e-xact transactions

Payment Processing Technology

MOVING BUSINESS. REAL TIME. REAL SMART

They don't get IT!

- Banks do what banks do.
- Take your money.
- Use your money.
- Charge you to access your money.

We get IT!

- We build software.
- Our code works.
- Download via the Web.
- Bank Agnostic.
- 24 x 7 Test System

You get IT!

- **FREE**
- http:developer.e-xact.com
- Influence our developement

E-xact Transactions (Canada)
Suite 304 - 134 Abbot Street
Vancouver, BC, V6B 2K4
Canada

Contact Information
Toll Free: 1-877-303-9228
Ph: 604.691.1670
Fax: 604.691.1678

Web: www.e-xact.com
Email: comments@exact.com

Trading Symbol: EXZ.U
Listing: (CDNX)

noloh.® NOTABLES

by Asher Snyder & Philip Ross

There are probably a hundred good reasons or so why you should check out the NOLOH framework today, but we thought we'd save you a little time and just mention some of our favorites. NOLOH's Notables are those brave and daring features which make it sharply and uniquely stand out in the saturated world of frameworks.

REQUIREMENTS

PHP: *5.1+*
NOLOH: *1.7.568*

Related URLs:

- http://www.noloh.com/
- http://dev.noloh.com/
- http://www.jscolor.com/
- http://www.phparch.com/
- http://www.slideshare.net/noloh/
- http://www.twitter.com/noloh

Choosing a framework can be a daunting task. After all, most everything is considered a framework nowadays, albeit incorrectly, which unfortunately makes searching for one that fits your needs an over saturated nightmare. To help you on your quest, we've decided to deviate from our usually long articles to present you with a list of the most notable features of the NOLOH PHP Framework.

Before we delve into these features, I would like to quickly remind you of what NOLOH is. NOLOH, which stands for Not One Line Of HTML, is an event-driven, lightweight, on-demand, object-oriented web development platform which allows you to develop your websites or applications in a unified server-side codebase, while maintaining the agility to extend your application as you see fit, including implementing existing HTML, third party modules, and safely interacting with JavaScript.

While the above description might sound like a lot of buzz, it's the most accurate description we can provide without going into significant depth. If you would like a more detailed explanation, please read our article *"NOLOH: The Comprehensive PHP Framework"* in the May 2010 issue of php|architect.

Out of Sight, Out of Mind

As is often the case in technology, the best features are the ones you never have to deal with. This is especially true with NOLOH as some of its most powerful features are at the core of the framework enabling you to develop sophisticated applications with unprecedented speed and ease. These hidden heroes are:

- **Unified Server-Side Development** — This means that you can actually develop in a single codebase on the server. By single codebase, we mean entirely in fully object-oriented PHP/NOLOH, no HTML, JavaScript, or CSS necessary (although you can still use them if you would like). NOLOH handles all the client-server communication and lifecycle management, allowing your application to have a stateful continuous lifecycle. This means no `<FORMS>`, `$_GET`, or `$_POST`. Many of our users are unaware that the client even exists, it just works.

- **No HTML or JavaScript Necessary** — We must emphasize again that no HTML or JavaScript is necessary. This means that your application can consist entirely of PHP classes, representing objects that make up your application. This often allows for speedier and more cost effective development as it is not necessary to constantly switch between client and server frames of mind, or have a multi-tiered guru on your team.

- **Code once, deploy everywhere** — Your NOLOH application will work across all browsers and operating systems, including screen readers, mobile devices and even text-based browsers such as Lynx and Links. NOLOH has different rendering engines that output unique versions of your application specifically for the target user. This also includes an ever improving non-JavaScript renderer, which allows non-JavaScript users to browse a non-JavaScript version of your application, enabling even those users to partake in your application's information.

- **Automatic Ajax** — NOLOH is automatically Ajax. You don't have to think about it or do anything. Every action, update, and request is automatically transmitted via optimized Ajax and processed for you automatically by NOLOH.

- **Unprecedented performance both on client and server** — NOLOH is truly lightweight and on-demand. This means that only the correct and highly-optimized code is loaded for each user only when necessary. Your users will not have to sit through long loading screens to receive resources they will never use. If you're not using it at the moment, it's not loaded at that moment. A comprehensive explanation of this can be found in the November 2010 php|architect article titled "Lightweight, On-demand, and Beyond".

- **Automatic SEO** — Since NOLOH generates different code for different users, this allows NOLOH to generate a specific version of your

site for search engines. This version of your site is highly optimized for search engines containing W3C Strict validated code, along with a semantically correct skeleton, and auto-generated links to dynamically generated "pages" allowing search engines to reach content not normally easily accessible, such as modal windows, action dependent content, and sub-sections.

> Many of our users are unaware that the client even exists, it just works.

Top 10 Useable Notables

Although NOLOH has many exciting features, the rest of this article will cover the ten most notable features. Each notable feature will contain sample code demonstrating that feature.

No CLI, or Unwieldy Configuration

Unlike many other frameworks, it is not necessary to start in the command line or fill out configuration files, you can literally start right away by including NOLOH and constructing your initial **WebPage** class. Below is ALL the required code for hello world in NOLOH.

```
require_once('PATH/TO/NOLOH');
class HelloWorld extends WebPage {
   function __construct() {
      //Instantiate WebPage with Title and Alert
      parent::WebPage('Hello World in NOLOH');
      System::Alert('Hello World');
   }
}
```

Completely Object-Oriented

Your NOLOH applications are comprised entirely of objects. You can use the built-in objects, extend them, or create your own. The following demonstrates instantiating, adding, and setting the Click events of two instances of a Button object:

```
$hello = new Button('Hello', 10, 10);
$goodbye = new Button('Goodbye', 10, 50);
//Set Events
$hello->Click
   = new ServerEvent('System', 'Alert', $hello->Text);
$goodbye->Click
   = new ServerEvent('System', 'Alert', $goodbye->Text);
//Add Buttons to Show
$this->Controls->AddRange($hello, $goodbye);
```

Shifting & Animation

NOLOH has extensive built-in support for shifting, and animation, including drag and drop. Every NOLOH object has **Shifts** which allow you to specify how the object can Shift something or how the object can Shift With something. You can add as many as you like to an object for as many properties as you like. You can even chain them to create cool effects and useful behaviors (such as column shifts).

The code sample below creates two Panels, manually sets their BackColor (you can use CSS for all visual properties if you like), sets a Shift on **$obj1** so that it's now draggable, and sets a **Shift::With** on **$obj1** so that it moves with **$obj1**. As mentioned previously, you can Shift whatever you like with as many constraints as you like.

```
$obj1 = new Panel();
$obj2 = new Panel($obj1->Right + 20);
//Manually set their BackColor
$obj1->BackColor = Color::Red;
$obj2->BackColor = Color::Green;
//Set Shifts
$obj1->Shifts[] = Shift::Location($obj1);
$obj2->Shifts[] = Shift::LocationWith($obj1);
//Add Panels
$this->Controls->AddRange($obj1, $obj2);
```

We can now use NOLOH's built-in **Animate** functions to animate **$obj1**, which will in turn also animate **$obj2**, due to its **Shift With:**

```
Animate::Left($obj1, 600);
```

Comet

One of the most exciting features of NOLOH is its Comet support via NOLOH's **Listener** object. You don't need to set up a different server, go through hoops, or interact with different Comet libraries. You simply instantiate a Listener, set it to listen to a data source (e.g. database, file, function, web service) and specify a function for it to call on update. Unlike most Comet libraries, NOLOH's built-in Comet support allows for all the main transports, including short-polling, long-polling and streaming, and best of all, you can use it within the confines of your application and not externally as is usually the norm. Check out Philip Ross's talk "Comet: Pushing the Web Forward" at http://www.slideshare.net/noloh/comet-by-pushing-server-data-we-push-the-web-forward, for an in-depth explanation.

See Listing 1 for the complete **Listener** example, showing real-time Flickr photos randomly falling to their doom. Note that below is the actual listener line from the file. As you can see, it is only one to three lines depending on your definition of a line, and the rest of the code creates and animates an image from Flickr's resulting XML.

LISTING 1

```
1. class FlickrRain extends WebPage {
2.    function FlickrRain() {
3.       parent::WebPage('Flickr Rain');
4.       //Add Listener Bound to Flickr YQL
5.       $this->Controls->Add($listener = new Listener(
6.       'http://query.yahooapis.com/v1/public/yql?q=select%20source%20
from%20flickr.photos.sizes%20WHERE%20photo_id%20in%20(select%20id%20from%20
flickr.photos.recent)%20and%20label%3D%22Thumbnail%22',
7.          $this->LoadImage));
8.    }
9.    function LoadImage() {
10.      $photos = simplexml_load_string(
11.         Listener::$Data)->results->size;
12.      foreach($photos as $photo) {
13.         $url = $photo['source'];
14.         $this->Controls->Add($image = new Image(
15.            (string)$url,
16.            rand(0, $this->Width),
17.            rand(0, 200), 100, 100));
18.         Animate::Top($image, $this->Height - $image->Height, 3000);
19.         Animate::Opacity($image, Animate::Oblivion, 3000);
20.      }
21.   }
22. }
```

```
$this->Controls->Add($listener = new Listener(
'http://query.yahooapis.com/v1/public/yql?q=select%20
source%20from%20flickr.photos.sizes%20WHERE%20photo_
id%20in%20(select%20id%20from%20flickr.photos.recent)%20
and%20label%3D%22Thumbnail%22', $this->LoadImage));
```

Backwards Compatible & Completely Separated

Although NOLOH stands for Not One Line of HTML, this does not mean that you can't use your existing HTML, CSS, or JavaScript. In fact, NOLOH was built from the ground up to allow you to use existing HTML, CSS, or JavaScript cleanly and efficiently.

First, we'll cover using CSS within NOLOH. As you saw earlier, you can directly assign visual properties via an object's corresponding properties, however, you can also do this via CSS. You can use CSS in NOLOH in 3 ways, directly assigned CSS properties, CSS classes coupled with style sheets, or generally through style sheets. Of course, you can mix and match to fit your needs.

Let's begin with direct assignment. NOLOH allows you to directly assign ANY CSS property via the CSS prefix syntactic sugar. For example:

```
$object = new Panel();
$object->CSSBackground = '#3366FF';
$object->CSSBorderLeft = '1px solid black';
$object->CSSBorderRight = '1px solid green';
```

Next, we can add a style sheet directly to our application via WebPage's CSSFiles property. Note that you can call the following line from anywhere within your application in any class. If adding to **CSSFiles** within your WebPage class, feel free to use **$this**, instead of NOLOH's **That()** Singleton sugar.

```
WebPage::That()->CSSFiles->Add('styles.css');
```

Finally, once we have a style sheet added, we can assign classes to our object directly via the **CSSClass** property.

```
//styles.css
.SomePanel {
   background: #3366FF;
   border: 5px dotted green;
}
//Then in one of our NOLOH classes.
$object->CSSClass = 'SomePanel';
```

We can even add multiple CSS classes via the **CSSClasses** property, via **Add**, or **AddRange** (and remove them via the corresponding remove functions). Please assume that we have the corresponding CSS classes added to our style sheet.

```
$object->CSSClasses->Add('SomePanel');
$object->CSSClasses->AddRange('GreatPanel', 'BestPanel');
```

Now, let's move on to integrating your existing

HTML. You can do this in two ways, via NOLOH's **MarkupRegion** and NOLOH's **RichMarkupRegion**. Using NOLOH's **MarkupRegion** is very simple, you just instantiate the object, similar to any other object, and specify a string of HTML, or a path to a file.

```
$homeCopy = new MarkupRegion('Content/home.htm');
```

We don't simply end there, we also have something called a **RichMarkupRegion**. A **RichMarkupRegion** not only allows you to display your existing HTML, it also allows you to have your HTML interact with your NOLOH application in a completely clean and separated manner, eliminating any programming logic from your static markup files and leaving the ultimate decisions to the developer. For example, consider the following **prose.htm** Markup file:

```
//prose.htm
<p>Yes. You gave me a <a>dollar</a> and some candy.
We don't have a brig. I daresay that Fry has discovered
the <a>smelliest</a> object in the known universe!</p>
```

We can specify **RichMarkupRegion** to use this file similar to **MarkupRegion**.

```
$activeCopy = new RichMarkupRegion('Content/prose.htm');
```

In this case, **RichMarkupRegion** will behave similarly to **MarkupRegion**. However, we can now specify what's known as an **Eventee** to our text. Eventees have a **Keyword** and **Value** pair that make up a **descriptor**.

```
//prose.htm
<p>Yes. You gave me a
<n:a descriptor='prose:dollar'>dollar</n:a> and some
candy.
We don't have a brig. I daresay that Fry has discovered
the <n:a descriptor='prose:smelliest'>smelliest</n:a>
object in the known universe!</p>

//Somewhere in your NOLOH class
$activeCopy = new RichMarkupRegion('Content/prose.htm');
```

However, you'll notice that this produces the same output as before. This is because we didn't specify that anything should happen. Now, let's assign each Eventee an event that logs the Value of the Eventee.

```
$activeCopy = new RichMarkupRegion('Content/prose.htm');
foreach($activeCopy->Eventees as $eventee) {
   $eventee->Click
      =new ServerEvent('System', 'Log', $eventee-
>Value);
}
```

Let's take a moment to think of the implications of this. We can now dynamically specify what we want to happen and when it should happen without having to specify any programming code in our markup. In the following example, only Eventees with the value **smelliest** will do anything.

```
$activeCopy = new RichMarkupRegion('Content/prose.htm');

foreach($activeCopy->Eventees as $eventee) {
   if($eventee->Value == 'smelliest')       {
      $eventee->Click
         = new ServerEvent(
            'System', 'Log', $eventee->Value);
   }
}
```

Please note that if you don't want to manually add namespaces or descriptors you can also access your MarkupRegion's elements by the corresponding XPath. You can also substitute your Markup's elements for real NOLOH object's using RichMarkupRegion's **Larva** concept.

Use Third-Party Scripts and Widgets with Ease

Continuing the spirit of the previous feature, we'll now show you how you can easily use your existing JavaScript in your applications. NOLOH has a very powerful class called **ClientScript**. ClientScript allows you to add, queue, and call existing JavaScript. While we won't cover all of ClientScript, we will cover the basics and the basic optional values. Please see the ClientScript API at http://dev.noloh.com/#/api/ClientScript for an extensive list of its functions and options.

```
//Adds JavaScript code to be run immediately on the
client
ClientScript::Add('alert("blah")');
//Adds the JavaScript myscript.js to your application
ClientScript::AddSource('scripts/myscript.js');
//Auto maps and synch an Object's client property
ClientScript::Observe($object, clientValue,
[serverValue])';
//Queues a JavaScript function until the object is added
ClientScript::Queue($this, 'alert', 'blah');
//Sets an Object's client property to some value
ClientScript::Set($object, 'CustomValue', 10);
```

Please note that all the above have **Race** variants where applicable that allow you to bind them to a client conditional, this allows you to avoid problems related to race conditions. NOLOH's ClientScript functions automatically convert any arguments to the proper client format for use. You can even pass in JavaScript objects as arguments, along with raw strings, and even closures.

Now that we know the basics, we can use the

above to easily wrap a third-party JavaScript widget and create a NOLOH module we can use within our applications. For example, the following is a wrapper to the popular **JSColor** module. You'll see that if we visit the documentation at http://jscolor.com/try.php#manual-binding, it suggests you add the following line to your client, in addition to the standard HTML:

```
var myPicker = new jscolor.color(document.
getElementById('myField1'), {})
```

Rather than adhere strictly to those directions, we can easily wrap this module using the above ClientScript functions. We can simply extend a TextBox, *Add* the JSColor source, and *Queue* a minor variation of the above line (_N' is equivalent to **document.getElementById**), and that's it.

```
class JSColor extends TextBox {
   function JSColor($left=0, $top=0) {
      parent::TextBox($left, $top);
      ClientScript::AddSource('jscolor/jscolor.js');
      ClientScript::Queue($this,
         "new jscolor.color(_N('$this'), {});");
   }
}
```

We now have a fully instantiable JSColor widget we can use like any other NOLOH object, for example:

```
$colorPicker = new JSColor();
$this->Controls->Add($colorPicker);
```

We'll leave it as an exercise to the reader to map the JSColor's Text or client value to a new **SelectedColor** property.

Bookmark & Search Engine Friendly

NOLOH applications are fully bookmark-able via NOLOH's **URL** class. This also has the added benefit of automatically generating links for traditionally difficult to spider content. To use, simply call **URL::SetToken()** during a normal course of events.

```
function LaunchSection($section)
{
   URL::SetToken('section', $section);
   $newSection = new Section($section);
   ...
}
```

Similarly, use **URL::GetToken()** to retrieve the value of the token during the normal course of events. Please note that in the following example, we use the optional second parameter specifying a default if no token is found.

```
//Assuming we have a Section class that takes in a
section
$activeSection
   = new Section(URL::GetToken('section', 'home')));
```

Data::$Links

NOLOH has extensive support for the most popular databases. In addition to this support, NOLOH **Data::$Links** allows you to easily access your database and safely query your data. The following instantiates a **DataConnection** and stores it in a globally accessible **Data::$Link MyDB**. Please note that we can do this for as many databases as we like.

```
Data::$Links->MyDB
   = new DataConnection(Data::Postgres, 'mydb_name');
```

We can now query our database via Data::$Links ExecSQL, ExecView, and ExecFunction functions.

```
$users = Data::$Links->MyDB
   ->ExecSQL('SELECT * FROM users');
//
$users = Data:::$Links->
   ->ExecView('v_get_all_users');
//
$users = Data:::$Links
   ->ExecFunction('sp_get_users', 'NY');
```

We can even query SQL with replacement parameters. **Data::$Links** will automatically safeguard any parameters you use to prevent SQL injection.

```
$users = Data:::$Links->MyDB
   ->ExecSQL('SELECT * FROM users state = $1', 'NY');
```

Similarly, I can use **Data::$Links** and its **CreateCommand** function to create and store a command to be passed for execution later:

```
$command = Data::$Links->MyDB
   ->CreateCommand(Data::SQL, 'SELECT * FROM users');
$command = Data:::$Links->MyDB
   ->CreateCommand(Data::SQL, 'SELECT * FROM users state
= $1', 'NY');
$command = Data:::$Links->
   ->CreateCommand(Data::View, 'v_get_all_users');
$command = Data:::$Links
   ->CreateCommand('sp_get_users', 'NY');
```

See our **Data::$Links** article at http://dev.noloh.com/#/articles/Data-Links/ for more detailed information and options.

Great Syntax

We consistently get feedback from our users that they find NOLOH easy, intuitive, and inexplicably fun to code in, and this is because we place such great emphasis on the importance of language design. This includes consistent notational schemes, descriptive names for classes, functions, and properties, very sensible class inheritance, and the liberal use of syntactic sugars. Some of these we have already mentioned (like the CSS sugar) and some of these we'll showcase here, but the true wealth and power of NOLOH's syntax cannot be covered by just one section.

First of all, there is an interchangeable use of properties and functions via the **Get** and **Set** sugars. This allows one to define their own properties in their classes (or overload any of NOLOH's existing properties) that will be used in the familiar way by users but that actually execute code (e.g., for the purposes of validation) in addition to the ordinary setting of class variables.

This is how a class with custom properties may be defined. Notice the Get and Set prefixes in the method names.

```
class Purchase extends Object {
   private $Price;
   function GetPrice() {
      return $this->Price;
   }
   function SetPrice($value) {
      if(is_numeric($value)) {
         $this->Price = $value;
      }
      else {
         System::Alert('Error: Invalid Price given');
      }
   }
}
```

This is how a class with custom properties may be used. Notice the ordinary and natural property notation instead of method calls.

```
$order = new Purchase();
/* Calls the corresponding Set method, validates
correctly,
 and sets internal variable to 42.*/
$order->Price = 42;
// Calls the corresponding Get method, thereby logging
42.
System::Log($order->Price);
/* Calls the corresponding Set method, fails validation,
 and alerts an error.*/
$order->Price = 'Hark, a vagrant!';
```

Another very convenient syntactic sugar related to setting properties is the cascading feature. By using the **Cas** prefix, several properties can be set consecutively without the need to keep re-referencing the object. This is especially useful when initializing some objects since quite often, multiple assignments need to be made.

```
/*Equivalent to a block of 4 statements individually
 setting the Left, Top, Text, and Click properties of
 $someObject.*/
$someObject->CasLeft(0)
   ->CasTop(0)
   ->CasText('Lorem ipsum, baby')
   ->CasClick(new ServerEvent($this, 'DoSomething'));
```

Because dealing with special collections of things is so important and commonplace in programming, we will cover the syntactic sugars that deal with them next. As one might naturally expect, these collections are treated like arrays so you don't have to remember two separate sets of syntaxes if you don't want to.

For example, we have seen how to use the **Add** method on an **ArrayList** before (e.g., a WebPage's **Controls** property), but did you know that the following are all also possible just like they are for PHP's native array type?

```
// Equivalent to an Add call
$this->Controls[] = new Label();
/* Adds to a particular index, and if it was previously
 used, whatever was in it is replaced*/
$this->Controls['somewhere'] = new TextBox();
//Retrieves from a particular index.
System::Log($this->Controls[0]);
// Iterates through all Controls and logs them
foreach($this->Controls as $control) {
   System::Log($control);
}
```

These syntaxes are not restricted to merely **ArrayLists**, but also work for other collections such as Groups or Events. For example, one may define multiple Event handlers as follows:

```
// Tells the Click to call some DoThis method
$this->Click = new ServerEvent($this, 'DoThis');
/* Tells the Click handler to ALSO call some DoThat
 method by appending another ServerEvent*/
$this->Click[] = new ServerEvent($this, 'DoThat');
```

There is even an **All** syntactic sugar to allow a developer to easily iterate over some collection and do the same kind of operation (e.g., setting a property or calling a method) on each of its elements.

```
/* Individually sets each child's Layout
property to Layout::Relative*/
$this->Controls->AllLayout = Layout::Relative;
/* Individually calls the Leave method on each child
```

```
(which is equivalent to calling Clear on the ArrayList)*/
$this->Controls->AllLeave();
```

Another interesting syntactic sugar is allowing methods to be fetched (in the form of **ServerEvents**) as if they were themselves properties, just like in functional programming languages. This sometimes allows for more readable code than having to instantiate ServerEvents manually.

```
class Sample extends WebPage {
   function Sample() {
      parent::WebPage('Sample');
      $button = new Button('Click here');
      /* Use of the syntactic sugar in order to not have
      to instantiate ServerEvent manually*/
      $button->Click = $this->Greet;
   }
   function Greet() { System::Alert('Good morning!'); }
}
```

That's just to give you a little taste of NOLOH's extensive and very powerful syntactic sugars. Pretty sweet, huh?

Fully Extensible

NOLOH is fully extensible allowing you to create your own objects and even override default functionality. In the following example, we create our own **Panel** with a custom Text property.

```
class CustomPanel extends Panel {
   private $Title;
   function __construct($left=0, $top=0, $width,
$height){
      parent::Panel($left, $top, $width, $height);
      $this->BackColor = Color::Red;
      $this->Controls->Add($this->Title = new Label());
   }
   function GetText()       {return $this->Title->Text;}
   function SetText($text) {$this->Title->Text = $text;}
}
```

As you can see this Panel's **Text** property will get and set the **Text** of its **Title** Label. If we instantiate and add a new CustomPanel and set and **Log** its Text, we can expect the Label's Text to be affected and displayed appropriately.

```
$customPanel = new CustomPanel(10, 20, 100, 200);
$customPanel->Text = 'php|architect';
//Add $customPanel
$this->Controls->Add($customPanel);
//Log Text of CustomPanel which returns Title's Text
System::Log($customPanel->Text);
```

Try It Out

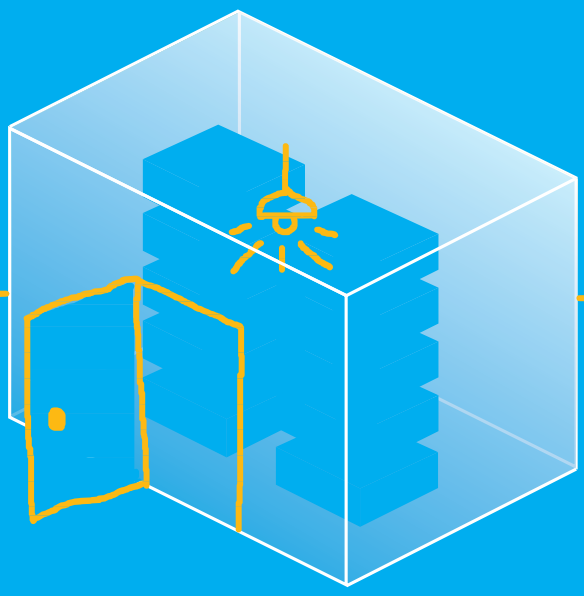
So there you have it. We hope you enjoyed reading through the above list of NOLOH's notable features. Hopefully you now see how NOLOH is a truly different PHP framework and how it might fit into your development plans. Please feel free to try any of the above features in your very own free, hosted development sandbox. Sign up for one at http://www.noloh.com today and make NOLOH's Notables work for you.

ASHER SNYDER is a co-founder of NOLOH, the company behind the NOLOH PHP Framework. A "technological polymath," he has extensive experience working with a vast number of programming languages and development methodologies from the desktop to the Internet to scaling servers and architecting advanced databases. When not working, talking, or writing, he contributes to open-source and the causes he believes in. Contact: asnyder@noloh.com.

PHILIP ROSS was born in Kiev, Ukraine. He is a co-founder of NOLOH and plays a crucial role in developing the NOLOH kernel. Among his extensive list of contributions, he was behind NOLOH's intuitive and advanced Comet functionality. Phill, as he prefers to be called, is well-versed and experienced with numerous programming languages and paradigms and particularly enjoys the subject of their design philosophies. Outside of development, his interests mainly lie in the foundations of mathematics - including set theory, logic, and related topics - and involving himself with research of alternate systems of set theory.

NEXCESS

beyond hosting.



HOSTING THAT'S DOWNRIGHT

DEDICATED.

Not robots, we're staffed by actual devoted human beings. At NEXCESS, we've focused on one thing since day one: superior hosting solutions. Today, our focus hasn't changed, but our products, our services and our facilities have. We are committed to providing reliable, scalable, dedicated hosting services to clients of all shapes and sizes. Come find out how we can take your hosting to the NEX level.

RELIABLE | DEDICATED | TANGIBLE | SCALABLE | WWW.NEXCESS.NET