HACKERMONTHLY Issue 35 April 2013

The Absolute Beginner's Guide to Arduino

Now you can hack on DuckDuckGo

DuckDuckHack

Create instant answer plugins for DuckDuckGo

duckduckhack.com

Curator
Lim Cheng Soon

Contributors
Martin Legeer
Andrew Chalkley
Grant Mathews
Bryan Kennedy
Patrick Wyatt
Pete Keen
Craig Kerstiens
Amber Feng
Alex Baldwin
John Biesnecker
Rachel Kroll

Proofreaders
Emily Griffin
Sigmarie Soto

Ebook Conversion
Ashish Kumar Jha

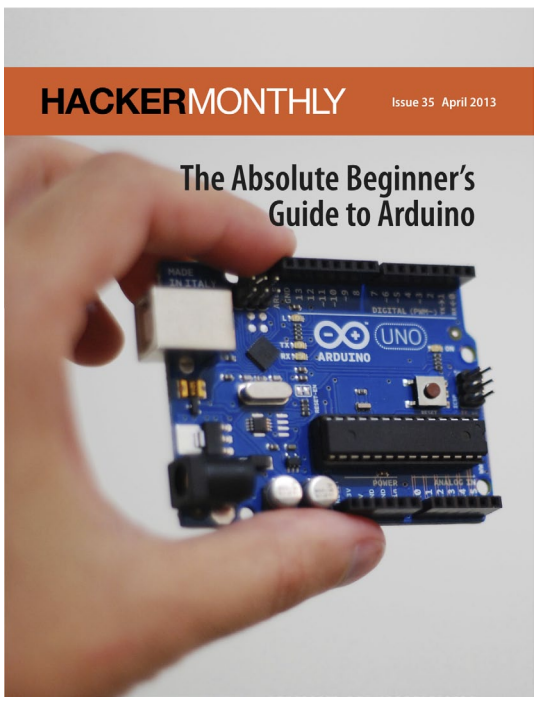
Printer
MagCloud

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format. For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.



Cover Photo: Beraldo Leal [flickr.com/photos/beraldoleal/6297076850]

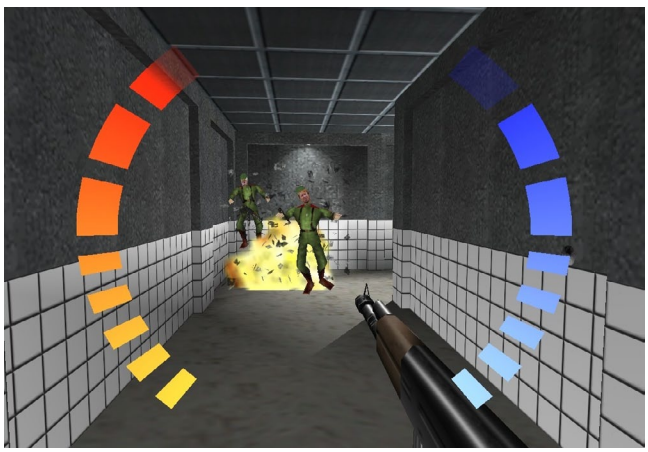
Contents

FEATURES

PROGRAMMING

Arduino Photo: Pete Prodoehl
[flickr.com/photos/raster/5933659066/]

SPECIAL



For links to Hacker News dicussions, visit *hackermonthly.com/issue-35*

How I Created a Matrix Bullet Time-Style Rig

By MARTIN LEGEER

Back in March, a client for whom I've done some light consulting work asked me if it was possible to capture a 360-degree image that can be rotated afterwards. I said of course, but I didn't think that much about the consequences — it's a project that would wake me up at night for the next few months.

Everything was fine until the moment he showed me the room I was supposed to make the rig in — it was his villa.

This type of project is generally done in warehouse-sized spaces, and there are good reasons for that (lighting being one of them). Well, this was a garage about 130-square-feet in area and a ceiling about 8 feet tall. My first reaction was to laugh for a little while. Then I asked if he was serious. I saw from the look on his face that he really did want me to build the rig there, so my next reaction was, "I have to think about it."

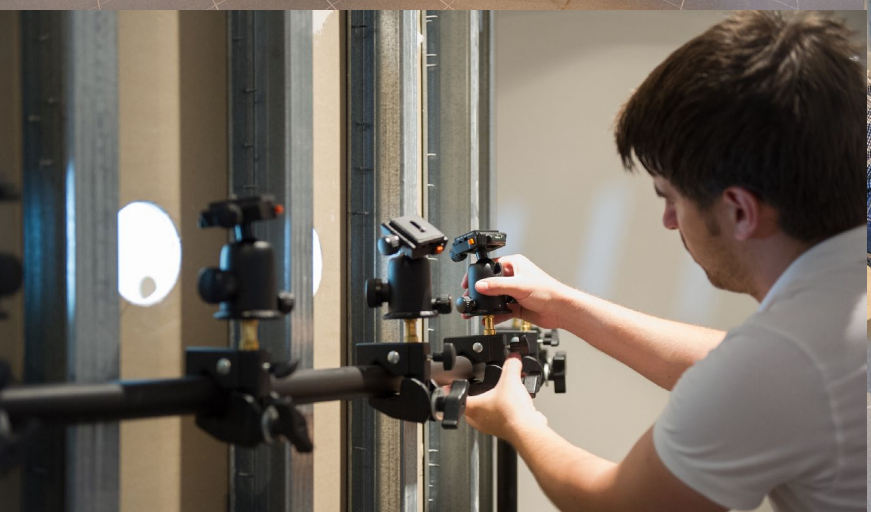
Don't get me wrong, I'm up for any badness, but this was beyond crazy and bananas.

The next day, we sat down in a restaurant and started talking about this a little more — what his vision was, what look he wanted to achieve, and so on. What I ended up with was that he would like to have a shadowless photo on a white background with maybe some contrast in there, but definitely shadowless.

This was a bit funny because the only thing I know that makes contrast (light-wise, not color-wise) is a shadow. But sure, let's make history.

Another request he made was to make it as easy as possible for further retouches, since he planned to use the rig extensively.

I took a few days to think about what direction I would go with the project — things like lighting the subject, the background, the fluency between the photos — the distance between the cameras, subject, camera count — just about everything. I've seen quite a few results from "bullet time" systems all around the world, so I knew what things and results I would like to avoid, especially with my white background.



I came up with some ideas, but just to be sure, I called my friend Daniel over (he's pretty much at the same technical level as I am) to have somebody to bounce ideas off. From that moment on, the concept just grew.

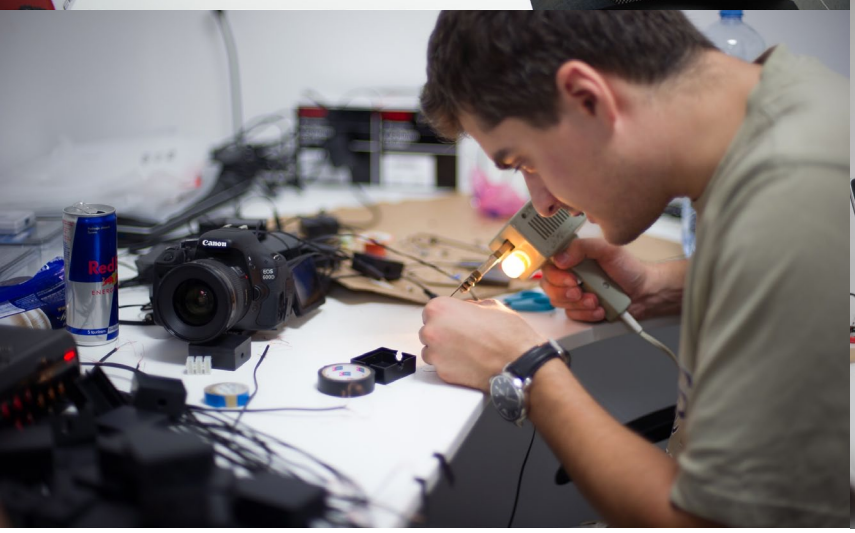
The next day, I called the client and accepted the challenge. I asked for an advance before putting everything together, making phone calls, etc. just to be sure there would be no trouble afterwards. I got the advance the very same day (to my surprise) and then we got started.

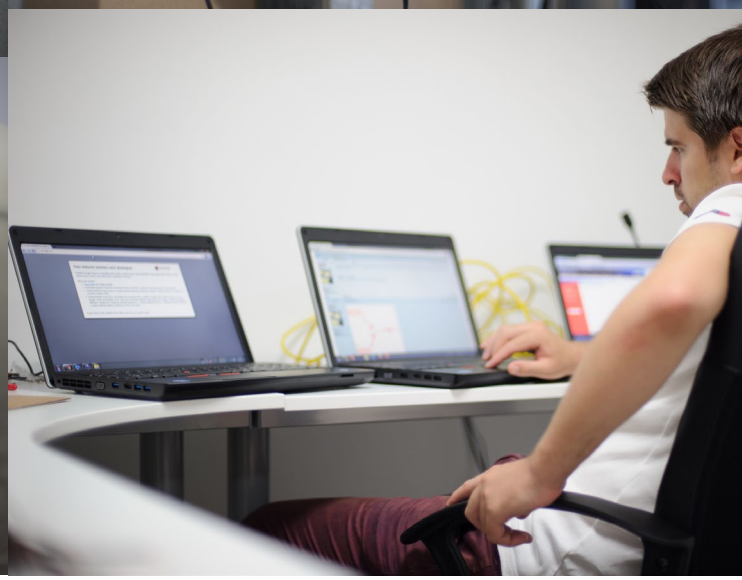
Since we had so little space to make the rig on (mainly due to the low ceiling), my initial idea was to put up some kind of a diffusing tent around the subject and light it from the outside with big octas around. However, we moved on from this idea because there were so many issues we had to address (possible flares, the quality of the image due to the lighting, how to spread the light evenly from the close distance, tension of the diffusing textile, the possibility of someone damaging the textile, etc.).

I called another colleague and we came up with the idea of making a cylinder, which we could paint from the inside and repaint if necessary in the future. The only problem we had to solve with this new design was how to light it from within. Since the request was to have a subject on a white background, we decided to build it completely in white so that the light would bounce around like crazy (eliminating shadows).

We started putting together the precise lighting setup. We wanted to give the client the option of using other styles of lighting for non-shadowless photos, so we arranged 6 lights on the ceiling in a circular formation.

For those of you wondering about the brand of the lights, I normally shoot with Profoto, but I chose to go with Elinchrom this time (to save some money). Elinchrom satisfies the requirements for professional use, since it has such a consistent amount of light, consistent color temperature, etc.





So that was step one. Step two was putting together a list of the gear we needed to buy. You wouldn't believe how much trouble it was to simply find a seller who was able to send us 50 cameras and prime lenses to the middle of Europe. It was unbelievable. If it were Canon 5D MKIIIs, Nikon D800s, or something of that class, that would be understandable, but Canon 600Ds? Come on!

We chose the cheaper Canon DSLRs over other brands simply due to budget constraints.

I originally wanted to go for Nikon (since it's the brand I shoot with), but we had trouble solving certain issues with Nikon technicians. We had to fire all 50 cameras at once and transfer all the data off the cameras, and there is currently no software on the market (that I am aware of) that can do the job (at least at the time we did this project). Maybe there is, but I couldn't find it.

There were potential programs we could use, but they were written for Linux. I also could have had programmers around me write the software, but that would have taken weeks or months, which we didn't have.

We eventually found a single program for Canon DSLRs, which turned out to be a huge waste of money. The trial version had limited options, so we had to buy the full software. After we did, we found out that although the software is able to trigger all the cameras at once, it has a slight delay between the cameras — like 1/5s between each camera.

This was a big issue for us since we needed to sync the flash between the 50 cameras as well (we can't light it with other types of lights, lest we fry the person in the cylinder).

Just like Leonardo DiCaprio in Inception, we had to go deeper. Finally, we reached the best and simplest solution. Since each camera can be fired via a cable trigger, we created a net of cables that does just that — triggers all the cameras at once. Two buttons are located at the end of the net: the focus/wakeup and trigger buttons. Voila! Firing problem solved.

The software allows us to download the RAW images onto computers (there has to be 4 laptops since the software can operate only up to 14 or 16 cameras and there are also bandwidth issues).

That's pretty much the entire build process. Of course, I simplified quite a few things to not bore you with my thoughts throughout the whole project, but all in all, I hope you enjoyed the ride. I know I did, and I would love to do it again — perhaps in a larger space.

The last thing I have to say will be a disappointment to many of you: I don't have any resulting images to share with you due to the client's request to not share any sample photos outside. Perhaps in the future some of his photos will begin to appear on the web. ■

Martin Legeer is a photographer who went from shooting events for companies like AXE, RedBull, Ferrero to commercial photography. He is trying to make his mark in fashion industry and taking some challenges along the way.

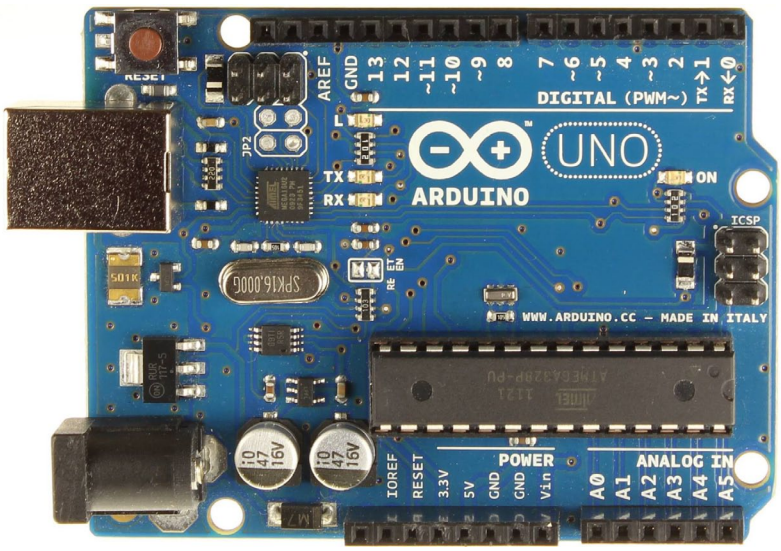
Reprinted with permission of the original author.
First appeared in *hn.my/bullet* (petapixel.com)

stripe

Accept payments online.

The Absolute Beginner's Guide to Arduino

By ANDREW CHALKLEY



Over the Christmas break from work I wanted to learn something new.

I've been eyeing up Arduino for some time now, and for Christmas I got an Arduino UNO R3 board.

What is Arduino?

> *Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It is intended for artists, designers, hobbyists, and anyone interested in creating interactive objects or environments.*
> *Source: arduino.cc*

Microcontroller

Arduino is a microcontroller on a circuit board which makes it easy to receive inputs and drive outputs.

A microcontroller is an integrated computer on a chip.

Inputs

Some examples of inputs would be a temperature sensor, a motion sensor, a distance sensor, a switch, and so forth.

Outputs

Some examples of outputs would be a light, a screen, a motor and so forth.

TL;DR

Arduino is a small computer that you can program to read and control electrical components connected to it.

Obtaining an Arduino Board

There are several online distributors that stock Arduino boards.

Often boards are bundled up with starter kits. Kits include a wide variety of inputs, outputs, resistors, wires and breadboards. Breadboards are solderless circuit prototyping boards that you can plug wires and components into.

Arduinos come in different flavors. Most people starting off go for the UNO board. Its current revision is the third, hence the R3 listed by stockists.

Most enthusiasts use sites like Adafruit [adafruit.com] and Element14 [element14.com].

You can even pick one up from your local RadioShack.

If you're just getting a single Arduino board or starter kit, be sure you have a USB A-to-B cable. Most, if not all, starter kits come with the USB A-to-B cable. Most printers have this type of interface so you may have this cable already lying around. You need the cable to program the device, so it's best to double check when ordering.

Programming Arduino

For the example I'm showing, you'll only need the Arduino UNO R3 board itself and the required USB cable to transfer the program from your computer to the board.

On the board left of the Arduino logo there's an LED, short for Light Emitting Diode, a small light, with the letter **L** next to it.

We're going to switch it on and off and then look into making it blink on and off for 2 seconds at a time.

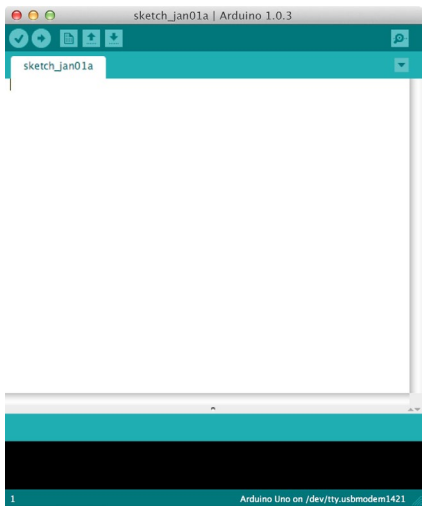
When you first plug your USB cable into your Arduino and your computer, you may notice that this LED is blinking. Not to worry! It's the default program stored on the chip. We're going to override this.

The USB cable powers the device. Arduinos can run standalone by using a power supply in the bottom left of the board. Once you're done programming and don't require it to be constantly connected to your machine, you can opt to power it separately. This is entirely dependant on the use case and circumstances you want to use the device in.

Download Arduino Software

You'll need to download the Arduino Software package for your operating system from the Arduino download page [hn.my/adl].

When you've downloaded and opened the application you should see something like this:



This is where you type the code you want to compile and send to the Arduino board.

The Initial Setup

We need to setup the environment to **Tools** menu and select **Board**.

Then select the type of Arduino you want to program, in our case it's the **Arduino Uno**.

The Code

The code you write for your Arduino are known as sketches. They are written in C++.

Every sketch needs two *void type* functions, `setup()` and `loop()`. A void type function doesn't return any value.

The `setup()` method is run once just after the Arduino is powered up and the `loop()` method runs continuously afterwards. The `setup()` is where you want to do any initialization steps, and in `loop()` you run the code you want to run over and over again.

So, your basic sketch or program should look like this:

```
void setup()
{

}

void loop()
{

}
```

Now that we have the basic skeleton in place, we can do the **Hello, World** program of microcontrollers, a blinking an LED.

Headers and Pins

If you notice on the top edge of the board there's two black rectangles with several squares in it. These are called headers. Headers make it easy to connect components to the Arduino. The places where they connect to the board are called pins. Knowing what pin something is connected to is essential for programming an Arduino.

The pin numbers are listed next to the headers on the board in white.

The onboard LED we want to control is on pin 13.

In our code above the `setup()` method let's create a variable called `ledPin`. In C++ we need to state what type our variable is beforehand, in this case it's an integer, so it's of type `int`.

```
int ledPin = 13;

void setup()
{

}

void loop()
{

}
```

Each line ends with a semicolon (;).

In the `setup()` method, we want to set the `ledPin` to the output mode. We do this by calling a special function called `pinMode()` which takes two variables, the first the pin number, and second, whether it's an input or output pin. Since we're dealing with an output, we need to set it to a constant called `OUTPUT`. If you were working with a sensor or input it would be `INPUT`.

```
int ledPin = 13;

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{

}
```

In our loop we are going to first switch off the LED to make sure our program is being transferred to the chip and overriding the default.

We do this by calling another special method called `digitalWrite()`. This also takes two values, the pin number and the level, `HIGH` or the on state or `LOW` the off state.

```
int ledPin = 13;

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  digitalWrite(ledPin, LOW);
}
```

Next we want to compile to machine code and deploy or upload it to the Arduino.

Compiling the Code

If this is your first time you've ever compiled code to your Arduino, before plugging it in to the computer, go to the **Tools** menu, then **Serial Port** and take note of what appears there.

Here's what mine looks like before plugging in the Arduino UNO:

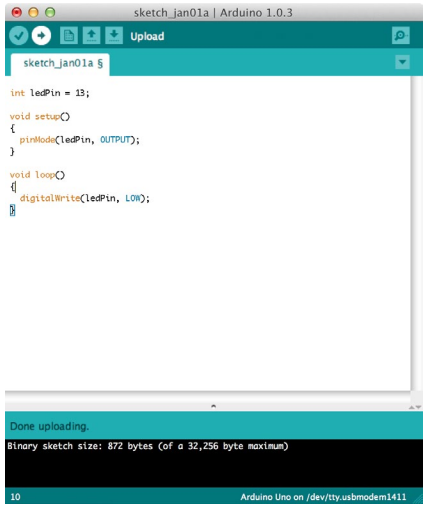
```
/dev/tty.Bluetooth-PDA-Sync
/dev/cu.Bluetooth-PDA-Sync
/dev/tty.Bluetooth-Modem
/dev/cu.Bluetooth-Modem
```

Plug your Arduino UNO board into the USB cable and into your computer. Now go back to the **Tools > Serial Port** menu and you should see at least 1 new option. On my Mac 2 new serial ports appear.

```
/dev/tty.Bluetooth-PDA-Sync
/dev/cu.Bluetooth-PDA-Sync
/dev/tty.Bluetooth-Modem
/dev/cu.Bluetooth-Modem
✓ /dev/tty.usbmodem1411
/dev/cu.usbmodem1411
```

The `tty` and `cu` are two ways that computers can talk over a serial port. Both seem to work with the Arduino software so I selected the tty.* one. On Windows you should see COM followed by a number. Select the new one that appears.

Once you have selected your serial or COM port, you can then press the button with the arrow pointing to the right.



Once that happens you should see the **TX** and **RX** LEDs below the **L** LED flash. This is the communication going on between the computer and the Arduino. The **L** may flicker, too. Once this dance is complete, your program should be running, and your LED should be off.

Now let's try to switch it on using the `HIGH` constant.

```
int ledPin = 13;

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  digitalWrite(ledPin, HIGH);
}
```

Press Upload again, and you should see your LED is now on!

Let's make this a little more interesting now. We're going to use another method called `delay()`, which takes an integer of a time interval in milliseconds, meaning the integer of 1000 is 1 second.

So after we switch the LED on, let's add `delay(2000)`, which is two seconds, then `digitalWrite(ledPin, LOW)` to switch it off and `delay(2000)` again.

```
int ledPin = 13;

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  digitalWrite(ledPin, HIGH);
  delay(2000);
  digitalWrite(ledPin, LOW);
  delay(2000);
}
```

Press Upload and you should see your LED blinking!

What next?

The Arduino platform is an incredibly easy and versatile platform to get started with. It's open-source hardware, meaning that people can collaborate to improve, remix and build on it.

It's the brains to some of the most popular devices that are driving the next Industrial Revolution, the 3D printer. [makerbot.com]

And as Massimo Banzi says, "You don't need anybody's permission to create something great." So what you waiting for? ■

Andrew Chalkley is an Expert Teacher at Treehouse, Co-founder of iOS app development company Secret Monkey Science and technical writer on *Screencasts.org*. In his spare time he hacks around with hardware such as Arduino, Raspberry Pi and Kinect.

Reprinted with permission of the original author.
First appeared in *hn.my/arduino* (forefront.io)

From AS3 to Haxe

By GRANT MATHEWS

I RECENTLY CONVERTED A codebase of about 5000 lines from ActionScript 3 to Haxe [haxe.org]. Here's what I learned.

Initial impressions:

1 **Haxe compiles really fast.**
I see compile times from 0.1 to 1 second — usually 0.1. As a comparison, the same project used to see compile times from 2-15 seconds in AS3. This is great when you're testing out lots of small changes rapidly.

2 **Autocompletion is built in.**
Haxe was designed with autocompletion in mind. This means almost any Haxe editor supports it, since the API is so simple. I've personally been using Sublime Text 2, which handles Haxe like a dream, and runs on Windows, OSX and Linux (I run OSX). If you run Windows, you'd be a fool not to use FlashDevelop, which is rock solid.

AS3 autocomplete was a shaky proposition outside of FlashDevelop, so having it available everywhere (and being able to even autocomplete the flash API) is a boon.

3 **The AS3 target is solid.**
Through the entire translation phase, I didn't encounter a single Haxe bug. Debugging was a breeze because the backtrace given was relative to the Haxe files (as you should expect).

Language differences

1 **Stronger type system.**

Generics

This is a huge, huge win for Haxe. If you've used AS3, you might be familiar with how they have a parameterized `Vector.<T>`. You're probably familiar with how you got your hopes up for properly generic types and functions, only to have them dashed when it turned out `Vector.<T>` is an Adobe hardcode and you can't do anything like it. Haxe, on the other hand, has generics built into the language, so you can make both functions and objects generic.

Function types

In AS3, functions have one type: `Function`. In Haxe, they have many. For instance, a function that takes an `Int` and converts it to a `String` would be `Int -> String`. This catches many bugs.

2 **No more Object.**
The problem with `Object` from AS3 is that it's not type safe. In AS3 you can do something like this:

```
var myObject:Object = {};

myObject[1] = "hi";
myObject["somekey"] = 4.3;
```

Obviously if you do a loop through that object, you couldn't specify the type of the key. Haxe gets around this by splitting `Object` into two types that encompass all of its expected functionality.

The first is `TypedDictionary<Key, Value>`. `TypedDictionary` is your typical key-value store: put in a key of one type, get out a value of another.

The second is `typedef`. `typedef` is really similar to `struct` from C. If you're not familiar with `struct`, you can also think of it as an AS3 `Object` that you can't add any more properties to. Here's an example.

```
typedef User = {
    var age : Int;
    var name : String;
}
var u : User = { age : 26, name : "Tom" };
u.age = 32;
trace(u.name);
u.xyz = 557; //Error!
```

Both of these have the advantage of type safety. Also notice how they really do separate the two use cases of AS3's `Object`. In AS3 you shouldn't be using `Object` for structs; you should be making classes — but I'd bet that you do anyways because `Objects` are so much more lightweight.

There's a nice interplay between typedefs and classes. For instance, the `Iterator` `typedef` defines two methods: `next` and `hasNext`. You can make an `Iterator` like so:

```
var a:Array<Int> = [1,2,3,4,5,6];
var loc:Int = 0;
var i:Iterator<Int> = {
    hasNext = function() return loc != a.length,
    next = function() return a[loc++]
    }
```

But you can also make an iterator like this:

```
class MyIterator<Int> {
    function hasNext(): Bool {
        // do some stuff
    }
    function next(): T {
        // do other stuff
    }
}
```

Both these two iterators are interchangeable. Nice!

Strictly speaking, you can simulate an AS3 object by declaring a var `Dynamic` and using `Reflect.setField` and `Reflect.getField`. I encourage you not to do this, though, because the built-in `Object` replacements are far superior in terms of type-safety.

3 Improved for loops.
This is a nice advantage over AS3. Haxe's loops only use iterators. A traditional for loop looks like this:

```
for (x in 0...10)
    trace(x)
```

If you want to loop through an `Array`, it looks like this:

```
var myArray:Array<Int> = [1,2,3,4,5];

for (val in myArray)
    trace(val)
```

This means that you can make any user-defined object loopable simply by defining an `iterator()` method on the object.

4 Different setter/getter syntax.
`public var someInt(getSomeInt, setSomeInt): Int;`
This indicates that the variable `someInt` has a setter and getter method named `getSomeInt` and `setSomeInt`, respectively.

5 Enumerations
`enum Color { Red; Green; Blue; }`
They type check; no mismatching enumerations because you did something like `var Red:Int = 1`.

Enumerations in Haxe are a bit more powerful than, say, those found in Java or C++. If you're familiar with Haskell, you'll see that they take influence from algebraic datatypes. They can have values and be recursive. (And if you're not familiar with Haskell, don't be scared away! It's quite simple.) Nicolas Cannasse wrote Haxe in OCaml, so the influence is obvious.

Here's your basic binary tree, where each node in the tree is either a leaf or a node with two trees beneath:

```
enum Tree {
    Leaf(val: Int);
    Node(left:Tree, right:Tree);
}
```

Of course, we don't need to be showing favoritism to `Int` — we can templatize!

```
enum Tree<T> {
    Leaf(val: T);
    Node(left:Tree<T>, right:Tree<T>);
}
```

Let's see AS3 typecheck that!

6 Using

The `using` keyword allows you to add additional methods onto existing types. The classic example of `using` is the `Lambda` class. The `Lambda` class has a bunch of static methods on it. We'll use `Lambda.exists` as an example. The definition looks like this:

```
static function exists<T>(it: Iterable<T>, F: T
-> Bool);
```

For example, you could use the function like this:

```
var myArray:Array<Int> = [1,2,3,4];
var is3:Int -> Bool = function(x: Int) return x
== 3;

if (Lambda.exists(myArray, is3)) {
    trace("I found a 3 in the array!");
}
```

The `using` keyword lets you drop `exists` right onto the `Array` object itself — or any other object that implements `Iterable<T>`, for that matter. Check it out:

```
using Lambda;

var myArray:Array<Int> = [1,2,3,4];
var is3:Int -> Bool = function(x: Int) return x
== 3;

if (myArray.exists(is3)) {
    trace("I found a 3 in the array!");
}
```

Nice, huh?

Problems

1 No cross-platform Dictionary type.

The AS3 target has `TypedDictionary`, but sadly it doesn't exist on all platforms. The NME target has `ObjectHash`, but the problem with `ObjectHash` is that it can't have primitive types (`Int,String, Float, Bool`) as keys.

To solve this problem, I wrote SuperObjectHash.hx [hn.my/soh] which combines `ObjectHash` and `Hash` into a single interface that you can use without having to worry about having primitive typed values.

(It was pointed out on #haxe that ObjectHash is planned to be introduced to Haxe, and will make it in by Haxe 3. Then my SuperObjectHash won't even be necessary.)

2 Overriding setters/getters is tricky.

Essentially, you can override a variable setter and getter, but only if you know the name of the functions you're overriding (which rules out extending some built-ins), and you're not permitted to use `super` to access the parent's property. From what I understand, these limitations stem from problems with target languages, primarily PHP. This essentially means that enhancing old behavior is impossible.

The good news is that the `super` limitation is going away in Haxe 3, too. The first Haxe 3 release candidate is coming out in late February, and I'm definitely looking forward to it.

Closing thoughts

My overall impression? As a suffering AS3 developer, Haxe is a dream come true. It has all the features I wished AS3 would have — and a few more. It compiles faster than AS3 and it has better autocompletion than AS3. It optimizes code better than AS3 (which is to say not at all— AS3 optimizes absolutely nothing). It even has macros. Yep, a language with macros that doesn't have parenthesis all over the place (not to speak badly of Lisp, of course). Haxe is impressive.

Even better, Haxe doesn't feel like a dead end language. I can cross-compile to any number of platforms with NME, which is exciting. I've been experimenting with using NME, which is admittedly a bit shakier than using the AS3 libraries, but it's there, and it's exciting. I no longer feel nervous about the world moving to HTML5. Nicolas Cannasse and the Haxe team move incredibly fast. Just the other day I noticed they were writing a Haxe shader language and a set of generic 3D bindings that will interoperate between Flash's Stage3D, HTML5's WebGL, and more. Wow.

I have to feel like one of the big reasons that Haxe hasn't seen more widespread attention is that it's not English. The documentation is full of imprecise wording that feels amateur. (In fact, I spent some time cleaning it up the other day.) It's easy to draw the conclusion that the language is like the docs — mismatched and awkward — but it's not.

Check it out. The possibilities are wild. ■

Grant Mathews is a 22-year-old senior currently attending Stanford University. He wants to prove that games can be art, and invent the tools to make it happen.

Reprinted with permission of the original author.
First appeared in *hn.my/haxe* (grantmathews.com)

My First 5 Minutes On A Server

Essential Security for Linux Servers

By BRYAN KENNEDY

Server security doesn't need to be complicated. My security philosophy is simple: adopt principles that will protect you from the most frequent attack vectors, while keeping administration efficient enough that you won't develop "security cruft." If you use your first 5 minutes on a server wisely, I believe you can do that.

Any seasoned sysadmin can tell you that as you grow and add more servers and developers, user administration inevitably becomes a burden. Maintaining conventional access grants in the environment of a fast-growing startup is an uphill battle— you're bound to end up with stale passwords, abandoned intern accounts, and a myriad of "I have sudo access to Server A, but not Server B" issues. There are account sync tools to help mitigate this pain, but IMHO the incremental benefit isn't worth the time nor the security downsides. Simplicity is the heart of good security.

Our servers are configured with two accounts: root and deploy. The deploy user has sudo access via an arbitrarily long password and is the account that developers log into. Developers log in with their public keys, not passwords, so administration is as simple as keeping the `authorized_keys` file up-to-date across servers. Root login over ssh is disabled, and the deploy user can only log in from our office IP block.

The downside to our approach is that if an `authorized_keys` file gets clobbered or mis-permissioned, I need to log into the remote terminal to fix it (Linode offers something called Lish, which runs in the browser). If you take appropriate caution, you shouldn't need to do this.

Note: I'm not advocating this as the most secure approach— just that it balances security and management simplicity for our small team. From my experience, most security breaches are caused either by insufficient security procedures or sufficient procedures poorly maintained.

Let's Get Started

Our box is freshly hatched, virgin pixels at the prompt. I favor Ubuntu; if you use another version of linux, your commands may vary. Five minutes to go:

```
passwd
```

Change the root password to something long and complex. You won't need to remember it, just store it somewhere secure. This password will only be needed if you lose the ability to log in over ssh or lose your sudo password.

```
apt-get update
apt-get upgrade
```

The above gets us started on the right foot.

Install Fail2ban

```
apt-get install fail2ban
```

Fail2ban is a daemon that monitors login attempts to a server and blocks suspicious activity as it occurs. It's well configured out of the box.

Now, let's set up your login user. Feel free to name the user something besides "deploy", it's just a convention we use:

```
useradd deploy
mkdir /home/deploy
mkdir /home/deploy/.ssh
chmod 700 /home/deploy/.ssh
```

Require Public Key Authentication

The days of passwords are over. You'll enhance security and ease of use in one fell swoop by ditching those passwords and employing public key authentication for your user accounts.

```
vim /home/deploy/.ssh/authorized_keys
```

Add the contents of the `id_rsa.pub` on your local machine and any other public keys that you want to have access to this server to this file.

```
chmod 400 /home/deploy/.ssh/authorized_keys
chown deploy:deploy /home/deploy -R
```

Test the New User and Enable Sudo

Now test your new account logging into your new server with the deploy user (keep the terminal window with the root login open). If you're successful, switch back to the terminal with the root user active and set a sudo password for your login user:

```
passwd deploy
```

Set a complex password. You can either store it somewhere secure or make it something memorable to the team. This is the password you'll use to sudo.

```
visudo
```

Comment all existing user/group grant lines and add:

```
root    ALL=(ALL) ALL
deploy  ALL=(ALL) ALL
```

The above grants sudo access to the deploy user when they enter the proper password.

Lock Down SSH

Configure ssh to prevent password and root logins and lock ssh to particular IPs:

```
vim /etc/ssh/sshd_config
```

Add these lines to the file, inserting the IP address from where you will be connecting:

```
PermitRootLogin no
PasswordAuthentication no
AllowUsers deploy@(your-ip) deploy@
(another-ip-if-any)
```

Now restart ssh:

```
service ssh restart
```

Setup A Firewall

No secure server is complete without a firewall. Ubuntu provides ufw, which makes firewall management easy. Run:

```
ufw allow from {your-ip} to any port 22
ufw allow 80
ufw allow 443
ufw enable
```

This sets up a basic firewall and configures the server to accept traffic over port 80 and 443. You may wish to add more ports depending on what your server is going to do.

Enable Automatic Security Updates

I've gotten into the apt-get update/upgrade habit over the years, but with a dozen servers, I found that servers I logged into less frequently weren't staying as fresh. Especially with load-balanced machines, it's important that they all stay up to date. Automated security updates scare me somewhat, but not as badly as unpatched security holes.

```
apt-get install unattended-upgrades
vim /etc/apt/apt.conf.d/10periodic
```

Update the file to look like this:

```
APT::Periodic::Update-Package-Lists "1";
APT::Periodic::Download-Upgradeable-Packages
"1";
APT::Periodic::AutocleanInterval "7";
APT::Periodic::Unattended-Upgrade "1";
```

One more config file to edit:

```
vim /etc/apt/apt.conf.d/50unattended-upgrades
```

Update the file to look like below. You should probably keep updates disabled and stick with security updates only:

```
Unattended-Upgrade::Allowed-Origins {
        "Ubuntu lucid-security";
//      "Ubuntu lucid-updates";
};
```

Install Logwatch To Keep An Eye On Things

Logwatch [hn.my/logwatch] is a daemon that monitors your logs and emails them to you. This is useful for tracking and detecting intrusion. If someone were to access your server, the logs that are emailed to you will be helpful in determining what happened and when, as the logs on your server might have been compromised.

```
apt-get install logwatch
vim /etc/cron.daily/00logwatch
```

Add this line:

```
/usr/sbin/logwatch --output mail --mailto test@
gmail.com --detail high
```

All Done!

I think we're at a solid place now. In just a few minutes, we've locked down a server and set up a level of security that should repel most attacks while being easy to maintain. At the end of the day, it's almost always user error that causes break-ins, so make sure you keep those passwords long and safe! ■

Bryan Kennedy is the Co-Founder and CTO of Sincerely, helping to scale thoughtfulness across the world. Bryan is a YCombinator alum and an angel investor. On warm summer nights he runs *MobMov.org*, a worldwide collective of guerrilla drive-ins.

Reprinted with permission of the original author.
First appeared in *hn.my/5mins* (plusbryan.com)

Whose Bug Is This Anyway?

By PATRICK WYATT

At a certain point in every programmer's career we each find a bug that seems impossible because the code is right, dammit! So it must be the operating system, the tools, or the computer that's causing the problem. Right?!?

Today's story is about some of those bugs I've discovered in my career.

This bug is Microsoft's fault… or not

Several months after the launch of Diablo in late 1995, the StarCraft team put on the hustle and started working extra long hours to get the game done. Since the game was "only two months from launch," it seemed to make sense to work more hours every day (and some weekends, too). There was much to do, because even though the team started with the Warcraft II game engine, almost every system needed rework. All of the scheduling estimates were willfully wrong (my own included), so this extra effort kept on for over a year.

I wasn't originally part of the StarCraft dev team, but after Diablo launched, when it became clear that StarCraft needed more "resources" (a.k.a. people), I joined the effort. Because I came aboard late I didn't have a defined role, so instead I just "used the force" to figure out what needed to happen to move the project forward.

I got to write fun features like implementing parts of the computer AI, which was largely developed by Bob Fitch. One was a system to determine the best place to create "strong-points" — places that AI players would gather units for defense and staging areas for attacks. I was fortunate because there were already well-designed APIs that I could query to learn which map areas were joined together by the path-finding algorithm and where concentrations of enemy units were located in order to select good strong-points, as it would otherwise be embarrassing to fortify positions that could be trivially bypassed by opponents.

I re-implemented some components like the "fog of war" system I had written for previous incarnations of the 'Craft series. StarCraft deserved to have a better fog-of-war system than its predecessor, Warcraft II, with finer resolution in the fog-map, and we meant to include line-of-sight visibility calculations so that units on higher terrain would be invisible to those on lower terrain, greatly increasing the tactical complexity of the game: when you can't see what the enemy is doing, the game is far more complicated. Similarly, units around a corner would be out of sight and couldn't be detected.

The new fog of war was the most enjoyable part of the project for me, as I needed to do some quick learning to make the system functional and fast. Earlier efforts by another programmer were graphically displeasing and moreover, ran so slowly as to be unworkable. I learned about texture filtering algorithms and Gouraud shading, and wrote the best x386 assembly language of my career — a skill now almost unnecessary for modern game development. Like many others I hope that StarCraft is eventually open-sourced, in my case so I can look with fondness on my coding efforts, though perhaps my memories are better than seeing the actual code!

> “The trick for effective bug-fixing is to discover how to reliably reproduce a problem.”

But my greatest contribution to the StarCraft code was fixing defects. With so many folks working extreme hours writing brand new code, the entire development process was haunted by bugs: two steps forward, one step back. While most of the team coded new features, I spent my days hunting down the problems identified by our Quality Assurance (QA) test team.

The trick for effective bug-fixing is to discover how to reliably reproduce a problem. Once you know how to replicate a bug, it's possible to discover why the bug occurs, and then it's often straightforward to fix. Unfortunately reproducing a "will o' the wisp" bug that only occasionally deigns to show up can take days or weeks of work. Even worse is that it is difficult or impossible to determine beforehand how long a bug will take to fix, so long hours investigating were the order of the day. My terse status updates to the team were along the lines of "yeah, still looking for it." I'd sit down in the morning and basically spend all day cracking on, sometimes fixing hundreds of issues, but many times fixing none.

One day I came across some code that wasn't working: it was supposed to choose a behavior for a game unit based on the unit's class ("harvesting unit", "flying unit", "ground unit", etc.) and state ("active", "disabled", "under attack", "busy", "idle", etc.). I don't remember the specifics after so many years, but something along the lines of this:

```
if (UnitIsHarvester(unit))
    return X;

if (UnitIsFlying(unit)) {
    if (UnitCannotAttack(unit))
        return Z;
    return Y;
}

... many more lines

if (! UnitIsHarvester(unit))    // "!" means "not"
    return Q;

return R;   <<< BUG: this code is never reached!
```

After staring at the problem for too many hours, I guessed it might be a compiler bug, so I looked at the assembly language code.

For the non-programmers out there, compilers are tools that take the code that programmers write and convert it into "machine code", which are the individual instructions executed by the CPU.

```
// Add two numbers in C, C#, C++ or Java
A = B + C

; Add two numbers in 80386 assembly
mov     eax, [B]    ; move B into a register
add     eax, [C]    ; add C to that register
mov     [A], eax    ; save results into A
```

> **“Crunch time is a failed development methodology; developers get tired and start making stupid mistakes.”**

After looking at the assembly code I concluded that the compiler was generating the wrong results, and sent a bug report off to Microsoft — the first compiler bug report I'd ever submitted. And I received a response in short order, which in retrospect is surprising: considering that Microsoft wrote the most popular compiler in the world, it's hard to imagine that my bug report got any attention at all, much less a quick reply!

You can probably guess — it wasn't a bug, there was a trivial error I had been staring at all along but didn't notice. In my exhaustion — weeks of 12+ hour days — I had failed to see that it was impossible for the code to work properly. It's not possible for a unit to be neither “a harvester” nor “not a harvester”. The Microsoft tester who wrote back politely explained my mistake. I felt crushed and humiliated at the time, only slightly mitigated by the knowledge that the bug was now fixable.

Incidentally, this is one of the reasons that crunch time is a failed development methodology, as I've mentioned in past posts on this blog; developers get tired and start making stupid mistakes. It's far more effective to work reasonable hours, go home, have a life, and come back fresh the next day. When I started ArenaNet with two of my friends, the “no crunch” philosophy was a cornerstone of our development effort, and one of the reasons we didn't buy foosball tables and arcade machines for the office. Work, go home at a reasonable time, and come back fresh!

This bug is actually Microsoft's fault

Several years later, while working on Guild Wars, we discovered a catastrophic bug that caused game servers to crash on startup. Unfortunately, this bug didn't occur in the “dev” (“development”) branch that the programming team used for everyday work, nor did it occur in the “stage” (“staging”) branch used by the game testers for final verification, it only occurred in the “live” branch which our players used to play the game. We had “pushed” a new build out to end-users, and now none of them could play the game! WTF!

Having thousands of angry players amps up the pressure to get that kind of problem fixed quickly. Fortunately we were able to “roll back” the code changes and restore the previous version of the code in short order, but now we needed to understand how we broke the build. Like many problems in programming, it turned out that several issues taken together conspired to cause the bug.

There was a compiler bug in Microsoft Visual Studio 6 (MSVC6), which we used to build the game. Yes! Not our fault! Well, except that our testing failed to uncover the problem. Whoops.

Under certain circumstances, the compiler would generate incorrect results when processing templates. What are templates? They're useful, but they'll blow your mind. [hn.my/fqa]

C++ is a complex programming language, so it is no surprise that compilers that implement the language have their own bugs. In fact the C++ language is far more complicated than other mainstream languages. Ruby is a complex and fully-featured language, but C++ is twice as complex, so we would expect it to have twice as many bugs, all other things being equal.

> **“Everyone, programmers and build servers alike, should be running the same version of the tools!”**

When we researched the compiler bug it turned out to be one that we already knew about, and that had already been fixed by the Microsoft dev team in MSVC6 Service Pack 5 (SP5). In fact all of the programmers had already upgraded to SP5. Sadly, though we had each updated our work computers, we neglected to upgrade the build server, which is the computer that gathers the code, artwork, game maps, and other assets and turns them into a playable game. So while the game would run perfectly on each programmers' computer, it would fail horribly when built by the build server. But only in the live branch!

Why only in live? Hmmm…. Well, ideally all branches (dev, stage, live) would be identical to eliminate the opportunity for bugs just like this one, but in fact there were a number of differences. For a start, we disabled many debugging capabilities for the live branch that were used by the programming and test teams. These capabilities could be used to create gold and items, or spawn monsters, or even crash the game.

We wanted to make sure that the ArenaNet and NCsoft staff didn't have access to cheat functions because we wanted to create a level playing field for all players. Many MMO companies have had to fire folks who abused their godlike "GM" powers, so we thought to eliminate that problem by removing capability.

A further change was to eliminate some of the "sanity checking" code that's used to validate that the game is functioning properly. This type of code, known as asserts or assertions by programmers, is used to ensure that the game state is proper and correct before and after a computation. These assertions come with a cost, however: each additional check that has to be performed takes time; with enough assertions embedded in the code, the game can run quite slowly. We had decided to disable assertions in the live code to reduce the CPU utilization of the game servers, but this had the unintended consequence of causing the C++ compiler to generate the incorrect results which led to the game crash. A program that doesn't run uses a lot less CPU, but that wasn't actually the desired result.

The bug was easily fixed by upgrading the build server, but in the end we decided to leave assertions enabled even for live builds. The anticipated cost-savings in CPU utilization (or more correctly, the anticipated savings from being able to purchase fewer computers in the future) were lost due to the programming effort required to identify the bug, so we felt it better to avoid similar issues in future.

Lesson learned: everyone, programmers and build servers alike, should be running the same version of the tools!

Your computer is broken

After my experience reporting a non-bug to the folks at Microsoft, I was notably shyer about suggesting that bugs might be caused by anything other than the code I or one of my teammates wrote.

During the development of Guild Wars (GW), I had occasion to review many bug reports sent in from players' computers. As GW players may remember, in the (hopefully unlikely) event that the game crashed, it would offer to send the bug report back to our "lab" for analysis. When we received

> **“Bugs come in all manner of shapes and sizes and some don’t have a clear owner, so several of us would take turns at fixing these bugs.”**

those bug reports we triaged to determine who should handle each report, but of course bugs come in all manner of shapes and sizes and some don’t have a clear owner, so several of us would take turns at fixing these bugs.

Periodically we’d come across bugs that defied belief, and we’d be left scratching our heads. While it wasn’t impossible for the bugs to occur, and we could construct hypothetically plausible explanations that didn’t involve redefining the space-time continuum, they just “shouldn’t” have occurred. It was possible they could be memory corruption or thread race issues, but given the information we had, it just seemed unlikely.

Mike O’Brien, one of the co-founders and a crack programmer, eventually came up with the idea that they were related to computer hardware failures rather than programming failures. More importantly, he had the bright idea for how to test that hypothesis, which is the mark of an excellent scientist.

He wrote a module (“OsStress”) which would allocate a block of memory, perform calculations in that memory block, and then compare the results of the calculation to a table of known answers. He encoded this stress-test into the main game loop so that the computer would perform this verification step about 30-50 times per second.

On a properly functioning computer this stress test should never fail, but surprisingly we discovered that on about 1% of the computers being used to play Guild Wars, it did fail! One percent might not sound like a big deal, but when one million gamers play the game on any given day that means 10,000 would have at least one crash bug. Our programming team could spend weeks researching the bugs for just one day at that rate!

When the stress test failed, Guild Wars would alert the user by closing the game and launching a web browser to a Hardware Failure page which detailed the several common causes that we discovered over time:

- **Memory failure**: in the early days of the IBM PC, when hardware failures were more common, computers used to have “RAM parity bits” so that in the event a portion of the memory failed, the computer hardware would be able to detect the problem and halt computation, but parity RAM fell out of favor in the early ’90s. Some computers use “Error Correcting Code” (ECC) memory, but because of the additional cost, it is more commonly found on servers rather than desktop computers.
- **Overclocking**: while less common these days, many gamers used to buy lower clock rate — and hence less expensive — CPUs for their computers, and would then increase the clock frequency to improve performance. Overclocking a CPU from 1.8 GHz to 1.9 GHz might work for one particular chip but not for another. I’ve overclocked computers myself without experiencing an increase in crash-rate, but some users ratchet up the clock frequency so high as to cause spectacular crashes as the signals bouncing around inside the CPU don’t show up at the right time or place.
- **Inadequate power supply**: many gamers purchase new computers every few years, but purchase new graphics cards more frequently. Graphics cards are an inexpensive system upgrade

> **“By focusing our efforts on the bugs that were actually our fault, the team was able to spend time creating features that players.”**

which generate remarkable improvements in game graphics quality. During the era when Guild Wars was released, many of these newer graphics cards had substantially higher power needs than their predecessors, and in some cases a computer power supply was unable to provide enough power when the computer was “under load,” as happens when playing games.

- **Overheating**: Computers don’t much like to be hot and malfunction more frequently in those conditions, which is why computer datacenters are usually cooled to 68-72F (20-22C). Computer games try to maximize video frame-rate to create better visual fidelity; that increase in frame-rate can cause computer temperatures to spike beyond the tolerable range, causing game crashes.

In college I had an external hard-drive on my Mac that would frequently malfunction during spring and summer when it got too hot. I purchased a six-foot SCSI cable that was long enough to reach from my desk to the mini-fridge (nicknamed Julio), and kept the hard-drive in the fridge year round. No further problems!

Once the Guild Wars tech support team was alerted to the overheating issue, they had success fixing many otherwise intractable crash bugs. When they received certain types of crash reports, they encouraged players to create more air flow by relocating furniture, adding external fans, or just blowing out the accumulated dust that builds up over years, and that solved many problems.

While implementing the computer stress test solution seems beyond the call of duty, it had a huge payoff: we were able to identify computers that were generating bogus bug reports and ignore their crashes. When millions of people play a game in any given week, even a low defect rate can result in more bug reports than the programming team can field. By focusing our efforts on the bugs that were actually our fault, the programming team was able to spend time creating features that players wanted instead of triaging unfixable bugs.

Ever more bugs

I don’t think that we’ll ever reach a stage where computer programs don’t have bugs — the increase in the expectations from users is rising faster than the technical abilities of programmers. The Warcraft 1 code base was approximately 200,000 lines of code (including in-house tools), whereas Guild Wars 1 eventually grew to 6.5 million lines of code (including tools). Even if it’s possible to write fewer bugs per line of code, the vast increase in the number of lines of code means it is difficult to reduce the total bug count. But we’ll keep trying.

To close, I wanted to share one of my favorite tongue-in-cheek quotes from Bob Fitch, whom I worked with back in my Blizzard days. He posited that, “All programs can be optimized, and all programs have bugs; therefore all programs can be optimized to one line that doesn’t work.” And that’s why we have bugs. ■

Patrick Wyatt is a lifelong programmer, game developer, and game-player, and as of 2004, a parent as well. He has worked on many popular games including Warcraft, Diablo, Starcraft, Guild Wars and TERA.

Reprinted with permission of the original author.
First appeared in *hn.my/bug* (codeofhonor.com)

How I Run My Own DNS Servers

By PETE KEEN

Photo: flickr.com/photos/zagrobot/2731084578/

For the longest time I used *zoneedit.com* as my DNS provider of choice. All of my important domains were hosted there, and they never really did me wrong. A few months back I decided that I wanted to learn how DNS actually works in the real world though. Like, what does it actually take to run my own DNS servers?

Step 0: Why would you ever do that?!

I'm mostly motivated by curiosity, but also by frustration. When something isn't going my way it just starts to make sense to do it myself. My frustration with zoneedit wasn't anything super specific. Their dynamic DNS system wasn't too terribly dynamic and adding and editing zones through their web interface got to be pretty tedious after awhile. I have a bunch of zones (32 at last count), most of which are very simple setups. `bugsplat.info` is way more complicated, but we'll get into that later.

Step 1: The Hardware

I decided that if I'm going to do this, I'm going to go all out. To that end, I rented two VPSs, one from *RamNode.com* in Atlanta and another from *Prgmr.com* in San Jose. Overall I would say that my RamNode experience has been more positive than my Prgmr experience. The network links have gone down twice in the past six months at Prgmr, which isn't the end of the world when you're running a redundant service, but it's still pretty annoying. RamNode has had 100% uptime so far.

Specs on these bad boys:

- prgmr (teroknor.bugsplat.info): 1 core, 1024MiB ram, 24GiB Disk, 160GiB transfer
- ramnode (empoknor.bugsplat.info): 4 core, 2048MiB ram, 30GiB SSD-backed Disk, 4000GiB transfer

I'm not even close to exploiting these two machines. I'm planning on moving more and more of my apps and sites over to them, but right now they're mainly handling this site and my email and DNS.

Why two machines? To host your own DNS servers, the registrars require you to list two IP addresses with the idea that you'll be providing redundant service. The one thing you don't want is downtime with DNS; it screws everything up.

Step 2: The Software

Once you decide to go down this DNS rabbit hole, there are a bunch of decisions to make on the software side. I considered PowerDNS and BIND and finally settled on *tinydns.org* managed via puppet and supply drop. Tinydns is a project started by Daniel J. Bernstein many years ago and has proven to be extremely reliable when run as intended (no axfr, configuration propogation via scp, etc). My setup is thus:

- Puppet [puppetlabs.com] managing the config for both boxes
- Supply drop [hn.my/supplydrop] deploys this configuration via Capistrano [hn.my/cap]
- Tinydns has a static config file checked into git, controlling most of my zones
- Tinydns also has a dynamic file that does my dynamic DNS updates for the home router

`bugsplat.info` is my oldest and thus most complicated domain. It's not even really that complicated; it just handles a lot of stuff. My Mac mini runs a cron job every minute that ssh's into both machines and rebuilds the tinydns config file if its IP has changed. That IP is then assigned to `subspace.bugsplat.info`, and I have a wildcard CNAME for `*.bugsplat.info` pointing at subspace. This lets me do things, like having various services running on that Mac mini with distinct hostnames, all hiding behind a common nginx. In addition, each VPS has a wildcard CNAME pointing to it from `*.<hostname>.bugsplat.info`, which lets me set up new apps and sites easily.

Step 3: The Email

One of the other problems I had with zoneedit was their free email forwarding setup. It was slow. So slow. Slower than molasses spread onto the back of the slowest dog. Even before this whole DNS adventure started I knew I wanted to get rid of that.

Each VPS runs its copy of my Postfix [postfix.org] setup (also managed via puppet), which mostly just forwards incoming email into my Gmail account. I don't send through it, since I haven't quite figured out all of the various DKIM and DMARC and SenderID and SPF things I need to do, and besides, Gmail won't send out through my SMTP server anyway.

Step 4: Logging

One of the more interesting aspects of this whole project has been getting a comprehensive view of everything that goes on in my little empire. The other day I set up global logging using Papertrail [papertrailapp.com], a hosted logging service. It doesn't do a whole lot; mostly it just seeps up logs from all of my services, including these two VPSs and a bunch of Heroku apps, makes them searchable for a few days, and drops tarballs of them onto S3 nightly. It has given me really valuable insight into at least two things: my Gmail backup wasn't working, and I get hit a lot by Chinese and India SSH breakin attempts. Still working on how to deal with that one, but the Gmail backup is up and running.

Conclusion

So after all of that, what have I learned? Mostly that I'm a very particular person with regards to this stuff. It's fun right now, but I can see it getting kind of tedious down the line. We'll find out! It's been an interesting ride thus far and I've learned quite a bit, which is the most important thing. ■

Pete Keen is a software developer currently residing in Portland Oregon. He writes articles about a variety of technology issues at *bugsplat.info*

Reprinted with permission of the original author. First appeared in *hn.my/dns* (bugsplat.info)

How I Work With Postgres

By CRAIG KERSTIENS

On at least a weekly basis and not uncommonly multiple times in a single week I get this question:

> *I've been hunting for a nice PG interface that works within other things. PGAdmin kinda works, except the SQL editor is a piece of shit.*
> *— @neilmiddleton*

Sometimes it leans more to, "what is the Sequel Pro equivalent for Postgres?" My default answer is: I just use `psql`, though I do have to then go on to explain how I use it. For those who are interested, you can read more below or just get the highlights here:

- Set your default `EDITOR` then use `\e`
- On postgres 9.2 and up `\x auto` is your friend
- Set history to unlimited
- `\d` all the things

Before going into detail on why `psql` works perfectly fine as an interface I want to rant for a minute about what the problems with current editors are and where I expect them to go in the future. First this is not a knock on the work that's been done on previous ones, for their time PgAdmin, phpPgAdmin, and others were valuable tools, but we're coming to a point where there's a broader set of users of databases than ever before and empowering them is becoming ever more important.

Empowering developers, DBA's, product people, marketers, and others to be comfortable with their database will lead to more people taking advantage of what's in their data. `pg_stat_statements` was a great start to this, laying a great foundation for valuable information being captured. Even with all of the powerful stats being captured in the statistics of PostgreSQL, so many are still terrified when they see something like:

```
QUERY PLAN
-----------------------------------------------
 Hash Join  (cost=4.25..8.62 rows=100 width=107)
(actual time=0.126..0.230 rows=100 loops=1)
   Hash Cond: (purchases.user_id = users.id)
   ->  Seq Scan on purchases  (cost=0.00..3.00
rows=100 width=84) (actual time=0.012..0.035
rows=100 loops=1)
   ->  Hash  (cost=3.00..3.00 rows=100 width=27)
(actual time=0.097..0.097 rows=100 loops=1)
         Buckets: 1024  Batches: 1  Memory
Usage: 6kB
         ->  Seq Scan on users  (cost=0.00..3.00
rows=100 width=27) (actual time=0.007..0.042
rows=100 loops=1)
 Total runtime: 0.799 ms
(7 rows)
```

Empowering more developers by surfacing this information in a digestible form, such as building on top of `pg_stat_statements` tools such as datascope [datascope.heroku.com] by @leinweber and getting this to be part of the default admin we will truly begin empowering a new set of users.

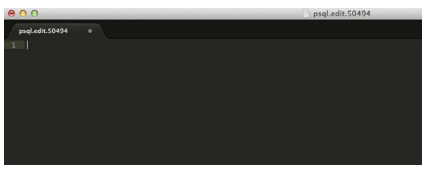
But enough of a detour, those tools aren't available today. If you're interested in helping build those to make the community better, please reach out. For now I live in a world where I'm quite content with simple ole `psql`. Here's how:

Editor

Ensuring you've exported your preferred editor to the environment variable `EDITOR` when you run `\e` will allow you to view and edit your last run query in your editor of choice. This works for vim, emacs, or even sublime text.

```
export EDITOR=subl
psql
\e
```

Gives me:



Note you need to make sure you connect with `psql` and have your editor set. Once you do that, saving and exiting the file will then execute the query.

\x auto

`psql` has long had a method of formatting output. You can toggle this on and off easily by just running the `\x` command. Running a basic query you get the output:

```
SELECT *
FROM users
LIMIT 1;
 id | first_name | last_name |       email        |    data
  1 | Rosemary   | Wassink   | rosemary@yahoo.com | "sex"=>"F"
```

With toggling the output and re-running the same query, we can see how it's now formatted:

```
\x
Expanded display is on.
craig=# SELECT * from users limit 1;
-[ RECORD 1 ]-------------------------
id         | 1
first_name | Rosemary
last_name  | Wassink
email      | rosemary@yahoo.com
data       | "sex"=>"F"
```

Using `\x auto` will automatically put this in what Postgres believes is the most intelligible format to read it in.

psql history

Hopefully this needs no justification. Having an unlimited history of all your queries is incredibly handy. Ensuring you set the following environment variables will ensure you never lose that query you ran several months ago again:

```
export HISTFILESIZE=
export HISTSIZE=
```

\d

The last item on the list of the first things I do when connecting to any database is check out what's in it. I don't do this by running a bunch of queries, but rather by checking out the schema and then poking at definitions of specific tables. `\d` and variations on it are incredibly handy for this. Here are a few highlights below:

Listing all relations with simply `\d`:

```
\d
                List of relations
 Schema |       Name       |     Type      | Owner
--------+------------------+---------------+-------
 public | products         | table         | craig
 public | products_id_seq  | sequence      | craig
 public | purchases        | table         | craig
 public | purchases_id_seq | sequence      | craig
 public | redis_db0        | foreign table | craig
 public | users            | table         | craig
 public | users_id_seq     | sequence      | craig
(7 rows)
```

List only all tables with `dt`:

```
\dt
         List of relations
 Schema |   Name    | Type  | Owner
--------+-----------+-------+-------
 public | products  | table | craig
 public | purchases | table | craig
 public | users     | table | craig
(3 rows)
```

Describe a specific relation with `\d RELATIONNAMEHERE`:

```
\d users
                                    Table "public.users"
   Column    |            Type             |                      Modifiers
-------------+-----------------------------+-----------------------------------------------------
 id          | integer                     | not null default nextval('users_id_seq'::regclass)
 first_name  | character varying(50)       |
 last_name   | character varying(50)       |
 email       | character varying(255)      |
 data        | hstore                      |
 created_at  | timestamp without time zone |
 updated_at  | timestamp without time zone |
 last_login  | timestamp without time zone |
```

■

Craig Kerstiens is part of the team at Heroku. He writes code in Python, curates *Postgresguide.com* and Postgres Weekly, and frequently speaks at conferences on those topics among others.

Reprinted with permission of the original author.
First appeared in *hn.my/postgres* (craigkerstiens.com)

MEET MANDRILL

By MailChimp

Mandrill is a new way to send transactional, triggered, and personalized emails.
It's also the world's largest species of monkey.

MANDRILL.COM

Building Stripe's API

By AMBER FENG

I THOUGHT IT WOULD be interesting to talk about Stripe's API, particularly lessons learned and what kind of things we did to try to make the API as easy to use as possible.

Make it easy to get started

It may sound like a no-brainer, but the best way to get people to try out (and hopefully eventually use) your API is to make it really easy to get started.

To that end, we do things like including pastable code snippets throughout our site and documentation. One of the first things you'll see on our front page is a curl snippet you can paste into a terminal to simulate charging a credit card.

Regardless of whether you have a Stripe account or not (if logged in, we fill in your test API key; otherwise, it's a sample account's API key), you can see the Stripe API in action.

All of our documentation code snippets are similarly possible to directly copy and paste — we try to embed as much information as possible (API keys, actual object IDs from the account, etc.) so our users don't have to.

Language-specific libraries and documentation

Since Stripe's API speaks HTTP and JSON, you could easily integrate it into your application with any standard HTTP client library. However, this still requires constructing requests and parsing responses on your own.

We maintain and support open-source libraries in some of today's most popular web languages. It turns out people are pretty attached to their favorite languages.

We had a lot of internal discussions about whether we actually wanted to support our own client bindings or allow the community to organically start and maintain the projects themselves. Is it worth owning the projects if it means that you might have to maintain libraries for languages or frameworks in which you don't have expertise?

Maybe.

Official libraries have the benefit of being consistent: they all have the same level of quality, support the same interface, and get updates at the same time. Having our own libraries also makes it easier for us to have language-specific documentation and help our users with any problems they might be having with a particular integration.

We decided that it was worth it, but this may not be the right answer for everyone.

Have a focused API, but allow flexibility

We've found that it's critically important to keep the API focused and simple.

It's often tempting to add new features that are not obviously necessary to the core API. For example, our users frequently want us to add better analytics, tax calculations, or to send customers receipts. While these things are nice, every feature you add makes the API more complex and cluttered.

You can instead give your users the tools to be able to write their own extensions. We allow our users (and third party applications) to hook into Stripe in a couple of ways:

Webhooks

Stripe uses webhooks to let our users know when some interesting event has happened. This ranges from events triggered by an API call, like `charge.succeeded` or `charge.refunded`, to asynchronous events like `customer.subscription.trial_will_end`.

Our aim was to make it easy to layer additional logic on top of Stripe events (like sending customer receipts or enabling push notifications). Giving our users the ability to build this kind of customized functionality allows them to control the entire experience for their users as well.

Stripe Connect

Stripe Connect, an API we released just last year, is another way of building on top of the Stripe platform.

Connect is an OAuth2 API [oauth.net/2] that allows a Stripe user to authorize access to their Stripe account to a third-party application. We've seen a variety of applications built on top of Stripe so far: marketplaces and checkout pages let users "plug in" their Stripe accounts to accept payments, and analytics dashboards fetch Stripe data in order to show interesting graphs or patterns.

Provide a testing environment

One of the most important things you need with an API is a great test/sandbox environment. This is particularly important for a payments API — our users shouldn't have to make live charges when they're trying to test their integration.

In our test environment, we allow users to send test webhooks of any type and provide handy test card numbers that trigger certain errors (like declines).

This allows them to easily test the behavior of their own application in the face of different scenarios instead of having to manually trigger things that are nondeterministic, like declines, or time-dependent, like expiring subscriptions.

Help your users debug

We're developers too. We know from experience that debugging is a disproportionately large portion of the development cycle. We also (unfortunately) know that sometimes you spend a lot of time debugging something that eventually turns out to be really obvious or silly.

For common or easy errors, you (the API) likely know exactly what's wrong. So why not try to help?

```
>> Stripe::Customer.create
Stripe::AuthenticationError:
No API key provided.  (HINT:
set your API key using "Stripe.
api_key = ".  You can generate
API keys from the Stripe web
interface.  See https://stripe.
com/api for details, or email
support@stripe.com if you have
any questions.)

>> Stripe.api_key = TEST_KEY
=> ...
>> Stripe::Charge.
retrieve(LIVE_CHARGE_ID)
Stripe::InvalidRequestError:
(Status 404) No such charge:
ch_17SOe5QQ2exd2S; a similar
object exists in live mode, but
a test mode key was used to
make this request.
```

On the other hand, some errors are harder to diagnose (especially from the API's end, since you have limited information about what your user is actually trying to accomplish).

Where possible, we absolutely think it's worthwhile to try to anticipate our users' errors and help as much as we can.

Dealing with Change

Lastly, dealing with change is never fun. As much as you hope you'll never have to change the API, sometimes you need to make changes and sometimes those changes are backwards-incompatible.

There's no easy answer for versioning APIs. We keep a per-user version which reflects the state of the API the first time the user made an API request. Most of our new features are additions that aren't backwards-incompatible, and they just work automatically for everyone.

Whenever we make a backwards-incompatible change, however, it doesn't affect the API behavior for any of our current users. Users can then choose to explicitly upgrade their version in the dashboard (after reviewing the detailed changelogs) or can send a version override header in any API request to test the behavior of a specific version. ■

Amber is an engineer at Stripe, and works primarily on the API. She loves all things web and distributed, and enjoys hacking on side projects and writing in her blog.

Reprinted with permission of the original author.
First appeared in *hn.my/stripeapi* (amberonrails.com)

Goldeneye 64's Inspirational Startup Story

By ALEX BALDWIN

Growing up, GoldenEye had a special place in my heart; it was the first game my parents wouldn't let me buy. I saved up allowances and dug up couch treasures for months to taste the forbidden fruit. The effort turned into one of the pillars of my childhood experiences. I still vividly remember where to place the proximity mines on Temple to get crazy spawn point kill streaks against my little brother. Fifteen years later, it's still inspiring me, but not for the proximity mines.

It's hard to imagine that this game almost didn't exist. Rare's studio head, Mark Betteridge, was quoted as saying,

> *When Nintendo asked if we wanted to do it, we said, "well not really"…we were trying to build our on IP, and film tie-ins meant a lot of ownership by the film company.*

The team faced insane amounts of adversity and uncertainty. Starting out, they didn't even know what the specs were for the new platform. Wikipedia on the game's development:

> *Final N64 specifications and development workstations were not initially available to Rare: a modified Sega Saturn controller was used for some early play testing, and the developers had to estimate what the finalized console's capabilities would be.*

Getting closer to the release date, the final platform specs were released and they had to make significant graphic cuts to make it work.

> *The final Nintendo 64 hardware could render polygons faster than the SGI Onyx workstations they had been using, but the game's textures had to be cut down by half. Karl Hilton explained one method of improving the game's performance: "A lot of GoldenEye is in black and white. RGB color textures cost a lot more in terms of processing power. You could do double the resolution if you used greyscale, so a lot was done like that. If I needed a bit of color, I'd add it in the vertex."*

While doing all this, their team had almost no idea what they were doing when they started out. Sound familiar?

> *GoldenEye 007 was developed by an inexperienced team, eight of whom had never previously worked on video games. David Doak commented in 2004, "Looking back, there are things I'd be wary of attempting now, but as none of the people working on the code, graphics, and game design had worked on a game before, there was this joyful naïveté."*

Scope was so slim that they didn't even originally plan out the legendary multiplayer mode that arguably made the game so successful. It was done almost exclusively by one guy as an afterthought.

> *The game's multiplayer mode was added late in the development process; Martin Hollis described it as "a complete afterthought." According to David Doak, the majority of the work on the multiplayer mode was done by Steve Ellis, who "sat in a room with all the code written for a single-player game and turned GoldenEye into a multiplayer game."*

Despite everything, the game went on to become the third highest selling N64 game, inspire console shooting games, and win a crazy amount of awards. Next time you're heading down the wrong way of the entrepreneurial roller-coaster, take a deep breath, make a cup of tea, and remember that you can make it happen. Persevere and dominate. ■

Alex is a designer at thoughtbot in San Francisco, previously with 500 Startups, Techstars, and *Console.fm*. He is a lover of lattes and dope beats.

Reprinted with permission of the original author.
First appeared in *hn.my/goldeneye* (alexbaldwin.com)

The Joys of Having a Forever Project

By JOHN BIESNECKER

I THINK MOST CREATIVE people have something that I call a Forever Project — a project that, despite its audacity and seeming impossibility, simply will not put itself to bed. A project that comes creeping back into your consciousness when you sit down for a break from "real work." A project that is hard to imagine actually embarking on, but whose mental cost of abandonment is far too high to even consider. A project that you'd totally do if you had the time, and the money, and the talent, and the…

I don't know about you, but I adore my Forever Project (mine happens to be a game that I've been punting around in various forms since the late 1990s, and I wouldn't be surprised if yours was also a game of some sort). I might not have made the progress on it that I wish I would have, but just having it out there as something to think about gives me a warm, fuzzy feeling.

Most people would say having a project that you can't put down but that you don't make any substantial progress on is silly — the antithesis of the various flavors of Getting Things Done that spring up now and again — but I disagree. While I may not have finished (or even really started) my game, poking around the edges of it have led me to wonderful tangents during which I've learned a lot about a lot of things, things that I may have never touched if it weren't for my Forever Project. Rather than be a source of disappointment, my Forever Project is a source of constant inspiration.

If I ever really completed it… well…I'm not sure what I would do. Probably replace it with a better version of itself. But that's silly because I'll never actually complete it and because a Forever Project is like the speed of light — you can get infinitely close to it, but you can never quite get there. It's just the nature of the beast.

Stop beating yourself up about not making progress on your "one big goal." Eating healthier, exercising more, and being a better spouse, friend, etc. are goals. Your Forever Project is not. Your Forever Project is your Beacon on the Hill, pushing you to be better, to learn, to stretch, to reach just a few finger widths beyond your grasp, over and over again.

Embrace your Forever Project, and never stop dreaming. ■

John Biesnecker (@biesnecker) is an American product designer and software developer based in Shanghai, China. He enjoys thinking about hard problems, telling good stories, and playing with his kids.

Reprinted with permission of the original author. First appeared in *hn.my/forever* (dev.gd)

Avoiding "The Stupid Hour"

By RACHEL KROLL

From time to time there is a romantic notion of teams pulling crazy hours and working all-nighters frequently. The idea is that you can cheat the night (or morning, for that matter) and continue coding, writing, or doing whatever it is you that you do. Sometimes this is driven by maniacal managers, but other times it comes from within.

Now, I've already written [hn.my/wrong] about the occasional flashes of insight which lead to a late evening here and there. That's something else. That's where you have a fire burning inside of you and you need to get that fire routed through your fingers and turned into code. You don't do this often. It's just when things get really good and all get flushed into the computer at once.

This is more about the relentless push to keep working night after night even when there's nothing special going on. Enough has been written about it, but it always seems to get really complicated in how it's described. I want to give it a simple name that anyone can remember and anyone else can understand.

I call it "the stupid hour." When talking about myself, I call it my stupid hour. It's the point when I've been awake for too long and anything I create is sure to be suboptimal. The late hour has drained enough out of me to where I turn stupid and my output shows this.

In my younger days, I used to feel this coming on and would just keep going. This was a spectacularly bad idea. The next morning, I'd get up and look at the code and would have no idea how it ever worked. A function I had written during the stupid hour might work for a specific test case, but I would have to sit and really dig at it to find out how. Then I would also discover that it didn't cover other test cases, either.

Since it was ugly and unmaintainable code, it needed to be fixed. The fact it didn't even work properly also meant it needed to go. More often than not I'd have to rip it out and redo that particular chunk of code. It was a net loss of time. I should have spent that time the night before just sleeping rather than trying to fight it while coding.

In recent times, I've grown to recognize this and appreciate it as a useful signal. I tend to stop earlier than I would have before and switch to other things after a certain point. Why write something that will have a good chance of being broken and will require an immediate fix? Leave it as a "to do" item and come back to it the next day.

There's another good reason for doing it this way. Have you ever come back to a project and been unsure of where to get started? If you had left off just one item sooner the day or week before, you'd already have a known starting point. Write it down on a post-it note and stick to your monitor, then go do something else.

The next day, not only will you have a nice place to resume, but you'll also have the benefit of several hours (or days, if over a weekend) of subconscious/background processing you didn't even realize was going on. It'll make for a better result overall.

Don't feed the stupid hour. It never ends well. ■

Rachel Kroll lives and works in Silicon Valley. Once a Googler, she now runs her own software consultancy business and writes daily about software, technology, sysadmin war stories, and productivity. Her first book, The Bozo Loop, is a collection of posts from 2011, with another on the way.

Reprinted with permission of the original author.
First appeared in *hn.my/stupidhour* (rachelbythebay.com)

```
{
    join: 'Intensive Online Bootcamp',
    learn: 'Web Development',
    goto: 'http://www.gotealeaf.com'
}
```

Tealeaf Academy
an online school for developers

Learn Ruby on Rails | Level up Skills | Launch Products | Get a Job

MEMSET® HOSTING

Rent your IT infrastructure from Memset and discover the incredible benefits of cloud computing.

MINISERVER™ CLOUD COMPUTE

From $0.020/hour
to 4 x 2.9 GHz Xeon cores
31 GBytes RAM
2.5TB RAID(1) disk

MEMSTORE™ CLOUD STORAGE

$0.091/GByte/month or less
99.999999% object durability
99.995% availability guarantee
RESTful API, FTP/SFTP and CDN Service

SCAN THE CODE FOR MORE INFORMATION

MEMSET® HOSTING

Find out more about us at
www.memset.com

or chat to our sales team on
0800 634 9270.

CarbonNeutral® hosting

THE BRITISH ASSESSMENT BUREAU ISO9001

BSI™ UKAS INFORMATION SECURITY MANAGEMENT 003

THE BRITISH ASSESSMENT BUREAU ISO14001