limits](#).

#### NOTE

We recently increased all default limits to their maximum limits. If there's no maximum limit column, the resource doesn't have adjustable limits. If you had these limits increased by support in the past and don't see updated limits in the following tables, [open an online customer support request at no charge](#)

RESOURCE	LIMIT
Virtual networks	1,000
Subnets per virtual network	3,000
Virtual network peerings per virtual network	500
Virtual network gateways (VPN gateways) per virtual network	1
Virtual network gateways (ExpressRoute gateways) per virtual network	1
DNS servers per virtual network	20
Private IP addresses per virtual network	65,536
Private IP addresses per network interface	256
Private IP addresses per virtual machine	256
Public IP addresses per network interface	256
Public IP addresses per virtual machine	256
Concurrent TCP or UDP flows per NIC of a virtual machine or role instance	500,000

RESOURCE	LIMIT
Network interface cards	65,536
Network Security Groups	5,000
NSG rules per NSG	1,000
IP addresses and ranges specified for source or destination in a security group	4,000
Application security groups	3,000
Application security groups per IP configuration, per NIC	20
IP configurations per application security group	4,000
Application security groups that can be specified within all security rules of a network security group	100
User-defined route tables	200
User-defined routes per route table	400
Point-to-site root certificates per Azure VPN Gateway	20
Point-to-site revoked client certificates per Azure VPN Gateway	300
Virtual network TAPs	100
Network interface TAP configurations per virtual network TAP	100

#### Public IP address limits

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Public IP addresses <sup>1,2</sup>	10 for Basic.	Contact support.
Static Public IP addresses <sup>1</sup>	10 for Basic.	Contact support.
Standard Public IP addresses <sup>1</sup>	10	Contact support.
Public IP addresses per Resource Group	800	Contact support.
Public IP Prefixes	limited by number of Standard Public IPs in a subscription	Contact support.
Public IP prefix length	/28	Contact support.

<sup>1</sup>Default limits for Public IP addresses vary by offer category type, such as Free Trial, Pay-As-You-Go, CSP. For example, the default for Enterprise Agreement subscriptions is 1000.

<sup>2</sup>Public IP addresses limit refers to the total amount of Public IP addresses, including Basic and Standard.

#### Load balancer limits

The following limits apply only for networking resources managed through Azure Resource Manager per region per subscription. Learn how to [view your current resource usage against your subscription limits](#).

#### Standard Load Balancer

RESOURCE	LIMIT
Load balancers	1,000
Rules (Load Balancer + Inbound NAT) per resource	1,500
Rules per NIC (across all IPs on a NIC)	300
Frontend IP configurations	600
Backend pool size	1,000 IP configurations, single virtual network
Backend resources per Load Balancer <sup>1</sup>	1,200
High-availability ports rule	1 per internal frontend
Outbound rules per Load Balancer	600
Load Balancers per VM	2 (1 Public and 1 internal)

<sup>1</sup> The limit is up to 1,200 resources, in any combination of standalone virtual machine resources, availability set resources, and virtual machine scale-set placement groups.

#### Basic Load Balancer

RESOURCE	LIMIT
Load balancers	1,000
Rules per resource	250
Rules per NIC (across all IPs on a NIC)	300
Frontend IP configurations <sup>2</sup>	200
Backend pool size	300 IP configurations, single availability set
Availability sets per Load Balancer	1
Load Balancers per VM	2 (1 Public and 1 internal)

<sup>2</sup> The limit for a single discrete resource in a backend pool (standalone virtual machine, availability set, or virtual machine scale-set placement group) is to have up to 250 Frontend IP configurations across a single Basic Public Load Balancer and Basic Internal Load Balancer.

The following limits apply only for networking resources managed through the **classic** deployment model per subscription. Learn how to [view your current resource usage against your subscription limits](#).

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Virtual networks	100	100
Local network sites	20	50
DNS servers per virtual network	20	20
Private IP addresses per virtual network	4,096	4,096
Concurrent TCP or UDP flows per NIC of a virtual machine or role instance	500,000, up to 1,000,000 for two or more NICs.	500,000, up to 1,000,000 for two or more NICs.
Network Security Groups (NSGs)	200	200
NSG rules per NSG	200	1,000
User-defined route tables	200	200
User-defined routes per route table	400	400
Public IP addresses (dynamic)	500	500
Reserved public IP addresses	500	500
Public IP per deployment	5	Contact support
Private IP (internal load balancing) per deployment	1	1
Endpoint access control lists (ACLs)	50	50

## ExpressRoute limits

RESOURCE	LIMIT
ExpressRoute circuits per subscription	50
ExpressRoute circuits per region per subscription, with Azure Resource Manager	10
Maximum number of IPv4 routes advertised to Azure private peering with ExpressRoute Standard	4,000
Maximum number of IPv4 routes advertised to Azure private peering with ExpressRoute Premium add-on	10,000
Maximum number of IPv6 routes advertised to Azure private peering with ExpressRoute Standard	100
Maximum number of IPv6 routes advertised to Azure private peering with ExpressRoute Premium add-on	100

RESOURCE	LIMIT
Maximum number of routes advertised from Azure private peering from the VNet address space for an ExpressRoute connection	1,000
Maximum number of routes advertised to Microsoft peering with ExpressRoute Standard	200
Maximum number of routes advertised to Microsoft peering with ExpressRoute Premium add-on	200
Maximum number of ExpressRoute circuits linked to the same virtual network in the same peering location	4
Maximum number of ExpressRoute circuits linked to the same virtual network in different peering locations	16 (For more information, see <a href="#">Gateway SKU</a> .)
Number of virtual network links allowed per ExpressRoute circuit	See the <a href="#">Number of virtual networks per ExpressRoute circuit</a> table.

#### Number of virtual networks per ExpressRoute circuit

CIRCUIT SIZE	NUMBER OF VIRTUAL NETWORK LINKS FOR STANDARD	NUMBER OF VIRTUAL NETWORK LINKS WITH PREMIUM ADD-ON
50 Mbps	10	20
100 Mbps	10	25
200 Mbps	10	25
500 Mbps	10	40
1 Gbps	10	50
2 Gbps	10	60
5 Gbps	10	75
10 Gbps	10	100
40 Gbps*	10	100
100 Gbps*	10	100

\* 100 Gbps ExpressRoute Direct Only

#### NOTE

Global Reach connections count against the limit of virtual network connections per ExpressRoute Circuit. For example, a 10 Gbps Premium Circuit would allow for 5 Global Reach connections and 95 connections to the ExpressRoute Gateways or 95 Global Reach connections and 5 connections to the ExpressRoute Gateways or any other combination up to the limit of 100 connections for the circuit.

## Virtual Network Gateway limits

RESOURCE	LIMIT
VNet Address Prefixes	600 per VPN gateway
Aggregate BGP routes	4,000 per VPN gateway
Local Network Gateway address prefixes	1000 per local network gateway
S2S connections	Depends on the gateway SKU
P2S connections	Depends on the gateway SKU
P2S route limit - IKEv2	256 for non-Windows / 25 for Windows
P2S route limit - OpenVPN	1000
Max. flows	100K for VpnGw1/AZ / 512K for VpnGw2-4/AZ

## NAT Gateway limits

RESOURCE	LIMIT
Public IP addresses	16 per NAT gateway

## Virtual WAN limits

RESOURCE	LIMIT
VPN (branch) connections per hub	1,000
Aggregate throughput per Virtual WAN Site-to-site VPN gateway	20 Gbps
Throughput per Virtual WAN VPN connection (2 tunnels)	2 Gbps with 1 Gbps/IPsec tunnel
Point-to-Site users per hub	100,000
Aggregate throughput per Virtual WAN User VPN (Point-to-site) gateway	200 Gbps
Aggregate throughput per Virtual WAN ExpressRoute gateway	20 Gbps
ExpressRoute Circuit connections per hub	8
VNet connections per hub	500 minus total number of hubs in Virtual WAN
Aggregate throughput per Virtual WAN Hub Router	50 Gbps for VNet to VNet transit
VM workload across all VNets connected to a single Virtual WAN hub	2000 (If you want to raise the limit or quota above the default limit, open an online customer support request.)

## Application Gateway limits

The following table applies to v1, v2, Standard, and WAF SKUs unless otherwise stated.

RESOURCE	LIMIT	NOTE
Azure Application Gateway	1,000 per subscription	
Front-end IP configurations	2	1 public and 1 private
Front-end ports	100 <sup>1</sup>	
Back-end address pools	100 <sup>1</sup>	
Back-end servers per pool	1,200	
HTTP listeners	200 <sup>1</sup>	Limited to 100 active listeners that are routing traffic. Active listeners = total number of listeners - listeners not active. If a default configuration inside a routing rule is set to route traffic (for example, it has a listener, a backend pool, and HTTP settings) then that also counts as a listener.
HTTP load-balancing rules	100 <sup>1</sup>	
Back-end HTTP settings	100 <sup>1</sup>	
Instances per gateway	V1 SKU - 32 V2 SKU - 125	
SSL certificates	100 <sup>1</sup>	1 per HTTP listener
Maximum SSL certificate size	V1 SKU - 10 KB V2 SKU - 16 KB	
Authentication certificates	100	
Trusted root certificates	100	
Request timeout minimum	1 second	
Request timeout maximum	24 hours	
Number of sites	100 <sup>1</sup>	1 per HTTP listener
URL maps per listener	1	
Maximum path-based rules per URL map	100	
Redirect configurations	100 <sup>1</sup>	
Number of rewrite rule sets	400	

RESOURCE	LIMIT	NOTE
Number of Header or URL configuration per rewrite rule set	40	
Number of conditions per rewrite rule set	40	
Concurrent WebSocket connections	Medium gateways 20k <sup>2</sup> Large gateways 50k <sup>2</sup>	
Maximum URL length	32KB	
Maximum header size for HTTP/2	16KB	
Maximum file upload size (Standard SKU)	V2 - 4 GB V1 - 2GB	
Maximum file upload size (WAF SKU)	V1 Medium - 100 MB V1 Large - 500 MB V2 - 750 MB V2 (with CRS 3.2 or newer) - 4GB	
WAF body size limit (without files)	V1 or V2 (with CRS 3.1 and older) - 128KB V2 (with CRS 3.2 or newer) - 2MB	
Maximum WAF custom rules	100	
Maximum WAF exclusions per Application Gateway	40	

<sup>1</sup> In case of WAF-enabled SKUs, you must limit the number of resources to 40.

<sup>2</sup> Limit is per Application Gateway instance not per Application Gateway resource.

## Network Watcher limits

RESOURCE	LIMIT	NOTE
Azure Network Watcher	1 per region	Network Watcher is created to enable access to the service. Only one instance of Network Watcher is required per subscription per region.
Packet capture sessions	10,000 per region	Number of sessions only, not saved captures.

## Private Link limits

The following limits apply to Azure private link:

RESOURCE	LIMIT
Number of private endpoints per virtual network	1000
Number of private endpoints per subscription	64000

RESOURCE	LIMIT
Number of private link services per subscription	800
Number of IP Configurations on a private link service	8 (This number is for the NAT IP addresses used per PLS)
Number of private endpoints on the same private link service	1000
Number of private endpoints per key vault	64
Number of key vaults with private endpoints per subscription	400
Number of private DNS zone groups that can be linked to a private endpoint	1
Number of DNS zones in each group	5

## Purview limits

The latest values for Azure Purview quotas can be found in the [Azure Purview quota page](#)

### Traffic Manager limits

RESOURCE	LIMIT
Profiles per subscription	200
Endpoints per profile	200

### Azure Bastion limits

WORKLOAD TYPE*	LIMIT**
Light	100
Medium	50
Heavy	5

\*These workload types are defined here: [Remote Desktop workloads](#)

\*\*These limits are based on RDP performance tests for Azure Bastion. The numbers may vary due to other on-going RDP sessions or other on-going SSH sessions.

### Azure DNS limits

#### Public DNS zones

RESOURCE	LIMIT
Public DNS Zones per subscription	250 <sup>1</sup>
Record sets per public DNS zone	10,000 <sup>1</sup>

RESOURCE	LIMIT
Records per record set in public DNS zone	20
Number of Alias records for a single Azure resource	20

<sup>1</sup>If you need to increase these limits, contact Azure Support.

## Private DNS zones

RESOURCE	LIMIT
Private DNS zones per subscription	1000
Record sets per private DNS zone	25000
Records per record set for private DNS zones	20
Virtual Network Links per private DNS zone	1000
Virtual Networks Links per private DNS zones with auto-registration enabled	100
Number of private DNS zones a virtual network can get linked to with auto-registration enabled	1
Number of private DNS zones a virtual network can get linked	1000
Number of DNS queries a virtual machine can send to Azure DNS resolver, per second	1000 <sup>1</sup>
Maximum number of DNS queries queued (pending response) per virtual machine	200 <sup>1</sup>

<sup>1</sup>These limits are applied to every individual virtual machine and not at the virtual network level. DNS queries exceeding these limits are dropped.

## Azure Firewall limits

RESOURCE	LIMIT
Data throughput	30 Gbps
Rule limits	10,000 unique source/destinations in network rules
Maximum DNAT rules	250
Minimum AzureFirewallSubnet size	/26
Port range in network and application rules	1 - 65535
Public IP addresses	250 maximum. All public IP addresses can be used in DNAT rules and they all contribute to available SNAT ports.

RESOURCE	LIMIT
IP addresses in IP Groups	Maximum of 100 IP Groups per firewall. Maximum 5000 individual IP addresses or IP prefixes per each IP Group.
Route table	<p>By default, AzureFirewallSubnet has a 0.0.0.0/0 route with the <b>NextHopType</b> value set to <b>Internet</b>.</p> <p>Azure Firewall must have direct Internet connectivity. If your AzureFirewallSubnet learns a default route to your on-premises network via BGP, you must override that with a 0.0.0.0/0 UDR with the <b>NextHopType</b> value set as <b>Internet</b> to maintain direct Internet connectivity. By default, Azure Firewall doesn't support forced tunneling to an on-premises network.</p> <p>However, if your configuration requires forced tunneling to an on-premises network, Microsoft will support it on a case by case basis. Contact Support so that we can review your case. If accepted, we'll allow your subscription and ensure the required firewall Internet connectivity is maintained.</p>
FQDNs in network rules	For good performance, do not exceed more than 1000 FQDNs across all network rules per firewall.

#### Azure Front Door Service limits

RESOURCE	LIMIT
Azure Front Door resources per subscription	100
Front-end hosts, which includes custom domains per resource	500
Routing rules per resource	500
Back-end pools per resource	50
Back ends per back-end pool	100
Path patterns to match for a routing rule	25
URLs in a single cache purge call	100
Custom web application firewall rules per policy	100
Web application firewall policy per subscription	100
Web application firewall match conditions per custom rule	10
Web application firewall IP address ranges per match condition	600
Web application firewall string match values per match condition	10

RESOURCE	LIMIT
Web application firewall string match value length	256
Web application firewall POST body parameter name length	256
Web application firewall HTTP header name length	256
Web application firewall cookie name length	256
Web application firewall exclusion limit	100
Web application firewall HTTP request body size inspected	128 KB
Web application firewall custom response body length	2 KB

### Azure Front Door Standard/Premium (Preview) Service Limits

\*\*\* Maximum 500 total Standard and Premium profiles per subscription.

RESOURCE	STANDARD SKU LIMIT	PREMIUM SKU LIMIT
Maximum endpoint per profile	10	25
Maximum custom domain per profile	100	200
Maximum origin group per profile	100	200
Maximum secrets per profile	100	200
Maximum security policy per profile	100	200
Maximum rule set per profile	100	200
Maximum rules per rule set	100	100
Maximum origin per origin group	50	50
Maximum routes per endpoint	100	200
URLs in a single cache purge call	100	100
Custom web application firewall rules per policy	100	100
Web application firewall match conditions per custom rule	10	10
Web application firewall IP address ranges per match condition	600	600
Web application firewall string match values per match condition	10	10

RESOURCE	STANDARD SKU LIMIT	PREMIUM SKU LIMIT
Web application firewall string match value length	256	256
Web application firewall POST body parameter name length	256	256
Web application firewall HTTP header name length	256	256
Web application firewall cookie name length	256	256
Web application firewall HTTP request body size inspected	128 KB	128 KB
Web application firewall custom response body length	2 KB	2 KB

#### Timeout values

##### Client to Front Door

- Front Door has an idle TCP connection timeout of 61 seconds.

##### Front Door to application back-end

- If the response is a chunked response, a 200 is returned if or when the first chunk is received.
- After the HTTP request is forwarded to the back end, Front Door waits for 30 seconds for the first packet from the back end. Then it returns a 503 error to the client. This value is configurable via the field sendRecvTimeoutSeconds in the API.
  - If a request is cached and it takes more than 30 seconds for the first packet from Front Door or from the backend, then a 504 error is returned to the client.
- After the first packet is received from the back end, Front Door waits for 30 seconds in an idle timeout. Then it returns a 503 error to the client. This timeout value is not configurable.
- Front Door to the back-end TCP session timeout is 90 seconds.

#### Upload and download data limit

	WITH CHUNKED TRANSFER ENCODING (CTE)	WITHOUT HTTP CHUNKING
Download	There's no limit on the download size.	There's no limit on the download size.
Upload	There's no limit as long as each CTE upload is less than 2 GB.	The size can't be larger than 2 GB.

#### Other limits

- Maximum URL size - 8,192 bytes - Specifies maximum length of the raw URL (scheme + hostname + port + path + query string of the URL)
- Maximum Query String size - 4,096 bytes - Specifies the maximum length of the query string, in bytes.
- Maximum HTTP response header size from health probe URL - 4,096 bytes - Specified the maximum length of all the response headers of health probes.
- Maximum rules engine action header value character: 640 characters.
- Maximum rules engine condition header value character: 256 characters.
- Maximum ETag header size: 128 bytes

## Notification Hubs limits

TIER	FREE	BASIC	STANDARD
Included pushes	1 million	10 million	10 million
Active devices	500	200,000	10 million
Tag quota per installation or registration	60	60	60

For more information on limits and pricing, see [Notification Hubs pricing](#).

## Service Bus limits

The following table lists quota information specific to Azure Service Bus messaging. For information about pricing and other quotas for Service Bus, see [Service Bus pricing](#).

QUOTA NAME	SCOPE	VALUE	NOTES
Maximum number of namespaces per Azure subscription	Namespace	1000 (default and maximum)	Subsequent requests for additional namespaces are rejected.
Queue or topic size	Entity	1, 2, 3, 4 GB or 5 GB. In the Premium SKU, and the Standard SKU with <a href="#">partitioning</a> enabled, the maximum queue or topic size is 80 GB.  Total size limit for a premium namespace is 1 TB per <a href="#">messaging unit</a> . Total size of all entities in a namespace can't exceed this limit.	Defined upon creation/updation of the queue or topic.  Subsequent incoming messages are rejected, and an exception is received by the calling code.
Number of concurrent connections on a namespace	Namespace	Net Messaging: 1,000. AMQP: 5,000.	Subsequent requests for additional connections are rejected, and an exception is received by the calling code. REST operations don't count toward concurrent TCP connections.
Number of concurrent receive requests on a queue, topic, or subscription entity	Entity	5,000	Subsequent receive requests are rejected, and an exception is received by the calling code. This quota applies to the combined number of concurrent receive operations across all subscriptions on a topic.

Quota name	Scope	Value	Notes
Number of topics or queues per namespace	Namespace	<p>10,000 for the Basic or Standard tier. The total number of topics and queues in a namespace must be less than or equal to 10,000.</p> <p>For the Premium tier, 1,000 per messaging unit (MU).</p>	Subsequent requests for creation of a new topic or queue on the namespace are rejected. As a result, if configured through the <a href="#">Azure portal</a> , an error message is generated. If called from the management API, an exception is received by the calling code.
Number of <a href="#">partitioned topics or queues</a> per namespace	Namespace	<p>Basic and Standard tiers: 100.</p> <p>Partitioned entities aren't supported in the <a href="#">Premium tier</a>.</p> <p>Each partitioned queue or topic counts toward the quota of 1,000 entities per namespace.</p>	<p>Subsequent requests for creation of a new partitioned topic or queue on the namespace are rejected. As a result, if configured through the <a href="#">Azure portal</a>, an error message is generated. If called from the management API, the exception <code>QuotaExceededException</code> is received by the calling code.</p> <p>If you want to have more partitioned entities in a basic or a standard tier namespace, create additional namespaces.</p>
Maximum size of any messaging entity path: queue or topic	Entity	-	260 characters.
Maximum size of any messaging entity name: namespace, subscription, or subscription rule	Entity	-	50 characters.
Maximum size of a message ID	Entity	-	128
Maximum size of a message session ID	Entity	-	128

Quota Name	Scope	Value	Notes
Message size for a queue, topic, or subscription entity	Entity	Incoming messages that exceed these quotas are rejected, and an exception is received by the calling code.	<p>Maximum message size: 256 KB for <a href="#">Standard tier</a>, 1 MB for <a href="#">Premium tier</a>.</p> <p>Due to system overhead, this limit is less than these values.</p> <p>Maximum header size: 64 KB.</p> <p>Maximum number of header properties in property bag: <code>byte/int.MaxValue</code>.</p> <p>Maximum size of property in property bag: Both the property name and value are limited at 32KB.</p>
Message property size for a queue, topic, or subscription entity	Entity	The exception <code>SerializationException</code> is generated.	Maximum message property size for each property is 32 KB. Cumulative size of all properties can't exceed 64 KB. This limit applies to the entire header of the brokered message, which has both user properties and system properties, such as sequence number, label, and message ID.
Number of subscriptions per topic	Entity	Subsequent requests for creating additional subscriptions for the topic are rejected. As a result, if configured through the portal, an error message is shown. If called from the management API, an exception is received by the calling code.	2,000 per-topic for the Standard tier and Premium tier.
Number of SQL filters per topic	Entity	Subsequent requests for creation of additional filters on the topic are rejected, and an exception is received by the calling code.	2,000
Number of correlation filters per topic	Entity	Subsequent requests for creation of additional filters on the topic are rejected, and an exception is received by the calling code.	100,000

Quota name	Scope	Value	Notes
Size of SQL filters or actions	Namespace	Subsequent requests for creation of additional filters are rejected, and an exception is received by the calling code.	Maximum length of filter condition string: 1,024 (1 K). Maximum length of rule action string: 1,024 (1 K). Maximum number of expressions per rule action: 32.
Number of shared access authorization rules per namespace, queue, or topic	Entity, namespace	Subsequent requests for creation of additional rules are rejected, and an exception is received by the calling code.	Maximum number of rules per entity type: 12. Rules that are configured on a Service Bus namespace apply to all types: queues, topics.
Number of messages per transaction	Transaction	Additional incoming messages are rejected, and an exception stating "Cannot send more than 100 messages in a single transaction" is received by the calling code.	100 For both <code>Send()</code> and <code>SendAsync()</code> operations.
Number of virtual network and IP filter rules	Namespace		128

## Site Recovery limits

The following limits apply to Azure Site Recovery.

Limit identifier	Limit
Number of vaults per subscription	500
Number of servers per Recovery Services vault	250
Number of protection groups per Recovery Services vault	No limit
Number of recovery plans per Recovery Services vault	No limit
Number of servers per protection group	No limit
Number of servers per recovery plan	100

## SQL Database limits

For SQL Database limits, see [SQL Database resource limits for single databases](#), [SQL Database resource limits for elastic pools and pooled databases](#), and [SQL Database resource limits for SQL Managed Instance](#).

The maximum number of private endpoints per Azure SQL Database logical server is 250.

# Azure Synapse Analytics limits

Azure Synapse Analytics has the following default limits to ensure customer's subscriptions are protected from each other's workloads. To raise the limits to the maximum for your subscription, contact support.

## Synapse Workspace Limits

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Synapse workspaces in an Azure subscription	20	20

## Synapse Pipeline Limits

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Synapse pipelines in a Synapse workspace	800	800
Total number of entities, such as pipelines, data sets, triggers, linked services, Private Endpoints, and integration runtimes, within a workspace	5,000	Contact support.
Total CPU cores for Azure-SSIS Integration Runtimes under one workspace	256	Contact support.
Concurrent pipeline runs per workspace that's shared among all pipelines in the workspace	10,000	10,000
Concurrent External activity runs per workspace per <a href="#">Azure Integration Runtime region</a>  External activities are managed on integration runtime but execute on linked services, including Databricks, stored procedure, HDInsight, Web, and others. This limit does not apply to Self-hosted IR.	3,000	3,000
Concurrent Pipeline activity runs per workspace per <a href="#">Azure Integration Runtime region</a>  Pipeline activities execute on integration runtime, including Lookup, GetMetadata, and Delete. This limit does not apply to Self-hosted IR.	1,000	1,000
Concurrent authoring operations per workspace per <a href="#">Azure Integration Runtime region</a>  Including test connection, browse folder list and table list, preview data. This limit does not apply to Self-hosted IR.	200	200

Resource	Default Limit	Maximum Limit
Concurrent Data Integration Units <sup>1</sup> consumption per workspace per <a href="#">Azure Integration Runtime region</a>	Region group 1 <sup>2</sup> : 6,000 Region group 2 <sup>2</sup> : 3,000 Region group 3 <sup>2</sup> : 1,500	Region group 1 <sup>2</sup> : 6,000 Region group 2 <sup>2</sup> : 3,000 Region group 3 <sup>2</sup> : 1,500
Maximum activities per pipeline, which includes inner activities for containers	40	40
Maximum number of linked integration runtimes that can be created against a single self-hosted integration runtime	100	Contact support.
Maximum parameters per pipeline	50	50
ForEach items	100,000	100,000
ForEach parallelism	20	50
Maximum queued runs per pipeline	100	100
Characters per expression	8,192	8,192
Minimum tumbling window trigger interval	15 min	15 min
Maximum timeout for pipeline activity runs	7 days	7 days
Bytes per object for pipeline objects <sup>3</sup>	200 KB	200 KB
Bytes per object for dataset and linked service objects <sup>3</sup>	100 KB	2,000 KB
Bytes per payload for each activity run <sup>4</sup>	896 KB	896 KB
Data Integration Units <sup>1</sup> per copy activity run	256	256
Write API calls	1,200/h	1,200/h  This limit is imposed by Azure Resource Manager, not Azure Synapse Analytics.
Read API calls	12,500/h	12,500/h  This limit is imposed by Azure Resource Manager, not Azure Synapse Analytics.
Monitoring queries per minute	1,000	1,000
Maximum time of data flow debug session	8 hrs	8 hrs

RESOURCE	DEFAULT LIMIT	MAXIMUM LIMIT
Concurrent number of data flows per integration runtime	50	Contact support.
Concurrent number of data flow debug sessions per user per workspace	3	3
Data Flow Azure IR TTL limit	4 hrs	4 hrs
Meta Data Entity Size limit in a workspace	2 GB	Contact support.

<sup>1</sup> The data integration unit (DIU) is used in a cloud-to-cloud copy operation, learn more from [Data integration units \(version 2\)](#). For information on billing, see [Azure Synapse Analytics Pricing](#).

<sup>2</sup> [Azure Integration Runtime](#) is [globally available](#) to ensure data compliance, efficiency, and reduced network egress costs.

REGION GROUP	REGIONS
Region group 1	Central US, East US, East US 2, North Europe, West Europe, West US, West US 2
Region group 2	Australia East, Australia Southeast, Brazil South, Central India, Japan East, North Central US, South Central US, Southeast Asia, West Central US
Region group 3	Other regions

<sup>3</sup> Pipeline, data set, and linked service objects represent a logical grouping of your workload. Limits for these objects don't relate to the amount of data you can move and process with Azure Synapse Analytics. Synapse Analytics is designed to scale to handle petabytes of data.

<sup>4</sup> The payload for each activity run includes the activity configuration, the associated dataset(s) and linked service(s) configurations if any, and a small portion of system properties generated per activity type. Limit for this payload size doesn't relate to the amount of data you can move and process with Azure Synapse Analytics. Learn about the [symptoms and recommendation](#) if you hit this limit.

### Dedicated SQL pool limits

For details of capacity limits for dedicated SQL pools in Azure Synapse Analytics, see [dedicated SQL pool resource limits](#).

### Web service call limits

Azure Resource Manager has limits for API calls. You can make API calls at a rate within the [Azure Resource Manager API limits](#).

## Azure Files and Azure File Sync

To learn more about the limits for Azure Files and File Sync, see [Azure Files scalability and performance targets](#).

## Storage limits

The following table describes default limits for Azure general-purpose v1, v2, Blob storage, and block blob

storage accounts. The *ingress* limit refers to all data that is sent to a storage account. The *egress* limit refers to all data that is received from a storage account.

**NOTE**

You can request higher capacity and ingress limits. To request an increase, contact [Azure Support](#).

RESOURCE	LIMIT
Number of storage accounts per region per subscription, including standard, and premium storage accounts.	250
Maximum storage account capacity	5 PiB <sup>1</sup>
Maximum number of blob containers, blobs, file shares, tables, queues, entities, or messages per storage account	No limit
Maximum request rate <sup>1</sup> per storage account	20,000 requests per second
Maximum ingress <sup>1</sup> per storage account (US, Europe regions)	10 Gbps
Maximum ingress <sup>1</sup> per storage account (regions other than US and Europe)	5 Gbps if RA-GRS/GRS is enabled, 10 Gbps for LRS/ZRS <sup>2</sup>
Maximum egress for general-purpose v2 and Blob storage accounts (all regions)	50 Gbps
Maximum egress for general-purpose v1 storage accounts (US regions)	20 Gbps if RA-GRS/GRS is enabled, 30 Gbps for LRS/ZRS <sup>2</sup>
Maximum egress for general-purpose v1 storage accounts (non-US regions)	10 Gbps if RA-GRS/GRS is enabled, 15 Gbps for LRS/ZRS <sup>2</sup>
Maximum number of virtual network rules per storage account	200
Maximum number of IP address rules per storage account	200

<sup>1</sup> Azure Storage standard accounts support higher capacity limits and higher limits for ingress by request. To request an increase in account limits, contact [Azure Support](#).

<sup>2</sup> If your storage account has read-access enabled with geo-redundant storage (RA-GRS) or geo-zone-redundant storage (RA-GZRS), then the egress targets for the secondary location are identical to those of the primary location. For more information, see [Azure Storage replication](#).

**NOTE**

Microsoft recommends that you use a general-purpose v2 storage account for most scenarios. You can easily upgrade a general-purpose v1 or an Azure Blob storage account to a general-purpose v2 account with no downtime and without the need to copy data. For more information, see [Upgrade to a general-purpose v2 storage account](#).

All storage accounts run on a flat network topology regardless of when they were created. For more information on the Azure Storage flat network architecture and on scalability, see [Microsoft Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency](#).

For more information on limits for standard storage accounts, see [Scalability targets for standard storage accounts](#).

### Storage resource provider limits

The following limits apply only when you perform management operations by using Azure Resource Manager with Azure Storage.

RESOURCE	LIMIT
Storage account management operations (read)	800 per 5 minutes
Storage account management operations (write)	10 per second / 1200 per hour
Storage account management operations (list)	100 per 5 minutes

### Azure Blob storage limits

RESOURCE	TARGET
Maximum size of single blob container	Same as maximum storage account capacity
Maximum number of blocks in a block blob or append blob	50,000 blocks
Maximum size of a block in a block blob	4000 MiB
Maximum size of a block blob	50,000 X 4000 MiB (approximately 190.7 TiB)
Maximum size of a block in an append blob	4 MiB
Maximum size of an append blob	50,000 x 4 MiB (approximately 195 GiB)
Maximum size of a page blob	8 TiB <sup>2</sup>
Maximum number of stored access policies per blob container	5
Target request rate for a single blob	Up to 500 requests per second
Target throughput for a single page blob	Up to 60 MiB per second <sup>2</sup>
Target throughput for a single block blob	Up to storage account ingress/egress limits <sup>1</sup>

<sup>1</sup> Throughput for a single blob depends on several factors, including, but not limited to: concurrency, request size, performance tier, speed of source for uploads, and destination for downloads. To take advantage of the performance enhancements of [high-throughput block blobs](#), upload larger blobs or blocks. Specifically, call the [Put Blob](#) or [Put Block](#) operation with a blob or block size that is greater than 4 MiB for standard storage accounts. For premium block blob or for Data Lake Storage Gen2 storage accounts, use a block or blob size that is greater than 256 KiB.

<sup>2</sup> Page blobs are not yet supported in accounts that have the [Hierarchical namespace](#) setting on them.

The following table describes the maximum block and blob sizes permitted by service version.

Service Version	Maximum Block Size (via Put Block)	Maximum Blob Size (via Put Block List)	Maximum Blob Size via Single Write Operation (via Put Blob)
Version 2019-12-12 and later	4000 MiB	Approximately 190.7 TiB (4000 MiB X 50,000 blocks)	5000 MiB (preview)
Version 2016-05-31 through version 2019-07-07	100 MiB	Approximately 4.75 TiB (100 MiB X 50,000 blocks)	256 MiB
Versions prior to 2016-05-31	4 MiB	Approximately 195 GiB (4 MiB X 50,000 blocks)	64 MiB

### Azure Queue storage limits

Resource	Target
Maximum size of a single queue	500 TiB
Maximum size of a message in a queue	64 KiB
Maximum number of stored access policies per queue	5
Maximum request rate per storage account	20,000 messages per second, which assumes a 1-KiB message size
Target throughput for a single queue (1-KiB messages)	Up to 2,000 messages per second

### Azure Table storage limits

The following table describes capacity, scalability, and performance targets for Table storage.

Resource	Target
Number of tables in an Azure storage account	Limited only by the capacity of the storage account
Number of partitions in a table	Limited only by the capacity of the storage account
Number of entities in a partition	Limited only by the capacity of the storage account
Maximum size of a single table	500 TiB
Maximum size of a single entity, including all property values	1 MiB
Maximum number of properties in a table entity	255 (including the three system properties, <b>PartitionKey</b> , <b>RowKey</b> , and <b>Timestamp</b> )
Maximum total size of an individual property in an entity	Varies by property type. For more information, see <b>Property Types</b> in <a href="#">Understanding the Table Service Data Model</a> .
Size of the <b>PartitionKey</b>	A string up to 1 KiB in size
Size of the <b>RowKey</b>	A string up to 1 KiB in size

RESOURCE	TARGET
Size of an entity group transaction	A transaction can include at most 100 entities and the payload must be less than 4 MiB in size. An entity group transaction can include an update to an entity only once.
Maximum number of stored access policies per table	5
Maximum request rate per storage account	20,000 transactions per second, which assumes a 1-KiB entity size
Target throughput for a single table partition (1 KiB-entities)	Up to 2,000 entities per second

### Virtual machine disk limits

You can attach a number of data disks to an Azure virtual machine. Based on the scalability and performance targets for a VM's data disks, you can determine the number and type of disk that you need to meet your performance and capacity requirements.

#### IMPORTANT

For optimal performance, limit the number of highly utilized disks attached to the virtual machine to avoid possible throttling. If all attached disks aren't highly utilized at the same time, the virtual machine can support a larger number of disks.

### For Azure managed disks:

The following table illustrates the default and maximum limits of the number of resources per region per subscription. The limits remain the same irrespective of disks encrypted with either platform-managed keys or customer-managed keys. There is no limit for the number of Managed Disks, snapshots and images per resource group.

RESOURCE	LIMIT
Standard managed disks	50,000
Standard SSD managed disks	50,000
Premium managed disks	50,000
Standard_LRS snapshots	75,000
Standard_ZRS snapshots	75,000
Managed image	50,000

**For Standard storage accounts:** A Standard storage account has a maximum total request rate of 20,000 IOPS. The total IOPS across all of your virtual machine disks in a Standard storage account should not exceed this limit.

You can roughly calculate the number of highly utilized disks supported by a single Standard storage account based on the request rate limit. For example, for a Basic tier VM, the maximum number of highly utilized disks is about 66, which is  $20,000/300$  IOPS per disk. The maximum number of highly utilized disks for a Standard tier

VM is about 40, which is 20,000/500 IOPS per disk.

**For Premium storage accounts:** A Premium storage account has a maximum total throughput rate of 50 Gbps. The total throughput across all of your VM disks should not exceed this limit.

For more information, see [Virtual machine sizes](#).

#### Disk encryption sets

There's a limitation of 1000 disk encryption sets per region, per subscription. For more information, see the encryption documentation for [Linux](#) or [Windows](#) virtual machines. If you need to increase the quota, contact Azure support.

### Managed virtual machine disks

#### Standard HDD managed disks

STANDARD DISK TYPE	S4	S6	S10	S15	S20	S30	S40	S50	S60	S70	S80
Disk size in GiB	32	64	128	256	512	1,024	2,048	4,096	8,192	16,384	32,767
IOPS per disk	Up to 500	Up to 1,300	Up to 2,000	Up to 2,000							
Throughput per disk	Up to 60 MB/sec	Up to 300 MB/sec	Up to 500 MB/sec	Up to 500 MB/sec							

#### Standard SSD managed disks

STANDARD SSD SIZES	E1	E2	E3	E4	E6	E10	E15	E20	E30	E40	E50	E60	E70	E80
Disk size in GiB	4	8	16	32	64	128	256	512	1,024	2,048	4,096	8,192	16,384	32,767
IOPS per disk	Up to 500	Up to 2,000	Up to 4,000	Up to 6,000										
Throughput per disk	Up to 60 MB/sec	Up to 400 MB/sec	Up to 600 MB/sec	Up to 750 MB/sec										

STA ND AR D SSD SIZ ES	E1	E2	E3	E4	E6	E10	E15	E20	E30	E40	E50	E60	E70	E80
Max burst IOPS per disk	600	600	600	600	600	600	600	600	1000					
Max burst throughput per disk	150 MB/sec	250 MB/sec												
Max burst duration	30 min													

#### Premium SSD managed disks: Per-disk limits

PRE MIU M SSD SIZ ES	P1	P2	P3	P4	P6	P10	P15	P20	P30	P40	P50	P60	P70	P80
Disk size in GiB	4	8	16	32	64	128	256	512	1,024	2,048	4,096	8,192	16,384	32,767
Provisioned IOPS per disk	120	120	120	120	240	500	1,100	2,300	5,000	7,500	7,500	16,000	18,000	20,000

PRE MIU M SSD SIZ ES	P1	P2	P3	P4	P6	P10	P15	P20	P30	P40	P50	P60	P70	P80
Pro visi one d Thr oug hpu t per disk	25 MB/ sec	25 MB/ sec	25 MB/ sec	25 MB/ sec	50 MB/ sec	100 MB/ sec	125 MB/ sec	150 MB/ sec	200 MB/ sec	250 MB/ sec	250 MB/ sec	500 MB/ sec	750 MB/ sec	900 MB/ sec
Ma x bur st IOP S per disk	3,5 00	30, 000 *	30, 000 *	30, 000 *	30, 000 *	30, 000 *	30, 000 *							
Ma x bur st thr oug hpu t per disk	170 MB/ sec	1,0 00 MB/ sec*	1,0 00 MB/ sec*	1,0 00 MB/ sec*	1,0 00 MB/ sec*	1,0 00 MB/ sec*	1,0 00 MB/ sec*							
Ma x bur st dur atio n	30 min	Unli mit ed*	Unli mit ed*	Unli mit ed*	Unli mit ed*	Unli mit ed*	Unli mit ed*							
Eligi ble for rese rvat ion	No	Yes, up to one yea r	Yes, up to one year	Yes, up to one year	Yes, up to one year	Yes, up to one year	Yes, up to one year							

\*Applies only to disks with on-demand bursting enabled.

#### Premium SSD managed disks: Per-VM limits

RESOURCE	LIMIT
Maximum IOPS Per VM	80,000 IOPS with GS5 VM

RESOURCE	LIMIT
Maximum throughput per VM	2,000 MB/s with GS5 VM

## Unmanaged virtual machine disks

### Standard unmanaged virtual machine disks: Per-disk limits

VM TIER	BASIC TIER VM	STANDARD TIER VM
Disk size	4,095 GB	4,095 GB
Maximum 8-KB IOPS per persistent disk	300	500
Maximum number of disks that perform the maximum IOPS	66	40

### Premium unmanaged virtual machine disks: Per-account limits

RESOURCE	LIMIT
Total disk capacity per account	35 TB
Total snapshot capacity per account	10 TB
Maximum bandwidth per account (ingress + egress) <sup>1</sup>	<=50 Gbps

<sup>1</sup>Ingress refers to all data from requests that are sent to a storage account. Egress refers to all data from responses that are received from a storage account.

### Premium unmanaged virtual machine disks: Per-disk limits

PREMIUM STORAGE DISK TYPE	P10	P20	P30	P40	P50
Disk size	128 GiB	512 GiB	1,024 GiB (1 TB)	2,048 GiB (2 TB)	4,095 GiB (4 TB)
Maximum IOPS per disk	500	2,300	5,000	7,500	7,500
Maximum throughput per disk	100 MB/sec	150 MB/sec	200 MB/sec	250 MB/sec	250 MB/sec
Maximum number of disks per storage account	280	70	35	17	8

### Premium unmanaged virtual machine disks: Per-VM limits

RESOURCE	LIMIT
Maximum IOPS per VM	80,000 IOPS with GS5 VM

RESOURCE	LIMIT
Maximum throughput per VM	2,000 MB/sec with GS5 VM

## StorSimple System limits

LIMIT IDENTIFIER	LIMIT	COMMENTS
Maximum number of storage account credentials	64	
Maximum number of volume containers	64	
Maximum number of volumes	255	
Maximum number of schedules per bandwidth template	168	A schedule for every hour, every day of the week.
Maximum size of a tiered volume on physical devices	64 TB for StorSimple 8100 and StorSimple 8600	StorSimple 8100 and StorSimple 8600 are physical devices.
Maximum size of a tiered volume on virtual devices in Azure	30 TB for StorSimple 8010 64 TB for StorSimple 8020	StorSimple 8010 and StorSimple 8020 are virtual devices in Azure that use Standard storage and Premium storage, respectively.
Maximum size of a locally pinned volume on physical devices	9 TB for StorSimple 8100 24 TB for StorSimple 8600	StorSimple 8100 and StorSimple 8600 are physical devices.
Maximum number of iSCSI connections	512	
Maximum number of iSCSI connections from initiators	512	
Maximum number of access control records per device	64	
Maximum number of volumes per backup policy	24	
Maximum number of backups retained per backup policy	64	
Maximum number of schedules per backup policy	10	
Maximum number of snapshots of any type that can be retained per volume	256	This amount includes local snapshots and cloud snapshots.
Maximum number of snapshots that can be present in any device	10,000	

LIMIT IDENTIFIER	LIMIT	COMMENTS
Maximum number of volumes that can be processed in parallel for backup, restore, or clone	16	<ul style="list-style-type: none"> <li>If there are more than 16 volumes, they're processed sequentially as processing slots become available.</li> <li>New backups of a cloned or a restored tiered volume can't occur until the operation is finished. For a local volume, backups are allowed after the volume is online.</li> </ul>
Restore and clone recover time for tiered volumes	<2 minutes	<ul style="list-style-type: none"> <li>The volume is made available within 2 minutes of a restore or clone operation, regardless of the volume size.</li> <li>The volume performance might initially be slower than normal as most of the data and metadata still resides in the cloud. Performance might increase as data flows from the cloud to the StorSimple device.</li> <li>The total time to download metadata depends on the allocated volume size. Metadata is automatically brought into the device in the background at the rate of 5 minutes per TB of allocated volume data. This rate might be affected by Internet bandwidth to the cloud.</li> <li>The restore or clone operation is complete when all the metadata is on the device.</li> <li>Backup operations can't be performed until the restore or clone operation is fully complete.</li> </ul>

LIMIT IDENTIFIER	LIMIT	COMMENTS
Restore recover time for locally pinned volumes	<2 minutes	<ul style="list-style-type: none"> <li>The volume is made available within 2 minutes of the restore operation, regardless of the volume size.</li> <li>The volume performance might initially be slower than normal as most of the data and metadata still resides in the cloud. Performance might increase as data flows from the cloud to the StorSimple device.</li> <li>The total time to download metadata depends on the allocated volume size. Metadata is automatically brought into the device in the background at the rate of 5 minutes per TB of allocated volume data. This rate might be affected by Internet bandwidth to the cloud.</li> <li>Unlike tiered volumes, if there are locally pinned volumes, the volume data is also downloaded locally on the device. The restore operation is complete when all the volume data has been brought to the device.</li> <li>The restore operations might be long and the total time to complete the restore will depend on the size of the provisioned local volume, your Internet bandwidth, and the existing data on the device. Backup operations on the locally pinned volume are allowed while the restore operation is in progress.</li> </ul>
Thin-restore availability	Last failover	
Maximum client read/write throughput, when served from the SSD tier*	920/720 MB/sec with a single 10-gigabit Ethernet network interface	Up to two times with MPIO and two network interfaces.
Maximum client read/write throughput, when served from the HDD tier*	120/250 MB/sec	
Maximum client read/write throughput, when served from the cloud tier*	11/41 MB/sec	Read throughput depends on clients generating and maintaining sufficient I/O queue depth.

\*Maximum throughput per I/O type was measured with 100 percent read and 100 percent write scenarios. Actual throughput might be lower and depends on I/O mix and network conditions.

## Stream Analytics limits

LIMIT IDENTIFIER	LIMIT	COMMENTS
Maximum number of streaming units per subscription per region	500	To request an increase in streaming units for your subscription beyond 500, contact <a href="#">Microsoft Support</a> .
Maximum number of inputs per job	60	There's a hard limit of 60 inputs per Azure Stream Analytics job.
Maximum number of outputs per job	60	There's a hard limit of 60 outputs per Stream Analytics job.
Maximum number of functions per job	60	There's a hard limit of 60 functions per Stream Analytics job.
Maximum number of streaming units per job	192	There's a hard limit of 192 streaming units per Stream Analytics job.
Maximum number of jobs per region	1,500	Each subscription can have up to 1,500 jobs per geographical region.
Reference data blob MB	5 GB	Up to 5 GB when using 6 SUs or more.
Maximum number of characters in a query	512000	There's a hard limit of 512k characters in an Azure Stream Analytics job query.

## Virtual Machines limits

### Virtual Machines limits

RESOURCE	LIMIT
Virtual machines per cloud service <sup>1</sup>	50
Input endpoints per cloud service <sup>2</sup>	150

<sup>1</sup> Virtual machines created by using the classic deployment model instead of Azure Resource Manager are automatically stored in a cloud service. You can add more virtual machines to that cloud service for load balancing and availability.

<sup>2</sup> Input endpoints allow communications to a virtual machine from outside the virtual machine's cloud service. Virtual machines in the same cloud service or virtual network can automatically communicate with each other.

### Virtual Machines limits - Azure Resource Manager

The following limits apply when you use Azure Resource Manager and Azure resource groups.

RESOURCE	LIMIT
VMs per <a href="#">subscription</a>	25,000 <sup>1</sup> per region.
VM total cores per <a href="#">subscription</a>	20 <sup>1</sup> per region. Contact support to increase limit.

RESOURCE	LIMIT
Azure Spot VM total cores per <a href="#">subscription</a>	20 <sup>1</sup> per region. Contact support to increase limit.
VM per series, such as Dv2 and F, cores per <a href="#">subscription</a>	20 <sup>1</sup> per region. Contact support to increase limit.
<a href="#">Availability sets</a> per subscription	2,500 per region.
Virtual machines per availability set	200
<a href="#">Proximity placement groups</a> per <a href="#">resource group</a>	800
Certificates per availability set	199 <sup>2</sup>
Certificates per subscription	Unlimited <sup>3</sup>

<sup>1</sup> Default limits vary by offer category type, such as Free Trial and Pay-As-You-Go, and by series, such as Dv2, F, and G. For example, the default for Enterprise Agreement subscriptions is 350. For security, subscriptions default to 20 cores to prevent large core deployments. If you need more cores, submit a support ticket.

<sup>2</sup> Properties such as SSH public keys are also pushed as certificates and count towards this limit. To bypass this limit, use the [Azure Key Vault extension for Windows](#) or the [Azure Key Vault extension for Linux](#) to install certificates.

<sup>3</sup> With Azure Resource Manager, certificates are stored in the Azure Key Vault. The number of certificates is unlimited for a subscription. There's a 1-MB limit of certificates per deployment, which consists of either a single VM or an availability set.

#### NOTE

Virtual machine cores have a regional total limit. They also have a limit for regional per-size series, such as Dv2 and F. These limits are separately enforced. For example, consider a subscription with a US East total VM core limit of 30, an A series core limit of 30, and a D series core limit of 30. This subscription can deploy 30 A1 VMs, or 30 D1 VMs, or a combination of the two not to exceed a total of 30 cores. An example of a combination is 10 A1 VMs and 20 D1 VMs.

## Shared Image Gallery limits

There are limits, per subscription, for deploying resources using Shared Image Galleries:

- 100 shared image galleries, per subscription, per region
- 1,000 image definitions, per subscription, per region
- 10,000 image versions, per subscription, per region

## Virtual machine scale sets limits

RESOURCE	LIMIT
Maximum number of VMs in a scale set	1,000
Maximum number of VMs based on a custom VM image in a scale set	600
Maximum number of scale sets in a region	2,500

## See also

- [Understand Azure limits and increases](#)
- [Virtual machine and cloud service sizes for Azure](#)
- [Sizes for Azure Cloud Services](#)
- [Naming rules and restrictions for Azure resources](#)