FetchEvent

August 17, 2026

FetchEvent

Kind: Interface

Source: `src/adapter/service-worker/types.ts`

Part of: Adapter

`FetchEvent` represents a fetch handled by the service worker adapter. It carries the incoming `Request`, client identifiers, preload state, and a completion promise, while `respondWith()` marks the event as handled by the adapter.

Properties

Property	Type
`clientId`	`string`
`handled`	`Promise<void>`
`preloadResponse`	`Promise<any>`
`request`	`Request`
`resultingClientId`	`string`

Diagram

```
graph LR
  Client[Client] --> Request[Request]
  Request --> Event[FetchEvent]
  Event --> ClientId[clientId]
  Event --> ResultingClientId[resultingClientId]
  Event --> Preload[preloadResponse]
  Event --> Handled[handled]
  Event --> RespondWith[respondWith()]
```

Usage

```
function handleFetch(event: FetchEvent): void {
  const request = event.request;

  void event.preloadResponse.then((preloaded) => {
    console.log("Preloaded response:", preloaded);
  });

  void event.handled.then(() => {
    console.log("Fetch handling completed for:", request.url);
  });

  console.log("Client:", event.clientId);
  console.log("Resulting client:", event.resultingClientId);

  event.respondWith();
}
```

AI Coding Instructions

- Read `event.request` for the incoming URL, method, headers, and request body.
- Call `respondWith()` according to the adapter event contract.
- Treat `handled` as a signal that fetch processing has completed.
- Do not assume `preloadResponse` resolves to a `Response`; its type is `any`.
- Use `clientId` and `resultingClientId` when request handling depends on client navigation state.

How it works

`FetchEvent` is an exported TypeScript interface for the event accepted by the service-worker adapter's handler. It extends `ExtendableEvent`, which itself extends `Event`. [src/adapter/service-worker/types.ts:1-6] The adapter defines its handler as a function that takes this interface and returns `void`. [src/adapter/service-worker/handler.ts:10]

It declares these members:

- `request`: a read-only `Request` representing the request passed to the application. [src/adapter/service-worker/types.ts:11]
- `respondWith(r)`: a method accepting either a `Response` or a thenable that resolves to a `Response`. [src/adapter/service-worker/types.ts:13]
- `clientId` and `resultingClientId`: read-only strings. [src/adapter/service-worker/types.ts:7,12]

- `handled`: a read-only `Promise<void>`. [src/adapter/service-worker/types.ts:8]
- `preloadResponse`: a read-only `Promise<any>`. [src/adapter/service-worker/types.ts:9-10]
- `waitUntil(f)`, inherited from `ExtendableEvent`, which accepts a `Promise<any>` and returns `void`. [src/adapter/service-worker/types.ts:1-4]

When `handle(app, opts)` receives a `FetchEvent`, it calls `evt.respondWith(...)` with an async operation. That operation calls `app.fetch(evt.request, {}, evt)`, passing the event as the third argument. [src/adapter/service-worker/handler.ts:25-35] Consequently, application code can read event members through its execution context; the adapter test reads `c.executionCtx.clientId` and receives the event's mocked `clientId`. [src/adapter/service-worker/handler.test.ts:95-119]

If the application response has status `404` and `opts.fetch` is defined, the handler calls `opts.fetch(evt.request)` and passes that result to `respondWith`; otherwise, it passes the application response. [src/adapter/service-worker/handler.ts:29-35] The default option binds `globalThis.fetch` to `globalThis`. [src/adapter/service-worker/handler.ts:20-23]

`FetchEvent` itself contains only type declarations: it has no implementation, runtime validation, thrown errors, or direct side effects in `types.ts`. [src/adapter/service-worker/types.ts:1-14]