Preparing for Different Databases

In the previous chapter, we created a `PostRepository` that returns some data from blog posts. While the implementation was adequate for learning purposes, it is quite impractical for real world applications; no one would want to modify the source files each time a new post is added! Fortunately, we an always turn to databases for the actual storage of posts; all we need to learn is how to interact with databases within our application.

There's one small catch: there are many database backend systems, including relational databases, documentent databases, key/value stores, and graph databases. You may be inclined to code directly to the solution that fits your application's immediate needs, but it is a better practice to create another layer in front of the actual database access that abstracts the database interaction. The *repository* approach we used in the previous chapter is one such approach, primarily geared towards *queries*. In this section, we'll expand on it to add *command* capabilities for creating, updating, and deleting records.

What is database abstraction?

"Database abstraction" is the act of providing a common interface for all database interactions. Consider a SQL and a NoSQL database; both have methods for CRUD (Create, Read, Update, Delete) operations. For example, to query the database against a

given row in MySQL you might use

```
$results = mysqli_query('SELECT foo FROM bar')`;
```

However, for MongoDB, for example you'd use something like:

```
$results = $mongoDbClient->app->bar->find([], ['foo' => 1, '_id' => 0])`;
```

Both engines would give you the same result, but the execution is different.

So if we start using a SQL database and write those codes directly into our `PostRepository` and a year later we decide to switch to a NoSQL database, the existing implementation is useless to us. And in a few years later, when a new persistence engine pops up, we have to start over yet again.

If we hadn't created an interface first, we'd also likely need to change our consuming code!

On top of that, we may find that we want to use some sort of distributed caching layer for *read* operations (fetching items), while *write* operations will be written to a relational database. Most likely, we don't want our controllers to need to worry about those implementation details, but we will want to ensure that we account for this in our architecture.

At the code level, the interface is our abstraction layer for dealing with differences in implementations. However, currently, we only deal with queries. Let's expand on that.

Adding command abstraction

Let's first think a bit about what possible database interactions we can think of. We need to be able to:

- find a single blog post
- find all blog posts
- insert new blog post
- update existing blog posts
- delete existing blog posts

At this time, our `PostRepositoryInterface` deals with the first two. Considering this is the layer that is most likely to use different backend implementations, we probably want to keep it separate from the operations that cause changes.

Let's create a new interface, `Blog\Model\PostCommandInterface`, in `module/Blog/src/Model/PostCommandInterface.php`, and have it read as follows:

```
namespace Blog\Model;

interface PostCommandInterface
{
    /**
     * Persist a new post in the system.
     *
     * @param Post $post The post to insert; may or may not have an identifier.
     * @return Post The inserted post, with identifier.
     */
    public function insertPost(Post $post);

    /**
     * Update an existing post in the system.
     *
     * @param Post $post The post to update; must have an identifier.
     * @return Post The updated post.
     */
    public function updatePost(Post $post);

    /**
     * Delete a post from the system.
     *
     * @param Post $post The post to delete.
     * @return bool
     */
    public function deletePost(Post $post);
}
```

This new interface defines methods for each *command* within our model. Each expects a `Post` instance, and it is up to the implementation to determine how to use that instance to issue the command. In the case of an insert operation, our `Post` does not require an identifier (which is why the value is nullable in the constructor), but will return a new instance that is guaranteed to have one. Similarly, the update operation will return the updated post (which may be the same instance!), and a delete operation will indicate if the operation was successful.

Conclusion

We're not quite ready to use the new interface; we're using it to set the stage for the next few chapters, where we look at using zend-db to implement our persistence, and later creating new controllers to handle blog post manipulation.

Copyright (c) 2016 Zend Technologies USA Inc. (http://www.zend.com/)

Learn more about Zend Framework (http://framework.zend.com)