Experiment JavaScript SDK

Copy page

Official documentation for Amplitude Experiment's Client-side JavaScript SDK implementation.

Install

Install the Experiment JavaScript Client SDK with one of the three following methods:

Unified SDK

Install the Browser Unified SDK to access the Experiment SDK along with other Amplitude products (Analytics, Session Replay). The Unified SDK provides a single entry point for all Amplitude features and simplifies the integration process by handling the initialization and configuration of all components.

npm | yarn | script

```bash
# Install Experiment SDK only
npm install --save @amplitude/experiment-js-client

# Or install Unified SDK to get access to all Amplitude products
npm install @amplitude/unified
```

torchlight.dev

Quick start

The right way to initialize the Experiment SDK depends on whether you use an Amplitude SDK for analytic
third party (for example, Segment).

Ask AI

2. Fetch variants
3. Access a flag's variant

```js
import { Experiment } from '@amplitude/experiment-js-client';

// (1) Initialize the experiment client with Amplitude Analytics.
const experiment = Experiment.initializeWithAmplitudeAnalytics(
    'DEPLOYMENT_KEY'
);

// (2) Fetch variants and await the promise result.
await experiment.fetch();

// (3) Lookup a flag's variant.
const variant = experiment.variant('FLAG_KEY');
if (variant.value === 'on') {
    // Flag is on
} else {
    // Flag is off
}
```

torchlight.dev

Initialize

Initialize the SDK in your application on startup. The deployment key argument you pass into the `apiKey` parameter must live in the same Amplitude project to which you send events.

Amplitude | Unified SDK | Third-party

```js
initializeWithAmplitudeAnalytics(apiKey: string, config?: ExperimentConfig): ExperimentClient
```

torchlight.d

`config`	The client configuration to customize SDK client behavior.

The initializer returns a singleton instance, so subsequent initializations for the same instance name always return the initial instance. To create multiple instances, use the `instanceName` configuration.

Amplitude | Unified SDK | Third-party

```js
import { Experiment } from '@amplitude/experiment-js-client';

const experiment = initializeWithAmplitudeAnalytics('DEPLOYMENT_KEY');
```

torchlight.dev

Configuration

Configure the SDK client once during initialization.

Configuration

Name	Description	Default Val
`debug`	Enable additional debug logging within the SDK. Should be set to false in production builds.	`false`
`fallbackVariant`	The default variant to fall back if a variant for the provided key doesn't exist.	`{}`
`initialVariants`	An initial set of variants to access. This field is valuable for bootstrapping	`{}`

	rendering (SSR).	
`source`	The primary source of variants. Set the value to `Source.InitialVariants` and configured `initialVariants` to bootstrap the SDK for SSR or testing purposes.	`Source.Loc`
`serverZone`	Select the Amplitude data center to get flags and variants from, `us` or `eu`.	`us`
`serverUrl`	The host to fetch remote evaluation variants from. For hitting the EU data center, use `serverZone`.	`https://ap`
`flagsServerUrl`	The host to fetch local evaluation flags from. For hitting the EU data center, use `serverZone`.	`https://fl`
`fetchTimeoutMillis`	The timeout for fetching variants in milliseconds.	`10000`
`retryFetchOnFailure`	Whether to retry variant fetches in the background if the request doesn't succeed.	`true`
`automaticExposureTracking`	If true, calling `variant()` will track an exposure event through the configured `exposureTrackingProvider`. If no exposure tracking provider is set, this configuration option does nothing.	`true`
`fetchOnStart`	If true or undefined, always fetch remote evaluation variants on start. If false, never fetch on start.	`true`

	minute on start.	
`automaticFetchOnAmplitudeIdentityChange`	Only matters if you use the `initializeWithAmplitudeAnalytics` initialization function to seamlessly integrate with the Amplitude Analytics SDK. If `true` any change to the user ID, device ID or user properties from analytics will trigger the experiment SDK to fetch variants and update it's cache.	`false`
`userProvider`	An interface used to provide the user object to `fetch()` when called.	`null`
`exposureTrackingProvider`	Implement and configure this interface to track exposure events through the experiment SDK, either automatically or explicitly.	`null`
`instanceName`	Custom instance name for experiment SDK instance. **The value of this field is case-sensitive.**	`null`
`initialFlags`	A JSON string representing an initial array of flag configurations to use for local evaluation.	`undefined`
`httpClient`	(Advanced) Use your own HTTP client implementation to handle network requests made by the SDK.	Default HTT

Integrations

exposures events.

Amplitude integration

The Amplitude Experiment SDK is set up to integrate seamlessly with the Amplitude Analytics SDK.

```js
import * as amplitude from '@amplitude/analytics-browser';
import { Experiment } from '@amplitude/experiment-js-client';

amplitude.init('API_KEY');
const experiment = Experiment.initializeWithAmplitudeAnalytics('DEPLOYM
```

torchlight.dev

Using the integration initializer configures implementations of the user provider and exposure tracking provider interfaces to pull user data from the Amplitude Analytics SDK and track exposure events.

Supported Versions

All versions of the next-generation Amplitude analytics Browser SDK support this integration.

Legacy Analytics SDK Version	Experiment SDK Version
`8.18.1+`	`1.4.1+`

Segment integration

Fetch

Fetches variants for a user and store the results in the client for fast access. This function remote evaluates the user for flags associated with the deployment used to initialize the SDK client.

Fetch on user identity change

If you want the most up-to-date variants for the user, it's recommended that you call `fetch()` whenever the user state changes in a meaningful way. For example, if the user logs in and receives a user ID, or has a user property set which may effect flag or experiment targeting rules.

Pass new **user properties** explicitly to `fetch()` instead of relying on user enrichment prior to remote evaluation. This is because user properties that are synced remotely through a separate system have no timing guarantees with respect to `fetch()`--for example, a race.

```js
fetch(user?: ExperimentUser, options?: FetchOptions): Promise<Client>
```

torchlight.dev

Parameter	Requirement	Description
`user`	optional	Explicit user information to pass with the request to evaluate. This user information is merged with user information provided from integrations via the user provider, preferring properties passed explicitly to `fetch()` over provided properties.
`options`	optional	Explicit flag keys to fetch.

Account-level bucketing and analysis (v1.5.6+)

```js
const user = {
    user_id: 'user@company.com',
    device_id: 'abcdefg',
```

```
  await experiment.fetch(user);
```

torchlight.dev

If you're using an integration or a custom user provider then you can fetch without inputting the user.

```js
  await experiment.fetch();
```

torchlight.dev

If `fetch()` times out (default 10 seconds) or fails for any reason, the SDK client will return and retry in the background with back-off. You may configure the timeout or disable retries in the configuration options when the SDK client is initialized.

Start

Fetch vs start

Use `start` if you're using client-side local evaluation. If you're only using remote evaluation, call fetch instead of `start`.

Start the SDK by getting flag configurations from the server and fetching remote evaluation variants for the user. The SDK is ready once the returned promise resolves.

```js
  start(user?: ExperimentUser): Promise<void>
```

torchlight.dev

Parameter	Requirement	Description
`user`	optional	Explicit user information to pass with the request to fetch variants. This user information is merged with user information provided from integrations via the

Call `start()` when your application is initializing, after user information is available to use to evaluate or fetch variants. The returned promise resolves after loading local evaluation flag configurations and fetching remote evaluation variants.

Configure the behavior of `start()` by setting `fetchOnStart` in the SDK configuration on initialization to improve performance based on the needs of your application.

- If your application never relies on remote evaluation, set `fetchOnStart` to `false` to avoid increased startup latency caused by remote evaluation.
- If your application relies on remote evaluation, but not right at startup, you may set `fetchOnStart` to `false` and call `fetch()` and await the promise separately.

Amplitude | Third party

```js
await experiment.start();
```

torchlight.dev

Variant

Access a variant for a flag or experiment from the SDK client's local store.

Automatic exposure tracking

When an integration is used or a custom exposure tracking provider is set, `variant()` will automatically track an exposure event through the tracking provider. To disable this functionality, configure `automaticExposureTracking` to be `false`, and track exposures manually using `exposure()`.

```js
variant(key: string, fallback?: string | Variant): Variant
```

`key`	required	The **flag key** to identify the flag or experiment to access the variant for.
`fallback`	optional	The value to return if no variant was found for the given `flagKey`.

When determining which variant a user has been bucketed into, you'll want to compare the variant `value` to a well-known string.

```js
const variant = experiment.variant('<FLAG_KEY>');
if (variant.value === 'on') {
    // Flag is on
} else {
    // Flag is off
}
```

torchlight.dev

Access a variant's payload

A variant may also be configured with a dynamic payload of arbitrary data. Access the `payload` field from the variant object after checking the variant's `value`.

```js
const variant = experiment.variant('<FLAG_KEY>');
if (variant.value === 'on') {
    const payload = variant.payload;
}
```

torchlight.dev

A `null` variant `value` means that the user hasn't been bucketed into a variant. You may use the built in **fallback** parameter to provide a variant to return if the store doesn't contain a variant for the given flag key.

```js
const variant = experiment.variant('<FLAG_KEY>', { value: 'control' });
if (variant === 'control') {
```

All

Access all variants stored by the SDK client.

```
js
all(): Variants
```


Clear

Clear all variants in the cache and storage.

```
js
clear(): void
```


You can call `clear` after user logout to clear the variants in cache and storage.

```
js
experiment.clear();
```


Exposure

Manually track an exposure event for the current variant of the given flag key through configured integration or custom exposure tracking provider. Generally used in conjunction with setting the `automaticExposureTracking` configuration optional to `false`.

Parameter	Requirement	Description
`key`	required	The **flag key** to identify the flag or experiment variant to track an exposure event for.

```js
const variant = experiment.variant('<FLAG_KEY>');

// Do other things...

experiment.exposure('<FLAG_KEY>');
if (variant === 'control') {
    // Control
} else if (variant === 'treatment') {
    // Treatment
}
```

torchlight.dev

Providers

Integrations

If you use Amplitude or Segment analytics SDKs along side the Experiment Client SDK, Amplitude recommends you use an integration instead of implementing custom providers.

Provider implementations enable a more streamlined developer experience by making it easier to manage user identity and track exposures events.

User provider

The user provider is used by the SDK client to access the most up-to-date user information only when it's needed (for example, when `fetch()` is called). This provider is optional, but helps if you have a user information store already set up in your application. This way, you don't need to

```
interface ExperimentUserProvider {
  getUser(): ExperimentUser;
}
```

torchlight.dev

To use your custom user provider, set the `userProvider` configuration option with an instance of your custom implementation on SDK initialization.

```js
const experiment = Experiment.initialize('<DEPLOYMENT_KEY>', {
    userProvider: new CustomUserProvider(),
});
```

torchlight.dev

Exposure tracking provider

Implementing an exposure tracking provider is highly recommended. Exposure tracking increases the accuracy and reliability of experiment results and improves visibility into which flags and experiments a user is exposed to.

```js
export interface ExposureTrackingProvider {
  track(exposure: Exposure): void;
}
```

torchlight.dev

The implementation of `track()` should track an event of type `$exposure` (a.k.a name) with two event properties, `flag_key` and `variant`, corresponding to the two fields on the `Exposure` object argument. Finally, the event tracked must eventually end up in Amplitude Analytics for the same project that the [deployment] used to initialize the SDK client lives within, and for the same user that variants were fetched for.

```
const experiment = Experiment.initialize('<DEPLOYMENT_KEY>', {
    exposureTrackingProvider: new CustomExposureTrackingProvider(),
});
```

torchlight.dev

Bootstrapping

You may want to bootstrap the experiment client with an initial set of flags or variants when variants are obtained from an external source (for example, not from calling `fetch()` on the SDK client). Use cases include local evaluation, server-side rendering, or integration testing on specific variants.

Bootstrapping variants

To bootstrap the client with a predefined set of variants, set the flags and variants in the `initialVariants` configuration object, then set the `source` to `Source.InitialVariants` so that the SDK client prefers the bootstrapped variants over any previously fetched & stored variants for the same flags.

```js
const experiment = Experiment.initialize('<DEPLOYMENT_KEY>', {
    // Map flag keys to variant objects. The variant object may either be
    // pre-evaluation (SSR) or input manually in for testing.
    initialVariants: {
        "<FLAG_KEY>": {
            "value": "<VARIANT>"
         }
    },
    source: Source.InitialVariants,
});
```

torchlight.dev

updated flag config or variant is loaded with start or fetch.

To download initial flags, use the evaluation flags API

```js
const experiment = Experiment.initialize('<DEPLOYMENT_KEY>', {
    initialFlags: "<FLAGS_JSON>",
});
```

torchlight.dev

HTTP client

You can provide a custom HTTP client implementation to handle network requests made by the SDK. This is useful for environments with specific networking requirements or when you need to customize request handling.

```js
export interface SimpleResponse {
  status: number;
  body: string;
}

export interface HttpClient {
  request(
    requestUrl: string,
    method: string,
    headers: Record<string, string>,
    data: string,
    timeoutMillis?: number,
  ): Promise<SimpleResponse>;
}
```

torchlight.dev

```
const experiment = Experiment.initialize('<DEPLOYMENT_KEY>', {
    httpClient: new CustomHttpClient(),
});
```

torchlight.dev

Was this page helpful?

June 4th, 2024

Need help? Contact Support

Visit Amplitude.com

Have a look at the Amplitude Blog

Learn more at Amplitude Academy