February 4, 2000

Z SQL Methods User’s Guide

Document Revision 2.1.0, February 4, 2000
Modified by Pam Crosby
For Zope version 2.1.1

Copyright © Digital Creations

Table of Contents

Introduction

Audience: Developers

Z SQL Methods provide access to relational and external databases from *Zope*. *Zope* applications are built on object oriented databases. All Zope data are stored in this object database. Database queries and commands can be used to publish relational data, collect and store data in relational databases, and create complex web-deployed database applications.

Z SQL Methods provide high performance by maintaining open database connections, so it is not necessary to connect to a database on each request.

Z SQL Methods utilize features of *Zope*, to enable complex applications that would be harder to create in other environments.

Purpose of The Z SQL Methods User's Guide

This guide was created to provide an overview of the attributes of ZSQL Methods. The format of the guide provides step by step instructions on how to publish relational data on the Web. With Zope examples and explanations, the user creates SQL queries using DTML.

- The Zope object database can be queried like relational databases using DTML commands.
- You can query existing relational databases within Zope to provide web based content management.
- Relational database queries results are first-class Zope objects.
- The Z SQL methods support the integration of relational data with the Zope object system.
- Database Connections explain how to manage and create connections to external databases.
- Instructions to create search interfaces between Zope and database queries are explored.

This guide will take you through creating, testing and debugging, query templates, and editing of SQL database methods. SQL Methods support a number of specialized tags for inserting values or comparisons into SQL source, an explanation of these various tags are provided in this guide.

Getting Started

Publishing relational data on the Web with *ZSQL Methods* typically requires three steps:

- Step One: Establish a database connection,
- Step Two: Create one or more database ZSQL methods, and
- Step Three: Create one or more search interfaces.

Establish a database connection

A database connection can provide several things; an abstraction of a database - relational or external and an interface to a relational or other external database.

To create a database connection:

- Pick a database to use
- Determine identifying information needed by the database,
- Decide where in *Zope* the connection will be used, and
- Add a database connection to the desired *Zope Folder*.

For example, *ZAcme, Inc*. has a parts database that they wish to make available for searching. They have a Gadfly[1] database that they wish to access within the Product area site. The *ZAcme* Product manager, Stan, adds a *Z Gadfly Database Connection*. The form is displayed in Figure 1 where Stan can enter the *id*, *title*, and select a *data source* for the database connection.

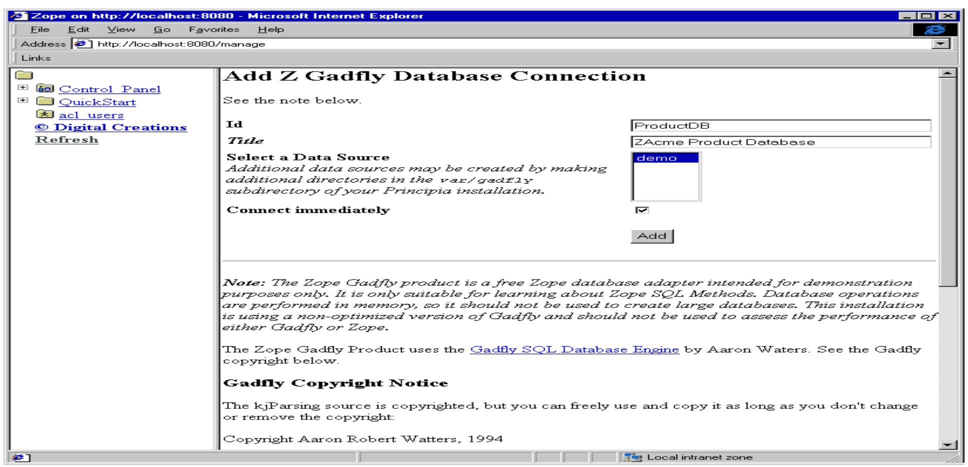

Figure 1. An input form used to add a database connection

1 The Zope Gadfly product is a free Zope database adapter intended for demonstration purposes only. It is only suitable for learning about Zope SQL Methods. Database operations are performed in memory, so it should not be used to create large databases.

Create a database method

The database method provides the mechanism for executing database queries and other commands. It also makes the results available to *Zope*.

Stan starts with a small products database consisting of the two styles of widgets *ZAcme* creates and the database, *ProductDB* using the connection. Stan needs to place some values into the database. To accomplish this task, Stan will add a *SQL Method*.

Add a *SQL Method* by selecting it from the *Available Objects* menu.

1. Name the *SQL Method* id `sqlCreateTables`.
2. The *Connection Id* will have the `ProductDB` highlighted.
3. In the *Arguments* field, enter `add product_number, product_description, product_price` and `ship_weight.`
4. In the *Query template*, make the `product_number` and `product_description` string type. The `product_price` and `ship_weight` are float type. Figure 2 shows the SQL Method after adding the method.

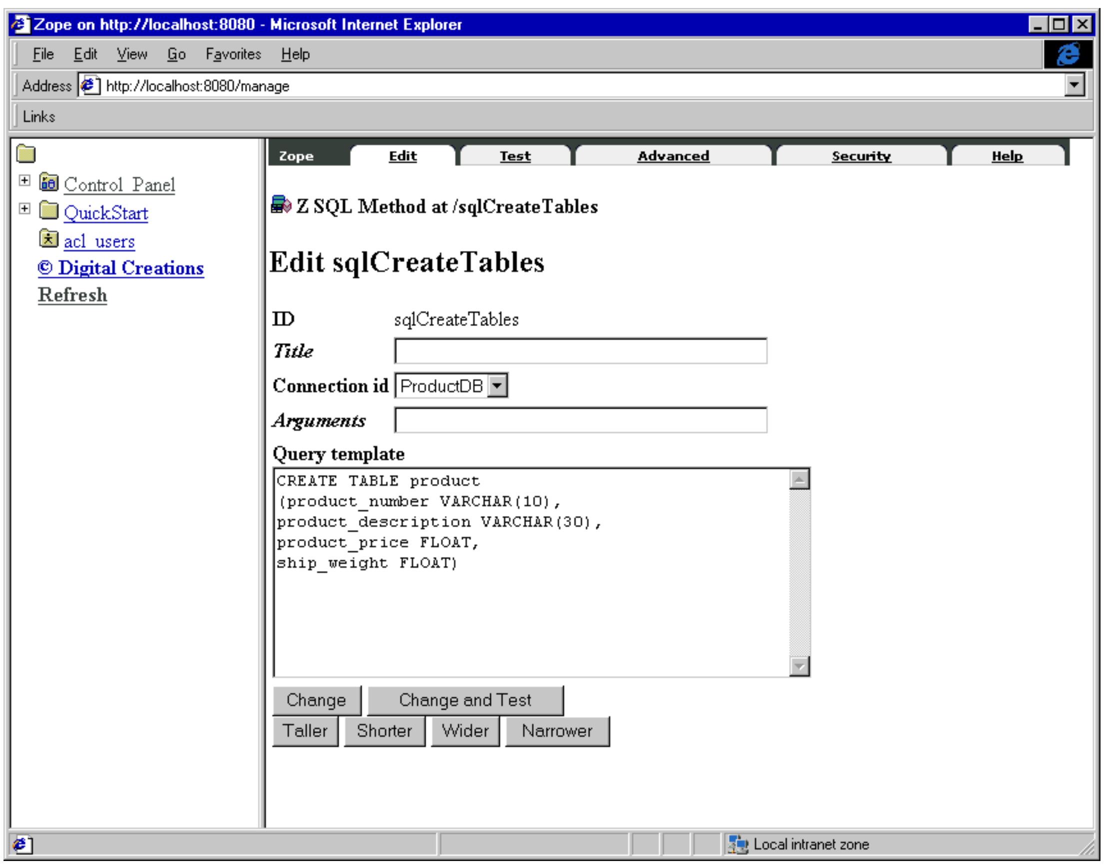

Figure 2. Created Table for Product Database

The SQL used to perform the query is specified as a *query template*. Query templates use the same *Document Template Markup Language* (*DTML*) used to create *Zope's DTML Documents* and *DTML Methods*. Query templates provide specialized tags for use in generating SQL.

Stan had created a product table and tested the method by pressing *Add and Test*. The screen shows that this method is not a query and the table creation code is returned. If there had been an error, a message would be printed on the top of the screen as shown in Figure 3.

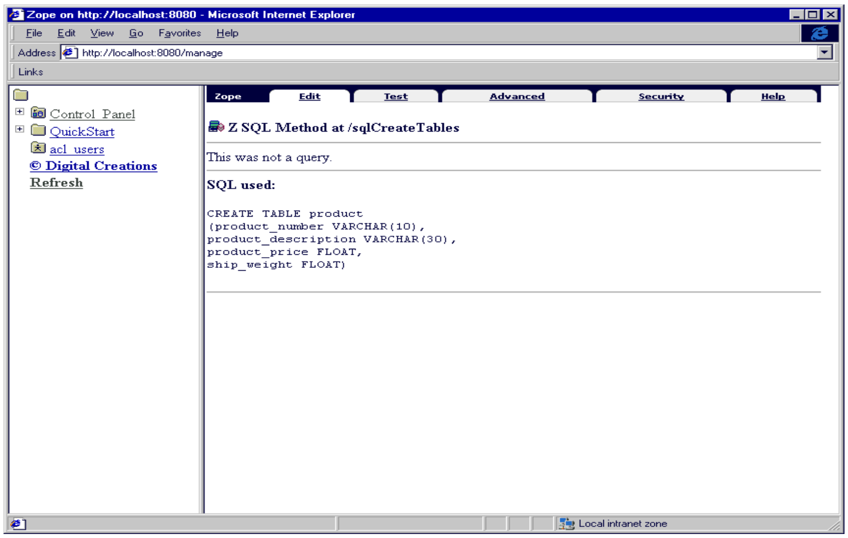

Figure 3. SQL Method Test

Data Entry

The next step is to create a data entry method for the table. Stan creates another *SQL Method* to insert products into the `product` table.

1. Enter `sqlInsertProduct` into the *id* field.
2. Add `Insert Products` for the *Title.*
3. In the *Arguments* field add `product_number ship_weight product_price product_description.`
4. In the *Query template* area add the SQL code shown in Figure 4.

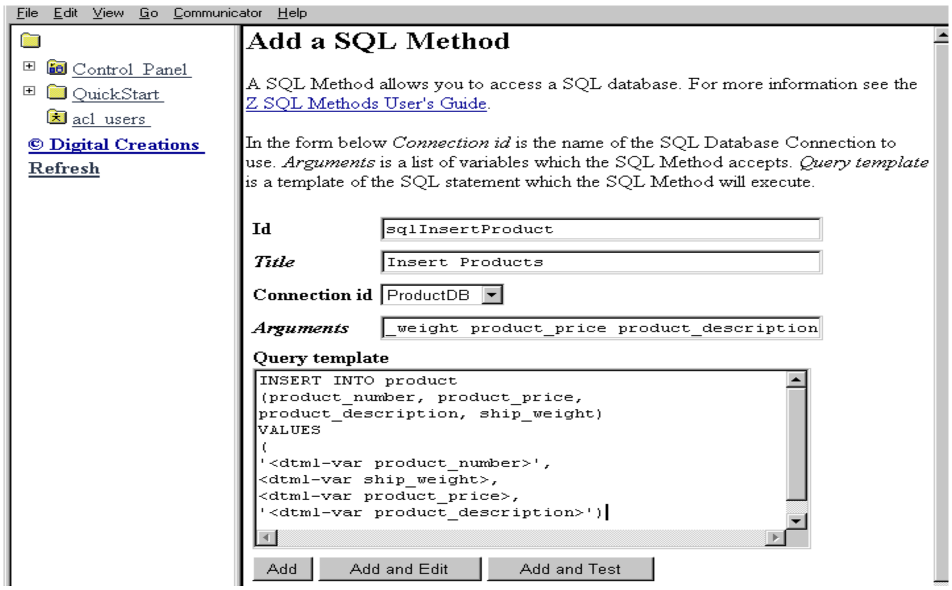

Figure 4. Insert Values SQL Method

After entering the information, Stan selects *Add and Test* to check for errors. The test view of the SQL method gives the fields with a form for entering the data. Stan enters information about the Historical Widgets and submits his query. .

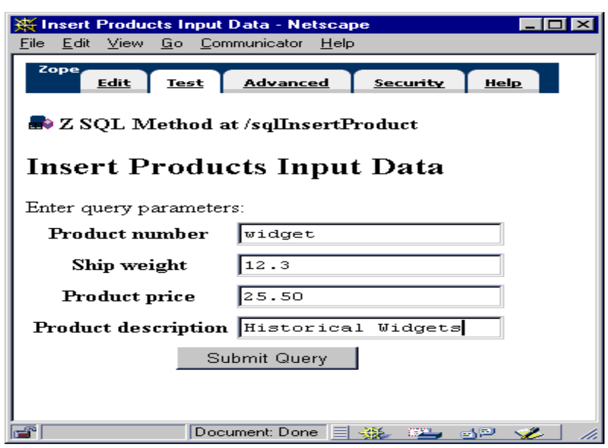

Figure 5. Insert Values Form from sqlInsertProduct

Data Verification

The next step in this procedure is to verify the values entered in the database. When Stan submits the query, the screen shows the SQL used and the entered data, Figure 6. Stan enters the Millennium Widgets and Products Manual data in the same way using the *sqlInsertProduct* method.

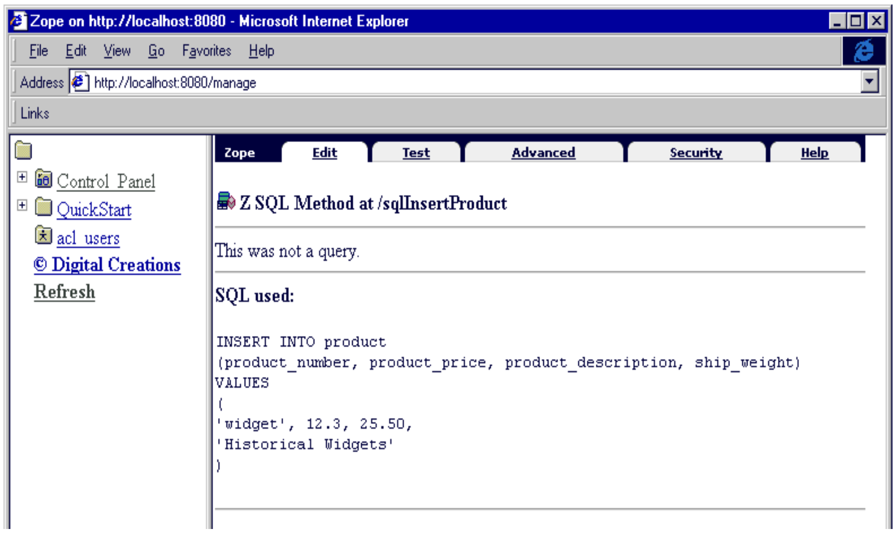

Figure 6. Insert Values Result

Data Retrieval

To retrieve data from the `products` database, Stan will have to create an query.

An *dtml-sqltest* tag is used to insert an SQL test that compares the column, `product_number` to a value given as input in the argument `product_number`. Stan adds another SQL Method named `lookup_product.`

1. Enter the `lookup_product` into the *id* field.
2. Add `Lookup a product given a product number` for the *title.*
3. The *argument* field only needs the `product_number.`
4. The *Query template* has the *sqltest* to allow comparisons. (Figure 7)

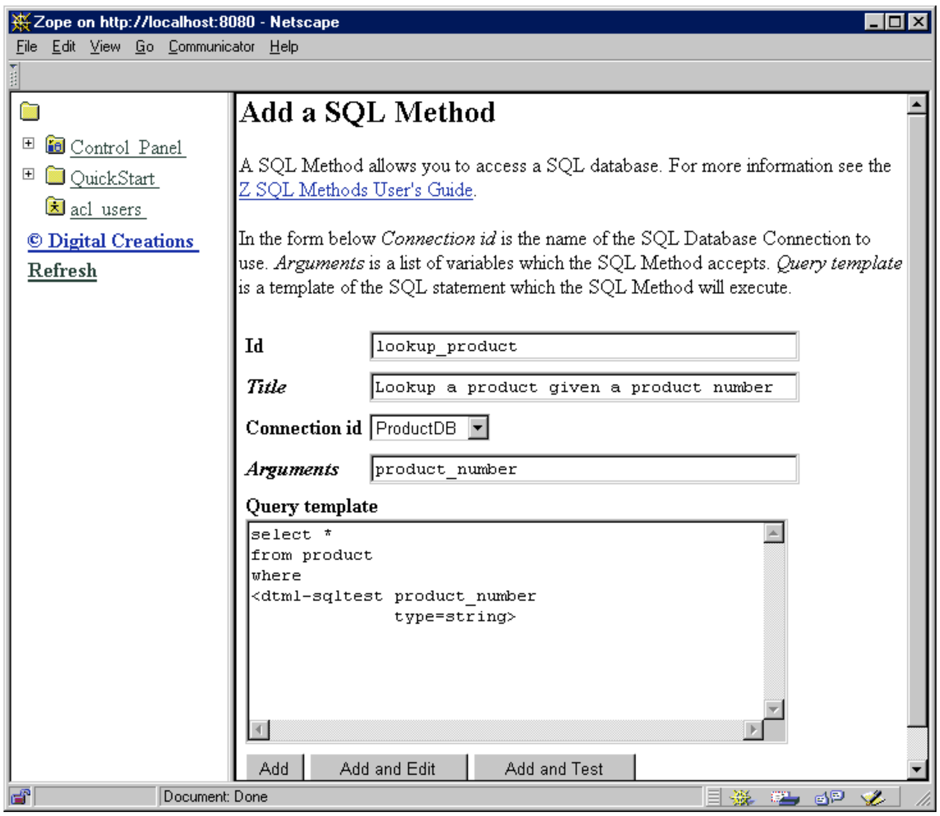

Figure 7. Add SQL method for Product Query

Stan adds and tests the SQL method by pressing the *Add and Test* option. An input form is displayed, as shown in Figure 8. Product number `widget` is added to the form and submitted.

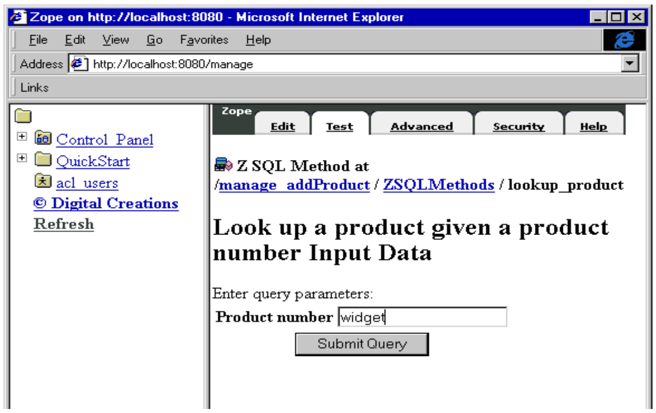

Figure 8. Lookup Product test form

When the query is submitted, the output is displayed along with the SQL that was generated as shown in figure 9.

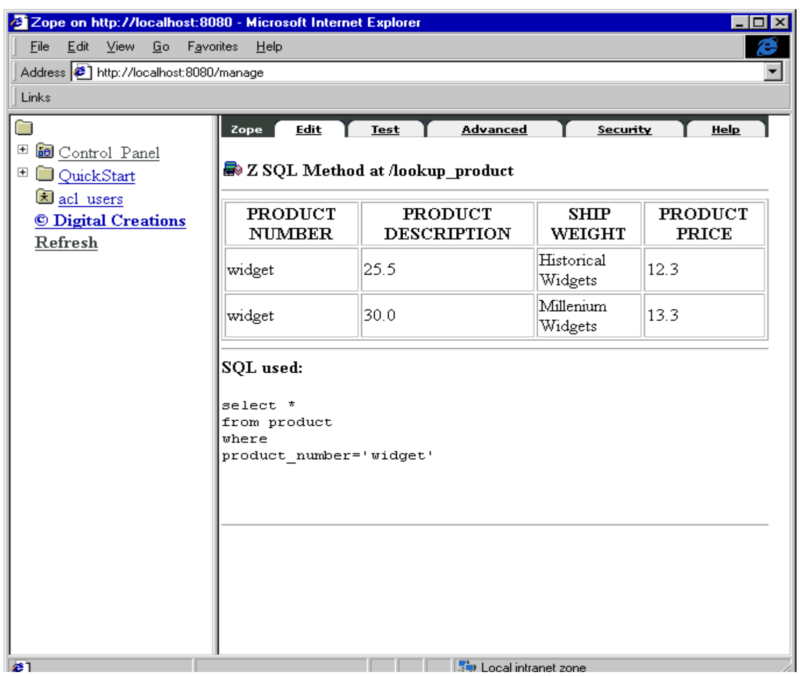

Figure 9. Test Output

After a database method has been created, it must be tested to:

- Make sure that the SQL statement performs the desired query or action, and
- To allow Zope to collect information about results needed for building a search interface.

A database method can be tested when it is created by clicking on the “Add and Test” button on the add form or at any time by selecting the test view. After a database method has been created and tested, a search interface can be created to publish the data on the web. This leads to exploring Search Interfaces in the next section.

Creating Search Interfaces

Database methods provide an interface between *Zope* and database queries. To publish data on the Web, *Zope DTML Documents* must be created if necessary, to collect input data, and to display reports. Input forms and reports can be created from scratch, but is easier to use the *Z Search Interface Creation Wizard* to automatically create search interfaces. The search interface wizard collects basic information, such as the name of a database method and names of input and report documents to be created and then writes new documents based on database schema information. After the documents have been created with the search interface wizard, they can be customized.

Figure 10 shows the *Search Interface Creation Wizard* being used to create a search interface for the `lookup_product` database method. The first option in the form is a list of searchable objects found in the current folder, or in folders above the current folder. Searchable objects are objects, like *ZSQL Database Methods*, that support searching for information[2]. You can select one or more searchable objects to be included in the search interface. If you select multiple searchable objects, the generated input form will collect information needed for all objects, and the generated report will show results for each object, in sequence.

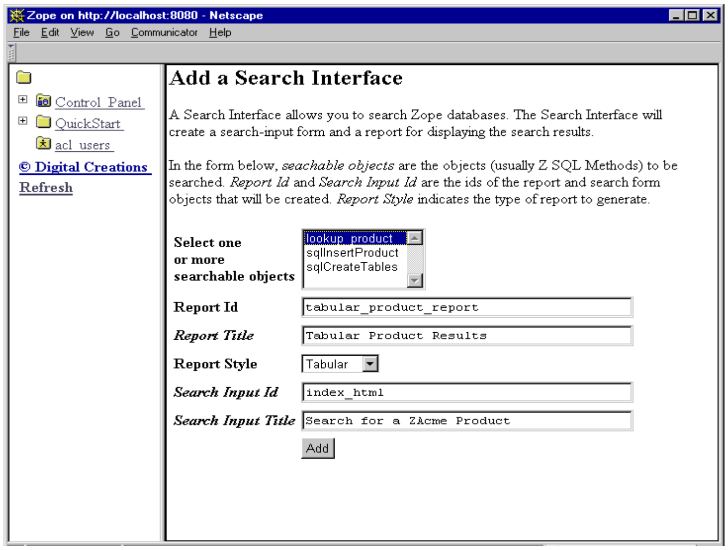

Figure 10. Search interface creation wizard

2 A number of Zope-based products implement the searchable object interface. Other searchable objects that are available for Zope include Tabula Collections, and Zope Network Clients.

Figure 11 shows the contents of the `Product` folder after the *New Search Interface* has been submitted. The contents list includes two new documents, `index_html`, and `tabular_product_report`. By naming the input form `index_html`, we have created a "home page" for the `Product` area that performs a product search. For example, if someone visited the URL `http://zacme.com/ZAcme/Product`, they would see the search input form shown in Figure 12.

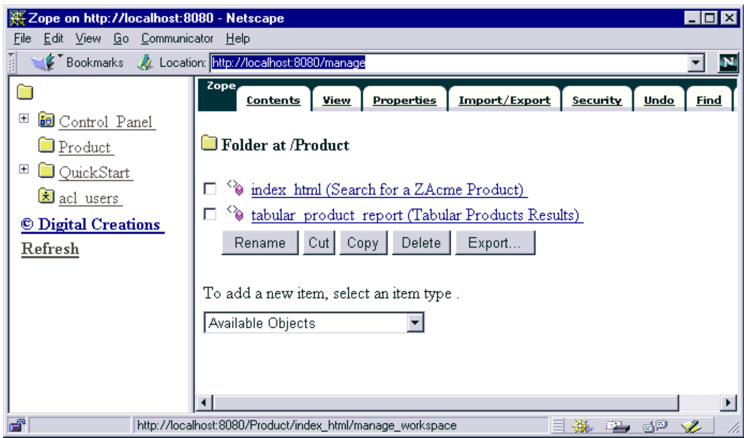

Figure 11. Contents view of the Products folder after adding a tabular search interface.

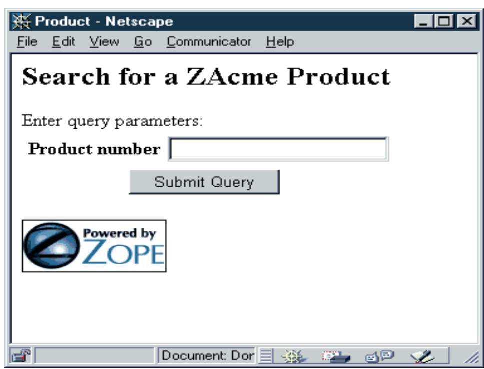

Figure 12. An automatically generated input form for a tabular product search

The source of the generated input form is shown screen capture in figure 13. The input form is a *Zope* DTML *Method* object.

There are a number of features of the input form that should be noted:

- *DTML var* and *if* tags are used to construct a “Cancel” button if the *HTTP Referer* header is supplied in the request. The DTML Method view shows the DTML script created with the Search interface.
- Stan wants to add the CANCEL button. He modifies the script.
- By adding the following lines after <tr><td colspan=2 align=center>

```
<dtml-if HTTP_REFERER>
<input type=”SUBMIT” name=”SUBMIT” value=”Cancel”>
<INPUT NAME=”CANCEL_ACTION” TYPE=”HIDDEN”
    VALUE=”<dtml-var HTTP_REFERER>”>
</dtml-if HTTP_REFERER>
```

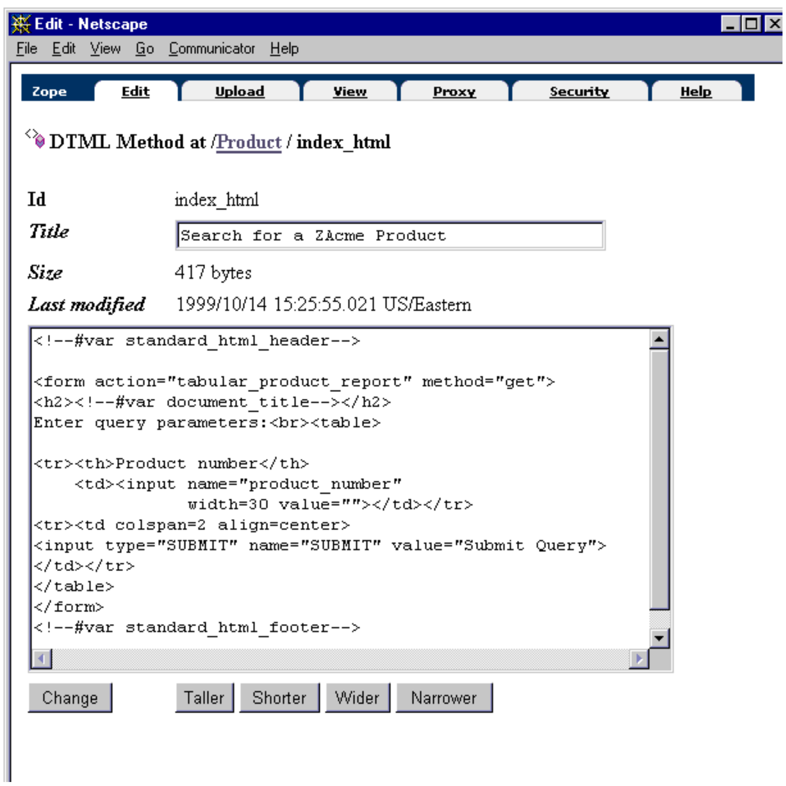

Figure 13. Index_html view

- The variables `standard_html_header` and `standard_html_footer` are used to provide standard look-and-feel components in the form.
- For example, we might want to define a logo for the ZAcme Products division to be included at the top of all documents. We add a *Zope Image* object, named `logo`, to the `Product` folder, and we add a *Zope DTML Document* named `standard_html_header` that includes a reference to the `logo` (Figure 14). Inserting an *Image* object with

a *dtml-var* tag causes an HTML *IMG* tag to be inserted referring to the image and including an *ALT* attribute with the title defined for the *Image* object. After these changes, the input form is displayed as shown in Figure 15.

```
<HTML>
<HEAD><TITLE>
<dtml-var title_or_id></TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF">
<dtml-var logo><br>
```

Figure 14. A `standard_html_header` document that includes a logo

When Stan submits the input form, the form has the ZAcme Logo across the top. Figure 16 shows a the output of the `tabular_product_report`.

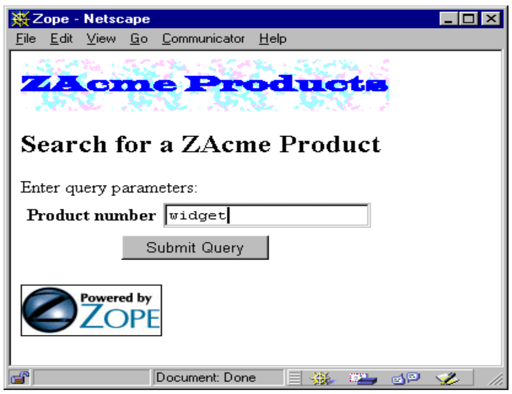

Figure 15. The input form for a tabular product search, after adding a standard_html_header and a logo to the Computer folder.

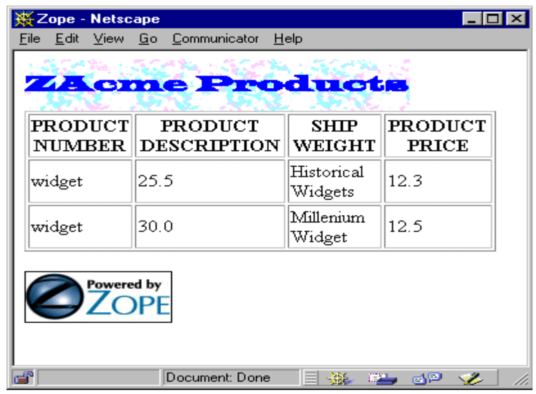

Figure 16. Output of a search using the `tabular_product_report` created by the search interface wizard.

The source of `tabular_product_report` is shown in figures 17 -19. It is useful to walk through the generated document template source. The generated source illustrates a number important document template concepts and features of database methods. The section “Interactive Insertion, the *dtml-in* Tag” in the ***Document Template Markup Language Reference*** provides a detailed description of document template *dtml-in* tags, including batch processing.

```
<dtml-var standard_html_header>
<dtml-in lookup_part size=50 start=query_start>
   <dtml-if sequence-start>

      <dtml-if previous-sequence>

        <a href="<dtml-var URL><dtml-var sequence-query>
                 query_start=<dtml-var
                 previous-sequence-start-number>">
        (Previous <dtml-var previous-sequence-size> results)
        </a>

      </dtml-if previous-sequence>

      <table border>
        <tr>
          <th>PRODUCT NUMBER</th>
          <th>SHIP WEIGHT</th>
          <th>PRODUCT PRICE</th>
          <th>PRODUCT DESCRIPTION</th>
        </tr>

   </dtml-if sequence-start>
```

Figure 17. Part one of the source of the `tabular_product_report` document that was generated by the search-interface creation wizard.

The source begins with the insertion of `standard_html_header`, as usual. An *dtml-in* tag follows the insertion of the standard header. The *dtml-in* tag is used to iterate over the results of the call method, `lookup_product`. The database method receives its parameters from the incoming HTTP request, so parameters need not be passed explicitly[3]. The *dtml-in* tag uses the size and start attributes to request batch processing support[4]. This causes a number of special variables to be defined, which are used later in the *dtml-in* tag source. These include `previous-sequence`, `sequence-query`, `previous-sequence-start-number` and `previous-sequence-size`.

3 Parameters can be passed explicitly using an *expr* attribute in the *in* tag. Parameters are passed using Python “keyword” parameter syntax. For example, to call the lookup_product method with a product_number of ‘widget’, the following *dtml-in* tag would be used:

```
<dtml-in expr=”lookup_product(product_number=’widget’)”
size=50 start=query_start>
<!--#in expr=lookup_product(product_number=’widget’)
size=50 start=query_start-->
```
(Zope 1.1x version depreciated)

4 See “Batch Processing” in the Document Template Markup Reference.

Immediately after the *dtml-in* tag is a *dtml-if* tag that inserts text when the variable `sequence-start` is true, or, in other words, when displaying the first item in a batch of items. The inserted text includes a link to previous batches of data and a table header showing the data column names. The link to previous batches is inserted only if there are previous batches, as indicated by the `previous-batches` variable. The link uses the *dtml-in* tag variables `sequence-query`, `previous-sequence-start-number`, and `previous-sequence-size`.

After the table header is inserted, the actual data for each item in the batch is inserted (figure 18).

```
<tr>
  <td><dtml-var product_number></td>
  <td><dtml-var ship_weight></td>
  <td><dtml-var product_price></td>
  <td><dtml-var product_number></td>
</tr>
```

Figure 18. Part two of the tabular_product_report Document, showing the DTML text to insert data items.

The last part of the text following the *dtml-in* tag is text to insert a table closing tag and to show a link to additional batches of data, if any (Figure 19). Two *dtml-if* tags are used to insert the text only when displaying the last item in a batch and to only include the link to following batches. Finally, the source ends with an *dtml-else* tag that inserts text if there are no results for the given inputs.

```
<dtml-if sequence-end>

      </table>      <dtml-if next-sequence>
         <a href="<dtml-var URL>
         <dtml-var sequence-query>
            query_start=<dtml-var
            next-sequence-start-number>">
         (Next <dtml-var next-sequence-size> results)
         </a>
      </dtml-if next-sequence>
   </dtml-if sequence-end>

<dtml-else>

  There was no data matching this <dtml-var title_or_id>
query.
```

Figure 19. Part three of the tabular_product_report document, showing source to display following batches and text to be inserted when there are no results.

After the search-interface creation wizard has been used to create report and input-form documents, the documents can be edited to meet specific needs.

Object-Relational Data Integration

ZSQL Methods support the integration of relational data with the Zope object system. Results of relational database queries are not just data. Rather, results of relational database queries are objects, which may have methods and can acquire information and behavior from the Zope environment. ZSQL Methods that use SQL select statements provide virtual collections of objects and can support direct object addressing through URLs.

The relational data object model

Data from relational database queries are returned as sequences of Python objects. Because each query result is a sequence, the *dtml-in* tag and the Python *for* statement may be used to iterate over results. Each element in a result sequence is a Python object that encapsulates a single record of a result table. The Python objects that encapsulates result records are called record objects.

Record objects provide access to result data by column name. Result columns are available as both attributes and as mapping keys of record objects. This allows columns to be accessed with simple *dtml-var* tags inside *dtml-in* tags when iterating over query results. For example, consider a Z SQL Method named `customers` that returns columns `CUSTOMER_ID, NAME`, and `PLANET`. From DTML, the customer names can be accessed with:

```
<dtml-in customers>
  <dtml-var NAME>
</dtml-in>
```

and from Python, the customer names can be accessed with:

```
for customer in customers():
    print customer.NAME
```

or

```
for customer in customers():
    print customer['NAME']
```

Column data can also be accessed using integer column numbers. Each record object is a sequence of column values. For example, the data for a record can be output without use of column names in DTML:

```
<table>
<dtml-in customers>
  <tr>
     <dtml-in sequence-item>
        <td><dtml-var sequence-item></td>
     </dtml-in>
  </tr>
</dtml-in>
</table>
```

and in Python:

```
for customer in customers():
  for data in customer:
     print data
```

Defining result classes

The *Advanced* view on database methods provides the ability to supply Python classes from which result objects can inherit methods. This facility allows rich behavior to be provided using the Python programming language. Individual database results are instances of subclasses and have data attributes from the relational database.

In the future, it will be possible to define ZClasses to provide behavior to database results.

Acquiring Data and Behavior

Database results can acquire data and methods from their environment. Methods, such as DTML Documents, External Methods and other SQL Methods that are defined in the folder containing an SQL method can be applied directly to objects returned from SQL queries.

Object Access

There are two ways to access objects through ZSQL Methods. Objects can be accessed by iterating over the results of calling a ZSQL Methods as demonstrated in the chapter before. The second way is to directly access object through its URL.

For database methods that return individual records, objects may be accessed directly through URL traversal. Consider the `lookup_part` query in figure 8. This method takes a single argument, a product number, and returns the corresponding part. A URL can be used to access a specific part by adding the input argument name, `product_number` and a specific product name to the URL for the database method. For example, to look up product number `widget`, a URL like:

```
http://zacme.com/ZAcme/Product/lookup_part/product_number/widget
```

might be used. Normally, record objects don't provide a default interface, so it will be necessary to invoke a method of the object, as in:

```
http://zacme.com/ZAcme/Product/lookup_part/product_number/widget/orders
```

This example uses the method `orders` on a product number. The `orders` method was not created in previous section. This method could be defined in a class specified in the ZSQL Method Advanced view, or it could be a method that is acquired from the Zope environment. In the Advanced view, select the "*allow transversal*" option for the URL access to function properly. Figure 20 shows the *transversal* option as well as the place for the class definition. More information is available in the **SQL Database Methods** chapter.

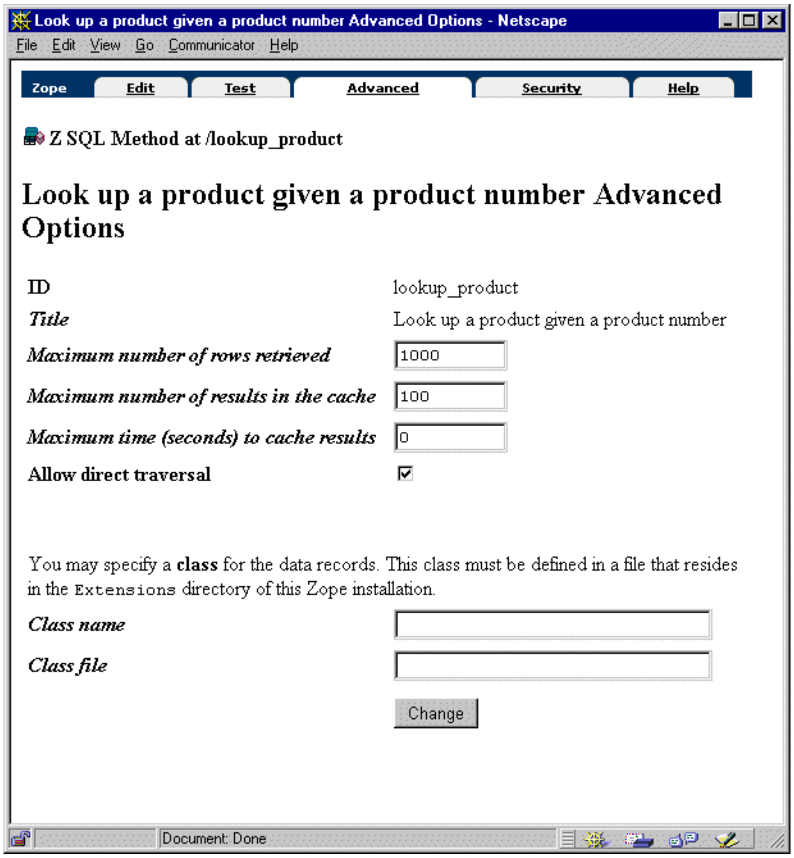

Figure 20. Advanced Options for SQL Method

Database Connections

Overview

Database Connections are used to establish and manage connections to external databases, such as relational databases. *Database Connections* must be established before database methods can be defined. Moreover, every Z SQL Method must be associated with a *Database Connection*.

Database Connections provide management interfaces for connecting to and disconnecting from the external database. Some database connections provide interfaces for browsing database schema information. Database connections are provided in Zope database adapter products. Database adapters are available for a number of databases, including ODBC, Solid, Oracle, MySQL and Gadfly.

The information needed to connect to a database depends on the specific database being used. Some database adapters provide database connection creation interfaces that let you select from a known set of databases, while others require you to enter a connection string.

Database connections are established when a database connection is created and later whenever a database connection is used. Database connections are automatically closed after a period of disuse and reopened when necessary.

Creating Database Connections

To create a database connection, display the *Contents* view of a *Folder*, and select "*Z Gadfly Database Connection*" from the available object list. The database connection add form will be displayed Figure 21. Enter the *id*, *title*, and information identifying the database, such as a data source name or connection string. Normally, a connection to the database is established immediately. To delay connecting to the database until later, select the *Connect immediately* check box.

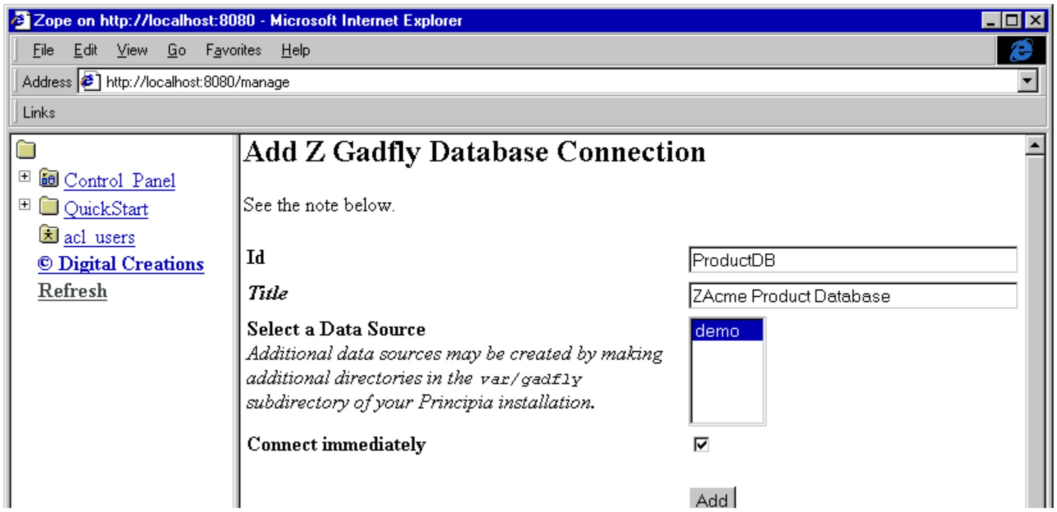

Figure 21. Add Connections Form

Connection Status

The *Status* view shows whether the database connection is open or not, and provides a button to open the connection if it is closed, or to close it if it is open (Figure 22).

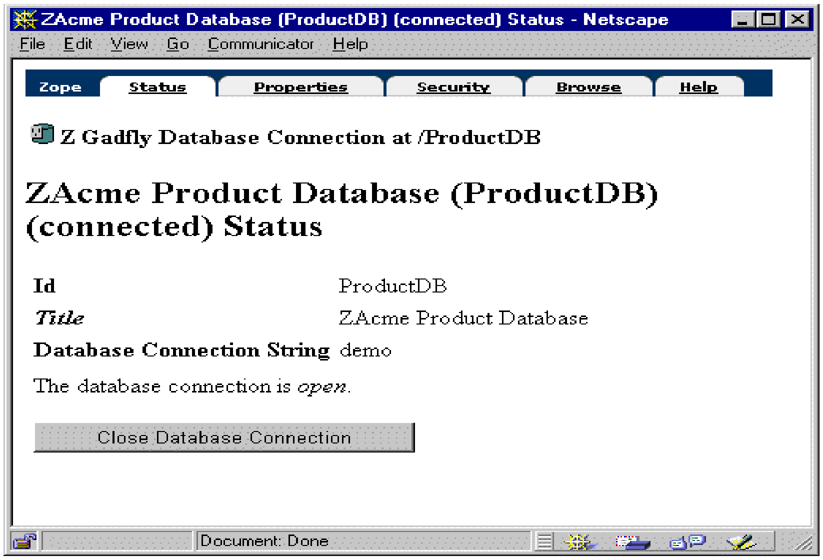

Figure 22. A database-connection status view

Connection Properties

The *Properties* view (Figure 23) provides a place to change the database connection *title* and other database-specific connection information.

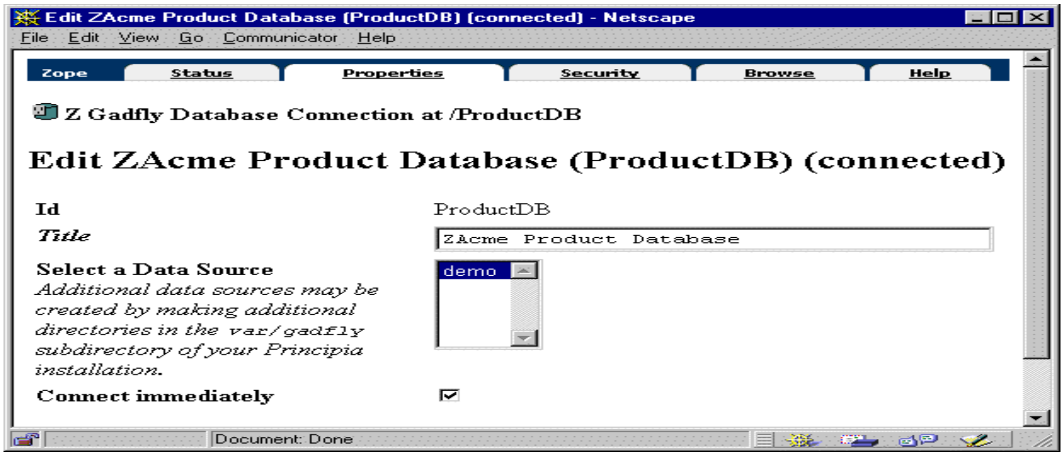

Figure 23. Database-connection properties view

Connection Browse

Some databases provide a ***Browse*** view for browsing the tables defined in a database. Database tables are listed and individual table listings may be expanded to show table schemas as shown in Figure 24.

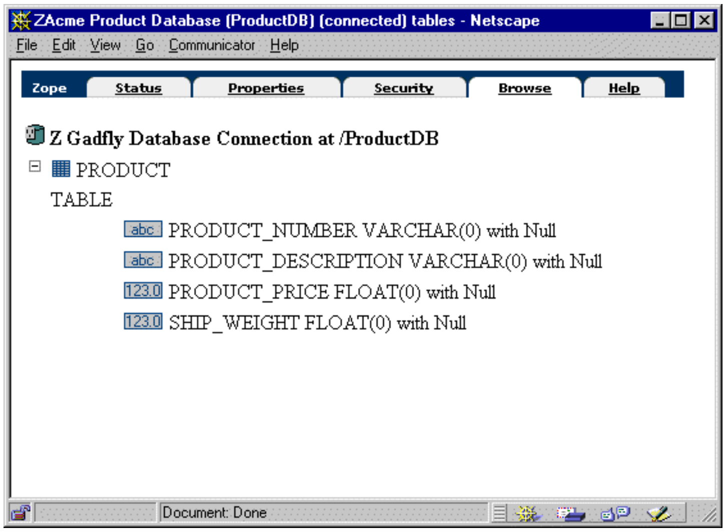

Figure 24. Database Connection Browse

Connection Security

As with all objects in Zope, you can specify security options for the database connection. These options can be given to individual user or groups. The options available are shown in Figure 25.

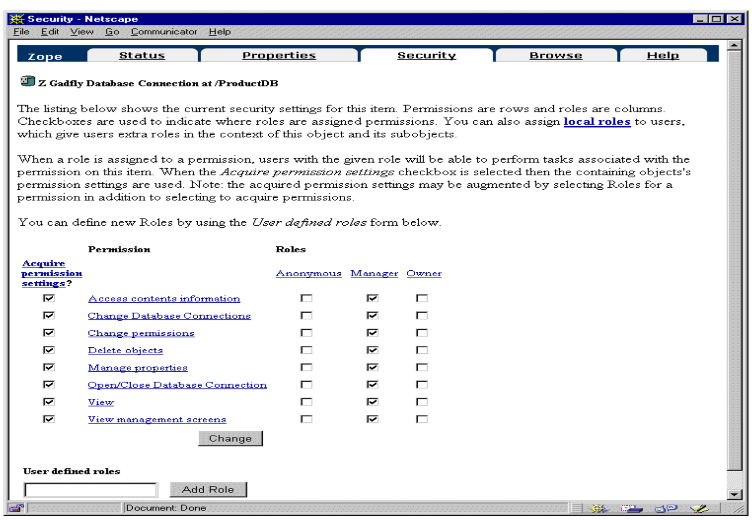

Figure 25. Database Connection Security

SQL Database Methods

Overview

Z SQL Database Methods provide a means for treating SQL database programs as objects that can be used to publish database data and create advanced *Zope applications*.

Separation of database management and presentation management is accomplished through database methods and documents. Unlike many other *Zope* objects, such as *DTML Methods* and *Images*, database methods are not meant to be accessed directly through the Web. Rather, they are used by other *Zope* objects, such as *DTML Documents* to obtain data to be displayed. *SQL Database Methods* are used to manage SQL, and *DTML Methods* are used to manage presentation.

Creating SQL database methods

An *SQL Database Method* is created by selecting "*Z SQL Database Method*" in the add list of a *Folder Contents* view, after which an input form is presented. In addition to the standard *Zope* properties, *id* and *title*, three additional properties may be specified.

The required property, *connection id* is used to specify which database connection will be used by the database method. A *Database Connection* object must be created prior to creating an *SQL Database Method*. The database connection can be created in the current folder, or in any folder above the current folder. The connection list shown in the input form shows all connections that can be found in the current folder or in folders above the current folder.

The optional property, *arguments*, is used to specify one or more input arguments. Input arguments are used to customize a query based on input data, such as data passed in a Web request, or in a *DTML expr* attribute. Arguments not specified in this list will not be acquired from the REQUEST environment (e.g., form variables). Multiple arguments are separated by one or more spaces or tab characters. Each argument is specified as an argument name, an optional argument type, and an optional argument default. The argument name should consist of letters, underscores, and digits and should start with a letter.

```
name:type="default"
```

The type should be one of the values shown in Table 1. The default type is *string*.

Type	Description
int	An integer value.
float	A floating-point number.
string	String. This is the default type.
required	A non-empty string
date	A date-time value. A wide variety of formats are accepted[1]. Year, months, and day cab be provided in any unambiguous order[2]. Month names and abbreviations of various forms may be provided. Hours, minutes and seconds are optional and are separated from the date by one or more spaces and from each other by colons. A 24-hour clock is assumed unless times are followed by *am*, *AM*, *pm* or *PM*.
list	A list of values. This is useful to insure that a sequence of values is available when the query template uses an *dtml-in* tag to iterate over inputs. The list type may be combined with the *int*, *float*, and *date* types to specify a list of integers, floating-point numbers, and dates.
lines	A single input string is split on line endings into a list of one or more values.
tokens	A single input string is split on sequences of one or more space, tab, new-line, or carriage-return characters into a list of values.
text	A single input string in which sequences of new-line and carriage-return characters are converted to single newline characters.

TABLE 1. Database-method input argument types

1. This includes the ISO standard format: *yyyy-mm-dd hh:mm:ss*
2. An input 1-2-1997 is interpreted as January 2, 1997.

Table 2 shows several examples of input arguments.

Arguments	Explanation
`x y`	Two string arguments, x and y
`name:required age:int`	A non-blank string argument name, and an integer argument, age
`name:required color:required="blue"`	A non-blank string argument name, and an optional non-blank string argument, color, that defaults to "blue"
`ids:list:int`	An argument, ids, that must be a list of integers.
`ids:tokens`	An argument, ids that is a string that will be broken into a list of tokens.

TABLE 2. Some sample database-method input arguments

The *query template* is the source of the desired SQL query or commands in *DTML* format. *DTML* tags may be used to substitute text into a query based on input data and on information defined in *Zope*.

Testing and debugging SQL database-methods

After an *SQL Database Method* has been created, it should be tested to make sure that the query template is correct and to generate search interfaces. To test an *SQL Database Method*, select its icon in the contents view of the containing *Folder*. Selecting the *Test* tab causes an input form to be displayed, as shown in Figure 5. Selecting the "*Submit Query*" button causes the database method to be executed and the results to be displayed in a table . The SQL used is also shown. If the results are not what was expected, the SQL may be inspected to see if the expected substitutions were performed.

If an error occurs, an error message is displayed as shown in Figure 26.

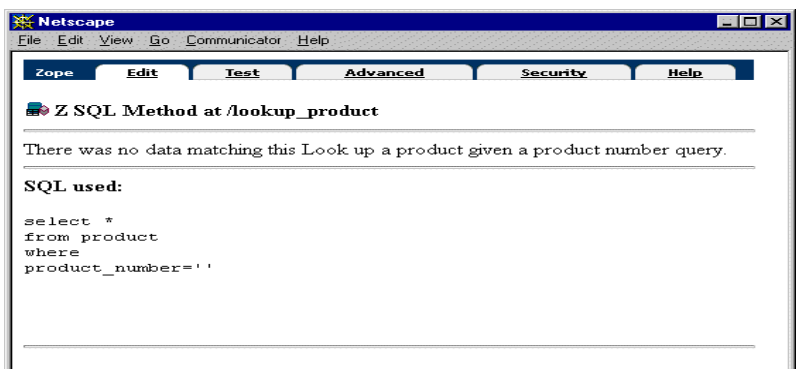

Figure 26. Test output showing an error message and SQL used.

Query templates

Query templates are used to generate SQL statements based on Web request input data and other Zope data. Consider the following example of a query template:

```
SELECT * FROM product
WHERE product_number = <dtml-sqlvar product_number type=string>
```

The intention is to dynamically create a query using data that either comes in with the HTTP REQUEST or is otherwise available to the *SQL Database Method* object (e.g., acquired data, folder property data, etc.). In this example, a *sqlvar* tag was used to insert a value into a SQL statement.

Consider an HTML form that contains the following INPUT tag:

```
<INPUT NAME="product_number" TYPE="text" SIZE="6">
```

If the user entered the value `widget` into the `product_number` input box then the query template shown above would be evaluated to:

```
SELECT * FROM product WHERE product_number = 'widget'
```

An ordinary *dtml-var* tag could be used as well, however, the `sql_quote` *var* tag attribute should be used to make sure that SQL quote characters are handled correctly. The *dtml-var* tag would be useful if special *dtml-var* tag features like the *lower* attribute were needed:

```
SELECT * FROM parts
WHERE part_no = <dtml-var part_number sql_quote lower>.
```

Because *query templates* are themselves document templates you have access to all of the DTML constructs. This includes all of the looping, conditional and iterative commands. Consider the following query template:

```
SELECT * FROM sales where part_no in
(
<dtml-in list_promotional_items>
  <dtml-unless sequence-start>,</dtml-unless>
  '<dtml-var promo_item_part_number fmt=sql-quote>'
</dtml-in list_promotional_items>
)
```

This SQL Database Method would executes another SQL Database Method called `list_promotional_items` and, for every row in the result, insert a case in the SQL *dtml-in* list. In this example, the variable `promo_item_part_number` is in the results of the `list_promotional_items` query.

The query templates can get their variables from either the HTTP request or from any variables available in the *Folder* containing the *SQL Database Method* object. It is often useful to develop *SQL Database Methods* with this lookup order/variable resolution in mind. The following rules control the order in which variable lookup is performed, depending on whether or not the variable name in question is in the *arguments* property of the *SQL Database Method*.

In arguments list?	First Lookup	Second lookup
YES	Variables in HTTP REQUEST	Variables in the folder containing the *SQL Database Method*.
NO	Variables in the folder containing the *SQL Database Method*	NONE

TABLE 3. Lookup order for SQL template variables

It is often necessary to execute more than one SQL statement in a single *SQL Database Method*. *SQL Database Methods* define a variable, `sql_delimiter`, that can be used to divide individual SQL statements. Consider the following two SQL statements which debit a user's checking account and credit a loan account by the same amount:

```
UPDATE checking_account_balances WHERE
  <dtml-sqltest customer_number column=customer_no type=string>
SET balance = balance - <dtml-sqlvar loan_payment type=float>

<dtml-var sql_delimiter>

UPDATE loan_account_balances WHERE
  <dtml-var customer_number column=customer_no type=string>
SET balance = balance + <dtml-sqlvar loan_payment type=float>
```

Note that no more than one SQL *select* statement may be used in a single *SQL Database Method*.

Editing SQL database methods

Editing database methods is done using the database method Edit view. In addition to the standard Zope property, *title*, the *connection id*, *arguments*, and *query template* can be changed.

It is recommended that you test a database method after editing it, using the *Test* view.

Advanced SQL database method configuration

The Advanced view (Figure 27) provides access to advanced configuration properties of *SQL Database Methods*.

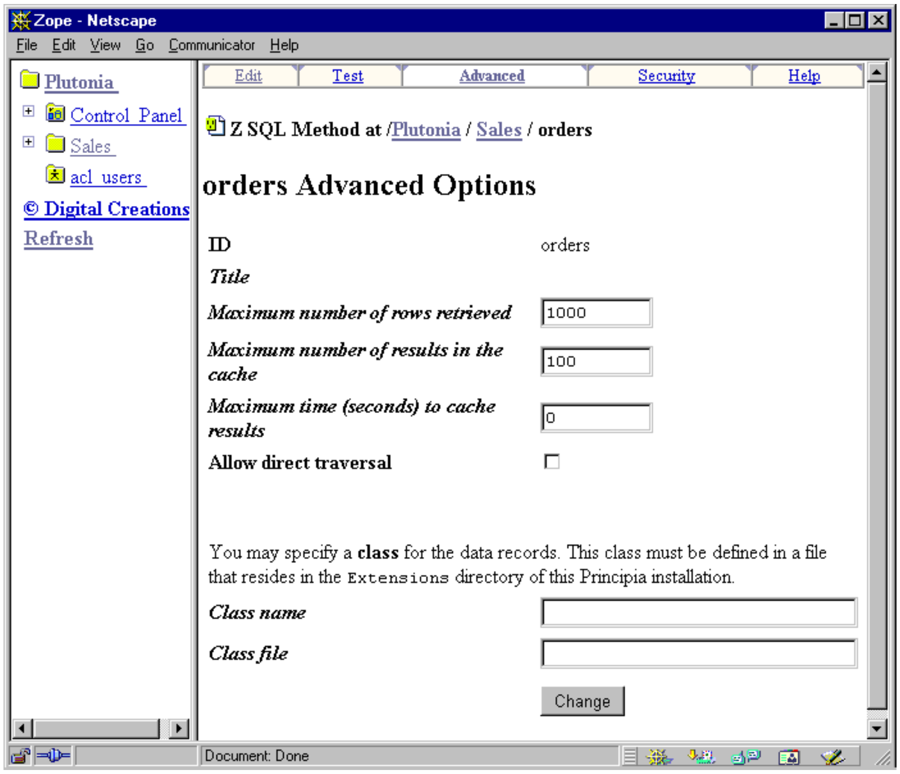

Figure 27. *SQL Database Method* Advanced view

Advanced properties include:

- *Maximum number of rows retrieved*
- *Maximum number of results in the cache*
- *Maximum time (seconds) to cache results*
- *Allow direct traversal*

You can also designate a class for the data records retrieved by the *SQL Database Method*. In order to use this advanced feature you need to define both a *class name* and a *class file* name. The class file must be stored in the Extensions directory of the Zope installation. A sample class is detailed in the Database Method Classes section.

Maximum Number of Rows Retrieved

The maximum number of rows retrieved can be limited by setting this value. This value cannot be blank. This parameter limits the number of rows returned from the underlying database to the *SQL Database Method* object.

Caching results

For performance reasons, it is possible to cache the results of queries. Caching is controlled by the properties: *maximum number of results in the cache* and *maximum time (seconds) to cache results*. The property, *maximum number of results in the cache*, limits how large the cache can grow. The property, *maximum time (seconds) to cache results*, sets the maximum age of cached results. Setting either of these to zero disables caching. the key for entries in this cache is the rendered SQL statement that generated the result.

Direct Traversal

Database methods that return a single record requested with a primary key can be traversed with a URL to access a record object directly. Normally, argument names must be included in the URL. If the database method has a single argument, then the *allow direct traversal* is displayed in the *Advanced* view. If the argument values are known to not conflict with database method names, the *allow direct traversal* option should be enabled and the argument name need not be included in the URL.

Database Method Classes

It is possible to assign a class to the records returned by a Database Method. This class is a Python[1] class defined in a Python file located in the *Extensions* directory[2] of the Zope installation being considered.

Classes can be used to augment otherwise static records returned from a database query with more interesting behavior. For example consider the following class definition (in a file called hardware.py located in the Extensions directory of the current Zope installation):

```
class ComputerHardwareItem:
  def dollar_volume_backordered(self):
     return self.unit_price * self.backordered
```

By virtue of assigning the `ComputerHardwareItem` class to this Database Method we now have access to the `dollar_volume_backordered` function for each record returned by the query. We can now refer to the `dollar_volume_backordered` DTML variable inside an iteration over query results.

1. Python is a very high-level object-oriented programming language. For more information on Python, visit `http://www.python.org`.
2. For security reasons, the Python file cannot be uploaded via the Web. You must have access to the file system of the server that is running Zope.

Database Method Document Template Markup Language Tags

Overview

SQL Methods support a number of specialized tags for inserting values or comparisons into SQL source. These tags provide a number of advantages:

- SQL Templates are simpler and clearer,
- Values are type-checked to make sure that the data are at least syntactically valid,
- Single quotes in string values are appropriately quoted,
- When doing comparisons in selections, multiple input values can be automatically rendered using the SQL `in` test.
- When doing comparisons in selections, inputs may be optional.

Inserting values with the *sqlvar* tag

The *sqlvar* tag is used to type-safely insert values into SQL text. The *sqlvar* tag is similar to the *dtml-var* tag, except that it replaces text formatting parameters with SQL type information.

The attributes used by the *sqlvar* tag are shown in Table 4.

Name	Description
name	The name of the variable to insert. As with other DTML tags, the `name=` prefix may be, and usually is, omitted.
type	The data type of the value to be inserted. This attribute is required and may be one of `string`, `int`, `float`, or `nb`. The `nb` (for non blank) data type indicates a string that must have a length that is greater than 0.
optional	A flag indicating that a value is optional. If a value is optional and is not provided (or is blank when a non-blank value is expected), then the string `null` is inserted.

TABLE 4. Attributes of the *sqlvar* tag

For example, given the tag:

```
<dtml-sqlvar x type=nb optional>
```

if the value of `x` is:

```
Let’s do it
```

then the text inserted is:

```
’Let’’s do it’
```

however, if x is omitted or an empty string, then the value inserted is `null`.

Inserting equality comparisons with *sqltest*

The *sqltest* tag is used to test whether an SQL column is equal to a value given in a DTML variable. .

name	description
name	The name of the variable to insert. As with other DTML tags, the `name=` prefix may be, and usually is, omitted.
type	The data type of the value to be inserted. This attribute is required and may be one of `string`, `int`, `float`, or `nb`. The `nb` data type indicates a string that must have a length that is greater than 0.
column	The name of the SQL column, if different than `name`.
multiple	A flag indicating whether multiple values may be provided.
optional	A flag indicating if the test is optional. If the test is optional and no value is provided for a variable, or the value provided is an invalid empty string, then no text is inserted.
op	A parameter used to choose the comparison operator that is rendered. Default is ‘=’.

TABLE 5. Attributes of the *sqltest* tag

For example, given the tag:

```
<dtml-sqltest color column=color_name type=nb multiple>
```

If the value of the color variable is `"red"`, then the following test is inserted:

```
column_name = ’red’
```

If a list of values is given, such as: `"red"`, `"pink"`, and `"purple"`, then an SQL `in` test is inserted:

```
column_name = in (red, pink, purple)
```

The optional parameter ’`op`’ support the following operations:

```
eq renders to '='      ne renders to '<>'
gt renders to '>'      ge renders to '>='      also gte
lt renders to '<'      le renders to '<='      also lte
```

So because of this, you can render:

```
<dtml-sqltest foo op=gt type=string>
```

results will be:

```
foo > 'bar'
```

One note is that if ’op’ doesn’t match the options listed above, it will render whatever is provided, so:

```
<dtml-sqltest foo op=like type=string>
```

would render as:

```
foo like 'bar'
```

Inserting optional tests with *sqlgroup*

It is sometimes useful to make inputs to an SQL statement optional. Doing so can be difficult, because not only must the test be inserted conditionally, but SQL boolean operators may or may not be needed depending on whether other, possibly optional, comparisons have been done. The *sqlgroup* tag automates the conditional insertion of boolean operators.

The *sqlgroup* tag is a block tag. It can have any number of **`and`** and **`or`** continuation tags. The attributes of the sqlgroup tag are shown in Table 6.

name	description
required	The required attribute is used to flag groups that must include at least one test. This is useful when you want to make sure that a query is qualified, but want to be flexible about how it is qualified.
where	The where flag is used to cause an sql “where” to be included if a group contains any text. This attribute is useful for queries that may be either qualified or unqualified.

TABLE 6. Attributes of the sqlgroup tag

The *sqlgroup* tag checks to see if text to be inserted contains other than whitespace characters. If it does, then it is inserted with the appropriate boolean operator, as indicated by use of an **’`and`’** or **’`or`’** tag, otherwise, no text is inserted.

Suppose we want to find people with a given first or nick name, city and minimum and maximum age. Suppose we want all inputs to be optional, but want to require *some* input. We can use DTML source like the following:

```
select * from people
<dtml-sqlgroup required where>
  <dtml-sqlgroup>
    <dtml-sqltest name column=nick_name type=nb multiple optional>
  <dtml-or>
    <dtml-sqltest name column=first_name type=nb multiple optional>
  </dtml-sqlgroup>
<dtml-and>
  <dtml-sqltest home_town type=nb optional>
<dtml-and>
  <dtml-if minimum_age>
     age >= <dtml-sqlvar minimum_age type=int>
  </dtml-if>
<dtml-and>
  <dtml-if maximum_age>
     age <= <dtml-sqlvar maximum_age type=int>
  </dtml-if>
</dtml-sqlgroup>
```

If we evaluated this template with values set for home town and name, we would get an SQL query like the following:

```
select * from people
where
((nick_name='Jim'
 or first_name='Jim'
)
 and home_town='Cleveland'
)
```

This example illustrates how groups can be nested to control boolean evaluation order. It also illustrates that the grouping facility can also be used with other DTML tags like *dtml-if* tags.

Index