# PHP migrations

Source file: doc/eng-us/guides/migrations-php.md Last changed 2026-11-04
https://doc.exponential.earth/learninminutes/exphelp/viewer/guides/migrations-php.md

This page is the reference for writing a migration in PHP: the file, the class, the methods the runner calls, what the context gives a migration, transactions, failures, references between migrations, and how to test one. Read the tutorial first if you have not written a migration yet.

Guides · Migrations: the hub · Migrations · Related: The file format, Rollback and captions, Troubleshooting

## The file and the class

A PHP migration is a file `YYYYMMDDHHMMSS_name.php` with its header and **one namespaced class that extends** `expMigrationPhp`:

**PHP**

```php
<?php
// description: A table for news notes and the News folder
// requires: 20261030090000_news_class
namespace MySite\Migrations;

final class AddNewsNotes extends \expMigrationPhp
{
    public function plan( $context )
    {
        return array( 'create the table news_note', 'create the folder News below the root' );
    }

    public function up( $context )
    {
        // ...
        return array( 'created news_note', 'created the folder News' );
    }
}
```

- The class name is free; the namespace is yours (the site's, or the extension's), so two migrations of different sources never clash. `exp:migration:generate --skeleton` writes `Site\Migrations` for the site and `Extension\<Name>\Migrations` for an extension, and adds the time to the class name.
- The file must declare exactly one class that extends `expMigrationPhp` and is not abstract; a file with none, two, or a class another file declares already is refused (`exp:migration:check` reports it before any run).
- The file is loaded with `require` when the migration is loaded; keep it free of code outside the class.
- The header is described in the file format; in a PHP file it is written with `//`, `#` or a `/** ... */` block before the `namespace` line.

Generate the skeleton for a new one:

Shell

```
./console exp:migration:generate --skeleton add_news_notes --description="A table for news notes"
 --apply
./console exp:migration:generate --skeleton news_folder --source=extension:expnews --apply
```

## The methods

| Method | Called by | Must | Default |
|---|---|---|---|
| `up( $context )` | a run with `--apply`, and the simulation | make the change and return what it did, one string per line; throw on failure | (abstract) |
| `plan( $context )` | the dry run | list what `up()` would do, without changing anything | the description |
| `down( $context )` | `exp:migration:rollback <id>` | undo `up()` and return what it did | throws: not reversible |
| `detect( $context )` | every state lookup | return `DONE`, `PENDING` or `NOT_NEEDED` when the installation can tell, else `null` | `null` (the record decides) |
| `transactional()` | the runner | `false` for a migration that commits by itself (a long batch) | `true` |

Declare these methods as the table shows them: public, not static, with their one parameter `$context` untyped (more parameters only with a default). The other methods of `expMigrationPhp` (`definition()`, `file()`, `id()`, `source()`, `checksum()`, `state()`, `requires()`, `engines()` ...) are its own and final, so a helper method must not have one of their names. PHP would refuse such a class with a fatal error while loading the file; `exp:migration:check` and every run read the class first and report the method by name instead (for example "...::definition() has the name of a method of expMigrationPhp that a migration cannot declare: give it another name"), without loading the file.

`plan()` runs on every dry run and on the status page of a run, so it must be cheap and must not write. Build it from pure functions where you can, so it can be tested without a database.

`up()` returns an array of strings; they appear in the output, the run log, the record (the last message) and the audit event. Return what a reader needs to know, never a secret.

### down() and reversible()

A class that defines `down()` is reversible (`reversible()` says so). Leave `down()` out when a change cannot be undone by itself, for example a removal of content: `exp:migration:rollback <id>` then refuses with "cannot be undone by itself (no down()); restore the caption taken before it", and the way back is the caption of the run (Rollback and captions). `down()` of a migration that is not `done` is refused too. After a successful `down()` the record of the migration is removed and it is pending again.

### detect(): the state without the record

By default a PHP migration is record-based: `done` once a run recorded it, `failed` when the last run failed, `pending` otherwise. A class that can ask the installation itself returns a state from `detect()`:

PHP

```php
public function detect( $context )
{
    return \expKernelTableSchema::exists( 'exp_news_note', $context->db(), true )
        ? \expMigration::DONE : null;
}
```

`null` falls back to the record. A migration that reports `NOT_NEEDED` or `DONE` this way is not run; this is how the kernel's own steps (and `expWebsiteSidebarMigration`) tell an installation that already has the change from one that needs it.

## The context

Every method gets the run's context, an `expMigrationContext`. Use it, never a connection or a user of your own, so the runner's transaction, the test's database and the run's user are the ones in effect.

| Call | Gives |
|---|---|
| `$context->db()` | the database connection (`eZDBInterface`) |
| `$context->engine()` | `mysql`, `postgresql`, `sqlite`, `oracle` or `mongodb` |
| `$context->isApply()` | `true` in a run that writes, `false` in a dry run |
| `$context->environment()` | the environment of the run (`site.ini [AdminEnvironment] Level`, `EXP_ENV`, or `--environment`) |
| `$context->runId()` | the run's id, `YYYYMMDD-HHMMSS-<6 hex>` |
| `$context->user()`, `$context->userLogin()` | the user the run runs as (`eZUser`) and its login |
| `$context->setReference( $name, $value )` | leaves a value for the migrations after this one in the same run |
| `$context->reference( $name, $default )`, `hasReference( $name )`, `references()` | reads them |
| `$context->log( $level, $text )` | a line in the run log and in `var/log/migration.log` (`debug`, `info`, `notice`, `warning`, `error`), cleaned of secrets |
| `$context->option( $name )` | an option of the run (`force`, `reason`, `only` ...) |
| `$context->schema()`, `$context->table( $name )` | the kernel schema (`share/db_schema.dba`) |

References live for one run. A later run that needs a value finds it again in the installation (by remote id, by identifier), not in a reference; that is also what makes a migration work on every installation, where ids differ.

During `up()` and `down()` the current user is the run's user, so content is created and owned by that user and policies apply as for that user. On the console that is `migration.ini [RunSettings] RunAsUser` (`admin`), or `--login=<login>`.

## Transactions

Each migration runs in its own transaction where the database has one:

| Database | `transactions:` in the output | What a failed migration leaves |
|---|---|---|
| PostgreSQL, SQLite | `full` | nothing: schema and data are undone |
| MySQL/MariaDB, Oracle | `data` | its schema changes (they commit at once); its data is undone |
| MongoDB | `none` | everything it wrote |

The transaction is committed when `up()` returns. While it is open, a failing statement throws instead of ending the process. A migration that must commit by itself, for example a batch of thousands of objects, returns `false` from `transactional()` and is responsible for being safe to run again. Whatever a transaction cannot undo, the caption taken before the run can: after a failure it is restored by default.

For such a batch, `$this->batches( $context, $fetch, $handle, $size )` commits every batch on its own and keeps how far it got in the migration's checkpoint, so the next run goes on from there: PHP migrations in batches.

Order your own steps so that a failure halfway does the least harm: create a table before writing to it, write data before the schema change that depends on it on MySQL. Split a large change into several migrations.

## Failure

Throw an exception from `up()`. The runner:

1. rolls back the migration's transaction (as far as the database can);
2. records the migration as `failed` with the exception's message;
3. stops the run (`[RunSettings] StopOnFailure=enabled`); the migrations after it are `not-run`;
4. takes a caption of the failed state (`migration-failure`) and restores the pre caption;
5. writes, commits and pushes the record of the failed run; exits with 1.

With `StopOnFailure=disabled` the run goes on with the migrations that do not require the failed one; those that do are `blocked`. A failed migration runs again on the next run, or with `exp:migration:resume`.

## Maintenance mode

A migration whose header says `maintenance: yes` switches maintenance mode on before it runs and off again after it (only when the run switched it on; a site that was in maintenance mode already stays in it). Use it for changes visitors must not see half done.

## Events and audit

The runner notifies `ezpEvent` listeners, registered in `site.ini` `[Event] Listeners[]=<event>@<callback>`; a listener that throws does not stop the run. The parameters, by position:

| Event | Parameters |
| --- | --- |
| `migration/run/before` | `string $runId`, `string[] $ids`, `expMigrationContext $context` |
| `migration/step/before` | `string $runId`, `expMigration $migration`, `expMigrationContext $context` |
| `migration/step/after` | `string $runId`, `expMigration $migration`, `array $entry`, `expMigrationContext $context` |
| `migration/run/after` | `string $runId`, `array $report`, `expMigrationContext $context` |

The audit log records `system.migration.run`, `system.migration.apply` (also a changed file accepted with `--force --reason`), `system.migration.fail`, `system.migration.rollback`, `system.migration.caption` and `system.migration.push` (enabled in `settings/audit.ini`).

## Migrations in an extension

Put them in `extension/<name>/migrations/` (the folder name is `[MigrationSettings] ExtensionDirectory`). They are found while the extension is active, with the source `extension:<name>`. A site migration that needs one names it: `requires: 20261101090000_expnews_tables` or `requires: extension:expnews` (the extension must be active). For changes every installation of an extension needs on upgrade, an upgrade step is still the right tool.

## Testing a migration

- `./console exp:migration:check` (or `--path=<file or folder>`) in CI: syntax, header, order, the class loads.
- `./console exp:migration:dry-run --simulate --diff` on a PostgreSQL or SQLite copy of the site: the migration runs for real and is rolled back.
- A PHPUnit test can load the file and run it on a throwaway SQLite database, as the kernel's own tests do (`tests/tests/kernel/classes/migration/expMigrationRunnerSqliteTest.php`):

PHP

```php
$locator = new expMigrationLocator( array( array( 'source' => 'site', 'path' => __DIR__ . '/migrations' ) ) );
$errors = array();
$migrations = expMigrationRunner::load( ( new expMigrationGraph( $locator->find() ) )->migrations(), $errors );
$context = new expMigrationContext( $db, true, array( 'environment' => 'testing', 'no-mirror' => true,
                                                      'user' => eZUser::fetchByName( 'admin' ) ) );
$report = ( new expMigrationRunner( $context ) )->run( $migrations );
$this->assertSame( 0, $report['exit'], implode( '; ', $report['errors'] ) );
```

Test `plan()` without a database by building it from pure functions, and run `up()` and `down()` twice in a test to prove a migration can be taken back and applied again.

## Related pages

- Migrations: the hub, All recipes, Frequently asked questions
- The guide, the tutorial, Rollback and captions, The migrations repository
- In the administration, Commands, Settings, Policies, views and events
- The file format, The feature
- When a migration fails, Troubleshooting messages
- Migrations versus migrating a site

Exponential Copyright © 1998-2026 Exponential Foundation and 7x
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".
This documentation is free documentation, licensed under the GNU Free Documentation License 1.3 or later:
https://www.gnu.org/licenses/fdl-1.3.html.