any information you want to tie to the user. When using any of the reserved user traits, be sure the information reflects the name of the trait. For example, `email` should always be a string of the user's email address.

To send updates for anonymous users who haven't yet signed up for your app, pass `null` for the `userId` like in the example below.

Method signature

Example use

Track

The Track method lets you record the actions your users perform. Every action triggers an event, which also has associated properties that the track method records.

Method signature

Example use

Screen

The Screen method lets you record whenever a user sees a screen in your mobile app, along with optional extra information about the page being viewed.

You'll want to record a screen event whenever the user opens a screen in your app. This could be a view, fragment, dialog, or activity depending on your app.

Not all integrations support screen, so when it's not supported explicitly, the screen method tracks as an event with the same parameters.

Method signature

Example use

For setting up automatic screen tracking, see the Automatic Screen Tracking instructions.

Group

The Group method lets you associate an individual user with a group— whether it's a company, organization, account, project, or team. This includes a unique group identifier and any additional group traits you may know, like company name, industry, or number of employees. You can include any information you want to associate with the group in the traits option. When using any of the reserved group traits, be sure the information reflects the name of the trait. For example, email should always be a string of the user's email address.

Method signature

Example use

Utility methods

The Analytics React Native utility methods help you to manage your data. They include:

- Alias
- Reset
- Flush
- Cleanup

Alias

The alias method is used to merge two user identities by connecting two sets of user data as one. This method is required to manage user identities in some of Segment's destinations.

Method signature

Example use

Reset

The reset method clears the internal state of the library for the current user and group. This is useful for apps where users can log in and out with different identities over time.

⚠
Note: Each time you call reset, a new AnonymousId is generated automatically.

Method signature

Example use

Flush

By default, the analytics client sends queued events to the API every 30 seconds or when 20 events accumulate, whichever occurs first. This also occurs whenever the app resumes if the user has closed the app with some events unsent. These values can be modified by the `flushAt` and `flushInterval` config options. You can also trigger a flush event manually.

Method signature

Example use

Cleanup

In case you need to reinitialize the client, that is, you've called createClient more than once for the same client in your application lifecycle, use this method on the old client to clear any subscriptions and timers first.

```
let client = createClient({
  writeKey: 'KEY'
});

client.cleanup();

client = createClient({
  writeKey: 'KEY'
});
```

If you don't do this, the old client instance would still exist and retain the timers, making all your events fire twice.

Ideally, you shouldn't have to use this method, and the Segment client should be initialized only once in the application lifecycle.

Advanced functionality

Analytics React Native was built to be as extensible and customizable as possible to give you the ability to meet your bespoke analytics needs.

- Control upload with flush policies
- Add or remove policies
- Create your own flush policies
- Automatic screen tracking
- React navigation
- React Native navigation
- Handle errors
- Report errors from plugins
- Native anonymousId
- Retrieving the anonymousId
- Configure iOS deep link tracking
- Device identifiers
- Using a WebView Component with React Native

Control upload with flush policies

To more granularly control when events are uploaded you can use `FlushPolicies`

A Flush Policy defines the strategy for deciding when to flush, this can be on an interval, on a certain time of day, after receiving a certain number of events or even after receiving a particular event. This gives you even

more flexibility on when to send event to Segment.

To make use of flush policies you can set them in the configuration of the client:

```
const client = createClient({
  // ...
  flushPolicies: [
    new CountFlushPolicy(5),
    new TimerFlushPolicy(500),
    new StartupFlushPolicy(),
  ],
});
```

Add or remove policies

One of the main advantages of FlushPolicies is that you can add and remove policies on the fly. This is very powerful when you want to reduce or increase the amount of flushes.

For example you might want to disable flushes if you detect the user has no network:

```
import NetInfo from "@react-native-community/netinfo";

const policiesIfNetworkIsUp = [
  new CountFlushPolicy(5),
  new TimerFlushPolicy(500),
];

// Create our client with our policies by default
const client = createClient({
  // ...
  flushPolicies: policies,
});

// If we detect the user disconnects from the network remove all flush policies,
// that way we won't keep attempting to send events to segment but we will still
// store them for future upload.
// If the network comes back up we add the policies back
const unsubscribe = NetInfo.addEventListener((state) => {
  if (state.isConnected) {
    client.addFlushPolicy(...policiesIfNetworkIsUp);
  } else {
    client.removeFlushPolicy(...policiesIfNetworkIsUp)
  }
});
```

Create your own flush policies

You can create a custom FlushPolicy special for your application needs by implementing the `FlushPolicy` interface. You can also extend the `FlushPolicyBase` class that already creates and handles the `shouldFlush` value reset.

A `FlushPolicy` only needs to implement 2 methods:

- `start()`: Executed when the flush policy is enabled and added to the client. This is a good place to start background operations, make async calls, configure things before execution
- `onEvent(event: SegmentEvent)`: Gets called on every event tracked by your client
- `reset()`: Called after a flush is triggered (either by your policy, by another policy or manually)

They also have a `shouldFlush` observable boolean value. When this is set to true the client will attempt to upload events. Each policy should reset this value to `false` according to its own logic, although it is pretty common to do it inside the `reset` method.

```
export class FlushOnScreenEventsPolicy extends FlushPolicyBase {

  onEvent(event: SegmentEvent): void {
    // Only flush when a screen even happens
    if (event.type === EventType.ScreenEvent) {
      this.shouldFlush.value = true;
    }
  }

  reset(): void {
    // Superclass will reset the shouldFlush value so that the next screen event triggers a flush again
    // But you can also reset the value whenever, say another event comes in or after a timeout
    super.reset();
  }
}
```

Automatic screen tracking

As sending a screen() event with each navigation action can get tiresome, it's best to track navigation globally. The implementation is different depending on which library you use for navigation. The two main navigation libraries for React Native are React Navigation and React Native Navigation.

React navigation

When setting up React Navigation, you'll essentially find the root level navigation container and call `screen()` whenever the user navigates to a new screen. Segment's example app is set up with screen tracking using React Navigation, so you can use it as a guide.

To set up automatic screen tracking with React Navigation:

1. Find the file where you used the `NavigationContainer`. This is the main top level container for React Navigation.
2. In the component, create a new state variable to store the current route name:

```
const [routeName, setRouteName] = useState('Unknown');
```

3. Create a utility function for determining the name of the selected route outside of the component:

```
const getActiveRouteName = (
  state: NavigationState | PartialState<NavigationState> | undefined
): string => {
  if (!state || typeof state.index !== 'number') {
    return 'Unknown';
  }

  const route = state.routes[state.index];

  if (route.state) {
    return getActiveRouteName(route.state);
  }

  return route.name;
};
```

4. Pass a function in the `onStateChange` prop of your `NavigationContainer` that checks for the active route name and calls `client.screen()` if the route has changes. You can pass in any additional screen parameters as the second argument for screen calls as needed.

```
<NavigationContainer
onStateChange={(state) => {
  const newRouteName = getActiveRouteName(state);

  if (routeName !== newRouteName) {
    segmentClient.screen(newRouteName);
    setRouteName(newRouteName);
  }
}}
>
```

React Native navigation

In order to set up automatic screen tracking while using React Native Navigation:

1. Use an event listener at the point where you set up the root of your application (for example, `Navigation.setRoot`).
2. Access your `SegmentClient` at the root of your application.

```
    // Register the event listener for *registerComponentDidAppearListener*
    Navigation.events().registerComponentDidAppearListener(({ componentName }) => {
      segmentClient.screen(componentName);
    });
```

Handle errors

You can handle analytics client errors through the `errorHandler` option.

The error handler configuration receives a function which will get called whenever an error happens on the analytics client. It will receive an argument of `SegmentError` type.

You can use this error handling to trigger different behaviours in the client when a problem occurs. For example if the client gets rate limited you could use the error handler to swap flush policies to be less aggressive:

```
const flushPolicies = [new CountFlushPolicy(5), new TimerFlushPolicy(500)];

const errorHandler = (error: SegmentError) => {
  if (error.type === ErrorType.NetworkServerLimited) {
    // Remove all flush policies
    segmentClient.removeFlushPolicy(...segmentClient.getFlushPolicies());
    // Add less persistent flush policies
    segmentClient.addFlushPolicy(
      new CountFlushPolicy(100),
      new TimerFlushPolicy(5000)
    );
  }
};

const segmentClient = createClient({
  writeKey: 'WRITE_KEY',
  trackAppLifecycleEvents: true,
  collectDeviceId: true,
  debug: true,
  trackDeepLinks: true,
  flushPolicies: flushPolicies,
  errorHandler: errorHandler,
});
```

The reported errors can be of any of the `ErrorType` enum values.

Report errors from plugins

Plugins can also report errors to the handler by using the `.reportInternalError` function of the analytics client, Segment recommends that you use the `ErrorType.PluginError` for consistency, and attaching the `innerError` with the actual exception that was hit:

```
  try {
    distinctId = await mixpanel.getDistinctId();
  } catch (e) {
    analytics.reportInternalError(
      new SegmentError(ErrorType.PluginError, 'Error: Mixpanel error calling getDistinctId', e)
    );
    analytics.logger.warn(e);
  }
```

Native AnonymousId

If you need to generate an `anonymousId` either natively or before the Analytics React Native package is initialized, you can send the anonymousId value from native code. The value has to be generated and stored by the caller.

For reference, you can find a working example in the app and reference the code below:

iOS

```
...
#import <segment_analytics_react_native-Swift.h>

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  ...
  // generate your anonymousId value
  // dispatch it across the bridge

  [AnalyticsReactNative setAnonymousId: @"My-New-Native-Id"];
  return yes
}
```

Android

```
// MainApplication.java
...
import com.segmentanalyticsreactnative.AnalyticsReactNativePackage;

...
private AnalyticsReactNativePackage analytics = new AnalyticsReactNativePackage();

...
   @Override
    protected List<ReactPackage> getPackages() {
      @SuppressWarnings("UnnecessaryLocalVariable")
      List<ReactPackage> packages = new PackageList(this).getPackages();
      // AnalyticsReactNative will be autolinked by default, but to send the anonymousId before RN startup you need t
o manually link it to store a reference to the package
      packages.add(analytics);
      return packages;
    }
...
  @Override
  public void onCreate() {
    super.onCreate();
    ...

  // generate your anonymousId value
  // dispatch it across the bridge

  analytics.setAnonymousId("My-New-Native-Id");
  }
```

Retrieving the anonymousId

The React Native library does not have a specific method for retrieving the anonymousId. However, you can access this value by calling the following in your code:

```
segmentClient.userInfo.get().anonymousId
```

Retrieving the anonymousId can be useful if you need to pass this value to your backend, or if you're using a web view component with Segment's Analytics.js library and need to link user activity.

Configure iOS deep link tracking

> ⚠️
> This is only required for iOS if you're using the `trackDeepLinks` option. Android doesn't require any additional setup.

To track deep links in iOS, add the following to your `AppDelegate.m` file:

```
#import <segment_analytics_react_native-Swift.h>
....
- (BOOL)application:(UIApplication *)application
          openURL: (NSURL *)url
          options:(nonnull NSDictionary<UIApplicationOpenURLOptionsKey, id> *)options {

  [AnalyticsReactNative trackDeepLink:url withOptions:options];
  return YES;
}
```

Device identifiers

On Android, Segment's React Native library generates a unique ID by using the DRM API as context.device.id. Some destinations rely on this field being the Android ID, so be sure to double-check the destination's vendor documentation. If you choose to override the default value using a plugin, make sure the identifier you choose complies with Google's User Data Policy. For iOS the context.device.id is set the IDFV.

To collect the Android Advertising ID provided by Play Services, Segment provides a plugin that can be used to collect that value. This value is set to context.device.advertisingId. For iOS, this plugin can be used to set the IDFA context.device.advertisingId property.

Using a WebView Component with React Native

If you use a webView component in your app that uses Segment's Analytics.js library, you can use Segment's Querystring API to pass the anonymousId from your React Native app to Analytics.js to ensure activity from anonymous users can be linked across these two sources.

To retrieve and pass the anonymousId:

1. Retrieve anonymousId from the React Native library using:

```
const anonymousId = segmentClient.userInfo.get().anonymousId
```

2. Pass this value into the querystring that opens the webview using the `ajs_aid` optional query string parameter noted in the documentation above. For example, the URL that opens your webview might look like:

```
http://segment.com/?ajs_aid={anonymousId}
```

3. When a user clicks the element that opens the webview, Analytics.js will read that parameter and automatically set the anonymousId to whatever value is passed in, linking your events across both libraries to the same user.

Changelog

View the Analytics React Native changelog on GitHub.

This page was last modified: 09 Feb 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