Video scripts for my videos

Philippe Pittoli

ABSTRACT

Check out for newer versions: `https://t.karchnu.fr/doc/videos-scripts.pdf`
And if you have questions: `karchnu@karchnu.fr`

Lastly compiled the **1/3/2023** (day/month/year, you know, like in any sane civilization).

1. Zig: real-life example with LibIPC

I really hoped for this video to come way sooner, but hey, LIFE HAPPENED. Also, thanks a lot for your very positive comments on my first video, I hope you'll like this video, too.

1.1 Introduction

In the past few weeks I did a library called LibIPC. This library provides inter process communications, making processes to exchange messages. I create a service and clients can come talk to it. Simple. No need to dig too much in what this library actually does for now, I'll explain that in another video. However, this piece of code forced me to explore many aspects of Zig. Today, I'll to provide a feedback on what I actually used, not in any particular order. Despite making a video on Zig (almost two years ago), I didn't code in Zig since then. So, this feedback is from the point of view of someone just starting (for real) with Zig. And I hope you will find this interesting.

1.1.1 Quick reminder

Zig still isn't ready for production. However, probably everything you've learned a few years back on the language itself is still valid. Changes happen mostly in the standard library.

1.1.2 In this video

- Documentation (official, ziglearn)
- OS abstraction
- Read and Write structures
- A few language constructions
  Anonymous structures serving as options, switch, orelse...
- Misc functions
  Timer, a bit of networking, Logging, Built-ins...
- Bindings
- Errors

1.1.3 Not in this video

- Parallelism and concurrency stuff
  - async, threading, etc.
- Build system

1.2 From C to Zig

LibIPC was firstly developed in C. Code wasn't too horrible but quite redundant, I repeated a few things regarding error cases. Every function returned a structure with a code (which was an enumeration telling if an error happened or not) and eventually an error message. To avoid this redundancy, I did some macros, but it quickly did get out of hand. I'm not too ashamed of my C library, but I wasn't confident either even though my tests worked.

I rewrote the library in Zig, and redundancy was gone. But since both libraries aren't exactly the same (and I'll spare you the details), it's hard to make a fair comparison. Let's say that despite my macros the C library was about 2 000 lines, and my Zig library is about half that. I think that for the same features, a rough estimate would be about 40 % fewer lines in the Zig version. So the code doesn't have ugly macros and still would be 40 % smaller.

1.3 Documentation

Official documentation of the standard library improved a lot since my last video. For example, when browsing function signatures, structures can be clicked on, leading to their

documentation. That's very basic but it wasn't there!

However, documentation is still experimental and there is a massive room for improvement. I'll give you a few examples I encountered during the development of LibIPC. But it's fine since it was completely expected and Zig developers said it several times: work has (kinda) just begun on documentation.

As a first example, let's follow std.MultiArrayList. We can see this function:

```
//.items is:
fn items(self: Self, comptime field: Field)
  []FieldType(field)
```

What is Self? Self is @This.

```
const Self: type = @This();
```

This is fine since it is everywhere in the standard library, "Self" means the current structure.

Okay, but what is "Field"?

```
const Field: meta.FieldEnum(S) = meta.FieldEnum(S);
```

And up to this, when FieldEnum is selected, there is a list of a single parameter which is a "type", but parameters of what, there is no function here? So documentation is still confusing (or just broken) on some parts.

Documentation can also be frustrating regarding types that are related to a specific OS (basically all C types). I was searching for a type, this type depends on a sub-type, which depends on the OS, which ultimately... cannot be documented automatically. Example:

```
std.fs.File.Mode => const Mode: "mode_t" = os.mode_t;
os.mode_t => const mode_t: "mode_t" = system.mode_t;
```

But again, that's fine. These C types can easily be found in the code, the ultimate source of truth.

Finally, some other types aren't correctly documented. For example, Writer types (such as in std.fs.File or std.ArrayListAligned):

```
const Writer: if (...) { ... } = if (...) { ... };
```

But I think I know why this one is a bit hard to document. We'll see that later.

So, what's the state of documentation? I'll say that it came to the point where it's actually useful. I don't remember reading the documentation at all a few years ago, now I do use it and it works for me almost everytime. I just presented a few examples where documentation fails so you won't be surprised: paint is still wet.

So, how to produce this documentation? One way I used:

```
zig build-exe -femit-docs src/main.zig
```

This creates a `docs` directory with the documentation of our program (everything that is mentionned in the file, directly or indirectly). To serve it, I use `darkhttpd`.

```
darkhttpd docs/ addr 127.0.0.1 port 8000
```

And thanks to Zig documentation being a web application, there are no latencies whatsoever. That's really great to switch from a page to another without waiting at all. Some shortcuts make the browsing delightful, even though it's not perfect in any way. A search doesn't provide the most useful results first, scrolling is often required.

Beside the official documentation of the Zig standard library, a few other documentations popped up over the years. I mostly used one of them: ziglearn.org which is a good source since there are a few examples. But it's not up-to-date, and it seems stale: no new chapters in a very long time. I wanted to learn how to do bindings (meaning building Zig code to use it in C and other languages). I wanted to do that more than a year ago, and it was missing in the chapter 4, and still is! Also, I wanted to learn about the event loop since libIPC is built upon it. But, bad luck, there isn't documentation on that either. I finally did my own event loop, which only is a simple call to the **poll** syscall, and that's will be fine for now.

1.4 Language constructions

1.4.1 Anonymous structures serving as options

Using anonymous structures as options is widespread in the standard API, since it's really convenient. An example of this can be seen in the *std.net* namespace.

```zig
const Server = struct {

    pub const Options = struct {
        some_option1: u32 = 128,
        some_option2: bool = false,
    };

    pub fn init(options: Options) Server {
        ...
    }
};

// No need to create a Service.Options structure.
var x = Server.init(.{.some_option1 = 10});
```

An anonymous structure is used as an associative array of optional parameters. No need to instanciate a **Server.Options** structure. Again, code like this is everywhere in the standard library.

1.4.2 orelse keyword

The *orelse* keyword helps writing clearer code. It is working with an optional value, and in case the value is *null* the right part of the keyword is executed.

```zig
var value = db.get(key) orelse return error.notHere;
```

Orelse is a small improvement from C that makes the code slightly more concise using a simple and widespread construct. And with**defer** and **errdefer** k eywords, previous allocations can be safely freed without explicit reference in the error case.

Also, you can use **orelse** to provide a default value.

```zig
var value = db.get(key) orelse 10;
```

This construction exists already in many other languages, it was just missing for C. That's fixed.

1.4.3 Switch

First, I really like that the compiler is crazy fast. My workflow is based on not to paying any attention to the documentation regarding errors, but just trusting the compiler. The compiler tells me what are the possible errors, then I chose what to do. For example, some errors can be safely ignored, like when I try to create a directory that already exists.

```zig
// Create the run directory, where all UNIX sockets will be.
std.os.mkdir(rundir, 0o0770) catch |err| switch(err) {
    error.PathAlreadyExists => {
        log.info("runtime directory ({s}) already exists (ignoring)", .{rundir});
    },
    else => {
        log.warn("runtime directory ({s}): {any}", .{rundir, err});
        return err;
    },
};
```

Another pretty similar example, working with environment variables. I want to do stuff if some environment variable is set (*IPC_NETWORK*), and there are three possible cases. First thing, the variable exists, it is pushed in the *network_envvar* variable, and the function goes on. Second, the variable doesn't exist, in this case that's still fine, I just return from the current function. Last case, there is a problem that isn't just a missing environment variable, it could be serious like being out of memory, so I return the error.

```zig
var network_envvar = std.process.getEnvVarOwned(fba, "IPC_NETWORK") catch |err| switch(err) {
    // error{ OutOfMemory, EnvironmentVariableNotFound, InvalidUtf8 } (ErrorSet)
    .EnvironmentVariableNotFound => { return; }, // no need to contact IPCd
    else => { return err; },
};
```

1.5 Misc functions

1.5.1 A few built-ins

Why do I talk about this? Because out of context, built-ins functions could be intimidating, specially for people coming from high level languages. But once you actually encounter the problem they solve, they are completely natural to use.

First, a built-in is just a function that comes directly from the compiler. No need to import anything. That's it!

Why are they useful? There is a video about this by Loris Cro called "A look at Zig's built-ins" posted two years ago. `https://www.youtube.com/watch?v=V0sthxzzN3U`

For example, I used a few built-ins (such as **@as**) mostly because I had to manage low level code. The **@as** built-in is for type coercion.

> *[@as] Performs Type Coercion. This cast is allowed when the conversion is unambiguous and safe, and is the preferred way to convert between types, whenever possible.*
>
> — official Zig reference

In clear terms, you tell the compiler you know types are different but they are compatible so it has to perform a safe conversion.

```zig
index = @as(usize, 0);
origin = @as(i32, 0);
```

@as is useful in many cases so you'll probably use it, but most built-ins are specific! Don't feel overwhelmed by reading the Zig reference, you can safely ignore them for now. Despite the fact that I did low-level code, I only used a few built-ins.

In normal code	
as	type coercion
truncate	truncate an integer
enumToInt	convert an enum to int
Within exchange-fd	(very specific code)
ptrCast	type coercion
sizeOf	size of a type
TypeOf	type info
alignOf	memory alignment
errorName	use error name in a string
intCast	type coercion

1.5.2 Zig API looks like libc API

A good thing coming from C, Zig standard library provides a similar API you're used to. A few examples:

libc	Zig std
open(path, flags,...)	std.os.open(path, flags, perms)
read(fd, buffer, size)	std.os.read(fd, buffer)
memcpy(dest, src, size)	std.mem.copy(dest, src, size)
mkdir(path, mode)	std.os.mkdir(path, mode)
unlink(path)	std.os.unlink(path)

And it works the same way on all operating systems. Yes, even on Windows, despite not really using file descriptors... more on that later.

And in the case you just want to call the libc directly:

umask(mask)	std.c.umask(mask)

1.5.3 Timer

From now and then I want to know the duration of some operations. In C, this involves a structure that isn't as simple as it could be.

```c
struct timeval tv_1;
struct timeval tv_2;

gettimeofday(&tv_1, NULL);

// ... some operation...

gettimeofday(&tv_2, NULL);

int nb_sec_ms       = (tv_2.tv_sec  - tv_1.tv_sec) * 1000;
int nb_usec_ms      = (tv_2.tv_usec - tv_1.tv_usec) / 1000;
int time_elapsed_ms = (nb_sec_ms + nb_usec_ms);
```

In the Zig standard library, there is a simple structure (**std.time.Timer**) that does the job perfectly:

```
var timer = try std.time.Timer.start();

// ... some operation...

var duration = timer.read() / 1_000_000; // ns -> ms
```

1.5.4 Logging

As we saw in some examples, a logging system can be used to provide runtime information when something happens. This can be an error, a warning, an information or a debug message. `std.log` is straightforward. Depending on the compilation mode, some of these instructions may or may not be compiled. For example, unless in Debug mode, debug prints won't be compiled.

```
std.log.err("something bad happened, I'll probably just crash", .{});
std.log.warn("something potentially bad happened", .{});
std.log.info("simple info", .{});
std.log.debug("hello this is debug", .{});
```

Your logs can also be "scoped", meaning that you can provide a bit of context. In this case, you create a "log" structure that can be called as `std.log`, but printed string will have a short prefix.

```
const log = std.log.scoped(.libipc_context);
log.err("something bad happened", .{});
log.debug("hello this is debug", .{});
```

1.5.5 ArrayList

A structure I used a few times: **ArrayList**. I just needed to store a dynamic list of something.

```
var al = ArrayList(SomeStructure).init(allocator);
al.append(.{ .value = 30 });
var v = al.items[index].value;
for(al.items) |*it| {
    log.info("{s}", .{it});
}
var item = al.swapRemove(index);
al.deinit();
```

This is an example of about everything I did with ArrayLists. Init, append, loop over items, remove an item based on its index, then I free the array. Very straightforward, simple.

Also, at some point I had to create two ArrayLists sharing their indexes. When I added a value in one of them, I added something in the other. When I removed an entry at an index I had to remove the same entry in the other ArrayList (same

index). Why? Forget about it. I will spare you the details. But in this case the MultiArrayList structure (advertised many times by Andrew), seems relevant. So, I'll probably use it in a near future.

1.5.6 Networking structures

Networking is performed through a few main structures. **StreamServer** and **Stream** structures represent a server and a client. They enable working with sockets and their specific error set, and that's mostly it. After that, there is the **Address** structure, which is generic enough to handle different addressing, such as IPv4, IPv6, and also UNIX socket paths. Having worked with C networking structures in the past, I have to say this is an improvement.

And why is this so great to use while it was a pain in the ass in C? Several reasons: because Zig has default values for structure attributes; anonymous structures use as options; dotted notation and namespaces; and surely because of the error system. None of that being complex features!

```
// connectUnixSocket -> Stream
var stream = net.connectUnixSocket(path)
// Stream has many read and write functions,
// including 'reader' and 'writer'.

// Listen on an UNIX socket.
var server = net.StreamServer.init(.{});
var socket_addr = try net.Address.initUnix(path);
try server.listen(socket_addr);
```

1.5.7 Tests

Testing code in Zig really is simple.

```
zig test src/main.zig
```

Test code blocks in the standard library taught me most of the language. For a first stab at an API, that's great. I won't show to you much tests, just read standard library tests.

1.5.8 Exchanging file descriptors

UNIX sockets, that I use intensively in LibIPC, can share a file descriptor to another process. A server can open a file or a socket then send it to a client, for example.

This rather obscure property is actually used in LibIPC. I did a few functions to exchange file descriptors thanks to a few redditors. They provided me parts of very specialized low level code, that I completed. Clearly, my code isn't perfect in any way, but it works.

I wanted to talk about this because I think it can be useful to have this in the standard library. So, if you have time, please, be my guest and make it happen. Code is free.

1.6 OS and Zig abstractions

A well-design operating system provides abstractions to painlessly work with your hardware. That's almost its entire job: to create a simple environment for developers. Working with files is a good example of this.

Opening a file is asking for a file to your kernel. You provide a path, your kernel provides a number on Unix systems and on Linux. Why a number? Because it is related to a table your kernel has regarding your process. Number 0 is your input (when you type on your keyboard). Number 1 is your standard output (when you print stuff). Number 2 is your error output. When you open files, you get new numbers: 3, 4, etc.

Then, to read or write something in your file, you use this number to tell what file you're working on. So, to write something in your file, in C you write something like this:

```
write(fd, buffer, size);

// Example
write(3, "hello", 5);
```

And a socket is just another entry in the same table. Direct benefit of this: you can use the exact same functions as before. A socket is a number, which you can use exactly as any other file descriptor.

So, I repeat. On UNIX and Linux, the API is simple: you get an integer for whatever you wanted to reach (socket or file), then you have access to a set of read and write functions, and that's it. Simple, coherent, elegant.

Nice, why I am talking about this? BECAUSE OF COURSE IT'S DIFFERENT ON WINDOWS. And I think that's one of the main reasons why it is so difficult to write portable code. We have beautiful abstractions on well-designed operating systems... that we cannot use anymore. Thus, the language (or its standard library) has to overcome differences between operating systems, it is forced to drop this simple and elegant API and make its own. Doing portable code involves (some) complexity.

But, good news everyone! The Zig standard library actually handles this complexity for us, and the **std.os** namespace is full of simple portable functions. The API tends to be as simple as on Linux and Unix systems while being mostly portable. That's even one of the things that got me interested in Zig, a few years back.

Also, complexity in the Zig standard library is stacked. Let me explain: in Zig you can access syscalls directly. No sugar. See the **std.c** namespace. But, if you want to benefit from the zig error system (and to get mostly portable code), you can use the **std.os** namespace, which is more or less direct syscalls with a very light overlay. Finally, a more "modern" approach could be considered with specialized structures for everything. The standard library provides structures such as File, StreamServer and so on. Zig can look like high-level code, as an object-oriented language for example.

So, if you want just C, you have it. If you want portable C with a less dumb error management, you have it. And if you want a library with high-level structures with tons of specialized functions, you have it, too.

As we'll see, having high-level structures doesn't imply (too much) code redundancy.

1.7 Reader and Writer structures

These both structures are very nice to read and write streams of binary data. Reader and Writer are specially relevant for network packets, since they are just serialized data. The UDP protocol for example is described in a simple 3-page RFC, its format is: src port, dest port, length, checksum. In Zig, using a Writer, this may look like this:

```
//udp_msg is a Writer
udp_msg.writeIntBig(u16, 9000);   // src
udp_msg.writeIntBig(u16, 80);     // dest (port 80, HTTP)
udp_msg.writeIntBig(u16, 100);    // length (100-byte message)
udp_msg.writeIntBig(u16, 0x8301); // checksum

//writeIntBig = Big-endianness, network endianness
```

That way, you don't have to handle an index, which is one of the perks of these structures.

Reader and Writer are also relevant to read and to write any file with a binary format. In this case, you can open a file then get a Writer structure out of it.

```
//in std.fs.File
fn writer(file: File) Writer
```

So, Reader and Writer structures are a convenient way to read and write streams of data. Files, network, in-memory data, logs, etc.

LibIPC is particularly simple in that regard: a LibIPC packet is a length and upper-layer payload. That's it. Even UDP is more complex than that.

Sometimes, I wanted a writer structure for in-memory data.

```
// Stacked buffer
var buffer: [1000]u8 = undefined;

// Stacked buffer -> fixed buffer stream
var fbs = std.io.fixedBufferStream(&buffer);

// Fixed buffer stream -> writer
var writer = fbs.writer();

// do something with the writer
writer.write("stuff\n");
writer.write("important stuff\n");
writer.write("another stuff\n");

// Get a slice of what has been written
var stuff = fbs.getWritten();
```

But that can be a bit complicated. In the case you just want a **snprintf** function, meaning to print in a buffer, there's **std.fmt.bufPrint** for that.

```
var buffer: [1000]u8 = undefined;
var path = try std.fmt.bufPrint(&buffer, "{s}/{s}"
          , .{ ctx.rundir, "simple-context-test" });
```

Now, the good news! You can actually see the documentation for a Writer. The Writer within**std.io** is correctly documented. And it's also quite confusing, because it's a function returning a structure. This structure actually is used to create other Writers. Let me explain. This**std.io.Writer** function is a way to create Writer structures, all having the same API, the same functions.

Why not just using a Writer structure once and for all? Because depending on where you write data, you'll get different errors. What can fail when writing to a file is different to what can fail writing to the network. Also, writing to a file is different from writing to an ArrayList structure. But because everything is kinda the same besides that, only three parameters are needed for this function to provide a Writer structure in all kinds of situations.

```
pub fn Writer(
    comptime Context: type,
    comptime WriteError: type,
    comptime writeFn: fn (context: Context,
       bytes: []const u8) WriteError!usize,
  ) type {
```

As you see in this function signature, three compile-time known parameters are required. The context, meaning the type of the structure where to write data, such as a file, a buffer or a socket. The second parameter is the set of errors that can happen when writing data. Finally, the function to actually write data. Both *Context* and *WriteError* are used within the

function signature *writeFn*. That is valid code and typing is verified by the compiler.

```
pub const Writer = io.Writer(File, WriteError, write);

pub fn writer(file: File) Writer {
    return .{ .context = file };
}
```

I wanted to show to you this function because **std.io.Writer** is a good example of generic programming. That's a way to ensure a consistent API accross different situations without requiring either new concepts (such as interfaces) or code redundancy. Sure, there is the**comptime** concept, meaning that a parameter has to be known at compile time (and can be a type instead of a variable). But that's manageable and there is no need to introduce Object Oriented Programming just for that. A function returns a structure, both being trivial and pervasive concepts in almost every known programming language.

As you probably guessed it by now, the Reader is the exact same thing.

```
pub const Reader = io.Reader(File, ReadError, read);

pub fn reader(file: File) Reader {
    return .{ .context = file };
}
```

1.8 anytype

Now we know about Reader and Writer, let's see an example. You have a function which writes stuff. Whatever it could be, like logs, serialized data, anything. You may want to write a function that is writing to a writer instead of writing directly to your target. This way, the code can now be tested and even used in a different context.

So, let's say you want to write a function taking a writer as a parameter. What is the type of the writer? You want your function to work with any writer, not just the File.Writer structure for example. Can be painful to write, while all you want is just something that accepts the functions you'll call on it.

```
// create a server path for the UNIX socket based on the service name
pub fn server_path(self: *Self, service_name: []const u8, writer: ??) !void {
    try writer.print("{s}/{s}", .{ self.rundir, service_name });
}
```

One easy answer to that is to write "anytype" as its type, and you're good to go. That is a little hack I used since I wasn't sure about the right type I should use, and it works. That's basically duck typing.

1.9 Bindings

A binding is a way to use code from another language. Most general purpose languages can use code (functions and structures) from C if they write bindings for it. How this is translates in actual code depends on the language. For example, Zig can directly import C code, no need for bindings. In Crystal, a binding can be a simple function signature declaration.

But in my case, I want to enable other languages to use my library written in Zig. To that end, we can "export" a Zig function this way:

```
export fn ipc_context_init(ptr: **Context) callconv(.C) i32 {
...
}

export fn ipc_service_init(ctx: *Context, servicefd: *i32,
    service_name: [*]const u8,
    service_name_len: u16) callconv(.C) i32 {
...
}
```

Thanks to this, it is possible to call Zig code from any language which can call C code, including the C language.

I made a git repository with an example of this, without any sugar, so the different interactions can be understood. This shows how to create a library in Zig which can be imported everywhere. I didn't try to export structures, just functions.

1.10 Errors: still room for improvement

I encountered few confusing errors. I had the presence of mind to keep a trace of at least one of them. But keep in mind that this doesn't matter much: developement was fairly straightforward and painless.

I got the following error when the result of a function call (**fetchSwapRemove**) was a single value (anonymous hash).

```
src/switch.zig:125:37: error: binary operator '|'
  has whitespace on one side, but not the other.
        self.db.fetchSwapRemove(fd) |k,v|{
```

This error is just plain confusing, it didn't help one bit.

I got another, when I had a function with a large array located in the stack. I guess this broke the compiler since it runs some of the code at compilation-time. But I tried to recreate the error and it doesn't fail anymore, so I guess this was fixed.

1.11 Hexdump

Lastly, since I did some networking code, I wanted to print an hexadecimal dump of packets I sent or received. I used a library I just copy-paste but it wasn't working propertly so I kinda rewrite the whole thing.

It's not in a separate repository, but you can copy-paste it!

1.12 Conclusion

My experience is simple: coming from C, I saw no drawbacks working with Zig, at all. An experienced C developer can use Zig as in C and it would still more concise and readable. I even bet the generated binary would be almost similar. In case you want to enjoy the Zig error system, AS I ADVISE YOU TO, you can use **std.os**, and maybe a few other namespaces such as **std.mem**. To me, that's a sweet spot. And if you want more, the standard library provides many convenient and portable structures. All of that without introducing any new complex concept. So my point of view about Zig hasn't changed since my first video. Zig really is C without the bullshit and I'm now even more confident to say it.

To be fair, I don't know if there is a domain where Zig really is outstanding. I will use it instead of C for writing libraries and system applications; programs I want to see working on all operating systems and architectures, including constraint ones, such as a raspberry pi, some routers, etc. Also, I'll use Zig to write libraries. I do prefer other languages, such as LISP and Haskell, but I consider the Zig tooling to be a better fit for system programing.

Also, most problems I mentionned (such as documentation) are well-known and expected.

1.13 Other projects and next videos

I did a learn x in y minutes on Zig A FREACKING YEAR AGO but it's not working because of some bullshit with syntax highlighting (a library that wasn't properly updated). It may or may not change soon.

Beside, I want to continue doing videos on Zig, at least a few. On the build system for example, maybe the network API, async and a few other things.

As a side note, about the build system.

> *When you have a hammer, everything looks like nails.*
>
> — Somebody, somewhere (at some point in time)

To me, this reasonates a lot with the Zig build system. Some consider it great, I have my concerns. Since it's a big subject, I'll dedicate an entire video on the matter.

Beside Zig, I spent a certain amount of time doing other stuff I want to talk about.

- Sure, I did LibIPC and I want to talk about it.
- I did it to serve as a foundation for the rewrite of netlib.re (freed network) which is a website I did 8 years ago to provide free domain names, currently in Perl (and French). And I want to rewrite it in Zig and probably Crystal and PureScript.
- Also, I had a lot of fun doing a package builder based on makefiles for a toy operating system.
- I learnt roff, which is an excellent tool to produce documents (first developed in the 70s), and I want to talk about it. And I did a template to help new users getting started with roff.
- And (with roff) I started a book on Haskell and I did a few book reviews and summaries.
- I learnt LISP and I played with it doing a database library to store documents (no SQL, no DBMS, no dependencies) with indexes mapped on the file system.
- Something I already did in Crystal, too.
- Also, do you know CBOR? CBOR is a very efficient serialization library, a binary JSON. Probably the most concise, simple and efficient serialization library, ever. Well, I did transparent bindings to CBOR for the Crystal language. Meaning that you can automatically serialize and deserialize any Crystal object in CBOR.

So as you can see, I did stuff, I want to do stuff, and I think it's time to share it with you. So I'll try to post more videos, more frequently than once every two years, at least. It really is good to be back. As the last time, everything I talked about can be found in the links below. If you want to talk, you can post comments. Have a good day.