docs.zendframework.com

SQL Abstraction and Object Hydration

1. Docs »
2. MVC Tutorials »
3. In-Depth Tutorial »
4. SQL Abstraction and Object Hydration

In the last chapter, we introduced database abstraction and a new command interface for operations that might change what blog posts we store. We'll now start creating database-backed versions of the `PostRepositoryInterface` and `PostCommandInterface`, demonstrating usage of the various `Zend\Db\Sql` classes.

Preparing the Database

This tutorial assumes you've followed the Getting Started tutorial, and that you've already populated the `data/zftutorial.db` SQLite database. We will be re-using it, and adding another table to it.

Create the file `data/posts.schema.sql` with the following contents:

```
CREATE TABLE posts (id INTEGER PRIMARY KEY AUTOINCREMENT, title
varchar(100) NOT NULL, text TEXT NOT NULL);

INSERT INTO posts (title, text) VALUES ('Blog
INSERT INTO posts (title, text) VALUES ('Blog
INSERT INTO posts (title, text) VALUES ('Blog
INSERT INTO posts (title, text) VALUES ('Blog
INSERT INTO posts (title, text) VALUES ('Blog
```

Now we will execute this against the existing `data/zftutorial.db` SQLite database using the `sqlite` command (or `sqlite3`; check your operating system):

```
$ sqlite data/zftutorial.db < data/posts.schema.sql
```

If you don't have a `sqlite` command, you can populate it using PHP. Create the following script in `data/load_posts.php`:

```
<?php
$db = new PDO('sqlite:' . realpath(__DIR__) . 'zftutorial.db');
$fh = fopen(__DIR__ . '/posts.schema.sql', 'r');
while ($line = fread($fh, 4096)) {
    $line = trim($line);
    $db->exec($line);
}
fclose($fh);
```

and execute it using:

```
$ php data/load_posts.php
```

Quick Facts Zend\Db\Sql

To create queries against a database using `Zend\Db\Sql`, you need to have a database adapter available. The "Getting Started" tutorial covered this in the database chapter, and we can re-use that adapter.

With the adapter in place and the new table populated, we can run queries against the database. The construction of queries is best done through the "QueryBuilder" features of `Zend\Db\Sql` which are `Zend\Db\Sql\Sql` for select queries, `Zend\Db\Sql\Insert` for insert queries, `Zend\Db\Sql\Update` for update queries and `Zend\Db\Sql\Delete` for delete queries. The basic workflow of these components is:

1. Build a query using the relevant class: `Sql`, `Insert`, `Update`, or `Delete`.
2. Create a SQL statement from the `Sql` object.
3. Execute the query.
4. Do something with the result.

Let's start writing database-driven implementations of our interfaces now.

Writing the repository implementation

Create a class named `ZendDbSqlRepository` in the `Blog\Model` namespace that implements `PostRepositoryInterface`; leave the methods empty for now:

```
namespace Blog\Model;

use InvalidArgumentException;
use RuntimeException;

class ZendDbSqlRepository implements PostRepositoryInterface
{

    public function findAllPosts()
    {
    }


    public function findPost($id)
    {
    }
}
```

Now recall what we have learned earlier: for `Zend\Db\Sql` to function, we will need a working implementation of the `AdapterInterface`. This is a *requirement*, and therefore will be injected using *constructor injection*. Create a `__construct()` method that accepts an `AdapterInterface` as its sole parameter, and stores it as an instance property:

```
namespace Blog\Model;

use InvalidArgumentException;
use RuntimeException;
use Zend\Db\Adapter\AdapterInterface;

class ZendDbSqlRepository implements PostRepositoryInterface
{

    private $db;
```

```
    public function __construct(AdapterInterface $db)
    {
        $this->db = $db;
    }


    public function findAllPosts()
    {
    }


    public function findPost($id)
    {
    }
}
```

Whenever we have a required parameter, we need to write a factory for the class. Go ahead and create a factory for our new repository implementation:

```
namespace Blog\Factory;

use Interop\Container\ContainerInterface;
use Blog\Model\ZendDbSqlRepository;
use Zend\Db\Adapter\AdapterInterface;
use Zend\ServiceManager\Factory\FactoryInterface;

class ZendDbSqlRepositoryFactory implements FactoryInterface
{

    public function __invoke(ContainerInterface $container,
$requestedName, array $options = null)
    {
        return new
ZendDbSqlRepository($container->get(AdapterInterface::class));
    }
}
```

We're now able to register our repository implementation as a service. To do so, we'll make two changes:

- Register a factory entry for the new repository.

- Update the existing alias for `PostRepositoryInterface` to point to the new repository.

Update `module/Blog/config/module.config.php` as follows:

```
 return [
    'service_manager' => [
        'aliases' => [

            Model\PostRepositoryInterface::class =>
Model\ZendDbSqlRepository::class,
        ],
        'factories' => [
            Model\PostRepository::class => InvokableFactory::class,

            Model\ZendDbSqlRepository::class =>
Factory\ZendDbSqlRepositoryFactory::class,
        ],
    ],
    'controllers'  => [  ],
    'router'       => [  ],
    'view_manager' => [  ],
];
```

With the adapter in place you're now able to refresh the blog index at `localhost:8080/blog` and you'll notice that the `ServiceNotFoundException` is gone and we get the following PHP Warning:

```
 Warning: Invalid argument supplied for foreach() in
{projectPath}/module/Blog/view/blog/list/index.phtml on line
{lineNumber}
```

This is due to the fact that our mapper doesn't return anything yet. Let's modify the `findAllPosts()` function to return all blog posts from the database table:

```
namespace Blog\Model;

use InvalidArgumentException;
use RuntimeException
use Zend\Db\Adapter\AdapterInterface;
```

```
use Zend\Db\Sql\Sql;

class ZendDbSqlRepository implements PostRepositoryInterface
{

    private $db;


    public function __construct(AdapterInterface $db)
    {
        $this->db = $db;
    }


    public function findAllPosts()
    {
        $sql    = new Sql($this->db);
        $select = $sql->select('posts');
        $stmt   = $sql->prepareStatementForSqlObject($select);
        $result = $stmt->execute();
        return $result;
    }


    public function findPost($id)
    {
    }
}
```

Sadly, though, a refresh of the application reveals another error message:

```
 PHP Fatal error:  Call to a member function getId() on array in
{projectPath}/module/Blog/view/blog/list/index.phtml on line
{lineNumber}
```

Let's not return the `$result` variable for now and do a dump of it to see what we get here. Change the `findAllPosts()` method and dump the result:

```
 public function findAllPosts()
{
    $sql    = new Sql($this->db);
```

```
    $select = $sql->select('posts');
    $stmt   = $sql->prepareStatementForSqlObject($select);
    $result = $stmt->execute();

    var_export($result);
    die();

    return $result;
}
```

Refreshing the application you should now see output similar to the following:

```
 Zend\Db\Adapter\Driver\Pdo\Result::__set_state(array(
    'statementMode'   => 'forward',
    'fetchMode'       => 2,
    'resource'        => PDOStatement::__set_state(array(
        'queryString' => 'SELECT "posts".* FROM "posts"',
    )),
    'options'         => null,
    'currentComplete' => false,
    'currentData'     => null,
    'position'        => -1,
    'generatedValue'  => '0',
    'rowCount'        => Closure::__set_state(array()),
))
```

As you can see, we do not get any data returned. Instead we are presented with a dump of some `Result` object that appears to have no data in it whatsoever. But this is a faulty assumption. This `Result` object only has information available for you when you actually try to access it. If you can determine that the query was successful, the best way to make use of the data within the `Result` object is to pass it to a `ResultSet` object.

First, add two more import statements to the class file:

```
 use Zend\Db\Adapter\Driver\ResultInterface;
use Zend\Db\ResultSet\ResultSet;
```

Now update the `findAllPosts()` method as follows:

```
 public function findAllPosts()
```

```
{
    $sql    = new Sql($this->db);
    $select = $sql->select('posts');
    $stmt   = $sql->prepareStatementForSqlObject($select);
    $result = $stmt->execute();

    if ($result instanceof ResultInterface && $result->isQueryResult())
{
        $resultSet = new ResultSet();
        $resultSet->initialize($result);
        var_export($resultSet);
        die();
    }

    die('no data');
}
```

Refreshing the page, you should now see the dump of a `ResultSet` instance:

```
 Zend\Db\ResultSet\ResultSet::__set_state(array(
    'allowedReturnTypes'   =>
        array(
            0 => 'arrayobject',
            1 => 'array',
        ),
    'arrayObjectPrototype' =>
        ArrayObject::__set_state(array(
        )),
    'returnType'           => 'arrayobject',
    'buffer'               => null,
    'count'                => null,
    'dataSource'           =>
        Zend\Db\Adapter\Driver\Pdo\Result::__set_state(array(
            'statementMode'   => 'forward',
            'fetchMode'       => 2,
            'resource'        =>
                PDOStatement::__set_state(array(
                    'queryString' => 'SELECT "album".* FROM "album"',
                )),
            'options'         => null,
            'currentComplete' => false,
```

```
            'currentData'     => null,
            'position'        => -1,
            'generatedValue'  => '0',
            'rowCount'        =>
                Closure::__set_state(array(
                )),
        )),
    'fieldCount'           => 3,
    'position'             => 0,
))
```

Of particular interest is the `returnType` property, which has a value of `arrayobject`. This tells us that all database entries will be returned as an `ArrayObject` instances. And this is a little problem for us, as the `PostRepositoryInterface` requires us to return an array of `Post` instances. Luckily the `Zend\Db\ResultSet` subcomponent offers a solution for us, via the `HydratingResultSet`; this result set type will populate an object of a type we specify with the data returned.

Let's modify our code. First, remove the following import statement from the class file:

```
use Zend\Db\ResultSet\ResultSet;
```

Next, we'll add the following import statements to our class file:

```
use Zend\Hydrator\Reflection as ReflectionHydrator;
use Zend\Db\ResultSet\HydratingResultSet;
```

Now, update the `findAllPosts()` method to read as follows:

```
public function findAllPosts()
{
    $sql       = new Sql($this->db);
    $select    = $sql->select('posts');
    $statement = $sql->prepareStatementForSqlObject($select);
    $result    = $statement->execute();

    if (! $result instanceof ResultInterface || !
$result->isQueryResult()) {
        return [];
    }
```

```
    $resultSet = new HydratingResultSet(
        new ReflectionHydrator(),
        new Post('', '')
    );
    $resultSet->initialize($result);
    return $resultSet;
}
```

We have changed a couple of things here. First, instead of a normal `ResultSet`, we are now using the `HydratingResultSet`. This specialized result set requires two parameters, the second one being an object to hydrate with data, and the first one being the `hydrator` that will be used (a `hydrator` is an object that will transform an array of data into an object, and vice versa). We use `Zend\Hydrator\Reflection` here, which is capable of injecting private properties of an instance. We provide an empty `Post` instance, which the hydrator will clone to create new instances with data from individual rows.

Instead of dumping the `$result` variable, we now directly return the initialized `HydratingResultSet` so we can access the data stored within. In case we get something else returned that is not an instance of a `ResultInterface`, we return an empty array.

Refreshing the page you will now see all your blog posts listed on the page. Great!

There's one little thing that we have done that's not a best-practice. We use both a hydrator and an `Post` prototype inside our `ZendDbSqlRepository`. Let's inject those instead, so that we can reuse them between our repository and command implementations, or vary them based on environment. Update your `ZendDbSqlRepository` as follows:

```
namespace Blog\Model;

use InvalidArgumentException;
use RuntimeException;

use Zend\Hydrator\HydratorInterface;
use Zend\Db\Adapter\AdapterInterface;
use Zend\Db\Adapter\Driver\ResultInterface;
use Zend\Db\ResultSet\HydratingResultSet;
use Zend\Db\Sql\Sql;

class ZendDbSqlRepository implements PostRepositoryInterface
```

```
{

    private $db;


    private $hydrator;


    private $postPrototype;

    public function __construct(
        AdapterInterface $db,
        HydratorInterface $hydrator,
        Post $postPrototype
    ) {
        $this->db            = $db;
        $this->hydrator      = $hydrator;
        $this->postPrototype = $postPrototype;
    }


    public function findAllPosts()
    {
        $sql       = new Sql($this->db);
        $select    = $sql->select('posts');
        $statement = $sql->prepareStatementForSqlObject($select);
        $result    = $statement->execute();

        if (! $result instanceof ResultInterface || !
$result->isQueryResult()) {
            return [];
        }

        $resultSet = new HydratingResultSet($this->hydrator,
$this->postPrototype);
        $resultSet->initialize($result);
        return $resultSet;
    }


    public function findPost($id)
    {
```

```
    }
}
```

Now that our repository requires more parameters, we need to update the `ZendDbSqlRepositoryFactory` and inject those parameters:

```
namespace Blog\Factory;

use Interop\Container\ContainerInterface;
use Blog\Model\Post;
use Blog\Model\ZendDbSqlRepository;
use Zend\Db\Adapter\AdapterInterface;
use Zend\Hydrator\Reflection as ReflectionHydrator;
use Zend\ServiceManager\Factory\FactoryInterface;

class ZendDbSqlRepositoryFactory implements FactoryInterface
{
    public function __invoke(ContainerInterface $container,
$requestedName, array $options = null)
    {
        return new ZendDbSqlRepository(
            $container->get(AdapterInterface::class),
            new ReflectionHydrator(),
            new Post('', '')
        );
    }
}
```

With this in place you can refresh the application again and you'll see your blog posts listed once again. Our repository no longer has hidden dependencies, and works with a database!

Finishing the repository

Before we jump into the next chapter, let's quickly finish the repository implementation by completing the `findPost()` method:

```
 public function findPost($id)
{
    $sql       = new Sql($this->db);
```

```
    $select    = $sql->select('posts');
    $select->where(['id = ?' => $id]);

    $statement = $sql->prepareStatementForSqlObject($select);
    $result    = $statement->execute();

    if (! $result instanceof ResultInterface || !
$result->isQueryResult()) {
        throw new RuntimeException(sprintf(
            'Failed retrieving blog post with identifier "%s"; unknown
database error.',
            $id
        ));
    }

    $resultSet = new HydratingResultSet($this->hydrator,
$this->postPrototype);
    $resultSet->initialize($result);
    $post = $resultSet->current();

    if (! $post) {
        throw new InvalidArgumentException(sprintf(
            'Blog post with identifier "%s" not found.',
            $id
        ));
    }

    return $post;
}
```

The `findPost()` function looks similar to the `findAllPosts()` method, with several differences.

- We need to add a condition to the query to select only the row matching the provided identifier; this is done using the `where()` method of the `Sql` object.
- We check if the `$result` is valid, using `isQueryResult()`; if not, an error occurred during the query that we report via a `RuntimeException`.
- We pull the `current()` item off the result set we create, and test to make sure we received something; if not, we had an invalid identifier, and raise an `InvalidArgumentException`.

Conclusion

Finishing this chapter, you now know how to *query* for data using the `Zend\Db\Sql` classes. You have also learned a little about the zend-hydrator component, and the integration zend-db provides with it. Furthermore, we've continued demonstrating dependency injection in all aspects of our application.

In the next chapter we'll take a closer look at the router so we'll be able to start displaying individual blog posts.
