

OpenVMS Technical Journal V6

Automatic Program Generation with MySQL and PHP



Automatic Program Generation with MySQL and PHP	2
Overview	2
The Computing Environment	2
Accepting Donations.....	3
User Interface.....	10
Syntactic Validation	12
Semantic Validation.....	13
The Complete Process	13
Conclusion	14
Acknowledgements.....	14
For more information	14

Automatic Program Generation with MySQL and PHP

Dick Munroe

Overview

It's all about leverage. During the heat of the last presidential election, I found myself between engagements and decided to go to Florida and work for the presidential campaign. After spending 3 days with another volunteer building PCs and installing networks for the Florida Democratic Party (FDP) headquarters and the north Florida offices, I found myself behind the keyboard building web applications with the rest of the IT department. I should qualify that. I was the IT department – at least the technical staff part of it. My responsibilities including designing, implementing, and deploying new web applications for everything from eCommerce (donations, and so forth) to volunteer and candidate management to results reporting to statistical analysis, whatever was needed; even better, it was all needed today (or tomorrow at the latest) and as inexpensively as possible.

This article is one part praise for the free software community; another part discussion of one of the techniques I used to get applications out the door fast enough to be useful in the time we had before the election.

The Computing Environment

What I did for the Florida Democratic Party was grind out web applications. At the state and local level of political parties, cost is everything and in the UNIX environment, the free software movement provides most generously. When I arrived, the principal web development tools were [MySQL](#) and [PHP](#), which were used to deploy content from an [Apache](#) web server running on [FreeBSD](#).

MySQL is an SQL compliant database that is capable of scaling smoothly from very small to very large data stores, and that supports transactions, queries over the net, and so forth. It's free, very well supported, and performs well. We used it in every web application deployed during my time with the FDP.

PHP (Personal Home Page) is yet another "swiss army knife" language for developing web pages. PHP version 4 is a 3rd generation programming language with object oriented extensions allowing inheritance, polymorphism, and introspection. PHP also has an extension interface and extensions (dynamically linked libraries) to the language have been written for everything from graphics to data base interfaces. The FDP used PHP 4. Since the election, PHP 5 has been released with significantly improved object-oriented capabilities. All the work discussed here is implemented in PHP 4 and easily ported to PHP 5.

A full discussion of Apache and FreeBSD is beyond the scope of this article. However nothing done by the FDP was specific to either Apache or FreeBSD. All that was needed was a server capable of running PHP and a platform that allowed integration of PHP with a web server and communication with a MySQL server. Technically speaking, the server running Apache didn't have to be capable of running MySQL but it was convenient that it did.

Each application at the FDP required the design of one or more tables in the FDP primary database. Database design wasn't all that difficult for the vast majority of applications that were on the table, but any tool is better than no tool. MySQL is not difficult to deal with when doing rapid, interactive database design, so initially that's what we did. Eventually we found and started using a terrific MySQL specific database design and maintenance tool called [phpMyAdmin](#). All the database table examples and figures for this article were created using phpMyAdmin. If you're going to be responsible for any aspect of MySQL database administration, phpMyAdmin is a must.

The development workstations were all Windows boxes running Windows 2000. It is not my favorite platform but the FDP couldn't argue with the price of the hardware (all hardware was donated and

working workstations were constructed from whatever parts could be salvaged) and had a site license for W2K. Fortunately there are free (or at least inexpensive) editors with language specific extensions for the Windows environment. I downloaded [Cygwin](#) (a UNIX CLI layer for Windows) and started using EMACS and other standard UNIX tools for development on the Windows PC I was using.

Application Development at the FDP

The first job I tackled was fixing up an application written by a couple of college student volunteers that did volunteer management.

[Present data to the user]
Get data from the user
Validate user data
Store/Update using MySQL

Table 1 - Structure of Volunteer Management Application

The structure of the volunteer management application is shown in Table 1. Each "layer" of the application was done in an ad hoc manner. Each page of the application basically duplicated whatever code was needed to interact with the user and the database, perform data validation, and so forth. An enormous amount of the code written for this application dealt with the interface to the MySQL database. Starting sessions, validating data, storing or updating data, and closing sessions were all coded explicitly. No attempt had been made to factor out the database access details into a separate function library or class and the quantity of the code dedicated to dealing with the database obscured the details of the application under development, making it a lot harder to extend. Eventually I finished this job and moved on to the next.

Accepting Donations

The next application was for accepting donations. The details of the application aren't important; what is important is that it would be another web-enabled database application similar in kind, if not in detail, with the volunteer management application.

Given what I had seen in the volunteer management application, I wanted to come up with a more general view of the FDP's web-enabled applications and then use that view to drive the creation of tools that would make it easy to implement those applications. After some thought I realized that most of the applications used regularly by the FDP could be modeled as shown in Table 2.

[Query MySQL Database]
Display data to the user
Collect data from the user
Organize collected data
Validate collected data
Store/Update data in MySQL

Table 2 - FDP Application Architecture

Overall, an application should do the following:

- If data were being changed, one or more queries would be made to the appropriate database.
- Data (if any) would be displayed to the user.
- Data would be collected from the user.

- Data would be organized in structures, making it simple for validation and eventual storing into the appropriate database tables.
- Data would be validated and any additional interactions with the user would be done to correct any errors.
- Data would be stored or updated in the MySQL database.

Given that data was being stored in a MySQL database, I felt that the underlying data abstractions should closely model the relational storage model (tables), making the usual sorts of queries done by the applications easy, while not prohibiting the writing of substantially more complex queries. Ideally, most of the "what data needs to be read/written" from the database would be "automagically" figured out so that the applications could say things like "update the data" and the right things would happen.

Clearly the place to start was with a decent data base abstraction layer – one that hid most of the tedious details of dealing with MySQL while not overly restricting access to MySQL's lower level features. This problem has been solved many times in the past and a quick session with Google turned up www.phpclasses.org, an enormously useful site if you're into PHP programming. www.phpclasses.org is dedicated solely to the collection and distribution of PHP class libraries. The code distributed by www.phpclasses.org comes from all over the world and varies in purpose from the sublime to the ridiculous and in quality from completely professional to totally amateur. I've saved myself a lot of time and my clients a lot of money by using things I've found from www.phpclasses.org either in whole or in part. If you're doing any serious PHP software development, you owe it to yourself to join www.phpclasses.org. (Yes, it is free, but donations are accepted.)

On www.phpclasses.org, I found the [DB](#) class. DB provided an object-oriented interface to MySQL. This solved my initial requirement for something a little higher level than the PHP interface to MySQL. It did not matter that the only database supported was MySQL. The FDP had standardized on a Linux, Apache, MySQL, PHP (aka "LAMP" – or LAMP-like, since the platform was actually FreeBSD) environment, so portability was not of immediate concern. One of the reasons that I chose the DB class was that, if necessary, the DB could easily be ported to support other or additional databases.

This solved the problem of the tedium of accessing the database but it didn't address the problem of a general database table-oriented data collection that could easily interact with any database.

To solve this problem, I designed and wrote the [SQLData](#) class. This class evolved over time, but it was designed to:

- Associate an instance of the class with a specific table in a database.
- Be organized so that data to be stored to or fetched from the database can be manipulated by the name of the field.
- Keep track of the state of data in the instance so that minimal updates can be automatically performed.
- Provide iterators so that loops processing entire tables can be easily written.
- Provide hooks for structuring data as arrays or objects, and potentially arrays or objects stored in other tables

After we did the design of the table for the donations application, I dove into the coding process. The table design for the donations application is shown in Figure 1. As you can see, it is straightforward, with much of the complexity resulting from the eCommerce interface rather than anything fundamental in the nature of the problem of donating money.

Obviously enough, for every field defined in the table, you need a way to access, modify, and store the field's data. Even using SQLData, I had a lot of work to do. Each field needed a read/modify interface (plus additional ones that came in handy once I got into the details of actually doing the work, like controlling the "dirty" state of fields). This was a lot of typing, and it was error-prone too.

Much of the work was largely "cut and paste," but if that's all a human is doing, then maybe it is possible to get a computer to do it.

Field	Type	Attributes	Null	Default	Extra
<u>id</u>	int(11)		No		auto_increment
first_name	varchar(50)		Yes	NULL	
last_name	varchar(50)		Yes	NULL	
company_name	varchar(50)		Yes	NULL	
address	varchar(60)		Yes	NULL	
city	varchar(40)		Yes	NULL	
county	varchar(50)		Yes	NULL	
state	char(2)		Yes	NULL	
zip_code	varchar(10)		Yes	NULL	
phone	varchar(15)		Yes	NULL	
fax	varchar(15)		Yes	NULL	
email	varchar(50)		Yes	NULL	
occupation	varchar(50)		Yes	NULL	
employer	varchar(50)		Yes	NULL	
card_number	varchar(4)		Yes	NULL	
amount	decimal(10,0)		Yes	NULL	
response	varchar(10)		Yes	NULL	
authorization	varchar(6)		Yes	NULL	
trans_id	int(11)		Yes	NULL	
trans_time	timestamp(14)		Yes	NULL	
reference	varchar(20)		Yes	NULL	

Figure 1 - Donations Table Design

One of the greatest things about the architecture of SQL databases is that the metadata (the data describing the database and its content), is stored in an SQL database. Simple queries with results like those shown in Figure 2 make it easy to introspect (literally "look at oneself") the tables in the database and the contents of those tables. Given that the metadata is available to programs in general and to PHP in particular, it is relatively easy to write programs to process database table structures in a very general fashion. Table names, fields within tables, data types of fields, use of fields as keys, and so forth are all available for processing.

Field	Type	Null	Key	Default	Extra
id	int(11)		PRI	NULL	auto_increment
first_name	varchar(50)	YES		NULL	
last_name	varchar(50)	YES		NULL	
company_name	varchar(50)	YES		NULL	
address	varchar(60)	YES		NULL	
city	varchar(40)	YES		NULL	
county	varchar(50)	YES		NULL	
state	char(2)	YES		NULL	
zip_code	varchar(10)	YES		NULL	
phone	varchar(15)	YES		NULL	
fax	varchar(15)	YES		NULL	
email	varchar(50)	YES		NULL	
occupation	varchar(50)	YES		NULL	
employer	varchar(50)	YES		NULL	
card_number	varchar(4)	YES		NULL	
amount	decimal(10,0)	YES		NULL	
response	varchar(10)	YES		NULL	
authorization	varchar(6)	YES		NULL	
trans_id	int(11)	YES		NULL	
trans_time	timestamp(14)	YES		NULL	
reference	varchar(20)	YES		NULL	

Figure 2 - Table Description Query

Once I realized that this data was available, I quickly designed and wrote a simple PHP program to generate classes from the metadata of a MySQL table. A day later, I had a program, `buildClass.php`, which reads the metadata of a MySQL table in a database and emits a class derived from `SQLData` that provides the framework for manipulating data within a single table of a database. Example 1 is a partial listing of the generated class included here for discussion of the generated classes.

```

<?php

//
// Class: Example
// Table: example
// Database: APG
//
// Generated by buildClass.php, written by Dick Munroe
(munroe@csworks.com)
//

include_once("SQLData/class.SQLData.php") ; // (1)
include_once("SDD/class.SDD.php") ;

class Example extends SQLData // (2)
{

    //
    // Private (or constant) variables.
    //

    var $__tableName = 'example' ; // (3)

    //
    // Constructor
    //

    function Example($_dataBase, // (4)

```

```

        $_host="localhost",
        $_login="",
        $_password="")
    {
        $this->SQLData($this->m__tableName, $_dataBase, $_host,
$_login, $_password) ;
    }

    //
    // Accessor Functions
    //

    function setId($theValue) // (5)
    {
        $this->set('id', $theValue) ;
    }

    function getId() // (6)
    {
        return $this->get('id') ;
    }

    function initId($theValue) // (7)
    {
        $this->init('id', $theValue) ;
    }

    function un_setId() // (8)
    {
        $this->un_set('id') ;
    }

    function is_setId() // (9)
    {
        return $this->is_set('id') ;
    }

    //
    // Default update selector
    //

    function needUpdateSelector() // (10)
    {
        if (($this->is_setId()))
            return "where `id` = '" . $this->escape_string($this-
>getId()) .
                "' " ;

        trigger_error("Internal Logic Error: No key data present",
            E_USER_ERROR) ;
    }

    //
    // Insert function
    //

    function insert() // (11)
    {
        $theReturnValue = parent::insert() ;
        if ($theReturnValue)
        {

```

```

        $this->initId($this->fetchLastInsertId()) ;
    }
    return $theValue ;
}

//
// Debugging Functions
//

function print_r()                                     // (12)
{
    $sdd = new SDD() ;
    print($sdd->dump($this)) ;
}
}

?>

```

Example 1- Automatically Generated Example Class

1. The various underlying components of the application generation, in particular the SQLData class from which all specific table classes are derived and the Structured Data Dumper class which is used. To make the code generated by the programs referred to here, SQLData and SDD must be installed in your PHP include path. See the PHP documentation for details.
2. Every table specific class is derived from SQLData, a class supporting generic table data storage and updating. Essentially, the table specific classes are convenience classes to make dealing with specific tables easier. Note that the first character of the table name is upper cased to make the class name.
3. This provides the binding between this class and the underlying MySQL table.
4. The constructor for the table-specific class. Since the table name is wired into the class, the remainder of the MySQL database access information must be provided when the class is instantiated. No data oriented constructors are provided, as most of the table-specific class data initialization occurs either as a side effect of accessing the table through the underlying SQLData interfaces or from interactions with a user through web forms.
5. set* member functions set the named field to a value and note that the value is now "dirty" and should be flushed to the database when the next update or insert operation is issued. A set* member function will be created for each field in the table metadata.
6. get* member functions get the named field value and return it to the caller. A get* member function will be created for each field in the table metadata.
7. init* member functions are the same as set* functions but the data is not set as "dirty" and will not be flushed to the database when the next update or insert operation is issued. An init* member function will be created for each field in the table metadata.
8. un_set* member functions delete data for a field from the underlying SQLData class. Once deleted, the field data is no longer considered in insert or update operations. An un_set* member function will be created for each field in the table metadata.
9. is_set* member functions are predicates returning true if the field has data in the underlying SQLData class. Data is stored for a field using either the set* or init* member functions. An is_set* member function will be created for each field in the table metadata.
10. The insert and update member functions (available using SQLData) both take an optional selector to indicate which record in the table should be modified. For properly designed tables with keys, it is generally possible to provide a set of default selectors, depending on which keys in the table currently have data associated with them, to be used when a selector is not

provided. The `needUpdateSelector` member function is overridden when possible to provide the default selectors.

11. For tables with indices that are `auto_increment` fields (see Figure 2), after an insert operation has succeeded the `auto_increment` fields are updated automatically by MySQL. The `insert` member function is overridden as necessary to make sure that the value of the `auto_increment` index is maintained when a new row is inserted into the table.
12. Last, but not least, debugging. The `print_r` member function dumps the content of the object (and its base objects) in a structured format that makes it easy (or at least possible) to see what's happening within the table specific object. If the execution environment is a web server, then the data is dumped in HTML format.

This simplified the job of developing the donations application enormously, and I began putting together the user interface that would use the mechanically generated table class.

Data Collection and Validation

Very shortly, it became clear that the user interface had a problem similar to that of the table classes; that is, there were one (or more) tables for which forms had to be generated. Furthermore, the data had to be syntactically and semantically verified, any errors correct by further interactions with the user, and the data stored in the appropriate tables in the MySQL database. Since I had a pattern for generating executable "stuff" from MySQL metadata, I decided to see what, if anything could be done to generate rough drafts of the forms necessary to collect data to be stored in the necessary table.

To keep the collected data from corrupting the tables ("garbage-in, garbage-out" applies here) the collected data would have to be verified. I partitioned the data verification into two distinct types:

- Syntactic
- Semantic

Verification could be done in two places, the client or the server. To improve responsiveness, syntactic validation using JavaScript (now ECMAScript) is done at the client (the web browser) and semantic validation at the server.

Examples of syntactic validation are:

- Zip codes must contain only digits and "-"s and can be either of Zip (01234) or Zip+4 (01234-5678) format.
- Telephone numbers have to be digits and must be 10 digits long.
- Required files must be non-blank.

Occasionally there would be one field that implied that others would no longer be optional and this sort of thing I defined to belong to syntactic verification.

I defined semantic validation to be "checking that data is meaningful in a given context." Examples are:

- Is the county or state name real?
- Does the city exist?
- Is the credit card number valid?

Basically, semantic verification answers the question "does the data represent reality" in the context of the application. In many cases, semantic validation can be built in by restricting the input values to a particular range, such as forcing the user to select from a list of values like country or state names.

There were three additional pieces to be examined for automation opportunities:

- User Interface

- Syntactic Validation
- Semantic Validation

User Interface

As shown in Figure 2, the table metadata has the basics, field name, data type, size of data and whether data is required (not null). Given this information, it was easy to write another program much like the buildClass.php program to construct a simple user interface using HTML forms to display and capture data and to link that interface to the client-side syntactic validation framework and to the server-side semantic validation.

* Required Field	
* first_name	<input type="text"/>
* last_name	<input type="text"/>
* company_name	<input type="text"/>
* trans_id	<input type="text"/>
trans_time	<input type="text"/>
* reference	<input type="text"/>
<input type="button" value="Save"/> <input type="button" value="New"/> <input type="button" value="Reload"/>	

Figure 3 - Generated User Interface

Figure 3 has been edited for space reasons and shows part of the user interface generated for the Example table.

It is important to remember that my goal for the FDP was not to produce a completely polished and fully functional web application solely from MySQL metadata. It was only to produce something that, with not much effort, could be turned into a "completely polished" and fully functional web application.

```
<link rel="stylesheet" type="text/css"
href="syntacticValidationFramework.css" /> (1)
```

```
<script type="text/javascript"
src="syntacticValidationFramework.js"></script> (1)
```

```
<script type="text/javascript"
src="form.Example.js"></script> (1)
```

```
<form name=data method=post action="process.Example.php"
onSubmit="return validate(this) ;"> (2)
```

```
<table name=dataTable id=dataTable border=1 cols=2>
```

```
<tr>
```

```
<td colspan=2>
```

```
* Required Field
```

```
</td>
```

```
</tr>
```

```
<tr id=errorRow> (3)
```

```
</tr>
```

```
<tr>
```

```
<td><span id="spanFirst_name" class="inline">*
first_name</span></td>
```

```
<td>
```

```
<input name="first_name" id="first_name" type=text size=40
maxlength=50 required="first_name is required" validate="return
```

```

        validateFirst_name(what)" value="">                                </td>
            (4)
    </tr>
    <tr>
        <td><span id="spanTrans_time"
            class="inline">trans_time</span></td>
        <td>
            <input name="trans_time" id="trans_time" type="text" size=10
            maxlength=10 required="" validate="return validateTrans_time(what)"
            value="">                                </td>
            (5)
    </tr>
    <tr>
        <td align=center colspan=2>                                (6)
            <button name=save id=save type=submit>Save</button>
            <button name=new id=new type=button
            onClick="window.location.search='?action=new'">New</button>
            <button name=reload id=reload type=button
            onClick="window.location.search='?action=reload'">Reload</button>
        </td>
    </tr>
</table>
</form>

```

Example 2 - Generated HTML

A quick look at the HTML generated by buildForm.php (Example 2) is instructive.

1. These are the hooks to the "syntactic validation framework" discussed more fully in the next section. Unless buildSyntacticValidation.php has been run before buildForm.php the JavaScript components of the syntactic validation will not be included.
2. This is where the syntactic validation framework is actually invoked. When a Submit button is clicked, the onSubmit code is called. The form is not actually submitted unless or until all syntactic errors have been corrected. If buildSyntacticValidation.php has not been run, the onSubmit code is omitted and no syntactic validation will be done.
3. Another hook for the syntactic validation framework. This row is where the error information (if any) will be displayed by the framework.
4. This is a typical required form field. Validation for the field is provided by the validation attribute. Note the required attribute for required fields is non-null.
5. This is a typical optional form field. Validation is still required (the validation may always succeed, of course) and is done for consistency. Note that the required attribute is the null string for optional fields.
6. The action portion of the user interface. **Save** causes the captured data to be (optionally) syntactically validated and sent to the server for further processing. **New** clears the form and starts the data capture process over. **Reload** discards any data changes and starts the data capture process over.

Syntactic Validation

The syntactic validation requires JavaScript (now known as ECMAScript). Most modern browsers support JavaScript so this requirement isn't all that restrictive, only leaving out text-only browsers such as lynx or links. It relies on the Document Object Model (DOM), defined by the World Wide Web Consortium. Unfortunately, Microsoft's Internet Explorer is not particularly compliant with the DOM. It was possible to design an adequate web-browser independent framework providing primitives to handle collection and displaying of error data, validation of required fields, and a driver to validate a form upon submission.

Since my goal was to generate most, but not all, of a web-enabled application automatically, the validation hooks had to be associated with the individual fields rather than generated monolithically. Further, not every form would necessarily need syntactic validation.

This is handled by running (or not running) `buildSyntacticValidation.php`. This program generates the JavaScript routines to do the syntactic validation for each field in the form. If `buildSyntacticValidation.php` is not run, when the form is generated by `buildForm.php` no syntactic validation will occur at the time the form is submitted.

The syntactic validation framework is provided in a separate JavaScript file, `syntacticValidationFramework.js`. Understanding the details of the DOM that allow the framework to work is left as an exercise to the reader.

```
<form name=data method=post action="process.Example.php"
  onSubmit="return validate(this) ;">
```

Example 3 - Syntactic Validation Hook

Example 3 shows the hook between the user interface (form) and the syntactic validation framework. The `onSubmit` action is taken when a submit button is clicked. A pointer to the form object requiring validation is passed to the framework, along with the contents of the form used to actually determine what validation needs to be done. If the validation framework returns false, the form's data is not sent to the server.

```
<input
  name="first_name"           (1)
  id="first_name"           (2)
  type=text
  size=40
  maxlength=50
  required="first_name is required" (3)
  validate="return validateFirst_name(what)" (4)
  value="">
```

Example 4 - Field Validation Details

The Document Object Model requires all attributes of a tag to be represented. This means that the page designer can put anything into an arbitrarily named attribute. Each field to be validated must have a unique id, an indication if the field is required, and a pointer to the routine to be used to validate the field's contents.

Example 4 shows a typical input tag with the hooks for the syntactic validation framework.

1. The name of the field received by the server during semantic validation and processing.
2. The unique id of the field. By convention it is always the same as the name field. The DOM interface is most easily used if each tag has an id by which the field can be found.
3. If the field in the database must have a value, the required field must be non-null and must contain the message to be presented to the user should the contents of the field be omitted.
4. Invocation of the validation framework routine for the field.

```

function validateFirst_name(what)
{
    //
    // Validate first_name field value
    //

    if (!isRequired(what))
    {
        return false ;
    }

    return true ;
}

```

Example 5 - Template Validation Routine

BuildSyntacticValidation.php generates a JavaScript routine identical in all but name for each field in the user interface. The validation framework calls the validation routines with a pointer to the field to be validated. Each validation routine must return either false (validation failed) or true (validation succeeded) and prepare any error text to be displayed at the end of validation.

The current syntactic validation framework can easily be modified to fit a variety of error reporting and interaction styles without modification to any of the generation code. Or something completely different can be written to meet site-specific requirements. After all, the generating code is in the public domain.

Semantic Validation

The data hits the database in semantic validation. Assuming that the form passes syntactic validation when the user presses the Save (Submit) button, the contents of the form are sent in HTTP format to the server. For the purposes of the FDP, semantic validation basically took the data from the form and put it in the databases.

The semantic validation code is generated by buildProcessForm.php. The semantic validation does not happen by default. As generated by buildProcessForm.php data was either inserted into or updated in the database and then control returned to the user for further work. Any semantic validation was considered to be custom code and to be written by hand. This is consistent with my goal of generating most but not all of the application.

The Complete Process

At this point, we had a set of tools that allowed the FDP applications to follow a very well defined data driven software development process. All of these tools were either free software, shareware, or easily developed in-house. The tools and process were:

1. phpMyAdmin	Design the database schema for the application. Database design was frequently a group interactive event, starting on a blackboard and quickly moving to a web browser or terminal. Only one database design only took more than an hour from discussion to completed design.
2. buildDatabaseConf.php	Generate the configuration file containing the information necessary to access the MySQL database.
3. buildClass.php	Create the PHP class to make it easier to manipulate the data. All the PHP code generated by the build* programs use the generated class.
4. buildSyntacticValidation	Create the individual field validation routines for the syntactic validation framework.
5. buildForm	Create the user interface.
6. buildProcessForm	Create the PHP code to store/update data in the

	database.
7. Apache, FreeBSD	Install the generated application on the development webserver.
8. Netscape, Internet Explorer, MySQL	Test the user interface/database interaction.
9. emacs, UltraEdit, Putty, WinSCP	Modify the generated application to full function and integrate with the production FDP web site.

Using this process, you can go from the completion of a database design to a prototype application in a matter of minutes.

Conclusion

The techniques associated with automatic program generation are well understood and widely used in many application generators on any number of platforms. By spending a few days writing tools, I made it much faster and easier to turn out finished applications. Experience showed that about 80% of the finished application can be automatically generated. Finished applications, fully integrated with the production FDP web site, are frequently produced in a day, with the vast majority completed in less than two days.

In all likelihood, similar productivity levels can be achieved elsewhere with site-specific versions of the tools described here.

All code developed for the FDP discussed in this paper is available from www.phpclasses.org. You will have to join www.phpclasses.org in order to download the code, but the membership is free. PHP is available from www.php.net and MySQL from www.mysql.com. These can be built and run on UNIX, Linux, and Windows platforms. Cygwin is available from www.cygwin.com if you want a UNIX-like environment for your Windows PC. If you're going to do any serious MySQL database development phpMyAdmin is a must and can be downloaded from www.phpMyAdmin.net. All are free software.

Acknowledgements

I'd like to thank Chris Sands, the director of IT for the Florida Democratic Party, for permission to publish this work. Many thanks to all the folks who supported the FDP during the 2004 campaign, lots of nights wouldn't have been possible without the free Diet Coke and junk food. I'd also like to thank Manuel Lemos, the creator of www.phpclasses.org. My job at the FDP would have been substantially more difficult if his site didn't exist. Last, but definitely not least, thanks to the many unnamed developers who have put together so much fine software and put it into the public domain and in particular to those folks who have contributed to PHP, MySQL, Apache, Cygwin, and FreeBSD. Without it the work we did in Florida during the last presidential campaign would have been impossible.

For more information

If you need more information about the work described here, contact Dick Munroe at munroe@csworks.com