

Page Lifecycle API



By Philip Walton

Engineer at Google working on the Web Platform

Modern browsers today will sometimes suspend pages or discard them entirely when system resources are constrained. In the future, browsers want to do this proactively, so they consume less power and memory. The Page Lifecycle API, shipping in Chrome 68, provides lifecycle hooks so your pages can safely handle these browser interventions without affecting the user experience. Take a look at the API to see whether you should be implementing these features in your application.

Background

Application lifecycle is a key way that modern operating systems manage resources. On Android, iOS, and recently Windows, apps can be started and stopped at any time by the OS. This allows these platforms to streamline and reallocate resources where they best benefit the user.

On the web, there has historically been no such lifecycle, and apps can be kept alive indefinitely. With large numbers of web pages running, critical system resources such as memory, CPU, battery, and network can be oversubscribed, leading to a bad end-user experience.

While the web platform has long had events that related to lifecycle states — like load, unload, and visibilitychange — these events only allow developers to respond to user-initiated lifecycle state changes. For the web to work reliably on low-powered devices (and be more resource conscious in general on all platforms) browsers need a way to proactively reclaim and re-allocate system resources.

In fact, browsers today already do take active measures to conserve resources for pages in background tabs, and many browsers (especially Chrome) would like to do a lot more of this — to lessen their overall resource footprint.

The problem is developers currently have no way to prepare for these types of system-initiated interventions or even know that they're happening. This means browsers need to be conservative or risk breaking web pages.

The Page Lifecycle API attempts to solve this problem by:

- Introducing and standardizing the concept of lifecycle states on the web.
- Defining new, system-initiated states that allow browsers to limit the resources that can be consumed by inactive tabs.
- Creating new APIs and events that allow web developers to respond to transitions to and from these new system-initiated states.

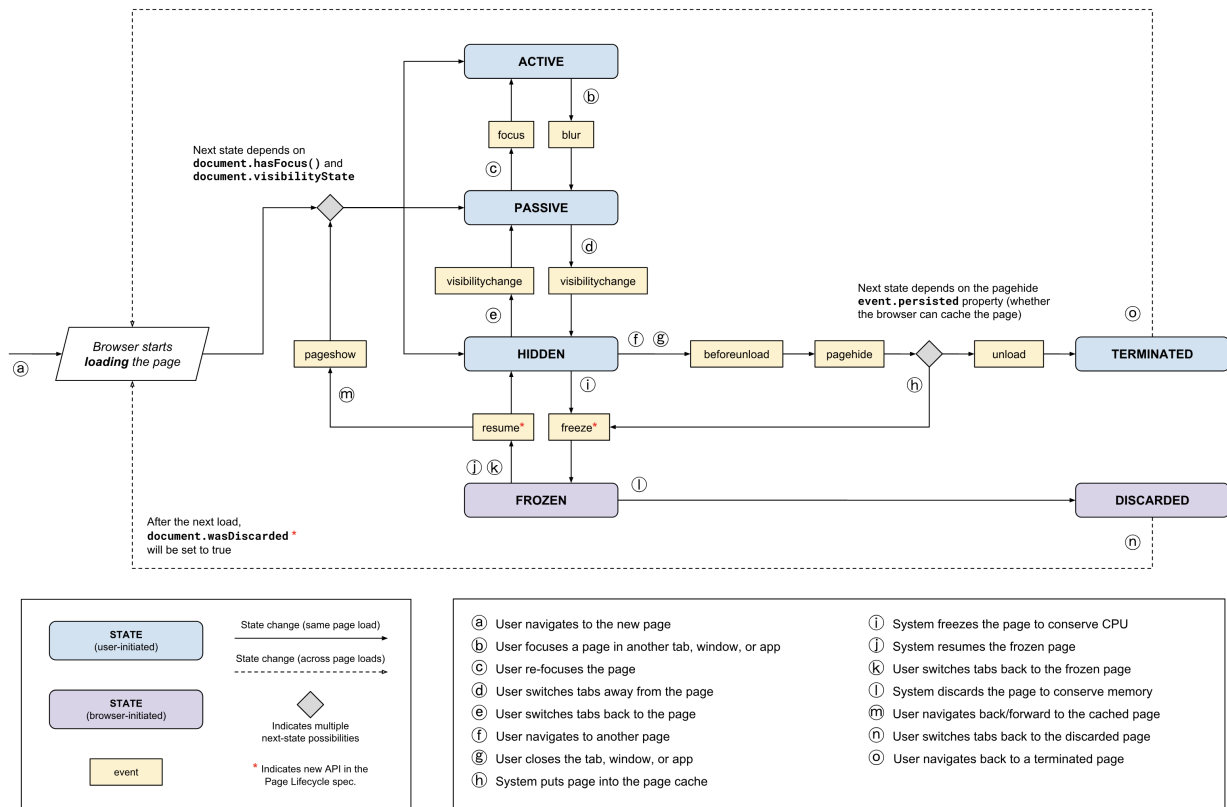
This solution provides the predictability web developers need to build applications resilient to system interventions, and it allows browsers to more aggressively optimize system resources, ultimately benefiting all web users.

The rest of this post will introduce the new Page Lifecycle features shipping in Chrome 68 and explore how they relate to all the existing web platform states and events. It will also give recommendations and best-practices for the types of work developers should (and should not) be doing in each state.

Overview of Page Lifecycle states and events

All Page Lifecycle states are discrete and mutually exclusive, meaning a page can only be in one state at a time. And most changes to a page's lifecycle state are generally observable via DOM events (see [developer recommendations for each state](#) for the exceptions).

Perhaps the easiest way to explain the Page Lifecycle states — as well as the events that signal transitions between them — is with a diagram:



States

The following table explains each state in detail. It also lists the possible states that can come before and after as well as the events developers can use to observe changes.

State	Description
Active	A page is in the <i>active</i> state if it is visible and has input focus. Possible previous states: <u>passive</u> (via the <u>focus</u> event) Possible next states: <u>passive</u> (via the <u>blur</u> event)
Passive	A page is in the <i>passive</i> state if it is visible and does not have input focus. Possible previous states: <u>active</u> (via the <u>blur</u> event) <u>hidden</u> (via the <u>visibilitychange</u> event) Possible next states: <u>active</u> (via the <u>focus</u> event) <u>hidden</u> (via the <u>visibilitychange</u> event)

Hidden A page is in the *hidden* state if it is not visible and has not been frozen.

Possible previous states:

passive (via the [visibilitychange](#) event)

Possible next states:

passive (via the [visibilitychange](#) event)

frozen (via the [freeze](#) event)

terminated (via the [pagehide](#) event)

Frozen In the *frozen* state the browser suspends execution of freezable tasks in the page's task queues until the page is unfrozen. This means things like JavaScript timers and request callbacks do not run. Already-running tasks can finish (most importantly the [freeze](#) callback), but they may be limited in what they can do and how long they can run.

Browsers freeze pages as a way to preserve CPU/battery/data; they also do it as a way to enable faster back/forward navigations — avoiding the need for a full page reload.

Possible previous states:

hidden (via the [freeze](#) event)

Possible next states:

active (via the [resume](#) event, then the [pageshow](#) event)

passive (via the [resume](#) event, then the [pageshow](#) event)

hidden (via the [resume](#) event)

Terminated A page is in the *terminated* state once it has started being unloaded and cleared from memory by the browser. No new tasks can start in this state, and in-progress tasks may be killed if they run too long.

Possible previous states:

hidden (via the [pagehide](#) event)

Possible next states:

NONE

Discarded A page is in the *discarded* state when it is unloaded by the browser in order to conserve resources. No tasks, event callbacks, or JavaScript of any kind can run in this state, as discards typically occur under resource constraints, where starting new processes is impossible.

In the discarded state the tab itself (including the tab title and favicon) is usually visible to the user even though the page is gone.

Possible previous states:



frozen (no events fired)

Possible next states:

NONE

Events

Browsers dispatch a lot of events, but only a small portion of them signal a possible change in Page Lifecycle state. The table below outlines all events that pertain to lifecycle and lists what states they may transition to and from.

Name	Details
<u>focus</u>	<p>A DOM element has received focus.</p> <p> Note: A focus event does not necessarily imply a state change. In many cases a focus event means a user is switching focus from one form element to another.</p> <p>Possible previous states: <u>passive</u></p> <p>Possible current states: <u>active</u></p>
<u>blur</u>	<p>A DOM element has lost focus.</p> <p> Note: A blur event does not necessarily imply a state change. In many cases a blur event means a user is switching focus from one form element to another.</p> <p>Possible previous states: <u>active</u></p> <p>Possible current states: <u>passive</u></p>
<u>visibilitychange</u>	<p>The document's visibilityState value has changed. This can happen when a user navigates to a new page, switches tabs, closes a tab, minimizes or closes the browser, or switches apps on mobile operating systems.</p> <p>Possible previous states: <u>passive</u> <u>hidden</u></p> <p>Possible current states: <u>passive</u> <u>hidden</u></p>
<u>freeze</u> *	<p>The page has just been frozen. Any <u>freezable</u> task in the page's task queues will not be started.</p> <p>Possible previous states: <u>hidden</u></p>

Possible current states:

frozen

resume*

The browser has resumed a frozen page.

Possible previous states:

frozen

Possible current states:

active (if followed by the [pageshow](#) event)

passive (if followed by the [pageshow](#) event)

hidden

pageshow

A session history entry is being traversed to.

This could be either a brand new page load or a page taken from the [page navigation cache](#). If the page was taken from the page navigation cache, the event's **persisted** property is **true**, otherwise it is **false**.

Possible previous states:

frozen (a [resume](#) event would have also fired)

Possible current states:

active

passive

hidden

pagehide

A session history entry is being traversed from.

If the user is navigating to another page and the browser is able to add the current page to the [page navigation cache](#) to be reused later, the event's **persisted** property is **true**. When **true**, the page is entering the frozen state, otherwise it is entering the terminated state.

Possible previous states:

hidden

Possible current states:

frozen (`event.persisted` is **true**, [freeze](#) event follows)

terminated (`event.persisted` is **false**, [unload](#) event follows)

beforeunload

The window, the document and its resources are about to be unloaded. The document is still visible and the event is still cancelable at this point.



Warning: The **beforeunload** event should only be used to alert the user of unsaved changes. Once those changes are saved, the event should be removed. It should never be added unconditionally to the page, as doing so can hurt performance in some cases. See the [legacy APIs section](#) for details.

Possible previous states:

hidden

Possible current states:

terminated

unload

The page is being unloaded.



Warning: Using the **unload** event is never recommended because it's unreliable and can hurt performance in some cases. See the [legacy APIs section](#) for more details.

Possible previous states:

hidden

Possible current states:

terminated

** Indicates a new event defined by the Page Lifecycle API*

New features added in Chrome 68

The chart above shows two states that are system-initiated rather than user-initiated: frozen and discarded. As mentioned above, browsers today already occasionally freeze and discard hidden tabs (at their discretion), but developers have no way of knowing when this is happening.

In Chrome 68, developers can now observe when a hidden tab is frozen and unfrozen by listening for the freeze and resume events on `document`.

```
document.addEventListener('freeze', (event) => {  
  // The page is now frozen.  
});
```



```
document.addEventListener('resume', (event) => {  
  // The page has been unfrozen.  
});
```

In Chrome 68 the `document` object also now includes a wasDiscarded property. To determine whether a page was discarded while in a hidden tab, you can inspect the value of this property at page load time (note: discarded pages must be reloaded to use again).

```
if (document.wasDiscarded) {  
  // Page was previously discarded by the browser while in a hidden tab.  
}
```



For advice on what things are important to do in the **freeze** and **resume** events, as well as how to handle and prepare for pages being discarded, see [developer recommendations for each state](#).

The next several sections offer an overview of how these new features fit into the existing web platform states and events.

Observing Page Lifecycle states in code

In the [active](#), [passive](#), and [hidden](#) states, it's possible to run JavaScript code that determines the current Page Lifecycle state from existing web platform APIs.

```
const getState = () => {  
  if (document.visibilityState === 'hidden') {  
    return 'hidden';  
  }  
  if (document.hasFocus()) {  
    return 'active';  
  }  
  return 'passive';  
};
```



The [frozen](#) and [terminated](#) states, on the other hand, can only be detected in their respective event listener ([freeze](#) and [pagehide](#)) as the state is changing.

Observing state changes

Building on the `getState()` function defined above, you can observe all Page Lifecycle state changes with the following code.

```
// Stores the initial state using the `getState()` function (defined above)  
let state = getState();  
  
// Accepts a next state and, if there's been a state change, logs the  
// change to the console. It also updates the `state` value defined above.  
const logStateChange = (nextState) => {  
  const prevState = state;  
  if (nextState !== prevState) {  
    console.log(`State change: ${prevState} >>> ${nextState}`);  
    state = nextState;  
  }  
};
```




```
// These lifecycle events can all use the same listener to observe state
// changes (they call the `getState()` function to determine the next state).
['pageshow', 'focus', 'blur', 'visibilitychange', 'resume'].forEach((type) => {
  window.addEventListener(type, () => logStateChange(getState()), {capture: true}
});

// The next two listeners, on the other hand, can determine the next
// state from the event itself.
window.addEventListener('freeze', () => {
  // In the freeze event, the next state is always frozen.
  logStateChange('frozen');
}, {capture: true});

window.addEventListener('pagehide', (event) => {
  if (event.persisted) {
    // If the event's persisted property is `true` the page is about
    // to enter the page navigation cache, which is also in the frozen state.
    logStateChange('frozen');
  } else {
    // If the event's persisted property is not `true` the page is
    // about to be unloaded.
    logStateChange('terminated');
  }
}, {capture: true});
```

The above code does three things:

1. Sets the initial state using the `getState()` function.
2. Defines a function that accepts a next state and, if there's a change, logs the state changes to the console.
3. Adds capturing event listeners for all necessary lifecycle events, which in turn call `logStateChange()`, passing in the next state.

Warning! This code yields different results in different browsers, as the order (and reliability) of events has not been consistently implemented. To learn how best to handle these inconsistencies see [managing cross-browsers differences](#).

One thing to note about the above code is that all the event listeners are added to `window` and they all pass `{capture: true}`. There are three reasons for this:

- Not all Page Lifecycle events have the same target. `pagehide`, and `pageshow` are fired on `window`; `visibilitychange`, `'freeze'`, and `'resume'` are fired on `document`, and `'focus'` and `'blur'` are fired on their respective DOM elements.

- Most of these events do not bubble, which means it's impossible to add non-capturing event listeners to a common ancestor element and observe all of them.
- The capture phase executes before the target or bubble phases, so adding listeners there helps ensure they run before other code can cancel them.

Managing cross-browsers differences

The chart in the beginning of this article outlines the state and event flow according to the Page Lifecycle API. But since this API has just been introduced, the new events and DOM APIs have not been implemented in all browsers.

Furthermore, the events that are implemented in all browsers today are not implemented consistently. For example:

- Some browsers do not fire a `blur` event when switching tabs. This means (contrary to the diagram and tables above) a page could go from the active state to the hidden state without going through passive first.
- Several browsers implement a page navigation cache, and the Page Lifecycle API classifies cached pages as being in the frozen state. Since this API is brand new, these browsers do not yet implement the `freeze` and `resume` events, though this state can still be observed via the `pagehide` and `pageshow` events.
- Older versions of Internet Explorer (10 and below) do not implement the `visibilitychange` event.
- The dispatch order of the `pagehide` and `visibilitychange` events has changed. Previously browsers would dispatch `visibilitychange` after `pagehide` if the page's visibility state was visible when the page was being unloaded. New Chrome versions will dispatch `visibilitychange` before `pagehide`, regardless of the document's visibility state at unload time.

To make it easier for developers to deal with these cross-browsers inconsistencies and focus solely on following the lifecycle state recommendations and best practices, we've released PageLifecycle.js, a JavaScript library for observing Page Lifecycle API state changes.

PageLifecycle.js normalizes cross-browser differences in event firing order so that state changes always occur exactly as outlined in the chart and tables in this article (and do so consistently in all browsers).

Developer recommendations for each state

As developers, it's important to both understand Page Lifecycle states *and* know how to observe them in code because the type of work you should (and should not) be doing depends largely on what state your page is in.

For example, it clearly doesn't make sense to display a transient notification to the user if the page is in the hidden state. While this example is pretty obvious, there are other recommendations that aren't so obvious that are worth enumerating.

State	Developer recommendations
Active	<p>The active state is the most critical time for the user and thus the most important time for your page to be <u>responsive to user input</u>.</p> <p>Any non-UI work that may block the main thread should be deprioritized to <u>idle periods</u> or <u>offloaded to a web worker</u>.</p>
Passive	<p>In the passive state the user is not interacting with the page, but they can still see it. This means UI updates and animations should still be smooth, but the timing of when these updates occur is less critical.</p> <p>When the page changes from <i>active</i> to <i>passive</i>, it's a good time to persist unsaved application state.</p>
Hidden	<p>When the page changes from <i>passive</i> to <i>hidden</i>, it's possible the user will not interact with it again until it's reloaded.</p> <p>The transition to <i>hidden</i> is also often the last state change that's reliably observable by developers (this is especially true on mobile, as users can close tabs or the browser app itself, and the beforeunload, pagehide, and unload events are not fired in those cases).</p> <p>This means you should treat the hidden state as the likely end to the user's session. In other words, persist any unsaved application state and send any unsent analytics data.</p> <p>You should also stop making UI updates (since they won't be seen by the user), and you should stop any tasks that a user wouldn't want running in the background.</p>
Frozen	<p>In the frozen state, <u>freezable tasks</u> in the <u>task queues</u> are suspended until the page is unfrozen — which may never happen (e.g. if the page is discarded).</p> <p>This means when the page changes from <i>hidden</i> to <i>frozen</i> it's essential that you stop any timers or tear down any connections that, if frozen, could affect other open tabs in the same origin, or affect the browser's ability to put the page in the <u>page navigation cache</u>.</p> <p>In particular, it's important that you:</p> <ul style="list-style-type: none">• Close all open <u>IndexedDB</u> connections.• Close open <u>BroadcastChannel</u> connections.• Close active <u>WebRTC</u> connections.

- Stop any network polling or close any open Web Socket connections.
- Release any held Web Locks.

You should also persist any dynamic view state (e.g. scroll position in an infinite list view) to sessionStorage (or IndexedDB via commit()) that you'd want restored if the page were discarded and reloaded later.

If the page transitions from *frozen* back to *hidden*, you can reopen any closed connections or restart any polling you stopped when the page was initially frozen.

Terminated You generally do not need to take any action when a page transitions to the *terminated* state.

Since pages being unloaded as a result of user action always go through the *hidden* state before entering the *terminated* state, the *hidden* state is where session-ending logic (e.g. persisting application state and reporting to analytics) should be performed.

Also (as mentioned in the recommendations for the hidden state), it's very important for developers to realize that the transition to the terminated state cannot be reliably detected in many cases (especially on mobile), so developers who depend on termination events (e.g. beforeunload, pagehide, and unload) are likely losing data.

Discarded The *discarded* state is not observable by developers at the time a page is being discarded. This is because pages are typically discarded under resource constraints, and unfreezing a page just to allow script to run in response to a discard event is simply not possible in most cases.

As a result, you should prepare for the possibility of a discard in the change from *hidden* to *frozen*, and then you can react to the restoration of a discarded page at page load time by checking `document.wasDiscarded`.

Legacy lifecycle APIs to avoid

The unload event

Key point: Never use the unload event on modern browsers.

Many developers treat the unload event as a guaranteed callback and use it as an end-of-session signal to save state and send analytics data, but doing this is **extremely unreliable**, especially on mobile! The unload event does not fire in many typical unload situations, including closing a tab from the tab switcher on mobile or closing the browser app from the app switcher.

For this reason, it's always better to rely on the visibilitychange event to determine when a session ends, and consider the hidden state the last reliable time to save app and user data.

Furthermore, the mere presence of a registered `unload` event handler (via either `onunload` or `addEventListener()`) can prevent browsers from being able to put pages in the page navigation cache for faster back and forward loads.

In all modern browsers (including IE11), it's recommended to always use the pagehide event to detect possible page unloads (a.k.a the terminated state) rather than the `unload` event. If you need to support Internet Explorer versions 10 and lower, you should feature detect the `pagehide` event and only use `unload` if the browser doesn't support `pagehide`:

```
const terminationEvent = 'onpagehide' in self ? 'pagehide' : 'unload';
```



```
addEventListener(terminationEvent, (event) => {  
  // Note: if the browser is able to cache the page, `event.persisted`  
  // is `true`, and the state is frozen rather than terminated.  
}, {capture: true});
```

For more information on page navigation caches, and why the `unload` event harms them, see:

- [WebKit Page Cache](#)
- [Firefox Page Cache](#)

The `beforeunload` event

Key point: Never add a `beforeunload` listener unconditionally or use it as an end-of-session signal. Only add it when a user has unsaved work, and remove it as soon as that work has been saved.

The `beforeunload` event has a similar problem to the `unload` event, in that when present it prevents browsers from caching the page in their page navigation cache.

The difference between `beforeunload` and `unload`, though, is that there are legitimate uses of `beforeunload`. For instance, when you want to warn the user that they have unsaved changes they'll lose if they continue unloading the page.

Since there are valid reasons to use `beforeunload` but using it prevents pages from being added to the page navigation cache, it's recommended that you *only* add `beforeunload` listeners when a user has unsaved changes and then remove them immediately after the unsaved changes are saved.

In other words, don't do this (since it adds a `beforeunload` listener unconditionally):

```
addEventListener('beforeunload', (event) => {  
  // A function that returns `true` if the page has unsaved changes.  
  if (pageHasUnsavedChanges()) {  
    event.preventDefault();  
    return event.returnValue = 'Are you sure you want to exit?';  
  }  
}, {capture: true});
```



Instead do this (since it only adds the `beforeunload` listener when it's needed, and removes it when it's not):

```
const beforeUnloadListener = (event) => {  
  event.preventDefault();  
  return event.returnValue = 'Are you sure you want to exit?';  
};  
  
// A function that invokes a callback when the page has unsaved changes.  
onPageHasUnsavedChanges(() => {  
  addEventListener('beforeunload', beforeUnloadListener, {capture: true});  
});  
  
// A function that invokes a callback when the page's unsaved changes are resolved  
onPageHasUnsavedChanges(() => {  
  removeEventListener('beforeunload', beforeUnloadListener, {capture: true});  
});
```



Note: The [PageLifecycle.js](#) library provides the convenience methods `addUnsavedChanges()` and `removeUnsavedChanges()`, that follow all the best practices outlined above. They're based on an early proposal to officially replace the `beforeunload` event with a declarative API that can be less easily abused and more reliable on mobile platforms.

If you want to use the `beforeunload` event properly and in a way that works cross-browser, the [PageLifecycle.js](#) library is our recommended solution.

FAQs

My page does important work when it's hidden, how can I stop it from being frozen or discarded?

There are lots of legitimate reasons web pages shouldn't be frozen while running in the hidden state. The most obvious example here is an app that plays music.

There are also situations where it would be risky for Chrome to discard a page, like if it contains a form with unsubmitted user input, or if it has a `beforeunload` handler that warns when the page is unloading.

For the moment, Chrome is going to be conservative when discarding pages and only do so when it's confident it won't affect users. For example, pages that have been observed to do any of the following while in the background will not be discarded unless under extreme resource constraints:

- Playing audio
- Using WebRTC
- Updating the tab title or favicon
- Showing alerts
- Sending push notifications

Note: In the case of pages that update the title or favicon to alert users of unread notifications, we currently have a [proposal to enable these kinds of updates from service worker](#), which would allow Chrome to freeze or discard the page but still show changes to the tab title or favicon.

What is the page navigation cache?

The page navigation cache is a general term used to describe a navigation optimization some browsers implement that makes using the back and forward buttons faster. Webkit calls it the [Page Cache](#) and Firefox calls it the [Back-Forwards Cache](#) (or bfcache for short).

When a user navigates away from a page, these browsers freeze a version of that page so that it can be quickly resumed in case the user navigates back using the back or forward buttons. Remember that adding a `beforeunload` or `unload` event handler prevents this optimization from being possible.

For all intents and purposes, this freezing is functionally the same as the freezing browsers perform to conserve CPU/battery; for that reason it's considered part of the [frozen](#) lifecycle state.

Why aren't the `load` or `DOMContentLoaded` events mentioned?

The Page Lifecycle API defines states to be distinct and mutually exclusive. Since a page can be loaded in either the active, passive, or hidden state, a distinct loading state does not make sense, and since the `load` and `DOMContentLoaded` events don't signal a lifecycle state change, they're not relevant to this discussion.

If I can't run asynchronous APIs in the frozen or terminated states, how can I save data to IndexedDB?

In frozen and terminated states, freezable tasks in a page's task queues are suspended, which means asynchronous and callback-based APIs such as IndexedDB cannot be reliably used.

In the future, we will add a `commit()` method to `IDBTransaction` objects, which will give developers a way to perform what are effectively write-only transactions that don't require callbacks. In other words, if the developer is just writing data to IndexedDB and not performing a complex transaction consisting of reads and writes, the `commit()` method will be able to finish before task queues are suspended (assuming the IndexedDB database is already open).

For code that needs to work today, however, developers have two options:

- **Use Session Storage:** Session Storage is synchronous and is persisted across page discards.
- **Use IndexedDB from your service worker:** a service worker can store data in IndexedDB after the page has been terminated or discarded. In the `freeze` or `pagehide` event listener you can send data to your service worker via `postMessage()`, and the service worker can handle saving the data.

Note: While option #2 above will work, it's not ideal in situations where the device is freezing or discarding the page due to memory pressure since the browser may have to wake up the service worker process, which will put more strain on the system.

Testing your app in the frozen and discarded states

To test how your app behaves in the frozen and discarded states, you can visit <chrome://discards> to actually freeze or discard any of your open tabs.

Discards

[Discard a tab now] [Urgent discard a tab now]

Utility Rank	Reactivation Score	Site Engagement Score	Tab Title	Tab URL	Visibility	Loading State	Lifecycle State	Can freeze?	Can discard?	Discard Count	Auto Discardable	Last Active	Actions
1	1.5781	8.1	Example Domain	https://example.com/	hidden	loaded	active	✓ [View Reason]	✓ [View Reason]	0	✓ [Toggle]	just now	[Load] [Freeze] [Discard] [Urgent Discard]
2	N/A	0.0	Discards	chrome://discards/	visible	loaded	active	✗ [View Reason]	✗ [View Reason]	0	✓ [Toggle]	just now	[Load] [Freeze] [Discard] [Urgent Discard]

This allows you to ensure your page correctly handles the `freeze` and `resume` events as well as the `document.wasDiscarded` flag when pages are reloaded after a discard.

Summary

Developers who want to respect the system resources of their user's devices should build their apps with Page Lifecycle states in mind. It's critical that web pages are not consuming excessive system resources in situations that the user wouldn't expect

In addition, the more developers start implementing the new Page Lifecycle APIs, the safer it will be for browsers to freeze and discard pages that aren't being used. This means browsers will consume less memory, CPU, battery, and network resources, which is a win for users.

Lastly, developers who want to implement the best practices described in this article but don't want to memorize all the possible state and event transitions paths can use PageLifecycle.js to easily observe lifecycle state changes consistently in all browsers.

Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](#), and code samples are licensed under the [Apache 2.0 License](#). For details, see our [Site Policies](#). Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 24, 2018.