From Technologies to Solutions

PHP Programming with PEAR

XML, Data, Dates, Web Services, and Web APIs

Maximize your PHP development productivity by mastering the PEAR packages for accessing and displaying data, handling dates, working with XML and Web Services, and accessing Web APIs

Stephan Schmidt
Carsten Lucke
Stoyan Stefanov
Aaron Wormus

PACKT PUBLISHING

PHP Programming with PEAR

XML, Data, Dates, Web Services, and Web APIs

Maximize your PHP development productivity by mastering the PEAR packages for accessing and displaying data, handling dates, working with XML and Web Services, and accessing Web APIs

Stephan Schmidt

Carsten Lucke

Stoyan Stefanov

Aaron Wormus

BIRMINGHAM - MUMBAI

PHP Programming with PEAR

XML, Data, Dates, Web Services, and Web APIs

First published: September 2006

Production Reference: 1160906

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 1-904811-79-5

www.packtpub.com

Cover Image by www.visionwt.com

Credits

Authors

Stephan Schmidt

Carsten Lucke

Stoyan Stefanov

Aaron Wormus

Reviewers

Lukas Smith

Shu-Wai Chow

Arnaud Limbourg

Development Editor

Douglas Paterson

Assistant Development Editor

Nikhil Bangera

Technical Editor

Ashutosh Pande

Editorial Manager

Dipali Chittar

Indexer

Mithil Kulkarni

Proofreader

Chris Smith

Layouts and Illustrations

Shantanu Zagade

Cover Designer

Shantanu Zagade

About the Authors

Stephan Schmidt is working for 1&1 Internet, the world's largest web hosting provider in Karlsruhe. He is leading a team of PHP and Java programmers and focusses on the development of the websites and online ordering systems of 1&1. He has been an active contributor to the PHP open source scene since 2001, when he founded the PHP Application Tools website (`http://www.php-tools.net`) together with some friends, which today is one of the oldest PHP OSS projects. He has also been working on more than 15 PEAR packages (with a focus on XML and web services), as well as the id3 extension. Recently he started the XJConf project (`http://www.xjconf.net`) and also contributes to the Java community.

He is the author of the (German language) *PHP Design Patterns* (O'Reilly Verlag, ISBN 3-89721-442-3) as well as a co-author of several other books on PHP and has been writing articles for several magazines. He has also spoken at various open-source conferences around the globe.

He devotes his spare time to American super-hero comics and the golden 50s.

Carsten Lucke studied computer science at the University of Applied Sciences in Brandenburg, Germany. He is currently working as a software engineer for the software design and management AG (sd&m AG) in Munich, Germany.

In his spare time he writes articles for various magazines and contributes to the open-source community (especially PHP). He is the developer of a handful of PEAR/PECL packages, founder of the 3rdPEARty pear channel-server project (`3rdpearty.net`) and the `tool-garage.de` open-source and freeware project.

Stoyan Stefanov is a web developer from Montreal, Canada, Zend Certified Engineer, book author, and contributor to the international PHP community. His personal blog is at `http://www.phpied.com`.

I would like to thank Tom Kouri and the team at High-Touch Communications in Montreal; special thanks to Derek Fong for introducing me to PEAR and to Michael Caplan for always being up to speed with the latest PEAR development.

Aaron Wormus is a freelance consultant working out of Frankfurt Germany. With a background in client/server development and intranet infrastructure, Aaron uses the power of PHP and Open Source tools to implement customized back-end solutions for his clients.

As a writer, Aaron contributes regular articles for *PHPMagazine*, *PHPArchitect* and *PHPSolutions* magazines. The topics of his articles have included PEAR Packages, core PHP programming, and programming methodologies. Aaron is also an avid blogger, and keeps his personal blog flowing with technical posts, political rants, and regular updates on the state of the weird and wonderful thing that is the Internet.

When Aaron is not at his computer, you can probably find him chasing his two daughters around, or wandering around the floor of a technology conference on a caffeine-induced high.

About the Reviewers

Lukas Kahwe Smith has been developing PHP since 2000 and joined the PEAR repository in 2001. Since then he has developed and maintained several PEAR packages, most notably MDB2 and LiveUser and has influenced the organization of the project itself as a founding member of the PEAR Group steering committee and QA core team. Aside from several magazine publications he is a well known speaker at various international PHP conferences.

Shu-Wai Chow has worked in the field of computer programming and information technology for the past eight years. He started his career in Sacramento, California, spending four years as the webmaster for Educaid, a First Union company and another four years at Vision Service Plan as an application developer. Through the years, he has become proficient in Java, JSP, PHP, ColdFusion, ASP, LDAP, XSLT, and XSL-FO. Shu has also been the volunteer webmaster and a feline adoption counselor for several animal welfare organizations in Sacramento.

He is currently a software engineer at Antenna Software in Jersey City, New Jersey.

Born in the British Crown Colony of Hong Kong, Shu did most of his alleged growing up in Palo Alto, California. He studied Anthropology and Economics at California State University, Sacramento. He lives along the New Jersey coast with seven very demanding cats, three birds that are too smart for their own good, a cherished Fender Stratocaster, and a beloved, saint-like girlfriend.

Arnaud Limbourg has been developing in PHP for 4 years. He is involved in the PEAR project as an assurance quality member and co-maintainer of the LiveUser package. He currently works for a telecom company doing VoIP as a developer.

Table of Contents

Preface

PEAR is the PHP Extension and Application Repository, and is a framework and distribution system for reusable, high-quality PHP components, available in the form of "packages". The home of PEAR is `pear.php.net`, from where you can download and browse this extensive range of powerful packages. For most things that you would want to use in your day-to-day development work, you will likely find a PEAR class or package that meets your needs. In addition to the functionality offered by the packages, PEAR code follows strict coding guidelines, bringing a consistency to your PEAR development experience.

In this book, you will learn how to use a number of the most powerful PEAR packages to boost your PHP development productivity. By focusing on the packages for key development activities, this book gives you an in-depth guide to getting the most from these powerful coding resources.

What This Book Covers

Chapter 1 provides an introduction to the MDB2 database abstraction layer. You will see how to connect to the database, instantiate MDB2 objects, execute queries and fetch data. There are a number of features and SQL syntax that are implemented differently in the database systems that MDB2 supports. MDB2 does its best to wrap the differences and provide a single interface for accessing those features, so that the developer doesn't need to worry about the implementation in the underlying database system. You will see how to use this SQL abstraction feature to provide auto-increment fields, perform "replace" queries that will update the records that already exist or do an insert otherwise, and make use of prepared statements, a convenient and security-conscious method of writing to the database. You will also learn about MDB2 modules and how to extend MDB2 to provide custom fetch and result classes, iterators, and modules.

Now that you've got data from your database, you want to display it.

Chapter 2 covers a range of PEAR packages commonly used for presenting data in different formats. You will see how to use HTML_Table and HTML_Table_Matrix to create and format tables, generate and format an Excel spreadsheet with the Excel_Spreadsheet_Writer package, create a flexible, pageable "datagrid" with Structures_Datagrid, and generate PDF documents on the fly with File_PDF.

XML is another favorite format for working with data, and PEAR does not let you down with its XML support.

In *Chapter 3* we take an in-depth look at working with XML in PEAR. The chapter covers creating XML documents using the XML_Util, XML_FastCreate, XML_Serializer, and XML_XUL packages. The chapter also covers reading XML documents using a SAX-based parser and transforming PHP objects into XML (and back again!) with XML_Serializer and XML_Unserialize.

Chapter 4 introduces you to PEAR's support for web services and Web APIs. You will learn about consuming SOAP and XML-RPC web services, access the Google API, search blog entries with Services_Technorati, access the Amazon web service, access the Yahoo API, and learn how to offer web services, either XML-RPC or SOAP based. You will also get a taste of offering a REST-based service with XML_Serializer.

Chapter 5 covers PEAR's date and time functions using PEAR::Calendar and PEAR::Date. You will learn about the benefits these packages offer over the standard PHP date and time functions, and then see how to create, manipulate, and compare Date objects, work with Date_Span arithmetic, handle timezones, keep track of public holidays with Date_Holiday, and use the Calendar class to display an HTML calendar.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

There are three styles for code. Code words in text are shown as follows: "This class also provides a `setId()` method, which is called by the `Label` object when the artist is added to the list of signed artists."

A block of code will be set as follows:

```
function getDGInstance($type)
{
    if (class_exists($type))
```

```
        {
            $datagrid =& new $type;
            return $datagrid;
        } else
          {
            return false;
          }
    }
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be made bold:

```
$driver        = Date_Holidays::factory($driverId, $year);
$internalNames = $driver->getInternalHolidayNames();
```

Any command-line input and output is written as follows:

```
$ pear-dh-compile-translationfile --help
```

New terms and **important words** are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "clicking the **Next** button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader Feedback

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer Support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the Example Code for the Book

Visit `http://www.packtpub.com/support`, and select this book from the list of titles to download any example code or extra resources for this book. The files available for download will then be displayed.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing this you can save other readers from frustration, and help to improve subsequent versions of this book. If you find any errata, report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **Submit Errata** link, and entering the details of your errata. Once your errata have been verified, your submission will be accepted and the errata added to the list of existing errata. The existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with some aspect of the book, and we will do our best to address it.

1
MDB2

The Web has matured and grown over the last decade and with it the need for more complex and dynamic sites. While storing information in a text file or simple database may have been suitable in the past, these days any serious application developer requires a firm knowledge of how to wield the relational database.

From the earliest versions of PHP, programmers have always been able to count on strong database support. However until the recent release of PDO there had been no standard way of interfacing with the multiple database drivers bundled with PHP. The lack of unified API has spawned several efforts to create database abstraction layers (DBAL). The primary goal of these efforts is to enable developers to write code that is not specific to the database back end being used, thereby enabling clients/users to deploy the application on whichever database platform they prefer.

The three most prominent full-featured database abstraction layers over the years have been AdoDB, PEAR::DB, and Metabase. In the last few years we have seen another very strong contender in the arena of database abstraction layers, and that is PEAR::MDB. This chapter is about MDB's second iteration—MDB2.

A Brief History of MDB2

It all started when Lukas Smith, a PEAR developer, submitted a few patches to the existing DBAL, Metabase. At some point he and the Metabase author started discussing bringing Metabase into PEAR as a new package. The goal of the new package was to merge the functionality of Metabase with the API of the existing and popular PEAR::DB into a feature-rich and well-performing database abstraction library, leveraging the PEAR infrastructure. Thus began the life of MDB2's predecessor PEAR::MDB.

After a few years of work on PEAR::MDB, it became apparent that the decision to keep a similar API to that of Metabase and PEAR::DB created some design issues, which hampered the growth of MDB into a full-featured DBAL. Since PEAR::MDB

had reached a stable state in PEAR, it was not possible to fix these API issues without breaking backwards compatibility, which was not an option. The solution was to take the lessons learned during the development of Metabase and MDB and apply them to a new package that would contain a well-designed and modern API. The new package became MDB2.

Abstraction Layers

Before we get into the details of how MDB2 handles database abstraction, we should take a look at database abstraction theory and find out exactly what it means. There are several different facets to database abstraction, and we will go over them and specify what their requirements are.

Database Interface Abstraction

Database interface abstraction is the most important of all; it allows a programmer to access every database using the same method calls. This means that instantiating a database connection, sending a query, and retrieving the data will be identical, regardless of which database you are interfacing with.

SQL Abstraction

Most modern databases support a standard subset of SQL, so most SQL that you write will work regardless of which database back end you are using. However, many databases have introduced database-specific SQL lingo and functions, so it is possible that the SQL that you write for one database will not work on another. As an RDBMS (Relational DataBase Management System) matures, sometimes it implements features that are not compatible with older versions of the same database. So if an application developer wants to write SQL compliant with all versions of a specific database (or which can be used on multiple database back ends), one option is to stick to SQL they know is supported on all platforms. The better option though, is to use an abstraction layer that emulates the functionality when it's not available on the specific platform.

While there is no possible way to encapsulate every possible SQL function, MDB2 provides support for many of the most common features of SQL. These features include support for LIMIT queries, sub-selects, and prepared queries among others. Using the MDB2 SQL abstraction will guarantee that you'll be able to use this advanced functionality, even though it's not natively supported in the database you're using. Further in this chapter you'll learn more about the different SQL abstraction functions that MDB2 provides.

Datatype Abstraction

The third type of abstraction is the datatype abstraction. The need for this type of abstraction stems from the fact that different databases handle data types differently.

Speed Considerations

Now that you are salivating over all these great features that are bundled in MDB2, you should think about speed and performance issues. When using a database abstraction layer you need to understand that in many cases you will need to sacrifice performance speed for the wealth of functionality that the package offers. This is not specific to MDB2 or even database abstraction layers, but to abstraction layers or software virtualization systems in general.

Thankfully, unlike VMWare or Microsoft Virtual PC, which abstract each system call made, MDB2 only provides abstraction when a feature is not available in a specific back end. This means that performance will depend on the platform on which you are using MDB2. If you are very concerned about performance, you should run an opcode cache, or turn on a database-specific query caching mechanism in your particular database. Taking these steps in PHP itself or your database back end will make the overhead, which is inevitable in your database abstraction layer, much smaller.

MDB2 Package Design

The API design of MDB2 was created to ensure maximum flexibility. A modular approach was taken when handling both database back ends and specific advanced functionality. Each database -specific **driver** is packaged and maintained as an independent PEAR module. These driver packages have a life of their own, which means individual release cycles and stability levels. This system allows the maintainers of the database drivers to release their packages as often as they need to, without having to wait for a release of the main MDB2 package. This also allows the MDB2 package to advance in stability regardless of the state of the driver packages, the effect being that while the state of MDB2 is stable, some of its drivers may only be beta. Also, when a new database driver is released, it is tagged as alpha and the release process progresses according to PEAR standards.

The second type of modularity built into MDB2 is used for adding extended functionality to MDB2. Rather than include the functions into MDB2 itself or extend MDB2 with a new class that adds this functionality, you have the option to create a separate class and then load it into MDB2 using the `loadModule()` method. Once a module is loaded into MDB2, you will be able to access your methods as if they were built into MDB2. MDB2 uses this internally to keep the core components as fast

as possible, and also makes it possible for the user to define and include their own classes into MDB2. You'll see the details of how to extend MDB2 later in this chapter.

Getting Started with MDB2

Let's discuss the necessary steps to install MDB2, to create an MDB2 object, and then set up some options to set the data fetch mode and finally disconnect from the database.

Installing MDB2

When installing MDB2, keep in mind that the MDB2 package does not include any database drivers, so these will need to be installed separately. MDB2 is stable, but as explained earlier, since the packages have different release cycles, the status of the package you plan to use may be beta, alpha, or still in development. This will need to be taken into consideration when installing a driver package.

The easiest way to install MDB2 is by using the PEAR installer:

```
> pear install MDB2
```

This command will install the core MDB2 classes, but none of the database drivers. To install the driver for the database you'll be using, type:

```
> pear install MDB2_Driver_mysql
```

This will install the driver for MySQL. If you wish to install the driver for SQLite, type:

```
> pear install MDB2_Driver_sqlite
```

The full list of currently available drivers is as follows:

- `fbsql`: FrontBase
- `ibase`: InterBase
- `mssql`: MS SQL Server
- `mysql`: MySQL
- `mysqli`: MySQL using the mysqli PHP extension; for more details, visit `http://php.net/mysqli`
- `oci8`: Oracle
- `pgsql`: PostgreSQL
- `querysim`: Querysim
- `sqlite`: SQLite

Connecting to the Database

To connect to your database after a successful installation, you need to set up the DSN (Data Source Name) first. The DSN can be a string or an array and it defines the parameters for your connection, such as the name of the database, the type of the RDBMS, the username and password to access the database, and so on.

DSN Array

If the DSN is defined as an array, it will look something like this:

```
$dsn = array ( 'phptype'  => 'mysql',
               'hostspec' => 'localhost:3306',
               'username' => 'user',
               'password' => 'pass',
               'database' => 'mdb2test'
             );
```

Here's a list of keys available to use in the DSN array:

- `phptype`: The name of the driver to be used, in other words, it defines the type of the RDBMS
- `hostspec`: (host specification) can look like `hostname:port` or it can be only the hostname while the port can be defined separately in a `port` array key
- `database`: The name of the actual database to connect to
- `dbsyntax`: If different than the `phptype`
- `protocol`: The protocol, for example TCP
- `socket`: Mentioned if connecting via a socket
- `mode`: Used for defining the mode when opening the database file

DSN String

A quicker and friendlier way (once you get used to it) to define the DSN is to use a string that looks similar to a URL. The basic syntax is:

```
phptype://username:password@hostspec/database
```

The example above becomes:

```
$dsn = 'mysql://user:pass@localhost:3306/mdb2test';
```

More details on the DSN and more DSN string examples are available in the PEAR manual at `http://pear.php.net/manual/en/package.database.mdb2.intro-dsn.php`.

Instantiating an MDB2 object

There are three methods to create an MDB2 object:

```
$mdb2 =& MDB2::connect($dsn);
$mdb2 =& MDB2::factory($dsn);
$mdb2 =& MDB2::singleton($dsn);
```

`connect()` will create an object and will connect to the database. `factory()` will create an object, but will not establish a connection until it's needed. `singleton()` is like `factory()` but it makes sure that only one MDB2 object exists with the same DSN. If the requested object exists, it's returned; otherwise a new one is created.

One scenario exists where you can "break" the singleton functionality by using `setDatabase()` to set the current database to a database different from the one specified in the DSN.

```
$dsn = 'mysql://root@localhost/mdb2test';
$mdb2_first =& MDB2::singleton($dsn);
$mdb2_first->setDatabase('another_db');
$mdb2_second =& MDB2::singleton($dsn);
```

In this case you'll have two different MDB2 instances.

All three methods will create an object of the database driver class. For example, when using the MySQL driver, the variable `$mdb2` defined above will be an instance of the `MDB2_Driver_mysql` class.

Options

MDB2 accepts quite a few options that can be set with the call to `connect()`, `factory()`, or `singleton()`, or they can be set later using the `setOption()` method (to set one option a time) or the `setOptions()` method (to set several options at once). For example:

```
$options = array ( 'persistent' => true,
                   'ssl' => true,
                 );
$mdb2 =& $MDB2::factory($dsn, $options);
```

or

```
$mdb2->setOption('portability', MDB2_PORTABILITY_NONE);
```

The full list of available options can be found in the package's API docs at: `http://pear.php.net/package/MDB2/docs/`. Let's take a look at two important ones right away.

Option "persistent"

This Boolean option defines whether or not a persistent connection should be established.

There is an informative article on `mysql.com` about the pros and cons of using persistent connections in MySQL; it's located at `http://www.mysql.com/news-and-events/newsletter/2002-11/a0000000086.html`.

The default value is `false`. If you want to override the default, you can set it when the object is created:

```
$options = array ( 'persistent' => true
                 );
$mdb2 =& MDB2::factory($dsn, $options);
```

Using `setOption()` you can define options after the object has been created:

```
$mdb2->setOption('persistent', true);
```

Option "portability"

MDB2 tries to address some inconsistencies in the way different DBMS implement certain features. You can define to which extent the database layer should worry about the portability of your scripts by setting the `portability` option.

The different portability options are defined as constants prefixed with `MDB2_PORTABILITY_*` and the default value is `MDB2_PORTABILITY_ALL`, meaning "do everything possible to ensure portability". The full list of portability constants and their meaning can be found at `http://pear.php.net/manual/en/package.database.mdb2.intro-portability.php`.

You can include several portability options or include all with some exceptions by using bitwise operations, exactly as you would do when setting error reporting in PHP. The following example will set the portability to all but lowercasing:

```
MDB2_PORTABILITY_ALL ^ MDB2_PORTABILITY_LOWERCASE
```

If you don't want use the full portability features of MDB2 but only trim white space in results and convert empty values to `null` strings:

```
MDB2_PORTABILITY_RTRIM | MDB2_PORTABILITY_EMPTY_TO_NULL
```

Probably the best thing to do is to leave the default `MDB2_PORTABILITY_ALL`; this way if you run into some problems with your application, you can double-check the database access part to ensure that the application is as portable as possible.

Setting Fetch Mode

One more setting you'd probably want to define upfront is the fetch mode, or the way results will be returned to you. You can have them as an enumerated list (default option), associative arrays, or objects. Here are examples of setting the fetch mode:

```
$mdb2->setFetchMode(MDB2_FETCHMODE_ORDERED);
$mdb2->setFetchMode(MDB2_FETCHMODE_ASSOC);
$mdb2->setFetchMode(MDB2_FETCHMODE_OBJECT);
```

Probably the friendliest and the most common fetch mode is the associative array, because it gives you the results as arrays where the keys are the names of the table columns. To illustrate the differences, consider the different ways of accessing the data in your result sets:

```
echo $result[0]; // ordered/enumerated array, default in MDB2
echo $result['name']; // associative array
echo $result->name; // object
```

There is one more fetch mode type, which is `MDB2_FETCHMODE_FLIPPED`. It's a bit exotic and its behavior is explained in the MDB2 API documentation as:

"For multi-dimensional results, normally the first level of arrays is the row number, and the second level indexed by column number or name. `MDB2_FETCHMODE_FLIPPED` switches this order, so the first level of arrays is the column name, and the second level the row number."

Disconnecting

If you want to explicitly disconnect from the database, you can call:

```
$mdb2->disconnect();
```

Even if you do not disconnect explicitly, MDB2 will do that for you in its destructor.

Using MDB2

Once you've connected to your database and have set some of the options and the fetch mode, you can start executing queries. For the purpose of the examples in this chapter, let's say you have a table called `people` that looks like this:

id	name	family	birth_date
1	Eddie	Vedder	1964-12-23
2	Mike	McCready	1966-04-05
3	Stone	Gossard	1966-07-20

A Quick Example

Here's a quick example, just to get a feeling of how MDB2 can be used. You'll learn the details in a bit, but take a moment to look at the code and see if you can figure it out yourself.

```
<?php
require_once 'MDB2.php';
// setup
$dsn = 'mysql://root:secret@localhost/mdb2test';
$options = array ('persistent' => true);
$mdb2 =& MDB2::factory($dsn, $options);
$mdb2->setFetchMode(MDB2_FETCHMODE_ASSOC);

// execute a query
$sql = 'SELECT * FROM people';
$result = $mdb2->query($sql);

// display first names
while ($row = $result->fetchRow())
{
    echo $row['name'], '<br />';
}

// release resources
$result->free();

// disable queries
$mdb2->setOption('disable_query', true);

// delete the third record
$id = 3;
$sql = 'DELETE FROM people WHERE id=%d';
$sql = sprintf($sql, $mdb2->quote($id, 'integer'));
echo '<hr />Affected rows: ';
echo $mdb2->exec($sql);

// close connection
$mdb2->disconnect();
?>
```

Executing Queries

To execute any query, you can use the `query()` or `exec()` methods. The `query()` method returns an `MBD2_Result` object on success, while `exec()` returns the number of rows affected by the query, if any. So `exec()` is more suitable for queries that modify data.

While you can basically perform any database operation with `query()`, there are other methods, discussed later, that are better suited for more specific common tasks.

Fetching Data

In the example above we had:

```
$sql = 'SELECT * FROM people';
$result = $mdb2->query($sql);
```

The variable `$result` will be an `MDB2_Result` object, or more specifically, it will be a database driver-dependent class that extends `MDB2_Result`, for example `MDB2_Result_mysql`. To navigate through the result set you can use the `fetchRow()` method in a loop.

```
while ($row = $result->fetchRow())
{
   echo $row['name'], '<br />';
}
```

Every time you call `fetchRow()`, it will move to the next record and will give you a *reference* to the data contained in it. Apart from `fetchRow()`, there are also other methods of the `fetch*()` family:

- `fetchAll()` will give you an array of all records at once.
- `fetchOne()` will return the value from first field of the current row if called without any parameters, or it can return any single field of any row. For example, `fetchOne(1,1)` will return **Mike**, the second column of the second row.
- `fetchCol($colnum)` will return all the rows in the column with number `$colnum`, or the first column if the `$colnum` parameter is not set.

Note that `fetchRow()` and `fetchOne()` will move the internal pointer to the current record, while `fetchAll()` and `fetchCol()` will move it to the end of the result set. So in the example above if you call `fetchOne(1)` twice, you'll get **Eddie** then **Mike**. You can also use `$result->nextResult()` to move the pointer to the next record in the result set or `$result->seek($rownum)` to move the pointer to any row specified

by `$rownum`. If in doubt, `$result->rowCount()` will tell you where in the result set your pointer currently is.

You also have access to the number of rows and the number of columns in a result set:

```
$sql = 'SELECT * FROM people';
$result = $mdb2->query($sql);
echo $result->numCols(); // prints 4
echo $result->numRows(); // prints 3
```

Shortcuts for Retrieving Data

Often it is much more convenient to directly get the data as associative arrays (or your preferred fetch mode) and not worry about navigating the result set. MDB2 provides two sets of shortcut methods - `query*()` methods and `get*()` methods. They take just one method call to do the following:

1. Execute a query
2. Fetch the data returned
3. Free the resources taken by the result

query*() Shortcuts

You have at your disposal the methods `queryAll()`, `queryRow()`, `queryOne()`, and `queryCol()`, which correspond to the four `fetch*()` methods explained above. Here's an example to illustrate the difference between the `query*()` and the `fetch*()` methods:

```
// the SQL statement
$sql = 'SELECT * FROM people';
// one way of getting all the data
$result = $mdb2->query($sql);
$data = $result->fetchAll();
$result->free(); // not required, but a good habit
// the shortcut way
$data = $mdb2->queryAll($sql);
```

In both cases if you `print_r()` the contents in `$data` and use the associative array fetch mode, you'll get:

```
Array (  [0] => Array ( [id] => 1
                        [name] => Eddie
                        [family] => Vedder
                        [birth_date] => 1964-12-23
```

```
                    )
        [1] => Array ( [id] => 2
                       [name] => Mike
                       [family] => McCready
                       [birth_date] => 1966-04-05
                    )
...
)
```

get*() Shortcuts

In addition to the `query*()` shortcuts, you have the `get*()` shortcuts, which behave in the same way, but also allow you to use parameters in your queries. Consider the following example:

```
$sql = 'SELECT * FROM people WHERE id=?';
$mdb2->loadModule('Extended');
$data = $mdb2->getRow($sql, null, array(1));
```

In this example the question mark in the statement is a placeholder that will be replaced by the value in the third parameter of `getRow()`.

You can also use named parameters, like this:

```
$sql = 'SELECT * FROM people WHERE id=:the_id';
$mdb2->loadModule('Extended');
$data = $mdb2->getRow( $sql,
                       null,
                       array('the_id' => 1)
                     );
```

Note that the `get*()` methods are in the Extended MDB2 module, which means that they are not available until you load that module using `$mdb2->loadModule('Extended')`.

Loading modules benefits from object overloading, which was not available before PHP5, so to get access to the methods of the Extended module in PHP4, you need to call them using:

```
$mdb2->extended->getAll($sql);
```

as opposed to:

```
$mdb2->getAll($sql);
```

getAssoc()

Another useful `get*()` method that doesn't have a directly corresponding `fetch*()` or `query*()` is `getAssoc()`. It returns results just like `getAll()`, but the keys in the result array are the values of the first column. In addition, if there are only two columns in the result set, since one of them is already used as an array index, the other one is returned as a string (as opposed to an array with just one element). A few examples to illustrate the differences between `getAll()` and `getAssoc()`:

```
$sql = 'SELECT id, name FROM people';
$mdb2->loadModule('Extended');
$data = $mdb2->getAll($sql);
```

`getAll()` will return an enumerated array and each element of the array is an associative array containing all the fields.

```
Array ( [0] => Array ( [id] => 1
                       [name] => Eddie
                     )
        [1] => Array ( [id] => 2
                       [name] => Mike
                     )
      ...
      )
```

If the same query is executed with `getAssoc()`, like `$data = $mdb2->getAssoc($sql);` the result is:

```
Array ( [1] => Eddie
        [2] => Mike
        [3] => Stone
      )
```

If your query returns more than two rows, each row will be an array, not a scalar. The code follows:

```
$sql  = 'SELECT id, name, family FROM people';
$mdb2->loadModule('Extended');
$data = $mdb2->getAssoc($sql);
```

And the result:

```
Array ( [1] => Array ( [name] => Eddie
                       [family] => Vedder
                     )
      ...
      )
```