Persistent Web Notification storage database

Author: Peter Beverloo <peter@chromium.org>; **Date**: March 5th, 2015. **Public**.

This document describes the design for a database for storing data associated with persistent Web Notifications on disk. Implementation is tracked in Issue 447628.

Problem statement

Persistent notifications are inherently more powerful than non-persistent ones: they can outlive the page they were created by, carry arbitrary data payloads specified by the Web developer and can be retrieved at any time from any page on the given origin.

While we *could* store persistent notification data in memory on desktop, the ability for authors to specify their own data payload means this can have unwanted effects on memory usage.

On Android, notifications can outlive the browser process entirely, so their data needs to be serialized somewhere. We currently do this in a Pickle stored in the notification's content intent, but this does not scale well.

Proposed implementation

The core part of the implementation will be done in //content/browser/notifications/. The following diagrams illustrate the ownership and threading model. Note that the entire implementation lives in the browser process.

Explanation of the important classes

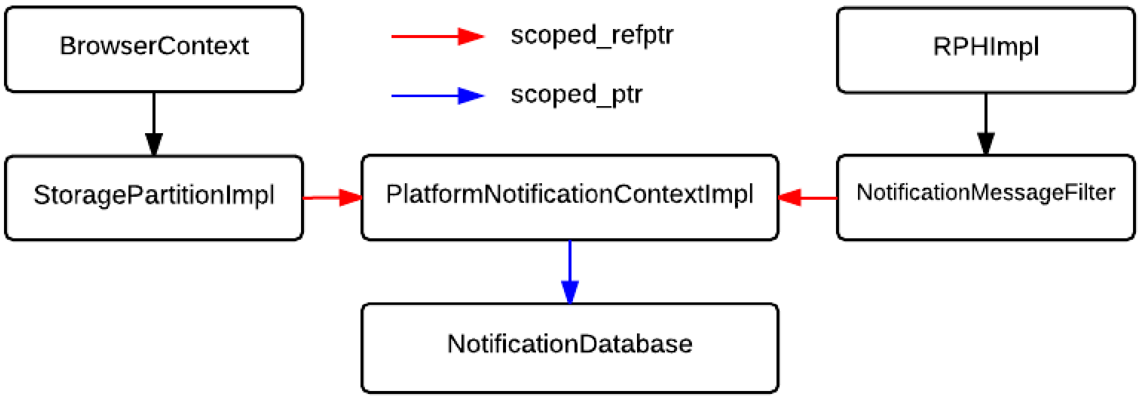
The `PlatformNotificationContextImpl` (implements the `PlatformNotificationContext` //content/ API) is the highest level interface which consumers can use to read, write or delete data using the notification database. It's owned by the `StoragePartitionImpl`. All methods for interacting with the database must be used on the IO thread.

It will own an instance of the `NotificationDatabase` class, which will be lazily initialized.

The `NotificationDatabase` class is responsible for actually creating and working with the LevelDB database. It must be noted that such operations will only happen using a SequencedTaskRunner on a thread assigned to us by BrowserThread's blocking pool, given that file I/O operations will end up being too expensive for other threads.

The `NotificationDatabase` class will also manage conversions between the input data, the protocol buffer and the serialized version of said buffer.

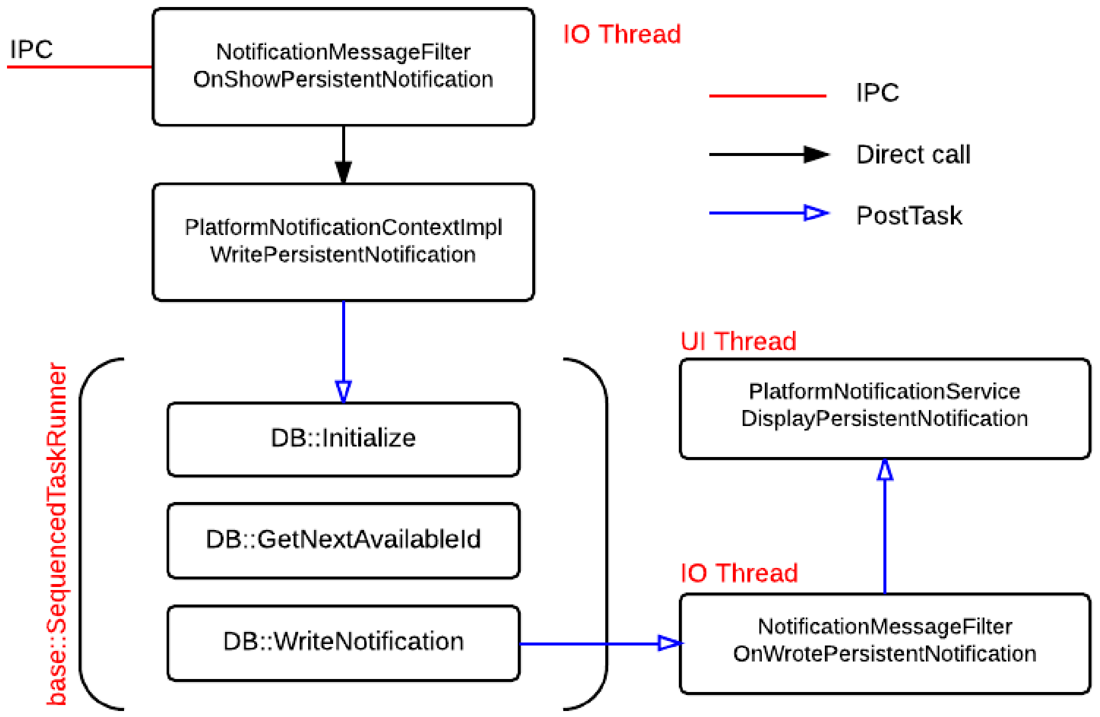
Ownership diagram



The `StoragePartitionImpl` will create the `PlatformNotificationContextImpl` instance on the UI thread. It's exposed as part of the content public `StoragePartition` interface.

When the `RenderProcessHostImpl` starts the `NotificationMessageFilter`, it will pass in the instance of the `PlatformNotificationContextImpl` retrieved from the storage partition.

Threading diagram: creating a persistent notification


IPC
NotificationMessageFilter OnShowPersistentNotification
IO Thread
IPC
Direct call
PostTask
PlatformNotificationContextImpl WritePersistentNotification
base::SequencedTaskRunner
DB::Initialize
DB::GetNextAvailableId
DB::WriteNotification
UI Thread
PlatformNotificationService DisplayPersistentNotification
IO Thread
NotificationMessageFilter OnWrotePersistentNotification

Today, when the browser process receives an IPC message requesting display of a persistent notification, we directly use the `PlatformNotificationService` on the UI thread.

In the new situation, the IPC message will be received on the IO thread. The `PlatformNotificationContextImpl` will use a sequenced task runner (retrieved from the BrowserThread's blocking pool) to initialize the database, get the next available notification Id and then write the notification's data, using that Id, to the database.

When this is done, it will post a task back to the IO thread (to have a consistent `PlatformNotificationContextImpl` API), which then tells the `PlatformNotificationService` on the UI thread to display the notification.

A similar version of this diagram can be imagined for closing notifications, where data would be removed from the database rather than added.

Usage of protocol buffers and levelDB

Following the reasoning put forward by the Service Worker team in using the protocol buffers stored in levelDB combination, we too have elected to use this for the notification database.

LevelDB database format

LevelDB is a fast key-value database. The notification database will imitate the format used by the ServiceWorkerDatabase implementation, with the following keys.

> **Key**: `DB_VERSION` (int)
>
> **Key**: `NEXT_NOTIFICATION_ID` (int64)
> Next id to assign to a persistent notification. Ever incrementing for the lifetime of the database, even when old persistent notifications get removed.
>
> **Key**: `INDEX:<origin identifier>:`
> `<service_worker_registration_id>:<notification_id>`(empty)
> Used as an index to find all notification ids associated with a given service worker registration id.
>
> **Key**: `DATA:<origin identifier>:<notification_id>` (string)
> Stores the serialized representation of a NotificationDatabaseData protobuf.

The protocol buffer stored in REG provides flexibility over the data stored in the database without having to upgrade the database itself to a new version.

As there is no known use-case for storing user data associated with Web Notifications, we'll consider this out of scope for the first version of the database.

Protocol buffer format

The following protocol buffer format is suggested to be used for the first iteration.

```
message NotificationDatabaseData {
  required int64 notification_id;

  optional string origin;
  optional int64 service_worker_registration_id;

  enum Direction {
    LEFT_TO_RIGHT = 0;
    RIGHT_TO_LEFT = 1;
  }

  // To be kept synchronized with PlatformNotificationData.
  optional string title;  // Message title.
  optional Direction direction;  // i18n direction.
```

```
    optional string language;  // BCP 47 language.
    optional string body;  // Message body.
    optional string tag;  // Replacement tag.
    optional string icon;  // Icon URL.
    optional string data;  // Author-provided, serialized data.
  }
```

Most fields are marked `optional` in accordance with experiences of Google engineers more familiar with protocol buffers, as noted in the language guide.

Integration with the Quota Manager

The Web Notification database will tie in with the Quota Manager to ensure that (a) notification data storage will be counted for the origin, and (b) we defer to the Quota Manager to determine when data for an origin has to be removed.

To achieve this, the `PlatformNotificationContextImpl` will register a `NotificationQuotaClient` with the quota manager proxy. Furthermore, the `NotificationStorage` class will notify the quota manager proxy when data for an origin gets accessed or modified.

This functionality will be covered with unit tests when the actual database storage is working.

Note that part of the decision to integrate with the quota manager is because authors will have the ability to provide arbitrary data. While a reasonable limit will be sought in implementation of that feature, it's anticipated to be higher than a megabyte.

Integration with Service Worker unregistrations

In order to handle dropped Service Workers, `PlatformNotificationContextImpl` will register itself as a `ServiceWorkerContextObserver` and listen to notifications about unregistered Service Workers, as well as complete data wipe.

Behavior when the database becomes corrupted

The database will not implement any mechanism for attempting recovery of data. Instead, it will blow away the storage and starts over, analogous to Service Worker's `DeleteAndStartOver()` functionality.

Privacy and security considerations

The notification database files will be stored in the user's profile directory:

```
    $PROFILE_DIR/Platform Notifications/
```

The database will not be encrypted or feature any other kind of additional protection, in line with similar features such as the Service Worker database.

The notification database will register a `NotificationQuotaClient` with the quota manager to be aware of requests for removing data for a certain origin. Notifications associated with the origin for which data is being cleared will be closed as well.

All information stored for a notification is provided by the developer, and may be sensitive to privacy. This is no different from storing such data in other forms of local storage, however.

Testing plan for the implementation

The project will largely be covered by the existing test suites for the Web Notification API, which include instrumentation tests[1], browser tests[2] and layout tests[3].

A series of unit tests will be added for the database implementation. They will include:

- Tests for opening, closing and using the database in memory.
- Tests for opening, closing and using the database when written to a file.
- Tests for integration with the quota manager, both for commands created by the manager (getting and clearing data for an origin) and command created by the notification database system (notifying of data modification / access).

Appendix: Features depending on the database

The following features will be implemented on top of this database.

- Implementation of the Notification.get() method, enabling Web developers to get a list of notifications displayed for their origin. Tracked in Issue 442143.

- Implementation of the Notification.data attribute, enabling Web developers to store some data with the notifications they show. Tracked in Issue 442129.

Appendix: Usage by features outside of //content/

The `PlatformNotificationContext` is being defined in the //content/public/ API, exposing minimal functionality, to enable existing users of persistent Web Notifications living higher up.

[1] NotificationUIManagerTest, Android-specific.
[2] PlatformNotificationServiceBrowserTest and NotificationBrowserTest.
[3] LayoutTests/http/tests/notifications/, serviceworkerregistration-*.

The notable example here is the Push API, our implementation of which currently requires the developer to display a notification. If they don't do this, a forced, persistent Web Notification will be created by the `PushMessagingServiceImpl`.

Appendix: Impact on the chrome.notifications extensions API

The chrome.notifications extensions API provides an alternative, Chrome proprietary method of showing notifications, available to Chrome Apps and extensions. This project **will not** impact that API, nor is the database being designed with this API in mind.

It is currently possible for authors to retrieve a list of all their notifications using the `chrome.notifications.getAll()` method. This has been implemented on top of the `NotificationUIManager` interface, which remains unchanged.

It will be possible to update this implementation to use the database in the future, but that is considered out of scope for this project.