Introduction to the Budget API

The Push Messaging API enables us to send notifications to a user even when the browser is closed. Many developers want to be able to use this messaging to update and synchronize content without the browser being open, but the API has one important restriction: you must always display a notification for every single push message received.

Being able to send a push message to synchronize data on a user's device or hide a notification you had previously shown can be extremely useful for users and developers, but allowing a web app to do work in the background without the user knowing is open to abuse.

The Budget API, is a new API designed to allow developers to perform limited background work without notifying the user, such as silent push or performing a background fetch. In Chrome 60 and above you'll be able to start using this API and the Chrome team is eager to get feedback from developers.

To allow developers to consume a user's resources in the background, the web platform is introducing the concept of a budget using the new Budget API. Each site will be awarded an amount of resource based on user engagement that they can consume for background actions, such as a silent push, where each operation will deplete the budget. When the budget is spent, background actions can no longer be performed without user visibility. The user agent will be responsible for determining budget assigned to a web app based on it's heuristics, for example budget allowance could be linked to user engagement. Each browser can decide it's own heuristic.

TL;DR The Budget API allows to you to reserve budget, use budget, get a list of remaining budget and understand the cost of background operations

Reserving Budget

In Chrome 60 and above, the `navigator.budget.reserve()` method will be available without any flags.

The `reserve()` method allows you to request budget for a specific operation and it'll return a boolean to indicate if the budget can be reserved. If the budget was reserved, there is no need to notify the user of your background work.

In the example of push notifications, you can attempt to reserve budget for a "silent-push" operation and if `reserve()` resolves with true, the operation is allowed. Otherwise it'll return false and you'll need to show a notification

```
self.addEventListener('push', event => {
 const promiseChain = navigator.budget.reserve('silent-push')
   .then((reserved) => {
     if (reserved) {
       // No need to show a notification.
       return;
     }

     // Not enough budget is available, must show a notification.
     return registration.showNotification(...);
   });
 event.waitUntil(promiseChain);
});
```

In Chrome 60, 'silent-push' is the only operation type that is available, but you can find a full list of operation types in the spec. There is also no easy way to increase your budget for testing or debugging purposes once it's used, but as a temporary workaround you can create a new profile in Chrome. Sadly you can't use incognito for this either as the Budget API will return a budget of zero in Incognito (although there is a bug that results in an error during my testing).

You should only call `reserve()` when you intend to perform the operation you are reserving at some point in the future. Note that if you called reserve in the above example but still showed a notification, the budget will still be used.

One common use case that isn't enabled by `reserve()` alone, is the ability to schedule a silent push from a backend. The Budget API does have API's to enable this use case but they are still being worked on in Chrome and are currently only available behind flags and / or an Origin Trial.

Budget API and Origin Trials

There are two methods, `getBudget()` and `getCost()`, that can be used by a web app to plan the usage of their budget.

In Chrome 60, both of these methods are available if you sign up for the origin trial but otherwise for testing you can use them locally by enabling the Experimental Web Platform features flag (Open chrome://flags/#enable-experimental-web-platform-features in Chrome).

Let's look how to use these APIs.

Get your Budget

You can find your available budget with the `getBudget()` method. Some browsers (like Chrome) will have budget 'decay' over time, so to give you full visibility this returns an array of `BudgetStates`, indicating what your budget will be at various times in the future.

To list the budget entries we can run:

```
navigator.budget.getBudget()
.then((budgets) => {
  budgets.forEach((element) => {
    console.log(\`At '${new Date(element.time).toString()}' \` +
      \`your budget will be '${element.budgetAt}'.\`);
  });
});
```

The first entry will be your current budget and additional values will show what your budget will be at various points in the future.

```
At 'Mon Jun 05 2017 12:47:20' you will have a budget of '3'.
At 'Fri Jun 09 2017 10:42:57' you will have a budget of '2'.
At 'Fri Jun 09 2017 12:31:09' you will have a budget of '1'.
```

One of the benefits of including future budget allowances is that developers can share this information with their backend to adapt their server side behavior (i.e. only send a push message to trigger an update when the client has budget for a silent push).

Note: At the time of writing there is a bug where the budget can decrease to a negative value. Your budget should never go below zero.

Get the Cost of an Operation

To find out how much an operation will cost, calling `getCost()` will return a number indicating the maximum amount of budget that will be consumed if you call `reserve()` for that operation.

For example, we can find out the cost of not showing a notification when you receive a push message (i.e. the cost of a silent push), with the following code:

```
navigator.budget.getCost('silent-push')
.then((cost) => {
  console.log('Cost of silent push is:', cost);
})
.catch((err) => {
  console.error('Unable to get cost:', err);
});
```

At the time of writing, Chrome 60 will print:

```
Cost of silent push is: 2
```

One thing to highlight with the `reserve()` and `getCost()` methods is that the actual cost of an operation can **be less than** the cost returned by `getCost()`. You may still be able to reserve an operation if your current budget is less than the indicated cost. The specific details from the spec are as follows:

> The reserved cost of certain background operations could be less than the cost indicated by getCost() when the user's device is in favorable conditions, for example because it's not on battery power.

Note: If you pass in an operation type that the browser does not support or is invalid, the promise will reject with a `TypeError`.

That's the current API in Chrome and as the web continues to support new API's that require the ability to perform background work, like background fetch, the Budget API can be used to manage the number of operations you can perform without notifying the user.

As you use the API please provide feedback on the GitHub Repo or file Chrome bugs at crbug.com.

Last updated July 2, 2018.