IBM

Getting started with CodeIgniter

Create MVC applications quickly and easily

Level: Intermediate

Thomas Myer (myerman@gmail.com), Author, Consultant, and Web developer, Triple Dog Dare Media

26 Aug 2008

Creating a CodeIgniter application is easier than you might think. Take a guided tour through your first project: a simple Web page with a contact form.

If you're a PHP developer, it doesn't take long to make a profound discovery when it comes to your programming language of choice: Large projects can get messy.

This isn't exactly PHP's fault. Yes, the language is feature-rich, and it has just enough idiomatic elasticity to distinguish one programmer's work from another's. In this regard, PHP is similar to Perl, which is one of the reasons some people love it (and others despise it). Any experienced PHP developer who has reviewed a legacy PHP project can easily detect the work of different developers over the phases of the project—it's as though you were an archaeologist peering into a deep vault and witnessing the march of different cultures in their respective epochs.

Regardless of the styles involved or the approaches used, PHP projects of more than a few thousand lines tend to get messy in a hurry. This is mostly because they aren't uniform in structure. Some programmers create classes to organize their work, but it seems that no two programmers have the same ideas about how to write their classes. Other programmers build massive include files full of functions. Still others use huge, monolithic libraries like PEAR.

How an MVC framework can help

Until a few years ago, PHP lacked a good, functional Model-View-Controller (MVC) framework. MVC frameworks let programmers organize their code into three distinct functional areas:

- *Models* contain any and all code that relates to your database and other data structures. If you had a table called pages, you'd have a model for it and functions within that for selecting, creating, updating, and deleting records from that table, among other things.
- *Views* contain all your display and UI elements—your JavaScript code, Cascading Style Sheets (CSS), HTML, and even PHP.
- *Controllers* hold it all together. Each function in a controller represents a destination or route. If you had a destination called /about, your controller would have a function called `about()`.

If you haven't worked with an MVC framework before, these three bullet points offer hardly any clue as to the power behind this organizational scheme. Once you start thinking in terms of MVC, your perspective about and attitude toward PHP development changes radically.

For example, instead of tucking database query code in every available nook and cranny of your project, you organize all of it into models. And to select a page from your database table, you use a function from the pages model.

Similarly, if you need to update the look and feel of a certain page, you work with the views. You don't mess around with the controller. Likewise, the controller is where you add destinations and other controlling code for your application; you don't put any of that stuff in your model.

Within a day of working with any MVC framework, you'll realize that you have a system that is easy to remember and scale up as needed. If the customer needs a change next week, no problem—you can handle it. The same goes if the

request comes in next year.

Convention-over-configuration MVC

The most famous of all MVC frameworks is Ruby on Rails. It stormed the Web development beaches several years ago and captured everyone's imagination. It wasn't merely an MVC framework, but a convention-over-configuration MVC framework.

Convention over configuration means that with Rails, you set a few key configuration items (the location of your database, certain usernames and paths, for example), and the rest is handled via smart defaults that you may or may not have to tweak later.

The result isn't just well-organized code, but an incredibly fast and easy-to-use Web development environment. Those in the PHP world drooled with anticipation for something so cool and wonderful. And over the next year or two, a slew of Rails-like tools came out: CakePHP, Symfony, and many others.

Enter CodeIgniter

Finally, the good folks at EllisLab released CodeIgniter. After working with and experimenting with all the available PHP MVC frameworks, CodeIgniter has come out as the winner at many firms, mostly because it supports just enough freedom within its organizational dynamic to allow developers to work fast.

Freedom means that with CodeIgniter, you don't have to name your database tables a certain way, nor do you have to name your models after your tables. This makes CodeIgniter the perfect choice to refactor a legacy PHP application, in which you may have all kinds of crazy structures that need porting over.

CodeIgniter doesn't require a huge footprint of code (the 1.6.2 release is a slender 2.8 MB, 1.3 MB of which is user documentation you can delete), nor does it ask you to plug in to huge libraries like PEAR. It works equally well in PHP 4 or PHP 5, allowing you to create fairly portable applications. Finally, you don't have to use a templating engine to create your views—just old-fashioned HTML and PHP will do.

But enough with the introductory material—it's time to build a simple project and see where it takes you.

Install and configure CodeIgniter

The first step in any new CodeIgniter project is to download the latest package (1.6.2 as of this writing; see the Resources section). Once you download the compressed (.zip) archive and unpack it, you have a codeigniter_<*version_number*> folder that holds everything you need to get started.

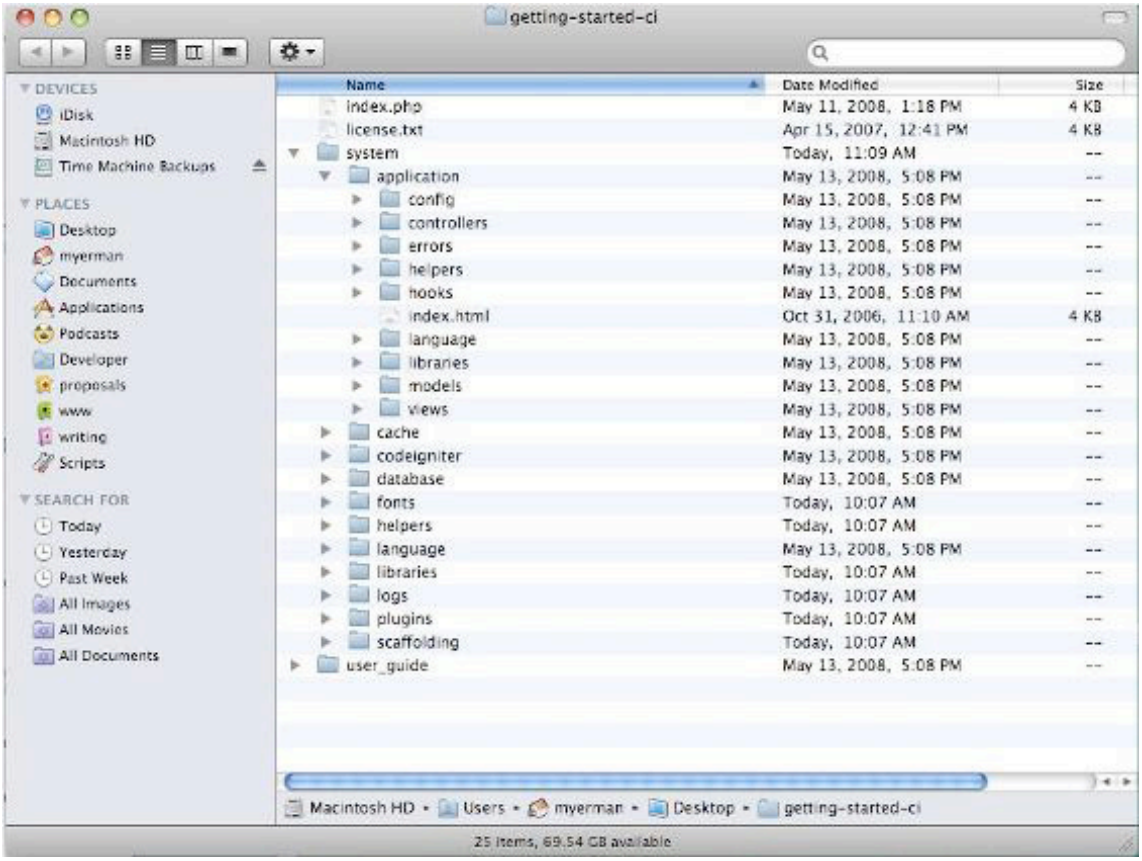
Before you make the minimal set of configuration changes you need to get started, this section takes a quick tour of CodeIgniter so you're familiar with the lay of the land.

The folder structure

When you open the CodeIgniter folder, you'll notice a folder called system. This is where all your CodeIgniter code resides. Inside that folder are a number of other folders, one of which is called application: 99.999% of the files you'll work with are in this folder. This folder is aptly named because it contains your application and everything that is part of it—the rest of the system folder contains CodeIgniter core code, libraries, and other files you shouldn't mess with.

The application folder is divided into various other folders (see Figure 1). Most of them are easy to figure out. Your models go in the models folder, the views into views, controllers into controllers, and so on. You also have folders where you store local extensions to CodeIgniter helpers and libraries, which this article doesn't discuss.

Figure 1. CodeIgniter's folder structure



For the moment, the most important folder in the system/application folder is config. Inside that folder are two files that need immediate attention: config.php and database.php.

The config.php file contains basic parameters and arguments for setting up CodeIgniter. The database.php file contains basic parameters and arguments for connecting to your database.

The only thing you have to do in the config.php file, for the moment, is set the `base_url` parameter, which is probably set to http://127.0.0.1/CodeIgniter/. Change it to match the address of the server you're working with:

```
$config['base_url']    = "http://www.example.com/";
```

Always add a trailing slash, even if you're setting up your CodeIgniter application in a subdirectory.

Next, open the database.php file, and set the connection parameters for your database server:

```
$db['default']['hostname'] = "your-db-host";
$db['default']['username'] = "your-username";
$db['default']['password'] = "your-password";
$db['default']['database'] = "your-db-name";
$db['default']['dbdriver'] = "mysql";
```

That's it. You could do other things (like set your autoload preferences and special routes), but as long as CodeIgniter knows where it is and can connect to its underlying database, you're free to get started with the coding.

Your first CodeIgniter project

Now that you have CodeIgniter installed and configured, you can build a project; it will take at most one hour of your time.

Instead of building a Hello World application, you'll create a simple Web site with CodeIgniter. It will have a home page that displays some promotional copy and a form that posts to a database table. No need to worry about look and feel—just concentrate on the parts that make sense from an application standpoint. In other words, let the aestheticians worry about the look of the thing—you just make sure that it works right and that it's finished quickly.

In CodeIgniter terms, these requirements translate into the following:

- One controller containing just a handful of functions (you can use the default Welcome controller)
- One model (along with a database table) for storing contact information
- One master view and some supporting includes

Create the database table and model

Starting with the models can help you understand the underlying database tables before you begin layering on functionality and UI. It's hard to design forms that talk to tables if you don't have some idea of what the table will hold.

In the case of the example application, you want to store contact information from a form. Well, what kind of contact information are you going to ask for? For now, stick to the basics and ask for a name, an e-mail address, a phone number, and a short note. You probably also want to store a timestamp and IP address on the back end.

The MySQL table looks like this:

```
CREATE TABLE `contacts` (
  `id` int(11) NOT NULL auto_increment,
  `name` varchar(128) NOT NULL,
  `email` varchar(255) NOT NULL,
  `notes` text NOT NULL,
  `stamp` timestamp NOT NULL default CURRENT_TIMESTAMP on update CURRENT_TIMESTAMP,
  `ipaddress` varchar(32) NOT NULL,
  PRIMARY KEY  (`id`)
) ENGINE=MyISAM;
```

Now that the table is in place, it's time to create your first model. In the system/application/models folder, create a file called mcontacts.php. Why mcontacts? It's a form of shorthand—placing an *m* before the name of the model in the filename can help you keep things organized, without having to use a longer prefix or suffix like *model_*.

All models are structured the same way:

```
class MContacts extends Model{

  function MContacts(){
    parent::Model();
  }
}
```

Notice that the class name matches the filename, and that you have to have a constructor for your class. In other words, a model is a PHP class. This means every function in the model is just a method of that class.

Once you figure that out, it's not a far leap to conclude that you need a function that will safely insert data into the contacts database table. Here's a function that does just that:

```
function addContact(){
  $now = date("Y-m-d H:i:s");
  $data = array(
    'name' => $this->input->xss_clean($this->input->post('name')),
    'email' => $this->input->xss_clean($this->input->post('email')),
    'notes' => $this->input->xss_clean($this->input->post('notes')),
    'ipaddress' => $this->input->ip_address(),
    'stamp' => $now
  );

  $this->db->insert('contacts', $data);
```

```
  }
```

It's important to note that you're taking input from a `POST` array, cleaning it up, and then storing it into the database table named contacts. Along the way, you're using various helpers to simplify your task.

For example, `$this->input->xss_clean()` untaints the data from form fields, `$this->input->post()` simplifies access to those form fields, `$this->input->ip_address()` grabs the IP address from the user's browser, and `$this->db->insert()` adds a new record to the database table.

The use of `$this->input->xss_clean()` in this context is vital—you're dealing with user input on the Web, which could be anything. Using the `xss_clean()` function is probably the least you should do, as the more careful among you might attest. Adding functionality that cuts the length of the input fields to certain sizes may be in order. For now, though, the `xss_clean()` routine keeps you safe enough.

In just a few minutes, you've created a reusable function that lets you store contact information in the database. Now, you'll move on to the controller.

Initialize the controller

In CodeIgniter, the controller is where you organize your project. Think of each function as being another page or destination in your site or application. If you have a home page, you need an `index()` function. If you have an About Us page, you need an `about()` or `about_us()` function—it depends on how you want to structure your URLs.

You can even organize your controllers into folders to create hierarchical structures. For example, in your system/application/controllers folder, you may have a folder called admin, and inside that, various controllers for each major part of your admin tool. You access those controllers (and functions) like this: http://www.example.com/admin/controller-name/function-name/.

For now, work with the default controller, which is called Welcome. It's stored as welcome.php in the system/application/controllers/ folder. When you open it, you should see the following:

```
class Welcome extends Controller {

  function Welcome(){
    parent::Controller();
  }

  function index(){
    $this->load->view('welcome_message');
  }
}
```

As you can see, the name of the class mirrors the name of the file. There's also a constructor that invokes the parent `Controller` class deep within the CodeIgniter core. So far, so good.

Next, notice the starter function called `index()`, which loads a view called welcome_message. Before you delete that function and write your own, it's important to note that this prototype `index()` function does pretty much the minimum required to display information to the application's end user.

Let's go ahead and build a new `index()` function. The first thing you need to do is load the valuable Form Helper—it assists you with the tedious task of creating the contact form.

Next, set a number of variables that can be used inside a view—that way, you can keep your application better organized. For example, you may want to set your title and headline in the controller. If you do that, you have to load the variables into the view. One of the variables you load is the name of the included view. You do that so you can set up a master view that holds all of your look and feel, along with various includes that contain your content:

```
function index(){
```

```
  $this->load->helper('form');
  $data['title'] = "Welcome to our Site";
  $data['headline'] = "Welcome!";
  $data['include'] = 'home';
  $this->load->vars($data);
  $this->load->view('template');
}
```

The `$data` array is passed into a view called template (which you create in the next step). The information inside that array can be accessed with the key names, so if you want to print out the headline, you access it with `$headline`.

Next, you create the template and home views (the latter being just an include) and finish off the controller.

Create the views

Your first view is extremely simple—it's the one called template. We'll keep it simple to show how flexible views can be. The template view is stored as template.php in system/application/views, and it initially looks like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<title></title>
</head>
<body>

</body>
</html>
```

But remember that you're passing in three variables: `$title`, `$headline`, and the name of an include in `$include`. Here's the template view again, with the additions in bold:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<title><?php echo $title;?></title>
<style>label { display:block;}</style>
</head>
<body>
<h1><?php echo $headline;?></h1>
<?php $this->load->view($include);?>
</body>
</html>
```

In the first two additions, you echo out the data you find in `$data['title']` and `$data['headline']`, respectively. Then, you use the value of `$data['include']` to load a second view. In this case, it's a view called home. (Also note that you add a bit of CSS to simplify some other things later.)

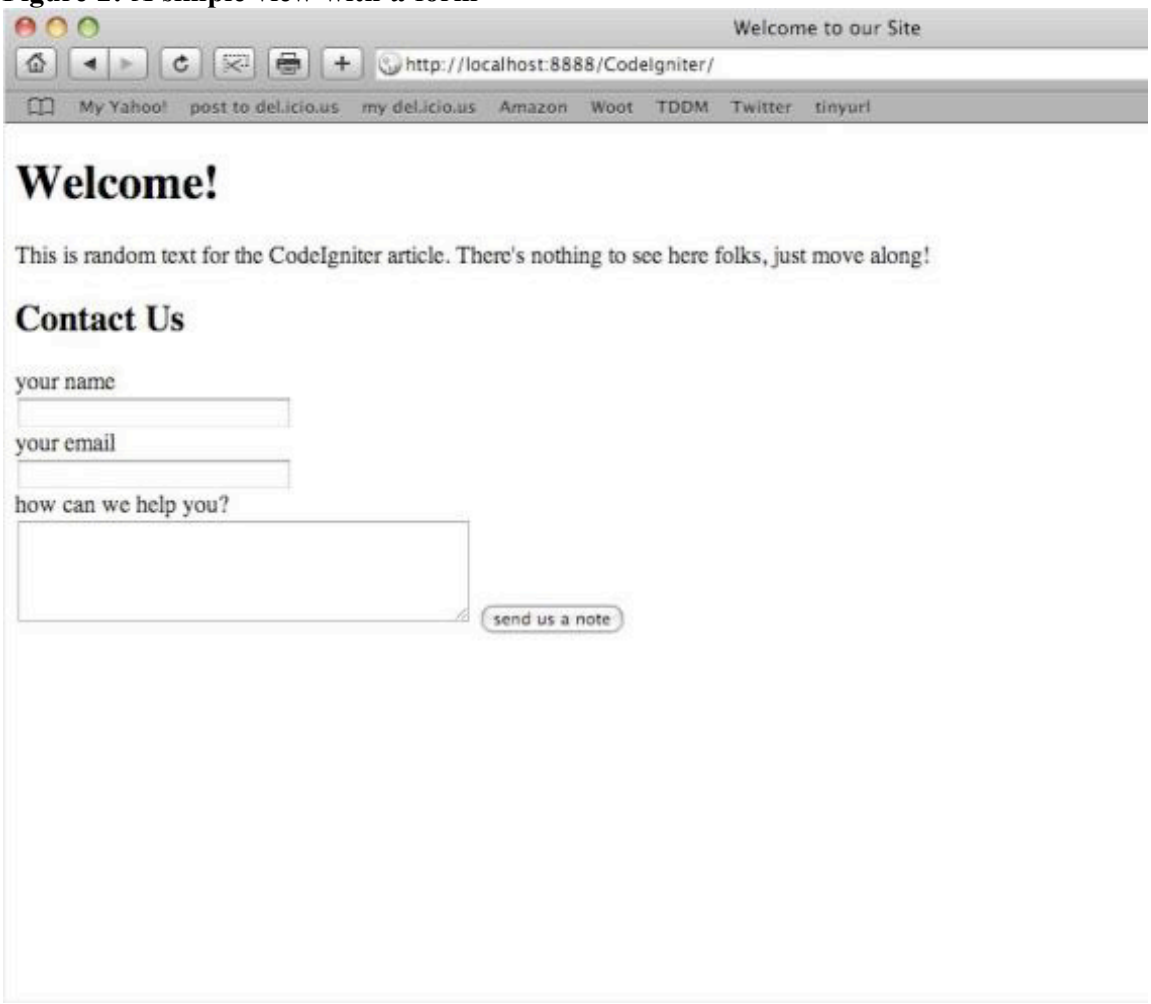
If you're going to call it, you'd better build it. Here's a simple view containing a block of text and a form that collects information from your site visitors:

```
<p>This is random text for the CodeIgniter article.
There's nothing to see here folks, just move along!</p>
<h2>Contact Us</h2>
<?php
echo form_open('welcome/contactus');
echo form_label('your name','name');
$ndata = array('name' => 'name', 'id' => 'id', 'size' => '25');
echo form_input($ndata);

echo form_label('your email','email');
$edata = array('name' => 'email', 'id' => 'email', 'size' => '25');
```

```
echo form_input($edata);

echo form_label('how can you help you?','notes');
$cdata = array('name' => 'notes', 'id' => 'notes', 'cols' => '40', 'rows' => '5');
echo form_textarea($cdata);

echo form_submit('submit','send us a note');
echo form_close();

?>
```

Figure 2 shows what all that stuff looks like once you load it in a browser.

Figure 2: A simple view with a form



Once again, you're using a valuable set of CodeIgniter shortcuts. This time, you use the Form Helper, which you loaded in the controller. The `form_open()` function allows you to open the form—it has one required argument, the destination that the form posts to. In a minute, you'll go back to your controller and add a `contact()` function to handle the form post data.

Throughout the form, you use `form_label()` to create accessibility labels, `form_input()` and `form_textarea()` to build form fields and text areas, and `form_submit()` to build an input button. Notice that with `form_input()` and `form_textarea()` (along with many other form functions), you can pass in an array of information to help you keep track of field names, ids, sizes, and other information.

Finally, close the form with `form_close()`.

Let's go back and finish the controller.

Finish the application

Now that your two views are in place, you need to go back to your controller and add two more functions. You know about the first already: It's the `contactus()` function that handles the incoming POST from the form on the home page. The second is a `thankyou()` function that serves as the eventual confirmation page for the form.

Why use the Form Helper?

You may wonder why you're taking the time to use the Form Helper. You may be incredibly skeptical: If you've been working on Web sites for a while, why do you need a helper for forms? In the end, it's all about working quickly and efficiently, and the Form Helper (and other helpers, too) removes a lot of the tedium you encounter when working with HTML.

The `contactus()` function is simple. Load the MContacts model, run the `addContact()` function inside that model, and then redirect the user to a thank-you page. Notice that to use the `redirect()` function, you must load the URL Helper.

Here is the code:

```
function contactus(){
  $this->load->helper('url');
  $this->load->model('MContacts','',TRUE);
  $this->MContacts->addContact();
  redirect('welcome/thankyou','refresh');
}
```

Here's the thankyou() function:

```
function thankyou(){
  $data['title'] = "Thank You!";
  $data['headline'] = "Thanks!";
  $data['include'] = 'thanks';
  $this->load->vars($data);
  $this->load->view('template');
}
```

And, easily enough, here's the thanks view:

```
<p>Thanks so much for contacting us. Someone will be in contact with you soon.</p>
```

You might wonder why you'd waste your time with such a small view. Why not set a variable in your controller and run it that way? You could do that, but it's always best to keep your functional components separate. That way, you don't risk getting into any trouble.

Add security

There's one more thing you ought to do. In the `contactus()` function of the Welcome controller, you run the risk of creating multiple blank records in the database—all it would take is someone who continuously loads the contact destination into their browser or via some kind of bot.

The easiest thing you can do to keep that from happening is to add a simple test in your controller. If there is `POST` data, load the model and function. If not, send them back to the home page. Here's the rewritten function:

```
function contact(){
  $this->load->helper('url');
  if ($this->input->post('email')){
    $this->load->model('MContacts','',TRUE);
    $this->MContacts->addContact();
    redirect('welcome/thankyou','refresh');
  }else{
    redirect('welcome/index','refresh');
  }
```

```
}
```

Conclusion

In under an hour, you've installed CodeIgniter, configured it, and created a Web site consisting of a home page, a form that adds information to a database, and a thank-you page.

There's more to learn, of course. For example, you can autoload the model and any helpers or libraries you need. You can make cache and performance tweaks to the application. You can add more CSS goodies to your views. And you can add an e-mail notification to the end of the database insert.

For now, you know just about everything you need to in order to get started with CodeIgniter.

Resources

Learn

- Stay current with developerWorks technical events and webcasts.
- Expand your Web development skills with articles and tutorials that specialize in Web technologies in the developerWorks Web development zone.

Get products and technologies

- Download the latest package of the CodeIgniter project (1.6.2 as of this writing).
- Read the online CodeIgniter documentation.
- Visit the CodeIgniter forums.
- Visit the CodeIgniter wiki.

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.

About the author

Thomas Myer is a technical book author, consultant, and Web developer. In 2001, he founded Triple Dog Dare Media in Austin, Texas. Triple Dog Dare Media helps companies create CodeIgniter-based publishing applications like content management, portals, and ecommerce systems. Myer is the author of *No Nonsense XML Web Development with PHP* (Sitepoint, 2004) and *Lead Generation on the Web* (O'Reilly 2007). His new book, *Professional CodeIgniter* from WROX, hit book stores in July 2008. He has also authored dozens of technical and business articles for IBM developerWorks, Amazon Web Services, AOL, Darwin Magazine, and others.

Share this....

Digg this story del.icio.us Slashdot it!