

The background features a complex network of thin grey lines and dots, forming a web-like structure on the left side. Scattered across the entire background are various triangles of different sizes and orientations, some with solid outlines and others with dashed or dotted outlines. The overall aesthetic is minimalist and technical.

THE BUND LANGUAGE

Writing your first “Hello World!” script



INTRO

Brief introduction
Into the Bund language

01

NAMESPACE

Dive into the first
important concept of the
Bund

02

STACK

The core of
the Bund language

03

TABLE OF CONTENTS

04

HELLO WORLD!

Anatomy of the
“Hello World” script

05

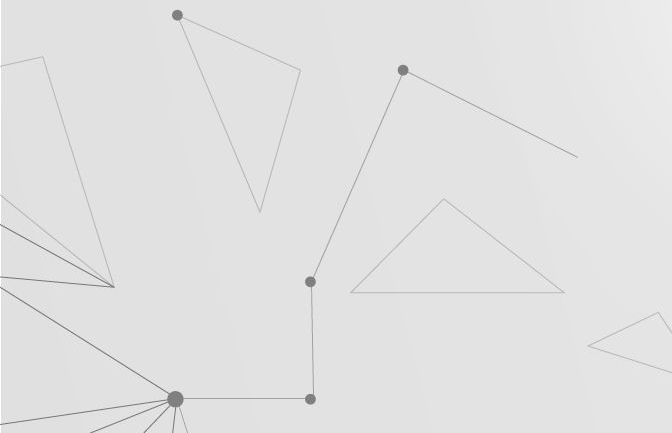
RUN IT

How to run
your scripts

06

FIN

What I can say
at the end



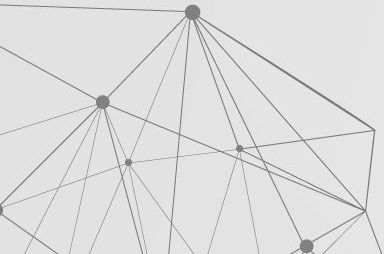
01

INTRO

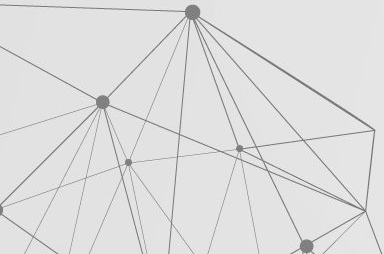
Brief introduction into the Bund language.



Bund is an interpreted high-level general-purpose programming language. It is shared a features from different “ancestors” while not being a next generation of any of them. Out of many BUND ancestors, I can specifically mention FORTH, Lisp and Haskell (as very distant relative).

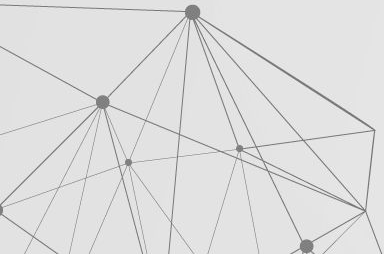


Like a FORTH, BUND is stack-based, but it is different from FORTH in respect that it is not a procedural language. It is functional. It is also different in how BUND stack machine works. Like LISP and Haskell (remember, that one is very distant relative, but relative nevertheless), BUND is functional. It supports lambda calculus with named and anonymous lambdas. Lambda functions in BUND are functionally strict, they are taking only one parameter and always return a value. And we will discuss them in a separate tutorial. In addition to functions, BUND offers separate concept of “operators”, which essentially functions, but taking two parameters and returning a value at all times. Like in Haskell, BUND offers a “variable immutability” and unlike in Haskell and LISP, BUND does not have variables. Unlike LISP, BUND virtual machine operates with Stack not with List and this BUND’s feature differs it from Haskell as well.



BUND is dynamically typed language, where the values somewhat monadic^(*) and immutable. How they are immutable ? Because once they are placed on the stack, you can not change them. How come that there are no variables ? Because all the data are existing on the stack. The only data existing outside of the stack is the lambda and operator definitions.

^(*) Monadic data types are containers, for which we can define imperative execution sequence of some operations over stored value.



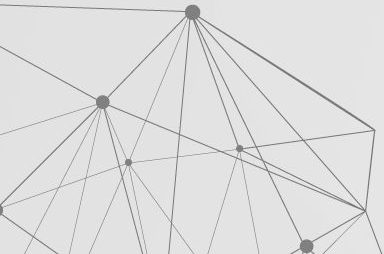
02

NAMESPACE

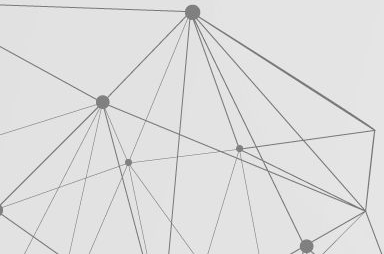
This is first and foremost concept of the BUND



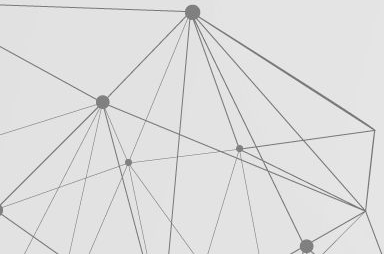
As you already know, BUND is stack-based language. But there are more than one stack in the BUND application. There are many stacks. Stacks everywhere. You can define as much stacks, as memory allows. And each stack defined in own "Namespace". "Namespace" is somewhat misleading definition, the "Stackspace" is more accurate description of what it is, but concept of the "Namespace" as something which is isolating parts of application from each other is more familiar to the developers and I will use this term.



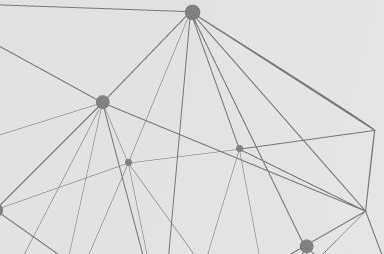
Namespace defines the stack. Every time when you are defining a new "Namespace", you are creating a new Stack and associate it as "current" for the BUND stack machine. Every time when you are referring an existing "Namespace", you are swapping current stack with already existing stack. All data between switches are preserved. You can not create your script outside of the "Namespace", because in that case, stack machine will not know where to place the data.



“Namespaces” could be “named”, i.e. you are defining the name for it, or “anonymous”. In that case, BUND interpreter will assign a temporary name for that namespace.



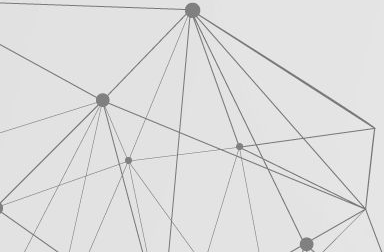
“Namespaces” are flat. Even if you can define them deep in hierarchy of namespace references, they are essentially defined in a flat key-value global structure. Where the key is a namespace name and value is a namespace definition. Including a Stack.



Named “Namespace” are defined by it’s “header”, where the name of the “Namespace” enclosed between [and : and termination of the namespace, indicated by ;; . All terms inside namespace will be operated within the scope of the current namespace and stack. Except of “Namespace” declaration, which are global by definition.



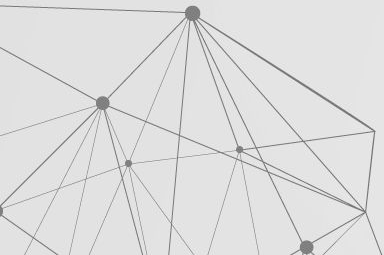
```
1 [ main :  
2   [ another/namespace :  
3   ;;  
4 ;;
```





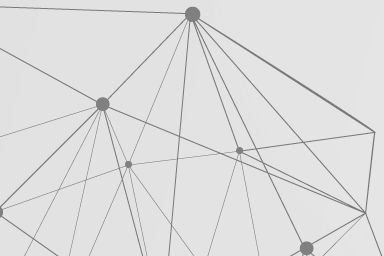
You can define anonymous namespace between (and). All terms inside anonymous namespace will be defined on Namespace own stack. You will not have an access to the name of anonymous Namespace. But there are ways to send data in and out of anonymous Namespace.

1 ()



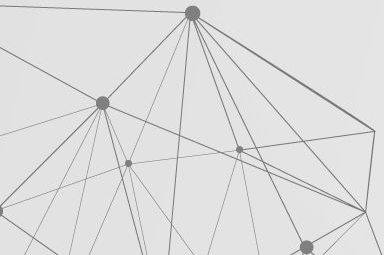
You can define and use named Namespaces inside anonymous ones.

```
1 (  
2   [ main :  
3     [ another/namespace :  
4       ;;  
5     ;;  
6   )
```



And vice versa. Definition and use of the named and anonymous
Namespaces are dictated only by your application logic.

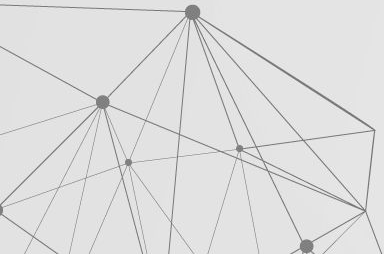
```
1  [ main :  
2    [ another/namespace :  
3      ( )  
4    ;;  
5    ( ( ) )  
6  ;;
```



If you enable debug output on BUND interpreter, you will see how interpreter creates, returns and otherwise manipulates with Namespaces.



```
19:39:52.806 | DEBUG | ENTERING Block  
19:39:52.806 | DEBUG | Creating NAMESPACE: 8c0d9aa9-c0d7-4e8d-af1b-6110e9c2d2db  
19:39:52.806 | DEBUG | Setting a Root Namespace
```



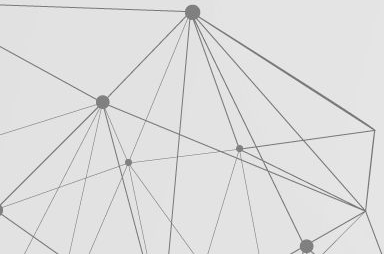
03

STACK

The core of the BUND language



BUND is a stack-based language. All data that BUND is processing exists only on the stack. There is no separate variable namespace. Unlike in FORT, BUND stack is in fact double ended queue and you can manipulate if you are pushing or pulling data from the “Head” or from the “Back” of the stack. More about this later



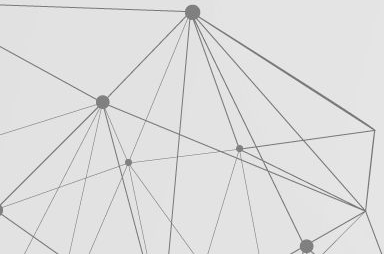
In the "Normal mode", BUND pushes data to the back of the stack. Every time, when new data is ready, existing data at the Back of the stack pushed towards Head.

```
1 ( 'This is an answer' 42 )
```

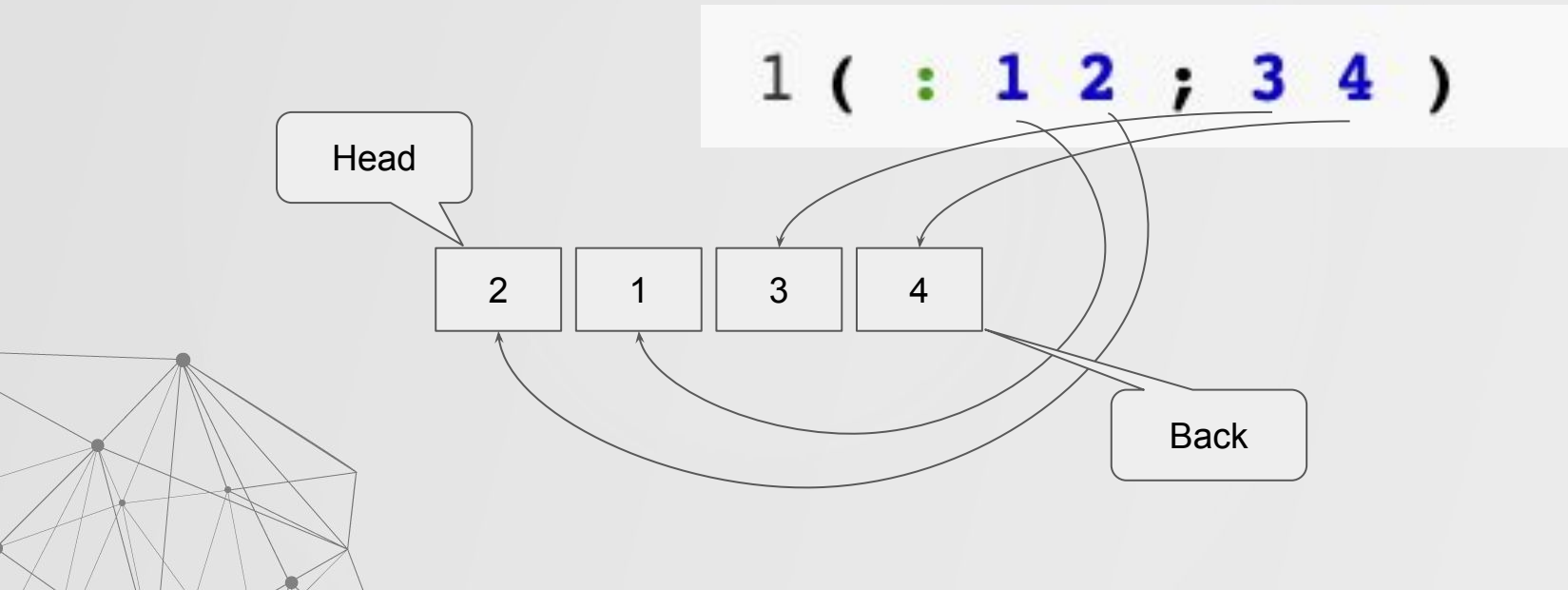


Command `:` instructs BUND stack machine to change the stack mode and begin to push next data term to the Head of the stack

```
1 ( : 'This is an answer' 42 )
```

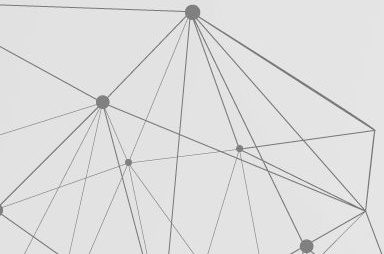


Command ; instructs BUND stack machine to return back to “normal” mode.
All data will be pushed to the back of the stack.



As I mentioned above, BUND is dynamically-typed language and you can mix and match data in the stack without explicit declaration. Supported data types (as of current):

- ❑ Float numbers. Represented by float64
- ❑ Integer numbers. Represented by int64
- ❑ Unsigned integer numbers. Represented by uint64
- ❑ Strings. Unicode supported.
- ❑ Boolean values (true and false)
- ❑ References to the lambdas and operators
- ❑ Anonymous lambda functions
- ❑ Datablocks. Vectors, which can mix different data types.
- ❑ Float, Integer, Unsigned integer blocks. Vectors containing floats, integers and unsigned integers only.



04

HELLO WORLD

Anatomy of the “mother of all programs”.

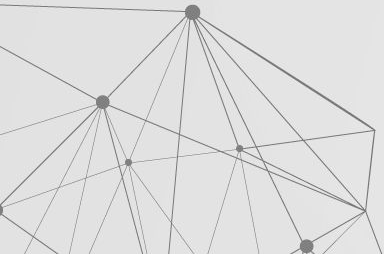


This is a one of the most basic programs, All it does is printing a string to the standard output. This program have three components in it. First one is a Namespace definition. You can not define your application outside of the Namespace.

```
1 [ main :  
2     "Hello World" println  
3 ;;
```

Definition of the
named namespace
called main.

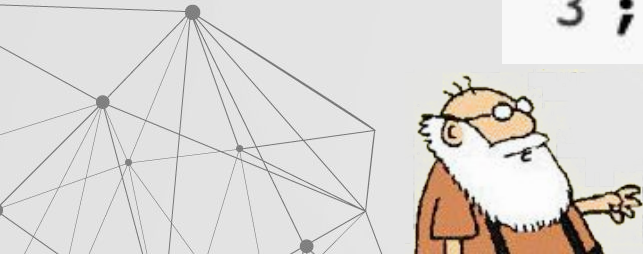
Termination of the
Namespace use



Next, we declare a string and define its content. The strings are Unicode UTF-8 byte arrays. Once declared and defined, string are pushed to the stack according to the stack Mode, ether to the Head or to the Back. While pushing a new data, BUND stack machine pushes existing data deeper into a stack. There is no empty cells in the stack. Stack grows from Head and from Back indefinitely.

Declaration of the
unicode string,
placed to the back of
the stack.

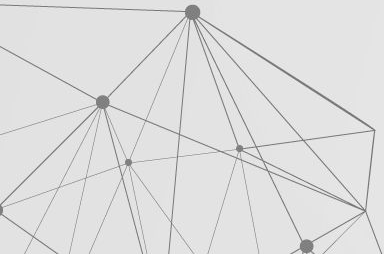
```
1 [ main :  
2   "Hello World" println  
3 ;;
```



And then, we are calling the named lambda function. Named lambda function differ from anonymous lambda, in the way that named lambda can be referred by the name and reference, and anonymous lambda by reference only. First, BUND will try to locate lambda defined in current namespace, so, you can redefine system-wide println if you wish. Then if it not found, it is searching through system-wide lambdas. If lambda is found, BUND stack machine executes lambda, in current Namespace. System-wide println lambda takes value from the stack (according to the stack Mode) and prints this value on the system console.

Calling of the named lambda function. We do know that this is the function as operations do not use alphanumeric characters in it's names.

```
1 [ main :  
2   "Hello World" println  
3 ;;
```



05

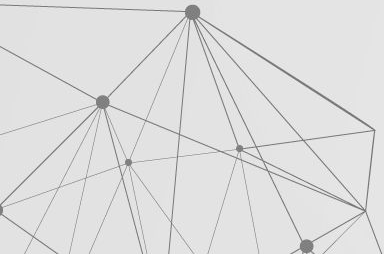
RUN IT

Anatomy of the “mother of all programs”.



To run BUND scripts, you will need a BUND interpreter. I've tested the interpreter on OSX and Linux. Likely, you will be able to compile it on all UNIX-like systems, on which you must have a GO of version 1.16. Result of compilation is a single executable binary named bund. You can copy this binary anywhere in the path. I will not go deep into BUND compilation, some day, I will create "BUND: Developer Manual", where I will go deeper into subject.

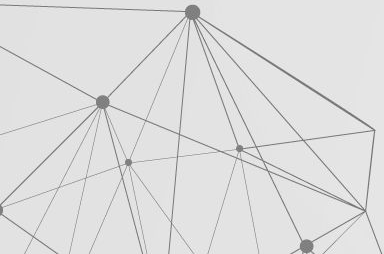
Bund binary supports contextual help, available through --help CLI parameter.



At the time of preparation of this manual, full BUND interactive shell is not operational and you do have two methods of running BUND code:

- ❑ Evaluation of the one-line string, available through command eval^(*).
- ❑ Execute BUND script from file, available through command run.

(*)While you can run a snippets of the code through eval command, the true designation of that command is to testing and debugging BUND interpreter.



Here is an example of running HelloWorld code snippet using eval command:

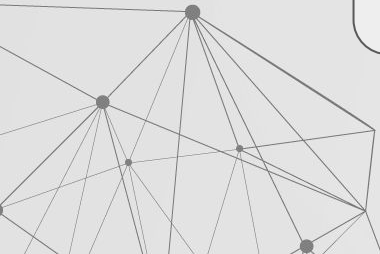
- ❑ To see the lexer output, you can pass `--lexer` CLI key to the eval command.
- ❑ To see the parser output, you can pass `--parser` CLI key to the eval command.

Here is your
HelloWorld code
snippet

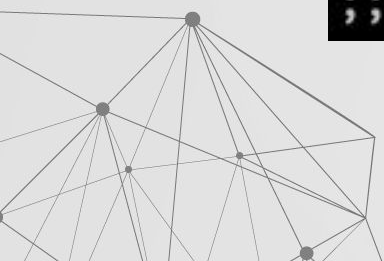
```
→ derBund git:(main) ./bin/darwin/bund eval "[ main : 'Hello world' println ;;"  
Hello world  
→ derBund git:(main)
```

Here is the output
that you are
expecting

You are running
BUND in eval mode.



Or, you store your script to the text file (at this point, this script shall look familiar to you, don't it ?) and execute interpreter for that file.



```
→ derBund git:(main) x cat examples/helloworldsimple.bund
[ main :
  'Hello world'
  println
;;
```

Or, you store your script to the text file (at this point, this script shall look familiar to you, don't it ?) and execute interpreter for that file. Declaration `--main` tells BUND in which file we do have our Main function.

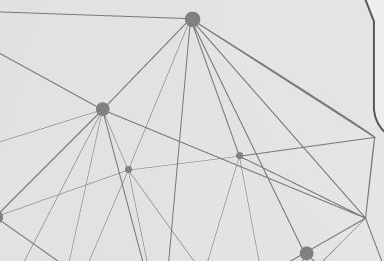
You are executing
BUND interpreter in
run mode.

And here is the file
with your script

```
→ derBund git:(main) x ./bin/darwin/bund run --main ./examples/helloworldsimple.bund
Hello world
→ derBund git:(main) x
```

Here is the output
that you are
expecting

You shall specify,
that this file contains
main code of your
application



Or, you store your script to the text file (at this point, this script shall look familiar to you, don't it ?) and execute interpreter for that file. Declaration `--main` tells BUND in which file we do have our Main function.

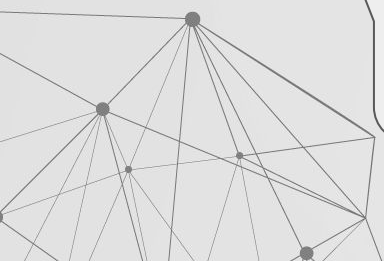
You are executing
BUND interpreter in
run mode.

And here is the file
with your script

```
→ derBund git:(main) x ./bin/darwin/bund run --main ./examples/helloworldsimple.bund
Hello world
→ derBund git:(main) x
```

Here is the output
that you are
expecting

You shall specify,
that this file contains
main code of your
application



To look “under the hood”, to see what is going in the BUND interpreter while it is processing your commands, you can add `--debug` to the `bund` command.

You are entering into a ‘main’ namespace and set it as a root namespace.

Here, you are executing script.

```
16:47:35.153 | DEBUG | [ BUND ] Executing ./examples/helloworldsimple.bund
16:47:35.156 | DEBUG | Creating VM: ./examples/helloworldsimple.bund
16:47:35.158 | DEBUG | Attempt to InLambda() Stack doesn't exists
16:47:35.158 | DEBUG | ENTERING Namespace: main
16:47:35.158 | DEBUG | Creating NAMESPACE: main
16:47:35.158 | DEBUG | Setting a Root Namespace
16:47:35.158 | DEBUG | String Value: 'Hello world'
16:47:35.159 | DEBUG | CALLING: println
Hello world
16:47:35.159 | DEBUG | EXITING Namespace: main
16:47:35.159 | DEBUG | NAMESPACE stack is empty
16:47:35.159 | DEBUG | No errors detected
16:47:35.159 | DEBUG | Exit requested. N=0
16:47:35.159 | DEBUG | [ BUND ] tsak.Fin() is reached
```

You are placing a string into stack and call `println` command.

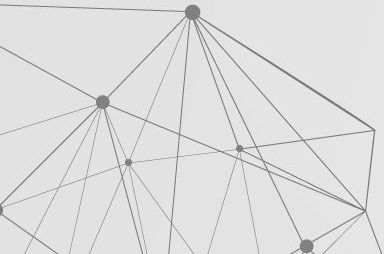
06

FIN

What else can I say ?



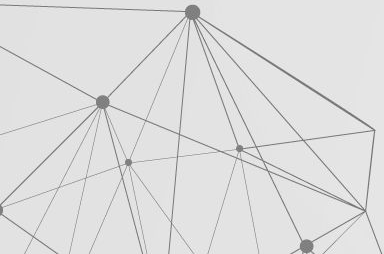
There is no shortages of the programming languages in the world. Some of them, old and gone with a History. Like MUMPS. Some of them, old and still kicking in one way or another, like LISP. Some of them, more or less newcomers in this field, like Julia. But you just learned a few basic ideas about perhaps newest addition into a programming languages ranks - BUND. As all programming languages, this one was created because author feels that "other languages are not expressive enough for my tasks". Or maybe he just want to experiment with some ideas that he have.

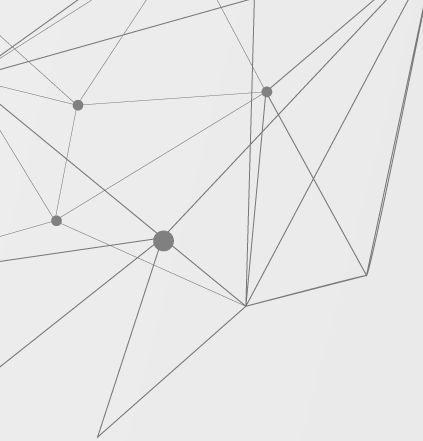


Regardless of author initial intentions, he is looking to make BUND to a practical tool, that can be useful and will allow people to create expressive, non-bloated and simply beautiful code.



```
( 'day wonderful a have shall you' )
```





THANK YOU

Does anyone have any questions?

vladimir.ulogov@me.com

