PHP

Additional Notes

About these Notes

These notes are supplementary to PHP training delivered by Mark Simon. They are also available in PDF form from `resources.comparity.net`. Permission is granted for students of Mark Simon to use and reprint these notes. However permission is *not* granted to distribute these notes to others, nor to make copies for any purposes other private study.

A more complete set of instructions and notes is available as part of the training material supplied by **101 Courseware**. Information is available from `www.101courseware.com`.

Notes developed by Mark Simon
Version: 0.1

Comparity Net
`resources.comparity.net`

Contents

A

Appendix A: Setting Up

Installing WAMP

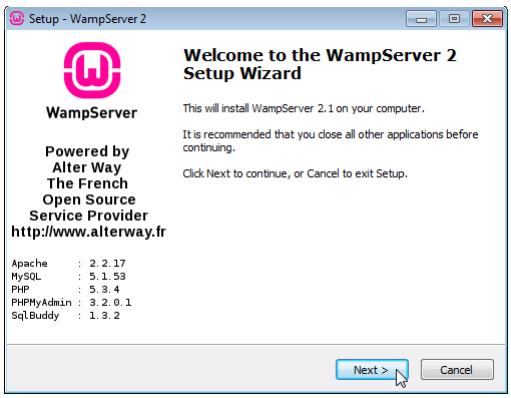
The easiest way to install a Web Server on Windows is to use the Wamp Server. It is a Free and Open Source packaging of Apache, PHP and MySQL.

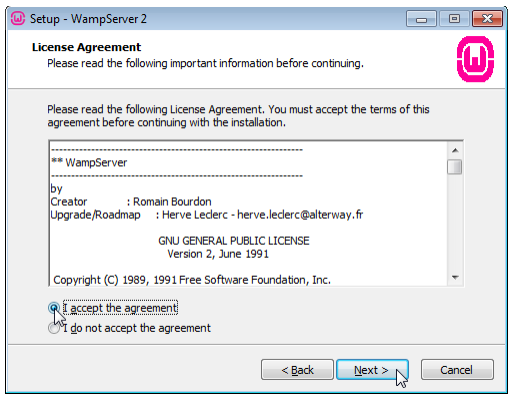
You can download the latest version of Wamp here:

`http://www.wampserver.com/en/`

Starting Up

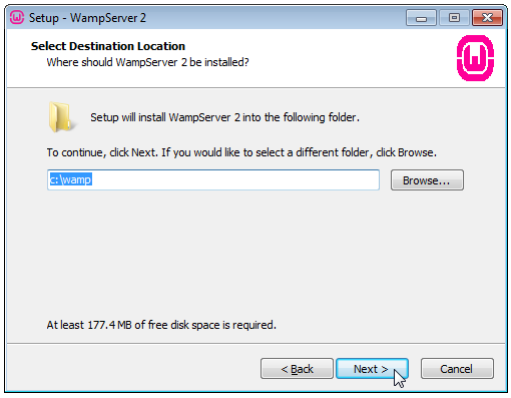
Start the installer and accept the License Agreement:





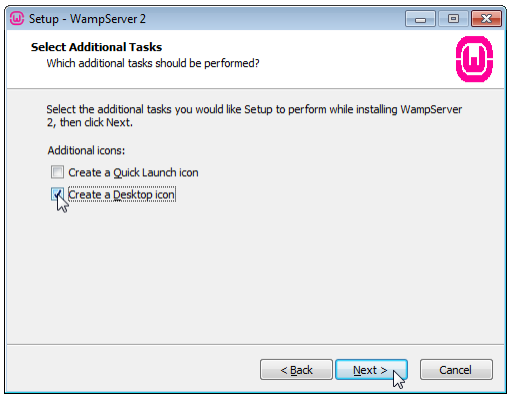
Location

Choose the location of the Package. The default location is OK; you should make sure that your package is installed in a path without spaces.



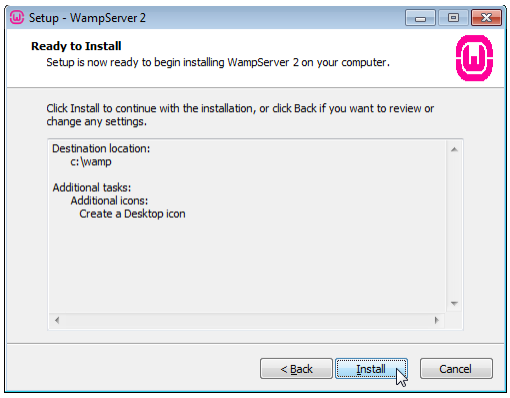
Shortcuts

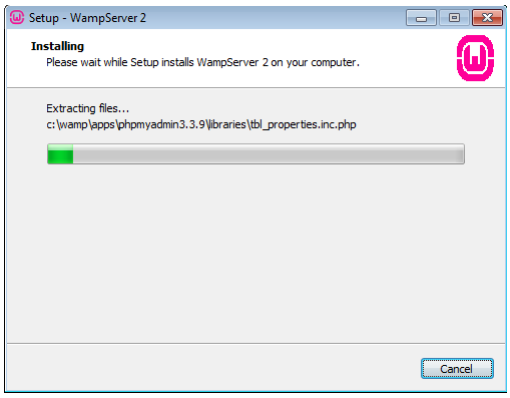
Choose whether you want Quick Launch or Desktop Shortcut icons.



Installing

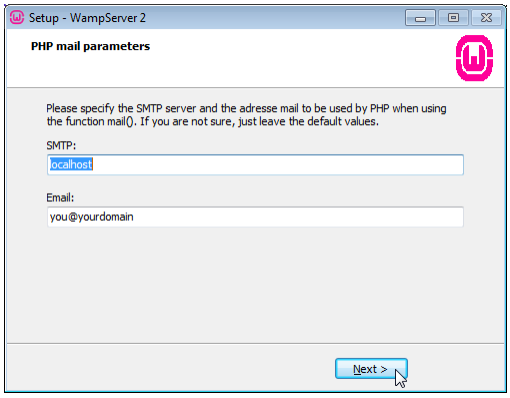
The package will now install.





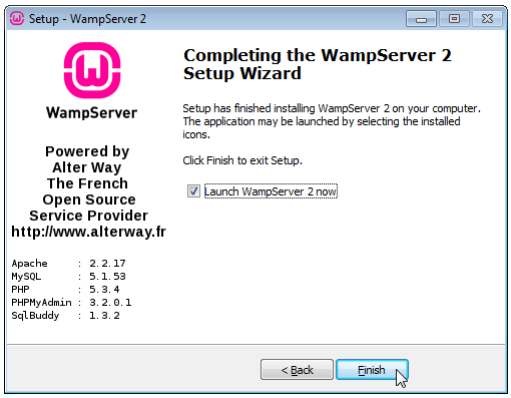
SMTP & Mail

If you know the IP address of a mail server, enter it now. This can be changed later.



Finish

You can now launch the Wamp Server.



Wamp does not install itself as a startup program. If you want to do this, you will need to copy a shortcut to your startup folder.

Installing & Setting Up Komodo Edit

To develop PHP, you can use any text editor of your choice. There are many text editors which are suited to developing code.

Komodo Edit is a Free and Open Source programming editor which has many additional features to text editing. Among other things, it is aware of the Web languages you will use, and help in checking simple typing errors.

Komodo Edit is available from:

```
http://www.activestate.com/komodo-edit
```

There is also Komodo IDE, a paid product which is a more complete development environment. Among other things, it will help you to debug running code.

Komodo Edit is very customisable. It also has the ability to store code snippets and run macros.

The Extras folder includes a folder of Snippets and Macros which you can use.

Suggested settings for Komodo Edit

- ☑ Turn On
- ☐ Turn Off
- ■ Choose for yourself

Category	Group	Settings
Appearance	(General)	? Number of Projects ? Number of Files
Editor	(General)	■ Show White Space characters ■ Show end-of-line characters ☑ Show line numbers
	Indentation	☑ Prefer Tab characters over spaces 4 Number of spaces per indent 4 Width of each Tab character Per Language Indentation Settings: (Set for HTML, PHP, JavaScript, SQL, CSS)
	Save Options	☑ Clean trailing white space and EOL markers ■ Ensure file ends with EOL marker Auto-save every 0 seconds
Fonts and Colors	Fonts	Encoding UTF-8
	Common Syntax	Choose a colour for your comments, and turn off italics.
Internationalization	Default Editor Encoding	☐ Use Encoding Defined in Environment: ... Custom Encoding UTF-8
	Custom Encoding	Language-Specific Default Encoding (Set for HTML, PHP, JavaScript, SQL, CSS) ☐ Signature (BOM)
New Files	New Files	Specify the default language for files created using the 'New' Button. (Set HTML, PHP or whatever; even Text will do); Specify the end-of-line (EOL) indicator for newly created files. DOS/Windows (\r\n) ☑ Assign new, empty files this EOL
Web & Browser		Which browser should Komodo use when opening URLs? (System Default, or Firefox if not set to default) Which method should be used to preview files in browser? ■ Preview in Komodo Tab, same tab group ■ Preview in external browser
Mapped URIs	Mapped URIs	URI http:// ... Maps to file:// ...

Useful Firefox Add-ons

Apart from the benefits of having a modern Web Browser which supports current standards, Firefox also supports a vast number of extensions or add-ons.

Below are some add-ons which may help you in your web development work:

Page Info Button

`https://addons.mozilla.org/en-US/firefox/addon/page-info-button/`

The Page Info button simply makes a button available for the Page Info, but it also includes a shortcut key, `Ctrl-I`.

View Cookies

`https://addons.mozilla.org/en-US/firefox/addon/view-cookies/`

Althougth all Cookies are already visible in Firefox, the View Cookies add-on makes Cookies for the current page visible and editiable in the Page Info.

Firebug

`https://addons.mozilla.org/en-US/firefox/addon/firebug/`

Firebug is a set of tools to help you follow and trouble shoot HTML, CSS and JavaScript on individual pages.

Html Validator

`https://addons.mozilla.org/en-US/firefox/addon/html-validator/`

The HTML Validator checks pages as they are loaded for HTML errors, and displays an icon to indicate the results. When you view the page source, HTML Validator gives more detail on HTML errors.

ColorZilla

`https://addons.mozilla.org/en-US/firefox/addon/colorzilla/`

CollorZilla is an eye-dropper and colour picker for Firefox. It allows you to select any colour on the page and copy its values for inclusion in CSS.

Pearl Crescent Page Saver

`http://pearlcrescent.com/products/pagesaver/`

The Pearl Crescent Page Saver allows you to take an image, not only of the visible portion of a page, but the whole of the page, including off-screen parts.

Web Developer

`https://addons.mozilla.org/en-US/firefox/addon/web-developer/`

Web Developer is a set of tools for

B

Appendix B: Virtual Domains

How to fake a Local Domain

The following instructions will show you how to emulate a domain on your local testing or development server, in particular, using WAMP. This uses Apache's ability to work with Virtual Hosts.

For our purposes, we will use the domain `example.com`, but you can use a real one if you like. Just note that if you do use a real one, the following will take precedence, and you will not be able to access the real domain.

Setting the hosts File

The first thing is to tell Windows that your domain is really just the localhost in disguise. For this we use the hosts file.

In most current versions of Windows, you will find the hosts file in the following location:

```
C:\WINDOWS\system32\drivers\etc\hosts
```

This is a text file with a lot of comments (`# ...`), and one or two entries. The entries take this form:

```
[ip address]  [domain name]
```

You will already see the entry

```
127.0.0.1     localhost
```

which tells you that the name `localhost` is really just a name for `127.0.0.1`, which is the standard IP address of the current machine. We say that `localhost` resolves to `127.0.0.1`

We need to add the following to resolve the `example.com` address to the local machine:

```
127.0.0.1     localhost
127.0.0.1     example.com
127.0.0.1     www.example.com
```

Now you can save the file.

Setting up Apache

The next step will be to tell Apache how to react if you access this domain.

httpd.conf

The main configuration file is called `httpd.conf`. Its actual location will vary from version to version, but it is something like this:

```
C:\wamp\bin\apache\Apache2.2.17\conf\httpd.conf
```

The current version of Wamp may include a different version of Apache, so you will need to make adjustments here.

About 400 lines down, find the following lines:

```
# Virtual hosts
#Include conf/extra/httpd-vhosts.conf
```

The lines are commented out (`#...`), which means that no virtual hosts are enabled.

Add the following (uncommented) line below it:

```
# Virtual hosts
#Include conf/extra/httpd-vhosts.conf
Include conf/extra/example.conf
```

This means that we will put our virtual host settings in a separate file called `example.conf`.

example.conf

Now create a new file, and save it as `example.conf` inside the extra folder, inside the `conf` folder.

Put the following in the `example.conf` file:

```
<VirtualHost *:80>
  ServerAdmin webmaster@localhost
  DocumentRoot "c:/wamp/www"
  ServerName localhost
  ErrorLog "logs/localhost-error.log"
  CustomLog "logs/localhost-access.log" common
</VirtualHost>
<VirtualHost *:80>
  DocumentRoot "c:/wamp/www/australia"
  ServerName www.example.com
  <directory "c:/wamp/www/australia">
    Options Indexes FollowSymLinks
    AllowOverride all
    Order Deny,Allow
    Deny from all
    Allow from 127.0.0.1
  </directory>
</VirtualHost>
```

Without going into too much detail, this tells us that accessing `localhost` will work as before, but accessing `www.example.com` (which also resolves to `127.0.0.1`) will take you directly to your nominated folder (in this case, `australia`).

Now, save the file, and restart your server. You should now be able to access `www.example.com` directly.

C

Appendix C: Using PHPMyAdmin

Contents

The most popular freely available programs include:

- The MySQL Command Line Interface (CLI). If you are happy typing in all of your commands, then this is a very simple tool. The CLI on Linux is much more sophisticated than that on Windows.
- The MySQL Workbench. This is a replacement for the older MySQL GUI tools, and is produced by MySQL. While the Workbench is much more sophisticated than the older GUI tools, it is harder to use, and many find it overwhelming.
- PHPMyAdmin. This is a MySQL management tool written entirely in PHP, to be used in a Browser. Being a Web based tool, some aspects are slow and awkward, but it makes managing a database over the Internet practical.

PHPMyAdmin

Although our main database work will be in PHP itself, we will use PHPMyAdmin to perform a number of important tasks.

Creating a Database & Users

The simplest way to create a new database is via PHPMyAdmin. Since it is not a task we will perform very often, we can happily rely on this tool.

For the administration of our database, we can use PHPMyAdmin to create database users. This allows us to control access to tables and databases, and to set passwords.

Creating & Browsing Tables

PHPMyAdmin has a table creation tool which allows us to define the columns with a few simple choices. The result is a CREATE TABLE command which we could also by hand issue if we prefer.

To check the contents of our tables, we can use PHPMyAdmin to browse the tables, and, to some extent, manage the data.

Backup and Restore

PHPMyAdmin can export the data into am SQL file, and re-import the data to create a new copy of the database. This will be useful when transferring the data to another server, or when keeping a copy of the data for backup purposes. Note that PHP may run into the 2Mb upload limit if there is too much data.

Using PHPMyAdmin

Generally, there are two ways to use PHPMyAdmin.

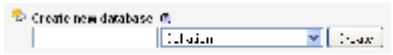
- The GUI offers a point-and-click approach to performing certain tasks. For some tasks, this is the simplest, as a deep knowledge of SQL is not required, though you will need to know enough to make the right choices. For some tasks, this can be tedious.
- The SQL Window allows you to dispense with the GUI and enter SQL commands to be executed. This is especially useful if you already have the SQL statements prepared, or if the GUI tool cannot handle it easily.

Using the GUI

Creating A Database

1. Select the Databases Tab in the main window:

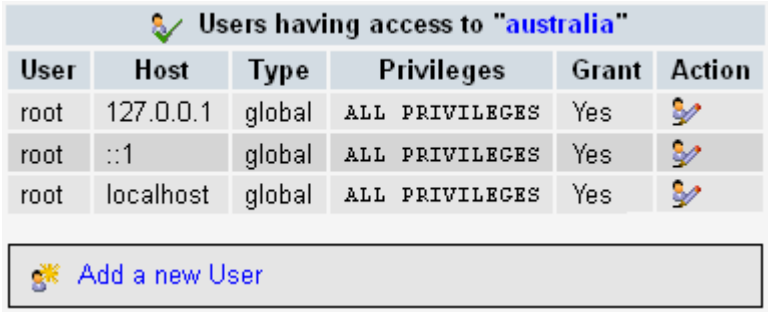
2. In the next window, enter a new database name, and click on Create:



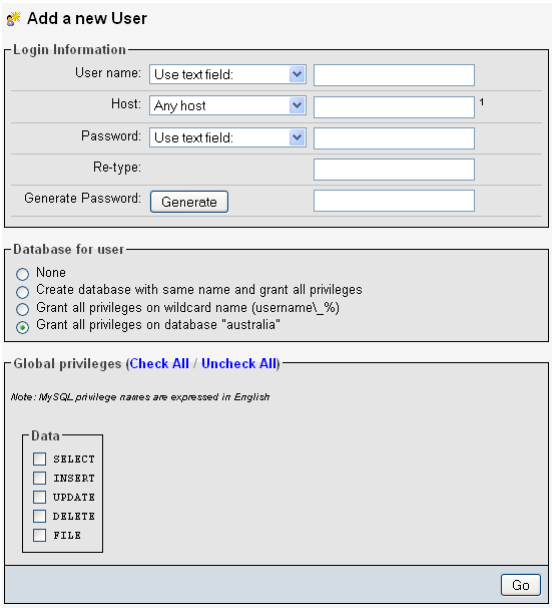
Creating a User

1. Select the Privileges Tab

2. Select `Add a new User`



3. Enter the Details.

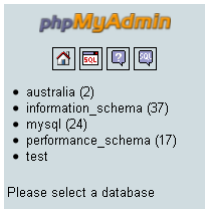


This will include:

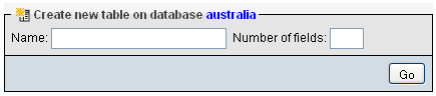
- The username and password
- The server address
- The databases which this user can use
- The data privileges. The first 4 are for the standard SQL statements, while the last allows importing from files.

Creating a Table

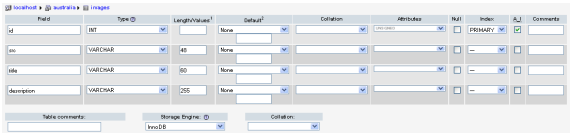
1. Select the database from the panel on the left:



2. In the next window, enter the name of the new table. Estimate the number of fields you will need. This is purely for layout; you can always add to their number later. Press Go:

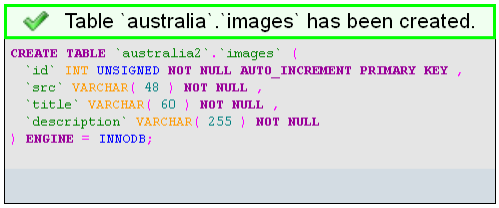


3. In the next window, enter the names and attributes of the fields in the table, and select Save:



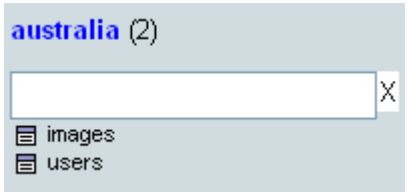
There is also a section to allow you to add more fields, with a Go button. Don't mistake this for saving the table.

4. When finished, you will see the following SQL statement:



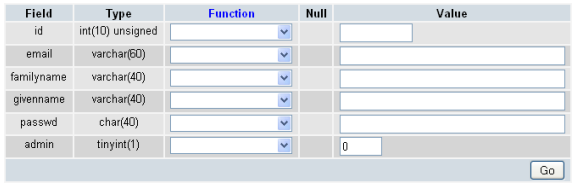
Adding a Record

1. Having selected the database, select the table from the panel on the left:



2. Select the Insert tab:

3. You will see a form to enter your data:



4. The form describes the type of data, whether null is permitted, and fills in the default values. You can also use a function from the function list to modify the value you enter.

D

Appendix D: Configuring PHP

How to Configure PHP

At its most fundamental level, you can reconfigure PHP by compiling it with different options, or by including different libraries to supplement its behaviour. However, the following three techniques are used for a working implementation of PHP:

php.ini

PHP uses settings from the php.ini file when it starts up. These settings affect all PHP scripts on the server, so changes to this file are global.

Generally, php.ini is inaccessible if you are not using your own personal server, such as a hosted or coporate server. If you do make changes to this file, then you will need to restart your web server to reload these settings.

.htaccess

If you are running PHP under the Apache web server, then you may be able to use the .htaccess file to adjust your configuration. (The dot at the beginning of its name tends to hide it from view on Unix-type servers such as Linux, and is common for configuration files.)

The .htaccess file should be placed in a directory where you want the changes to be made. It will be applicable to the current directory as well as all of the subdirectories. You can have different .htaccess files in different directories: the current settings will always override the settings of parent directories.

Apache will apply the settings when it loads a file. This means that changes are instantaneous, and you will not have to restart your server. It also means there is a slight overhead of additional file handling when you load pages.

php ini functions

PHP will also allow you to re-configure settings using PHP functions. This is most suitable if you want to make changes for a single script (or a number of scripts using an included configuration script). It is also an option if your server cannot support .htaccess files, or if it deliberately ignores them.

Not all PHP settings can be changed this way. In particular, some changes will be made too late, since they may be required if there are problems with processing the script altogether.

Setting PHP options

Generally each setting has a name and a value. In `php.ini` the format is:

```
php.ini   setting_name = value
```

For `.htacess` files, which may include other non-php settings, the format is:

```
.htacess  php_value   setting-name value
```

For a PHP function, you use the function `ini_set(name,value)`. In this function, both parameters are supposed to be strings, though the second parameter (value) may be a number:

```
function  ini_set(name,value);
```

Some Useful Configuration Options

Below are some PHP configuration options which you may find useful for individual scripts or projects. They will be described for php.ini, .htaccess and php ini functions.

PHP Execution

The following settings may be useful for some scripts which unusually large processing requirements.

`max_execution_time = 30`

This setting forces PHP to timeout if the script has been going too long, such as in an infinite loop. Generally speaking, 30 seconds should be plenty, though more time may be needed occasionally for processing a large amount of data.

`max_input_time = 60`

Input Timeout measured in Seconds

`memory_limit = 16M`

Memory Limit measured in bytes

SMTP Server

`SMTP = localhost`

Generally, the SMTP host is the same as the Web server (as with the database server), but on some systems, you may need to set it to a different IP address. This is especially the case when developing on a Windows platform without a built-in mail program.

File Uploads

`upload_max_filesize = 2M`
`post_max_size = 8M`

PHP limits the size of individual files, as well as the total size for all uploaded data, including uploaded files. If you wich to accept larger files, then change these settings.

Legacy Settings

PHP originally used some settings to make scripting easier. Unfortunately, they also backfired, and so should be disabled. In modern versions they are disabled by default, but you should make sure.

```
register_globals = Off
```

`register_globals` enabled all data from the web browser, including cookes, POST data and GET data (including anything in a query string) to be copied into PHP variables. Needless to say this is inherently insecure, as the script has no way of determining the source of data.

If your server has this turned on, you should use this configuration option and look for another host.

```
magic_quotes_gpc = Off
```

As an early attempt to avoid SQL injection attacks, PHP automatically "escaped" all quote characters (putting a back-slash `\` in front of them). However, this is an incorrect solution (proper SQL should use two single quotes `''` to escape them), and generally made a mess of legitmate quote characters, and should therefore be disabled.

A better way of avoiding SQL injection attacks is to use prepared SQL statements, available in all decent modern SQL databases.

Sessions

PHP Sessions allow you to keep track of user-related data between pages. Session data is automatically refreshed every time the user re-visits a page in your application, but, since you cannot rely on a user notifying you when they have left the site, PHP will automatically expire the data after a certain time.

```
session.gc_maxlifetime = 600
```

This settings, measured in seconds, determines when the session can be regarded as expired. If the user visits a page within this time, then the timeout will be renewed.

```
session.gc_probability = 1
session.gc_divisor = 100
```

PHP does not actually expire the session data directly after the lifetime of the session. Rather, the old session will remain until PHP gets around to it, so the lifetime is really a minimum life time, but not guaranteed to be the end. PHP will remove all data based on a probability calculation:

$$probability = \frac{session.gc\_probability}{session.gc\_divisor}$$

By default `session.gc_divisor` is 100, meaning that sessions will expire 1 time in 100. In a busy server environment, this may be a few seconds (after the `max_lifetime` setting), but for testing and development, this may be a much longer time. For this reason, you may wish to change this to 1 for testing only.

Error Reporting

PHP will normally report on errors, either on the screen or in a log file. Error reporting can be set to different levels, and you may choose whether PHP will display only serious errors, or less serious errors from which PHP can continue. Generally, you should display all possible errors during development (to help eliminate all bugs in your code), but hide them for a production site (displaying errors is not only unprofessional, but can create security issues).

```
error_reporting = E_ALL
display_errors = On
log_errors = On
```

These settings are best for development.

Note that these settings are best set from `.htaccess`, rather than from a PHP function. Since a script may have errors which affect the whole of the script, it may not have the opportunity to enable error reporting. This will result in the "white screen of death", a blank screen where an error message might have been.

Sample Configuration Settings

Below are sample configuration settings in both `.htaccess` form and as PHP functions (you may gather those PHP function calls into a configuration file to be included).

Note the subtle differences between formats and values.

The `.htaccess` file options all begin with `php_value`, since `.htaccess` can include non-PHP settings, and do not have equals (=) before the values. Also, values such as `E_ALL` are converted to `-1`, since `E_ALL` has a specific meaning to PHP, but not to Apache.

The PHP functions all follow the same pattern. All values have been written as strings, though numbers can be written bare. Also note that PHP `ini` functions may be individually included in individual scripts

.htaccess

```
# .htaccess
# SMTP
  php_value   SMTP "localhost"

# Uploads
  php_value   post_max_size 8M
  php_value   upload_max_filesize 2M

# Legacy
  php_flag register_globals Off
  php_flag magic_quotes_gpc Off

# Sessions
  php_value   session.gc_probability 1
  php_value   session.gc_divisor 1
  php_value   session.gc_maxlifetime 600

# Errors
  php_value   error_reporting -1
  php_flag display_errors On
  php_flag log_errors On

# Misc
  php_value   max_execution_time 30
```

config.php

```
<?php
  //php.ini changes
  //see  http://www.php.net/manual/en/function.ini-set.php
  //   http://www.php.net/manual/en/configuration.changes.php
  //   http://www.php.net/manual/en/ini.list.php

  //SMTP
    ini_set('SMTP','localhost');

  //Uploads
    ini_set('post_max_size','8M');
    ini_set('upload_max_filesize','2M');

  //Legacy
    ini_set('register_globals','0');
    ini_set('magic_quotes_gpc','0');

  //Sessions
    ini_set('session.gc_probability','1');
    ini_set('session.gc_divisor','100');
    ini_set('session.gc_maxlifetime','1440');

  //Errors
    ini_set('error_reporting','E_ALL');
    ini_set('display_errors','1');
    ini_set('log_errors','1');

  //Misc
    ini_set('max_execution_time','30');
?>
```

E

Appendix E: PDO

PDO (PHP Data Objects) provides a vendor-neutral method of accessing a database through PHP. This means that, once you have established a connection to the specific database, the methods used to access and manipulate data are all generic, and do not require re-writing if you change the type of database. Features which may not be present in a particular database will generally be emulated, or at least ignored.

The main references for PDO are:

http://www.php.net/manual/en/class.pdo.php
http://www.php.net/manual/en/class.pdostatement.php

PDO Objects

PDO makes use of two main objects. The PDO object itself represents a connection to the database, and provides simple methods to execute an SQL statement. It also provides a method to prepare an SQL statement for later use. The `PDOStatement` object represents a prepared statement, as well as a result set from an executed SQL statement.

`PDO` Object

This represents a connection to the Database. All database operations are initiated through the PDO object. The PDO object is created when you connect to the database. After that, you use its methods to access the database. The most useful methods are:

`exec()`	Execute an SQL statement returning the number of rows affected
`query()`	Execute an SQL statement returning a result set as a `PDOStatement`
`prepare()`	Prepares a statement returning a result set as a `PDOStatement`. You can use question marks (?) for values. You can then call the `execute(array())` method

`PDOStatement` Object

The `PDOStatement` represents a prepared statement, as well as a returned result set. The name is possibly confusing, since it represents a prepared statement before it is executed, as well as the result after it is executed.

A `PDOStatement` is created as a result of a `PDO->query` operation (where it represents a result set), a `PDO->prepare` operation (where it represents a prepared statement) or a `PDO->execute` operation (where it represents a result set from your prepared statement).

The most useful methods are:

For a prepared statement:

`execute()`	Execute the prepared statement. You can use an array of values to replace the question mark parameters

For a result set:

`fetch()`	Returns the next row. Useful arguments: `PDO::FETCH_ASSOC`, `PDO::FETCH_NUM`, `PDO::FETCH_BOTH` (default)
`fetchAll()`	Returns the whole result set as an array
`fetchColumn()`	Returns a single column of the next row.

`PDOStatement` allows you to iterate through the result set can be iterated with `foreach()`.

Working With PDO

PHP Data Objects allow you to work with a database without having to worry about the details of the database functions. In principal, you can use the same code to work with different database types, though some SQL statements may need adjustment, due to differences between databases.

PDO makes use of two main objects. The PDO object itself represents a connection to the database, and provides simple methods to execute an SQL statement. It also provides a method to prepare an SQL statement for later use. The PDOStatement object represents a prepared statement, as well as a result set from an executed SQL statement.

Establishing a Connection

Before working with PDO, you will need to create a connection. This is in the form of a PDO object, which represents the connection.

You will require the following:

- A connection string: this informs PDO which database you are connecting to. This will also include the location of the data.
- User name and password. Depending on the database, you may need to authenticate your connection.

```
$database='things';
$user='me';
$password='secret';

$dsn="mysql:host=localhost;dbname=$database"; //  mysql
$dsn="sqlite:$database.sqlite";               //  sqlite
```

A connection attempt may result in an error. The normal behaviour is to display as much information as possible, but this is probably more than you wish to share with others. For this reason, it is best to wrap the connection inside a try ... catch block:

```
try {
   $pdo = new PDO($dsn,$user,$password);   //  mysql
   $pdo = new PDO($dsn);                   //  sqlite
} catch(PDOException $e) {
   die ('Oops');                //  Exit, displaying an error message
}
```

Other Databases

If you want to connect to a different database, the following connection strings may be used:

```
$dsn="pgsql:host=localhost;dbname=$database";
$dsn="odbc:Driver={Microsoft Access Driver (*.mdb)};
   Dbq=C:\$database.mdb;Uid=Admin";
$dsn="sqlite::memory";
```

For the connection script you may use:

```
try {
   $pdo = new PDO($dsn);            //  sqlite, MSAccess
   $pdo = new PDO($dsn,$user,$password);   //  mysql, postgresql, etc
} catch(PDOException $e) {
   die ('Oops');                //  Exit, displaying an error message
}
```

Prepared Statements and SQL Injection

The Risk: SQL Injection

The biggest risk to your database comes from including user data in your SQL statements. This may be miss-interpreted as part of the SQL statement. Where a user is deliberately supplying this data to break into the database, this is called SQL Injection.

For example, suppose you are performing a simple login using an email and password supplied by the user. The SQL statement might be something like this:

```
SELECT count(*) FROM users WHERE email='...' AND passwd='...'
```

Now, suppose the user supplies the following as their email address:

```
fred' OR 1=1; --
```

This clearly is not a proper email address, but it might still be inserted as follows:

```
SELECT count(*) FROM users WHERE email='fred' OR 1=1; -- ' AND passwd='...'
```

The condition `OR 1=1` will always be true, and what follows after the comment code `--` will be ignored. This simple injection will allow a user to break into the database.

The problem arises because the inserted data will be interpreted with the rest of the SQL.

Prepared Statements

Most databases allow you to prepare a statement before executing it. SQL statements need to be interpreted, checked for errors, analysed and optimised, all before executing them with actual data.

To protect yourself against SQL injection, you prepare your SQL statement first, and then execute it with the data afterwards. When the data is inserted, it can no longer be interpreted, and so will be passed purely as data. Note that the above email address would presumably not be in the database, and would result simply in a failed login.

To prepare and execute the data, you would follow these steps:

1. Define your SQL, using question marks as place holders
2. Using the `PDO` object, prepare the SQL. This will result in a `PDOStatement` object
3. Execute the `PDOStatement` object with an array of the data to replace the question marks
4. If your SQL is a `SELECT` statement, you will need to examine the results (later).

For example:

```
$sql='SELECT * FROM users WHERE email=? AND passwd=?';
$sql='INSERT INTO users(email,passwd) VALUES(?,?)';
$sql='UPDATE users SET email=?, passwd=? WHERE id=?';
$sql='DELETE FROM users WHERE id=?';
```

```
$pds=$pdo->prepare($sql);
$pds->execute(array(...,...));
```

Note that you do not put the question mark place holders inside quotes even if they represent strings. If you do, the quotes will be added to the data.

Note also that execute always takes an array argument, even if there is only one value.

Remember, preparing your SQL statements is important if your data comes from a user. This is is essential to protect yourself from SQL injection

If there is no user data involved, or if the data is guaranteed be numeric (which could not possibly contain spurious SQL), then you might prefer the direct methods below.

Repeated Execution

Another use of prepared statements is with repeated execution. Whether the data is suspect or not, if you need to execute the same statement many times, it can be more efficient to prepare the statement once, and to execute the prepared statement many times, as in a loop.

For example, suppose you have a number of rows to be inserted, the data for which may already be inside an array. Then you could execute the SQL as follows:

```
$sql='INSERT INTO products(description) VALUES(?)';
$pds=$pdo->prepare($sql);
foreach($products as $p) {
  $pds->execute(array($p));
}
```

Even if the data isn't suspect, the above code needs to prepare the statement only once, and so the overhead of interpreting, analysing and optimising the statement is reduced. The multiple executes will run much faster.

Unprepared (Direct) SQL Statements

If there is no risk of malicious user data, then you may not need to prepare your statements first. This will result in slightly simpler code. In this case, you can use one of two PDO functions to run your SQL statement.

SELECT Statements

SELECT Statements expect a result set. In some cases, the result set will have only one row, while in some other cases, the result set may have many.

To get data using an unprepared statement:

1. Define your SQL, including the data. This may include data in variables, if you use a double-quoted string.
2. Using the PDO object, use the `query()` function on the SQL statement. This will also result in a `PDOStatement` object, but this will contain the result set if any.
3. In the case of a `SELECT` statement, you will need to examine the results (later).

For example:

```
$sql='SELECT code,description,price FROM products';
$sql="SELECT code,description,price FROM products WHERE id=$id";

$pds=$pdo->query($sql);
```

The variable `$pds` will contain the result set. It is technically a `PDOStatement` object, though, in this case, does not contain a prepared statement.

The variable `$id` in the second SQL statement above may be subject to SQL injection unless your data has already been tested for this. For example, the PHP `intval()` function will always guarantee an integer, which cannot contain malicious SQL.

INSERT, UPDATE, and DELETE Statements.

`INSERT`, `UPDATE` and `DELETE` statements do not expect a result set. In each case PDO will return a value which is the number of records affected by the SQL statement, but you may choose to ignore this result.

To put data using an unprepared statement:

1. Define your SQL, including the data
2. Using the PDO object, use the exec() function on the SQL statement. This will return the number of rows affected.

For example:

```
$price=20; $id=3;
$sql="UPDATE products SET price=$price,modified=now() WHERE id=$id";
$pdo->exec($sql); //  or $rowcount=$pdo->exec($sql);
```

The variable, `$rowcount`, will contain the number of rows affected. Typically for an `INSERT` statement, or when a `WHERE` clause has been used to identify a single row, this will be 1. However, it may contain 0 or any other number, depending on the SQL statement.

Again, as above, your variables need to be checked for malicious SQL before including them directly into an SQL statement.

PDO Techniques

SELECT Data

To select data from a database table, use the SELECT command:

```
SELECT ... FROM ...;
SELECT ... FROM ... WHERE ...;
```

Prepared Statements	Unprepared Statements

`$sql='SELECT ... FROM ... ...'`
`$pds=$pdo->prepare($sql);`
`$pds->execute(array(...));`
	`$sql="SELECT ... FROM ... ...";`
	`$pds=$pdo->query($sql);`

In both cases, you will have a result set in `$pds`.

Fetching Data

To retrieve the data, you can fetch one row at a time, or you can iterate through collection.

To fetch a single row:

```
$row=$pds->fetch();
```

To iterate through the collection:

```
while($row=$pds->fetch()) {
   ...
}
```

or, more simply:

```
foreach($pds as $row) {
   ...
}
```

The Result Set

Each row in a result set, unless set otherwise, will be an array containing the data twice, both with numbered keys, and with associative keys.

For example:

```
SELECT code,description,price FROM products
```

will return rows of the following arrays:

key	value
code	[code]
description	[description]
price	[price]
0	[code]
1	[description]
2	[price]

This redundancy will allow you to read the values in a convenient way. For example, to use the row data inside a string, you may wish to use the associative keys:

```
$tr="<tr><th>$row[code]</th><th>$row[description]</th>
  <th>$row[price]</th></tr>";
```

On the other hand, you can use the numeric keys as follows:

```
$code=$row[0];
$description=$row[1];
$price=$row[2];
```

In the above example, you can also use PHP's list construct:

```
list($code,$description,$price)=$row;
```

The list construct only works with numeric keys.

Fetching the Whole Result Set

You can fetch the entire result set into a array with all of the rows:

```
$rows=$pds->fetchAll();
```

You can, but you probably shouldn't, unless you can be sure that your result set isn't too big for memory.

Fetching a Single Column

Sometimes, you need only one column of the result set. For this you can use `fetchColumn()`. The optional parameter is the column number (starting at 0, which is the default). Each subsequent call to `fetchColumn()` will fetch the same column from the next row.

Simple PDO Recipes

Count Records

Count All Records

```
$sql='SELECT count(*) FROM ...';
$count=$pdo->query($sql)->fetchColumn();
```

or

```
$count=$pdo->query('SELECT count(*) FROM ...')->fetchColumn();
```

Count Selected Records (Prepared)

```
$sql='SELECT count(*) WHERE ...';
$pds=$pdo->prepare($sql);
$pds->execute(array(...));

$count=$pds->fetchColumn();
```

Login Script

```
$sql='SELECT ... FROM users WHERE email=? AND passwd=?';
$pds=$pdo->prepare($sql);
$pds->execute(array(...));

if($row=$pds->fetch()) {
  // successful; $row now contains the rest of the details
}
else {
  // unsuccessful ($row is FALSE)
}
```

Summary of PDO

Connection

To connect to a database

```
$pdo=new PDO(DSN[,USER,PASSWORD]);
```

Because the default error reporting might give away to much detail, it is normal to include the connection inside a try ... catch block:

```
try {
   $pdo=new PDO(DSN[,USER,PASSWORD]);
}
catch (PDOException $e) {
   // Handle Error
}
```

Executing Simple Statements

Simple statements include any data directly in the SQL string.

INSERT, UPDATE & DELETE

```
$sql="...";
$count=$pdo->exec($sql);
```

The returned value will be the number of rows affected.

SELECT Statements

```
$sql="SELECT ... FROM ... ";
$result=$pdo-?query($sql);
```

The returned value will be a `PDOStatement` pointing to the result set. See Reading Data (below) on how to use this.

Executing Prepared Statements

```
PDOStatement=PDO->prepare($sql);
PDOStatement->execute(array(...));
```

Although some data may be included directly in the SQL string, the major benefit from preparing statements is the ability to insert the data after the SQL string has been prepared. In this case you replace the data with question mark place holders; place holders are never to be quoted, even if they are strings.

```
$sql='INSERT into ... VALUES(?,?)';
$pds=$pdo->prepare($sql);
$pds->execute(array(...,...));
```

Reading Data

Whether or not the SQL statement was prepared, the data set will always be in a `PDOStatement`.

Reading a Single Row

Each row is an array containing data with both numeric and associative keys. You may use either (or both) types of key as convenient.

To fetch a single row:

```
PDOStatement->fetch();
$row=$pds->fetch();
```

This will fetch the next row, which may, of course be the first or only row.

If there is no next row (or no result to begin with), fetch() will return FALSE.

Reading Multiple Rows

To fetch multiple rows:

```
while($row=PDOStatement->fetch() {
   …
}
```

or

```
foreach(PDOStatement as $row) {
   …
}
```

Each will produce exactly the same result. The `foreach` statement is similar to iterating through an array, and automatically fetches the next row and assigns it to `$row`.

Reading a Single Column

For convenience there is a function which will read a single value from a row. This will return a simple value, and avoids having to deal with the data in an array.

```
PDOStatement->fetchColumn([col]);
```

The optional parameter is the number of the column, and defaults to 0, the first column.

This is particularly handy when the result set itself has only one row.

For example, to count the number of records in a table:

```
$sql='SELECT count(*) FROM users';
$result=$pdo->query($sql);
$count=$result->fetchColumn(0);
```

or, more simply,

```
$count=$pdo->query('SELECT count(*) FROM users')->fetchColumn();
```

Getting the Last Auto-Incremented Key

Many databases offer an auto-incremented value for a field, typically a primary key. This is either as an auto-incremented attribute of the field, or as a special sequence. When a new roe is inserted into the table, and the auto-incremented field is omitted or set to `NULL`, its value will be set to the next number in the sequence.

Generally speaking Auto-Incremented fields are a non-standard feature of SQL. Although most databases offer a version of this feature, they are implemented differently. In particular, it can be difficult to get the last auto-incremented value reliably.

`PDO->lastInsertId()`

PDO wraps the various techniques for getting the last auto-incremented value inside the `PDO->lastInsertId()` function. Note that this will give the last auto-incremented value from the database, which may or may not be that of your table of interest. Or to put it another way, you should call this function immediately after you insert the record, before its value is lost on the next insert.

Error Reporting

By default, PDO is silent about errors. This can make trouble shooting very difficult if there is an error with your SQL statement, but the PHP code itself is OK. Sometimes, if you are expecting a record set, the error will be apparent in the next few lines, as you will end up trying to read from an empty record set.

At the development stage, you will want your errors to be as clear as possible, so you might want to change your error reporting to be less silent. For this we use the `PDO->setAttribute()` to change the `ATTR_ERRMODE` property:

```
// Default
   PDO->setAttribute(PDO::ATTR_ERRMODE,PDO::ERRMODE_SILENT);

// Warning Only
   PDO->setAttribute(PDO::ATTR_ERRMODE,PDO::ERRMODE_WARNING);

// Die, displaying Error
   PDO->setAttribute(PDO::ATTR_ERRMODE,PDO::ERRMODE_EXCEPTION);
```

For development, you should use the `ERRMODE_EXCEPTION` value. You might want to set it back to `ERRMODE_SILENT` for a production environment.

Summary of Process

F

Appendix F: Code Snippets

In the following exercise, you will develop a database for your code snippets. Apart from the practice, you may find this database a useful resource for your development.

To begin with, you will need to create a new database called “snippets” and a new table, also called “snippets”.

The Snippets Table

You can use the following structure for your table[1]:

```
CREATE  TABLE snippets (
  id INT NOT NULL AUTO_INCREMENT,
  collection INT NULL DEFAULT NULL,
  owner INT NULL DEFAULT NULL,
  title VARCHAR(48) NOT NULL,
  content TEXT NULL DEFAULT NULL,
  entered DATETIME NULL,
  modified DATETIME NULL,
  PRIMARY KEY (id)
) ENGINE = InnoDB;
```

The fields are as follows:

`id`	The auto-incremented primary key
`title`	A descriptive title
`content`	The text of the snippet
`entered`	The date the snippet was created
`modified`	The date the snippet was last update

The two fields, `collection` and `owner` will not be used at this stage, but will allow a future refinement involving multiple collections and contributors.

1 This SQL statement is for MySQL. For testing purposes, a file called `snippets.sqlite` is included, which allows you use the built-in SQLite database.

The Page

The `index.php` page contains mainly a single form with multiple submit buttons.

```
<form action="" method="post">
  <div id="titles">
    <select name="snippets" size="16">
      <option value="0">New Snippet</option>
      <!-- Snippet Titles -->
    </select>
    <button type="submit" name="select">Select Snippet</button>
  </div>
  <div id="editnote">
    <p><input type="text" name="id"
         value="<?php print $id; ?>" /></p>
    <p><label for="addtitle">Title:</label><br />
      <input type="text" name="title" id="title" class="text"
      value="..." />
    </p>
    <p><label for="content">Content:</label><br />
      <textarea name="content" id="content" cols="20" rows="12">
      ...</textarea>
    </p>
    <div id="control">
      <!-- if new -->
        <p>
         <button type="submit" name="insert">Add</button>
        </p>
      <!-- else -->
        <p>
         <button type="submit" name="update">Update</button>
         <button name="delete">Delete</button>
        </p>
      <!-- end if -->
    </div>
  </div>
</form>
```

The select element will contain a list of existing snippet titles to select one to be edited or deleted. Its first element (`New Snippet`) will be used to insert a new snippet.

The text box and text area will contain the data from an existing snippet for editing or reviewing.

The control area will contain either an insert button, if a new one is selected, or update and delete buttons if an existing one is selected.

Connecting to the Database

Create a file called `db.php`, and save it into your includes folder.

The connection script is a standard connection using PDO. We will use the variable `$pdo` in future scripts.

Enter the following[2]:

```
$database='snippets';
$user='snippets';       // or whatever
$password='password';   // or whatever
//$dsn="sqlite:includes/$database.sqlite";
$dsn="mysql:host=localhost;dbname=$database";
try {
  //$pdo = new PDO($dsn);
  $pdo = new PDO($dsn,$user,$password);
} catch(PDOException $e) {
  die ("Cannot connect to database $database");
}
```

At the top of your `index.php` file, include the following:

```
require_once('includes/db.php');
```

Adding a Record

The form contains a button named "insert". Currently the update and delete buttons are also visible, but later we will choose between them.

To test whether the form has been submitted via the insert button, test for its existence in the $_POST array.

```
require_once('includes/db.php');

if(isset($_POST['insert'])) {

}
```

We don't care what its actual value is, just whether it has been set.

First, we will read in the text fields. Just in case, we will also trim them, removing spaces at the beginning or end:

```
if(isset($_POST['insert'])) {
  $title=trim($_POST['title']);
  $content=trim($_POST['content']);
}
```

(At this point, we should check whether they have been filled, but we will deal with that later).

2 This code is for a connection to MYSQL. If you are using the sample `snippets.sqlite` database, you can use the commented code instead (the `$dsn` and `$pdo` assignment statements). Don't forget to comment out the MYSQL versions.

Because the data comes from the user, it should be treated with care. All outside data should be regarded as a potential SQL injection attack, but is easily managed using prepared statements.

The SQL for inserting the data is:

```
$sql='INSERT INTO snippets(title,content) VALUES(?,?);';
```

The question mark place holders will be replaced with data after the statement has been prepared, preventing the data from being misinterpreted as additional SQL.

We will now prepare the statement:

```
$pdoStatement=$pdo->prepare($sql);
```

Although you can bind the data separately to the statement at this point, it is much simpler to bind the data when you execute the prepared statement. This is done by passing an array of values:

```
$pdoStatement->execute(array($title,$content));
```

Remember that the data must be in an array (even if there is only one value), every question mark must have a corresponding value, and that the values must be in the correct order.

This will give us:

```
if(isset($_POST['insert'])) {
   $title=trim($_POST['title']);
   $content=trim($_POST['content']);
   $sql='INSERT INTO snippets(title,content) VALUES(?,?);';
   $pdoStatement=$pdo->prepare($sql);
   $pdoStatement->execute(array($title,$content));
}
```

For convenience, we will want a copy of the id of the new record. The id will be used for display later. However, since the id is and auto incremented value, we will need to get the value from the database. This is given by the `lastInsertId()` method of PDO:

```
$id=$pdo->lastInsertId();
```

The finished code reads:

```
if(isset($_POST['insert'])) {
  $title=trim($_POST['title']);
  $content=trim($_POST['content']);
  $sql='INSERT INTO snippets(title,content) VALUES(?,?);';
  $pdoStatement=$pdo->prepare($sql);
  $pdoStatement->execute(array($title,$content));
  $id=$pdo->lastInsertId();
}
```

Populating the List

Now that we have something in the table, we will want to see it.

We will create a variable called `$snippets`. This will contain the option elements to go into the select element in the form.

An option element has this form:

```
<option value="...">...</option>
```

The text content of the element is what the user sees. The value of the element is what will be posted. In our case, the text will be the title of the snippet, while the value will be the id.

To begin with, this array will be an empty array.

Towards the end of the PHP block (after the processing of the form data), add the following:

```
$snippets=array();
```

Now, we will read the table for a list of ids and titles.

```
$sql='SELECT id,title FROM snippets';
```

Since we are not working with user data, we can query the database without having to prepare a statement first:

```
$pdo->query($sql)
```

This will give us a result set (which is technically a `PDOStatement`), which we can assign to a variable:

```
$results=$pdo->query($sql);
```

The results can then be iterated like an array:

```
foreach($results as $row) {

}
```

More simply, you do not have to use an intervening variable, and you can iterate through the query result set directly:

```
foreach($pdo->query($sql) as $row) {

}
```

Each `$row` variable will be an array with the data. Although the array also contains numbered elements with the same data, we will be using the associative elements:

```
foreach($pdo->query($sql) as $row) {
  $snippets[]="<option value=\"$row[id]\" $selected>
    $row[title]</option>";
}
```

Since the associative elements are placed inside the string, *you do not quote the keys*, even though they are text keys!

The `$snippets[]=` notation means that a new value will be added to the end of the array. This is also known as pushing a value onto the array.

When the array is complete, we will convert into a string, since it is this string which must be printed into the HTML.

The `implode()` function (also known as `join()`) takes two parameters: the glue (what comes between the values), and the array.

```
$snippets=implode('',$array);
```

In this case, we want nothing between the values (in HTML the only thing permissible between option elements is space).

Note that by re-using the `$snippets` variable, we have replace an array with a string. However, we are still using the variable for the same purpose.

The completed code should look like this:

```
$snippets=array();
$sql='SELECT id,title FROM snippets';
foreach($pdo->query($sql) as $row) {
  $selected= $id==$row['id'] ? ' selected="selected"' : '';
  $snippets[]="<option value=\"$row[id]\" $selected>$row[title]</option>";
}
$snippets=implode('',$snippets);
```

Displaying an Existing Snippet

Although part of the same form as the rest, the select element has a button closely associated with it.

Later we will include some JavaScript which will replace this button with an action. This is not required, especially if JavaScript has been turned off, so the JavaScript will only be applied if available.

We will put the code before the other input handling code, so we can use the results in later code if we need.

The select element is named "snippets". To check whether one has been selected, we will check the post array. If one has been selected, we will set the id:

```
require_once(...);

if(isset($_POST['snippets'])) {
  $id=...;
}
```

Remember that the first option, New Snippet, has a value of 0. Since the snippet ids start at 1, this will (later) be use to indicate that we want a new record.

As a zero, 0, may also have been selected, we really want to test whether the value is non-zero, rather than whether it has been set:

```
if($_POST['snippets']) ...
```

However, if no value has been selected at all, the above would generate an error, so we add the error suppression operator (`@`):

```
if(@$_POST['snippets']) {
  $id=...;
}
```

The error suppression operator ignores the potential error, and can be used in cases where "no value" is an acceptable alternative.

The above could be interpreted as "if snippets, where it exists, is non-zero, then ...".

If none has been selected, then we will set the `$id` to 0:

```
if(@$_POST['snippets']) {

}
else $id=0;
```

Although the data should have come from a form, and we have constructed this data ourselves, it is still best convert the value to an integer:

```
if(@$_POST['snippets']) {
  $id=intval($_POST['snippets']);
}
```

The `intval()` function will convert as much as possible to an integer, stopping at the first invalid character. If there is no valid character, the result will be 0.

The SQL statement will select the title and content from the snippets table matching the id. For good measure, we will also count the results.

Since the id is unique, you will never get more than one row, and a corresponding count of 0. An invalid id would result in an empty row. However, if you include a count, you will get one row again, with a count of 0, and NULL for the remaining fields. This will simplify testing.

```
if(isset($_POST['snippets'])) {
  $id=intval($_POST['snippets']);
  $sql="SELECT count(*), title, content FROM snippets WHERE id=$id;";
}
```

It is perfectly safe to include this user data into this SQL statement, since the `intval()` function has converted it into an integer, making it incapable of being interpreted as additional SQL.

Since the SQL is safe, we can run the query directly:

```
$pdo->query($sql)
```

Since there will only be exactly one row, we can fetch it immediately:

```
$pdo->query($sql)->fetch();
```

The result, as usual, will be an array of values. This time, it is more convenient to use the numbered keys, since we can use them in a list():

```
list($count,$title,$content)=$pdo->query($sql)->fetch();
```

The `list()` operator (this is not technically a function, though it looks like one) will copy the numbered elements of an array into the listed variables. It is a convenient way of working with multiple variables.

In the event of an invalid id, we will have a `$count` of `0`, and `NULL` for the `$title` and `$content`, which we will deal with presently.

If the count is 0, we will set the id to 0, which will also indicate a new entry:

```
if(!$count) $id=0;
```

This will give us:

```
if(isset($_POST['snippets'])) {
  $id=intval($_POST['snippets']);
  $sql="SELECT count(*), title, content FROM snippets WHERE id=$id;";
  list($count,$title,$content)=$pdo->query($sql)->fetch();
  if(!$count) $id=0;
}
```

If a valid snippet has not been selected, and hence the id is 0, we will set the title and content to empty strings:

```
else $id=0;
if(!$id) $title=$content='';
```

Note that is possible to assign the empty string to both variables in one statement.

Displaying the Content on the Form

Having read the title and content, or set them to empty strings, we will display them on the form.

For a text box, an input element, you display the old or default value in the value attribute:

```
<input type="text" name="..." value="..." />
```

In this case we will insert a PHP print statement of the variable. To be safe we will also include the error suppression operator, in case the value was not set:

```
<input type="text" name="title" value="<?php print @$title; ?>" />
```

For a text area, the old or default value is between the opening and closing tags:

```
<textarea name="...">...</textarea>
```

In this case we will use:

```
<textarea name="content"><?php print @$content; ?></textarea>
```

Finally, though this will not be displayed, we will need to store the id. This will be used when later we update or delete the record, and so the record will need to be identified.

To store a value in a form without user intervention, we use a hidden field. This field is not displayed on the page (though it is still visible in the page source).

```
<input type="hidden" name="..." value="..." />
```

Generally you would hard code the value, since the user is unable to enter the value otherwise. A JavaScript might also be used to set the value.

In our case, we will set the hidden field for the id:

```
<input type="hidden" name="id" value="<?php print @$id; ?>" />
```

The Submit Buttons

Although a form may have many submit buttons, you should still hide the buttons which are not relevant, or which might interfere with you application.

In this case, we will display an insert button if the record is a new record, or the update and delete buttons if the record is an existing record.

The simplest way to distinguish between a new record and an existing record is with the id. A value of `0` (zero) will indicate a new record, while a non-0 id will indicate an existing record.

We have a `div` (`control`) which contains the various buttons.

PHP allows you to intersperse PHP blocks with ordinary HTML (or other text). Remember, that what is not inside a PHP block is regarded as PHP output.

A useful structure is:

```
<?php if(...) { ?>
  <!-- PlanA -->
<?php } else { ?>
  <!-- PlanB -->
<?php } ?>
```

If the condition is true, then the `PlanA` text will be output. Otherwise the `PlanB` text will be output.

For this project, we will test whether the id is 0 (new record). If so, we will display the insert button. Otherwise we will display the update and delete buttons.

Enter the following:

```
<?php if(...) { ?>
  <!-- PlanA -->
<?php } else { ?>
  <!-- PlanB -->
<?php } ?>
```

Replace the condition with:

```
<?php if(!$id) { ?>
```

This will be true if `$id` is not something, that is, it is zero.

Replace `PlanA` with the insert button, and `PlanB` with the update and delete buttons.

The code should read:

```
<div id="control">
  <?php if(!$id) { ?>
    <p><button type="submit" name="insert">Add</button></p>
  <?php } else { ?>
    <p><button type="submit" name="update">Update</button>
      <button name="delete">Delete</button></p>
  <?php } ?>
</div>
```

Updating Records

Now that we can add and display records, we will look at changing them.

The key to updating is to have an id to identify the record being updated. This id comes from the selected snippet earlier, copied into the hidden id field.

To check whether a record is being updated, we check its submit button. This is logically placed after the test for insert, thought it doesn't have to be:

```
if(isset($_POST['insert'])) {
  ...
}
elseif(isset($_POST['update'])) {

}
```

Note that as you can only press one submit button, you do not need the `elseif` clause - you could simply have used another `if`. The `elseif` is only required when both conditions might be true, but you only want one to be executed. However, the use of `elseif` here helps to group the tests together.

Just as with the insert script, we will read in the title and content from the POST array. In addition, we will also read in the id, which comes from the hidden field previously set. This value will be processed through `intval()`:

```
elseif(isset($_POST['update'])) {
  $id=intval($_POST['id']);
  $title=trim($_POST['title']);
  $content=trim($_POST['content']);
```

```
}
```

As before, we should really check whether the title and content have been supplied.

To update a record, we use the SQL UPDATE command.

```
UPDATE ... SET ...=..., ...=...;
```

The UPDATE command is potentially disastrous to a database, since it will change all of the records unless you qualify it with a WHERE clause.

```
UPDATE ... SET ...=..., ...=... WHERE ...=...;
```

In this case we will set new values for the title and the content, and use the WHERE clause to select for the record's id:

```
UPDATE snippets SET title=...,content=... WHERE id=...;
```

Since we will be adding user data, we will need to prepare the statement first, and execute it with data later. The values will be set with question mark place holders:

```
elseif(isset($_POST['update'])) {
  …
  $sql='UPDATE snippets SET title=?,content=? WHERE id=?;';
  $pdoStatement=$pdo->prepare($sql);

}
```

Finally, we execute the command with data:

```
elseif(isset($_POST['update'])) {
  …

  $pdoStatement->execute(array($title,$content,$id));
}
```

The finished script should look like this:

```
elseif(isset($_POST['update'])) {
  $id=intval($_POST['id']);
  $title=trim($_POST['title']);
  $content=trim($_POST['content']);
  $sql='UPDATE snippets SET title=?,content=? WHERE id=?;';
  $pdoStatement=$pdo->prepare($sql) or die('oops');
  $pdoStatement->execute(array($title,$content,$id));
}
```

Deleting Records

Now we can delete records using the SQL DELETE command. Although this command is also potentially disastrous for the same reasons, some would argue that it is better to lose your data than to have bad data.

We will add another `elseif` block underneath the others. As before, the `elseif` is not strictly necessary, since we can only press one submit button.

```
elseif(isset($_POST['delete'])) {

}
```

Again, as before, we will extract the id of the record:

```
elseif(isset($_POST['delete'])) {
  $id=intval($_POST['id']);
}
```

To delete a record, we use the SQL DELETE command:

```
DELETE FROM ...   DELETE FROM ... WHERE ...
```

The first form will delete all records. The second will delete only selected records.

Since the id has been processed through `intval()`, it cannot contain any harmful SQL. This means it is safe to include it directly into the SQL string, and to execute it directly:

```
$sql="DELETE FROM snippets WHERE id=$id;";
$pdo->exec($sql);
```

Note that unlike SELECT statements, we use the `exec()` method of PDO, not the query() method.

Finally, we reset the id, title and content:

```
$id=0;
$title=$content='';
```

The final code should look like this:

```
elseif(isset($_POST['delete'])) {
  $id=intval($_POST['id']);
  $sql="DELETE FROM snippets WHERE id=$id;";
  $pdo->exec($sql);
  $id=0;
  $title=$content='';
}
```

Finishing Off

The project will now work and can be used for storing your code snippets or other useful notes. However there are a few features which might make your project more useful.

Data Validation

When we submit or update a snippet, you will need to check whether you have a title, and depending on your policy, content (you may wish to allow empty content if you plan on filling it later).

To begin with, we will have a variable called $error, and set it to an empty string. This variable is best set just before the code which checks the insert and update operations:

```
$error='';
if(isset($_POST['insert'])) {
}
elseif(isset($_POST['update'])) {
}
```

After the `title` and `content` variables have been set in the `insert` code, you will need to test whether they both have a non-empty value. The if structure should embrace the remaining code:

```
if(isset($_POST['insert'])) {
  $title=trim($_POST['title']);
  $content=trim($_POST['content']);
  if($title && $content) {
    $sql='INSERT INTO snippets(title,content) VALUES(?,?);';
    $pdoStatement=$pdo->prepare($sql) or die('oops');
    $pdoStatement->execute(array($title,$content));
    $id=$pdo->lastInsertId();
  }
}
```

Note that a convenient shorthand to test whether a variable has an empty value is simply to test the variable itself. A non-empty string evaluates as `true`, while an empty string evaluates as `false`.

Note also the use of the `and` operator (`&&`).

If either the title or content are empty, the else clause will assign an error message:

```
if($title && $content) {

}
else $error='<p class="error">Please complete
  the Title & Contents</p>';
```

The same needs to be done for the `update` code.

Finally, the error message needs to be displayed. In the form, just below the paragraph with the hidden form element, add the following PHP block:

```
<p><input type="hidden" name="id" value="<?php print $id; ?>" /></p>
<?php print $error; ?>
```

Since the $error variable will have been set to an empty string or to an error paragraph, the print statement will safely print the results.

Highlighting the Current Record

When you select a snippet title and submit it, the newly refreshed page will not highlight the selected title in the list.

To highlight a selected option in HTML, you can use the selected attribute:

```
<option value="..." selected="selected">...</option>
```

When we generate the list of options, we will insert the selected attribute when (or if) the id of the current item matches the id selected with the previous submit. One simple way of inserting an occasional attribute is to define a variable, and set it using the conditional (ternary) operator:

```
$selected = test ? planA : planB;
```

The conditional operator uses either the first or the second value, depending on whether the test evaluates as true.

The `id` selected will be in the variable `$i`, which will be `0` if no item has been selected. We will compare this against the current row's id:

```
$id==$row['id']
```

If true, we will use the string `' selected="selected"'`; otherwise we will use the empty string `''`. Note the space at the beginning of the first string. This makes it easier to insert into the HTML which requires a space between attributes.

```
$selected = $id==$row['id'] ? ' selected="selected"' : '';
```

It remains only to insert this string into the code for the option:

```
$snippets[]="<option value=\"$row[id]\" $selected>$row[title]</option>";
```

The resulting code should look like this:

```
foreach($pdo->query($sql) as $row) {
  $selected= $id==$row['id'] ? ' selected="selected"' : '';
  $snippets[]="<option value=\"$row[id]\" $selected>$row[title]</option>";
}
```

Auto Submit

Currently, the snippets list comes with a submit button. Using JavaScript, you can have the snippets list auto-submit. The following JavaScript will enable auto-submit, as well as hide the submit button:

```
<script type="text/javascript">
  window.onload=function() {
    document.forms[0].select.style.visibility='hidden';
    document.forms[0].snippets.onchange=function() {
      document.forms[0].submit();
    };
  }
</script>
```