Phinx Documentation

Release 0.8.1

Rob Morgan

Jun 21, 2017

Contents

Phinx makes it ridiculously easy to manage the database migrations for your PHP app. In less than 5 minutes, you can install Phinx using Composer and create your first database migration. Phinx is just about migrations without all the bloat of a database ORM system or application framework.

CHAPTER 1

Contents

Introduction

Good developers always version their code using a SCM system, so why don't they do the same for their database schema?

Phinx allows developers to alter and manipulate databases in a clear and concise way. It avoids the use of writing SQL by hand and instead offers a powerful API for creating migrations using PHP code. Developers can then version these migrations using their preferred SCM system. This makes Phinx migrations portable between different database systems. Phinx keeps track of which migrations have been run, so you can worry less about the state of your database and instead focus on building better software.

Goals

Phinx was developed with the following goals in mind:

- Be portable amongst the most popular database vendors.
- Be PHP framework independent.
- Have a simple install process.
- Have an easy to use command-line operation.
- Integrate with various other PHP tools (Phing, PHPUnit) and web frameworks.

Installation

Phinx should be installed using Composer, which is a tool for dependency management in PHP. Please visit the Composer website for more information.

Note: Phinx requires at least PHP 5.4 (or later).

To install Phinx, simply require it using Composer:

```
php composer.phar require robmorgan/phinx
```

Create folders in your project following the structure `db/migrations` with adequate permissions. It is where your migration files will live and should be writable.

Phinx can now be executed from within your project:

```
vendor/bin/phinx init
```

Writing Migrations

Phinx relies on migrations in order to transform your database. Each migration is represented by a PHP class in a unique file. It is preferred that you write your migrations using the Phinx PHP API, but raw SQL is also supported.

Creating a New Migration

Generating a skeleton migration file

Let's start by creating a new Phinx migration. Run Phinx using the `create` command:

```
$ php vendor/bin/phinx create MyNewMigration
```

This will create a new migration in the format `YYYYMMDDHHMMSS_my_new_migration.php`, where the first 14 characters are replaced with the current timestamp down to the second.

If you have specified multiple migration paths, you will be asked to select which path to create the new migration in.

Phinx automatically creates a skeleton migration file with a single method:

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Change Method.
     *
     * Write your reversible migrations using this method.
     *
     * More information on writing migrations is available here:
     * http://docs.phinx.org/en/latest/migrations.html#the-abstractmigration-class
     *
     * The following commands can be used in this method and Phinx will
     * automatically reverse them when rolling back:
     *
     *    createTable
     *    renameTable
     *    addColumn
```

```
     *     renameColumn
     *     addIndex
     *     addForeignKey
     *
     * Remember to call "create()" or "update()" and NOT "save()" when working
     * with the Table class.
     */
    public function change()
    {

    }
}
```

All Phinx migrations extend from the `AbstractMigration` class. This class provides the necessary support to create your database migrations. Database migrations can transform your database in many ways, such as creating new tables, inserting rows, adding indexes and modifying columns.

The Change Method

Phinx 0.2.0 introduced a new feature called reversible migrations. This feature has now become the default migration method. With reversible migrations, you only need to define the `up` logic, and Phinx can figure out how to migrate down automatically for you. For example:

```
<?php

use Phinx\Migration\AbstractMigration;

class CreateUserLoginsTable extends AbstractMigration
{
    /**
     * Change Method.
     *
     * More information on this method is available here:
     * http://docs.phinx.org/en/latest/migrations.html#the-change-method
     *
     * Uncomment this method if you would like to use it.
     */
    public function change()
    {
        // create the table
        $table = $this->table('user_logins');
        $table->addColumn('user_id', 'integer')
              ->addColumn('created', 'datetime')
              ->create();
    }

    /**
     * Migrate Up.
     */
    public function up()
    {

    }

    /**
     * Migrate Down.
```

```
     */
    public function down()
    {

    }
}
```

When executing this migration, Phinx will create the `user_logins` table on the way up and automatically figure out how to drop the table on the way down. Please be aware that when a `change` method exists, Phinx will automatically ignore the `up` and `down` methods. If you need to use these methods it is recommended to create a separate migration file.

Note: When creating or updating tables inside a `change()` method you must use the Table `create()` and `update()` methods. Phinx cannot automatically determine whether a `save()` call is creating a new table or modifying an existing one.

Phinx can only reverse the following commands:

- createTable
- renameTable
- addColumn
- renameColumn
- addIndex
- addForeignKey

If a command cannot be reversed then Phinx will throw a `IrreversibleMigrationException` exception when it's migrating down.

The Up Method

The up method is automatically run by Phinx when you are migrating up and it detects the given migration hasn't been executed previously. You should use the up method to transform the database with your intended changes.

The Down Method

The down method is automatically run by Phinx when you are migrating down and it detects the given migration has been executed in the past. You should use the down method to reverse/undo the transformations described in the up method.

Executing Queries

Queries can be executed with the `execute()` and `query()` methods. The `execute()` method returns the number of affected rows whereas the `query()` method returns the result as a PDOStatement

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
```

```
    /**
     * Migrate Up.
     */
    public function up()
    {
        // execute()
        $count = $this->execute('DELETE FROM users'); // returns the number of␣
↪affected rows

        // query()
        $rows = $this->query('SELECT * FROM users'); // returns the result as an array
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

Note: These commands run using the PHP Data Objects (PDO) extension which defines a lightweight, consistent interface for accessing databases in PHP. Always make sure your queries abide with PDOs before using the `execute()` command. This is especially important when using DELIMITERs during insertion of stored procedures or triggers which don't support DELIMITERs.

Warning: When using `execute()` or `query()` with a batch of queries, PDO doesn't throw an exception if there is an issue with one or more of the queries in the batch.

As such, the entire batch is assumed to have passed without issue.

If Phinx was to iterate any potential result sets, looking to see if one had an error, then Phinx would be denying access to all the results as there is no facility in PDO to get a previous result set nextRowset() - but no `previousSet()`).

So, as a consequence, due to the design decision in PDO to not throw an exception for batched queries, Phinx is unable to provide the fullest support for error handling when batches of queries are supplied.

Fortunately though, all the features of PDO are available, so multiple batches can be controlled within the migration by calling upon nextRowset() and examining errorInfo.

Fetching Rows

There are two methods available to fetch rows. The `fetchRow()` method will fetch a single row, whilst the `fetchAll()` method will return multiple rows. Both methods accept raw SQL as their only parameter.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
```

```
    /**
     * Migrate Up.
     */
    public function up()
    {
        // fetch a user
        $row = $this->fetchRow('SELECT * FROM users');

        // fetch an array of messages
        $rows = $this->fetchAll('SELECT * FROM messages');
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

Inserting Data

Phinx makes it easy to insert data into your tables. Whilst this feature is intended for the *seed feature*, you are also free to use the insert methods in your migrations.

```
<?php

use Phinx\Migration\AbstractMigration;

class NewStatus extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        // inserting only one row
        $singleRow = [
            'id'    => 1,
            'name'  => 'In Progress'
        ];

        $table = $this->table('status');
        $table->insert($singleRow);
        $table->saveData();

        // inserting multiple rows
        $rows = [
            [
              'id'    => 2,
              'name'  => 'Stopped'
            ],
            [
              'id'    => 3,
              'name'  => 'Queued'
```

```
            ]
        ];

        // this is a handy shortcut
        $this->insert('status', $rows);
    }

    /**
     * Migrate Down.
     */
    public function down()
    {
        $this->execute('DELETE FROM status');
    }
}
```

Note: You cannot use the insert methods inside a *change()* method. Please use the *up()* and *down()* methods.

Working With Tables

The Table Object

The Table object is one of the most useful APIs provided by Phinx. It allows you to easily manipulate database tables using PHP code. You can retrieve an instance of the Table object by calling the `table()` method from within your database migration.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('tableName');
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

You can then manipulate this table using the methods provided by the Table object.

The Save Method

When working with the Table object, Phinx stores certain operations in a pending changes cache.

When in doubt, it is recommended you call this method. It will commit any pending changes to the database.

Creating a Table

Creating a table is really easy using the Table object. Let's create a table to store a collection of users.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $users = $this->table('users');
        $users->addColumn('username', 'string', array('limit' => 20))
              ->addColumn('password', 'string', array('limit' => 40))
              ->addColumn('password_salt', 'string', array('limit' => 40))
              ->addColumn('email', 'string', array('limit' => 100))
              ->addColumn('first_name', 'string', array('limit' => 30))
              ->addColumn('last_name', 'string', array('limit' => 30))
              ->addColumn('created', 'datetime')
              ->addColumn('updated', 'datetime', array('null' => true))
              ->addIndex(array('username', 'email'), array('unique' => true))
              ->save();
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

Columns are added using the `addColumn()` method. We create a unique index for both the username and email columns using the `addIndex()` method. Finally calling `save()` commits the changes to the database.

Note: Phinx automatically creates an auto-incrementing primary key column called `id` for every table.

The `id` option sets the name of the automatically created identity field, while the `primary_key` option selects the field or fields used for primary key. The `primary_key` option always defaults to the value of `id`. Both can be disabled by setting them to false.

To specify an alternate primary key, you can specify the `primary_key` option when accessing the Table object. Let's disable the automatic `id` column and create a primary key using two columns instead:

```php
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('followers', array('id' => false, 'primary_key' =>␣
↪array('user_id', 'follower_id')));
        $table->addColumn('user_id', 'integer')
              ->addColumn('follower_id', 'integer')
              ->addColumn('created', 'datetime')
              ->save();
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

Setting a single `primary_key` doesn't enable the `AUTO_INCREMENT` option. To simply change the name of the primary key, we need to override the default `id` field name:

```php
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('followers', array('id' => 'user_id'));
        $table->addColumn('follower_id', 'integer')
              ->addColumn('created', 'timestamp', array('default' => 'CURRENT_
↪TIMESTAMP'))
              ->save();
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

In addition, the MySQL adapter supports following options:

Option	Description
comment	set a text comment on the table
engine	define table engine *(defaults to ''InnoDB'')*
collation	define table collation *(defaults to ''utf8_general_ci'')*
signed	whether the primary key is `signed`

By default the primary key is `signed`. To simply set it to unsigned just pass `signed` option with a `false` value:

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('followers', array('signed' => false));
        $table->addColumn('follower_id', 'integer')
              ->addColumn('created', 'timestamp', array('default' => 'CURRENT_
↪TIMESTAMP'))
              ->save();
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

Valid Column Types

Column types are specified as strings and can be one of:

- biginteger
- binary
- boolean
- date
- datetime
- decimal
- float
- integer
- string
- text

- time
- timestamp
- uuid

In addition, the MySQL adapter supports `enum`, `set`, `blob` and `json` column types. (`json` in MySQL 5.7 and above)

In addition, the Postgres adapter supports `smallint`, `json`, `jsonb` and `uuid` column types (PostgreSQL 9.3 and above).

For valid options, see the *Valid Column Options* below.

Determining Whether a Table Exists

You can determine whether or not a table exists by using the `hasTable()` method.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $exists = $this->hasTable('users');
        if ($exists) {
            // do something
        }
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

Dropping a Table

Tables can be dropped quite easily using the `dropTable()` method. It is a good idea to recreate the table again in the `down()` method.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
```

```
    public function up()
    {
        $this->dropTable('users');
    }

    /**
     * Migrate Down.
     */
    public function down()
    {
        $users = $this->table('users');
        $users->addColumn('username', 'string', array('limit' => 20))
              ->addColumn('password', 'string', array('limit' => 40))
              ->addColumn('password_salt', 'string', array('limit' => 40))
              ->addColumn('email', 'string', array('limit' => 100))
              ->addColumn('first_name', 'string', array('limit' => 30))
              ->addColumn('last_name', 'string', array('limit' => 30))
              ->addColumn('created', 'datetime')
              ->addColumn('updated', 'datetime', array('null' => true))
              ->addIndex(array('username', 'email'), array('unique' => true))
              ->save();
    }
}
```

Renaming a Table

To rename a table access an instance of the Table object then call the `rename()` method.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('users');
        $table->rename('legacy_users');
    }

    /**
     * Migrate Down.
     */
    public function down()
    {
        $table = $this->table('legacy_users');
        $table->rename('users');
    }
}
```

Working With Columns

Valid Column Types

Column types are specified as strings and can be one of:

- biginteger
- binary
- boolean
- char
- date
- datetime
- decimal
- float
- integer
- string
- text
- time
- timestamp
- uuid

In addition, the MySQL adapter supports `enum`, `set` and `blob` column types.

In addition, the Postgres adapter supports `smallint`, `json`, `jsonb` and `uuid` column types (PostgreSQL 9.3 and above).

Valid Column Options

The following are valid column options:

For any column type:

Option	Description
limit	set maximum length for strings, also hints column types in adapters (see note below)
length	alias for `limit`
default	set default value or action
null	allow `NULL` values (should not be used with primary keys!)
after	specify the column that a new column should be placed after
comment	set a text comment on the column

For `decimal` columns:

Option	Description
precision	combine with `scale` set to set decimal accuracy
scale	combine with `precision` to set decimal accuracy
signed	enable or disable the `unsigned` option *(only applies to MySQL)*

For `enum` and `set` columns:

Option	Description
values	Can be a comma separated list or an array of values

For `integer` and `biginteger` columns:

Option	Description
identity	enable or disable automatic incrementing
signed	enable or disable the `unsigned` option *(only applies to MySQL)*

For `timestamp` columns:

Option	Description
default	set default value (use with `CURRENT_TIMESTAMP`)
update	set an action to be triggered when the row is updated (use with `CURRENT_TIMESTAMP`)
time-zone	enable or disable the `with time zone` option for `time` and `timestamp` columns *(only applies to Postgres)*

You can add `created_at` and `updated_at` timestamps to a table using the `addTimestamps()` method. This method also allows you to supply alternative names.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Change.
     */
    public function change()
    {
        // Override the 'updated_at' column name with 'amended_at'.
        $table = $this->table('users')->addTimestamps(null, 'amended_at')->create();
    }
}
```

For `boolean` columns:

Option	Description
signed	enable or disable the `unsigned` option *(only applies to MySQL)*

For `string` and `text` columns:

Option	Description
collation	set collation that differs from table defaults *(only applies to MySQL)*
encoding	set character set that differs from table defaults *(only applies to MySQL)*

For foreign key definitions:

Option	Description
update	set an action to be triggered when the row is updated
delete	set an action to be triggered when the row is deleted

You can pass one or more of these options to any column with the optional third argument array.

Limit Option and PostgreSQL

When using the PostgreSQL adapter, additional hinting of database column type can be made for `integer` columns. Using `limit` with one the following options will modify the column type accordingly:

Limit	Column Type
INT_SMALL	SMALLINT

```
use Phinx\Db\Adapter\PostgresAdapter;
```

```
//...

$table = $this->table('cart_items');
$table->addColumn('user_id', 'integer')
      ->addColumn('subtype_id', 'integer', array('limit' => PostgresAdapter::INT_
↪SMALL))
      ->create();
```

Limit Option and MySQL

When using the MySQL adapter, additional hinting of database column type can be made for `integer`, `text` and `blob` columns. Using `limit` with one the following options will modify the column type accordingly:

Limit	Column Type
BLOB_TINY	TINYBLOB
BLOB_REGULAR	BLOB
BLOB_MEDIUM	MEDIUMBLOB
BLOB_LONG	LONGBLOB
TEXT_TINY	TINYTEXT
TEXT_REGULAR	TEXT
TEXT_MEDIUM	MEDIUMTEXT
TEXT_LONG	LONGTEXT
INT_TINY	TINYINT
INT_SMALL	SMALLINT
INT_MEDIUM	MEDIUMINT
INT_REGULAR	INT
INT_BIG	BIGINT

```
use Phinx\Db\Adapter\MysqlAdapter;

//...

$table = $this->table('cart_items');
$table->addColumn('user_id', 'integer')
      ->addColumn('product_id', 'integer', array('limit' => MysqlAdapter::INT_BIG))
      ->addColumn('subtype_id', 'integer', array('limit' => MysqlAdapter::INT_SMALL))
      ->addColumn('quantity', 'integer', array('limit' => MysqlAdapter::INT_TINY))
      ->create();
```

Get a column list

To retrieve all table columns, simply create a *table* object and call *getColumns()* method. This method will return an array of Column classes with basic info. Example below:

```
<?php

use Phinx\Migration\AbstractMigration;

class ColumnListMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
```

```
    {
        $columns = $this->table('users')->getColumns();
        ...
    }

    /**
     * Migrate Down.
     */
    public function down()
    {
        ...
    }
}
```

Checking whether a column exists

You can check if a table already has a certain column by using the `hasColumn()` method.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Change Method.
     */
    public function change()
    {
        $table = $this->table('user');
        $column = $table->hasColumn('username');

        if ($column) {
            // do something
        }

    }
}
```

Renaming a Column

To rename a column, access an instance of the Table object then call the `renameColumn()` method.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('users');
```

```
        $table->renameColumn('bio', 'biography');
    }

    /**
     * Migrate Down.
     */
    public function down()
    {
        $table = $this->table('users');
        $table->renameColumn('biography', 'bio');
    }
}
```

Adding a Column After Another Column

When adding a column you can dictate its position using the `after` option.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Change Method.
     */
    public function change()
    {
        $table = $this->table('users');
        $table->addColumn('city', 'string', array('after' => 'email'))
              ->update();
    }
}
```

Dropping a Column

To drop a column, use the `removeColumn()` method.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate up.
     */
    public function up()
    {
        $table = $this->table('users');
        $table->removeColumn('short_name')
              ->save();
    }
}
```

Specifying a Column Limit

You can limit the maximum length of a column by using the `limit` option.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Change Method.
     */
    public function change()
    {
        $table = $this->table('tags');
        $table->addColumn('short_name', 'string', array('limit' => 30))
              ->update();
    }
}
```

Changing Column Attributes

To change column type or options on an existing column, use the `changeColumn()` method. See **'Valid Column Types'_** and *Valid Column Options* for allowed values.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $users = $this->table('users');
        $users->changeColumn('email', 'string', array('limit' => 255))
              ->save();
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

Working With Indexes

To add an index to a table you can simply call the `addIndex()` method on the table object.

```php
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('users');
        $table->addColumn('city', 'string')
              ->addIndex(array('city'))
              ->save();
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

By default Phinx instructs the database adapter to create a normal index. We can pass an additional parameter `unique` to the `addIndex()` method to specify a unique index. We can also explicitly specify a name for the index using the `name` parameter.

```php
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('users');
        $table->addColumn('email', 'string')
              ->addIndex(array('email'), array('unique' => true, 'name' => 'idx_users_
↪email'))
              ->save();
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

The MySQL adapter also supports `fulltext` indexes. If you are using a version before 5.6 you must ensure the

table uses the `MyISAM` engine.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    public function change()
    {
        $table = $this->table('users', ['engine' => 'MyISAM']);
        $table->addColumn('email', 'string')
              ->addIndex('email', ['type' => 'fulltext'])
              ->create();
    }
}
```

Removing indexes is as easy as calling the `removeIndex()` method. You must call this method for each index.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('users');
        $table->removeIndex(array('email'));

        // alternatively, you can delete an index by its name, ie:
        $table->removeIndexByName('idx_users_email');
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

Note: There is no need to call the `save()` method when using `removeIndex()`. The index will be removed immediately.

Working With Foreign Keys

Phinx has support for creating foreign key constraints on your database tables. Let's add a foreign key to an example table:

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('tags');
        $table->addColumn('tag_name', 'string')
              ->save();

        $refTable = $this->table('tag_relationships');
        $refTable->addColumn('tag_id', 'integer')
                 ->addForeignKey('tag_id', 'tags', 'id', array('delete'=> 'SET_NULL',
↪'update'=> 'NO_ACTION'))
                 ->save();

    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

"On delete" and "On update" actions are defined with a 'delete' and 'update' options array. Possibles values are 'SET_NULL', 'NO_ACTION', 'CASCADE' and 'RESTRICT'. Constraint name can be changed with the 'constraint' option.

It is also possible to pass `addForeignKey()` an array of columns. This allows us to establish a foreign key relationship to a table which uses a combined key.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('follower_events');
        $table->addColumn('user_id', 'integer')
              ->addColumn('follower_id', 'integer')
              ->addColumn('event_id', 'integer')
              ->addForeignKey(array('user_id', 'follower_id'),
                              'followers',
                              array('user_id', 'follower_id'),
                              array('delete'=> 'NO_ACTION', 'update'=> 'NO_ACTION',
↪'constraint' => 'user_follower_id'))
```

```
                ->save();
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

We can add named foreign keys using the `constraint` parameter. This feature is supported as of Phinx version 0.6.5

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('your_table');
        $table->addForeignKey('foreign_id', 'reference_table', array('id'),
                              array('constraint'=>'your_foreign_key_name'));
              ->save();
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

We can also easily check if a foreign key exists:

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('tag_relationships');
        $exists = $table->hasForeignKey('tag_id');
        if ($exists) {
            // do something
```

```
        }
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

Finally, to delete a foreign key, use the `dropForeignKey` method.

```
<?php

use Phinx\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{
    /**
     * Migrate Up.
     */
    public function up()
    {
        $table = $this->table('tag_relationships');
        $table->dropForeignKey('tag_id');
    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

Database Seeding

In version 0.5.0 Phinx introduced support for seeding your database with test data. Seed classes are a great way to easily fill your database with data after it's created. By default they are stored in the *seeds* directory; however, this path can be changed in your configuration file.

Note: Database seeding is entirely optional, and Phinx does not create a *seeds* directory by default.

Creating a New Seed Class

Phinx includes a command to easily generate a new seed class:

```
$ php vendor/bin/phinx seed:create UserSeeder
```

If you have specified multiple seed paths, you will be asked to select which path to create the new seed class in.

It is based on a skeleton template:

```
<?php

use Phinx\Seed\AbstractSeed;

class MyNewSeeder extends AbstractSeed
{
    /**
     * Run Method.
     *
     * Write your database seeder using this method.
     *
     * More information on writing seeders is available here:
     * http://docs.phinx.org/en/latest/seeding.html
     */
    public function run()
    {

    }
}
```

The AbstractSeed Class

All Phinx seeds extend from the `AbstractSeed` class. This class provides the necessary support to create your seed classes. Seed classes are primarily used to insert test data.

The Run Method

The run method is automatically invoked by Phinx when you execute the *seed:run* command. You should use this method to insert your test data.

Note: Unlike with migrations, Phinx does not keep track of which seed classes have been run. This means database seeders can be run repeatedly. Keep this in mind when developing them.

Inserting Data

Using The Table Object

Seed classes can also use the familiar *Table* object to insert data. You can retrieve an instance of the Table object by calling the `table()` method from within your seed class and then use the *insert()* method to insert data:

```
<?php

use Phinx\Seed\AbstractSeed;

class PostsSeeder extends AbstractSeed
{
    public function run()
    {
```

```
        $data = array(
            array(
                'body'    => 'foo',
                'created' => date('Y-m-d H:i:s'),
            ),
            array(
                'body'    => 'bar',
                'created' => date('Y-m-d H:i:s'),
            )
        );

        $posts = $this->table('posts');
        $posts->insert($data)
              ->save();
    }
}
```

Note: You must call the *save()* method to commit your data to the table. Phinx will buffer data until you do so.

Integrating with the Faker library

It's trivial to use the awesome Faker library in your seed classes. Simply install it using Composer:

```
$ composer require fzaninotto/faker
```

Then use it in your seed classes:

```
<?php

use Phinx\Seed\AbstractSeed;

class UserSeeder extends AbstractSeed
{
    public function run()
    {
        $faker = Faker\Factory::create();
        $data = [];
        for ($i = 0; $i < 100; $i++) {
            $data[] = [
                'username'      => $faker->userName,
                'password'      => sha1($faker->password),
                'password_salt' => sha1('foo'),
                'email'         => $faker->email,
                'first_name'    => $faker->firstName,
                'last_name'     => $faker->lastName,
                'created'       => date('Y-m-d H:i:s'),
            ];
        }

        $this->insert('users', $data);
    }
}
```

Truncating Tables

In addition to inserting data Phinx makes it trivial to empty your tables using the SQL *TRUNCATE* command:

```
<?php

use Phinx\Seed\AbstractSeed;

class UserSeeder extends AbstractSeed
{
    public function run()
    {
        $data = [
            [
                'body'    => 'foo',
                'created' => date('Y-m-d H:i:s'),
            ],
            [
                'body'    => 'bar',
                'created' => date('Y-m-d H:i:s'),
            ]
        ];

        $posts = $this->table('posts');
        $posts->insert($data)
              ->save();

        // empty the table
        $posts->truncate();
    }
}
```

Note: SQLite doesn't natively support the *TRUNCATE* command so behind the scenes *DELETE FROM* is used. It is recommended to call the *VACUUM* command after truncating a table. Phinx does not do this automatically.

Executing Seed Classes

This is the easy part. To seed your database, simply use the *seed:run* command:

```
$ php vendor/bin/phinx seed:run
```

By default, Phinx will execute all available seed classes. If you would like to run a specific class, simply pass in the name of it using the *-s* parameter:

```
$ php vendor/bin/phinx seed:run -s UserSeeder
```

You can also run multiple seeders:

```
$ php vendor/bin/phinx seed:run -s UserSeeder -s PermissionSeeder -s LogSeeder
```

You can also use the *-v* parameter for more output verbosity:

```
$ php vendor/bin/phinx seed:run -v
```

The Phinx seed functionality provides a simple mechanism to easily and repeatably insert test data into your database.

Commands

Phinx is run using a number of commands.

The Breakpoint Command

The Breakpoint command is used to set breakpoints, allowing you to limit rollbacks. You can toggle the breakpoint of the most recent migration by not supplying any parameters.

```
$ phinx breakpoint -e development
```

To toggle a breakpoint on a specific version then use the `--target` parameter or `-t` for short.

```
$ phinx breakpoint -e development -t 20120103083322
```

You can remove all the breakpoints by using the `--remove-all` parameter or `-r` for short.

```
$ phinx breakpoint -e development -r
```

Breakpoints are visible when you run the `status` command.

The Create Command

The Create command is used to create a new migration file. It requires one argument: the name of the migration. The migration name should be specified in CamelCase format.

```
$ phinx create MyNewMigration
```

Open the new migration file in your text editor to add your database transformations. Phinx creates migration files using the path specified in your `phinx.yml` file. Please see the *Configuration* chapter for more information.

You are able to override the template file used by Phinx by supplying an alternative template filename.

```
$ phinx create MyNewMigration --template="<file>"
```

You can also supply a template generating class. This class must implement the interface `Phinx\Migration\CreationInterface`.

```
$ phinx create MyNewMigration --class="<class>"
```

In addition to providing the template for the migration, the class can also define a callback that will be called once the migration file has been generated from the template.

You cannot use `--template` and `--class` together.

The Init Command

The Init command (short for initialize) is used to prepare your project for Phinx. This command generates the `phinx.yml` file in the root of your project directory.

```
$ cd yourapp
$ phinx init .
```

Open this file in your text editor to setup your project configuration. Please see the *Configuration* chapter for more information.

The Migrate Command

The Migrate command runs all of the available migrations, optionally up to a specific version.

```
$ phinx migrate -e development
```

To migrate to a specific version then use the `--target` parameter or `-t` for short.

```
$ phinx migrate -e development -t 20110103081132
```

The Rollback Command

The Rollback command is used to undo previous migrations executed by Phinx. It is the opposite of the Migrate command.

You can rollback to the previous migration by using the `rollback` command with no arguments.

```
$ phinx rollback -e development
```

To rollback all migrations to a specific version then use the `--target` parameter or `-t` for short.

```
$ phinx rollback -e development -t 20120103083322
```

Specifying 0 as the target version will revert all migrations.

```
$ phinx rollback -e development -t 0
```

To rollback all migrations to a specific date then use the `--date` parameter or `-d` for short.

```
$ phinx rollback -e development -d 2012
$ phinx rollback -e development -d 201201
$ phinx rollback -e development -d 20120103
$ phinx rollback -e development -d 2012010312
$ phinx rollback -e development -d 201201031205
$ phinx rollback -e development -d 20120103120530
```

If a breakpoint is set, blocking further rollbacks, you can override the breakpoint using the `--force` parameter or `-f` for short.

```
$ phinx rollback -e development -t 0 -f
```

Note: When rolling back, Phinx orders the executed migrations using the order specified in the `version_order` option of your `phinx.yml` file. Please see the *Configuration* chapter for more information.

The Status Command

The Status command prints a list of all migrations, along with their current status. You can use this command to determine which migrations have been run.

```
$ phinx status -e development
```

This command exits with code 0 if the database is up-to-date (ie. all migrations are up) or one of the following codes otherwise:

- 1: There is at least one down migration.
- 2: There is at least one missing migration.

The Seed Create Command

The Seed Create command can be used to create new database seed classes. It requires one argument, the name of the class. The class name should be specified in CamelCase format.

```
$ phinx seed:create MyNewSeeder
```

Open the new seed file in your text editor to add your database seed commands. Phinx creates seed files using the path specified in your `phinx.yml` file. Please see the *Configuration* chapter for more information.

The Seed Run Command

The Seed Run command runs all of the available seed classes or optionally just one.

```
$ phinx seed:run -e development
```

To run only one seed class use the `--seed` parameter or `-s` for short.

```
$ phinx seed:run -e development -s MyNewSeeder
```

Configuration File Parameter

When running Phinx from the command line, you may specify a configuration file using the `--configuration` or `-c` parameter. In addition to YAML, the configuration file may be the computed output of a PHP file as a PHP array:

```php
<?php
    return array(
        "paths" => array(
            "migrations" => "application/migrations"
        ),
        "environments" => array(
            "default_migration_table" => "phinxlog",
            "default_database" => "dev",
            "dev" => array(
                "adapter" => "mysql",
                "host" => $_ENV['DB_HOST'],
                "name" => $_ENV['DB_NAME'],
                "user" => $_ENV['DB_USER'],
                "pass" => $_ENV['DB_PASS'],
                "port" => $_ENV['DB_PORT']
            )
        )
    );
```

Phinx auto-detects which language parser to use for files with `*.yml` and `*.php` extensions. The appropriate parser may also be specified via the `--parser` and `-p` parameters. Anything other than `"php"` is treated as YAML.

When using a PHP array, you can provide a `connection` key with an existing PDO instance. It is also important to pass the database name too, as Phinx requires this for certain methods such as `hasTable()`:

```
<?php
    return array(
        "paths" => array(
            "migrations" => "application/migrations"
        ),
        "environments" => array(
            "default_migration_table" => "phinxlog",
            "default_database" => "dev",
            "dev" => array(
                "name" => "dev_db",
                "connection" => $pdo_instance
            )
        )
    );
```

Running Phinx in a Web App

Phinx can also be run inside of a web application by using the `Phinx\Wrapper\TextWrapper` class. An example of this is provided in `app/web.php`, which can be run as a standalone server:

```
$ php -S localhost:8000 vendor/robmorgan/phinx/app/web.php
```

This will create local web server at http://localhost:8000 which will show current migration status by default. To run migrations up, use http://localhost:8000/migrate and to rollback use http://localhost:8000/rollback.

The included web app is only an example and should not be used in production!

Note: To modify configuration variables at runtime and override `%%PHINX_DBNAME%%` or other another dynamic option, set `$_SERVER['PHINX_DBNAME']` before running commands. Available options are documented in the Configuration page.

Configuration

When you initialize your project using the *Init Command*, Phinx creates a default file called `phinx.yml` in the root of your project directory. This file uses the YAML data serialization format.

If a `--configuration` command line option is given, Phinx will load the specified file. Otherwise, it will attempt to find `phinx.php`, `phinx.json` or `phinx.yml` and load the first file found. See the *Commands* chapter for more information.

Warning: Remember to store the configuration file outside of a publicly accessible directory on your webserver. This file contains your database credentials and may be accidentally served as plain text.

Note that while JSON and YAML files are *parsed*, the PHP file is *included*. This means that:

- It must *return* an array of configuration items.

- The variable scope is local, i.e. you would need to explicitly declare any global variables your initialization file reads or modifies.
- Its standard output is suppressed.
- Unlike with JSON and YAML, it is possible to omit environment connection details and instead specify `connection` which must contain an initialized PDO instance. This is useful when you want your migrations to interact with your application and/or share the same connection. However remember to also pass the database name as Phinx cannot infer this from the PDO connection.

```
require 'app/init.php';

global $app;
$pdo = $app->getDatabase()->getPdo();

return array('environments' =>
         array(
           'default_database' => 'development',
           'development' => array(
             'name' => 'devdb',
             'connection' => $pdo
           )
         )
       );
```

Migration Paths

The first option specifies the path to your migration directory. Phinx uses `%%PHINX_CONFIG_DIR%%/db/migrations` by default.

Note: `%%PHINX_CONFIG_DIR%%` is a special token and is automatically replaced with the root directory where your `phinx.yml` file is stored.

In order to overwrite the default `%%PHINX_CONFIG_DIR%%/db/migrations`, you need to add the following to the yaml configuration.

```
paths:
    migrations: /your/full/path
```

You can also provide multiple migration paths by using an array in your configuration:

```
paths:
    migrations:
        - application/module1/migrations
        - application/module2/migrations
```

You can also use the `%%PHINX_CONFIG_DIR%%` token in your path.

```
paths:
    migrations: %%PHINX_CONFIG_DIR%%/your/relative/path
```

Migrations are captured with `glob`, so you can define a pattern for multiple directories.

```
paths:
    migrations: %%PHINX_CONFIG_DIR%%/module/*/{data,scripts}/migrations
```

Custom Migration Base

By default all migrations will extend from Phinx's *AbstractMigration* class. This can be set to a custom class that extends from *AbstractMigration* by setting `migration_base_class` in your config:

```
migration_base_class: MyMagicalMigration
```

Seed Paths

The second option specifies the path to your seed directory. Phinx uses `%%PHINX_CONFIG_DIR%%/db/seeds` by default.

Note: `%%PHINX_CONFIG_DIR%%` is a special token and is automatically replaced with the root directory where your `phinx.yml` file is stored.

In order to overwrite the default `%%PHINX_CONFIG_DIR%%/db/seeds`, you need to add the following to the yaml configuration.

```
paths:
    seeds: /your/full/path
```

You can also provide multiple seed paths by using an array in your configuration:

```
paths:
    seeds:
        - /your/full/path1
        - /your/full/path2
```

You can also use the `%%PHINX_CONFIG_DIR%%` token in your path.

```
paths:
    seeds: %%PHINX_CONFIG_DIR%%/your/relative/path
```

Environments

One of the key features of Phinx is support for multiple database environments. You can use Phinx to create migrations on your development environment, then run the same migrations on your production environment. Environments are specified under the `environments` nested collection. For example:

```
environments:
    default_migration_table: phinxlog
    default_database: development
    production:
        adapter: mysql
        host: localhost
        name: production_db
        user: root
        pass: ''
        port: 3306
        charset: utf8
        collation: utf8_unicode_ci
```

would define a new environment called `production`.

In a situation when multiple developers work on the same project and each has a different environment (e.g. a convention such as `<environment type>-<developer name>-<machine name>`), or when you need to have separate environments for separate purposes (branches, testing, etc) use environment variable *PHINX_ENVIRONMENT* to override the default environment in the yaml file:

```
export PHINX_ENVIRONMENT=dev-`whoami`-`hostname`
```

Table Prefix and Suffix

You can define a table prefix and table suffix:

```
environments:
    development:
        ....
        table_prefix: dev_
        table_suffix: _v1
    testing:
        ....
        table_prefix: test_
        table_suffix: _v2
```

Socket Connections

When using the MySQL adapter, it is also possible to use sockets instead of network connections. The socket path is configured with `unix_socket`:

```
environments:
    default_migration_table: phinxlog
    default_database: development
    production:
        adapter: mysql
        name: production_db
        user: root
        pass: ''
        unix_socket: /var/run/mysql/mysql.sock
        charset: utf8
```

External Variables

Phinx will automatically grab any environment variable prefixed with `PHINX_` and make it available as a token in the config file. The token will have exactly the same name as the variable but you must access it by wrapping two `%%` symbols on either side. e.g: `%%PHINX_DBUSER%%`. This is especially useful if you wish to store your secret database credentials directly on the server and not in a version control system. This feature can be easily demonstrated by the following example:

```
environments:
    default_migration_table: phinxlog
    default_database: development
    production:
        adapter: mysql
        host: %%PHINX_DBHOST%%
```

```
        name: %%PHINX_DBNAME%%
        user: %%PHINX_DBUSER%%
        pass: %%PHINX_DBPASS%%
        port: 3306
        charset: utf8
```

Supported Adapters

Phinx currently supports the following database adapters natively:

- MySQL: specify the `mysql` adapter.
- PostgreSQL: specify the `pgsql` adapter.
- SQLite: specify the `sqlite` adapter.
- SQL Server: specify the `sqlsrv` adapter.

SQLite

Declaring an SQLite database uses a simplified structure:

```
environments:
    development:
        adapter: sqlite
        name: ./data/derby
    testing:
        adapter: sqlite
        memory: true     # Setting memory to *any* value overrides name
```

SQL Server

When using the `sqlsrv` adapter and connecting to a named instance you should omit the `port` setting as SQL Server will negotiate the port automatically. Additionally, omit the `charset: utf8` or change to `charset: 65001` which corresponds to UTF8 for SQL Server.

Custom Adapters

You can provide a custom adapter by registering an implementation of the *Phinx\Db\Adapter\AdapterInterface* with *AdapterFactory*:

```
$name  = 'fizz';
$class = 'Acme\Adapter\FizzAdapter';

AdapterFactory::instance()->registerAdapter($name, $class);
```

Adapters can be registered any time before *$app->run()* is called, which normally called by *bin/phinx*.

Aliases

Template creation class names can be aliased and used with the `--class` command line option for the *Create Command*.

The aliased classes will still be required to implement the `Phinx\Migration\CreationInterface` interface.

```
aliases:
    permission: \Namespace\Migrations\PermissionMigrationTemplateGenerator
    view: \Namespace\Migrations\ViewMigrationTemplateGenerator
```

Version Order

When rolling back or printing the status of migrations, Phinx orders the executed migrations according to the `version_order` option, which can have the following values:

- `creation` (the default): migrations are ordered by their creation time, which is also part of their filename.
- `execution`: migrations are ordered by their execution time, also known as start time.

Copyright

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index