

XML Subversion

You remember when you first learned about relational databases—the seductive simplicity of the grid found a special place in your heart. Its unwavering uniformity assured you that you would always, always, find what you are looking for.

But since then, reality has set in. Those rigid little boxes have become oppressive. As your projects grow, your data often doesn't fit into the perfect little rows that you have crafted.

As an alternative, the free-form structure of XML is tantalizing. You'd love to be able to adjust the structure of your data at a moment's notice. But what about speed and efficiency? You can't parse an entire XML tree every time you need some data. You need the speed of a relational database, but crave the organic structure of XML.

Your solution is *XML::EasySQL*, which is a two-way SQL/XML base class for Perl. Here are some of the benefits it provides:

- Two-way transforms between XML and relational data.
- Smart SQL updates: Only altered tables are updated.
- Unlimited tree depth.
- Multiple tables can merge into one XML tree, then back again.
- Precise control over how data is translated.
- Either an easy XML interface or plain DOM.
- Database independency.

XML::EasySQL works by first taking data that is output from DBI and turning it into an XML tree. The programmer is then free to modify the data using the easy XML interface that's provided, or start hacking directly on the underlying *XML::DOM*. When you're ready to dump the changed data back to the database, you only have to call one method. The tree is stored as shown in Figure 1.

XML::EasySQL consists of two classes: *XML::EasySQL* and *XML::EasySQL::XMLnode*. *XML::EasySQL* is the actual data ob-

ject class. Its methods transform data between XML and SQL forms. You probably want to use *XML::EasySQL* as the base class for your data objects.

XML::EasySQL::XMLnode is optional, although it's highly recommended. It's really just a simplified DOM interface for Perl. The class is derived from a fork of Robert Hanson's excellent *XML::EasyOBJ* module, which seeks to offer a more "Perlish" interface to the *XML::DOM*. (So far, Perl doesn't have native support for the DOM, which seems the natural ascendant of its built-in hashes. But a hacker can dream, can't he?)

Listing 1 shows a synopsis from the *XML::EasySQL* manual, which shows both classes working together.

Installing *XML::EasySQL*

The easiest way to install *XML::EasySQL* is with the CPAN module:

```
$ su -l
$ perl -MCPAN -e shell
cpan> install XML::EasySQL
```

Or you could manually download and install the module. Just snag the latest version from CPAN at <http://search.cpan.org/~curtisf/>.

After you've downloaded it, invoke the standard incantation:

```
$ tar fxzv XML-EasySQL-version.tar.gz
$ cd XML-EasySQL-version
$ perl Makefile.PL
$ make
$ su
$ make install
```

The *XML::EasySQL* Class

I recommend that you use *XML::EasySQL* as a base class for your data objects. You don't have to, but if you do use the class directly, you'll wind up polluting your constructor with some pretty hairy parameters.

Curtis is a journalist and programmer who lives in Portland, Oregon. He can be reached at curtisf@fultron.net.

Listing 2 shows a simple base class (*User*) that's derived from *XML::EasySQL*. See what I mean about the constructor parameters? You're not going to want to pass that chunk of XML to your data object constructor every time you need a new object. If you simply use *XML::EasySQL* as a base class, you can neatly tuck all that XML inside the package.

But what is all that XML anyway? We'll get to that later. For now, just keep in mind that the XML chunk in Listing 1 is used to define how your SQL maps to XML.

So the new class *User* has inherited all of *XML::EasySQL*'s methods. The constructor passes that XML chunk to the base class constructor, *XML::EasySQL::new*. You'd use the *User* class as shown in Listing 3.

Note that the *User* class, like its base class, still needs its data argument passed through the constructor. You will probably find that too messy and want to make a base class of your own that handles all of the SQL communication, and use that as the base class for all your data objects. For example, a base class called "*Base*" could look something like Listing 4. Our new *User* class, with its new base class, is shown in Listing 5.

Now that the SQL query is hidden, the *User* object could be constructed this way:

```
my $user = User->new({db=>$db, user_id=>2, comment_id=>183});
```

And to save any changes made to the XML, all that is needed is:

```
$user->save();
```

The rest of the *User* interface remains unchanged. If you are writing a large program with many different types of data objects, you'll probably want to make more than one base class.

The XML Schema

Remember that XML chunk we keep passing to the *XML::EasySQL* constructor? Here's the skinny: Every *XML::EasySQL* object needs an XML schema. The schema tells *XML::EasySQL* how each column is supposed to map in and out of the XML tree.

Table columns can map to an XML tree one of three ways:

- **attrib.** This simply applies the column value as an XML attribute on the root node. For example, if a column named *id* was mapped as an *attrib*, the resulting XML document would look like this: `<root id="23">...`
- **string.** This is an XML string that's a child of the root node. If a column named *headline* was mapped as a string, the resulting XML document would look like this: `<root><headline>Man Bites Dog</headline>...`
- **element.** This is the most important feature in the schema. If a column is mapped as an element, the data is parsed into an XML branch and grafted onto the root node of the XML document. This means you can add new nodes on the fly, but they'll still be recorded in the database. For example, say a column named *bio* in a database contained this string: `<bio><name>Curtis Lee Fulton</name></bio>`. It would get parsed into XML and grafted onto the DOM. You could then manipulate it like any other XML document, including adding new branches.

Here's a simple schema:

```
<schema name="users" default="string"></schema>
```

Here's a more complex one:

```
<schema name="users" default="attrib" default_table="users">
<columns>
<id type="attrib"/>
<group_id type="attrib"/>
```

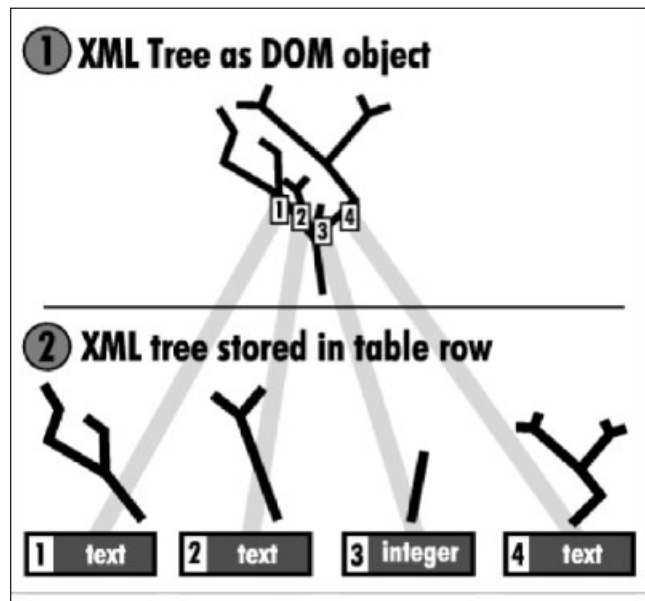


Figure 1: How EasySQL stores an XML tree in a relational database.

```
<email type="string"/>
<bio type="element"/>
<history table="comments" type="element"/>
</columns>
</schema>
```

The *XML::EasySQL* schema can have three root attributes: *name*, *default*, and *default_table*.

- **name.** Sets the name of the root XML element. If missing, it defaults to "xml."
- **default.** The default type, which controls how incoming SQL column data is processed. If *default* is missing, then *XML::EasySQL* will ignore SQL columns that aren't specified in the schema. See "type" in the following list of column elements for more details on the possible types.
- **default_table.** The default table a column belongs to. If missing, it defaults to what the *name* attribute is set to.

The schema can have multiple column entries. Each entry must have a unique tag name that matches a real column name in a SQL table. Column elements can have two attributes:

- **type.** This describes how the SQL data will map onto the XML tree. There are three types: *attrib*, *string*, and *element*.
- **table.** The table the column belongs to. If missing, it defaults to "default_table."

The Node Interface

EasySQL's built-in XML node interface is *XML::EasySQL::XMLObj*. The interface is a fork of Robert Hanson's wonderful *XML::EasyOBJ* module, which he wrote because he wanted a more "Perlish" interface to *XML::DOM*. The best part about Hanson's interface is that its underlying DOM is always available. You can get at the underlying *XML::DOM* element from any *XMLObj* object by calling the *getDomObj* method.

XMLObj can do almost anything *XML::DOM* can do, but with less demand on the user. Listing 6 gives a synopsis.

Here's an example with actual XML. Say you've got a tree that looks like this:

```

<root>
  <branch>
    <twig some_attr="I am">
      <leaf1>leaf1a</leaf1>
      <leaf2>leaf2b</leaf2>
      <leaf3>leaf3c</leaf3>
    </twig>
    <twig some_attr="the Walrus">
      <leaf1>leaf1d</leaf1>
      <leaf2>leaf2e</leaf2>
      <leaf3>leaf3f</leaf3>
    </twig>
  </branch>
</root>

```

Navigating an XML tree with *XML::EasySQL::XMLObj* is almost like pawing through a bunch of nested hash references, except that the syntax is different. Also, hashes have a 1:1 ratio of name-to-value pairs, while each XML tag can have a string value and any number of attributes.

Let's say we wanted to print the value "leaf2e" from the above XML fragment. Assuming that *\$doc* is the root element of the XML page ("root"), your code is going to look like this:

```
print $doc->branch->twig(2)->leaf2->getString();
```

The *getString* method returns the text within an element.

The *XMLObj* manual is fairly straightforward, but to get you started, here's a brief rundown of how to accomplish XML tasks with *XMLObj*:

Retrieve an element. There are two ways to get at an element. The easiest way is to just call it as a method:

```
$doc->element_i_want;
```

Sometimes, you'll have a tag name that won't fly as a method call. If so, just use the method *getTagName*:

```
$doc->getTagName('element_i_want');
```

Remove an element. To remove an element, just call the method *remElement*. It removes a child element of the current element:

```
remElement( TAG_NAME, INDEX )
```

The name of the child element and the index must be supplied, although leaving the INDEX parameter undefined will simply remove the first occurrence of the named element.

Retrieve an attribute. Not rocket science, this one. Call *getAttr* like so, and you'll get the value of the named attribute:

```
$element->getAttr('attribute_name');
```

Fetch a tag name. If you're enumerating through a bunch of child elements, you might want to know their names. To do so, just call *getTagName*. Here's how:

```
foreach my $element ( $doc->getElement() ) {
  my $name = $element->getTagName();
}
```

Set an attribute. There's a good chance that at some point in your life, you'll want to set an attribute value. *setAttr* is the way to go. In return for its services, all it asks for is a series of name/value pairs. Provide that, and it will do the deed:

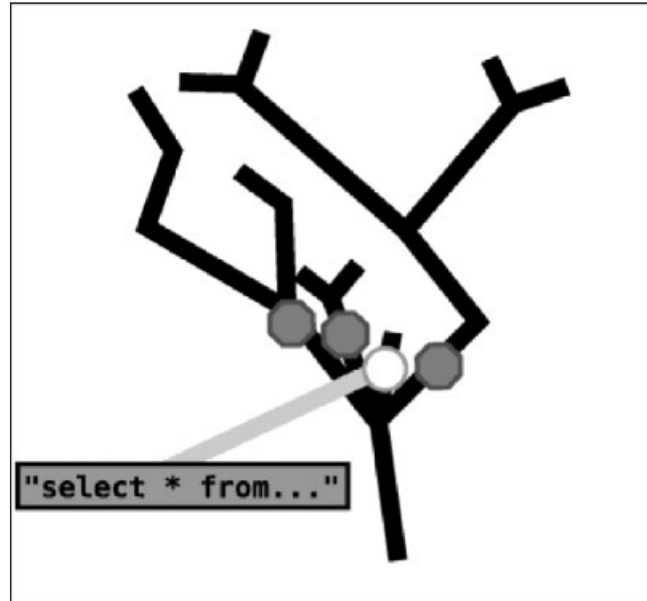


Figure 2: Only strings and attributes living in the root level of an XML tree can be referenced in an SQL query.

```
$element->setAttr('attrib_name_1', 'what do we want?', 'attrib_name_2', 'XML',
'attrib_name_3', 'When do we want it', 'attrib_name_4', 'Now!');
```

Remove an attribute. When the attribute has served its purpose, send it to to sleep with the fishes, like so:

```
$element->removeAttr('Don Corleone');
```

If you find you need more explicit control over the underlying DOM, you can get to it from any node with the *getDomObj* method:

```
$doc->getDomObj
```

The method simply returns the DOM object associated with the current node. This is useful when you need fine access via the DOM to perform a specific function. If you end up working directly with *XML::DOM*, you're going to have to flag parts of the XML yourself as needing updates. If you don't, changes you make to the XML tree won't get saved in the database.

If you've used *XML::DOM* to make a change anywhere in the XML tree except for an attribute of the root element, call *EasySQL::flagSync(base_name)*. *Base name* is the name of the table (or subroot element) that holds the changes. If you have changed an attribute of the root element, call *EasySQL::flagAttribSync(attribute_name)*.

If you don't want to muck with *flagAttribSync* and *flagSync* and you don't mind the performance hit, just call *EasySQL::getSQL(true)* before you write to your database. Calling *getSQL* with a *true* parameter will cause the entire table to be updated, whether or not it's changed.

Designing Tables for EasySQL

A good way to design a relational database for *EasySQL* is to start with the rudimentary XML tree we think we'd like to work with and then work backwards.

EasySQL allows for fairly flexible database design, although there is one rule you are constrained to: Strings and attributes can only be referenced in a SQL query if they exist on the root level of the XML tree (see Figure 2). The result is that any data living deeper than root level in the XML tree can't be explicitly queried.

Let's say we decided on an XML tree that looks like this:

```
<user id="23">
  <bio>
    <username>curtisf</username>
    <password>secret</password>
    <name>Curtis Lee Fulton</name>
    <resume>xxxxxxxxxxx</resume>
  </bio>
  <plan>
    <entry date="6/01/04">hello world</entry>
    <entry date="6/12/04">goodbye cruel world</entry>
  </plan>
</user>
```

Every XML::EasySQL object needs an XML schema. The schema tells XML::EasySQL how each column is supposed to map in and out of the XML tree

Pretty simple. It looks good, but remember that only root level trees get their own tables in *EasySQL*. That's fine for most of the data here, but not for all.

It's a good bet that the *id* attribute, as well as the *name* and *username* strings, are going to be queried. *id* is okay where it is because it's an attribute of the root tag, but the other two elements must be moved. Let's reformat the tree to look like this:

```
<user id="23">
  <username>curtisf</username>
  <name>Curtis Lee Fulton</name>
  <bio>
    <password>secret</password>
    <resume>xxxxxxxxxxx</resume>
  </bio>
  <plan>
    <entry date="6/01/04">hello world</entry>
    <entry date="6/12/04">goodbye cruel world</entry>
  </plan>
</user>
```

Not bad, but *username* probably doesn't need its own tag, so let's tighten things up a bit by making it an attribute of the root element:

```
<user id="23" username="curtisf">
  <name>Curtis Lee Fulton</name>
  <bio>
    <password>secret</password>
    <resume>xxxxxxxxxxx</resume>
  </bio>
```

```
<plan>
  <entry date="6/01/04">hello world</entry>
  <entry date="6/12/04">goodbye cruel world</entry>
</plan>
</user>
```

Looks good. Now, let's design our XML schema for our data object:

```
<schema name="users" default="attrib" default_table="users">
  <columns>
    <id type="attrib"/>
    <username type="attrib"/>
    <name type="string"/>
    <bio type="element"/>
    <plan type="element"/>
  </columns>
</schema>
```

Of course, since we've set *attrib* as the default, we don't need to explicitly set the first two:

```
<schema name="users" default="attrib" default_table="users">
  <columns>
    <name type="string"/>
    <bio type="element"/>
    <plan type="element"/>
  </columns>
</schema>
```

So now we just need to design our database table. Here's a PostgreSQL schema to match the example:

```
CREATE TABLE users (
  id integer,
  username character varying,
  name character varying,
  bio text,
  plan text
);
```

We know we're going to use the attribute *id* as the unique ID for each record, so let's tweak the SQL schema a bit:

```
CREATE TABLE users (
  id serial NOT NULL,
  username character varying,
  name character varying,
  bio text,
  plan text
);
ALTER TABLE ONLY users
ADD CONSTRAINT users_pkey PRIMARY KEY (id);
```

Sweet. Now *id* is constrained as a unique primary key and will autoincrement whenever a new record is added.

Once we start coding, we'll have a pretty flexible data object *User*. We can add as many new branches to *bio* and *plan* as we want, and the branches will get stored in the database.

Just for kicks, let's create a data object for our new schema:

```
package User;
use XML::EasySQL;
@ISA = ('XML::EasySQL');

use strict;
```

```

sub new {
    my $proto = shift;
    my $params = shift;

    # the XML schema string
    $params->{schema} = q(
<schema name="users" default="attrib" default_table="users">
<columns>
<name type="string"/>
<bio type="element"/>
<plan type="element"/>
</columns>
</schema>
);
    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new($params);
    bless $self, $class;
};

1;

```

Let's put our new object into action:

```

# fetch the data from the database
my $data->{users} = $db->selectrow_hashref('select * from users where username =
                                     'curtisf');

# construct the data object
my $user = User->new((data=>$data));

my $xml = $user->getXML();

# fetch some data
my $id = $xml->getAttr('id');
my $password = $xml->bio->getString('password');

# add and modify some data
$xml->bio->setAttr('age', 22);
$xml->bio->city->setAttr('zip', 97201);
$xml->bio->city->setString('Portland');

# write the changes to the database
my $sql = $user->getSQL();
my $q = "update users set ".$sql->{users}." where id = 23";
$db->do($q);

```

Conclusion

There's no reason to shoehorn your data into rigid tables. The *EasySQL* approach allows you to access and modify your data from a unified XML tree, while using any relational database to quickly find the data you need.

TPJ

(Listings are also available online at <http://www.tpj.com/source/>.)

Listing 1

```

# fetch a database row as hash ref
my $data = {};
$data->{users} = $db->selectrow_hashref('select * from users where id = 2');

# init the new EasySQL data object
my $data_object = EasySqlChildClass->new((data=>$data));

# get the root XML element
my $xml = $data_object->getXML();

# make changes to the XML document
$xml->username->setString('curtisleefton');
$xml->bio->setAttr('age', 22);
$xml->bio->city->setString('Portland');

```

101 Perl Articles!



From the pages of *The Perl Journal*, *Dr. Dobb's Journal*, *Web Techniques*, *Webreview.com*, and *Byte.com*, we've brought together 101 articles written by the world's leading experts on Perl programming. Including everything from programming tricks and techniques, to utilities ranging from web site searching and embedding dynamic images, this unique collection of *101 Perl Articles* has something for every Perl programmer.

Plus, this collection of articles is fully searchable, and includes a cross-platform search engine so you can immediately find answers you're looking for. Delivered as HTML files in a ZIP archive or CD-ROM image, download *101 Perl Articles* and burn your own CD-ROM or store it on hard disk.

\$9.95 For subscribers to
The Perl Journal

\$12.95 For nonsubscribers to
The Perl Journal

\$24.95 To subscribe to
The Perl Journal and
receive *101 Perl Articles*

Go to

<http://www.tpj.com/>

now!