Action Scheduler

Action Scheduler □□□□□□□□□□□□□□□□□□□

- Scheduled Actions Administration Screen
- API Reference
- FAQ
- Background Processing at Scale
- Usage
- Version 3.0 FAQ
- WP CLI

Scheduled Actions Administration Screen

Action Scheduler has a built in administration screen for monitoring, debugging and manually triggering scheduled actions.

The administration interface is accessible through both:

1. **Tools > Scheduled Actions**
2. **WooCommerce > Status > Scheduled Actions**, when WooCommerce is installed.

Among other tasks, from the admin screen you can:

- Run a pending action.
- View the scheduled actions with a specific status, like all actions which have failed or are in-progress (screenshot).
- View the log entries for a specific action to find out why it failed.
- Sort scheduled actions by hook name, scheduled date, group, claim ID or status.
- Search for actions by hook, args, action ID or claim ID.

Still have questions? Check out the FAQ.



API Reference

Action Scheduler provides a range of functions for scheduling hooks to run at some time in the future on one or more occasions.

To understand the scheduling functions, it can help to think of them as extensions to WordPress' `do_action()` function that add the ability to delay and repeat when the hook will be triggered.

WP-Cron APIs vs. Action Scheduler APIs

The Action Scheduler API functions are designed to mirror the WordPress WP-Cron API functions.

Functions return similar values and accept similar arguments to their WP-Cron counterparts. The notable differences are:

- `as_schedule_single_action()` & `as_schedule_recurring_action()` will return the ID of the scheduled action rather than boolean indicating whether the event was scheduled
- `as_schedule_recurring_action()` takes an interval in seconds as the recurring interval rather than an arbitrary string
- `as_schedule_single_action()` & `as_schedule_recurring_action()` can accept a `$group` parameter to group different actions for the one plugin together.
- the `wp_` prefix is substituted with `as_` and the term `event` is replaced with `action`

API Function Availability

As mentioned in the Usage - Load Order section, Action Scheduler will initialize itself on the `'init'` hook with priority `1`. While API functions are loaded prior to this and can be called, they should not be called until after `'init'` with priority `1`, because each component, like the data store, has not yet been initialized.

Do not use Action Scheduler API functions prior to `'init'` hook with priority `1`. Doing so could lead to unexpected results, like data being stored in the incorrect location. To make this easier:

- Action Scheduler provides `Action_Scheduler::is_initialized()` for use in hooks to confirm that the data stores have been initialized.
- It also provides the `'action_scheduler_init'` action hook. It is safe to call API functions during or after this event has fired (tip: you can take advantage of WordPress's did_action() function to check this).

Function Reference / `as_enqueue_async_action()`

Description

Enqueue an action to run one time, as soon as possible.

Usage

```
as_enqueue_async_action( $hook, $args, $group, $unique, $priority );
```

Parameters

- **$hook** (string)(required) Name of the action hook.
- **$args** (array) Arguments to pass to callbacks when the hook triggers. Default: `array()`.
- **$group** (string) The group to assign this job to. Default: ''.
- **$unique** (boolean) Whether the action should be unique. Default: `false`.
- **$priority** (integer) Lower values take precedence over higher values. Defaults to 10, with acceptable values falling in the range 0-255.

Return value

`(integer)` the action's ID. Zero if there was an error scheduling the action. The error will be sent to `error_log`.

Function Reference / `as_schedule_single_action()`

Description

Schedule an action to run one time at some defined point in the future.

Usage

```
as_schedule_single_action( $timestamp, $hook, $args, $group, $unique, $priority );
```

Parameters

- **$timestamp** (integer)(required) The Unix timestamp representing the date you want the action to run.
- **$hook** (string)(required) Name of the action hook.
- **$args** (array) Arguments to pass to callbacks when the hook triggers. Default: `array()`.
- **$group** (string) The group to assign this job to. Default: ''.
- **$unique** (boolean) Whether the action should be unique. Default: `false`.
- **$priority** (integer) Lower values take precedence over higher values. Defaults to 10, with acceptable values falling in the range 0-255.)

Return value

`(integer)` the action's ID. Zero if there was an error scheduling the action. The error will be sent to `error_log`.

Function Reference / `as_schedule_recurring_action()`

Description

Schedule an action to run repeatedly with a specified interval in seconds.

Usage

```
as_schedule_recurring_action( $timestamp, $interval_in_seconds, $hook, $args, $group, $unique, $priority );
```

Parameters

- **$timestamp** (integer)(required) The Unix timestamp representing the date you want the action to run.
- **$interval_in_seconds** (integer)(required) How long to wait between runs.
- **$hook** (string)(required) Name of the action hook.
- **$args** (array) Arguments to pass to callbacks when the hook triggers. Default: `array()`.
- **$group** (string) The group to assign this job to. Default: ''.
- **$unique** (boolean) Whether the action should be unique. Default: `false`.
- **$priority** (integer) Lower values take precedence over higher values. Defaults to 10, with acceptable values falling in the range 0-255.

Return value

`(integer)` the action's ID. Zero if there was an error scheduling the action. The error will be sent to `error_log`.

Function Reference / `as_schedule_cron_action()`

Description

Schedule an action that recurs on a cron-like schedule.

If execution of a cron-like action is delayed, the next attempt will still be scheduled according to the provided cron expression.

Usage

```
as_schedule_cron_action( $timestamp, $schedule, $hook, $args, $group, $unique, $priority );
```

Parameters

- **$timestamp** (integer)(required) The Unix timestamp representing the date you want the action to run.
- **$schedule** (string)(required) $schedule A cron-like schedule string, see http://en.wikipedia.org/wiki/Cron.
- **$hook** (string)(required) Name of the action hook.
- **$args** (array) Arguments to pass to callbacks when the hook triggers. Default: `array()`.
- **$group** (string) The group to assign this job to. Default: ''.
- **$unique** (boolean) Whether the action should be unique. Default: `false`.
- **$priority** (integer) Lower values take precedence over higher values. Defaults to 10, with acceptable values falling in the range 0-255.

Return value

`(integer)` the action's ID. Zero if there was an error scheduling the action. The error will be sent to `error_log`.

Function Reference / `as_unschedule_action()`

Description

Cancel the next occurrence of a scheduled action.

Usage

```
as_unschedule_action( $hook, $args, $group );
```

Parameters

- **$hook** (string)(required) Name of the action hook.

- **$args** (array) Arguments passed to callbacks when the hook triggers. Default: `array()`.
- **$group** (string) The group the job is assigned to. Default: ''.

Return value

`(null)`

Function Reference / `as_unschedule_all_actions()`

Description

Cancel all occurrences of a scheduled action.

Usage

```
as_unschedule_all_actions( $hook, $args, $group )
```

Parameters

- **$hook** (string)(required) Name of the action hook.
- **$args** (array) Arguments passed to callbacks when the hook triggers. Default: `array()`.
- **$group** (string) The group the job is assigned to. Default: ''.

Return value

`(string|null)` The scheduled action ID if a scheduled action was found, or null if no matching action found.

Function Reference / `as_next_scheduled_action()`

Description

Returns the next timestamp for a scheduled action.

Usage

```
as_next_scheduled_action( $hook, $args, $group );
```

Parameters

- **$hook** (string)(required) Name of the action hook. Default: *none*.
- **$args** (array) Arguments passed to callbacks when the hook triggers. Default: `array()`.
- **$group** (string) The group the job is assigned to. Default: ''.

Return value

`(integer|boolean)` The timestamp for the next occurrence of a pending scheduled action, true for an async or in-progress action or false if there is no matching action.

Function Reference / `as_has_scheduled_action()`

Description

Check if there is a scheduled action in the queue, but more efficiently than as_next_scheduled_action(). It's recommended to use this function when you need to know whether a specific action is currently scheduled. *Available since 3.3.0.*

Usage

```
as_has_scheduled_action( $hook, $args, $group );
```

Parameters

- **$hook** (string)(required) Name of the action hook. Default: *none*.
- **$args** (array) Arguments passed to callbacks when the hook triggers. Default: `array()`.
- **$group** (string) The group the job is assigned to. Default: ''.

Return value

`(boolean)` True if a matching action is pending or in-progress, false otherwise.

Function Reference / `as_get_scheduled_actions()`

Description

Find scheduled actions.

Usage

```
as_get_scheduled_actions( $args, $return_format );
```

Parameters

- **$args** (array) Arguments to search and filter results by. Possible arguments, with their default values:
  - `'hook' => ''` - the name of the action that will be triggered
  - `'args' => NULL` - the args array that will be passed with the action
  - `'date' => NULL` - the scheduled date of the action. Expects a DateTime object, a unix timestamp, or a string that can parsed with strtotime().
  - `'date_compare' => '<=` ' - operator for testing "date". accepted values are '!=', '>', '>=', '<', '<=', '='
  - `'modified' => NULL` - the date the action was last updated. Expects a DateTime object, a unix timestamp, or a string that can parsed with strtotime().
  - `'modified_compare' => '<='` - operator for testing "modified". accepted values are '!=', '>', '>=', '<', '<=', '='

    - `'group' => ''` - the group the action belongs to
    - `'status' => ''` - ActionScheduler_Store::STATUS_COMPLETE or ActionScheduler_Store::STATUS_PENDING
    - `'claimed' => NULL` - TRUE to find claimed actions, FALSE to find unclaimed actions, a string to find a specific claim ID
    - `'per_page' => 5` - Number of results to return
    - `'offset' => 0`
    - `'orderby' => 'date'` - accepted values are 'hook', 'group', 'modified', or 'date'
    - `'order' => 'ASC'`
- **$return_format** (string) The format in which to return the scheduled actions: 'OBJECT', 'ARRAY_A', or 'ids'. Default: '*OBJECT*'.

Return value

`(array)` Array of action rows matching the criteria specified with `$args`.

Function Reference / `as_supports()`

Description

Check if a specific feature is supported by the current version of Action Scheduler.

Available since 3.9.3.

Usage

```
as_supports( $feature );
```

Parameters

$feature (string) (required)
The feature to check support for.
Currently supported:

- `'ensure_recurring_actions_hook'` — Indicates support for the `action_scheduler_ensure_recurring_actions` hook.

Return value

`(boolean)`
True if the feature is supported, false otherwise.

Example

```
if ( as_supports( 'ensure_recurring_actions_hook' ) ) {
    // Safe to depend on the 'action_scheduler_ensure_recurring_actions' hook.
    add_action( 'action_scheduler_ensure_recurring_actions',
'my_plugin_schedule_my_recurring_action' );
}
```

FAQ

FAQ

Is it safe to release Action Scheduler in my plugin? Won't its functions conflict with another copy of the library?

Action Scheduler is designed to be used and released in plugins. It avoids redeclaring public API functions when more than one copy of the library is being loaded by different plugins. It will also load only the most recent version of itself (by checking registered versions after all plugins are loaded on the `'plugins_loaded'` hook).

To use it in your plugin (or theme), simply require the `action-scheduler/action-scheduler.php` file. Action Scheduler will take care of the rest. **Note:** Action Scheduler is only loaded from a theme if it is not included in any active plugins.

I don't want to use WP-Cron. Does Action Scheduler depend on WP-Cron?

By default, Action Scheduler is initiated by WP-Cron (and the `'shutdown'` hook on admin requests). However, it has no dependency on the WP-Cron system. You can initiate the Action Scheduler queue in other ways with just one or two lines of code.

For example, you can start a queue directly by calling:

```
ActionScheduler::runner()->run();
```

Or trigger the `'action_scheduler_run_queue'` hook and let Action Scheduler do it for you:

```
do_action( 'action_scheduler_run_queue', $context_identifier );
```

Further customization can be done by extending the `ActionScheduler_Abstract_QueueRunner` class to create a custom Queue Runner. For an example of a customized queue runner, see the `ActionScheduler_WPCLI_QueueRunner`, which is used when running WP CLI.

Want to create some other method for initiating Action Scheduler? Open a new issue, we'd love to help you with it.

I don't want to use WP-Cron, ever. Does Action Scheduler replace WP-Cron?

By default, Action Scheduler is designed to work alongside WP-Cron and not change any of its behaviour. This helps avoid unexpectedly overriding WP-Cron on sites installing your plugin, which may have nothing to do with WP-Cron.

However, we can understand why you might want to replace WP-Cron completely in environments within your control, especially as it gets you the advantages of Action Scheduler. This should be possible without too much code.

You could use the `'schedule_event'` hook in WordPress to use Action Scheduler for only newly scheduled WP-Cron jobs and map the `$event` param to Action Scheduler API functions.

Alternatively, you can use a combination of the `'pre_update_option_cron'` and `'pre_option_cron'` hooks to override all new and previously scheduled WP-Cron jobs (similar to the way Cavalcade does it).

If you'd like to create a plugin to do this automatically and want to share your work with others, open a new issue to let us know, we'd love to help you with it.

How does Action Scheduler store its data?

Action Scheduler 3.0 and newer stores data in custom tables prefixed with `actionscheduler_`. For the list of all tables and their schemas, refer to the `ActionScheduler_StoreSchema` class.

Prior to Action 3.0, actions were a custom post type, and data was stored in `wp_posts`, `wp_postmeta` and related tables.

Action Scheduler 3+ migrates data from the custom post type to custom tables.

Can I use a different storage scheme?

Of course! Action Scheduler data storage is completely swappable, and always has been.

If you choose to, you can actually store them anywhere, like in a remote storage service from Amazon Web Services.

To implement a custom store:

1. extend the abstract `ActionScheduler_Store` class, being careful to implement each of its methods
2. attach a callback to `'action_scheduler_store_class'` to tell Action Scheduler your class is the one which should be used to manage storage, e.g.

```
function eg_define_custom_store( $existing_storage_class ) {
	return 'My_Radical_Action_Scheduler_Store';
}
add_filter( 'action_scheduler_store_class', 'eg_define_custom_store', 10, 1 );
```

Take a look at the `classes/data-stores/ActionScheduler_DBStore.php` class for an example implementation of `ActionScheduler_Store`.

If you'd like to create a plugin to do this automatically and release it publicly to help others, open a new issue to let us know, we'd love to help you with it.

Can I use a different storage scheme just for logging?

Of course! Action Scheduler's logger is completely swappable, and always has been. You can also customise where logs are stored, and the storage mechanism.

To implement a custom logger:

1. extend the abstract `ActionScheduler_Logger` class, being careful to implement each of its methods
2. attach a callback to `'action_scheduler_logger_class'` to tell Action Scheduler your class is the one which should be used to manage logging, e.g.

```
function eg_define_custom_logger( $existing_storage_class ) {
	return 'My_Radical_Action_Scheduler_Logger';
}
add_filter( 'action_scheduler_logger_class', 'eg_define_custom_logger', 10, 1 );
```

Take a look at the `classes/data-stores/ActionScheduler_DBLogger.php` class for an example implementation of `ActionScheduler_Logger`.

I want to run Action Scheduler only on a dedicated application server in my cluster. Can I do that?

Wow, now you're really asking the tough questions. In theory, yes, this is possible. The `ActionScheduler_QueueRunner` class, which is responsible for running queues, is swappable via the `'action_scheduler_queue_runner_class'` filter.

Because of this, you can effectively customise queue running however you need. Whether that means tweaking minor things, like not using WP-Cron at all to initiate queues by overriding `ActionScheduler_QueueRunner::init()`, or completely changing how and where queues are run, by overriding `ActionScheduler_QueueRunner::run()`.

Is Action Scheduler safe to use on my production site?

Yes, absolutely! Action Scheduler is actively used on tens of thousands of production sites already. Right now it's responsible for scheduling everything from emails to payments.

In fact, every month, Action Scheduler processes millions of payments as part of the WooCommerce Subscriptions extension.

It requires no setup, and won't override any WordPress APIs (unless you want it to).

How does Action Scheduler work on WordPress Multisite?

Action Scheduler is designed to manage the scheduled actions on a single site. It has no special handling for running queues across multiple sites in a multisite network. That said, because its storage and Queue Runner are completely swappable, it would be possible to write multisite handling classes to use with it.

If you'd like to create a multisite plugin to do this and release it publicly to help others, open a new issue to let us know, we'd love to help you with it.

How can I change the Action Scheduler User-Agent to better identify its requests?

Action Scheduler has a filter available named `as_async_request_queue_runner_post_args` which can be used to filter the arguments that are being sent to the `wp_remote_post` call.

The User-Agent parameter is just one of them and can be adjusted as follows:

```
function eg_define_custom_user_agent( $args ) {
	$versions           = ActionScheduler_Versions::instance();
	$args['user-agent'] = 'Action Scheduler/' . $versions->latest_version();

	return $args;
}
add_filter( 'as_async_request_queue_runner_post_args', 'eg_define_custom_user_agent', 10, 1 );
```

My site has past-due actions; what can I do?

Actions that are past-due have missed their scheduled date; because of how WP Cron works, it is normal to have some past-due actions.

If there are several past-due actions more than one day old, there may be something wrong with your site. If you are using WooCommerce and need help determining the issue, please contact us on our forum page.

Background Processing at Scale

Action Scheduler's default processing is designed to work reliably across all different hosting environments. To achieve that, the default processing thresholds are more conservative than many servers could support.

Specifically, Action Scheduler will only process actions in a request until:

- 90% of available memory is used
- processing another 3 actions would exceed 30 seconds of total request time, based on the average processing time for the current batch
- in a single concurrent queue

While Action Scheduler will initiate an asynchronous loopback request to process additional actions, on sites with very large queues, this can result in slow processing times.

While using WP CLI to process queues is the best approach to increasing processing speed, on occasion, that is not a viable option. In these cases, it's also possible to increase the processing thresholds in Action Scheduler to increase the rate at which actions are processed by the default queue runner.

Increasing Time Limit

By default, Action Scheduler will only process actions for a maximum of 30 seconds in each request. This time limit minimizes the risk of a script timeout on unknown hosting environments, some of which enforce 30-second timeouts.

If you know your host supports longer than this time limit for web requests, you can increase this time limit. This allows more actions to be processed in each request and reduces the lag between processing each queue, greatly speeding up the processing rate of scheduled actions.

For example, the following snippet will increase the time limit to 2 minutes (120 seconds):

```
function eg_increase_time_limit( $time_limit ) {
	return 120;
}
```

```
add_filter( 'action_scheduler_queue_runner_time_limit', 'eg_increase_time_limit' );
```

Some of the known host time limits are:

- 60 seconds on WP Engine
- 120 seconds on Pantheon
- 120 seconds on SiteGround

Increasing Batch Size

By default, Action Scheduler will claim a batch of 25 actions. This small batch size is because the default time limit is only 30 seconds; however, if you know your actions are processing very quickly, e.g. taking milliseconds not seconds, or that you have more than 30 seconds available to process each batch, increasing the batch size can slightly improve performance.

This is because claiming a batch has some overhead, so the less often a batch needs to be claimed, the faster actions can be processed.

For example, to increase the batch size to 100, we can use the following function:

```
function eg_increase_action_scheduler_batch_size( $batch_size ) {
return 100;
}
add_filter( 'action_scheduler_queue_runner_batch_size',
'eg_increase_action_scheduler_batch_size' );
```

Increasing Concurrent Batches

By default, Action Scheduler will run only one concurrent batch of actions. This is to prevent consuming a lot of available connections or processes on your web server.

However, your server may allow a large number of connections, for example, because it has a high value for Apache's `MaxClients` setting or PHP-FPM's `pm.max_children` setting.

If this is the case, you can use the `'action_scheduler_queue_runner_concurrent_batches'` filter to increase the number of concurrent batches allowed, and therefore speed up large numbers of actions scheduled to be processed simultaneously.

For example, to increase the allowed number of concurrent queues to 10, we can use the following code:

```
function eg_increase_action_scheduler_concurrent_batches( $concurrent_batches ) {
	return 10;
}
add_filter( 'action_scheduler_queue_runner_concurrent_batches',
'eg_increase_action_scheduler_concurrent_batches' );
```

> WARNING: because the async queue runner introduced in Action Scheduler 3.0 will continue asynchronous loopback requests to process actions, increasing the number of concurrent batches can substantially increase server load and take down a site. WP CLI is a better method to achieve higher throughput.

Increasing Initialisation Rate of Runners

By default, Action Scheduler initiates at most one queue runner every time the `'action_scheduler_run_queue'` action is triggered by WP Cron.

Because this action is only triggered at most once every minute, then there will rarely be more than one queue processing actions even if the concurrent runners are increased.

To handle larger queues on more powerful servers, it's possible to initiate additional queue runners whenever the `'action_scheduler_run_queue'` action is run.

That can be done by initiating additional secure requests to our server via loopback requests.

The code below demonstrates how to create 5 loopback requests each time a queue begins:

```
/**
 * Trigger 5 additional loopback requests with unique URL params.
 */
function eg_request_additional_runners() {

	// allow self-signed SSL certificates
	add_filter( 'https_local_ssl_verify', '__return_false', 100 );

	for ( $i = 0; $i < 5; $i++ ) {
		$response = wp_remote_post( admin_url( 'admin-ajax.php' ), array(
```

```
□□□'method'      => 'POST',
□□□'timeout'     => 45,
□□□'redirection' => 5,
□□□'httpversion' => '1.0',
□□□'blocking'    => false,
□□□'headers'     => array(),
□□□'body'        => array(
□□□□'action'     => 'eg_create_additional_runners',
□□□□'instance'   => $i,
□□□□'eg_nonce' => wp_create_nonce( 'eg_additional_runner_' . $i ),
□□□),
□□□'cookies'     => array(),
□□) );
□}
}
add_action( 'action_scheduler_run_queue', 'eg_request_additional_runners', 0 );

/**
 * Handle requests initiated by eg_request_additional_runners() and start a queue runner if
the request is valid.
 */
function eg_create_additional_runners() {

□if ( isset( $_POST['eg_nonce'] ) && isset( $_POST['instance'] ) && wp_verify_nonce(
$_POST['eg_nonce'], 'eg_additional_runner_' . $_POST['instance'] ) ) {
□□ActionScheduler_QueueRunner::instance()->run();
□}

□wp_die();
}
add_action( 'wp_ajax_nopriv_eg_create_additional_runners', 'eg_create_additional_runners', 0
);
```

> WARNING: because of the processing rate of scheduled actions, this kind of increase can very easily take down a site. Use only on high-powered servers and be sure to test before attempting to use it in production.

Cleaning Failed Actions

As of version 4.0.0, Action Scheduler retains failed actions for three months by default. This retention period is independent of `action_scheduler_retention_period`. If 'failed' is included in the purge statuses, it is cleaned alongside the other statuses using their retention period instead.

The retention period for failed actions can be customized. Zero purges them on the next cleanup pass; negative values are treated as zero:

```
add_filter( 'action_scheduler_retention_period_for_failed', function( $retention_period ) {
    return 12 * MONTH_IN_SECONDS;
} );
```

To retain failed actions indefinitely (the pre-4.0.0 behavior), disable failed cleanup entirely:

```
add_filter( 'action_scheduler_enable_failed_action_cleanup', '__return_false' );
```

Prior to version 4.0.0, by default Action Scheduler did not automatically delete old failed actions. There are two optional methods of removing these actions:

- Include the failed status in the list of statuses to purge:

```
add_filter( 'action_scheduler_default_cleaner_statuses', function( $statuses ) {
    $statuses[] = ActionScheduler_Store::STATUS_FAILED;
    return $statuses;
} );
```

- Use WP CLI:

```
// Example
wp action-scheduler clean --status=failed --batch-size=50 --before='90 days ago' --pause=2
```

High Volume Plugin

It's not necessary to add all of this code yourself, there is a handy plugin to get access to each of these increases - the Action Scheduler - High Volume plugin.

Usage

Using Action Scheduler requires:

1. installing the library
2. scheduling an action

Installation

There are two ways to install Action Scheduler:

1. regular WordPress plugin; or
2. a library within your plugin's codebase.

Note that Action Scheduler follows an L-2 dependency version policy. That is, the library requires at least the "latest minus two" version of WordPress and the PHP minimum version requirement of that WordPress version.

Usage as a Plugin

Action Scheduler includes the necessary file headers to be used as a standard WordPress plugin.

To install it as a plugin:

1. Download the .zip archive of the latest stable release
2. Go to the **Plugins > Add New > Upload** administration screen on your WordPress site
3. Select the archive file you just downloaded
4. Click **Install Now**
5. Click **Activate**

Or clone the Git repository into your site's `wp-content/plugins` folder.

Using Action Scheduler as a plugin can be handy for developing against newer versions, rather than having to update the subtree in your codebase. **When installed as a plugin, Action Scheduler does not provide any user interfaces for scheduling actions**. The only way to interact with Action Scheduler is via code.

Usage as a Library

To use Action Scheduler as a library:

1. include the Action Scheduler codebase
2. load the library by including the `action-scheduler.php` file

Using a subtree in your plugin, theme or site's Git repository to include Action Scheduler is the recommended method. Composer can also be used.

To include Action Scheduler as a git subtree:

Step 1. Add the Repository as a Remote

```
git remote add -f subtree-action-scheduler https://github.com/woocommerce/action-scheduler.git
```

Adding the subtree as a remote allows us to refer to it in short from via the name `subtree-action-scheduler`, instead of the full GitHub URL.

Step 2. Add the Repo as a Subtree

```
git subtree add --prefix libraries/action-scheduler subtree-action-scheduler trunk --squash
```

This will add the `trunk` branch of Action Scheduler to your repository in the folder `libraries/action-scheduler`.

You can change the `--prefix` to change where the code is included. Or change the `trunk` branch to a tag, like `2.1.0` to include only a stable version.

Step 3. Update the Subtree

To update Action Scheduler to a new version, use the commands:

```
git fetch subtree-action-scheduler trunk
git subtree pull --prefix libraries/action-scheduler subtree-action-scheduler trunk --squash
```

Loading Action Scheduler

Regardless of how it is installed, to load Action Scheduler, you only need to include the `action-scheduler.php` file, e.g.

```
<?php
require_once( plugin_dir_path( __FILE__ ) . '/libraries/action-scheduler/action-scheduler.php'
);
```

There is no need to call any functions or do else to initialize Action Scheduler.

When the `action-scheduler.php` file is included, Action Scheduler will register the version in that file and then load the most recent version of itself on the site. It will also load the most recent version of all API functions.

Load Order

Action Scheduler will register its version on `'plugins_loaded'` with priority `0` - after all other plugin codebases has been loaded. Therefore **the `action-scheduler.php` file must be included before `'plugins_loaded'` priority `0`**.

It is recommended to load it *when the file including it is included*. However, if you need to load it on a hook, then the hook must occur before `'plugins_loaded'`, or you can use `'plugins_loaded'` with negative priority, like `-10`.

Action Scheduler will later initialize itself on `'init'` with priority `1`. Action Scheduler APIs should not be used until after `'init'` with priority `1`. As described in API Function Availability, you can also use the `'action_scheduler_init'` action hook for this purpose.

Usage in Themes

When using Action Scheduler in themes, it's important to note that if Action Scheduler has been registered by a plugin, then the latest version registered by a plugin will be used, rather than the version included in the theme. This is because of the version dependency handling code using `'plugins_loaded'` since version 1.0.

Scheduling Recurring Actions on Activation and Ensuring They Remain Scheduled

When developing a plugin or extension that relies on recurring actions, it is essential to schedule those actions when the plugin is first activated or updated.

Best Practice: Schedule on Activation

Plugins should always schedule any important recurring actions during activation or upgrade routines. This ensures the action is registered as soon as the plugin is enabled.

Example: Scheduling on Activation

```
/**
 * Schedule recurring action on plugin activation.
 */
function my_plugin_activate() {
    if ( ! as_has_scheduled_action( 'my_plugin_recurring_action' ) ) {
        as_schedule_recurring_action(
            time(),
            HOUR_IN_SECONDS,
            'my_plugin_recurring_action'
        );
    }
}
register_activation_hook( __FILE__, 'my_plugin_activate' );
```

Ensuring Recurring Actions Remain Scheduled

After initial scheduling, there may be rare cases where a recurring action is lost (e.g., due to a database failure or repeated action failures). The `action_scheduler_ensure_recurring_actions` hook provides a reliable opportunity to check and reschedule the action if necessary.

Example: Using the Ensure Hook

```
/**
 * Ensure recurring action remains scheduled.
 */
add_action( 'action_scheduler_ensure_recurring_actions', function() {
    if ( ! as_has_scheduled_action( 'my_plugin_recurring_action' ) ) {
```

```
        as_schedule_recurring_action(
            time(),
            HOUR_IN_SECONDS,
            'my_plugin_recurring_action'
        );
    }
});
```

Compatibility Notes:

When using Action Scheduler, be aware of version-specific features and functions. If your plugin may run with older versions of Action Scheduler because it is relying on ActionScheduler to be included separately from your plugin where you cannot be sure of the version being loaded, use the following guidance to ensure compatibility.

New in 3.9.3: `action_scheduler_ensure_recurring_actions` Hook and `as_supports()`

- The `action_scheduler_ensure_recurring_actions` hook and the `as_supports()` API method were introduced in Action Scheduler 3.9.3.
- To use this hook, first check that the current Action Scheduler version supports it.

Example: Ensuring a Recurring Action Remains Scheduled in a Backward Compatible Way

```
/**
 * Schedules the plugin's recurring action if it is not already scheduled.
 */
function my_plugin_schedule_my_recurring_action() {
	if ( ! as_has_scheduled_action( 'my_plugin_recurring_action' ) ) {
		as_schedule_recurring_action(
			time(),
			HOUR_IN_SECONDS,
			'my_plugin_recurring_action'
		);
	}
}

/**
 * Add the hooks needed for my_plugin_schedule_my_recurring_action() so it can make sure our hook stays scheduled.
 */
```

```
add_action( 'init', function () {
	if ( function_exists( 'as_supports' ) && as_supports( 'ensure_recurring_actions_hook' ) ) {
		// Preferred: runs periodically in the background.
		add_action( 'action_scheduler_ensure_recurring_actions',
'my_plugin_schedule_my_recurring_action' );
	} elseif ( is_admin() ) {
		// Fallback: runs on every admin request.
		my_plugin_schedule_my_recurring_action();
	}
} );
```

Note on `as_has_scheduled_action()` (Added in 3.3.0)

- The `as_has_scheduled_action()` function was introduced in Action Scheduler 3.3.0.
- If you need to support even older versions, check if the `as_has_scheduled_action` function exists and fallback to `as_next_scheduled_action()` instead.

For more details on all available API functions, their parameters, and when they were added, refer to the API Reference.

Passing arguments

It is possible to pass arguments to your callbacks. When you initially supply the arguments via a call to `as_schedule_single_action()` or one of its sister functions, they should be in an array. However, your callback function will receive each array item as an individual parameter. Here's an example:

```
// You must specify the number of arguments to be accepted (in this case, 2).
add_action( 'purchase_notification', 'send_purchase_notification', 10, 2 );

// When scheduling the action, provide the arguments as an array.
as_schedule_single_action( time(), 'purchase_notification', array(
    'bob@foo.bar',
    'Learning Action Scheduler (e-book)',
) );

// Your callback should accept the appropriate number of parameters (again, in this case, 2).
function send_purchase_notification( $customer_email, $purchased_item ) {
    wp_mail(
        $customer_email,
        'Thank you!',
```

```
        "You purchased $purchased_item successfully."
    );
}
```

The above is a pretty simple illustration, and you would of course need to make changes if you wish to do anything more complex, such as accept a variable number of arguments. However, it hopefully illustrates the basic principles involved.

Version 3.0 FAQ

Version 3.0 FAQ

Do we need to wait for this to be bundled with WooCommerce?

No. Action Scheduler can be run as a standalone plugin. Action Scheduler 3.X is also part of WooCommerce 4.0 and WooCommerce Subscriptions 3.0, so when updating WooCommerce or WooCommerce Subscriptions, Action Scheduler will also be updated.

Can we safely switch to action scheduler version 3.0 and ditch the custom tables plugin?

Yes! The Action Scheduler Custom Tables plugin code is now part of Action Scheduler itself (with a few improvements). We recommend disabling the Custom Tables plugin immediately after activating Action Scheduler 3.0, or a plugin containing Action Scheduler 3.0, like WooCommerce Subscriptions 3.0 and newer.

How do we migrate from our own custom data store?

By default, Action Scheduler will only initiate a migration from the internal `WPPostStore` data store. To enable migration from any custom datastore add the following filter `add_filter( 'action_scheduler_migrate_data_store', '__return_true' );`.

I'm currently on PHP <5.5. When I update PHP will the migration start automatically?

Yes! The migration is initiated as soon as all dependencies are met.

I would like to update a plugin for testing that includes Action Scheduler 3.0 but would like to postpone the migration until that testing is complete. Is that possible?

Yes, while we recommend migrating to custom tables as soon as possible for performance reasons, you can use `add_filter( 'action_scheduler_migration_dependencies_met', '__return_false' );` to prevent the migration from initiating.

I updated to Action Scheduler 3.0+, the migration is running, and I'm seeing a lot of `admin-ajax.php?action=as_async_request_queue` requests. Is it possible to reduce this load?

You can increase the delay (in seconds) between these requests with

```
add_filter( 'action_scheduler_async_request_sleep_seconds', function( $sleep ) {
	return 10;
} );
```

The default value for this filter is `1` in versions 3.0.X and `5` in versions 3.1.X.

Alternatively, the async queue runner can be disabled while your migration is in process with `add_filter( 'action_scheduler_allow_async_request_runner', '__return_false' );`.

Once the migration is complete, these filters should be removed.

Is it safe to use Action Scheduler function in the WordPress `shutdown` hook.

No, you should avoid calls to Action Scheduler functions in the `shutdown` hook because the `shutdown` hook is still triggered after a fatal error. If this occurs while WordPress is loading, Action Scheduler is in an unknown state.

Is there a strong likelihood of migration issues with any of the above?

There is always the possibilities of issues, but it is not a strong likelihood. We tested migrating from Action Scheduler 2.n with the custom data stores (including Action Scheduler Custom Tables plugin) active to Action Scheduler 3.0 on a number of test sites.

As with all major, and minor, upgrades, we still testing the update on a staging site before updating the live site. We also recommending taking a backup before updating the live site.

If you wish to undertake more comprehensive testing on a development or staging site before updating, or want to closely monitor the update on a live site, follow these steps:

Stage 1: Prepare your site:

1. Take a backup of your database (AS 3.0 migrates data to custom tables, this can not be undone, you'll need to restore a backup to downgrade).
2. Go to Tools > Action Scheduler
3. Take a screenshot of the action status counts at the top of the page. Example screenshot: https://cld.wthms.co/kwIqv7

Stage 2: Install & Activate Action Scheduler 3.0+ as a plugin

1. The migration will start almost immediately
2. Keep an eye on your error log
3. Report any notices, errors or other issues on GitHub

Stage 3: Verify Migration

1. Go to Tools > Action Scheduler

2. Take a screenshot of the action status counts at the top of the page.
3. Verify the counts match the status counts taken in Stage 1 (the Completed counts could be higher because actions will have been completed to run the migration).

WP CLI

Action Scheduler has custom WP CLI commands available for processing actions.

For large sites, WP CLI is a much better choice for running queues of actions than the default WP Cron runner. These are some common cases where WP CLI is a better option:

- long-running tasks - Tasks that take a significant amount of time to run
- large queues - A large number of tasks will naturally take a longer time
- other plugins with extensive WP Cron usage - WP Cron's limited resources are spread across more tasks

With a regular web request, you may have to deal with script timeouts enforced by hosts, or other restraints that make it more challenging to run Action Scheduler tasks. Utilizing WP CLI to run commands directly on the server give you more freedom. This means that you typically don't have the same constraints of a normal web request.

If you choose to utilize WP CLI exclusively, you can disable the normal WP CLI queue runner by installing the Action Scheduler - Disable Default Queue Runner plugin. Note that if you do this, you **must** run Action Scheduler via WP CLI or another method, otherwise no scheduled actions will be processed.

Commands

These are the commands available to use with Action Scheduler:

- `action-scheduler clean`
  Options:
  - `--batch-size` - This is the number of actions per status to clean in a single batch. Default is `20`.
  - `--batches` - This is the number of batches to process. Default 0 means that batches will continue to process until there are no more actions to delete.
  - `--status` - Process only actions with specific status or statuses. Default is `canceled` and `complete`. Define multiple statuses as a comma separated string (without spaces), e.g. `--status=complete,failed,canceled`
  - `--before` - Process only actions with scheduled date older than this. Defaults to 31 days. e.g `--before='7 days ago'`, `--before='02-Feb-2020 20:20:20'`
  - `--pause` - The number of seconds to pause between batches. Default no pause.
- `action-scheduler migrate`
  Note: This command is only available while the migration to custom tables is in progress.

- `action-scheduler run`
  Options:
  - `--batch-size` - This is the number of actions to run in a single batch. The default is `100`.
  - `--batches` - This is the number of batches to run. Using 0 means that batches will continue running until there are no more actions to run.
  - `--hooks` - Process only actions with specific hook or hooks, like `'woocommerce_scheduled_subscription_payment'`. By default, actions with any hook will be processed. Define multiple hooks as a comma separated string (without spaces), e.g. `--hooks=woocommerce_scheduled_subscription_trial_end,woocommerce_scheduled_subscription_payment,woocommerce_scheduled_subscription_expiration`
  - `--group` - Process only actions in a specific group, like `'woocommerce-memberships'`. By default, actions in any group (or no group) will be processed.
  - `--exclude-groups` - Ignore actions from the specified group or groups (to specify multiple groups, supply a comma-separated list of slugs). This option is ignored if `--group` is also specified.
  - `--force` - By default, Action Scheduler limits the number of concurrent batches that can be run at once to ensure the server does not get overwhelmed. Using the `--force` flag overrides this behavior to force the WP CLI queue to run.
- `action-scheduler action cancel`
- `action-scheduler action create`
- `action-scheduler action delete`
- `action-scheduler action generate`
- `action-scheduler action get`
- `action-scheduler action list`
- `action-scheduler action next`
- `action-scheduler action run`
- `action-scheduler datastore`
- `action-scheduler runner`
- `action-scheduler status`
- `action-scheduler version`

The best way to get a full list of commands and their available options is to use WP CLI itself. This can be done by running `wp action-scheduler` to list all Action Scheduler commands, or by including the `--help` flag with any of the individual commands. This will provide all relevant parameters and flags for the command.

Cautionary Note on Action Dependencies when using `--group`

or `--hooks` Options

The `--group` and `--hooks` options should be used with caution if you have an implicit dependency between scheduled actions based on their schedule.

For example, consider two scheduled actions for the same subscription:

- `scheduled_payment` scheduled for `2015-11-13 00:00:00` and
- `scheduled_expiration` scheduled for `2015-11-13 00:01:00`.

Under normal conditions, Action Scheduler will ensure the `scheduled_payment` action is run before the `scheduled_expiration` action. Because that's how they are scheduled.

However, when using the `--hooks` option, the `scheduled_payment` and `scheduled_expiration` actions will be processed in separate queues. As a result, this dependency is not guaranteed.

For example, consider a site with both:

- 100,000 `scheduled_payment` actions, scheduled for `2015-11-13 00:00:00`
- 100 `scheduled_expiration` actions, scheduled for `2015-11-13 00:01:00`

If two queue runners are running alongside each other with each runner dedicated to just one of these hooks, the queue runner handling expiration hooks will complete the processing of the expiration hooks more quickly than the queue runner handling all the payment actions.

Because of this, the `--group` and `--hooks` options should be used with caution to avoid processing actions with an implicit dependency based on their schedule in separate queues.

Improving Performance with `--group` or `--hooks`

Being able to run queues for specific hooks or groups of actions is valuable at scale. Why? Because it means you can restrict the concurrency for similar actions.

For example, let's say you have 300,000 actions queued up comprised of:

- 100,000 renewals payments
- 100,000 email notifications
- 100,000 membership status updates

Action Scheduler's default WP Cron queue runner will process them all together. e.g. when it claims a batch of actions, some may be emails, some membership updates and some renewals.

When you add concurrency to that, you can end up with issues. For example, if you have 3 queues running, they may all be attempting to process similar actions at the same time, which can lead to querying the same database tables with similar queries. Depending on the code/queries running, this can lead to database locks or other issues.

If you can batch based on each action's group, then you can improve performance by processing like actions consecutively, but still processing the full set of actions concurrently.

For example, if one queue is created to process emails, another to process membership updates, and another to process renewal payments, then the same queries won't be run at the same time, and 3 separate queues will be able to run more efficiently.

The WP CLI runner can achieve this using the `--group` option to create queue runners that focus on a specific group and, optionally, the `--exclude-groups` option to create one or more 'catch-all' queue runners that ignore those groups.