Upgrade to the latest Analytics Node.js package

This is the doc for the legacy npm package (`analytics-node`). Segment will deprecate this version of Node on December 31, 2023.Upgrade to the latest package (`@segment/analytics-node`). See the updated Analytics Node.js docs to learn more.

Segment's Node.js library lets you record analytics data from your node code. The requests hit Segment's servers, and then Segment routes your data to any destinations you have enabled.

The Segment Node.js library is open-source on GitHub.

All of Segment's server-side libraries are built for high-performance, so you can use them in your web server controller code. This library uses an internal queue to make `identify` and `track` calls non-blocking and fast. It also batches messages and flushes asynchronously to Segment's servers.

Want to stay updated on releases? Subscribe to the release feed.

Getting Started

Run:

```
npm install --save analytics-node
```

This will add Segment's Node library module to your `package.json`. The module exposes an `Analytics` constructor, which you need to initialize with your Segment source's **Write Key**, like so:

```
var Analytics = require('analytics-node');
var analytics = new Analytics('YOUR_WRITE_KEY');
```

Of course, you'll want to replace `YOUR_WRITE_KEY` with your actual **Write Key** which you can find in Segment under your source settings.

This will create an instance of `Analytics` that you can use to send data to Segment for your project. The default initialization settings are production-ready and queue 20 messages before sending any requests. In development you might want to use development settings.

Regional configuration

For Business plans with access to Regional Segment, you can use the `host` configuration parameter to send data to the desired region:

1. Oregon (Default) — `api.segment.io/v1`
2. Dublin — `events.eu1.segmentapis.com`

An example of setting the host to the EU endpoint using the Node library would be:

```
var analytics = new Analytics('YOUR_WRITE_KEY', {
    host: "https://events.eu1.segmentapis.com"
  });
```

Identify

Good to know: For any of the different methods described on this page, you can replace the properties and traits in the code samples with variables that represent the data collected.

`identify` lets you tie a user to their actions and record traits about them. It includes a unique User ID and/or anonymous ID, and any optional traits you know about them.

You should call `identify` once when the user's account is first created, and then again any time their traits change.

Example of an anonymous `identify` call:

```
analytics.identify({
  anonymousId: '48d213bb-95c3-4f8d-af97-86b2b404dcfe',
  traits: {
    friends: 42
  }
});
```

This call identifies the user and records their unique anonymous ID, and labels them with the `friends` trait.

Example of an `identify` call for an identified user:

```
analytics.identify({
  userId: '019mr8mf4r',
  traits: {
    name: 'Michael Bolton',
    email: 'mbolton@example.com',
    plan: 'Enterprise',
    friends: 42
  }
});
```

The call above identifies Michael by his unique User ID (the one you know him by in your database), and labels him with the `name`, `email`, `plan` and `friends` traits.

The `identify` call has the following fields:

FIELD	DETAILS
`userId` *String, optional*	The ID for this user in your database. *Note: at least one of `userId` or `anonymousId` must be included in any identify call.*
`anonymousId` *String, optional*	An ID associated with the user when you don't know who they are (for example, the anonymousId generated by `analytics.js`). *Note: You must include at least one of `userId` or `anonymousId` in all identify calls.*
`traits` *Object, optional*	A dictionary of traits you know about the user. Things like: `email`, `name` or `friends`.
`timestamp` *Date, optional*	A JavaScript date object representing when the identify call took place. If the identify just happened, leave it out as Segment will use the server's time. If you're importing data from the past make sure you to send a `timestamp`.
`context` *Object, optional*	A dictionary of extra context to attach to the call. *Note: `context` differs from `traits` because it is not attributes of the user itself.*

Find details on the **identify method payload** in the Segment Spec.

Track

`track` lets you record the actions your users perform. Every action triggers what Segment calls an "event", which can also have associated properties.

You'll want to track events that are indicators of success for your site, like **Signed Up**, **Item Purchased** or **Article Bookmarked**.

To get started, Segment recommends tracking just a few important events. You can always add more later.

Example anonymous `track` call:

```
analytics.track({
  anonymousId: '48d213bb-95c3-4f8d-af97-86b2b404dcfe',
  event: 'Item Purchased',
  properties: {
    revenue: 39.95,
    shippingMethod: '2-day'
  }
});
```

Example identified `track` call:

```
analytics.track({
  userId: '019mr8mf4r',
  event: 'Item Purchased',
  properties: {
    revenue: 39.95,
    shippingMethod: '2-day'
  }
});
```

This example `track` call tells that your user just triggered the **Item Purchased** event with a revenue of $39.95 and chose your hypothetical '2-day' shipping.

`track` event properties can be anything you want to record. In this case, revenue and shipping method.

The `track` call has the following fields:

FIELD	DETAILS
`userId` *String, optional*	The ID for this user in your database. _Note: at least one of `userId` or `anonymousId` must be included in any track call.
`anonymousId` *String, optional*	An ID associated with the user when you don't know who they are (for example, the anonymousId generated by `analytics.js`). *Note: You must include at least one of `userId` or `anonymousId` in all track calls.*
`event` *String*	The name of the event you're tracking. Segment recommends you use human-readable names like `Song Played` or `Status Updated`.
`properties` *Object, optional*	A dictionary of properties for the event. If the event was `Product Added`, it might have properties like `price` or `product`.
`timestamp` *Date, optional*	A JavaScript date object representing when the track took place. If the track just happened, leave it out as Segment will use the server's time. If you're importing data from the past make sure you to send a `timestamp`.
`context` *Object, optional*	A dictionary of extra context to attach to the call. *Note: `context` differs from `traits` because it is not attributes of the user itself.*

Find details on **best practices in event naming** as well as the `track` **method payload** in the Segment Spec.

Page

The `page` method lets you record page views on your website, along with optional extra information about the page being viewed.

If you're using Segment's client-side set up in combination with the Node.js library, page calls are already tracked for you by default.

Example `page` call:

```
analytics.page({
  userId: '019mr8mf4r',
  category: 'Docs',
  name: 'Node.js Library',
  properties: {
    url: 'https://segment.com/docs/connections/sources/catalog/librariesnode',
    path: '/docs/connections/sources/catalog/librariesnode/',
    title: 'Node.js Library - Segment',
    referrer: 'https://github.com/segmentio/analytics-node'
  }
});
```

The `page` call has the following fields:

FIELD	DETAILS
`userId` *String, optional*	The ID for this user in your database. _Note: at least one of `userId` or `anonymousId` must be included in any page call.
`anonymousId` *String, optional*	An ID associated with the user when you don't know who they are (for example, the anonymousId generated by `analytics.js`). *Note: at least one of `userId` or `anonymousId` must be included in any page call.*
`category` *String, optional*	The category of the page. Useful for things like ecommerce where many pages often live under a larger category.

FIELD	DETAILS
`name` *String, optional*	The name of the page, for example **Signup** or **Home**.
`properties` *Object, optional*	A dictionary of properties of the page. A few properties specially recognized and automatically translated: `url`, `title`, `referrer` and `path`, but you can add your own too.
`timestamp` *Date, optional*	A JavaScript date object representing when the track took place. If the track just happened, leave it out as Segment will use the server's time. If you're importing data from the past make sure you to send a `timestamp`.
`context` *Object, optional*	A dictionary of extra context to attach to the call. *Note: `context` differs from `traits` because it is not attributes of the user itself.*

Find details on the `page` **payload** in the Segment Spec.

Group

`group` lets you associate an identified user with a group. A group could be a company, organization, account, project or team! It also lets you record custom traits about the group, like industry or number of employees.

This is useful for tools like Intercom, Preact and Totango, as it ties the user to a **group** of other users.

Example `group` call:

```
analytics.group({
  userId: '019mr8mf4r',
  groupId: '56',
  traits: {
    name: 'Initech',
    description: 'Accounting Software'
  }
});
```

The `group` call has the following fields:

FIELD	DETAILS
`userId` *String, optional*	The ID for this user in your database. _Note: at least one of `userId` or `anonymousId` must be included in any group call.
`anonymousId` *String, optional*	An ID associated with the user when you don't know who they are (for example, the anonymousId generated by `analytics.js`). *Note: at least one of `userId` or `anonymousId` must be included in any group call.*
`groupId` _string	The ID of the group.
`traits` *dict, optional*	A dict of traits you know about the group. For a company, they might be things like `name`, `address`, or `phone`. Learn more about traits.
`context` *dict, optional*	A dict containing any context about the request. To see the full reference of supported keys, check them out in the context reference
`timestamp` *datetime, optional*	A `datetime` object representing when the group took place. If the group just happened, leave it out as Segment uses the server's time. If you're importing data from the past make sure you send `timestamp`.
`integrations` *dict, optional*	A dictionary of destinations to enable or disable.

Find more details about `group`, including the `group` **payload**, in the Segment Spec.

Alias

The `alias` call allows you to associate one identity with another. This is an advanced method and should not be widely used, but is required to manage user identities in *some* destinations. Other destinations do not support the alias call.

In Mixpanel it's used to associate an anonymous user with an identified user once they sign up. For Kissmetrics, if your user switches IDs, you can use 'alias' to rename the 'userId'.

Example `alias` call:

```
analytics.alias({
  previousId: 'old_id',
  userId: 'new_id'
});
```

The `alias` call has the following fields:

FIELD	DETAILS
`userId` *String*	The ID for this user in your database.
`previousId` *String*	The previous ID to alias from.

Here's a full example of how Segment might use the `alias` call:

```
// the anonymous user does actions ...
analytics.track({ userId: 'anonymous_user', event: 'Anonymous Event' })
// the anonymous user signs up and is aliased
analytics.alias({ previousId: 'anonymous_user', userId: 'identified@example.com' })
// the identified user is identified
analytics.identify({ userId: 'identified@example.com', traits: { plan: 'Free' } })
// the identified user does actions ...
analytics.track({ userId: 'identified@example.com', event: 'Identified Action' })
```

For more details about `alias`, including the **`alias` call payload**, check out the Spec.

Configuration

The second argument to the `Analytics` constructor is an optional dictionary of settings to configure the module.

```
var analytics = new Analytics('YOUR_WRITE_KEY', {
  flushAt: 20,
  flushInterval: 10000,
  enable: false
});
```

SETTING	DETAILS
`flushAt` *Number*	The number of messages to enqueue before flushing.
`flushInterval` *Number*	The number of milliseconds to wait before flushing the queue automatically.
`enable` *Boolean*	Enable (default) or disable flush. Useful when writing tests and you do not want to send data to Segment Servers.

Error Handling

Additionally there is an optional `errorHandler` property available to the class constructor's options. If unspecified, the behaviour of the library does not change. If specified, when an axios request fails, `errorHandler(axiosError)` will be called instead of re-throwing the axios error.

Example usage:

```
const Analytics = require('analytics-node');

const client = new Analytics('write key', {
  errorHandler: (err) => {
    console.error('analytics-node flush failed.')
    console.error(err)
  }
});

client.track({
  event: 'event name',
  userId: 'user id'
});
```

If this fails when flushed no exception will be thrown, instead the axios error will be logged to the console.

Development

You can use this initialization during development to make the library flush every time a message is submitted, so that you can be sure your calls are working properly before pushing to production.

```
var analytics = new Analytics('YOUR_WRITE_KEY', { flushAt: 1 });
```

Selecting Destinations

The `alias`, `group`, `identify`, `page` and `track` calls can all be passed an object of `integrations` that lets you turn certain destinations on or off. By default all destinations are enabled.

Here's an example with the `integrations` object shown:

```
analytics.track({
  event: 'Membership Upgraded',
  userId: '97234974',
  integrations: {
    'All': false,
    'Vero': true,
    'Google Analytics': false
  }
})
```

In this case, Segment specifies the `track` to only go to Vero. `All: false` says that no destination should be enabled unless otherwise specified. `Vero: true` turns on Vero, etc.

Destination flags are **case sensitive** and match the destination's name in the docs (for example, "AdLearn Open Platform", "awe.sm", "MailChimp"). In some cases, there may be several names for a destination; if that happens you'll see a "Adding (destination name) to the Integrations Object" section in the destination's doc page with a list of valid names.

Note:

- Available at the business level, filtering track calls can be done right from the Segment UI on your source schema page. Segment recommends using the UI if possible since it's a much simpler way of managing your filters and can be updated with no code changes on your side.
- If you are on a grandfathered plan, events sent server-side that are filtered through the Segment dashboard will still count towards your API usage.

Historical Import

You can import historical data by adding the `timestamp` argument to any of your method calls. This can be helpful if you've just switched to Segment.

Historical imports can only be done into destinations that can accept historical timestamped data. Most analytics tools like Mixpanel, Amplitude, Kissmetrics, etc. can handle that type of data just fine. One common destination that does not accept historical data is Google Analytics since their API cannot accept historical data.

Note: If you're tracking things that are happening right now, leave out the `timestamp` and Segment's servers will timestamp the requests for you.

Batching

Segment's libraries are built to support high performance environments. That means it is safe to use the Segment Node library on a web server that's serving hundreds of requests per second.

Every method you call **does not** result in an HTTP request, but is queued in memory instead. Messages are then flushed in batch in the background, which allows for much faster operation.

By default, Segment's library flushes:

- The very first time it gets a message.
- Every 20 messages (controlled by `options.flushAt`).
- If 10 seconds has passed since the last flush (controlled by `options.flushInterval`)

There is a maximum of `500KB` per batch request and `32KB` per call.

If you don't want to batch messages, you can turn batching off by setting the `flushAt` option to `1`, like so:

```
var analytics = new Analytics('YOUR_WRITE_KEY', { flushAt: 1 });
```

Batching means that your message might not get sent right away. But every method call takes an optional `callback`, which you can use to know when a particular message is flushed from the queue, like so:

```
analytics.track({
  userId: '019mr8mf4r',
  event: 'Ultimate Played'
}, function(err, batch){
  if (err) // There was an error flushing your message...
  // Your message was successfully flushed!
});
```

You can also flush on demand. For example, at the end of your program, you need to flush to make sure that nothing is left in the queue. To do that, call the `flush` method:

```
analytics.flush(function(err, batch){
  console.log('Flushed, and now this program can exit!');
});
```

Long running process

You should call `client.track(...)` and know that events will be queued and eventually sent to Segment. To prevent losing messages, be sure to capture any interruption (for example, a server restart) and call flush to know of and delay the process shutdown.

```
import { randomUUID } from 'crypto';
import Analytics from 'analytics-node'

const WRITE_KEY = '...';

const analytics = new Analytics(WRITE_KEY, { flushAt: 10 });

analytics.track({
  anonymousId: randomUUID(),
  event: 'Test event',
  properties: {
    name: 'Test event',
    timestamp: new Date()
  }
});

const exitGracefully = async (code) => {
  console.log('Flushing events');
  await analytics.flush(function(err, batch) {
    console.log('Flushed, and now this program can exit!');
    process.exit(code);
  });
};

[
  'beforeExit', 'uncaughtException', 'unhandledRejection',
  'SIGHUP', 'SIGINT', 'SIGQUIT', 'SIGILL', 'SIGTRAP',
  'SIGABRT','SIGBUS', 'SIGFPE', 'SIGUSR1', 'SIGSEGV',
  'SIGUSR2', 'SIGTERM',
].forEach(evt => process.on(evt, exitGracefully));

function logEvery2Seconds(i) {
    setTimeout(() => {
        console.log('Infinite Loop Test n:', i);
        logEvery2Seconds(++i);
    }, 2000);
}

logEvery2Seconds(0);
```

Short lived process

Short-lived functions have a predictably short and linear lifecycle, so use a queue big enough to hold all messages and then await flush to complete its work.

```
import { randomUUID } from 'crypto';
import Analytics from 'analytics-node'


async function lambda()
{
  const WRITE_KEY = '...';
  const analytics = new Analytics(WRITE_KEY, { flushAt: 20 });
  analytics.flushed = true;

  analytics.track({
    anonymousId: randomUUID(),
    event: 'Test event',
    properties: {
      name: 'Test event',
      timestamp: new Date()
    }
  });
  await analytics.flush(function(err, batch) {
    console.log('Flushed, and now this program can exit!');
  });
}

lambda();
```

Multiple Clients

Different parts of your application may require different types of batching, or even sending to multiple Segment sources. In that case, you can initialize multiple instances of `Analytics` with different settings:

```
var Analytics = require('analytics-node');
var marketingAnalytics = new Analytics('MARKETING_WRITE_KEY');
var appAnalytics = new Analytics('APP_WRITE_KEY');
```

Troubleshooting

The following tips often help resolve common issues.

No events in my debugger

1. Double check that you've followed all the steps in the Quickstart.
2. Make sure that you're calling a Segment API method once the library is successfully installed—`identify`, `track`, etc.
3. Make sure your application isn't shutting down before the `Analytics.Client` local queue events are pushed to Segment. You can manually call `Analytics.Client.Flush()` to ensure the queue is fully processed before shutdown.

Other common errors

If you are experiencing data loss from your source, you may be experiencing one or more of the following common errors:

- **Payload is too large**: If you attempt to send events larger than 32KB per normal API request or batches of events larger than 500KB per request, Segment's tracking API responds with `400 Bad Request`. Try sending smaller events (or smaller batches) to correct this error.
- **Identifier is not present**: Segment's tracking API requires that each payload has a `userId` and/or `anonymousId`. If you send events without either the `userId` or `anonymousId`, Segment's tracking API responds with an `no_user_anon_id` error. Check the event payload and client instrumentation for more details.
- **Track event is missing name**: All Track events to Segment must have a name in string format.
- **Event dropped during deduplication**: Segment automatically adds a `messageId` field to all payloads and uses this value to deduplicate events. If you're manually setting a `messageId` value, ensure that each event has a unique value.
- **Incorrect credentials**: Double check your credentials for your downstream destination(s).
- **Destination incompatibility**: Make sure that the destination you are troubleshooting can accept server-side API calls. You can see compatibility information on the Destination comparison by category page and in the documentation for your specific destination.
- **Destination-specific requirements**: Check out the destination's documentation to see if there are other requirements for using the method and destination that you're trying to get working.

This page was last modified: 29 May 2024

Need support?

Questions? Problems? Need more info? Contact Segment Support for assistance!

Visit our Support page

Help improve these docs!

Edit this page

Request docs change

Was this page helpful?

Yes

No

Get started with Segment

Segment is the easiest way to integrate your websites & mobile apps data to over 300 analytics and growth tools.

Your work e-mail

Request Demo

or

Create free account

© 2025 Segment.io, Inc.

Privacy

Terms

Website Data Collection Preferences