

Background

In response to recent discussions and concerns ([internal notes](#)/[external notes](#)) around the API shape of [WICG/pending-beacon](#), instead of creating an independent PendingBeacon class family, this document proposes `class PendingRequest`, a subclass of fetch's [Request](#).

API Requirements

The following requirements are critical:

1. Support a reliable mechanism for delaying operation until page discard, including unloading and bfcache eviction.
2. Support an optional timeout to allow early sending after visibility hidden or bfcached.
3. Behave like a keepalive fetch request when 1's mechanism triggers.
4. Allow pending requests to be updated to reduce network usage.

PendingRequest API

```
// A type of request that will only be sent by the browser in the scenario:
//   1. on page discarded
//   2. on page evicted from BackForwardCache
interface PendingRequest : Request {
  constructor(RequestInfo input,
              optional PendingRequestInit init = {});
  // Returns true if this request is still pending; returns false if it is being
  // sent or fails to send.
  // Defaults to true.
  readonly attribute boolean pending;

  // Transforms this request into the non-pending state if not already.
  // After this call, this request should be treated as a regular Request.
  void sendNow();

  // Specifies a timeout in milliseconds for a timer that only starts after
  // the page enters the next "hidden" visibility state.
  // If x >= 0, the request will be queued for sending at
  // (time at next "hidden" + x) ms, regardless of whether or not the page
  // has been discarded yet.
  // If x < 0, it is equivalent to no timeout, i.e. default behavior.
  // The timeout timer will be reset if the page enters visible state again before
  // the timeout timer ends.
```

```

// Defaults to -1.
// TL;DR: A timeout to accelerate request sending, as the duration
// between page "hidden" & page discarded may be very long.
readonly attribute long sendAfterBeingBackgroundedTimeout;

// The following fields are mutable if `pending` is true.
attribute ByteString method;
attribute USVString url;
attribute Headers headers;
attribute ReadableStream? body;
}

```

TODO: Decide if to enforce type safety by providing PendingGetRequest/PendingPostRequest.

Details

fetch()

A PendingRequest object can be passed into `fetch()`, which should be extended to only kick off sending the request on page discard.

PendingRequestInit

```

// For constructing a PendingRequest.
dictionary PendingRequestInit {
  // One of "GET" or "POST".
  ByteString method;
  HeadersInit headers;
  // Can only be set when `method` is "POST". Otherwise, throws TypeError.
  BodyInit? body;
  AbortSignal? signal;

  // See PendingRequest.
  long sendAfterBeingBackgroundedTimeout;
};

```

To limit the scope, PendingRequest's constructor will only accept a subset of RequestInit field values, as a new dictionary `PendingRequestInit`:

- `url`: supported.
- `method`: one of GET or POST.
- `headers`: supported.
- `body`: only supported for POST.
- `credentials`: enforcing `same-origin` to be consistent.
- `cache`: not supported.
- `redirect`: enforcing `follow`.

- `referrer`: enforcing same-origin URL.
- `referrerPolicy`: enforcing `same-origin`.
- `keepalive`: enforcing `true`.
- `integrity`: not supported.
- `priority`: enforcing `auto`;
- `signal`: supported.
- `sendAfterBeingBackgroundedTimeout`: equivalent to `backgroundTimeout` from `PendingBeacon`'s proposal. We think it's critical to support customized early sending when the page is in `BFCached`. See the original [backgroundTimeout](#) discussion.

Request Size Limit

As `keepalive` is enforced to `true`, a `PendingRequest` object has to share the same size limit budget as a regular keepalive request's [one](#): *"the sum of `contentLength` and `inflightKeepaliveBytes` <= 64 KB"*.

There are several options to communicate the error with API users:

1. The `PendingRequest` throws `TypeError` when the budget has exceeded. An existing `PendingRequest` contributes to the size limit budget until it is sent. This error can happen in the following places: (a) Calling `PendingRequest` constructor (b) Updating `PendingRequest.body`. The problem is that malicious sites can easily eat up all budget to block all other usages.
2. The browser forces sending out existing `PendingRequest` objects (FIFO) when the budget has exceeded. The problem is that it's still possible to have accumulated size that exceeds the limit, if requests take too long to send.

TODO: Consider ignoring the size limit if [BackgroundFetch Permission](#) is enabled for the page. But will there be privacy issues as `PendingRequest` is not visible in the download manager?

3P Frame

Permission-Policy to potentially allow in 3p frames.

Examples

Queue a GET beacon that will be sent on page discard

```
const beacon = new PendingRequest("http://example.test", {method: "GET"});
fetch(beacon).then(response => { /* This Promise may never be resolved! */ });
```

Queue a POST beacon that will be sent on page discard

```
const beacon = new PendingRequest("http://example.test", {
  method: "POST",
  body: {"foo": "bar"}
});
fetch(beacon).then(response => { /* This Promise may never be resolved! */ });
```

Ask UA to send out a pending beacon

```
const beacon = new PendingRequest("http://example.test", {method: "GET"});
fetch(beacon).then(response => {
  // Resolved after `sendNow()`.
});
beacon.sendNow();
```

Update beacon data when it's still pending

```
const beacon = new PendingRequest("http://example.test", {method: "POST"});
fetch(beacon);
if (beacon.pending) {
  beacon.body = "new data";
}
```

Cancel a pending beacon

```
const controller = new AbortController();
const beacon = new PendingRequest("http://example.test", {
  method: "POST",
  signal: controller.signal
});
fetch(beacon);
controller.abort();
```

Queue a beacon that will be sent out roughly 1 minute after this page enters `hidden` visibility state

```
const beacon = new PendingRequest("http://example.test", {
  method: "GET",
  sendAfterBeingBackgroundedTimeout: 60000 /* 1 minute */});
fetch(beacon);
```

Queue a beacon that will be sent immediately after this page enters
`hidden` visibility state

```
const beacon = new PendingRequest("http://example.test", {
  method: "GET",
  sendAfterBeingBackgroundedTimeout: 0 /* immediately on "hidden" */});
fetch(beacon).then(response => {
  // Resolved on "hidden".
});
```

PendingRequest VS keepalive Request in "hidden" listener

```
// Executes ONLY in the next "hidden" event.
const beacon = new PendingRequest("http://example.test", {
  method: "GET",
  sendAfterBeingBackgroundedTimeout: 0 /* immediately on "hidden" */});
fetch(beacon);

document.addEventListener("hidden", e => {
  // Repeatedly executes in EVERY "hidden" event.
  fetch(new Request("http://example.test", {method: "GET",
    keepalive: true}));
});
```

Queue a beacon and update its URL if still pending

```
// Queue a beacon that is sent immediately on page "hidden".
let beacon = new PendingRequest("http://example.test", {
  method: "GET",
  sendAfterBeingBackgroundedTimeout: 0 /* immediately on "hidden" */});
fetch(beacon);

// Check if it is sent before updating, as the page can be hidden anytime.
if (beacon.pending) {
  // Update beacon's URL.
  beacon.url = beacon.url + extraParams;
} else {
  // Create a new beacon, as the previous one is gone.
  beacon = new PendingRequest(beacon.url + extraParams, {
    method: "GET",
    sendAfterBeingBackgroundedTimeout: 0});
  fetch(beacon);
}
```

```
// Send it out immediately instead. (no-op if already sent by browser)
beacon.sendNow();
```