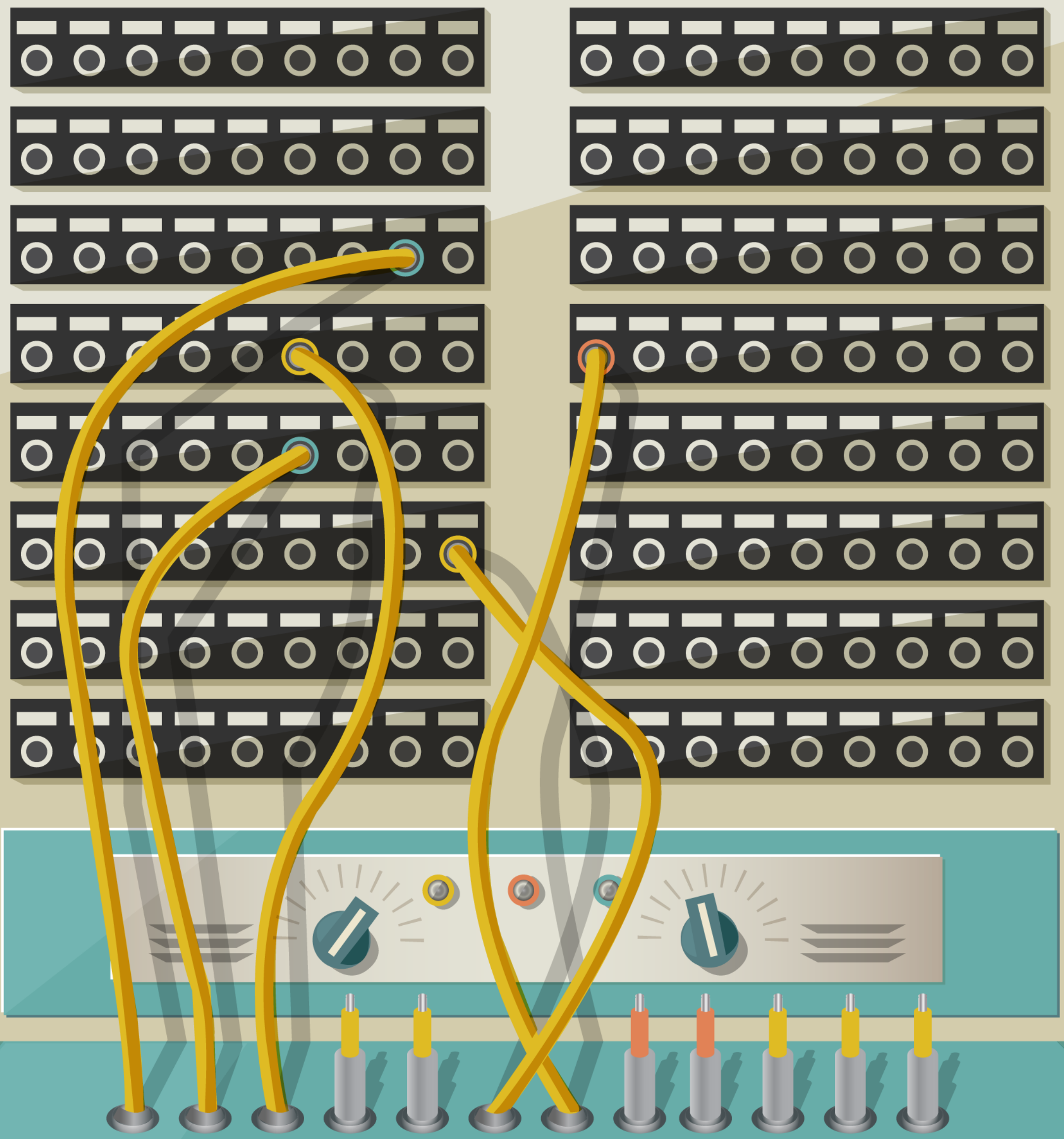
MARK NIJHOF



LEARNING BY EXAMPLE Erlang

Getting functional with Erlang

Mark Nijhof

This book is for sale at http://leanpub.com/functionalerlang

This version was published on 2015-02-09

Tweet This Book!

Please help Mark Nijhof by spreading the word about this book on Twitter!

The suggested hashtag for this book is #gettingfunctionalwitherlang.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#gettingfunctionalwitherlang

Contents

Introduction

I have been a professional developer since 2000, I have spend this time happily programming in C#, Delphi, Ruby, Javascript, and many other languages. But I have never worked with a proper functional programming language. That has now changed; about a year ago I started working with Erlang. And I have never been happier (while programming that is).

This book will get you started writing Erlang applications right from the get go. After some initial chapters introducing the language syntax and basic language features we will dive straight into building Erlang applications. While writing actual code you will discover and learn more about the different Erlang and OTP features. Each application we create is geared towards a different use-case, exposing the different mechanics of Erlang and we will introduce some common third party OSS libraries as well.

This is a book I wanted to read myself, I want it to be simple and to the point. Something to help you get functional with Erlang quickly. I imagine you with one hand holding your e-reader while typing code with your other hand, suddenly realising it is way past your bedtime.

I have made a broad assumption; only smart people would want to learn Erlang (that is you), you are probably also smart enough to find your way to all the language specifics when needed. So this book is not meant as a complete reference guide for Erlang. But it will teach you enough to give you a running start.

When you have reached the end of this book you will be able to build a full blown Erlang application and release it into production. You will understand the core Erlang features like; pattern matching, message passing, working with processes, and hot code swapping.

If you are completely new to Erlang then following the book from start to end will probably result in the best experience, but when you are already comfortable with a topic feel free to skip it.

What is Erlang

Erlang is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance.

Created in 1986 by Joe Armstrong, Robert Virding, and Mike Williams, Erlang was designed to help improve the development of telephony applications at Ericsson. In 1998 the language was released as open source.

What is OTP

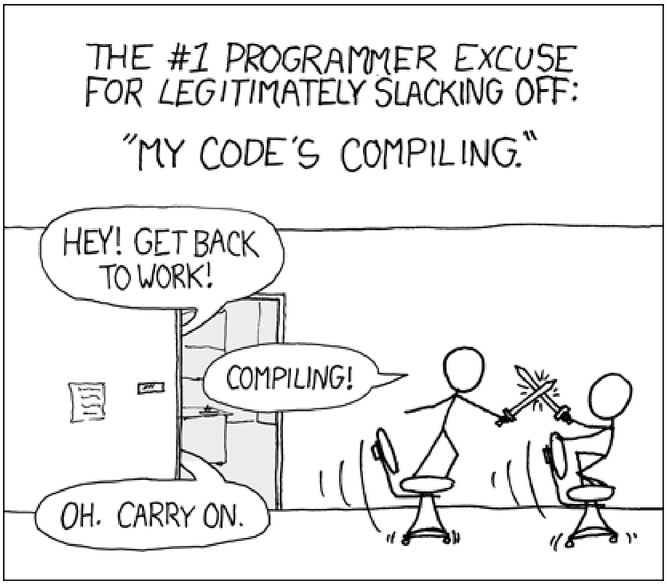
OTP (Open Telecom Platform) is a set of Erlang libraries and design principles providing middleware to develop these systems. It includes its own distributed database, applications to interface towards other languages, debugging and release handling tools.

The name Open Telecom Platform really shows its heritage, but don't let that name fool you; OTP provides you with very useful functionality, even when you are not building a telecom platform.

Allthough you can use Erlang without OTP, the functionality provided by OTP most often outweighs the added memory footprint, for the purpose of this book OTP usage is assumed.

Installation

This whole process takes a long time, so don't worry about getting coffee right now. You will have plenty of time to make and drink coffee while Erlang and its dependencies are compiling.



XKCD: http://xkcd.com/303/

Note to the early access reader

Because I am currently focusing more on the coding part of Erlang then on installing Erlang the following chapter is barely enough for you to work with. These instructions have been tested in MacOS but should also work on Linux based systems like Ubuntu.

Before the book is finished this chapter will be revisited and rewritten in something a bit more resilient.

Manage Erlang versions with Kerl

Kerl[1] is a version manager for Erlang which enables you to switch between versions. It also makes installing Erlang on your system quite easy. Kerl will download the source code and build it on your system, so it is not just downloading some pre-compiled libraries. Having said that if you rather install Erlang from source yourself then just go to the next chapter.

To start the process of installing Erlang using Kerl, open your favourite terminal and execute the following commands:

Installing kerl

```
cd ~/
curl -O https://raw.githubusercontent.com/spawngrid/ker\
l/master/kerl
chmod +x ./kerl
echo "ERL_CONFIGURE_OPTIONS=\"--enable-hipe\"" >> .kerl\
rc
```

Kerl is now installed on your system. Next up is getting executing Kerl to download and install Erlang for you.

Installing Erlang/OTP

```
cd ~/
./kerl update releases
./kerl build 17.0 17.0
./kerl install 17.0 ~/erlang_17_0
. ~/erlang_17_0/activate
```

The last line will activate Erlang 17.0 for use in the current terminal session. When you start a new terminal session then Erlang has not

[1] https://github.com/spawngrid/kerl

been activated yet. If you want to always use this version then add the last line to your bash profile.

You can read more about what Kerl can do for you on the github repository website: https://github.com/spawngrid/kerl[2].

If you have followed these steps then you can skip the next chapter and go straight to **The Erlang Shell** to verify that the installation of Erlang was successful.

[2]https://github.com/spawngrid/kerl

Or build Erlang from source

You can also manually build Erlang from source.

Ok lets start the installation of Erlang/OTP[3]. Open your terminal and follow the script below to install Erlang on your system. For Windows; just download either the 32-bit or 64-bit installer.

Installing Erlang/OTP

```
wget http://www.erlang.org/download/otp_src_17.0.tar.gz
tar xf otp_src_17.0.tar.gz
cd otp_src_17.0
export ERL_TOP=`pwd`
./configure --enable-hipe
make
sudo make install
cd ..
rm otp_src_17.0.tar.gz
```

Learn more?
For complete and proper documentation about the Erlang/OTP installation procedure go to the **otp_src_17.0** directory and open the sub folder **HOWTO** and find the specific installation document for your environment.

To verify that your installation was successful, you just have to start the Erlang Shell. Which incidentally is also what the next chapter is all about.

[3]http://www.erlang.org

The Erlang Shell

The Erlang shell is an emulator that executes compiled code, this can be in the form of compiled modules (.beam files) or just commands you type into the Erlang shell. It is an easy way to quickly test hypothesis, but it is not as powerful as some of the REPLs (Read Execute Print Loop) of other languages.

However it is the ideal place for us to start learning Erlang. To start, open your favourite terminal and type **erl** to let the magic begin. The result should look very similar to this:

iterm> starting the Erlang shell

```
~ ▯ erl
Erlang/OTP 17 [erts-6.0] [source] [64-bit] [smp:8:8]
[async-threads:10] [hipe] [kernel-poll:false]

Eshell V6.0  (abort with ^G)
1>
```

Awesome! You can now start bragging on Twitter that you have Erlang running on your machine. Let's quickly exit the Erlang shell before things get too exciting. You do this by typing **q()**. or **Ctrl+c** two times.

As you may have noticed, clicking **Ctrl+c** just once gives you some more options; but for now exiting is probably the most useful action.

iterm> abort menu

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
```

Feel free to repeat this exercise a few times to get more comfortable with it. The excitement will soon be reduced to an acceptable level, if not then this might be a good time to take a short break.

Shell functions

In the next chapter we will be doing some coding exercises using the Erlang shell. You might find the following commands useful while following the exercises.

Command	
f().	unbinds all variables, perfect for between exercises. Instead of restarting the Erlang shell you can basically reset it by just typing **f()**.
f(Foo).	unbinds the specific variable Foo, you made a mistake and want to correct it without having to do everything over again.
v(N).	Uses the return value of the command N in the current command, if N is positive. If it is negative, the return value of the Nth previous command is used. Say you forgot to bind a value with a variable: **10**. then **X = v(-1)**. will fix that for you.
q().	Quits the Erlang shell in a nice way, I usually just kill it using **Ctrl+c + Ctrl+c** as that is the Erlang way ;)

Erlang, the basics

The first step into learning Erlang is to get familiar with the language syntax. Erlang is actually a rather simple language, the Core Erlang language specification (PDF[4]) describing it is only 31 pages long. Having said that, we won't be covering all the language syntax and/or features in this chapter. Just enough to get us through the initial code examples.

Punctuation

Erlang uses **dots**, **commas** and **semicolons** to separate different **forms**, **expressions** and **clauses** from each other.

In Erlang the **dot** is used to terminate a **form**. A form is either a module attribute or a function declaration. We will cover those in more detail in later chapters. For now it is more important to know that the **dot** is also used to terminate **expressions** in the Erlang shell.

Except from module attributes and function declarations pretty much everything else are **expressions** in Erlang. And the **comma** is used to separate different expressions from each other.

Almost like English
Forms are very similar to how the English language constructs sentences. A sentence can be divided into multiple parts using a **comma**, and is always terminated with a **dot**.

[4]http://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf

The **semicolon** is also important in Erlang as it separates different clauses, for example function clauses or case clauses. We will discover the semicolon further in the next main chapter.

Here are a few expressions to get you started, open your Erlang shell and play along:

erl> Single Erlang expression

```
1> 5+5.
10
```

We executed the expression **5+5.** and the emulator returns the result which is 10 (hopefully no surprises here). Next up another exercise:

erl> Multi-part Erlang expression

```
1> io:format("Hello ~s~n", ["World"]),
1> io:format("My name is ~p~n", ["Yourname"]),
1> io:format("And I am ready to learn Erlang~n").
Hello World
My name is "Yourname"
And I am ready to learn Erlang
ok
```

Here we have 3 expressions separated by a comma. You may have noticed while typing along, but the Erlang shell only executes the expressions once it sees the dot. The output for the first three lines you probably expected, but the text **ok** perhaps not. Let's examine that a bit closer.

erl> What will be returned?

```
1> 21+21,
1> 63-21,
1> 6*7,
1> 126/3,
1> 252 div 6.
42
```

In Erlang the result of the last expression is always returned to the caller. Here it is the result from the expression **252 div 6**. In the previous example the function **io:format** returned **ok** which got printed. There. One more mystery solved.

Finally, when nothing seems to happen, make sure you terminated the expression with a dot.

erl> Always terminate the expression with a dot

```
1> io:format("I like this bo")
1>
1>
1> .
I like this book
```

Being able to do math and write to the console using Erlang is of-course awesome, but we are going to need a bit more to build an actual application. So let's quickly move to the next chapter and discover what else we can do.

Variables

Variables have been used to store values in the same scope by programming languages since the beginning of times, and Erlang is no exception here.

The convention in Erlang is that a variable name must start with an uppercase letter or an underscore. For example correct variable names are **Foo** and **Bar**, but **5Foos** is not and neither is **fooBar**.

Defining a variable and then failing to use it will result in a compiler warning (not applicable in the Erlang shell). You can easily fix this by prepending the variable with an underscore like, **_FooBar**. This will tell the compiler not to worry about that variable. You can also just use an underscore, something that is useful in pattern matching.

Having a variable start with an underscore but with the full name will help make the code more readable. Programmers read code at least ten times more then they actually write. So please write it for readability, I might have to fix a bug in there one day.

Enough talk, let's play:

erl> Variable names

```
1> Five = 5.
5
2> Four = 4.
4
3> Nine = Five + Four.
9
4> _NotUsed = <<“Variable is not important here">>.
<<“Variable is not important here">>
```

Awesome! Now we can store state locally. This Erlang thing is starting to look like a normal programming language already.

erl> Variable names

```
1> SelectedValue = 5.
5
2> SelectedValue = 4.
** exception error: no match of right hand side value 4
```

Errr!?

All state is immutable

It is so easy to think of state as something you can change whenever you want, but in Erlang all state is immutable. Which is a fancy way of saying; once you define a variable to be **XYZ** it will remain **XYZ** forever and ever and ever.

Having immutable state has many benefits for concurrent programming and the code itself is often much easier to understand. Initially, this may seem like a pain to deal with, but over time you will come to see its value and start appreciating it. I promise.

No defensive coding required

When writing code we often try to prevent crashes, because a crash is expensive to deal with. I have seen some weird defensive programming practices in my career, just because programmers are afraid to crash the application.

Not in Erlang. You probably didn’t even notice, but in the previous code example, part of the Erlang shell crashed and was restarted again. The reason you might not have noticed this is because it is lightning fast.

Erlang embraces the fact that no system can be perfect and that failure will happen from time to time. Erlang is designed in such a way that recovery from a crash is fast and deterministic, and it offers enough isolation between processes so that a crash will not bring down the whole application.

Crashing is an expected scenario in Erlang. It means you write code to handle things you expect to happen. When something unexpected happens you just let it crash. No more compulsive null checking before each function call, if it’s undefined and it should not be undefined, then you just crash.

A true war story
We have had a case where the system was crashing and recovering constantly and still processing a live video stream without anybody noticing. The only trace that something was wrong was in the log files. It was a bug and should not have been there, but impressive nonetheless.

Now that we have improved your understanding, lets look at that crash again:

erl> Let it crash

```
1> SelectedValue = 5.
5
2> SelectedValue = 4.
** exception error: no match of right hand side value 4
3> SelectedValue.
5
```

What you just experienced consciously is a little bit of the Erlang shell crashing and recovering again. The Erlang shell has logic to restore previous declared variables after a crash and thus **Selected-Value** was still defined as 5 after the crash. This same logic is also what makes calling **f()**. and **f(SelectedValue)**. work.

erl> Reset the Erlang shell

```
1> SelectedValue = 5.
5
2> f(SelectedValue).
ok
3> SelectedValue = 4.
4
```

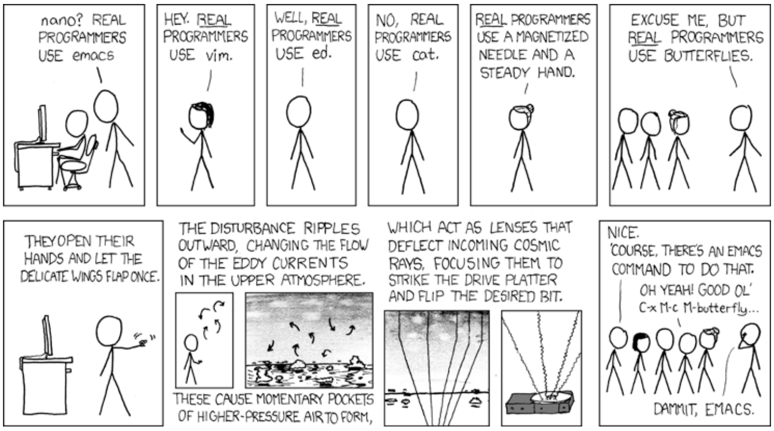
But variables where immutable, right? In reality, the Erlang shell has been cheating. This behaviour does not work in your code!

Crashes can be helpful

Trying to make the Erlang shell crash is sometimes an easy way to check whether everything is still ok. When nothing seems to work anymore, type **1=2**. and wait for the crash. If the crash didn't happen then you probably forgot a quote or something in the previous expression.

Modules

A module is the basic unit of code in Erlang, this is where all your functions go. Modules contain a series of module attributes and forms.


nano? REAL PROGRAMMERS USE emacs
HEY. REAL PROGRAMMERS USE vim.
WELL, REAL PROGRAMMERS USE ed.
NO, REAL PROGRAMMERS USE cat.
REAL PROGRAMMERS USE A MAGNETIZED NEEDLE AND A STEADY HAND.
EXCUSE ME, BUT REAL PROGRAMMERS USE BUTTERFLIES.
THEY OPEN THEIR HANDS AND LET THE DELICATE WINGS FLAP ONCE.
THE DISTURBANCE RIPPLES OUTWARD, CHANGING THE FLOW OF THE EDDY CURRENTS IN THE UPPER ATMOSPHERE.
THESE CAUSE MOMENTARY POCKETS OF HIGHER-PRESSURE AIR TO FORM,
WHICH ACT AS LENSES THAT DEFLECT INCOMING COSMIC RAYS, FOCUSING THEM TO STRIKE THE DRIVE PLATTER AND FLIP THE DESIRED BIT.
NICE.
'COURSE, THERE'S AN EMACS COMMAND TO DO THAT.
OH YEAH! GOOD OL' C-x M-c M-butterfly...
DAMMIT, EMACS.

XKCD: http://xkcd.com/378/

Let's open a new file in your favourite code editor, copy the following text and press **control+x control+s** to save it.

emacs hello_world.erl

```
-module(hello_world).
```

This creates a valid, but useless module called **hello_world**. The module name should be the same as the file name without its extension.

There is no namespacing mechnism
Unfortunately Erlang does not have a mechanism for namespacing. The module name is the only thing that makes it unique in the application, including its dependencies. Which means that whatever name you choose should be unique within your code, but also unique within your dependencies code.

It is regarded good practise to prefix your module names with for example the project name. Especially when your code is meant to be used by others.

The extensions
Erlang source code files have **.erl** extensions and compiles files have **.beam** extensions.

Ok back to code again:

emacs hello_world.erl

```
-module(hello_world).

say_hello_to(Name) ->
  io:format("Hello ~s, how are you?~n", [Name]).
```

Now we added the function **say_hello_to** that takes one parameter **Name** and it prints a nice welcome message. Lets try and compile the **hello_world** module. Open the Erlang shell and start typing:

erl> compiling: hello_world.erl

```
1> c(hello_world).
hello_world.erl:3: Warning: function say_hello_to/1
                             is unused
{ok,hello_world}
```

Hmm a warning, warnings are not critical, right? Lets run it anyway!

erl> using: hello_world.erl

```
2> hello_world:say_hello_to("Mark").
** exception error: undefined function
                        hello_world:say_hello_to/1
```

Interesting; let's take a look at both the warning and the error. The warning says, the function say_hello_to/1 is **unused**. But the error says, the function say_hello_to/1 is **undefined**? That sounds like a contradiction to me.

The problem is that the function say_hello_to/1 is not visible outside the module hello_world. All functions are private by default. To expose them use **-export([function_name/N])** where the function name is an atom and N is the number of arguments.

Atoms
Atoms are literals, constants with their own name as the value. These are purely for you, the developer. To make your code more readable. Be careful to not create atoms from user generated data, because each atom once created stays in memory. Users could crash your server by generating these atoms until it runs out of memory. Bad, but they are absolutely great otherwise, you'll see.

emacs hello_world.erl

```
-module(hello_world).

-export([say_hello_to/1]).

say_hello_to(Name) ->
  io:format("Hello ~s, how are you?~n", [Name]).
```

The export function takes a list with functions to export, and it can be called multiple times as well. This is a nice way to group different exports together.

emacs hello_world.erl

```
-module(hello_world).

-export([
         say_hello_to/1
        ,say_hello_to/2
        ]).

-export([say_bye_to/1]).

say_hello_to(Name) ->
    io:format("Hello ~s, how are you?~n", [Name]).

say_hello_to(Name1, Name2) ->
    io:format("Hello ~s and ~s, how are you?~n",
              [Name1, Name2]).

say_bye_to(Name) ->
    io:format("Bye ~s!~n", [Name]).
```

Let’s try to compile and run it now, again in the Erlang shell:

erl> compiling: hello_world.erl

```
1> c(hello_world).
{ok,hello_world}
2> hello_world:say_hello_to("Mark").
Hello Mark, how are you?
ok
```

Success! How awesome is this? You are almost a senior Erlang programmer by now. You know how to compose modules that expose functions, also you can print welcome messages.

Wow!

Afterword

So, why write a book? Because as you may know, writing a book is really, really, (pause to breath) really difficult. It takes a lot of time and energy away from your family, work and other fun things in life. What possesses a person to do such a thing? Short side-step; did you notice I included work in the fun things of life collection? Make sure you are in the same situation!

The main drivers behind the book

First of all, the main driver behind this book comes from the idea that you can only explain something to someone else if you truly understand the topic yourself. By writing this book I have learned so much more about Erlang then I would have by just coding Erlang. Instead of taking things at face value, I now need to understand them properly. It is funny how that works.

Next up is that I want to improve my writing skills. And the only way to do that is by writing more and taking it seriously. I used to blog quite a bit, but writing a book puts writing at a completely different level. Writing a blog post is quite easy, but as I am still writing this chapter, I have already re-read/edited it more times then I care to tell you.

How does it feel to write a book?

It is scary you know, because unlike a blog post, people tend to take books serious. And when people take something serious they can get pretty mean about it.

I saw this image of Harpo and his kids on Twitter today and it shows pretty much how I imagine people when they hear I am writing this book, or any book for that matter.

Harpo and Kids - wikipedia

Then why do it?

Because it is also a lot of fun!

Thanks for reading,

-Mark

Mark Nijhof
https://twitter.com/MarkNijhof