



By icarus

This article copyright [Melonfire](#) 2000–2002. All rights reserved.

Table of Contents

<u>Kicking The Tyres</u>	1
<u>Making The Connection</u>	2
<u>Adapting To The Environment</u>	3
<u>The Magic Of The Movies</u>	5
<u>Visiting The Box Office</u>	7
<u>Adding Things Up</u>	9
<u>Submitting To The King</u>	12
<u>Erasing The Past</u>	14
<u>Of Methods And Madness</u>	16
<u>An Object Lesson</u>	21

Kicking The Tyres

As a recent convert to Zope from PHP, I was curious: could Zope really give me all the capabilities I was used to in PHP?

I decided to find out, by using one of my most common activities with PHP as the benchmark: connecting to and retrieving records from a MySQL database. As it turned out, Zope manages this quite well, and it's also nowhere near as complicated as you might think. And so, over the course of this article, I'm going to show you how to hook your Zope server up to a MySQL database, and write DTML code that allows you to retrieve and manipulate MySQL table records using standard SQL commands. If this sounds interesting, keep reading.



Making The Connection

First up, that all-important question: why? Why on earth would you want to connect your Zope system to an external database, especially since Zope already comes with its own, very cool ZODB?

There are a couple of reasons why you might want to do this:

First, if your data is already in an existing RDBMS, you're usually going to find it tedious and time-consuming to migrate it all to Zope's own database. It's far easier (not to mention less disruptive to your users) to leave your data where it is, and simply get Zope to talk to your existing RDBMS so that you can access the data within it in a simple, transparent manner.

Second, Zope's own database is not really meant to perform the same tasks as a full-fledged RDBMS. The ZODB is a very powerful object database which keeps track of all the different objects you use in Zope, and it comes with some very neat transaction and versioning features. However, it's optimized to read data, not write it, and so INSERT and UPDATE queries tend to be sub-optimal on this database.

For a more detailed discussion of why you might prefer an SQL database over the ZODB, take a look at [http://www.zope.org/Members/anthony/sql vs ZODB](http://www.zope.org/Members/anthony/sql_vs_ZODB)

It should be noted at this stage that, in addition to the ZODB, Zope does come with a small, SQL-compliant RDBMS named Gadfly, which can be used if your requirements aren't too complex. If, however, you're dealing with large record sets or complex queries, you will probably find Gadfly too slow and inefficient for your needs, and you will need to explore the possibility of connecting Zope to a more professional and full-featured RDBMS.

Adapting To The Environment

Normally, setting up a connection between Zope and an external RDBMS like MySQL, Oracle, PostgreSQL or even Gadfly requires two things: a Database Connection object, which is used to perform communication between Zope and the specified database, and one or more Z SQL methods, which are used to execute queries on the database.

Zope comes with adapters for a variety of different databases, including ODBC, Oracle, MySQL and Interbase. I'll be connecting Zope to a MySQL database in this article; however, the process is similar for other database systems.

Hooking Zope up to MySQL is a little tedious, especially if you attempt to do it solely on the basis of the information available online. This is because much of the online documentation relating to Zope–MySQL connectivity is outdated and no longer relevant, and you can lose a whole day just trying out different variants of the techniques suggested. So let me make it as simple as possible.

Getting Zope and MySQL to talk nice to each other requires you to install the following two pieces of software:

1. The Zope MySQL Database Adapter (ZMySQLDA), available from
<http://www.zope.org/Members/adustman/Products/ZMySQLDA>
2. The Python MySQL Database Interface (MySQLdb), a Python MySQL driver available from
<http://www.zope.org/Members/adustman/Products/MySQLdb>

You can also find this software at <http://sourceforge.net/projects/mysql-python>

The installation process is fairly simple.

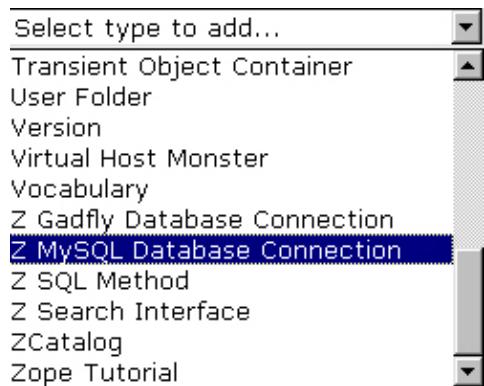
1. Install (copy) the ZMySQLDA files into Zope's Products directory (usually <Zope>/lib/python/Products).
2. Extract the MySQLdb files to the same place. Configure and compile the driver (instructions are available in the source archive) using the Python binary that shipped with your version of Zope. This is extremely important, because if you don't use the exact binary that ships with your Zope installation, Zope won't know how to handle the driver.
3. Install the compiled driver into your Zope build (again, instructions to accomplish this are available in the driver's documentation).
4. Restart Zope.

This process works on Linux; however, despite my best efforts, I was unable to get Zope and MySQL talking on Windows. It's certainly possible, though, and the process is about the same – take a look at http://zope.org/Members/philh/ZMySQLDA_html for some hints.

Assuming all goes well, you should now see a new Z MySQL Database Connection object type in the Type drop-down list within the Zope management interface.



Zope And MySQL



If you don't see this item, it usually means that something bad happened during the installation process. Drop by http://www.zope.org/Members/alanpog/zmysqlda_steps and see what you did wrong.



The Magic Of The Movies

With Zope and MySQL now talking to each other, it's time to start actually doing something with them. Here's the MySQL table I'll be using throughout this article – it stores information on my DVD collection.

```
CREATE TABLE dvd (
    id int(10) unsigned DEFAULT '0' NOT NULL,
    title varchar(255) NOT NULL,
    cast varchar(255) NOT NULL,
    director varchar(255) NOT NULL,
    genre varchar(255) NOT NULL
);

INSERT INTO dvd VALUES ( '1', 'Don\'t Say A Word', 'Michael Douglas',
    Sean Bean and Brittany Murphy', 'Gary Fleder', 'Suspense');

INSERT INTO dvd VALUES ( '2', 'Captain Corelli\'s Mandolin',
    'Penelope Cruz and Nicolas Cage', 'John Madde', 'Romance');
```

Now, the second step in getting Zope to communicate with this database involves creating an instance of the Z MySQL Database Connection object. You can create one of these via Zope's management interface; when you do, you'll see something like this:

Add Z MySQL Database Connection

Id	MySQL_database_connection
Title	Z MySQL Database Connection
Enter a Database Connection String ¹	
Connect immediately	<input checked="" type="checkbox"/>
Add	

¹ Connection Strings
The connection string used for Z MySQL Database Connection is of the form:
[+/-] database[@host[:port]] [user [password [unix_socket]]]
or typically:
database user password

At this point, it's necessary to specify a database connection string, so that Zope can open a connection to the database. This string is usually of the form:

```
database[@host[:port]] [user [password [unix_socket]]]
```

Zope And MySQL

So, if I wanted to set up a connection to the database "mystuff" for user "john" with password "secret", my connection string would look like this:

```
mystuff john secret
```

Once you've created the Connection object, you can test it via the Test function that appears within the Zope management interface. This function allows you to execute any SQL query on the database, and returns the result to you instantly. Try executing the query

```
SELECT * FROM dvd
```

and see what happens. If your screen fills up with data, it means that your database connection is working just fine.

Once a Database Connection object has been created, it's possible to instantiate Z SQL Methods to execute queries via this connection (I'll be doing this over the next few pages).

Visiting The Box Office

Now, if you've worked with databases before, you will be aware that there are four basic things you can do with them:

1. Select records from a table
2. Add records to a table
3. Update existing records in a table
4. Delete records from a table

In Zope, each of these functions can be implemented via Z SQL Methods. Each Z SQL Method stores one or more SQL queries, and these queries can be executed simply by calling the method in your DTML scripts.

Let's see how this works by creating a Z SQL Method to list all the DVDs in my collection.

Go back to the Zope management interface and create a new Z SQL Method from the drop-down list. I named my method "selectAllMethod"; feel free to name yours whatever you like. Remember to associate the Z SQL Method with an appropriate Database Connection object from the drop-down list; most often, this will be the Z MySQL Database Connection you just created on the previous page.

Finally, the meat. Input the following SQL query into the query template:

```
SELECT * FROM dvd
```

This query template represents the query (or set of queries) to be executed when the "selectAllMethod" object is called. As you will see over the next few pages, it is possible to dynamically construct this query template on the basis of environment or form variables.

Once the query template has been saved, you can test it via its Test function. When you do this, Zope will execute the newly-minted method and return the results to you for verification. This is an easy way to verify that your Z SQL Method works the way it is supposed to, and I would recommend that you do it every time you create a new one.

Right. Moving on, let's now see how this Z SQL Method can be used with DTML. Create a new DTML Method – I've called mine "list" – containing the following code:

```
<dtml-var standard_html_header>

<h2>My DVD Collection</h2>
<ul>
<dtml-in selectAllMethod>
<li><dtml-var title>
```

Zope And MySQL

```
<br>
<font size="-1"><a href="edit?id=<dtml-var id>">edit</a> | <a href="delete?id=<dtml-var id>">delete</a></font>
<p>
</dtml-in>
</ul>

<a href="add">Add a new DVD</a>

<dtml-var standard_html_footer>
```

As you can see, this is extremely simple. All I'm doing is invoking the "selectAllMethod" Z SQL Method created above and then using a DTML loop (via <dtml-in>) to iterate through the result set. This is identical to the procedure you would use in Perl or PHP to retrieve the results of a database query.

Don't worry too much about the "edit", "add" and "delete" links in the list above – all will be explained shortly. For the moment, just study the manner in which the DTML Method invokes the Z SQL Method and iterates through the returned data set. Notice also that the fields of the returned record set are automatically converted to DTML variables, and can be accessed in the normal way, via the <dtml-var> construct.

Here's what the output looks like:

My DVD Collection

- Don't Say A Word
[edit](#) | [delete](#)
- Captain Corelli's Mandolin
[edit](#) | [delete](#)
- Moulin Rouge
[edit](#) | [delete](#)
- Big Fat Liar
[edit](#) | [delete](#)

[Add a new DVD](#)

Adding Things Up

Next, how about adding a new record to the table?

Here too, the process is similar – first create a Z SQL Method containing an INSERT query, and then invoke the Method from a DTML script. There's one important difference here, though – since the data to be entered is not hard-wired, but rather user-defined, the query string must be constructed dynamically.

In order to see how this works, create a new Z SQL Method – I've called mine "insertMethod" – and add the following values to the argument field:

```
title director cast genre
```

These arguments can then be used to dynamically construct a query. Here's what I would put into the query template:

```
INSERT INTO dvd (title, director, cast, genre) values
(<dtml-sqlvar
title type="string">, <dtml-sqlvar director type="string">,
<dtml-sqlvar cast type="string">, <dtml-sqlvar genre
type="string">)
```

I'm sure the linkage between the arguments and the query template is now clear – the arguments passed to the Z SQL Method can be accessed in the query template to dynamically create a new INSERT query every time the method is invoked.

In case you're wondering, the <dtml-sqlvar> construct is very similar to the <dtml-var> construct, except that it comes with a couple of additional, SQL-specific features – for example, the "type" attribute, which makes it possible to specify the data type of the values being inserted and thereby catch errors before the data gets inserted into the database, and the ability to automatically "quote" values before inserting them into the database.

So that takes care of the SQL. Now, how about a form to invoke this method and pass arguments to it?

```
<dtml-var standard_html_header>

<h2>Add DVD</h2>

<form action="someFormProcessor" method="POST">





```

Zope And MySQL

```
<td><input name="title" width=30 value=""></td>
</tr>

<tr>
<td>Director</td>
<td><input name="director" width=30 value=""></td>
</tr>

<tr>
<td>Cast</td>
<td><input name="cast" width=30 value=""></td>
</tr>

<tr>
<td>Genre</td>
<td><input name="genre" width=30 value=""></td>
</tr>

<tr>
<td colspan=2 align=center>
<input type="submit" name="submit" value="Add DVD">
</td>
</tr>

</table>
</form>

<dtml-var standard_html_footer>
```

Here's what it looks like:

Add DVD

Title	<input type="text"/>
Director	<input type="text"/>
Cast	<input type="text"/>
Genre	<input type="text"/>
<input type="button" value="Add DVD"/>	

When this form is submitted, the form processor will need to invoke the "insertMethod" Z SQL Method, use the form variables to dynamically fill up the query template and execute the INSERT query. Here's what it looks like:

Zope And MySQL

```
<dtml-var standard_html_header>

<dtml-call insertMethod>

<h2>Item added!</h2>

<p>

<a href="list">View the entire collection</a> or <a
href="add">add
another title</a>

<dtml-var standard_html_footer>
```

Fairly simple, this – the `<dtml-call>` construct is used to invoke the selected Z SQL Method, and a neat little result screen is displayed. The form variables submitted to the form processor are automatically picked up by the Z SQL Method and used to construct the INSERT query string.

Here's what it looks like:

Item added!

[View the entire collection](#) or [add another title](#)



Submitting To The King

If you're familiar with DTML, you'll know that the two objects above can be combined into a single one by creative use of an "if" conditional test keyed to the form's "submit" variable. And that's exactly what I've done next – a DTML Method named "add" which includes both the initial form and the form processor.

```
<dtml-var standard_html_header>

<dtml-if submit>

<dtml-call insertMethod>

<h2>Item added!</h2>

<p>

<a href="list">View the entire collection</a> or <a
href="add">add
another title</a>

<dtml-else>

<h2>Add DVD</h2>

<form action="add" method="POST">
<table border=0>

<tr>
<td>Title</td>
<td><input name="title" width=30 value=""></td>
</tr>

<tr>
<td>Director</td>
<td><input name="director" width=30 value=""></td>
</tr>

<tr>
<td>Cast</td>
<td><input name="cast" width=30 value=""></td>
</tr>

<tr>
<td>Genre</td>
<td><input name="genre" width=30 value=""></td>
</tr>
```

Zope And MySQL

```
<tr>
<td colspan=2 align=center>
<input type="submit" name="submit" value="Add DVD">
</td>
</tr>

</table>
</form>

</dtml-if>

<dtml-var standard_html_footer>
```

Using this two-in-one technique can reduce the number of objects in your collection, and perhaps make things easier on the eyes.

Erasing The Past

So that takes care of adding records. Now, how about deleting them?

You'll remember, from the "list" DTML Method created at the beginning of this exercise, that every item in the DVD list has links pointing to "edit" and "delete" objects, and that these objects are passed the record ID of the corresponding item.

```
<font size="-1"><a href="edit?id=<dtml-var id>">edit</a> | <a href="delete?id=<dtml-var id>">delete</a></font>
```

So, in other words, the "delete" object – another DTML Method – needs simply to use this ID to execute a DELETE query on the table. Let's take a look at the code for this DTML Method:

```
<dtml-var standard_html_header>

<dtml-call deleteMethod>

<h2>Item deleted!</h2>

<p>

<a href="list">View the entire collection</a> or <a href="add">add another title</a>

<dtml-var standard_html_footer>
```

No biggie. Like the "add" DTML Method before it, this one too simply functions as a wrapper for a Z SQL Method, which actually does all the work. In this case, the Z SQL Method is named "deleteMethod" and it receives a record ID as argument. Let's see what that looks like:

```
DELETE FROM dvd WHERE id = <dtml-sqlvar id type="int">
```

The "deleteMethod" Z SQL Method contains a simple DELETE query, with the record ID passed to it inserted dynamically into the query template.

Here's what the result page looks like:

Item deleted!

[View the entire collection](#) or [add another title](#)



Of Methods And Madness

The final item on the agenda involves building an interface and methods to edit existing records in the database table. Again, I'll hark back to the "list" object created right at the beginning. You'll remember that this "list" object also contained a link to an "edit" object, which was passed a record ID (in much the same way as the "delete" object discussed on the previous page).

```
<font size="-1"><a href="edit?id=<dtml-var id>">edit</a> | <a href="delete?id=<dtml-var id>">delete</a></font>
```

Let's look at this "edit" object in detail:

```
<dtml-var standard_html_header>

<dtml-if submit>

<dtml-call updateMethod>

<h2>Item edited!</h2>

<p>

<a href="list">View the entire collection</a> or <a href="add">add another title</a>

<dtml-else>

<h2>Edit DVD</h2>

<dtml-in selectOneMethod>

<form action="edit" method="POST">
<table border=0>

<tr>
<td>Title</td>
<td><input name="title" width=30 value="
```



Zope And MySQL

```
<tr>
<td>Cast</td>
<td><input name="cast" width=30 value=<dtml-var
name="cast">"></td>
</tr>

<tr>
<td>Genre</td>
<td><input name="genre" width=30 value=<dtml-var
name="genre">"></td>
</tr>

<tr>
<td colspan=2 align=center>
<input type="hidden" name="id" value=<dtml-var name="id">>
<input
type="submit" name="submit" value="Edit DVD"> </td> </tr>

</table>
</form>
</dtml-in>

</dtml-if>

<dtml-var standard_html_footer>
```

Again, this consists of both a form and a form processor, separated from each other by a DTML "if" test and the "submit" variable. Let's look at the form first:

```
<dtml-if submit>

// snip

<dtml-else>

<h2>Edit DVD</h2>

<dtml-in selectOneMethod>

<form action="edit" method="POST">
<table border=0>

<tr>
<td>Title</td>
<td><input name="title" width=30 value=<dtml-var
name="title">"></td>
```

Zope And MySQL

```
</tr>

<tr>
<td>Director</td>
<td><input name="director" width=30 value=<dtml-var
name="director">"></td> </tr>

<tr>
<td>Cast</td>
<td><input name="cast" width=30 value=<dtml-var
name="cast">"></td>
</tr>

<tr>
<td>Genre</td>
<td><input name="genre" width=30 value=<dtml-var
name="genre">"></td>
</tr>

<tr>
<td colspan=2 align=center>
<input type="hidden" name="id" value=<dtml-var name="id">>
<input
type="submit" name="submit" value="Edit DVD"> </td> </tr>

</table>
</form>
</dtml-in>

</dtml-if>
```

The first thing this form does is call the Z SQL Method named "selectOneMethod". This Z SQL Method is similar to the "selectAllMethod" discussed a few pages back, except that it includes an additional modifier – a record ID – in order to return a single record from the table rather than a list of all available records. Here's what it looks like:

```
SELECT * FROM dvd WHERE id = <dtml-sqlvar id type="int">
```

The record ID, obviously, gets passed to this method as an argument from the "list" object.

After the Z SQL method has been invoked, the fields in the resulting record set are converted into DTML variables and used in the form, via the <dtml-var> construct, to pre-fill the various form fields. Here's what it looks like:

Edit DVD

Title	<input type="text" value="Don't Say A Word"/>
Director	<input type="text" value="Gary Fleder"/>
Cast	<input type="text" value="Michael Douglas, Sean E"/>
Genre	<input type="text" value="Suspense"/>
<input type="button" value="Edit DVD"/>	

Notice, from the DTML code above, that the record ID is again passed forward to the form processor via a hidden field in the form.

```
<input type="hidden" name="id" value=<dtml-var name="id">>
```

Once this form is submitted, the form processor takes over.

```
<dtml-if submit>

<dtml-call updateMethod>

<h2>Item edited!</h2>

<p>

<a href="list">View the entire collection</a> or <a
href="add">add
another title</a>

<dtml-else>

// snip

</dtml-if>
```

As you can see, the form processor invokes a Z SQL Method, "updateMethod", to update the database with the new information for the selected record. Here's what "updateMethod" looks like:

```
UPDATE dvd SET title=<dtml-sqlvar title type="string">,
director=<dtml-sqlvar director type="string">,
cast=<dtml-sqlvar cast
type="string">, genre=<dtml-sqlvar genre type="string"> WHERE
```

Zope And MySQL

```
id =  
<dtml-sqlvar id type="int">
```

As with the original "insertMethod", the UPDATE query above is dynamically constructed on the basis of form input variables.

Once the Z SQL Method is successfully executed, the resulting output looks like this:

Item edited!

[View the entire collection](#) or [add another title](#)

And you're done! You now have a Zope interface to add, edit and delete records from a MySQL database. Wasn't all that hard, was it?



An Object Lesson

If you've been paying attention, the process of communicating with MySQL through Zope should be fairly clear. Assuming a working database connection, there are two basic things you need: Z SQL Methods to execute queries (these Z SQL Methods can be passed arguments so that queries can be constructed dynamically, as demonstrated in this article) and DTML Methods to invoke the Z SQL Methods (and pass them arguments, where required).

If you're used to PHP or Perl, this hard separation between methods and their invocation may be somewhat difficult to grasp at first. However, if you persist, you'll soon find that Zope's object-based approach to SQL is actually preferable to the looser approach in those languages. By encapsulating specific SQL functionality into objects, Zope immediately allows reuse of those objects – and, by implication, their specific functionality – across a Web site; this, in turn, makes it possible to package complex queries into a single object and invoke it wherever required, as many times as required.

One of the most obvious advantages of this approach becomes visible when you need to perform a series of related queries – for example, insert a record into table A, retrieve the ID of the inserted record, use that ID as a foreign key when inserting a record into Table B, and so on – multiple times. Zope makes it possible to create a generic Z SQL Method that accepts a series of arguments and performs – internally – as many queries as are needed to achieve the desired result. A user of this Z SQL Method can invoke it transparently, as many times as required, blissfully unaware of the activities that take place within it. Take it one step further: if changes are required, they can take place within the Z SQL Method, requiring no changes at all to the DTML Methods that invoke the object; this obviously makes maintenance easier.

Anyway, that's about all I have time for. I hope you found this article interesting, and that it gave you some insight into how Zope can be connected to external database systems. In case you need more information, these links should offer you a starting point:

A discussion of the ZODB versus a regular RDBMS, at
http://www.zope.org/Members/anthony/sql_vs_ZODB

Zope–MySQL software, at <http://sourceforge.net/projects/mysql-python>

Zope–MySQL installation instructions, at http://www.zope.org/Members/alanpog/zmysqlda_steps

Form processing with Zope, at http://www.devshed.com/Server_Side/Zope/ZopeForm

The official Zope Web site, at <http://www.zope.org/>

Zope documentation, at <http://www.zope.org/Documentation>

Till next time...stay healthy! Note: All examples in this article have been tested on Linux/i586 with Zope 2.5. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!