

DSN Parsing Package

Georg Bauhaus*

Date: Fri, 14 Jan 2005 06:48:28 +0100

1 Data Source Name Parts

The *DSN* package and children provide a parser that selects components from a Data Source Name (DSN). A data source name is a string value that states how to connect to a database. A DSN is made from a few parts. In its standard form, the first part of a DSN specifies a database system (brand), the second part names a user, the third part a server, and the fourth part is a database name.

The packages assume DSN syntax conventions similar to the conventions used for PEAR::DB, a PHP database abstraction layer. Details about acceptable data source names are spelled out in the grammar. For example, some parts of a DSN can be left out, some parts can be optionally added.

The parser can handle some syntax errors in DSNs. It first follows the grammar rules, including error rules, then it performs a few additional checks. The checks help detect spurious input, to some extent. [TODO: Write the predicates, easy enough, but not there yet. Try for example, `db2 : //usr : 4 !@tcp@user`, which is clearly wrong. The parser does not accept this string without using error rules. But with the checks, it could report two @s and just say No. Likewise, the presence of an @ indicates where the server part can be at all.]

The rest of this section is about some pitfalls of DSN construction and can be skipped.

(When seen from the perspective of a parser, a DSN might be fine. But seen from the perspective of the application that uses the parser's results a syntactically valid DSN might lead to quite unwanted behavior. On the other hand a DSN can have syntax errors that stop the application when the syntax errors are no more than typos. These properties of DSN are relevant when DSNs are input to a program at runtime.

The typical construction of environment variables in shell scripts provides an example of a DSN that is syntactically correct, and plausible. Yet the person who has written the DSN string might not have wanted the result. A shell script can combine a DSN from several variables. Consider a variable in the position of the server name, as in

```
FRUIT_DB=mysql://$SERVER/fruits.
```

Let \$SERVER have the value `frank@home`, which is not a server part string according to the DSN grammar, but a user part followed by '@' and a server part. The result of shell variable expansion is

*send electric mail to bauhaus@futureapps.de

```
FRUIT_DB=mysql://frank@home/fruits.
```

This is a correct and plausible DSN. It might be harmless to use this DSN, even if the intention was not to connect the program to the `fruits` database running at a server home as user `frank`. Or it might not be harmless. In general, when parsing succeeds this is not yet an indication of a valid DSN. There are false positives. (You can check the pieces found by the parser using the functions from the *DSN* package described in a later section.)

As an example of the difficulty of finding spurious input, assume that in some DSN a server part shell variable is given. A query string shell variable has been appended, like so:

```
LOCAL_DEFAULT_DB=pgsql://$SERVER$QS.
```

If the `'?'` starting the query string part has been accidentally omitted when assigning to `$QS`, it is not easy for the DSN parser to know what to do with the text resulting from shell variable expansion. It cannot know whether an `'@'` appearing in the query string part is in fact from the query string part stored in `$QS`. Rather, it will interpret the `'@'` as introducing a valid server part. (Whether or not `'@'` is a valid character in a query string is subject to similar considerations. Someone might just have placed it in `$QS` by accident, or in an attempt to fool the program, or as part of a clever solution.)

On the other hand, if `'?'` is present the parser can decide that an `'@'` following the `'?'` is probably wrong. Likewise, a missing slash, `'/'`, can lead to some speculation about the possible intent of the rest of the text.

The parser doesn't fully solve the problem of finding spurious input (of course?). It is assumed that programmers use a suitable set of predicates to test the pieces of input provided by the parser. Still the parser tries to be helpful and reports when an input could be correct provided some symbol were added.)

1.1 The Grammar of DSNs

The grammar has *DSN* as the start symbol. There are two rules with *DSN* on the left, the first produces strings in the traditional DSN form, the second produces strings in variant syntax. (Rules dealing with errors have been left out here, they can be found in the spec of package *DSN*.) The \perp sign stands for the end of input.

| | | |
|-------------|---|--------------------------------------------------------|
| DSN | → | DB Traditional |
| DSN | → | DBSpeak Variant |
| DB | → | “access” “dbase” “db2” “ifx” “mysql” “mssql” |
| DB | → | “mysql” “navision” |
| DB | → | “oci8” “pgsql” “solid” “sqlite” “sybase” |
| Traditional | → | \perp |
| Traditional | → | ‘:’ ‘/’ ‘/’ DBTail |
| Traditional | → | ‘:’ ‘/’ ‘/’ Login QS |
| Traditional | → | ‘:’ ‘/’ ‘/’ Login DBTail |
| Login | → | Server |
| Login | → | User ‘@’ Server |
| User | → | NAME |
| User | → | NAME ‘:’ PASSWORD |
| Server | → | Protocol ‘+’ HostPort |
| Server | → | HostPort |
| HostPort | → | Host |

| | | |
|----------|---|------------------------------|
| HostPort | → | Host ':' Port |
| Host | → | IP HOSTNAME |
| DBTail | → | '/' '/' Path QS |
| DBTail | → | '/' Simple QS |
| Path | → | UNIXPATH |
| Path | → | DRIVELETTER ':' '/' UNIXPATH |
| Simple | → | UNIXPATH |
| Port | → | DECIMAL_DIGITS |
| QS | → | '?' QUERYSTRING |
| QS | → | ⊥ |
| DBSpeak | → | "odbc" '(' DB ')' |
| Variant | → | ⊥ |
| Variant | → | ':' '/' '/' ".....TODO....." |

Symbols in all upper case that don't appear in the grammar have their usual meaning. (The comments in the spec of package *DSN* give a more precise description.)

1.2 Using the Parser

The file `INSTALL.txt` describes how to build the library.

When you are ready to use the library, write `with DSN.Messages, DSN.Parsers;` at the start of your compilation unit. Instantiate the generic procedure *parse*. (The generic takes another procedure for reporting errors, and one that is called for each successful derivation.) The parser instance consumes a DSN string and (re)fills an object of type *Data_Source_Details* with details found in the DSN string. (The object is a third generic parameter.)

Each time the *parse* instance succeeds, the calling program is notified via the generic formal procedure *got_one*. The programmer can investigate the result using any of the functions described in the next section. If the *parse* instance fails at some point, every failure is reported through the generic formal *report*. This allows a library user to keep track of what has happened during parsing.

1.2.1 The Interface of *Data_Source_Details*

d stands for an object of type *Data_Source_Details* that has been filled during parsing. *d* has been declared in the client program, and has been passed as an argument to the *parse* instance.

database_system_name(d)

Returns a database system name, like "pgsql". The grammar rules for *DB* list the known values. There is an alternative function of the same name that returns a value of enumeration type *Database*. This type consists of one value for each known database system.

[TODO: extend the following descriptions, which have been copied from package *DSN* and only slightly edited.]

dialect(d)

an empty string or, if variant syntax is used, the substring following the database system name in parenthesis

username(d)

a possibly empty string

password(d)

a possibly empty string

connection_protocol(d)

A possibly empty string. There is an alternative function of the same name that returns an enumeration value of type *Protocol* (or the *Unknown* protocol). This type consists of one value for each known protocol.

hostname(d)

if not an empty string, a HOSTNAME string as explained in the grammar

port(d)

The port number specified in the DSN or else 0 (assuming a database system doesn't listen on port 0.) (The value returned can be greater than the highest port number available, the parser doesn't check.)

database_name(d)

This is either a file system path or a simple name. The string can be empty.

query(d)

The part after the question mark, if any.

1.2.2 The `dsnt` Demo Program

`dsnt` is a simple interactive application that reads DSN strings and prints any findings of the parser. (There is a short introduction in the units “`dsnt.adb`”, and “`dsnt-io.adb`”.) The author hopes the program text is useful as a sample of how to construct an application using the parser provided by the DSN packages.

1.3 Configuration Parameters

The package *DSN.Config* declares some fixed capacity constraints. They should affect only the parser. The constraints have an effect on the maximum size of parser buffers, and one the index constraints of input strings. [TODO: Fix the details and explain.] Input strings should have values whose bounds are within the limits implied by *DSN.Config*.

The variable *verbose* in *DSN.Config* can be set to *True*. This will result in trace output on *current_error* showing each completed derivation. A default printing procedure from package *DSN.Support* is used for this. You can set this procedure to your own by setting a pointer variable in the package.