Centralised notifications core API

Introduction

Totara 14 introduces a new way of handling notifications throughout the system. Currently, it works in parallel with the legacy notification system for backwards compatibility, but the legacy notification system will be deprecated. Existing notifications will be converted to the new system and no new notifications will be created using the legacy notification subsystem.

This page describes the most important concepts and API's of centralised notifications.

The new notification system called "Centralised Notifications" (CN) has been implemented as the **totara_notification** plugin. Its core source code is located in the **totara/notification** folder and name-spaced as "**totara_notification**".

The main software design principles behind the new system are a centralised approach to handling of notifications, high extensibility and ease of integration.

Notifications are made context aware, which allows overrides of most notification functionality aspects following standard context rules on the API level. In other words, notification messages and settings can be overridden in system, tenant, course and other contexts.

The new notification system is based on notifiable event resolvers (something happened) and configurable responses (notifications) to those events. While events are hard-coded, notifications have default implementations that are shipped with the system, but also can be extended and overridden by administrators and content creators when required.

Interfaces and classes

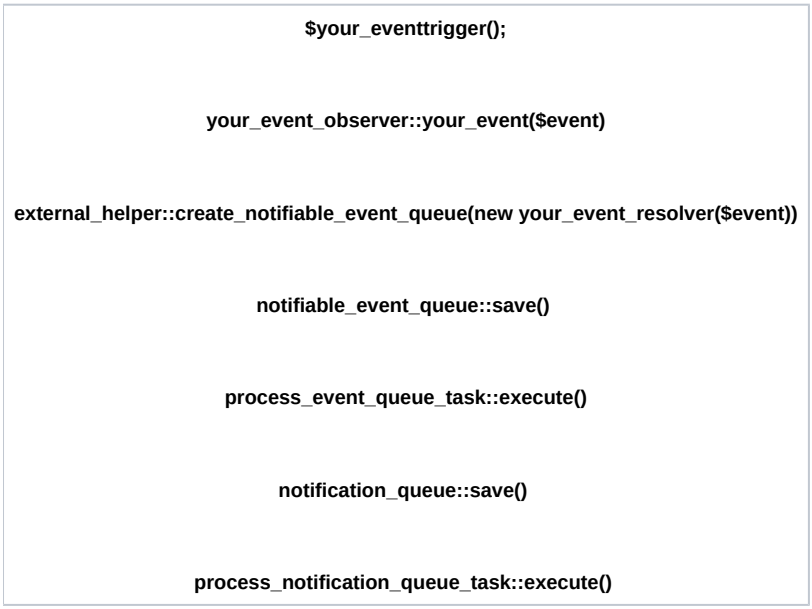
Centralised Notifications core consists of the following parts:

Part	Description
\totara_notification\resolver\notifiable_event_resolver	The notifiable event resolver abstract class is designed to be extended by Totara notification resolver classes to provide information such as notification name, applicable scheduling rules, available recipient types, and other static properties required by Centralised Notifications core processing. Each notifiable event interface needs a corresponding notifiable event resolver. This class manages the data relating to an event that has occurred. It's main purpose is to transform the raw data of the event and answer questions such as "when did the event occur" and "who are the recipient users in relation to this event". Classes extending this class are considered as event parts for which either embedded or custom notifications can be made. This classes must be placed in component's totara_notification\resolver namespace to allow auto-discovery (e.g. namespace your_component\totara_notification\resolver).
totara_notification\resolver\abstraction\additional_criteria_resolver totara_notification\resolver\abstraction\permission_resolver totara_notification\resolver\abstraction\scheduled_event_resolver	These three interfaces can be added to a notifiable event resolver (in any combination) to add additional functionality. • When configuring a notification, a resolver which implements the additional_criteria_resolver interface can add form fields allowing notification admins to configure criteria which must be met before the notification will be sent. • If a plugin has capabilities which specifically grant capability to configure notifications of that type, then they should be specified by implementing the permission_resolver interface. • Notifications that can be sent either before or after the event occurred need to implement the scheduled_event_resolver. The resolver will then be required to implement a function which retrieves all events which occurred between two given points in time. See the documentation inside these interfaces for more details about how they work, when they should be used, and how to use them.
\totara_notification\external_helper::create_notifiable_event_queue()	Helper method to use for notifications triggering. It should be called whenever notifications should be sent. Typically it will be various event observers since notifiable_event_observer constructor requires event data to work.

\totara_notification\entity\notifiable_event_queue	The notifiable events queue stores the information about a notifiable event that needs to be processed into notifications. This is done for performance reasons, since processing events can take a lot of time, so when an event is triggered, it doesn't do much, it just puts the information into a queue. This is entity class for the 'notifiable_event_queue' table.
\totara_notification\notification\built_in_notification	The notification abstract class is used to define built-in notifications which are wired to notifiable events. A notification subclass implements all the default properties of a notification which should be sent when the event occurs. Built-in notifications can be modified, by "overriding" one or more of the default properties, by an administrator through the interface.
\totara_notification\entity\notification_queue	The notifications queue stores information about notifications that were produced during processing of the event queue. Each record in the queue represents one notification which should be sent to some set of recipients, through the configured output channels, at the scheduled time.
\totara_notification\task\process_event_queue_task	The process events queue scheduled task takes records from the notifiable events queue and prepares notification records (one event can lead to several notifications). However, it doesn't send those notifications (this is done by process notifications queue scheduled task).
\totara_notification\task\process_notification_queue_task	The process notifications queue scheduled task takes records to be sent and actually sends them to the chosen recipients, taking into account their settings, admin settings, notification outputs, etc.
\totara_notification\task\process_scheduled_event_task	The process scheduled event scheduled task process all notifications with enabled schedulers. For each notification scheduled on future or past time, it calculates if this notification has any related event that should trigger this notification within timeframe between last run of this task and now, and adds them into notification queue.

Simple notifications workflow

The notifications workflow is schematically described in the the following diagram:



Similarly to usual Totara events workflow, when event is triggered it is passed to observers as defined in db/events.php. Observer processes event as usually as well as instantiating event resolver class using provided event data and passes it as argument to **\totara_notification\external_helper::create_notifiable_event_queue()** method.

Method **\totara_notification\external_helper::create_notifiable_event_queue()** registers resolver class, event data, and provided extended context into **notifiable_event_queue**.

Afterwards, the process events queue scheduled task takes the record from the notifiable event queue, finds all on-time notifications associated with the given notifiable event and adds them into **notification_queue**.

During processing this task will do the following:

1. Get notifiable event record from the queue.
2. Get information about event.
3. Get information about settings.
4. Lookup notifications that should be created for this event.
5. Create all notifications records (one record per notification) using **totara_notification\entity\notification_queue** entity.
6. Remove notifiable event record from the queue.

Notifications potentially can be scheduled to be sent at a specific time later, so they are not sent directly by this task. Also, some site setups might have a dedicated mail server running in their infrastructure, which will have only one scheduled task enabled which is **process_notification_queue_task**.

During the same or next cron run, scheduled task **totara_notification\task\process_notification_queue_task** will be executed as scheduled (recommended every minute). It will process records from **totara_notification\entity\notification_queue** and send the actual notifications.

During processing **process_notification_queue_task** will get all records that are scheduled to be sent and do the following:

1. Get notification record from the queue.
2. Get information about the notification: recipients, delivery outputs to be used.
3. Iterate over each recipient:
    a. Substitute all placeholders in the message subject and body
    b. Get delivery output settings from system and user preferences
    c. For each delivery output: send notification
4. Remove notification record from the queue

This will conclude all notification workflow from triggering event to sending actual notifications.

Creating new notifiable event resolvers

Notifiable events are generally attached to the Totara event subsystem, so to create a new notifiable event you need to implement an event and trigger it in the required place.

When processing event in the observer, use **\totara_notification\external_helper::create_notifiable_event_queue()** to trigger notifications:

```
<?php
class your_component_observer {
    // ...

    /**
     * Handler method for your event
     *
     * @param \your_component\event\your_event $event
     * @return bool Success status
     */
    public static function handle_event(\your_component\event\your_event $event) {
        // ...

        \totara_notification\external_helper::create_notifiable_event_queue(new
\your_component\totara_notification\resolver\your_resolver($event->get_data()));

// ...
}
```

When event resolver is processed by Centralised Notification core it should be extended to provide the information necessary for notifications. Do this by extending the **totara_notification\resolver\notifiable_event_resolver** class:

```
<?php

namespace your_component\totara_notification\resolver;
```

```
use lang_string;
use totara_core\extended_context;
use totara_notification\resolver\notifiable_event_resolver;
use totara_notification\placeholder\placeholder_option;


class your_resolver extends notifiable_event_resolver {
    /**
     * Returns the title for this notifiable event, which should be used
     * within the tree table of available notifiable events.
     *
     * @return string
     */
    public static function get_notification_title(): string {
        return get_string('notification_your_resolver_title', 'your_component');
    }

    /**
     * Returns an array of available recipients (metadata) for this event (concrete class).
     *
     * @return array
     */
    public static function get_notification_available_recipients(): array {
        return [
             \your_component\totara_notification\recipient\your_recipient:class,
        ];
    }

    /**
     * Returns the default delivery channels that defined for the event by developers.
     * However, note that admin can override this default delivery channels.
     *
     * If nothing/a specific channel is not listed here, it will fallback to the built in default.
     * To disable it, specify the actual default here.
     *
     * @return array
     */
    public static function get_notification_default_delivery_channels(): array {
        return ['email', 'popup'];
    }

    /**
     * Returns the list of available placeholder options.
     *
     * @return placeholder_option[]
     */
    public static function get_notification_available_placeholder_options(): array {
        return [
            placeholder_option::create(
                'your_placeholder',
                \your_component\totara_notification\placeholder\your_placeholder::class,
                new lang_string('notification_your_placeholder_group', 'your_component'),
                function (array $event_data): \your_component\totara_notification\placeholder\your_placeholder {
                    return \your_component\totara_notification\placeholder\your_placeholder::from_id($event_data
['item_id']);
                }
            ),
        ];
    }

    /**
     * Returns the extended context of where this event occurred. Note that this should almost certainly be
     * either the same as the natural context (but wrapped in the extended context container class) or an
     * extended context where the natural context is the immediate parent.
     *
     * @return extended_context
     */
    public function get_extended_context(): extended_context {
        return extended_context::make_with_context(
            context_program::instance($this->event_data['item_id']),
            'your_component',
```

```
            'area',
            $this->event_data['item_id']
        );
    }

    /**
     * This is to check whether the resolver is processed through event queue or not and also it could be
override if
     * dev want to skip queueing up.
     *
     * @return bool
     */
    public static function uses_on_event_queue(): bool {
        return true;
    }

    /**
     * Indicates whether the resolver supports the given context.
     * By default, resolvers support the system context.
     * Override this function to support other contexts.
     *
     * @param extended_context $extend_context
     * @return bool
     */
    public static function supports_context(extended_context $extended_context): bool {
        $context = $extended_context->get_context();

        if ($extended_context->is_natural_context()) {
            return in_array($context->contextlevel, [CONTEXT_SYSTEM, CONTEXT_COURSECAT, CONTEXT_COURSE]);
        }

        return $context->contextlevel === CONTEXT_COURSE && $extended_context->get_component() ===
'your_component';
    }
}
```

This will be enough to add a notifiable event into user preferences and administration settings for creating notifications for this event.

Creating new default notifications

A notifiable event on its own does not produce notifications. An administrator can create a notification based on the notifiable event, through the interface. However, often notifiable events have some default notifications attached to them.

Default notifications for specific events can be implemented in any component, there is no strict coupling between component and notification, so you can implement additional default notifications for core notifiable events as part of your custom component.

To implement new default notification you need to place it within **your_component\totara_notification\notification** namespace and it must extend **totara_notification\notification\built_in_notification** class.

Implement all required abstract methods and the notification will be picked up by auto-discovery.

```php
<?php

namespace your_component\totara_notification\notification;

use lang_string;
use totara_notification\notification\built_in_notification;
use totara_notification\schedule\schedule_on_event;

final class your_notification extends built_in_notification {
    /**
     * Returning the event resolver class name which this notification is belonging to.
     * It is a one-to-many relationship, meaning that one event can produce multiple
     * notifications (like the children of this one).
     *
     * @return string
     */
    public static function get_resolver_class_name(): string {
        return your_component\totara_notification\resolver\your_resolver::class;
    }

    /**
     * Returning the notification's title.
     * Note this does not use any lang_string because we don't need to do sort
     * of placeholders for the title of the built in notification.
     *
     * Please do not use placeholders with title. It has to be a static data, and must
     * come from the language pack.
     *
     * @return string
     */
    public static function get_title(): string {
        return get_string('notification_assigned_for_managers_title', 'your_component');
    }

    /**
     * Return the recipient class name.
     *
     * @return string
     */
    public static function get_recipient_class_name(): string {
        return your_component\totara_notification\recipient\manager::class;
    }

    /**
     * @return lang_string
     */
    public static function get_default_body(): lang_string {
        return new lang_string('your_notification_body', 'your_component');
    }

    /**
     * @return lang_string
     */
    public static function get_default_subject(): lang_string {
        return new lang_string('your_notification_subject', 'your_component');
    }

    /**
     * Returns the schedule offset value, translated for storage.
     * Note: it must be in seconds unit.
     *
     * @return int
     */
    public static function get_default_schedule_offset(): int {
        return schedule_on_event::default_value();
    }
}
```

After implementation, this notification will appear in the notifications list of the relevant notifiable event for administrators. Properties of the built-in notification can then be overridden and changed through the interface, the same way that custom notifications can be overridden and changed in context below where they were created.

Notification preference

The table description.

Column Name	Type	Nullable	Description
id	INT	NOT NULL	
ancestor_id	INT	NULL	This column is to keep track of the original notification (either built-in or custom) which is being overridden. If the value of this column is null, then it is saying that the notification preference record is a custom one at context/identifier.
resolver_class_name	VARCHAR	NOT NULL	The notifiable event resolver class name, which this notification_preference had been created for.
notification_class_name	VARCHAR	NULL	The built-in notification class name. The value of this column can help the notification_preference fallback to whatever the value for body, subject, body_format and so on had been defined by developer in code. When the record is a custom notification preference, then this field MUST be NULL.
context_id	INT	NOT NULL	The context's id which the notification preference is for.
component	VARCHAR	NULL	Part of extended context.
area	VARCHAR	NULL	Part of extended context.
item_id	INT	NOT NULL	Part of extended context.
title	VARCHAR	NULL	The notification preference's title. This field only has to be filled when the user first create a new custom notification. Otherwise it MUST always be null and MUST NOT be updated for overridden record.
recipient	VARCHAR	NULL	The recipient's name of notification preference record.
subject	VARCHAR	NULL	The subject of notification. If this is a custom notification then it is required, otherwise null to fallback to the default defined in ancestor notification.
subject_format	INT	NULL	The text format for subject of notification. If this is a custom notification then it is required, otherwise null to fallback to the default defined in ancestor notification.
body	VARCHAR	NULL	The body of notification. If this is a custom notification then it is required, otherwise null to fallback to the default defined in ancestor notification.
body_format	INT	NULL	The text format for body of notification. If this is a custom notification then it is required, otherwise null to fallback to the default defined in ancestor notification.
time_created	INT	NOT NULL	Timestamp of when record was created.
schedule_offset	INT	NULL	When notification should be sent in relation to event time in seconds.
enabled	INT	NULL	Enable/Disable sending notification.
forced_delivery_channels	VARCHAR	NULL	what channels should be sent regardless of user preferences.

When a system first installs it, the notification plugin will fetch all the built-in notifications from the codebase and insert the records to the table **notification_preference** above. However, all the non-required or overridden-able fields will not be populated and will be left as null as they will fallback to the built-in notification class when requested. System administrators are able to update these fields and they will no longer fall-back to the built-in notification, but instead use the updated values.

> **No overridden at system context**
>
> You MUST not create an overridden record at the system context. This means that when ancestor's ID is set then the context's id must not be a system context's id.
>
> Any overridden record at the system context will be created via **upgrade/system** and they are only overriding the built-in notification.

The overridden record must only be created at a different context with ancestor's id. This means that there must not be two records that had the same ancestor's ID and same context's ID. Exception will be thrown if you are trying to duplicate the records that have the same context's ID and ancestor's ID.