Developer Guide

AWS AppSync Events

AWS AppSync Events: Developer Guide

Table of Contents

What is AWS AppSync Events? 1
AWS AppSync Events features 2
Pricing for AWS AppSync Events 3
Core concepts 4
API 4
Event 4
Channel 4
Channel namespace 4
Event handler 5
Publishing 5
Subscribing 5
Getting started 6
Prerequisites 6
Sign up for an AWS account 6
Create a user with administrative access 7
Account credentials 8
Set up the AWS Command Line Interface 8
Creating an Event API 8
Step 1: Create an event API using the AWS AppSync console 9
Step 2: Publish and subscribe to receive your first event 9
Step 3: Use wildcards in your channel subscription 10
Step 4: Publish in batches 11
Using the Amplify client 11
Step 1: Create an event API 11
Step 2: Deploy a React app with Vite 12
Step 3: Configure the Amplify client 13
Step 4: Connect to a channel and receive messages 14
Step 5: Send a message from your app 17
Channel namespaces 19
Channel namespace authorization 19
Channel namespace handlers and event processing 19
Using the onPublish event handler 20
Using the onSubscribe event handler 22
Authorizing and authenticating Event APIs 24

Authorization types ... 24
API_KEY authorization ... 25
AWS_LAMBDA authorization ... 26
AWS_IAM authorization ... 29
OPENID_CONNECT authorization ... 31
AMAZON_COGNITO_USER_POOLS authorization ... 32
Circumventing SigV4 and OIDC token authorization limitations ... 33
Publish via HTTP ... 34
Event API WebSocket protocol ... 36
Handshake details to establish the WebSocket connection ... 39
Discovering the real-time endpoint from the Event API endpoint ... 40
Authorization formatting based on the AWS AppSync API authorization mode ... 41
API key subprotocol format ... 41
Amazon Cognito user pools and OpenID Connect (OIDC) subprotocol format ... 41
AWS Lambda subprotocol format ... 42
AWS Identity and Access Management (IAM) subprotocol format ... 42
Real-time WebSocket operations ... 44
Configuring message details ... 45
Disconnecting the WebSocket ... 49
Configuring custom domain names ... 50
Registering and configuring a domain name ... 50
Creating a custom domain name ... 51
Wildcard custom domain names ... 50
CloudWatch logging and monitoring ... 53
Setting up and configuring logging on an Event API ... 53
Manually creating an IAM role with CloudWatch Logs permissions ... 54
CloudWatch metrics ... 55
HTTP endpoint metrics ... 55
Real-time endpoint metrics ... 57
Configuring CloudWatch Logs on Event APIs ... 61
Using token counts to optimize your requests ... 62
Using AWS WAF to protect APIs ... 64
Integrate an AppSync API with AWS WAF ... 64
Creating rules for a web ACL ... 66
Document History ... 69

What is AWS AppSync Events?

AWS AppSync Events lets you create secure and performant serverless WebSocket APIs that can broadcast real-time event data to millions of subscribers, without you having to manage connections or resource scaling. With AWS AppSync Events, there is no API code required to get started, so you can create production-ready real-time web and mobile experiences in minutes.

AWS AppSync Events further simplifies the management and scaling of real-time applications by shifting tasks like message transformation and broadcast, publish and subscribe authentication, and the creation of logs and metrics to AWS, while delivering reduced time to market, low latency, enhanced security, and lower total costs.

With Event APIs, you can enable the following network communication types.

- Unicast
- Multicast
- Broadcast
- Batch publishing and messaging

This allows you to build the following types of interactive and collaborative experiences.

- Live chat and messaging
- Sports and score updates
- Real-time in-app alerts and notifications
- Live commenting and activity feeds

AWS AppSync Events simplifies real-time application development by providing the following features.

- Automatic management of WebSocket connections and scaling
- Built-in support for broadcasting events to large numbers of subscribers
- Flexible event filtering and transformation capabilities
- Fine-grained authentication and authorization
- Seamless integration with other AWS services and external systems for event-driven architectures

Whether you're building a small prototype or a large-scale production application, AWS AppSync Events enables you to incorporate real-time experiences using a fully managed and scalable platform, so you can focus on your application logic instead of the undifferentiated heavy lifting of managing infrastructure.

AWS AppSync Events features

WebSocket and HTTP Support

Clients can publish events over HTTP and subscribe to channels using WebSockets.

Event APIs provide WebSocket endpoints that enable real-time and pub/sub capabilities.

Channel namespaces and channels

Events are published to channels, that are grouped under namespaces.

Namespaces allow you to define authentication and authorization rules and serverless functions that apply to all channels within that namespace.

Namespace handlers

You can use the following two types of handlers to configure functions that run in response to publish and subscribe actions.

- OnPublish - Runs when an event is published to a channel, allowing you to transform, filter, and reject events.
- OnSubscribe - Runs when a client subscribes to a channel, allowing you to customize the behavior or reject the subscription request.

Flexible authentication and authorization

Event APIs supports various authentication mechanisms (API key, IAM, Amazon Cognito, OIDC, and Lambda authorizers) that can be configured at the API level and customized at the channel namespace level.

Channel subscriptions

Clients receive events for channels they are subscribed to.

Wildcard Channel Subscriptions

Clients can subscribe to a group of related channels using a wildcard syntax (e.g., "namespace/channel/*"), allowing them to receive events from multiple channels without explicitly subscribing to each one.

Scalable Event Broadcasting

The Event API automatically scales to handle large numbers of concurrent connections and can efficiently broadcast events to all subscribed clients.

Integration with the AWS Ecosystem

AWS AppSync Events integrates with other AWS services like Amazon CloudWatch Logs, CloudWatch metrics, and AWS WAF. You can easily implement event-driven architectures by publishing directly from services like Amazon EventBridge, and AWS Lambda. Amazon Cognito is directly supported as an authorization type.

Pricing for AWS AppSync Events

When you use AWS AppSync Events, you pay only for what you use with no minimum fees or mandatory service usage. For more information, see AWS AppSync pricing.

AWS AppSync Events concepts

Before you get started, review the following topics to help you understand the fundamental concepts of AWS AppSync Events.

API

An Event API provides real-time capabilities by enabling you to publish events over HTTP and subscribe to events over WebSockets. An Event API has one or more channel namespaces that define the capabilities and behavior of channels that events can be addressed to. To learn more about configuring an API, see Configuring authorization and authentication to secure Event APIs and Configuring custom domain names for Event APIs.

Event

An event is a JSON-formatted unit of data that can be published to channels on your API and received by clients that are interested in that channel. Events can contain any arbitrary data you want to transmit in real-time, such as user actions, data updates, system notifications, or sensor readings. Events are designed to be lightweight and efficient, with a maximum size of 240 KB per event.

Channel

Channels are the routing mechanism for directing events from publishers to subscribers. You can think of a channel as a "topic" or "subject" that represents a stream of related events. Clients subscribe to channels in order to receive events published to those channels in real-time. Channels are ephemeral and can be created on-demand.

Channel namespace

A channel namespace (or just namespace for short) provides a way to define the capabilities and behaviors of the channels associated with it. Each namespace has a name. This name represents the starting segment (the prefix) of a channel path. Any channel where the first segment in the path matches the name of a namespace, belongs to that namespace. For example, any channel with a first segment of `default`, such as `/default/messages`, `/default/greetings`, and

`/default/inbox/user` belongs to the namespace named `default`. To learn more about namespaces, see Understanding channel namespaces.

Event handler

An event handler is a function defined in a namespace. An event handler is a custom function to process published events before they are broadcast to subscribers. A event handler can also be used to process subscription requests when clients try to subscribe to a channel.

Publishing

Publishing is the act of sending a batch of events to your Event API. Published events can be broadcasted to subscribed clients. Publish is done over HTTP.

Subscribing

Subscribing is the act of listening for events on a specific channel or subset of channels over an Event API WebSocket. Clients that subscribe can receive broadcast events in real-time. Clients can establish multiple subscriptions over a single WebSocket.

Getting started with AWS AppSync Events

You can quickly get started with AWS AppSync Events by creating an AWS AppSync Event API and accessing it from a client. The first tutorial in this section will guide you through creating your first AWS AppSync Event API in the AWS AppSync console. Then you will learn to publish and subscribe to your event. The second tutorial guides you through creating a React app with Vite and an Amplify client. Then you will use this Amplify client to publish and subscribe messages.

Topics

- Prerequisites
- Creating an AWS AppSync Event API
- Getting started with the Amplify Events client

Prerequisites

Before you begin the getting started tutorials, confirm that you have completed the prerequisites to get set up with an AWS account.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

Account credentials

Although you can use your root user credentials to access AWS AppSync, we recommend that you use an AWS Identity and Access Management (IAM) account instead. You can use the AWSAppSyncAdministrator managed policy to grant your IAM account the correct permissions to manage your AWS AppSync resources.

Set up the AWS Command Line Interface

You can use the AWS Command Line Interface (CLI) to manage your AWS AppSync resources. For information about how to install and configure the AWS CLI, see Getting started with the AWS CLI in the *AWS Command Line Interface User Guide*.

Creating an AWS AppSync Event API

AWS AppSync Events allows you to create Event APIs to enable real-time capabilities in your applications. In this section, you create an API with a default channel namespace. You'll then use the AWS AppSync console to publish messages and subscribe to messages sent to channels in the namespace.

In this tutorial you will complete the following tasks.

Topics

- Step 1: Create an event API using the AWS AppSync console

- Step 2: Publish and subscribe to receive your first event
- Step 3: Use wildcards in your channel subscription
- Step 4: Publish in batches

Step 1: Create an event API using the AWS AppSync console

1. Sign in to the AWS Management Console and open the AWS AppSync console.
2. On the AWS AppSync console service page, choose **Create API**, then choose **Event API**.
3. On the **Create Event API** page, in the **API details** section, do the following:
   a. For **API** enter the name of your API.
   b. (optional) Enter the contact name for the API.
4. Choose **Create**.

You have now created an Event API. The API is configured with API Key as an authorization mode for connect, publish, and subscribe actions. A default channel namespace with the name “default” has also been created.

To learn more about customizing authorization, see Configuring authorization and authentication to secure Event APIs. To learn more about channel namespaces, see Understanding channel namespaces

Step 2: Publish and subscribe to receive your first event

Use the following instructions to publish an event.

1. In the AWS AppSync console choose the **Pub/Sub Editor** tab for the Event API that you created in step 1.
2. In the **Publish** section, for **Channel** enter **`default`** in the first text box and enter **`/messages`** in the second text box.
3. In the code editor, enter the following JSON payload.

```
[{"message": "Hello world!"}]
```

4. Choose **Publish**.
5. The publisher logs table displays a response similar to the following to confirm success.

```
{
  "failed": [],
  "successful": [
    {
      "identifier": "53287bee-ae0d-42e7-8a90-e9d2a49e4bd7",
      "index": 0
    }
  ]
}
```

Now use the following instructions to subscribe to recieve messages.

1. In the **Subscribe** section of the editor, choose **Connect** to connect to your WebSocket endpoint.
2. For **Channel**, enter the name of the channel you want to subscribe to, **`/default/messages`**, and then choose **Subscribe**.
3. In the code editor, choose **Publish** again. You should receive a new data event in the subscriber logs table with the published event.

Step 3: Use wildcards in your channel subscription

You can specify a wildcard "*" at the end of a channel path to receive events published to all channels that match. Use the following instructions to set up a wildcard channel subscription for your Event API.

1. If you are still subscribed to the channel from step 2, choose **Unsubscribe**.
2. In the **Subscribe** section, for **Channel** enter **`/default/*`**.
3. In the code editor section, choose **Publish** again to send another event. You should receive a new data event in the subscriber logs table
4. In the **Publish** section, change the **Channel** name to **`/default/greetings/tutorial`**.
5. Choose **Publish**. You receive the message in the **Subscribe** section

Step 4: Publish in batches

You can publish events in batches of up to five. Subscribed clients receive each message individually.

1. In the AWS AppSync console, continue in the **Pub/Sub Editor** tab using the Event API that you were working with in step 3.
2. In the **Publish** section JSON code editor, enter the following:

```
[
  {"message": "Hello world!"},
  {"message": "Bonjour lemonde!"},
  "Hola Mundo!"
]
```

3. Choose **Publish**.
4. In the **Subscribe** log table, you receive 3 data events.

Getting started with the Amplify Events client

You can connect to your AWS AppSync Event API using any HTTP and WebSocket client, and you can also use the Amplify client for JavaScript. This getting started tutorial guides you through connecting to an Event API from a JavaScript React application.

In this tutorial you will complete the following tasks.

Topics

- Step 1: Create an event API
- Step 2: Deploy a React app with Vite
- Step 3: Configure the Amplify client
- Step 4: Connect to a channel and receive messages
- Step 5: Send a message from your app

Step 1: Create an event API

1. Sign in to the AWS Management Console and open the AWS AppSync console.
2. On the AWS AppSync console service page, choose **Create API**, then choose **Event API**.

3. On the **Create Event API** page, in the **API details** section, do the following:
   a. For **API** enter the name of your API.
   b. (optional) Enter the contact name for the API.
4. Choose **Create**.

You have now created an Event API. The API is configured with API Key as an authorization mode for connect, publish, and subscribe actions. A default channel namespace with the name “default” has also been created.

To learn more about customizing authorization, see Configuring authorization and authentication to secure Event APIs. To learn more about channel namespaces, see Understanding channel namespaces

Step 2: Deploy a React app with Vite

1. From your local work environment, run the following command in a terminal window to create a new Vite app for React.

```
npm create vite@latest appsync-events-app -- --template react
```

2. Run the following command to switch to the `appsync-events-app` directory and install the dependencies and the Amplify library.

```
cd appsync-events-app
npm install
npm install aws-amplify
```

3. Open a different terminal window and `cd` into your `appsync-events-app` directory. Run the following command to start your sever in dev mode.

```
npm run dev
```

(Optional) Configure Tailwind CSS

You can set up Tailwind CSS to style your project.

1. Open a terminal window and run the following commands to install the dependencies.

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

2. Update the `tailwind.config.js` file with the following code.

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

3. Set the content of `./src/index.css` to the following.

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

4. Use the following command to start and restart your Vite app from the terminal where it is currently running.

```
npm run dev
```

Step 3: Configure the Amplify client

1. Sign in to the AWS Management Console and open the AWS AppSync console.
2. Open the **Integration** tab for the Event API that you created in step 1.
3. Download your configuration file.
4. Save the `amplify_outputs.json` file in your project's `src` directory. Your configuration file will look like the following.

```
{
  "API": {
```

```
    "Events": {
      "endpoint": "https://abc1234567890.aws-appsync.us-west-2.amazonaws.com/
event",
      "region": "us-west-2",
      "defaultAuthMode": "apiKey",
      "apiKey": "da2-your-api-key-1234567890"
    }
  }
}
```

Important

You must set `defaultAuthMode` to `apiKey` and *not* `API_KEY`.

Step 4: Connect to a channel and receive messages

1. Update your `App.jsx` file with the following code.

```
import { useEffect, useState, useRef } from 'react'
import './App.css'

import { Amplify } from 'aws-amplify'
import { events } from 'aws-amplify/data'
import config from './amplify_outputs.json'

Amplify.configure(config)

export default function App() {
  const [messages, setMessages] = useState([])
  const [room, setRoom] = useState('')
  const counterRef = useRef(null)

  useEffect(() => {
    if (!room || !room.length) {
      return
    }
    let timeoutID
    const pr = events.connect(`/default/${room}`)
    pr.then((channel) => {
      channel.subscribe({
        next: (data) => {
```

```
          setMessages((messages) => [...messages, data.message])
          if (timeoutID) {
            clearTimeout(timeoutID);
          }
          counterRef.current?.classList.add('animate-bounce')
          timeoutID = setTimeout(() => {
            counterRef.current?.classList.remove('animate-bounce')
          }, 3000);
        },
        error: (value) => console.error(value),
      })
    })

    return () => {
      pr?.then((channel) => channel?.close())
    }
  }, [room])

  return (
    <div className='max-w-screen-md mx-auto'>
      <h2 className='my-4 p-4 font-semibold text-xl'>AppSync Events - Messages</h2>
      <button
        type="button"
        className='border rounded-md px-4 py-2 items-center text-sm font-medium
 transition-colors focus-visible:outline-none focus-visible:ring-1 focus-
visible:ring-ring disabled:pointer-events-none disabled:opacity-50 bg-sky-200
 shadow hover:bg-sky-200/90'
        onClick={() => {
          const room = prompt('Room:')
          if (room && room.length) {
            setMessages([])
            setRoom(room.trim().replace(/\W+/g, '-'))
          }
        }}
      >
        set room
      </button>
      <div className='my-4 border-b-2  border-sky-500 py-1 flex justify-between'>
        <div>
          {room ? (
            <span>
              Currently in room: <b>{room}</b>
            </span>
          ) : (
```

```
            <span>Pick a room to get started</span>
          )}
        </div>
        <div className='flex items-center uppercase text-xs tracking-wider font-
semibold'>
          <div className='mr-2'>Messages count:</div>
          <span ref={counterRef} className='transition-all inline-flex
 items-center rounded-md bg-sky-100 px-2.5 py-0.5 text-xs font-medium text-
sky-900'>{messages.length}</span></div>
      </div>
      <section id="messages" className='space-y-2'>
        {messages.map((message, index) => (
          <div
            key={index}
            className='border-b py-1 flex justify-between px-2'
          ><div>
              {message}
            </div>
            <div> </div>
          </div>
        ))}
      </section>
    </div>
  )
}
```

2. Open the app in your browser and choose the **set room** button to set the room to "greetings".
3. Open the AWS AppSync console, and go to the **Pub/Sub Editor** tab for your API.
4. Set your channel to `/default/greetings`. Choose the "default" namespace and set the rest of the path to `/greetings`.
5. Paste the following into the editor to send the event.

```
[
    {
        "message": "hello world!"
    }
]
```

6. You should see the message in your app.
7. Choose the **set room** button again to select another room. Send another event in the Pub/Sub Editor to the `/default/greetings` channel. You do not see that message in your app.

Step 5: Send a message from your app

1. Open your App component and add the following line of code at the top of the file.

```
const [message, setMessage] = useState('')
```

2. In the same file, locate the last `section` element and add the following code.

```
<section>
        <form
          disabled={!room}
          className='w-full flex justify-between mt-8'
          onSubmit={async (e) => {
            e.preventDefault()
            const event = { message }
            setMessage('')
            await events.post(`/default/${room}`, event)
          }}
        >
          <input
            type="text"
            name="message"
            placeholder="Message:"
            className='flex flex-1 rounded-md border border-input px-3
 py-1 h-9 text-sm shadow-sm transition-colors focus-visible:outline-none
 focus-visible:ring-1 focus-visible:ring-ring disabled:cursor-not-allowed
 disabled:opacity-50 bg-transparent'
            value={message}
            disabled={!room}
            onChange={(e) => setMessage(e.target.value)}
          />
          <button
            type="submit"
            className='ml-4 border rounded-md px-4 flex items-center text-sm font-
medium transition-colors focus-visible:outline-none focus-visible:ring-1 focus-
visible:ring-ring disabled:pointer-events-none disabled:opacity-50 bg-sky-200
 shadow hover:bg-sky-200/90'
            disabled={!room || !message || !message.length}>
            <svg xmlns="http://www.w3.org/2000/svg" className='size-4' width="24"
 height="24" viewBox="0 0 24 24" fill="none" stroke="currentColor" strokeWidth="2"
 strokeLinecap="round" strokeLinejoin="round"><path d="M14.536 21.686a.5.5 0 0
 0 .937-.024l6.5-19a.496.496 0 0 0-.635-.635l-19 6.5a.5.5 0 0 0-.024.937l7.93
 3.18a2 2 0 0 1 1.112 1.11z" /><path d="m21.854 2.147-10.94 10.939" /></svg>
```

```
          </button>
        </form>
      </section>
```

3. You can now send a message to a room that you select, directly from your app.

Understanding channel namespaces

Channel namespaces (or just namespaces for short) define the channels that are available on your Event API, and the capabilities and behaviors of these channels. Channel namespaces provide a scalable approach to managing large numbers of channels. Instead of configuring each channel individually, developers can apply settings across an entire namespace.

Each namespace has a name. This name represents the starting segment (the prefix) of a channel path. Any channel where the first segment in the path matches the name of a namespace, belongs to that namespace. For example, any channel with a first segment of `default`, such as `/default/messages`, `/default/greetings`, and `/default/inbox/user` belongs to the namespace named `default`. A channel namespace is made up of a maximum of five segments.

In a sports-related Event API example, you could have namespaces such as `basketball`, `soccer`, and `baseball`, with channels within each namespace such as `basketball/games/1`, `soccer/scores`, and `baseball/players`.

You can only publish and subscribe to channels that belong to a defined namespace. However, a channel is ephemeral and is created on-demand when a client needs to publish or subscribe to it.

Channel namespace authorization

When you define your Event API, you must configure the default authorization for connecting, publishing and subscribing to an Event API. The publishing and subscribing authorization configuration is automatically applied to all namespaces. You can override this configuration at the namespace. To learn more about authorization, see Configuring authorization and authentication to secure Event APIs.

Channel namespace handlers and event processing

You can define event handlers on channel namespaces. Event handlers are functions that run on AWS AppSync's JavaScript runtime and enable you to run custom business logic. You can use an event handler to process published events or process and authorize subscribe requests.

Using the onPublish event handler

The onPublish handler runs when events are received on a channel. The handler is called with the list of events. You can use the handler to either filter events or transform the events before they are sent to subscribed clients.

The default API behavior for the onPublish handler simply forwards all received events to be broadcast. The onPublish function receives a context object as its first argument.

The following code demonstrates the default behavior for the onPublish handler.

```
export function onPublish(ctx) {
  return ctx.events
}
```

The Context object holds a list of events that are being handled. The type of each event is the following.

```
type IncomingEvent<T> = {
  /**
   * The ID associated with the event.
   */
  id: String;
  /**
   * The payload associated with the event.
   */
  payload: T;
};
```

Processing events

The onPublish handler can act as an interceptor that can modify the list of events before they are broadcast. The following processing rules apply.

- Your function must return an array.
- Each event must match the Event type shape, and must have an ID that matches the ID of an incoming event.
- Duplicate event IDs are not allowed.
- Null objects in the returned array are dropped.

The type of outgoing events is the following.

```
export type OutgoingEvent<T extends any = any> = {
  /**
   * The ID associated with the event.
   */
  id: String;
  /**
   * The payload associated with the event. Not required if an error is defined.
   */
  payload?: T;
  /**
   * The error associated with the event. Populating this field with a value will
   * result in this error being marked as a failure and not propogated through the
   * system.
   */
  error?: String;
}
```

Filtering

You can filter out events, and only return the list of events you want subscribers to receive. For example, the handler below filters events and only forwards those that have “odds” greater than zero.

```
export function onPublish(ctx) {
  return ctx.events.filter((event) => event.payload.odds > 0)
}
```

Transforming

You can also use a handler to transform events. Do this by mapping your events to the shape that you want. For example, the handler below formats each event to include a timestamp and changes the format of the message property.

```
import { util } from '@aws-appsync/utils'
export function onPublish(ctx) {
  return ctx.events.map(event => ({
    id: event.id,
    payload: {
      ...event.payload,
```

```
      message: event.payload.message.toUpperCase()
      timestamp: util.time.nowISO8601()
    }
  }))
}
```

> **Note**
>
> Returning an event with an unknown id value results in an error.

Returning an error

To explicitly reject an event and notify the publisher that it was not accepted, add an error property containing an error message to the processed event. The following example, rejects any event that does not include a message property.

```
export function onPublish(ctx) {
  return ctx.events.map(event => {
    if (!event.payload.message || event.payload.message.length === 0) {
      event.error = "A message must be provided"
    }
    return event
  })
}
```

Using the onSubscribe event handler

The onSubscribe handler is called each time a client attempts to subscribe to a channel. You can use this handler to run custom business logic, such as logging specific information, or applying additional static authorization checks.

In the following example, the onSubscribe handler logs a message whan an admin user subscribes to a channel.

```
export function onSubscribe(ctx) {
  if (ctx.identity.groups.includes('admin')) {
    console.log(`Admin ${ctx.identity.sub} subscribed to ${ctx.channel}`)
  }
}
```

You can reject a subscription by calling `util.unauthorized()`. The following onSubscribe handler restricts Amazon Cognito User Pool authenticated users to their own channel:

```
export function onSubscribe(ctx: Context) {
  if (ctx.info.channel.path !== `/messages/inbox/${ctx.identity.username}`) {
    console.log(`user ${ctx.identity.username} tried connecting to wrong channel:
 ${ctx.channel}`)
    util.unauthorized()
  }
```

Now clients can only subscribe to channels that match the pattern `/messages/inbox/[username]`.

Configuring authorization and authentication to secure Event APIs

AWS AppSync Events offers the following authorization types to secure Event APIs: API keys, Lambda, IAM, OpenID Connect, and Amazon Cognito user pools. Each option provides a different method of security.

1. **API Key authorization**: A simple key-based security option, with keys generated by the AppSync service.
2. **Lambda authorization**: Enables custom authorization logic, evaluated by an Lambda function .
3. **IAM authorization**: Utilizes AWS's signature version 4 signing process, allowing fine-grained access control through IAM policies.
4. **OpenID Connect authorization**: Integrates with OIDC-compliant services for user authentication.
5. **Amazon Cognito user pools**: Implements group-based access control using Amazon Cognitos user management features.

Authorization types

There are five ways you can authorize applications to interact with your AWS AppSync Event API. You specify which authorization type you use by specifying one of the following authorization type values in your AWS AppSync API or AWS CLI call:

- **`API_KEY`**

  For using API keys.
- **`AWS_LAMBDA`**

  For using an AWS Lambda function.
- **`AWS_IAM`**

  For using AWS Identity and Access Management permissions.
- **`OPENID_CONNECT`**

  For using your OpenID Connect provider.

- **`AMAZON_COGNITO_USER_POOLS`**

  For using an Amazon Cognito user pool.

When you define your API, you configure the authorization mode to connect to your Event API WebSocket. You also configure the default authorization modes to use when publishing and subscribing to messages. You can use different authorization modes for each configuration. For example, you might want your publisher to use IAM authorization for a backend process running on an Amazon EC2 instance, but you want your clients to use an API key to subscribe to messages.

Optionally, you can also configure the authorization mode to use for publishing and subscribing to messages on a namespace. When defined, these settings override the default configuration on your API. This enables you to have different settings for different namespaces on your API.

To learn more about using authorization types in your WebSocket operations, see Understanding the Event API WebSocket protocol

To learn more about publishing events using the HTTP endpoint, see Publish via HTTP.

API_KEY authorization

API Keys allow unauthenticated clients to securely use your API. An API key is a hard-coded value in your application that is generated by the AWS AppSync service. You can rotate API keys from the AWS Management Console, the AWS CLI, or from the AWS AppSync API Reference.

API keys are configurable for up to 365 days, and you can extend an existing expiration date for up to another 365 days from that day.

On the client, the API key is specified by the header `x-api-key`. For example, if your `API_KEY` is `ABC123`, you can publish a message to a channel using the HTTP endpoint via curl as follows:

```
curl --location "https://YOUR_EVENT_API_ENDPOINT/event" \
--header 'Content-Type: application/json' \
--header "x-api-key:ABC123" \
--data '{
    "channel":"/news",
    "events":["\"Breaking news!\""]
}'
```

AWS_LAMBDA authorization

You can implement your own API authorization logic using an AWS Lambda function. When you use Lambda functions for authorization, the following constraint applies.

- A Lambda function authorizer must not return more than 5MB of contextual data.

For example, if your authorization token is `'ABC123'`, you can publish via curl as follows:

```
curl --location "https://YOUR_EVENT_API_ENDPOINT/event" \
--header 'Content-Type: application/json' \
--header "Authorization:ABC123" \
--data '{
    "channel":"/news",
    "events":["\"Breaking news!\""]
}'
```

Lambda functions are called before connection, publish, and subscription attempts. The return value can be cached based on the API ID and the authentication token. By default, caching is not turned on, but this can be enabled at the API level or by setting the `ttlOverride` value in a function's return value.

You can also specify a regular expression that validates authorization tokens before the function is called. These regular expressions are used to validate that an authorization token is of the correct format before your function is called. Any request using a token which does not match this regular expression will be denied automatically.

Lambda functions used for authorization require a principal policy for `appsync.amazonaws.com` to be applied on them to allow AWS AppSync to call them. This action is done automatically in the AWS AppSync console; The AWS AppSync console does *not* remove the policy. For more information on attaching policies to Lambda functions, see Working with resource-based IAM policies in Lambda in the *AWS Lambda Developer Guide*.

The Lambda function you specify will receive an event with the following shape:

```
{
  "authorizationToken": "ExampleAUTHtoken123123123",
  "requestContext": {
    "apiId": "aaaaaa123123123example123",
```

```
    "accountId": "111122223333",
    "requestId": "f4081827-1111-4444-5555-5cf4695f339f",
    "operation": "EVENT_CONNECT",
    "channelNamespaceName": "news",
    "channel": "/news/latest"
  },
  "requestHeaders": {
    "header": "value"
  }
}
```

The operation property indicates the operation that is being evaluated and can have the following values:

- EVENT_CONNECT
- EVENT_SUBSCRIBE
- EVENT_PUBLISH

The event object contains the headers that were sent in the request from the application client to AWS AppSync.

The authorization function must return at least `isAuthorized`, a boolean indicating whether the request is authorized to execute the operation on the Event API.

If this value is true, execution of the Event API continues. If this value is false, an `UnauthorizedException` is raised.

Accepted keys

`isAuthorized` (boolean, required)

A boolean value indicating if the value in `authorizationToken` is authorized to execute the operation on the Event API.

If this value is true, execution of the Event API continues. If this value is false, an `UnauthorizedException` is raised

`handlerContext` (JSON object, optional)

A JSON object visible as `$ctx.identity.handlerContext` in your handlers. The object is a map of strings. For example, if the following structure is returned by a Lambda authorizer:

```
{
  "isAuthorized":true
  "handlerContext": {
    "banana":"very yellow",
    "apple":"very green"
  }
}
```

The value of `ctx.identity.handlerContext.apple` in handlers will be `very green`. The `handlerContext` object only supports key-value pairs. Nested keys are not supported.

> **Warning**
>
> The total size of this JSON object must not exceed 5MB.

`ttlOverride` (integer, optional)

The number of seconds that the response should be cached for. If no value is returned, the value from the API is used. If this is 0, the response is not cached.

Lambda authorizers have a timeout of 10 seconds. We recommend designing functions to execute in the shortest amount of time possible to scale the performance of your API.

Multiple AWS AppSync APIs can share a single authentication Lambda function. Cross-account authorizer use is not supported.

The following example describes a Lambda function that demonstrates the various authentication and failure states that a Lambda function can have when used as an AWS AppSync authorization mechanism.

```
def handler(event, context):
  # This is the authorization token passed by the client
  token = event.get('authorizationToken')
  # If a lambda authorizer throws an exception, it will be treated as unauthorized.
  if 'Fail' in token:
    raise Exception('Purposefully thrown exception in Lambda Authorizer.')

  if 'Authorized' in token and 'ReturnContext' in token:
```

```
    return {
      'isAuthorized': True,
      'handlerContext': {
        'key': 'value'
      }
    }

  # Authorized with no context
  if 'Authorized' in token:
    return {
      'isAuthorized': True
    }

  # never cache response
  if 'NeverCache' in token:
    return {
      'isAuthorized': True,
      'ttlOverride': 0
    }


  # not authorized
  if 'Unauthorized' in token:
    return {
      'isAuthorized': False
    }

  # if nothing is returned, then the authorization fails.
  return {}
```

AWS_IAM authorization

This authorization type enforces the AWS signature version 4 signing process on your API. You can associate AWS Identity and Access Management (IAM) access policies with this authorization type. Your application can leverage this association by using an access key (which consists of an access key ID and secret access key) or by using short-lived, temporary credentials provided by Amazon Cognito Federated Identities.

Use the following example if you want an IAM role that has permission to perform all data operations on an API.

```
{
```

```
"Version": "2012-10-17",
   "Statement": [
      {
"Effect": "Allow",
         "Action": [
            "appsync:connect",
            "appsync:publish",
            "appsync:subscribe",
         ],
         "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/{APIID}/*"
         ]
      }
   ]
}
```

Use the following example to restrict access to a specific API and a specific channel namespace.

```
{
"Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "appsync:connect",
         ],
         "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/{APIID}"
         ]
      },
      {
         "Effect": "Allow",
         "Action": [
            "appsync:publish",
            "appsync:subscribe"
         ],
         "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/{APIID}/channelNamespace/
{NAME}"
         ]
      }
   ]
}
```

OPENID_CONNECT authorization

This authorization type enforces OpenID connect (OIDC) tokens provided by an OIDC-compliant service. Your application can leverage users and privileges defined by your OIDC provider for controlling access.

An Issuer URL is the only required configuration value that you provide to AWS AppSync (for example, `https://auth.example.com`). This URL must be addressable over HTTPS. AWS AppSync appends `/.well-known/openid-configuration` to the issuer URL and locates the OpenID configuration at `https://auth.example.com/.well-known/openid-configuration` per the OpenID Connect Discovery specification. It expects to retrieve an RFC5785 compliant JSON document at this URL. This JSON document must contain a `jwks_uri` key, which points to the JSON Web Key Set (JWKS) document with the signing keys. AWS AppSync requires the JWKS to contain JSON fields of `kty` and `kid`.

AWS AppSync supports a wide range of signing algorithms.

Signing algorithms
RS256
RS384
RS512
PS256
PS384
PS512
HS256
HS384
HS512
ES256
ES384

Signing algorithms
ES512

We recommend that you use the RSA algorithms. Tokens issued by the provider must include the time at which the token was issued (`iat`) and may include the time at which it was authenticated (`auth_time`). You can provide TTL values for issued time (`iatTTL`) and authentication time (`authTTL`) in your OpenID Connect configuration for additional validation. If your provider authorizes multiple applications, you can also provide a regular expression (`clientId`) that is used to authorize by client ID. When the `clientId` is present in your OpenID Connect configuration, AWS AppSync validates the claim by requiring the `clientId` to match with either the `aud` or `azp` claim in the token.

To validate multiple client IDs use the pipeline operator (“|”) which is an “or” in regular expression. For example, if your OIDC application has four clients with client IDs such as 0A1S2D, 1F4G9H, 1J6L4B, 6GS5MG, to validate only the first three client IDs, you would place 1F4G9H|1J6L4B|6GS5MG in the client ID field.

AMAZON_COGNITO_USER_POOLS authorization

This authorization type enforces OIDC tokens provided by Amazon Cognito user pools. Your application can leverage the users and groups in your handlers to apply custom business rules.

When using Amazon Cognito user pools, you can create groups that users belong to. This information is encoded in a JWT token that your application sends to AWS AppSync in an authorization header with each request.

```
curl --location "https://YOUR_EVENT_API_ENDPOINT/event" \
--header 'Content-Type: application/json' \
--header "Authorization:JWT_TOKEN" \
--data '{
    "channel":"/news",
    "events":["\"Breaking news!\""]
}'
```

Circumventing SigV4 and OIDC token authorization limitations

The following methods can be used to circumvent the issue of not being able to use your SigV4 signature or OIDC token as your Lambda authorization token when certain authorization modes are enabled.

If you want to use the SigV4 signature as the Lambda authorization token when the `AWS_IAM` and `AWS_LAMBDA` authorization modes are enabled for AWS AppSync's API, do the following:

- To create a new Lambda authorization token, add random suffixes and/or prefixes to the SigV4 signature.
- To retrieve the original SigV4 signature, update your Lambda function by removing the random prefixes and/or suffixes from the Lambda authorization token. Then, use the original SigV4 signature for authentication.

If you want to use the OIDC token as the Lambda authorization token when the `OPENID_CONNECT` authorization mode or the `AMAZON_COGNITO_USER_POOLS` and `AWS_LAMBDA` authorization modes are enabled for AWS AppSync's API, do the following:

- To create a new Lambda authorization token, add random suffixes and/or prefixes to the OIDC token. The Lambda authorization token should not contain a Bearer scheme prefix.
- To retrieve the original OIDC token, update your Lambda function by removing the random prefixes and/or suffixes from the Lambda authorization token. Then, use the original OIDC token for authentication.

Publish via HTTP

AWS AppSync Events allows you to publish events via your API's HTTP endpoint using a POST operation. Publishing is the only supported action over the endpoint.

Publish steps

1. Send a POST request to the address: https://HTTP_DOMAIN/event.
2. Add the authorization header(s) required to authorize your request.
3. Specify the following in the request body:
   - The channel that you are publishing to.
   - The list of events you are publishing. You can publish up to 5 events in a batch.

Each specified event in your publish request must be a stringified valid JSON value.

Publish example

The following is an example of a request.

```
{
  "method": "POST",
  "headers": {
    "content-type": "application/json",
    "x-api-key": "da2-your-api-key"
  },
  "body": {
    "channel": "default/channel",
    "events": [
      "{\"event_1\":\"data_1\"}",
      "{\"event_2\":\"data_2\"}"
    ]
  }
}
```

You can use your Browser's `fetch API` to publish the events. The following example demonstrates this.

```
await fetch(`https://${HTTP_DOMAIN}/event`, {
  "method": "POST",
```

```
  "headers": {
    "content-type": "application/json",
    "x-api-key": "da2-your-api-key"
  },
  "body": {
    "channel": "default/channel",
    "events": [
      "{\"event_1\":\"data_1\"}",
      "{\"event_2\":\"data_2\"}"
    ]
  }
})
```

To learn more about the different authorization types that AWS AppSync Events supports, see Configuring authorization and authentication to secure Event APIs.

Understanding the Event API WebSocket protocol

AWS AppSync Events' WebSocket API allows a client to subscribe and receive events in real-time. Establishing a valid connection and subscribing to receive events is a simple multi-step process.

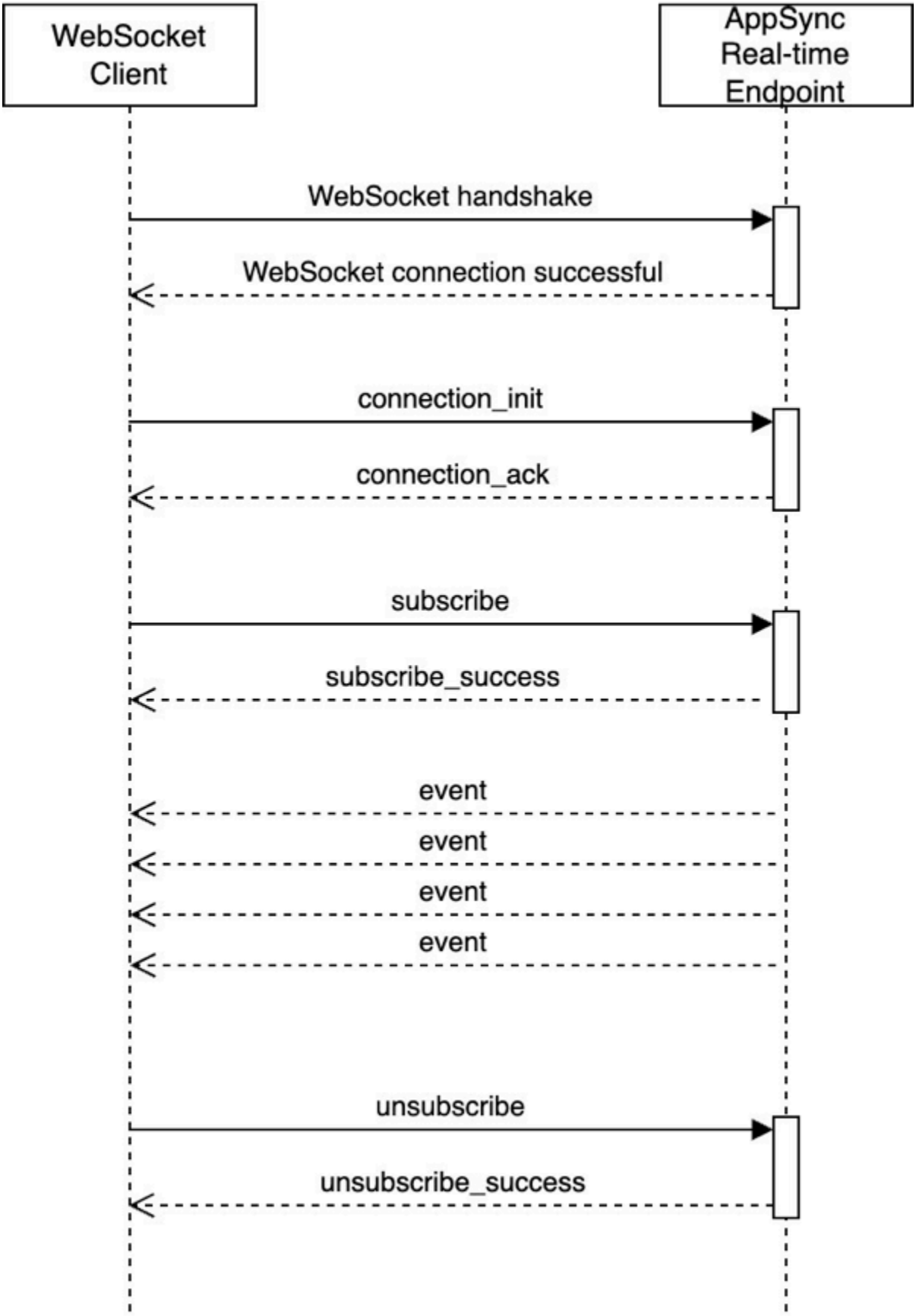
First, a client establishes a WebSocket connection with the AWS real-time endpoint, sends a connection initialization message, and waits for acknowledgment.

After a successful connection is established, the client registers subscriptions by sending a “subscribe” message with a unique ID and a channel path of interest. AWS AppSync confirms successful subscriptions with acknowledgment messages. The client then listens for subscription events, which are triggered when a publisher publishes events that are broadcast by the service. To maintain the connection, AWS AppSync sends periodic keep-alive messages.

When finished, the client unsubscribes by sending “unsubscribe” messages. This system supports multiple subscriptions on a single WebSocket connection and accommodates various authorization modes, including API keys, Amazon Cognito user pools, IAM, and Lambda.

The following diagram demonstrates the WebSocket protocol message flow between the WebSocket client and the real-time endpoint.

WebSocket protocol overview


WebSocket Client
AppSync Real-time Endpoint
WebSocket handshake
WebSocket connection successful
connection_init
connection_ack
subscribe
subscribe_success
event
event
event
event
unsubscribe
unsubscribe_success

In the preceeding diagram, the following WebSocket steps occur in the message flow.

- A client establishes a WebSocket connection with the AWS AppSync real-time endpoint. If there is a network error, the client should do a jittered exponential backoff. For more information, see Exponential backoff and jitter on the *AWS Architecture Blog*.

- After successfully establishing the WebSocket connection, the client sends a `connection_init` message.

- The client waits for a `connection_ack`connection_ack message from AWS AppSync. This message includes a `connectionTimeoutMs` parameter, which is the maximum wait time in milliseconds for a "ka" (keep-alive) message.

- AWS AppSync sends "ka" messages periodically. The client keeps track of the time that it received each "ka" message. If the client doesn't receive a "ka" message within `connectionTimeoutMs` milliseconds, the client should close the connection.

- The client registers the subscription by sending a subscribe message. A single WebSocket connection supports multiple subscriptions, even if they are in different authorization modes.

- The client waits for AWS AppSync to send `subscribe_success` messages to confirm successful subscriptions.

- The client listens for subscription events, which are sent after events are published to the channel of interest.

- The client unregisters the subscription by sending an `unsubscribe` subscription message.

- After unregistering all subscriptions and checking that there are no messages transferring through the WebSocket, the client can disconnect from the WebSocket connection.

Handshake details to establish the WebSocket connection

All interactions with the AWS AppSync real-time endpoint begin with establishing a WebSocket connection. The connection remains open as long as the client remains connected, up to a maximum of 24 hours. Connecting is an operation that requires authorization credentials to complete the handshake. To connect and initiate a successful handshake with AWS AppSync, a WebSocket client needs the following information:

- The AWS AppSync Events realtime and HTTP endpoints
- The authorization details

To authorize your WebSocket connection establishment, send the authorization information as a WebSocket subprotocol. To do this, a client must wrap the appropriate authorization credentials in a JSON object, encode the object in Base64URL format, and append the encoded header string in the list of subprotocols.

The following JavaScript example converts an authorization object into a base64URL encoded string.

```
/**
 * Encodes an object into Base64 URL format
 * @param {*} authorization - an object with the required authorization properties
 **/
function getBase64URLEncoded(authorization) {
  return btoa(JSON.stringify(authorization))
    .replace(/\+/g, '-') // Convert '+' to '-'
    .replace(/\//g, '_') // Convert '/' to '_'
    .replace(/=+$/, '') // Remove padding `=`
}
```

Next, this example creates the required subprotocol value.

```
function getAuthProtocol(authorization) {
  const header = getBase64URLEncoded(authorization)
  return `header-${header}`
}
```

The following example uses bash to create a header, and then uses wscat to connect. You must specify `aws-appsync-event-ws` as one of the subprotocols.

```
$ REALTIME_DOMAIN='example1234567890000.appsync-realtime-api.us-west-2.amazonaws.com'
$ HTTP_DOMAIN='example1234567890000.appsync-api.us-east-1.amazonaws.com'
$ API_KEY='da2-12345678901234567890123456'

$ header="{\"host\":\"$HTTP_DOMAIN\", \"x-api-key\":\"$API_KEY\"}"
$ header=`echo "$header" | base64 | tr '+/' '-_' | tr -d '\n='`
$ wscat -p 13 -s "header-$header" -s "aws-appsync-event-ws" -c "wss://$REALTIME_DOMAIN/
event/realtime"

Connected (press CTRL+C to quit)
```

Discovering the real-time endpoint from the Event API endpoint

AWS AppSync Event APIs are configured with two endpoints: a realtime endpoint and an HTTP endpoint. You can retrieve your endpoint information by visiting your API's **Settings** page in the AWS Management Console or by running the AWS CLI command `aws appsync get-api`.

AWS AppSync Events HTTP endpoint

https://example1234567890000.appsync-api.us-east-1.amazonaws.com/event

AWS AppSync Events real-time endpoint

wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/event/realtime

Applications can connect to the HTTP endpoint (https://) using any HTTP client, and can connect to the real-time endpoint (wss://) using any WebSocket client.

With custom domain names, you can interact with both endpoints using a single domain. For example, if you configure api.example.com as your custom domain, you can interact with your HTTP and real-time endpoints using the following URLs.

AWS AppSync Events HTTP endpoint

https://api.example.com/event

AWS AppSync Events real-time endpoint

wss://api.example.com/event/realtime

Authorization formatting based on the AWS AppSync API authorization mode

The format of the authorization subprotocol varies depending on the AWS AppSync authorization mode. AWS AppSync supports API key, Amazon Cognito user pools, OpenID Connect (OIDC), AWS Lambda, and IAM authorization modes. The host field in the object refers to the AWS AppSync Events HTTP endpoint, which is used to validate the connection even if the wss:// call is made against the real-time endpoint.

Use the following sections to learn how to format the authorization subprotocol for the supported authorization modes.

API key subprotocol format

Header content

- `"host": <string>`: The host for the AWS AppSync Events HTTP endpoint or your custom domain name.
- `"x-api-key": <string>`: The API key configured for the AWS AppSync Event API.

Example

```
{
  "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com",
  "x-api-key":"da2-12345678901234567890123456"
}
```

Amazon Cognito user pools and OpenID Connect (OIDC) subprotocol format

Header content

- `"host": <string>`: The host for the AWS AppSync Events HTTP endpoint or your custom domain name.
- `"Authorization": <string>`: A JWT ID token. The header can use a Bearer scheme.

Example

```
{

 "Authorization":"eyEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJHWEFLSXBieU5WNHhsQjEXAMPLEnM2WldvPSIsImFsZ
zEE2DJH7sHOl2zxYi7f-SmEGoh2AD8emxQRYajByz-rE4Jh0QOymN2Ys-ZIkMpVBTPgu-
TMWDyOHhDUmUj2OP82yeZ3wlZAtr_gM4LzjXUXmI_K2yGjuXfXTaa1mvQEBG0mQfVd7SfwXB-
jcv4RYVi6j25qgow9Ew52ufurPqaK-3WAKG32KpV8J4-Wejq8t0c-
yA7sb8EnB551b7TU93uKRiVVK3E55Nk5ADPoam_WYE45i3s5qVAP_-InW75NUoOCGTsS8YWMfb6ecHYJ-1j-
bzA27zaT9VjctXn9byNFZmEXAMPLExw",
  "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

AWS Lambda subprotocol format

Header content

- `"host": <string>`: The host for the AWS AppSync Events HTTP endpoint or your custom domain name.
- `"Authorization": <string>`: The value that is passed as `authorizationToken`.

Example

```
{

 "Authorization":"M0UzQzM1MkQtMkI0Ni00OTZCLUI1NkQtMUM0MTQ0QjVBRTczCkI1REEzRTIxLTk5NzItNDJENi1BQ
  "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

AWS Identity and Access Management (IAM) subprotocol format

Header content

- `"accept": "application/json, text/javascript"`: A constant string parameter.
- `"content-encoding": "amz-1.0"`: A constant string parameter.
- `"content-type": "application/json; charset=UTF-8"`: A constant string parameter.
- `"host": <string>`: This is the host for the AWS AppSync Events HTTP endpoint.
- `"x-amz-date": <string>`: The timestamp must be in UTC and in the following ISO 8601 format: YYYYMMDD'T'HHMMSS'Z'. For example, 20150830T123600Z is a valid timestamp.

Don't include milliseconds in the timestamp. For more information, see Elements of an AWS API request signature in the *IAM User Guide*.

- `"X-Amz-Security-Token": <string>`: The AWS session token, which is required when using temporary security credentials. For more information, see Use temporary credentials with AWS resources in the *IAM User Guide*.
- `"Authorization": <string>`: Signature Version 4 (SigV4) signing information for the AWS AppSync endpoint. For more information on the signing process, see Create a signed AWS API request in the *IAM User Guide*.

The SigV4 signing HTTP request includes a canonical URL, which is the AWS AppSync HTTP endpoint with `/event` appended. The service endpoint AWS Region is the same Region where you're using the AWS AppSync API, and the service name is 'appsync'.

The HTTP request to sign to connect is the following.

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/event",
  data: "{}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

The following is the request to sign when sending a subscribe message. The channel name is specified in the request.

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/event",
  body: "{\"channel\":\"/your/channel/*\"}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

Authorization header example

```
{
  "accept": "application/json, text/javascript",
  "content-encoding": "amz-1.0",
  "content-type": "application/json; charset=UTF-8",
  "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
  "x-amz-date": "20200401T001010Z",
  "X-Amz-Security-Token":
 "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXRoZWFEXAMPLEcwRQIgAh97Cljq7wOPL8KsxP3YtDuyc/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKEn0zyUhBEXAMPLEjj/////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUdAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtwR+9zF7NaMMMse07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZOzlZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnGlSNVx7SMcfovkFDqQamm
+88y1OwwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVYlj9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEfwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCkcCGFnwBNFLmfnbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpgqJsPKt582caFKArIFIvO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+tOLnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOKtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtKOOweDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA==",
  "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
 Signature=83EXAMPLEbcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}
```

Real-time WebSocket operations

After initiating a successful WebSocket handshake with AWS AppSync, the client must send a subsequent message to connect to AWS AppSync for different operations. The WebSocket API has the following properties.

WebSocket API message properties

id

The client provided ID of the operation. This property is required and is used to correlate response error and success messages. For subscriptions, this property must be unique for all subscriptions within a connection. The property is a string and is limited to a maximum of 128 alphanumeric + special character (_,+,-) characters. /^[a-zA-Z0-9-_+]{1,128}$/

type

The type of operation being performed. Supported client operations are subscribe, unsubscribe, publish. The property is a string and must be one of the message types defined in the next section, *Configuring message details*.

channel

The channel to subscribe to events. The property is a string made up of one to five segments separated by a slash. Each segment is limited to 50 alphanumeric + dash characters. The property is case sensitive. For example: `channelNamespaceName` or `channelNamespaceName/sub-segment-1/subSegment-2` `/^\/?[A-Za-z0-9](?:[A-Za-z0-9-]{0,48}[A-Za-z0-9])?(?:\/[A-Za-z0-9](?:[A-Za-z0-9-]{0,48}[A-Za-z0-9])?){0,4}\/?$/`

authorization

The authorization headers necessary to authorize the operation. For example, ApiKey will contain both `host` and `x-api-key` but for IAM this will contain `host`, `x-amz-date`, `x-amz-security-token`, and `authorization`.

Configuring message details

This section provides information about the syntax to use to configure the details for various message types.

Connection init message

After the client has established the WebSocket connection, the client sends an init message to initiate the connection session.

```
{ "type": "connection_init" }
```

Connection acknowledge message

AWS AppSync responds with an “ack” message that contains a connection timeout value. If the client doesn’t receive a keep-alive message within the connection timeout period, the client should close the connection. The connection timeout period is 5 minutes.

```
{
    "type": "connection_ack",
    "connectionTimeoutMs": 300000
}
```

Keep-alive message

AWS AppSync periodically sends a keep-alive message to the client to maintain the connection. If the client doesn’t receive a keep-alive message within the connection timeout period, the client should close the connection. The keep-alive interval is 60 seconds. Clients do not need to acknowledge these messages.

```
{ "type": "ka" }
```

Subscribe message

After receiving a `connection_ack` message, the client can send a subscription registration message to listen for events on a channel.

- “id” is the ID of the subscription. This ID must be unique per client connection otherwise AWS AppSync returns an error message indicating the subscription message is duplicated.
- “channel” is the channel to which the subscribed client is listening. Any messages published to this channel will be delivered to the subscribed client.
- “authorization” is an object containing the fields required for authorization. The authorization object follows the same rules as the headers for connecting to the WebSocket.

```
{
  "type": "subscribe",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "channel": "/namespaceA/subB/subC",
  "authorization": {
    "x-api-key": "da2-12345678901234567890123456",
    "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
   }
```

```
}
```

Subscription acknowledgment message

AWS AppSync acknowledges with a success message. “id” is the ID of the corresponding subscribe operation that succeeded.

```
{
  "type": "subscribe_success",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

In case of an error, AWS AppSync sends a `subscribe_error` response.

```
{
  "type": "subscribe_error",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "errors": [
    {
      "errorType": "SubscriptionProcessingError",
      "message": "There was an error processing the operation"
    }
  ]
}
```

Data message

When an event is published to a channel the client is subscribed to, the event is broadcast and delivered in a data message. “id” is the ID of the corresponding subscription for the channel to which the message was published.

```
{
  "type": "data",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "event": ["\"my event content\""]
}
```

In case of an error, such as a broadcasting error, an error can be received at the client:

```
{
```

```
  "type": "broadcast_error",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "errors": [
    {
      "errorType": "MessageProcessingError",
      "message": "There was an error processing the message"
    }
  ]
}
```

Unsubscribe message

When the client wants to stop listening to a subscribed channel, the client sends a message to unregister the subscription. “id” is the ID of the corresponding subscription to which the client wants to unregister.

```
{
  "type": "unsubscribe",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

AWS AppSync acknowledges with a success message. “id” is the ID of the corresponding subscribe operation that succeeded.

```
{
  "type": "unsubscribe_success",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

If an error occurs, an error message is sent back to the client

```
{
  "type": "unsubscribe_error",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "errors": [
    {
      "errorType": "UnknownOperationError",
      "message": "Unknown operation id ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
    }
  ]
}
```

Disconnecting the WebSocket

Before disconnecting the WebSocket, to avoid data loss, the client should have the necessary logic to check that no operation is currently in place through the WebSocket connection. All subscriptions should be unregistered before disconnecting from the WebSocket.

Configuring custom domain names for Event APIs

With AWS AppSync, you can use custom domain names to configure a single, memorable domain that works for your Event APIs.

When you configure an AWS AppSync Event API, two endpoints are provisioned: An HTTP endpoint and a real-time endpoint. These endpoints have the following format.

AWS AppSync Events HTTP endpoint

https://example1234567890000.appsync-api.us-east-1.amazonaws.com/event

AWS AppSync Events real-time endpoint

wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/event/realtime

With custom domain names, you can interact with both endpoints using a single domain. For example, if you configure api.example.com as your custom domain, you can interact with both your HTTP and real-time WebSocket endpoints using the following URLs.

AWS AppSync Events HTTP endpoint

https://api.example.com/event

AWS AppSync Events real-time endpoint

wss://api.example.com/event/realtime

> **Note**
>
> AWS AppSync APIs support only TLS 1.2 and TLS 1.3 for custom domain names.

Registering and configuring a domain name for an Event API

To set up custom domain names for your AWS AppSync APIs, you must have a registered internet domain name. You can register an internet domain using Amazon Route 53 domain registration or a third-party domain registrar of your choice. For more information about using Route 53, see What is Amazon Route 53 in the *Amazon Route 53 Developer Guide*.

An API's custom domain name can be the name of a subdomain or the root domain (also known as the "zone apex") of a registered internet domain. After you create a custom domain name in AWS AppSync, you must create or update your DNS provider's resource record to map to your API endpoint. Without this mapping, API requests bound for the custom domain name cannot reach AWS AppSync.

Creating a custom domain name in AWS AppSync

Creating a custom domain name for an AWS AppSync API sets up an Amazon CloudFront distribution. You must set up a DNS record to map the custom domain name to the CloudFront distribution domain name. This mapping is required to route API requests that are bound for the custom domain name in AWS AppSync through the mapped CloudFront distribution.

You must also provide a certificate for the custom domain name. To set up the custom domain name or to update its certificate, you must have permission to update CloudFront distributions and describe the AWS Certificate Manager (ACM) certificate that you plan to use. To grant these permissions, attach the following AWS Identity and Access Management (IAM) policy statement to an IAM user, group, or role in your account.

```
{
"Version": "2012-10-17",
  "Statement": [
    {
"Sid": "AllowUpdateDistributionForAppSyncCustomDomainName",
      "Effect": "Allow",
      "Action": ["cloudfront:updateDistribution"],
      "Resource": ["*"]
    },
    {
"Sid": "AllowDescribeCertificateForAppSyncCustomDomainName",
      "Effect": "Allow",
      "Action": "acm:DescribeCertificate",
      "Resource": "arn:aws:acm:Region:account-id:certificate/certificate_ID"
    }
  ]
}
```

AWS AppSync supports custom domain names by leveraging Server Name Indication (SNI) on the CloudFront distribution. For more information about using custom domain names on a CloudFront

distribution, including the required certificate format and the maximum certificate key length, see Using HTTPS with CloudFront in the *Amazon CloudFront Developer Guide*.

To set up a custom domain name as the API's hostname, the API owner must provide an SSL/TLS certificate for the custom domain name. To provide a certificate, do one of the following.

- Request a new certificate in ACM, or import a certificate issued by a third-party certificate authority into ACM in the US East (N. Virginia) (us-east-1) AWS Region. For more information about ACM, see What is AWS Certificate Manager in the *AWS Certificate Manager User Guide*.
- Provide an IAM server certificate. For more information, see Manage server certificates in IAM in the *IAM User Guide*.

Wildcard custom domain names in AWS AppSync

AWS AppSync supports wildcard custom domain names. To configure a wildcard custom domain name, specify a wildcard character (*) as the first subdomain of a custom domain. This represents all possible subdomains of the root domain. For example, the wildcard custom domain name `*.example.com` results in subdomains such as `a.example.com`, `b.example.com`, and `c.example.com`. All these subdomains route to the same domain.

To use a wildcard custom domain name in AWS AppSync, you must provide a certificate issued by ACM containing a wildcard name that can protect several sites in the same domain. For more information, see ACM certificate characteristics and limitations in the *AWS Certificate Manager User Guide*.

Using CloudWatch to monitor and log Event API data

You can log and debug your Event API using CloudWatch metrics and CloudWatch logs. These tools enable developers to monitor performance, troubleshoot issues, and optimize their AWS AppSync API operations effectively.

CloudWatch metrics is a tool that provides a wide range of metrics to monitor API performance and usage. These metrics fall into two main categories:

1. **HTTP API Metrics for Publish**: These include `4XXError` and `5XXError` for tracking client and server errors, `Latency` for measuring response times, `Requests` for monitoring total API calls, `TokensConsumed` for tracking resource usage, and *Events* related to metrics for tracking event publishing performance.
2. **Real-time Subscription Metrics**: These metrics focus on WebSocket connections and subscription activities. They include metrics for connection requests, successful connections, subscription registrations, message publishing, and active connections and subscriptions.

CloudWatch Logs is a tool that enables logging capabilities for your Eent APIs. Logs can be set at two levels of the API:

1. **Request-level Logs**: These capture overall request information, including HTTP headers, operation summaries, and subscription registrations.
2. **Handler-level Logs**: These provide detailed information about handler evaluation, including request and response mappings, and tracing information for each field.

You can configure logging, interpret log entries, and use log data for troubleshooting and optimization. AWS AppSync provides various log types that provide insight into your API's behavior.

Setting up and configuring logging on an Event API

Us the following instruction to turn on automatic logging on a Event API using the AWS AppSync console.

1. Sign in to the AWS Management Console and open the AppSync console.
2. On the **APIs** page, choose the name of an Event API.

3. On the API's homepage, in the navigation pane, choose **Settings**.
4. Under **Logging**, do the following:

   a. Turn on **Enable Logs**.

   b. (Optional) For **Log level**, choose your preferred field-level logging level (**None**, **Error**, or **All**).

   c. The procedure for adding a service role varies depending on whether you want to create a new role or use an existing one.

      - To create a new role:

        - For **Create or use an existing role**, choose **New role**. This creates a new IAM role that allows AWS AppSync to write logs to CloudWatch.

      - To use an existing role:

        i. Choose **Existing role**.

        ii. In the service role list, select the ARN of an existing IAM role in your AWS account.

        For information about the configuration of the IAM role, see Manually creating an IAM role with CloudWatch Logs permissions.

5. Choose **Save**.

Manually creating an IAM role with CloudWatch Logs permissions

If you choose to use an existing IAM role, the role must grant AWS AppSync the required permissions to write logs to CloudWatch. To configure this manually, you must provide a service role ARN so that AWS AppSync can assume the role when writing the logs.

In the IAM console, create a new policy with the name `AWSAppSyncPushToCloudWatchLogsPolicy` that has the following definition:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
```

```
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

Next, create a new role with the name **AWSAppSyncPushToCloudWatchLogsRole**, and attach the newly created policy to the role. Edit the trust relationship for this role to the following:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Effect": "Allow",
        "Principal": {
            "Service": "appsync.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
        }
    ]
}
```

Copy the role ARN and use it when setting up logging for an AWS AppSync Event API.

CloudWatch metrics

You can use CloudWatch metrics to monitor and provide alerts about specific events that can result in HTTP status codes or from latency.

HTTP endpoint metrics

`4XXError`

Errors resulting from requests that are not valid due to an incorrect client configuration. For example, these errors can occur when the request includes an incorrect JSON payload or an incorrect query, when the service is throttled, or when the authorization settings are misconfigured.

Unit: *Count*. Use the Sum statistic to get the total occurrences of these errors.

`5XXError`

Errors encountered during the execution of a request. This could also happen if AWS AppSyncc encounters an issue during processing of a request.

Unit: *Count*. Use the Sum statistic to get the total occurrences of these errors.

`Latency`

The time between when AWS AppSync receives a request from a client and when it returns a response to the client. This doesn't include the network latency encountered for a response to reach the end devices.

Unit: *Millisecond*. Use the Average statistic to evaluate expected latencies.

`Requests`

The number of requests (queries + mutations) that all APIs in your account have processed, by Region.

Unit: *Count*. The number of all requests processed in a particular Region.

`TokensConsumed`

Tokens are allocated to `Requests` based on the amount of resources (processing time and memory used) that a `Request` consumes. Usually, each `Request` consumes one token. However, a `Request` that consumes large amounts of resources is allocated additional tokens as needed.

Unit: *Count*. The number of tokens allocated to requests processed in a particular Region.

`DroppedEvents`

The count of input events filtered by a OnPublish handler.

Unit: *Count*.

`FailedEvents`

The count of input events that encountered error during processing.

Unit: *Count*.

`SuccessfulEvents`

The count of input events that were processed successfully and submitted for broadcast in the OnPublish handler.

Unit: *Count*.

`PublishedHandlerInvocations`

The number of OnPublish handler invocations.

Unit: *Count*.

Real-time endpoint metrics

`ConnectRequests`

The number of WebSocket connection requests made to AWS AppSync, including both successful and unsuccessful attempts.

Unit: *Count*. Use the Sum statistic to get the total number of connection requests.

`ConnectSuccess`

The number of successful WebSocket connections to AWS AppSync. It is possible to have connections without subscriptions.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the successful connections.

`ConnectClientError`

The number of WebSocket connections that were rejected by AWS AppSync because of client-side errors. This could imply that the service is throttled or that the authorization settings are misconfigured.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the client-side connection errors.

`ConnectServerError`

The number of errors that originated from AWS AppSync while processing connections. This usually happens when an unexpected server-side issue occurs.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the server-side connection errors.

`DisconnectSuccess`

The number of successful WebSocket disconnections from AWS AppSync.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the successful disconnections.

`DisconnectClientError`

The number of client errors that originated from AWS AppSync while disconnecting WebSocket connections.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the disconnection errors.

`DisconnectServerError`

The number of server errors that originated from AWS AppSync while disconnecting WebSocket connections.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the disconnection errors.

`SubscribeSuccess`

The number of subscriptions that were successfully registered to AWS AppSync through WebSocket. It's possible to have connections without subscriptions, but it's not possible to have subscriptions without connections.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the successful subscriptions.

`SubscribeClientError`

The number of subscriptions that were rejected by AWS AppSync because of client-side errors. This can occur when a JSON payload is incorrect, the service is throttled, or the authorization settings are misconfigured.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the client-side subscription errors.

`SubscribeServerError`

The number of errors that originated from AWS AppSync while processing subscriptions. This usually happens when an unexpected server-side issue occurs.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the server-side subscription errors.

`UnsubscribeSuccess`

The number of unsubscribe requests that were successfully processed.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the successful unsubscribe requests.

`UnsubscribeClientError`

The number of unsubscribe requests that were rejected by AWS AppSync because of client-side errors.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the client-side unsubscribe request errors.

`UnsubscribeServerError`

The number of errors that originated from AWS AppSync while processing unsubscribe requests. This usually happens when an unexpected server-side issue occurs.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the server-side unsubscribe request errors.

`BroadcastEventSuccess`

The number of events that were successfully broadcast to subscribers.

Unit: *Count*. Use the Sum statistic to get the total of events that were successfully broadcast.

`BroadcastEventClientError`

The number of events that failed to broadcast because of client-side errors.

`Unit`: *Count*. Use the Sum statistic to get the total occurrences of the client-side broadcast events errors

`BroadcastEventServerError`

The number of errors that originated from AWS AppSync while broadcasting events . This usually happens when an unexpected server-side issue occurs.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the server-side broadcast errors.

`BroadcastEventSize`

The size of events broadcast.

Unit: *Bytes*.

`ActiveConnections`

The number of concurrent WebSocket connections from clients to AWS AppSync in 1 minute.

Unit: *Count*. Use the Sum statistic to get the total opened connections.

`ActiveSubscriptions`

The number of concurrent subscriptions from clients in 1 minute.

Unit: *Count*. Use the Sum statistic to get the total active subscriptions.

`ConnectionDuration`

The amount of time that the connection stays open.

Unit: *Milliseconds*. Use the Average statistic to evaluate connection duration.

`InboundMessages`

The number of inbound metered events. One metered event equals 5 kB of received event.

Unit: *Count*.

`OutboundMessages`

The number of metered messages successfully published. One metered message equals 5 kB of delivered data.

Unit: *Count*. Use the Sum statistic to get the total number of successfully published metered messages.

`InboundMessageDelayed`

The number of delayed inbound messages. Inbound messages can be delayed when either the inbound message rate quota or outbound message rate quota is breached.

Unit: *Count*. Use the Sum statistic to get the total number of inbound messages that were delayed.

`InboundMessageDropped`

The number of delayed inbound messages. Inbound messages can be delayed when either the inbound message rate quota or outbound message rate quota is breached.

Unit: *Count*. Use the Sum statistic to get the total number of inbound messages that were dropped.

`SubscribeHandlerInvocations`

The number of Subscribe handlers invoked.

Unit: *Count*.

Configuring CloudWatch Logs on Event APIs

You can configure two types of logging on any new or existing API: request-level logs and and handler logs.

Request-level logs

When request-level logging is configured for an API, the following information is logged.

- The number of tokens consumed
- The request and response HTTP headers
- The overall operation summary

Handler logs

When handler logging is configured for an API, the following information is logged.

- Generated request mapping with source and arguments for each field
- The transformed response mapping for each field, which includes the data as a result of resolving that field
- Tracing information for each field

If you turn on logging, AWS AppSync manages the CloudWatch Logs. The process includes creating log groups and log streams, and reporting to the log streams with these logs.

When you turn on logging on an AWS AppSync API and make requests, AWS AppSync creates a log group and log streams under the log group. The log group is named following the `/aws/appsync/apis/{api_id}` format. Within each log group, the logs are further divided into log streams. These are ordered by **Last Event Time** as logged data is reported.

Every log event is tagged with the **x-amzn-RequestId** of that request. This helps you filter log events in CloudWatch to get all logged information about that request. You can get the RequestId from the response headers of every AWS AppSync request.

The field-level logging is configured with the following log levels:

- **None - No handler logs are captured.**
- **Error - Logs the following information *only* for the fields that are in error:**
  - The error section in the server response
  - Handler errors and `console.error` logging from handlers
  - The generated request/response functions that got resolved for error fields
- **All - Logs the following information for *all* fields in the query:**
  - Custom logging from handlers
  - The generated request/response functions that got resolved for each field

Using token counts to optimize your requests

Requests that consume less than or equal to 1,500 KB-seconds of memory and vCPU time are allocated one token. Requests with resource consumption greater than 1,500 KB-seconds receive additional tokens. For example, if a request consumes 3,350 KB-seconds, AWS AppSync allocates three tokens (rounded up to the next integer value) to the request. By default, AWS AppSync allocates a maximum of 5,000 or 10,000 request tokens per second to the APIs in your account, depending upon the AWS Region in which it's deployed. If your APIs each use an average of two tokens per second, you'll be limited to 2,500 or 5,000 requests per second, respectively. If you need more tokens per second than the allotted amount, you can submit a request to increase the default quota for the rate of request tokens. For more information, see AWS AppSync endpoints and quotas in the *AWS General Reference guide* and Requesting a quota increase in the *Service Quotas User Guide*.

A high per-request token count could indicate that there's an opportunity to optimize your requests and improve the performance of your API. Factors that can increase your per-request token count include:

- The complexity of your handlers
- The amount of data returned from your handlers
- Logging configuration, and the amount of custom logging in your handlers

> **Note**
>
> In addition to AWS AppSync metrics and logs, clients can access the number of tokens consumed in a request via the response header `x-amzn-appsync-TokensConsumed`.

Log size limits

By default, if logging has been enabled, AWS AppSync will send up to 1 MB of logs per request.

Using AWS WAF to protect AWS AppSync Event APIs

AWS WAF is a web application firewall that helps protect web applications and APIs from attacks. It allows you to configure a set of rules, called a web access control list (web ACL), that allow, block, or monitor (count) web requests based on customizable web security rules and conditions that you define. When you integrate your AWS AppSync API with AWS WAF, you gain more control and visibility into the HTTP traffic accepted by your API. To learn more about AWS WAF, see How AWS WAF Works in the AWS WAF Developer Guide.

You can use AWS WAF to protect your AppSync API from common web exploits, such as SQL injection and cross-site scripting (XSS) attacks. These could affect API availability and performance, compromise security, or consume excessive resources. For example, you can create rules to allow or block requests from specified IP address ranges, requests from CIDR blocks, requests that originate from a specific country or region, requests that contain malicious SQL code, or requests that contain malicious script.

You can also create rules that match a specified string or a regular expression pattern in HTTP headers, method, query string, URI, and the request body (limited to the first 8 KB). Additionally, you can create rules to block attacks from specific user agents, bad bots, and content scrapers. For example, you can use rate-based rules to specify the number of web requests that are allowed by each client IP in a trailing, continuously updated, 5-minute period.

To learn more about the types of rules that are supported and additional AWS WAF features, see the AWS WAF Developer Guide and the AWS WAF API Reference.

> **Important**
>
> AWS WAF is your first line of defense against web exploits. When AWS WAF is enabled on an API, AWS WAF rules are evaluated before other access control features, such as API key authorization, IAM policies, OIDC tokens, and Amazon Cognito user pools.

Integrate an AppSync API with AWS WAF

You can integrate an Appsync API with AWS WAF using the AWS Management Console, the AWS CLI, AWS CloudFormation, or any other compatible client.

To integrate an AWS AppSync API with AWS WAF

1. Create an AWS WAF web ACL. For detailed steps using the AWS WAF Console, see Creating a web ACL.
2. Define the rules for the web ACL. A rule or rules are defined in the process of creating the web ACL. For information about how to structure rules, see AWS WAF rules. For examples of useful rules you can define for your AWS AppSync API, see Creating rules for a web ACL.
3. Associate the web ACL with an AWS AppSync API. You can perform this step in the AWS WAF Console or in the AppSync Console.
   - To associate the web ACL with an AWS AppSync API in the AWS WAF Console, follow the instructions for Associating or disassociating a Web ACL with an AWS resource in the AWS WAF Developer Guide.
   - To associate the web ACL with an AWS AppSync API in the AWS AppSync Console
     a. Sign in to the AWS Management Console and open the AppSync Console.
     b. Choose the API that you want to associate with a web ACL.
     c. In the navigation pane, choose **Settings**.
     d. In the **Web application firewall** section, turn on **Enable AWS WAF**.
     e. In the **Web ACL** dropdown list, choose the name of the web ACL to associate with your API.
     f. Choose **Save** to associate the web ACL with your API.

> **Note**
>
> After you create a web ACL in the AWS WAF Console, it can take a few minutes for the new web ACL to be available. If you do not see a newly created web ACL in the **Web application firewall** menu, wait a few minutes and retry the steps to associate the web ACL with your API.

> **Note**
>
> AWS WAF integration only supports the `Subscription registration message` event for real-time endpoints. AWS AppSync will respond with an error message instead of a

`start_ack` message for any `Subscription registration message` blocked by AWS WAF.

After you associate a web ACL with an AWS AppSync API, you will manage the web ACL using the AWS WAF APIs. You do not need to re-associate the web ACL with your AWS AppSync API unless you want to associate the AWS AppSync API with a different web ACL.

Creating rules for a web ACL

Rules define how to inspect web requests and what to do when a web request matches the inspection criteria. Rules don't exist in AWS WAF on their own. You can access a rule by name in a rule group or in the web ACL where it's defined. For more information, see AWS WAF rules. The following examples demonstrate how to define and associate rules that are useful for protecting an AppSync API.

Example web ACL rule to limit request body size

The following is an example of a rule that limits the body size of requests. This would be entered into the **Rule JSON editor** when creating a web ACL in the AWS WAF Console.

```
{
    "Name": "BodySizeRule",
    "Priority": 1,
    "Action": {
        "Block": {}
    },
    "Statement": {
        "SizeConstraintStatement": {
            "ComparisonOperator": "GE",
            "FieldToMatch": {
                "Body": {}
            },
            "Size": 1024,
            "TextTransformations": [
                {
                    "Priority": 0,
                    "Type": "NONE"
                }
             ]
          }
```

```
        },
        "VisibilityConfig": {
            "CloudWatchMetricsEnabled": true,
            "MetricName": "BodySizeRule",
            "SampledRequestsEnabled": true
         }
}
```

After you have created your web ACL using the preceding example rule, you must associate it with your AppSync API. As an alternative to using the AWS Management Console, you can perform this step in the AWS CLI by running the following command.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

It can take a few minutes for the changes to propagate, but after running this command, requests that contain a body larger than 1024 bytes will be rejected by AWS AppSync.

> **Note**
>
> After you create a new web ACL in the AWS WAF Console, it can take a few minutes for the web ACL to be available to associate with an API. If you run the CLI command and get a `WAFUnavailableEntityException` error, wait a few minutes and retry running the command.

Example web ACL rule to limit requests from a single IP address

The following is an example of a rule that throttles an AppSync API to 100 requests from a single IP address. This would be entered into the **Rule JSON editor** when creating a web ACL with a rate-based rule in the AWS WAF Console.

```
{
  "Name": "Throttle",
  "Priority": 0,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
```

```
    "MetricName": "Throttle"
  },
  "Statement": {
    "RateBasedStatement": {
      "Limit": 100,
      "AggregateKeyType": "IP"
    }
  }
}
```

After you have created your web ACL using the preceding example rule, you must associate it with your AppSync API. You can perform this step in the AWS CLI by running the following command.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

Document History for AWS AppSync Events

The following table describes the documentation for this release of AWS AppSync Events.

- **API version: 1.1**
- **Latest documentation update:** October 30, 2024

Change	Description	Date
AWS AppSync Events release	This release introduces AWS AppSync Events.	October 30, 2024