John Squibb

Home | Articles | Tutorials | Projects | Tools | Travel | Music

Build a PHP MVC Framework in One Hour - Part Two: Building Libraries and Drivers

If you haven't already completed Part One of this tutorial, I highly recommend doing so before continuing with this section.

Creating Libraries

Let's kick part two of the MVC tutorial off by creating exploring a new concept, Libraries. When writing code we often repeat basic tasks without even thinking about it. Whether our script requires us to connect to a database, or perform a file upload, or connect to a resource on a remote server, or another of many potential tasks, we often write the same bits of code over and over. This is counterproductive and contrary to the purpose of our framework, right? Our first inclination may be to simply write a model to handle database connections of file uploads, which would be a completely valid solution and would certainly work. But how might we organize our framework so that models that act as utilities can be distinguished from models that perform the workload of our various controllers? Perhaps a separate naming convention, or subfolders within the models directory? Better yet, we could move the utility classes of our framework out of models entirely, and into a new folder, libraries.

Let's go ahead and create a new folder, *libraries* in our application root. As developers of dynamic web applications, one of our common tasks includes connecting to and querying a database. This may be a MySQL, Postgresql, or another database engine. You may be using any of several different APIs to interact with the database of your choice, and I am always one to encourage trying different flavors until you find the one that suits your application best. One thing we know for certain, however, is that there is a bare minimum of common functionality that we must implement regardless of which engine we choose:

- connect to the database.
- prepare a query.
- execute our query.
- fetch the results of our query.
- disconnect from the database.

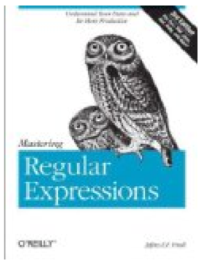
There can be a lot more going on in a typical script, such as caching of results and queries, locking of tables, transactions for engines that support them, and more, but this gives us a starting point. Now, let's pick an engine. For the purposes of this tutorial, I will be using the MySQL Improved functionality built into PHP, and enabled through the proper compilation flags. I will assume that

WINNER 2004 TONY® TRIPLE CROWN
BEST MUSICAL. BEST SCORE. BEST BOOK.
Avenue Q
The Broadway Musical
"SWEETLY SATIRIC and SURPRISINGLY FRESH!"
THE NEW YORK TIMES
"JAW DROPPINGLY HILARIOUS!"
ENTERTAINMENT WEEKLY
BUY TICKETS

Suggested Reading

Mastering Regular Expressions

by Jeffrey E.F. Friedl



This is a must-read for anyone interested in *learning* the complexities of regular expressions usage. Make no mistake, this book is not a quick reference, but rather an all-in tome of regular expression goodness. Even if you only use the occasional regex to validate emails, phone numbers, credit cards, or other, this book will give you an excellent understanding of how regular expressions perform such tasks.

you know it, or choose to use the engine of your liking.

Go into the libraries folder you created and create a new file called database.php. In database.php we will create an abstract class with the following abstract methods (more on that here if you are unfamiliar with class abstraction):

Aprende Lo Que Sea por $14 dólares
Más de 17,000 Cursos a $14 dólares c/u en nuestra venta de Black Friday
Encuentra un Curso
udemy

```php
<?php
/**
 * The Database Library handles database interaction for the application
 */
abstract class Database_Library
{
    abstract protected function connect();
    abstract protected function disconnect();
    abstract protected function prepare();
    abstract protected function query();
    abstract protected function fetch();
}
```

Toggle plain-text

Now we've got a very generic database library 'Database'. In the next paragraph, we will discuss a new concept, known as *drivers*, which we will utilize to extend our database functionality.

Extending Libraries with Drivers

Rather than define our methods directly in this class, we can use this as a base template for more specific classes. Just like your printer needs a driver or your car needs a driver to make it go, your database library will require a MySQL or some other driver to make it do what it needs to.

Let's go ahead and create a *drivers* folder inside of the libraries folder. Inside the drivers folder, create a new file called mysqlimproved.php. Since our base database class is abstract, it will act as a requirements list for our individual drivers, though it may include more base functionality down the road as well. Inside mysqlimproved.php, add the following:

```php
<?php
/**
 * The MySQL Improved driver extends the Database_Library to provide
 * interaction with a MySQL database
 */
class MysqlImproved_Driver extends Database_Library
{
    /**
     * Connection holds MySQLi resource
     */
    private $connection;

    /**
     * Query to perform
     */
    private $query;

    /**
     * Result holds data retrieved from server
     */
    private $result;

    /**
```

```
     * Create new connection to database
     */
    public function connect()
    {
        //connection parameters
        $host = 'localhost';
        $user = 'username';
        $password = 'password';
        $database = 'my_test';

        //your implementation may require these...
        $port = NULL;
        $socket = NULL;

        //create new mysqli connection
        $this->connection = new mysqli
        (
            $host , $user , $password , $database , $port , $socket
        );
    }

    public function disconnect(){}

    public function prepare(){}

    public function query(){}

    public function fetch(){}
}
```

Toggle plain-text

This creates the basic shell for our MySQL Improved driver. Before continuing on, let's go back to our base library to more narrowly define our parameters for our methods. In order to prepare a query, we would have to provide the query to prepare. Also, when fetching a result we may wish to have data returned in the form of an object or an array. Modify database.php to look like the this:

```
<?php
/**
 * The Database Library handles database interaction for the application
 */
abstract class Database_Library
{
    abstract protected function connect();
    abstract protected function disconnect();
    abstract protected function prepare($query);
    abstract protected function query();
    abstract protected function fetch($type = 'object');
}
```

Toggle plain-text

in our driver, we can now fill in the gaps a bit more:

```
<?php
/**
 * The MySQL Improved driver extends the Database_Library to provide
 * interaction with a MySQL database
 */
class MysqlImproved_Driver extends Database_Library
{
    /**
     * Connection holds MySQLi resource
     */
    private $connection;

    /**
     * Query to perform
     */
    private $query;

    /**
     * Result holds data retrieved from server
     */
    private $result;

    /**
     * Create new connection to database
     */
    public function connect()
    {
        //connection parameters
        $host = 'localhost';
        $user = 'username';
        $password = 'password';
        $database = 'my_test';

        //your implementation may require these...
        $port = NULL;
        $socket = NULL;

        //create new mysqli connection
        $this->connection = new mysqli
        (
            $host , $user , $password , $database , $port , $socket
        );

        return TRUE;
    }

    /**
     * Break connection to database
     */
    public function disconnect()
    {
        //clean up connection!
        $this->connection->close();

        return TRUE;
    }

    /**
     * Prepare query to execute
     *
     * @param $query
     */
    public function prepare($query)
    {
        //store query in query variable
        $this->query = $query;

        return TRUE;
    }

    /**
     * Execute a prepared query
     */
    public function query()
    {
        if (isset($this->query))
        {
            //execute prepared query and store in result variable
            $this->result = $this->connection->query($this->query);

            return TRUE;
        }

        return FALSE;
    }

    /**
     * Fetch a row from the query result
     *
     * @param $type
     */
    public function fetch($type = 'object')
    {
        if (isset($this->result))
        {
            switch ($type)
            {
                case 'array':

                    //fetch a row as array
                    $row = $this->result->fetch_array();

                break;

                case 'object':

                //fall through...

                default:

                    //fetch a row as object
                    $row = $this->result->fetch_object();

                break;
            }

            return $row;
        }

        return FALSE;
    }
}
```

Toggle plain-text

Modifying the Router

Now in order to try out our new library and driver setup, we have to first make some changes to the way files are served in our framework. Open up the router.php file located in the controllers folder that we created in the first part of this tutorial. If we look at our __autoload function we'll see the code we wrote to handle the 'lazy loading' of our models. Since we used the same naming convention for our libraries and drivers, a quick modification to this code will allow us to load those as easily.

Modify the router __autoload method to look like this:

```
function __autoload($className)
{
    // Parse out filename where class should be located
    // This supports names like 'Example_Model' as well as 'Example_Two_Model'
    list($suffix, $filename) = preg_split('/_/', strrev($className), 2);
    $filename = strrev($filename);
    $suffix = strrev($suffix);

    //select the folder where class should be located based on suffix
    switch (strtolower($suffix))
    {
        case 'model':

            $folder = '/models/';

        break;

        case 'library':

            $folder = '/libraries/';

        break;

        case 'driver':

            $folder = '/libraries/drivers/';

        break;
    }

    //compose file name
    $file = SERVER_ROOT . $folder . strtolower($filename) . '.php';

    //fetch file
    if (file_exists($file))
    {
        //get file
        include_once($file);
    }
    else
    {
        //file does not exist!
        die("File '$filename' containing class '$className' not found in
'$folder'.");
    }
}
```

Toggle plain-text

Notice how we added the switch statement to check the suffix of our class names for the appropriate folder to search for classes in. This allows the router to serve files of all different types so long as they are organized according to our evolving naming convention.

Adding a Database

We are almost ready to use our database library, however doing so would be rather difficult since we don't actually have any data to query against! If you already have a schema set up with data you would like to practice with, please feel free to skip this seciton and do so. If you lack data, use the following mysqldump file to create some data in your local database. Please ensure that you do not have a conflicting schema in your database before running this script:

```
DROP TABLE IF EXISTS `articles`;
SET @saved_cs_client     = @@character_set_client;
SET character_set_client = utf8;
CREATE TABLE `articles` (
  `id` int(11) NOT NULL auto_increment,
  `date` varchar(25) NOT NULL,
  `title` varchar(50) NOT NULL,
  `content` text NOT NULL,
  `author` varchar(100) NOT NULL,
  PRIMARY KEY  (`id`)
) ENGINE=MyISAM AUTO_INCREMENT=3 DEFAULT CHARSET=latin1;
SET character_set_client = @saved_cs_client;

LOCK TABLES `articles` WRITE;
/*!40000 ALTER TABLE `articles` DISABLE KEYS */;
INSERT INTO `articles` VALUES (1,'Dec 12, 2008','How to generate Lorem
Ipsum','Nam accumsan enim tristique urna commodo mollis. Etiam eget leo est.
Donec tincidunt quam nec nulla pulvinar sed tristique lorem tincidunt.
Pellentesque nibh lectus; suscipit sed ullamcorper sed, laoreet ut tortor. Morbi
ut ante tellus. Integer vitae felis id justo tempor adipiscing. Curabitur eget
ipsum et urna ultricies pulvinar. Fusce enim dolor, interdum eu egestas vel,
iaculis eget nisl. Aenean pretium diam accumsan quam tincidunt sit amet dictum
lorem scelerisque. In gravida ultricies aliquet. Phasellus porta erat vel augue
sodales feugiat! Pellentesque mattis malesuada ultrices. Mauris eleifend mi quis
arcu tincidunt vehicula! Nam sodales commodo lacus, et commodo metus venenatis
vel. Sed mollis molestie congue. Nulla ante leo, aliquet et convallis sed;
consequat sed turpis. Duis augue leo, adipiscing at venenatis eget, eleifend
vitae velit!
','John Squibb'),(2,'Jan 03, 1988','Using __autoload','Now in order to try out
our new library and driver setup, we have to first make some changes to the way
files
are served in our framework. Open up the router.php file located in the
controllers folder that we created in the first part of this tutorial.
If we look at our __autoload function we\'ll see the code we wrote to handle the
\'lazy loading\' of our models. Since we used the same naming convention
for our libraries and drivers, a quick modification to this code will allow us
to load those as easily.','Frank Rabbit');
```

Toggle plain-text

Modifying the News Model

Now we should have some data for testing our library out! In the News_Model class we created in the first part of this tutorial, let's remove the hard-coded articles and instead use our database library to pull the dynamic article content from the database by author name. Modify /models/news.php to read:

```
<?php
/**
 * The News Model does the back-end heavy lifting for the News Controller
 */
class News_Model
{
    /**
     * Holds instance of database connection
     */
    private $db;

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

    /**
     * Fetches article based on supplied name
     *
     * @param string $author
     *
     * @return array $article
     */
    public function get_article($author)
    {
        //connect to database
        $this->db->connect();

        //can you detect the gaping security flaw here?

        //prepare query
        $this->db->prepare
        (
            "
            SELECT
                `date`,
                `title`,
                `content`,
                `author`
            FROM
                `articles`
            WHERE
                `author` = '$author'
            LIMIT
                1
            ;
            "
        );

        //execute query
        $this->db->query();

        $article = $this->db->fetch('array');

        return $article;
    }

}
```

Toggle plain-text

In order to pull articles by author name, we are going to have to adjust our router.php file again to allow URL-encoded characters such as space. In a nutshell, if we want to pull the article written by 'John Squibb', when we provide this string in the URL, it will be encoded as 'John%20Squibb'.

Modify the router.php file by changing:

```
        $getVars[$variable] = $value;
```

to:

```
        $getVars[$variable] = urldecode($value);
```

Filtering Input

While we are on the topic of filtering input, it is important to note that since we are accepting data from the user to search our database with, it is essential that we clean it. Any data that is provided to your application must be filtered, cleaned, groomed, validated, or what have you. Never trust data from your users! A simple function is available to clean data before passing it to MySQL. We will create a method to add to the mysqlimproved.php file:

```
/**
 * Sanitize data to be used in a query
 *
 * @param $data
 */
public function escape($data)
{
    return $this->connection->real_escape_string($data);
}
```

Toggle plain-text

Let's modify our get_article function in /models/news.php to make our code a bit safer:

```
/**
 * Fetches article based on supplied name
 *
 * @param string $author
 *
 * @return array $article
 */
public function get_article($author)
{
    //connect to database
    $this->db->connect();

    //sanitize data
    $author = $this->db->escape($author);

    //prepare query
    $this->db->prepare
    (
        "
        SELECT
            `date`,
            `title`,
            `content`,
            `author`
        FROM
            `articles`
        WHERE
            `author` = '$author'
        LIMIT
            1
        ;
        "
    );

    //execute query
    $this->db->query();

    $article = $this->db->fetch('array');
    return $article;
}
```

Toggle plain-text

Note, the quick addition of the `escape()` method in our mysql class should not be the final say in cleansing data. You should strive to validate all incoming data rather than just make it query-safe before throwing it at mysql. Something like preg_replace() would do nicely here.

Modifying the News Controller

We'll need to make a quick change to the News Controller, since we were previously selecting articles in a different way. Change the following line:

```
$article = $newsModel->get_article($getVars['article']);
```

to:

```
$article = $newsModel->get_article($getVars['author']);
```

That should bring the News_Controller file up to speed on the changes we have made.

Viewing Changes

Save your changes, and call up http://yoursite.com/index.php?news&author=John Squibb
Voila! You now have a working database library and a modified article system.

Download Source

You can download the source files from the following links. They contain the full framework from part 1 of this tutorial, with the modifications made in this section of the tutorial.

- .zip (click to download...)
- .tar.gz (click to download...)

What Next?

Part three of this tutorial wraps up with a couple of suggestions for rounding out what we have put together so far.

If you enjoyed this tutorial, check out the Articles and Tutorials sections for more great stuff. Thanks for reading!

Tags: PHP,MVC,framework,one hour,part two
Short URL: http://sqb.in/a2tm