{VA:MP}

versatile application meta-programming

VAMP - Orientation

V e r s a t i l e   A p p l i c a t i o n   M e t a - P r o g r a m m i n g

Introduction

As its acronym suggests, VAMP is indeed versatile.
This document is to familiarize yourself with the basics, installing VAMP, and make a "Hello World!" app.
However tedious it may seem reading through some of the text below, take your time, it won't hurt :)

The default VAMP setup may seem larger in file-size than some "normal" programming languages; however, in comparrison with software stacks such as the infamous "LAMP", -VAMP's default "package" size is close in comparison.

What you get with the VAMP package -and what you can do with it, well...

About VAMP

Here's a few things you should know:

- it has an interpreter, a compiler, and a transpiler -which are all configurable & extensible
- it can transpile any mime-type it has a mime-library extension for
- it is securely scalable across several projects, servers -or domains, for n-tier software architecture
- it can be used for both server-side & client side and its syntax is clean & easy to learn
- it can create, read & listen on any permissible resource, path, socket, & child-process
- it can be used as an active database, or used for programming, configuration and content authoring
- it has a configurable "view" mechanism, both for CLI & GUI - which have similar instructions
- it is built to be stable, so it will not crash (under normal conditions)
- it runs in its own directory which is your runtime "project root" & every project has its own daemon
- its directory and file structure is part of the live runtime structure where folders are "nodes" (objects)
- structures (like "nodes") can be defined inside VAMP files, not only folders
- its structure is loaded at "boot time" and lives in memory; more on this in a bit, keep reading
- the live structure entities (and attributes) are referenced as "pointers", so it runs fast
- anything in the runtime has "meta-attributes", called "aspects" some of which are "intrinsic"
- the intrinsic concepts & aspects are designed for rapid prototyping and development
- the mathematical expressions are designed to be "safe", so it handles things like "devision by zero" well
- the aritmetic operators handle any data-type, -not only numbers

The power of VAMP is in its structure, expressive language, simplicity, useful concepts, extensibility, portability, and runtime stability; alas -it is up to the systems engineer to lay out the proper architecture.
VAMP is not a "framework". It sticks with the structure you define, so it is only as complex as your own structure. The structure you define is your folders & files, together with their relationships with one-another.

.. but hang on a second.. - no frameworks?
YES! - Why let your creative thinking be handicapped by limited ideologies?

You create your structure; VAMP lets you stick to it; -unless you explicitly need your structure mutable during runtime; which is also supported.
Of coarse you can make a "framework" -or "boiler-plate" with VAMP, all this means is that with VAMP you don't really need a bulky framework to get things done as it has a lot of useful tools and methods built into the language and its libraries, which are extensible. The point here is: "freedom with rules", and you are the author of these "rules" - or rather: "structure".

Extensibility

VAMP's default mime-type library comes with transpilers for:

- HTML
- CSS
- JavaScript
- PlainText

The mime library can be extented either by installing it from the repository, or building your own.
The punch-line here is that you can code in one language and have your instructions translated (or compiled) to any other mime-type, depending on what you're authoring and what the target platform is of coarse.

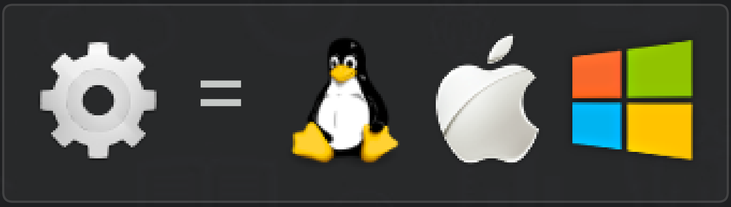
So what's the big deal?
Well, as we all know, some languages are great for some things and others are better at other things, depending on what you need to do, or on which platform it should be done on; however, these languages have different syntax, and some languages have the power with which you can blow your foot off, or do something really interesting - if done right.

Now, using the "best practices" & the right tools for whatever language you're authoring may become tricky and most of the time -developers make "boiler-plates" in which ever language they use so they can have handy tools at their disposal during new project development, or maintenance work.

Wouldn't it be great if we could have a language that bring these tools together, for which ever language, and you use one language to program a dozen other languages - at once? A language that does not get in your way, but is expressive and extensible, so you do not need to repeat yourself all the time.

TO SOLVE THE PROBLEM WE NEED TO RECOMBINE OURSELVES FROM WITHIN
AND THE CHANGE IS INITIATED BY A SIMPLE THOUGHT A SIMPLE ELECTRIC IMPULSE
A STRING OF INSTRUCTIONS

You can also compile your VAMP project into a stand-alone exectutable; which could be anything, from a systems API or service, a plug-in or module for another service -to native desktop GUI applications.
Depending on your development setup, the VAMP compiler can compile executable that runs natively on Linux, OSX & Windows.



above :: illustration of VAMP's native binary compatible platforms

So based on the fact that we can extend any VAMP feature, and also create & edit its mime-libraries; it means that you can make a "boiler-plate" once, tell VAMP how to use it and which words relate to what and use it anywhere in your future VAMP projects.

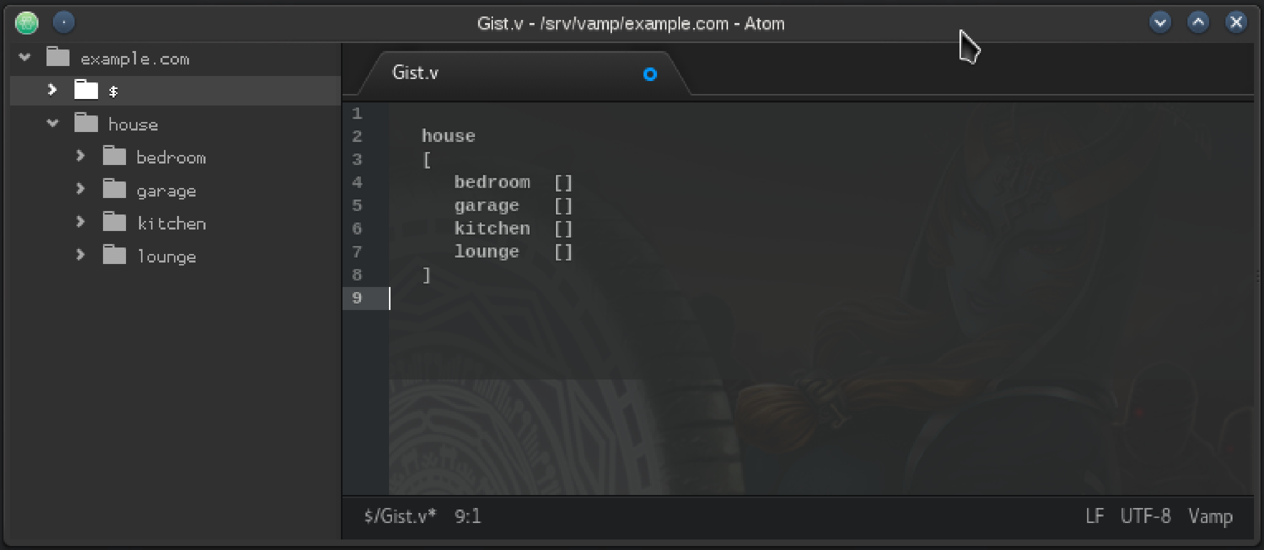
By "anywhere" I really mean it: server back-end, data-base, web interface, native desktop GUI apps; -even interactive 3D web applications -or installable 3D games; videos with syncronized audio & sub-titles; still images with super-imposed overlays, etc, etc..

It is important to know that this is in deed possible, but VAMP does not come with these compilers & transpilers by default, so, if it exists in the VAMP repo, by all means, help yourself, otherwize, buy it, -or get someone to develop it, or build it yourself - then sell it - or donate it to the VAMP repository for free, if you want :)

However you decide how to use VAMP is up to you. The point is that it's an incredibly powerful system, that is easy to learn, simple to use, easy to extend, and applicable anywhere.

Structure

VAMP is all about structure. From your project's folders & files to the code in those files - it is all used as a whole.
Its content operators shows very close correlation between folder-tree and VAMP code.
Here's an example:



```

house
[
    bedroom  []
    garage   []
    kitchen  []
    lounge   []
]

```

above :: illustration of folder-tree structure in correlation with VAMP code

View - GUI & CLI

VAMP comes default with configurable & programmable "view" mechanisms that use the same methods to "show" things to the user. This is directly related to VAMP's "Bios" (basic input-output system).

The GUI part is available for both web browsers and native desktop applications, depending if you compile your project to native binary -or run it as a server/service on the internet.

above :: VAMP's default GUI web browser compatiblity list

The CLI part is for system administrators and hardcore devls (developers). It can be used to create -or manage projects on the fly. The VAMP runtime supports "hot-loading", so you do not need to restart your project daemon when minor changes are made to the structure; remember, it works like a data-base; you don't need to restart a database if you insert or delete information, that would be quite pointless :)

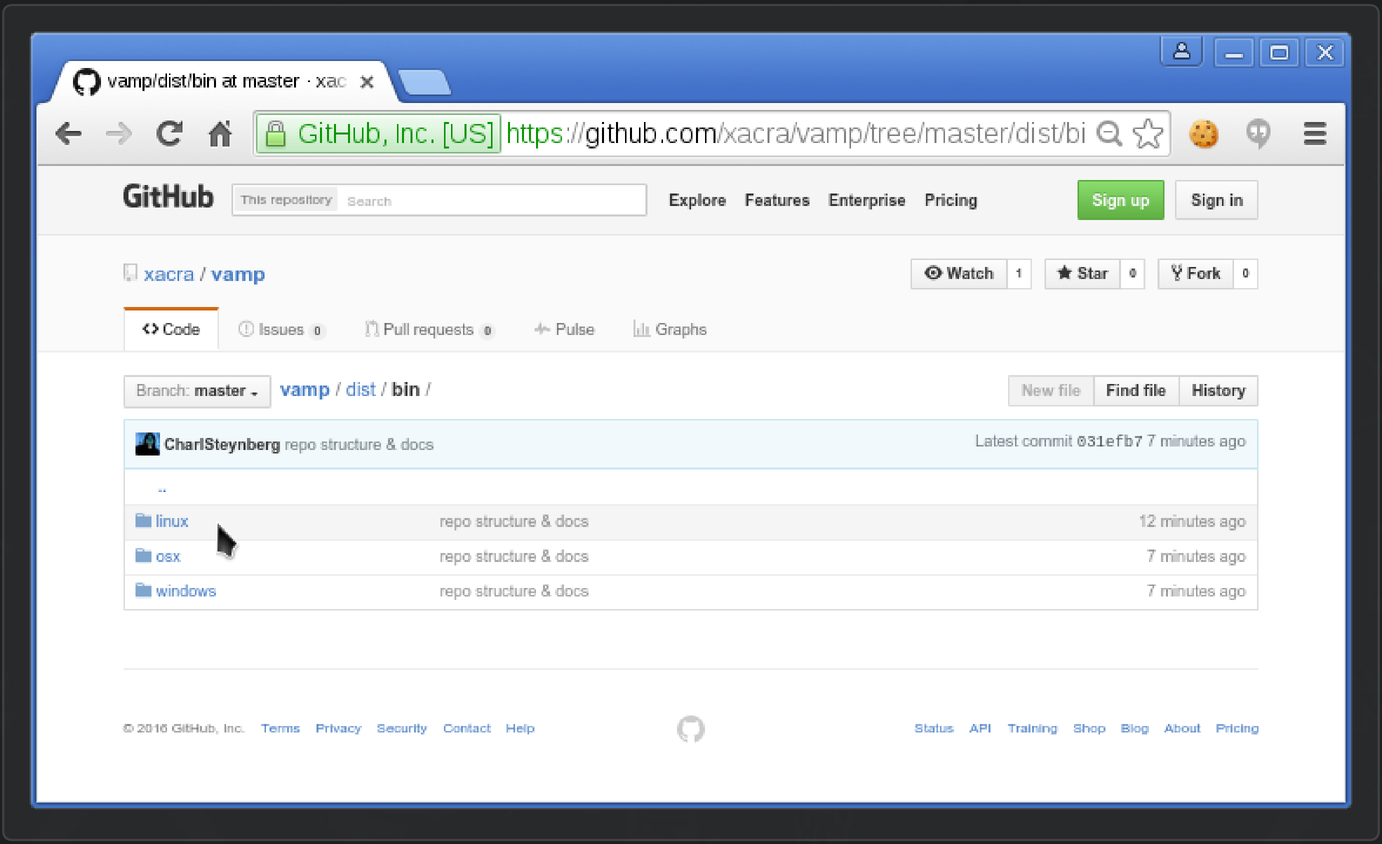
When working with the CLI you have a lot of power though. You can "save" the new structures you create in code -to disk, -where you also have the option to save it as a "vamp file" or tree structure. This is covered later.

The VAMP Bios (basic input - output system) adapts its input and output according to the transmission used; however, the developer does not need to apply a different way of sending messages, or reading input to and from any of these. This can be configured -per Bios; you have the freedom to customize pretty much anything.

This needs elaboration, but I hope this introduction shared enough info to inspire your into reading the rest -and start using VAMP!

Installation

VAMP can be installed from its repository here: https://github.com/xacra/vamp
Navigate to the `dist/bin` folder, in which you will find a list of folders named according to the supported operating systems.


vamp/dist/bin at master · xac

GitHub, Inc. [US] https://github.com/xacra/vamp/tree/master/dist/bi

GitHub This repository Search Explore Features Enterprise Pricing Sign up Sign in

xacra / vamp Watch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Pulse Graphs

Branch: master vamp / dist / bin / New file Find file History

CharlSteynberg repo structure & docs Latest commit 031efb7 7 minutes ago

..

linux	repo structure & docs	12 minutes ago
osx	repo structure & docs	7 minutes ago
windows	repo structure & docs	7 minutes ago

© 2016 GitHub, Inc. Terms Privacy Security Contact Help Status API Training Shop Blog About Pricing

above :: the VAMP repository on GitHub

Linux

- Download the `vampSetup.sh` file from the `linux` folder.
- From the terminal, navigate to where it downloaded and make sure it is executable.
- Type: `./vampSetup.sh` and follow the prompts.
- To test if VAMP was sucessfully installed, type: `vamp -v` and you should see something like: `0.1.0`

OSX & Windows

- Coming soon!

Development Environment

VAMP does not require a specific development environment at all. You can simply install VAMP on which ever host-machine you want and use a plain text editor for coding, and a terminal to run or compile your projects; however if you're serious about your work you should at least run your test server on a separate host (or virtual machine) - and use some a proper text editor for programming.

Throughout this documentation, the file extension: `.v` is used for "vamp" files. Your local machine's operating system may already have the ".v" file extension associated with some other application; however, you do not need to use the `.v` file extension exclusively; - see the "Hello World" section below for more info on which alternative file extensions to use.

File extensions in VAMP are important. Not only is it easy to spot for a human, but also, the VAMP interpreter can recognize combination file extensions, like: `style.css.v` and process it "explicitly"; -where if the `.css.` part is missing, then it is processed "implicitly" - which could be slower.
If you have the VAMP SDK package installed, the icons in your local file browser will show recognizable icons for VAMP files, and combinations also.

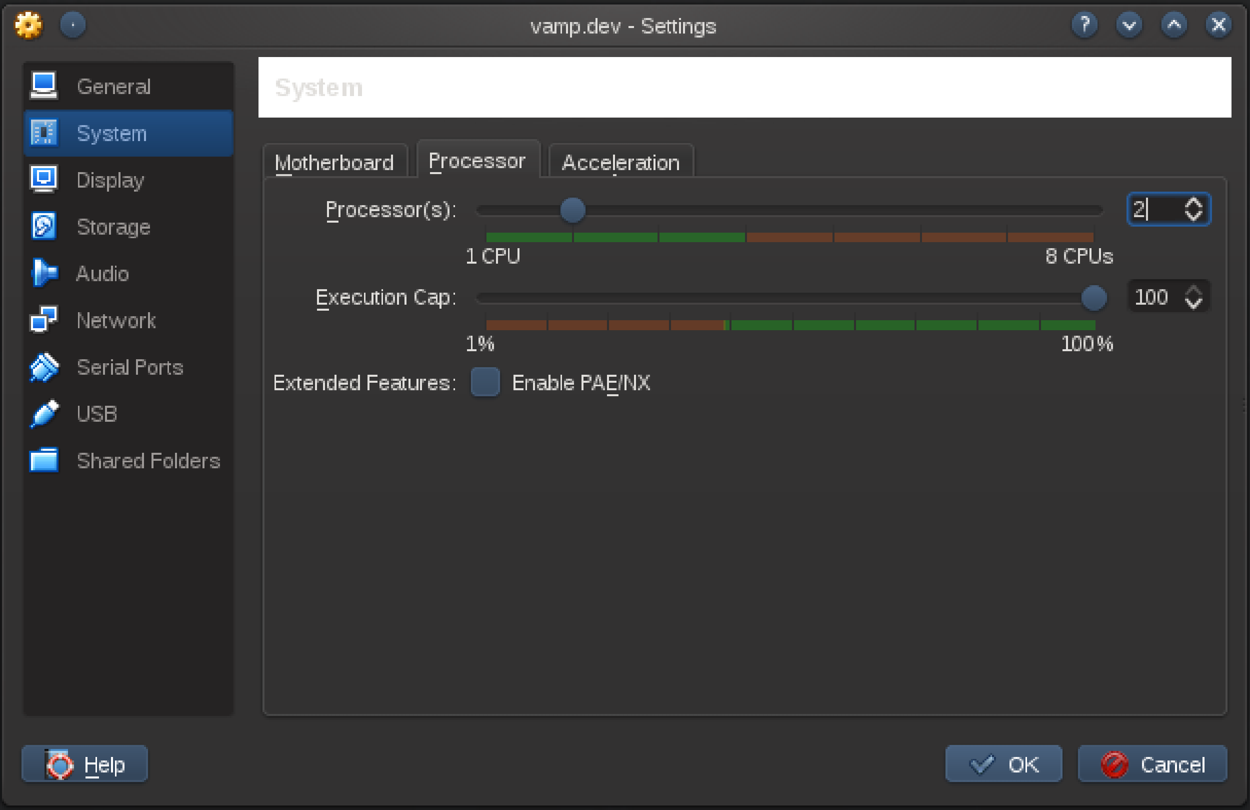
Virtual machine

There are many VM solutions out there, the solution I recommend is using `VirtualBox` with `Arch Linux` running on a machine you configured; but you can use anything you like.

To take advantage of "compiling" it is recommended that you assign at least 2 virtual CPU's to your VM -with at least 2Gb of RAM. The machine does not need a GUI, so you can just use a minimal installation and set it up the way you want. You can also launch your VBox "headless" (without a "window") from the terminal and have it run in the background.

You can also "mount" a folder (path) which is on your virtual machine into a folder of your local machine. This is very useful for quick navigation, screen-shots, etc - using your local machine's file-manager to manage files & folders on your Vbox.

Consult the internet for more info on this as there are plenty tutorials on how to do these things - which is not in the scope of this document, but I thought it's worth mentioning to have something that "just works" without much hassle. It is very useful; not only for VAMP, but for your other projects as well.



Processor(s): 2
1 CPU 8 CPUs
Execution Cap: 100
1% 100%
Extended Features: Enable PAE/NX

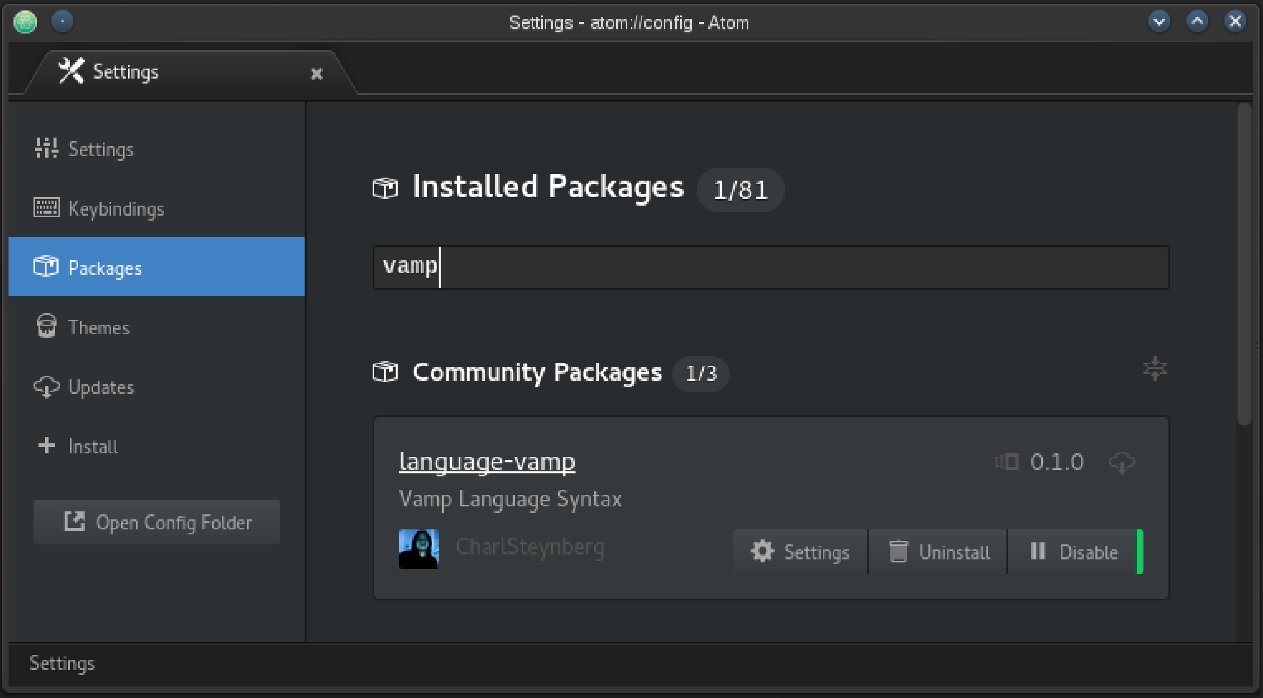
above :: VirtualBox

Syntax highlighting

It is important that your text editor supports syntax highlighting of some kind as it makes it easier to identify what your're typing.

If your text editor does not support VAMP syntax by default, there may be a syntax highlighting package for your editor in the VAMP repository.
Have a look here: https://github.com/xacra/vamp/tree/master/devl/ext/editor

At the time of this writing there is a syntax highlighting package available for `Atom`, so you can just install it.



above :: VAMP syntax highlighting in Atom

Hello World

This excercize is actually very quick, but for the sake of getting a firm grasp of VAMP, we'll take it step by step. Note that you do not need to do the "mounting" thing, and you also do not need the VAMP SDK; these are just to make things more "user friendly" for yourself while you develop VAMP projects.

Before you dive in, read the following first - so you know exactly what to expect:

1. Any VAMP project needs its own folder (which defines the runtime scope)
2. VAMP files can have any of the following file extensions: `.v` `.va` `.vam` `.vamp`
3. `$` (meta) files are optional, but mandatory in `project root` and refer to the folder they are in
4. items in folders are loaded sequentially in ascending order by name, so `$` files are loaded first
5. the contents of VAMP files are statements in the context of the file-name and folder hierarchy
6. statements are evaluated by context, parsed from left-to-right and top-to-bottom
7. the return value of any context is determined implicitly or explicitly

For clarity & brevity, I'll assume the following:

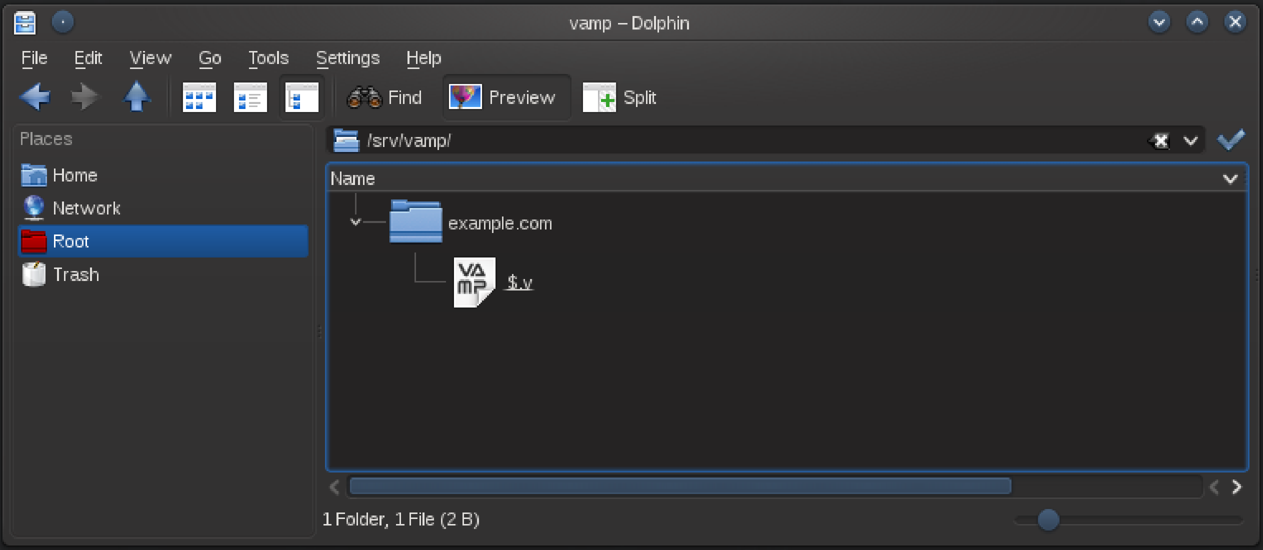
- you have installed VAMP on your development server and have installed the VAMP SDK on your local machine;
- you're using a `Linux` machine as project server (where VAMP is installed);
- you have mounted the folder path of this server in which your VAMP projects will reside --into your local file-system;
- you either have a terminal window open that is connected to this dev server via SSH, or you're using a rich IDE with SSH support -OR -simply just the open VBOX window that runs your dev server.
- your text editor (or IDE) supports navigating file systems.

With all the above in in mind, it's time to kick some booty!

Let's get started:

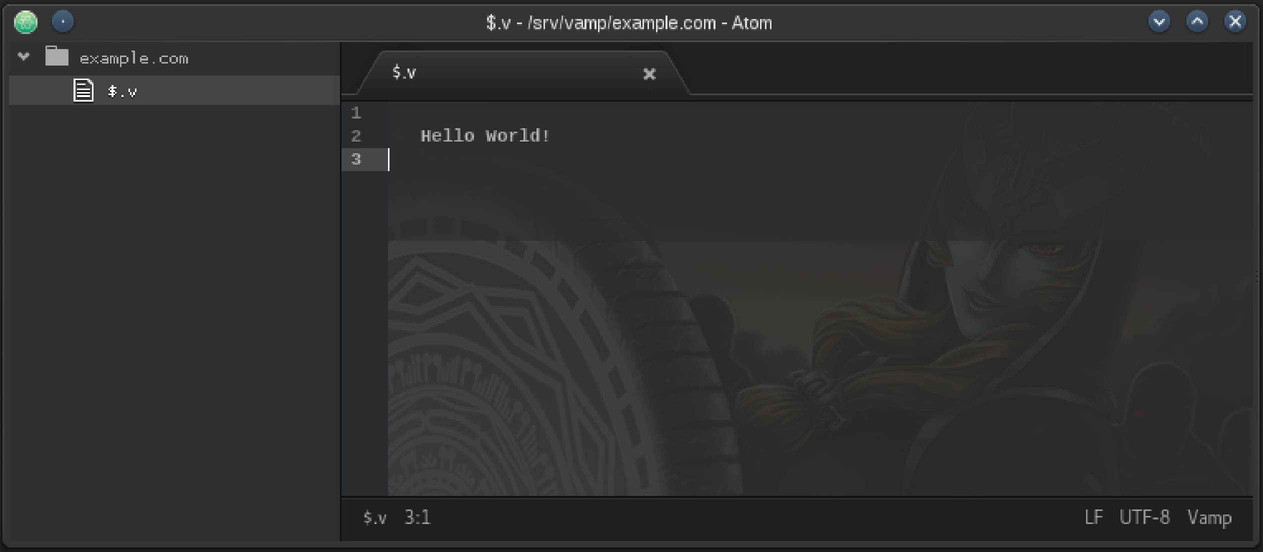
- Within the target file-system, navigate to your main projects (plural) container (folder).
- Create a new folder in there, so the path to it will be something like: `/srv/vamp/example.com`
- Inside this `example.com` folder, create a text file named: `$.v` (take note of the file extension)

Your `project root` structure of "example.com" should look something like this:



above :: project root of "example.com"

Using your editor, open the `$.v` file and type `Hello World!` inside it, and **save**.
Your project should look something like this:



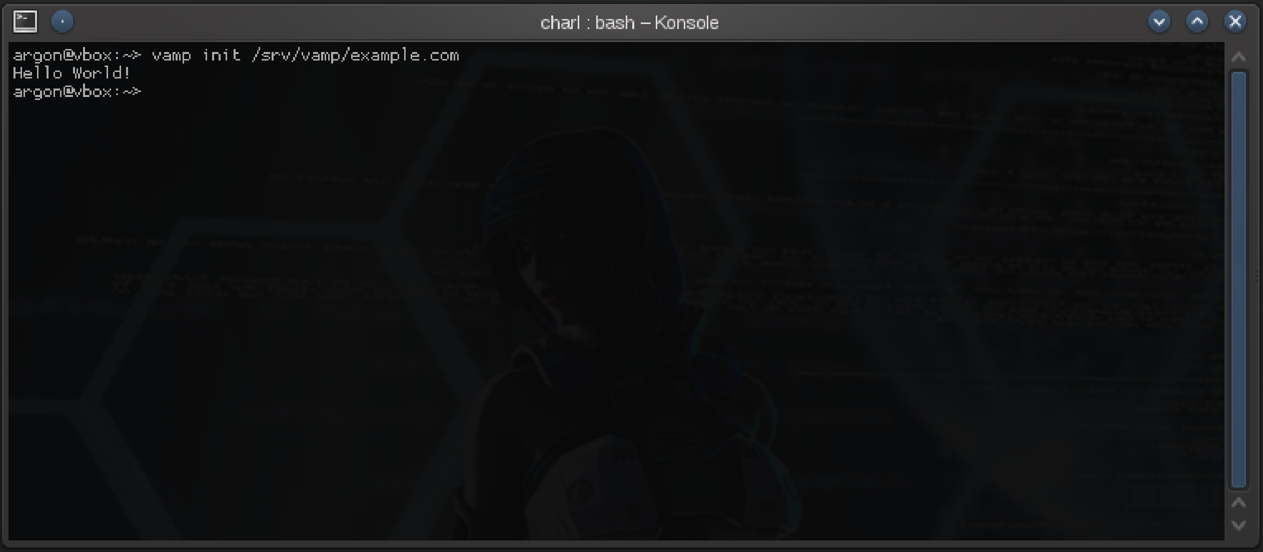
```

Hello World!

```

above :: project: "example.com"

Testing

To run your project, use your connected SSH interface (terminal) -or VBox window, and inside it type something like: `vamp init /srv/vamp/example.com` and hit "enter" (or return) on your keyboard.

If all went well, you should see something like this:



```
argon@vbox:~> vamp init /srv/vamp/example.com
Hello World!
argon@vbox:~>
```

above :: project: "example.com" output

If your results are not as above, then consult the "troubleShoot" doc, but if all is good, then: Congrads!

Operators

Operator categories

The operators in VAMP are categorized in terms of usage. These are:

- comment
- context
- delimit
- reference
- logical
- arithmetic

This document is for referenece purposes and will be covered in more detail later.

Comment Operators

Comment in VAMP is used both for documentation and developer reference. The porpose of all these comments is to have code look well explained, and has the ability to make `Markdown` -or HTML documentation from VAMP files. It is recommended to "minify" your code when publishing to a live environment as this speeds up loading & parsing, not only for VAMP, but with any other language or library commonly used.

These are the comment operators:

```
--       double-dash(+space)  block-comment, at line start, ends with `--` or `::(+newline)`
#        hashtag(+space)      line-comment, used anywhere
##       double-tag           line-comment, used anywhere
#!       shebang              line-comment, used at doc start
!#       bangshe              line-comment, used anywhere, or in docs as highligted tags
.:       secton-begin         line-comment, used for: `section start`, or: `thus`
::       double-colon         line-comment, used anywhere, or section end
//       double-FWD-slash     line comment, used anywhere, or section start
\\       double-BCK-slash     line comment, used anywhere, or section end
---      triple-dash          line-comment, used for code -or comment-block separation
/* */    slash-star pair      block-comment, used anywhere
```

"Line-comment" is typically ended with a `newline` character, but "block-comment" is typically ended with a pairing sequence, as shown above.

The `bangshe` (or "banshee") line-comment can be used in both coding & documentation as it has the same effect, wich is to highlight certain key-words and let the accompanying text stand out from the rest. The "accompanying text" can be whatever the developer types in there; the "quoted text" below is only an indication of what the key-word could mean; however, if no text is typed next to these key-words, then the quoted text below will be used instead.

The words & colours of the "banshee comment" can be configured, but the defaults are:

- `TODO` - dark grey - "to be done in the future"
- `TEST` - green - "not thouroghly tested yet"
- `HELP` - pink - "i need help with this"
- `NEED` - purple - "the specification -or technology needs to be updated"
- `DONE` - blue - "completed and tested"
- `BUSY` - orange - "work in progress"
- `WARN` - orange - "it could work but may fail under specific circumstances"
- `HACK` - orange - "working but not properly done and may have side effects"
- `FAIL` - red - "broken, fix this -or remove it to prevent issues"
- `VOID` - black - "deprecated - do not use in the future"

Below is an example of how to use these comments resulting in well documented code.

```
#!/usr/bin/vamp


----------------------
::      v0.1.0      ::
----------------------



## Heading
--
   Documentation paragraph with some **bold** text
--
!# TODO : not implemented yet




.: bgn section :.
-- -----------------------------------------------------------
   documentaion stuff
-- -----------------------------------------------------------
   "Hello world!"
-- -----------------------------------------------------------
:: end section ::




// begin section -- quick, but not so pretty as above
-----------------------------------------------------------
"Hello world!"
-----------------------------------------------------------
\\ end section
```

Context Operators

The importance of "context" in VAMP is paramount as it defines what any statement -or expression means or implies and to what it applies - and in which scope it can be valid. Comment operators also define a "context" but the contents inside the "comment context" is ignored during runtime.
Your project structure in folders and files define context (scope), as well as the following operators:

```
( )   ::    wrap, express, calculate, call
[ ]   ::    wrap, list, contain
{ }   ::    wrap, describe
" "   ::    text, quoted - unformatted plain text - single-line
' '   ::    text, quoted - unformatted plain text - single-line
` `   ::    text, quoted - formatted text, multi-line, embedded values & expressions
```

Delimit Operators

As with any (normal) language, "spaces" devide "randomtext" into "some words". The same applies here, but just with different punctuation. As with some languages, like "Python", VAMP does not completely ignore "white-space". White-space delimit text into lists & statements where appropriate.
List and statement delimitation does not apply to "quoted text", except when this text is to be parsed.

From the VAMP interpreter's perspective: any comment is ignored and the white-space at the beginning and ending of any context (except quoted text) is removed. Any white-space remaining is reduced to a single white-space character. This is important because in VAMP a white-space character is a delimiter.

VAMP does not require commas -or semi-colons between list items and statements - ONLY if these are separated by white-space. Examples will follow later, but for now, remember that VAMP compiles (or transpiles) into other forms where the "implied" items & statements are realized and treated accordingly. When the compiler minifies VAMP code it fixes the delimiters accordingly.

Speaking of lists; "text" is a list of characters (a character "string"), hence list operations are natural to text.
This brings us to: `indexing`.

Indexing in VAMP starts with 1 (NOT 0) and applies to reverse lookup also. This solves a lot of common problems - especially in boolean truth comparisson where "something found" appears first in the list, -or the first item from the end of a list (or text).
Having to do extra calculations when coding just to compensate for that zero'th key, or to calculate the length of a list just to get to the last item - is insane, and, negative zero is a bit crazy, so, it's important to remember as it will save you a lot of headache.

The following delimiters are valid:

```
,     ::    comma
;     ::    semi-colon
      ::    white-space
```

Reference Operators

VAMP's reference operators together with the context-operators and inheritable meta-attributes are very powerful; you can do a lot more with less code. This does not nescessarily mean it is slow, because VAMP does not (normally) pass whole data structures around, but rather creates pointers to these structures in memory and passes the pointers around instead.

The following operators are used as reference operators:

```
:     ::    is   - name is value, where "value" is in the context of "name"
$     ::    self - aspects of an entity
@     ::    at   - point "at", or "in", to find or retrieve
.     ::    's   - as in: "Dog's balls"  in code:  Dog.balls
```

Here are a few examples of how to use the operators above:

```
foo:bar                    ::    {foo:'bar'}
bar:foo                    ::    {bar:'foo'}

bar:$foo                   ::    {bar:'bar'}
bar:$.foo                  ::    {bar:'bar'}

('foo'):('bar')            ::    {foo:'bar'}

(foo.Name + 'Var'):bar     ::    {fooVar:'bar'}
```

Logical Operators

In VAMP, "logical operators" & "arithmetic operators" must be delimited from text by white-space; else they are just plain text.
Remember that VAMP expressions are always evaluated from left to right, no exceptions (not even multiply). If you need part of an expression to take precedence you need to enclose it inside (any) context operators.

Truth in VAMP is calculated logically. Since we need not worry about "zero'th" keys it is safe to assume that the following is logically false: `undefined`,`null`,`0`,`empty`,`error`,`false` - anything else is logically true.
We will get to the "data types" in a bit; the "logically false" references above are not part of the data-types, so you don't need to remember them, just that those kinds of things are "logically false" and anything else is "logically true".

Here are the logical operators:

```
?     ::    IF    - boolean truth confirm
|     ::    OR    - choose the first boolean true comparable value, "this OR that OR bark"
&     ::    AND   - boolean comparisson used to affirm & compound expressions

!     ::    NOT   - boolean negation
=     ::    same  - strict boolean comparisson
~     ::    like  - loose boolean comparisson
<     ::    less  - is true if "left of" is logically less than "right of", else false
>     ::    more  - is true if "left of" is logically more than "right of", else false
```

Some logical operators can be used together as compound operators.
The logical compound operators are:

- `!=` not same
- `!~` not like
- `>=` more same
- `<=` less same

Arithmetic Operators

In addition to "normal" arithmetic operation, VAMP also uses arithmetic to do data manipulation on various data-types. This saves a lot of time & duplication of function names. More on this later.

These are the arithmetic operators:

```
+     ::    add
-     ::    subtract
*     ::    multiply
/     ::    divide
%     ::    modulus
^     ::    exponent
```

VAMP is extremely versatile as a language, hence it has some rules of what belongs where.
The logical and arithmetic operators do not have any calculation value outside of "expression context", so, they MUST be inside expression context in order to be "calculated" (evaluated) accordingly.

When logcal -or arithmetic operators are used outside of expression context, then it is plain text.

Take note that VAMP requires white-space between operators in "expression context" -with only 4 exceptions to this rule, the following can be used directly in front of anything, -except logical, reference & arithmetic operators:

- `!` boolean negative-truth test
- `?` negative number
- `-` negative number
- `+` positive umber

Context

About VAMP context

Everything in your entire VAMP project structure is based on: "concepts", "contexts", "aspects" and "values".
At "project root" level, everything directly inside it are referred to as "concepts" - which exist in the "context" of your "project root".

The "aspects" of every "context" -or "concept" has "values" which define its attributes, relationships, methods, and contents. The collective for "aspects" is simply referred to as `meta`, represented by the `$` character wich means "self".

Contexts can be defined anywhere in your project tree by folders & files (or symbolic links), as each of these need to be named something.
Contexts can also be defined inside VAMP files using the "context operators".
As you may have guessed by now that you cannot simply access any "path" (folder or file) on the local system (where VAMP is installed), because your VAMP runtime (project) can only access its own contents; however, if you need to access something outside of your project tree, then you can use "symlinks" to do so. These links can be defined either "implicitly" or "explicitly". The reasons for this is both for security and structure simplity, but we'll get to these later.

It is important to note in these docs about references to "implicit" & "explicit". Vamp is designed to make your life easier, so if things get done "implicitly" it simply means: "automatically done the way it it is implied"; where "explicitly" means: "do the followng exactly".

The difference between "implicit" and "explicit" is:

- "implicit" is less coding, but may run a bit slower at runtime (in some cases)
- "explicit" may run a bit faster at runtime, but also may require a bit more coding

With the above in mind, when using the following operators after a named (or unnamed) entity, it usually means this:

- the `:` operator defines -or re-defines an entity (or result) explicitly;
- the `{}` and `[]` and `()` operators defines -or `{modifies}` -or `[extends]` -or `(calls)` an entity (or result) implicitly.

The rest of this doc will explain.

Aspects

Here is an example of how to define some "thing" and give it some properties:

```
ride {color:'blue', roof:'none', status:'notorious'}
```

From the examle above, the curly braces turned the word "ride" into a "named entity".
This means that it got registered as a "named thing" within the context of wherever the code above resides. It automatically gained a meta-property: "Name" and its Name is: "ride".
It has 3 data-properties: "color", "roof" & "status".

You can access these properties by using the "select" operator `.` (dot) like this:

```
ride.$          ::    {color:'blue', roof:'none', status:'notorious'}
ride.Name       ::    ride
ride.roof       ::    none
```

Contents

To assign contents to an entity, we can either define it with the "content operators", or with the "Gist" aspect like this:

```
ride [ 'some text contents' ]       :: implicitly

ride {Gist:'some text contents'}    :: explicitly
```

You can also define an entity with aspects and contents with the `{}[]` operator sequence like this:

```
ride {color:'blue', roof:'gone'}
[
   'seats'
]
```

In the example above, the `{}` denotes the aspects of "ride" and the `[]` denotes the "contents" of "ride".

Functions & Methods

A function (or "called on method") is defined with the `()[]` operator sequence like this:

```
ride ()
[
   'vrooom!'
]
```

In the example above, the `()` denotes the arguments of function:"ride" and the `[]` denotes the "contents" of "ride".
When calling `ride()` it will yield: `vrooom!`.

Functions & methods in VAMP are both synchronous AND asynchronous -depending on they are called.
The `()()` context sequence-operators simply means: "call the return value"; however, the `()()[]` operator-sequence means: "attach this call-back to this call-instance".
The following example will explain:

```
ride()          ::    returns its yield

ride()(yield)   ::    call-back function
[
   :: do something with `yield`
]
```

To explain the above example, you need to know how VAMP handles function (-or method) -calls:

- the "call handler" initiates an instance of itself when a call is made to it
- if no call-back is defined for this instance then it returns its yield
- if a call-back is defined it calls the call-back with its yield when all is done

---

Definition

VAMP offers different ways of defining, modifying & extending things. This is by design as some things are better to define one way and modify in another way, depending on the dynamics of your logic.

The following example shows how to create an entity and modify it after:

```
ride: {}

ride.roof: 'no'
ride.Gist: (ride.roof + ' seats')
ride.Call: ()
[
   'vrooom!'
]
```

The example above shows a rather "long winded" way of creating & modifying an entity, but it shows how things are usually done in other languages; however, it works in VAMP also.

In VAMP, the above can be done explicitly without repeating yourself, like this:

```
ride:
{
   roof: 'no'
   Gist: ($.roof + ' seats')
   Call: ()
   [
      'vrooom!'
   ]
}
```

In the example above, the value of "Gist" is assigned in an interesting way.
Here's how it works:

- when an entity is being defined, its defined aspects are availale to the next statement within the same context
- the expression that follows the `:` inherits the current context so `$` (self) refers to "ride"
- `$` has an aspect named: "roof" which has a text value: "no".

So, the value of "Gist" in the example above is "no seats"; hence, the contents of "ride" is "no seats".
Erpessions will be covered in detail later, but the above should help getting a firm grasp on how things operate in context.

Calls & some expressions

A "call" is an "expression" that operates on the context of what is "called upon".
In VAMP, anything can be "called" "implicitly" or "explicitly" (with arguments), even if "the thing" does not exist as a named entity.

When something is called, it does 1 of 4 things, depending on how this thing being called is defined:

1. if it does not exist then the "return value" would be "Void" (nothing)
2. else-if it has a "Call" aspect (discussed below) then that is `called` accordingly
3. else-if it has "contents" then its contents are `searched` accordingly
4. else its aspects are `searched` accordingly

When an entity is called and it does not have a "Call" aspect, it implies that you are making a "data-lookup" and in such cases, the following apply:

- if the lookup is done on aspects, then "auto-aspects" are skipped
- lookups always yield a list, even if nothing is found -an empty list is returned
- the lookup result list contain pointers to the "real entities", so modifications can be made implicitly or explicitly

Here are some examples of how to "call" things "implicitly"; the comment on the right shows what is returned:

```
ride:
{color:'blue', roof:'none', status:'notorious'}
-------------------------------------------------------------------------

ride()                      ::    [color:'blue', roof:'none', status:'notorious']
ride(`moo`)                 ::    []
ride(`color`)               ::    [color:blue]
ride(`moo` | `color`)       ::    [color:blue]
ride(`roof`, `status`)      ::    ['roof:none', 'status:notorious']
```

Let's go through the example above line by line:

- `ride()` no arguments are given, so a copy of "ride" is returned
- `ride('moo')` "ride" has no aspect named "moo", so: "nothing" is returned
- `ride('color')` the value of aspect "color" in "ride"
- `ride('moo' | 'color')` value of "moo" -or value of "color" in "ride"
- `ride('roof', 'status')` value of "roof" -and value of "status" in "ride"

The example and explanation above shows how things are called "implicitly", meaning: what is "implied" by the call - depending on what it is called upon.

To call things "explicitly" - it works exactly the same, but in this case you specify exactly what to "call upon", like this:

```
ride.$(`roof`, `status`)      ::    ['roof:none', 'status:notorious']
ride.Gist()                   ::    []
```

In the example here (above):

1. the `$` is used to select the "aspects" of "ride", so the call is done on the aspects explitly;
2. the "content" (or "Gist aspect") of "ride" is selected, so the call is done on the contents explicitly -which in this case is "nothing" because our "ride" has no contents.

Data Types

The "data types" in VAMP are divided into 2 categories:

- `Type` primary data-type
- `Kind` secondary data-type

These are global meta-attributes and is automatically defined & updated as entities are created & modified. The data-types also have complimentary globals which are used for comparrison. We say "data-types" here because, that is what it is known as; however, "data" is a very broad term. All information imaginable that can be stored on a disk -or transmitted via any medium is simply called: "data" and so is the code you type in programming as well.

Eventually the data changes as it is interpreted, but we need a way to distinguish what belongs where.
VAMP has a `Data` "data-type" which simply means anything that can be stored as a bunch of characters, from text through to binary data: anything not parsed (yet) is simply just plain: `Data`. Before you start sweating - have a look below - after the examples that follow; you will see that the `Data` primary type has many secondary types.

From here on out we will simply refer to primary "data-type" as: `Type` and secondary "data-type" as `Kind`. So, we can say: "the Type of this", or "the Kind of that". Remember that VAMP is extensible, so you can extend the types & kinds to suit your specific needs, either in your project or in your VAMP base code; -the latter will be available for all your projects.

The concepts (global nodes, functions & constants) you see below are covered in the "concepts" doc, so keep reading ;)

Type

The types are the corner stones in VAMP. Some of these type names are not only static constants, but also functional nodes. However strange and wonderful they are, they do a lot of work for you so you can focus on the project and not spend so much time in persuading the compuer to do what you want. This is especially true with the "kinds" as you will see in a bit.

These are the types:

```
Void        ::    ()                                 ::    undefined
Spin        ::    +, ?, -                            ::    polarity
Nume        ::    1, 0.9, #123.kg                    ::    numbers
Data        ::    'abc', "123", `*.*`                ::    text or binary
List        ::    []                                 ::    list of contents
Node        ::    {}                                 ::    functional object
Fail        ::    (!?/*)                             ::    holds an error
Emit        ::    Emit(every:#100.ms)()[]            ::    interval, event
Time        ::    Time(`2016-03-03`)                 ::    date, time
Bios        ::    Bios.sock(`ws:123.0.0.1:8080`)()[] ::    input and output
```

Spin

The word "spin" was decided on this data-type because of its possible use-cases and what it could mean during conversation -either with coleges -or in your own contemplation of how to do things logically.

This type can be used as "boolean", -or "polarity", -or a "state mechanism", -or "flavor", -or "spin-off type" of anything, -depending on how it is used in your logic.

It is based on the theory and application of "Qubits" (quantum bits).
As you may know, traditional bits have only 2 possible states: 1 or 0 (`true` or `false`). Qubit values are calculated from "probability" in the form of "spin up" & "spin down" as the polarity of an atom rapidly changes.
This means it could be in 1 of 3 states: "positive", "negative", or "both". The latter (both) could be explained as "undecided", -or "not enough pull on either side to make any conclusion".
This concept is adopted in VAMP as "polarity" and it is part of the meta-aspects, named: "Spin".

The meta-aspect: "Spin" has 1 of 3 possible states and its initial value is determined by the `type` of the entity it belongs to and the entity's value:

- `+` : `True`, numbers more than `0`
- `?` : `Void`, numbers equal to `0`, anything else
- `-` : `Fals`, numbers less than `0`

However you decide to use it, being able to set -or change -or view the polarity of something in 3 states is very useful, especially in data-type declarations, and expressions as you will see in the next doc.

Fail

The `Fail` type is a meta-object reference that is only created and returned upon error. VAMP does its best to prevent errors; even so, sometimes you may code something horrendous in the early hours of the morning. In such cases VAMP fails "gracefully". Instead of having the entire process break; the `Fail` type flows along as far as possible until it reaches the output eventually through your own process flow.
A `Fail` value is logically "boolean false" and you can easily check if a value `type` is "Fail" like this:
`(! foo)` -OR- `IsFail(foo)` -OR- `foo.IsType(Fail)`.

Emit

The `Emit` type is used to identify "event emitters" & "event listenners".

Kind

The "kinds" are dependent on the "types" and they extend & compliment the types in extraordinary ways. These easily cut a lot of hours - when you take debugging in consideration of how many times you have to create & modify functions & expressions to identify some piece of data for you.

```
NoneVoid    ::    undefined

PosiSpin    ::    +           (boolean true)
IffySpin    ::    ?           (logically null)
NegaSpin    ::    -           (boolean false)

BareNume    ::    0           (zero)
IntgNume    ::    1           (integers)
FracNume    ::    0.1         (has a fraction)
UnitNume    ::    #FFF        (measured)

BareData    ::    ""
VbitData    ::    "True"      "Bare"      "Fals"
NumeData    ::    "123"       "0.4"       "#2.5.hz"
TextData    ::    "abc"       "foo bar"   some text
ListData    ::    "a,b,c"     "[1,2,3]"
NodeData    ::    "foo:1"     "{a:b}"     "(){}"      "foo(){}"
HardData    ::    (binary)
BiosData    ::    "/pth/dir"  "http://example.com"

BareList    ::    []          (empty array)
FlatList    ::    [a,b,c]     (no items are lists or nodes)
TreeList    ::    [a,[bleh]]  (some items are lists or nodes)

BareNode    ::    {}          (empty node)
FlatNode    ::    {a:1,b:2}   (no items are lists or nodes)
TreeNode    ::    {a:[c]}     (some items are lists or nodes)
FuncNode    ::    ()[]        {}()[]      foo{}()[]

TimeTick    ::    Time.Tick(every:#1000ms){}
BiosTick    ::    Bios.sock('ws:localhost:9000').Tick(){}

PathBios    ::    Bios.path('/some/dir/or/file').Tick(){}      (watch dir -or file)
SockBios    ::    Bios.sock('ws:localhost:9000').Tick(){}      (socket transmission)
UserBios    ::    Bios.user('wb3d:localhost:9000').Tick(){}    (GUI)
ProcBios    ::    Bios.proc('bash command').Tick(){}           (child process)
```

Here are a couple of examples for clarity:

```
numb: 123
frag: 123.0
loaf: #0.7.kg

tnum: '123'
ptxt: 'abc'
path: '/some/place'

elst: []
nlst: [1,2,3]
tlst:
[
   ['a']
]

---------------------------------------

numb.Type                ::     <Nume>
tnum.Type                ::     <Data>
elst.Type                ::     <List>

---------------------------------------

numb.Kind                ::     <IntgNume>
frag.Kind                ::     <FracNume>
loaf.Kind                ::     <RateNume>

tnum.Kind                ::     <NumeData>
ptxt.Kind                ::     <TextData>
path.Kind                ::     <BiosData>

elst.Kind                ::     <BareList>
nlst.Kind                ::     <FlatList>
tlst.Kind                ::     <treeList>

---------------------------------------

(numb.Type = Nume)       ::     True
(tnum.Type = Data)       ::     True
(elst.Type = List)       ::     True

(numb.Kind = IntgNume)   ::     True
(tnum.Kind = NumeData)   ::     True
(elst.Kind = BareList)   ::     True

---------------------------------------

numb.IsType(Nume)        ::     True
tnum.IsType(Data)        ::     True
elst.IsType(List)        ::     True

numb.IsKind(IntgNume)    ::     True
tnum.IsKind(NumeData)    ::     True
elst.IsKind(BareList)    ::     True

---------------------------------------
```

Expressions

Graceful handling

You may have seen some strange uses of expressions in the docs so far, so this doc will clear up things quite a bit.
VAMP's expressions are geared towards "graceful handling" - which means it will not generate any failure unless the expression is malformed. This means your variables can have any data-type and your expression will hapily calculate it, either explicitly, or impliclitly; meaning that it will take care or things like: "devide by zero", or "negative zero", or "to the power of zero", etc, etc.

Expression Governors

Every expression in VAMP is preceded either by an "explicit" or "implicit" `governor`.
VAMP expressions have strict rules; however, an "expression governor" dictates how "the following expression" will operate.

For instance:

- "method-calls" are governed by the "caller" governor, which tests the validity of arguments, and handles synchronous returns & asynchronous call-backs
- "data-lookups" are governed by the "lookup" governor, which injects a data-source into an expression and guides the expression to skip "meta-aspects", create a result list and insert into it pointers to what the expression found
- "truth-tests" are governed by the "tester" governor, which injects data to be tested and runs the expression arguments recursively - checking the data accordingly

The above are just a few examples, but you get the idea. You can also define your own "expression governors", but we'll get to that later.

Arithmetic expressions

I'm sure you're keen to try out the raw power that VAMP offers with its data-type arithmetics.
The examples below are not complete lists of all possible cases, but merely indications of how things work in general.
Data-types other than numbers are sensibly handled as numerics and results are calculated implicitly.

Void & Zero

Void is numerically treated as `0` (absolute zero), else it is treated as "nothing".
Zero is numerically treated as `0` (absolute zero); so "negative zero" is `0`, and "positive zero" is `0`; the same applies to Void in terms of "polarity".

```
()                  ::
().Kind             ::    Void

v:()                ::    "v" is "nothing" -or 0

-----------------------------------------------------------

(v + v)             ::
(v - v)             ::
(v * v)             ::
(v / v)             ::
(v % v)             ::
(v ^ v)             ::

(v + 0)             ::    0
(v - 0)             ::    0
(v * 0)             ::    0
(v / 0)             ::    0
(v % 0)             ::    0
(v ^ 0)             ::    0

(0 + 0)             ::    0
(0 - 0)             ::    0
(0 * 0)             ::    0
(0 / 0)             ::    0
(0 % 0)             ::    0
(0 ^ 0)             ::    0

(v + 3)             ::    3
(v - 3)             ::    -3
(v * 3)             ::    0
(v / 3)             ::    0
(v % 3)             ::    0
(v ^ 3)             ::    0

(3 + v)             ::    3
(3 - v)             ::    3
(3 * v)             ::    0
(3 / v)             ::    3
(3 % v)             ::    3
(3 ^ v)             ::    0

(0 + 3)             ::    3
(0 - 3)             ::    -3
(0 * 3)             ::    0
(0 / 3)             ::    0
(0 % 3)             ::    0
(0 ^ 3)             ::    0
```

Spin

Spin is numerically treated as 1 of 3 values:

- `-` -1
- `?` 0
- `+` 1

The numbers to these values are "numeric barriers" of the `Spin`, from: `-1` to: `1`.
Any number less than `0` is `-`
Any number exactly `0` is `~`
Any number more than `0` is `+`

Before any calculation is done on the `Spin`, these numerics are processed as described above.

Here are some examples:

```
ns:-                       ::     NegaSpin
is:?                       ::     IffySpin
ps:+                       ::     PosiSpin

------------------------------------------------------------

(-True)                    ::     -
(?True)                    ::     +
(+True)                    ::     +

------------------------------------------------------------

($ns)                       ::     -
($is)                       ::     ?
($ps)                       ::     +

------------------------------------------------------------

($ps + $ns)                  ::     ?
($ps + $is)                  ::     +
($ps + $ps)                  ::     +

($ps - $ns)                  ::     +
($ps - $is)                  ::     +
($ps - $ps)                  ::     ?

($ps * $ns)                  ::     -
($ps * $is)                  ::     ?
($ps * $ps)                  ::     +

------------------------------------------------------------

($ns / 0)                   ::     -
($is / 1)                   ::     ?
($ps / 2)                   ::     +

($ns % 0)                   ::     ?
($is % 1)                   ::     ?
($ps % 2)                   ::     ?

($ns ^ 0)                   ::     ?
($is ^ 1)                   ::     ?
($ps ^ 2)                   ::     +
```

Nume

Numerals are the easiest in VAMP expressions. Here are a couple of examples:

```
(-1)                    ::    -1
(0)                     ::    0
(1)                     ::    1

(-1)                    ::    -1
(+0)                    ::    0
(+1)                    ::    1

(-(1))                  ::    -1
(?(1))                  ::    +
(+(1))                  ::    1

(-(-1))                 ::    1
(?(-1))                 ::    -
(+(-1))                 ::    1

------------------------------------------------------------

(1 + 1)                 ::    2
(2 - 2)                 ::    0
(3 * 3)                 ::    9
(4 / 4)                 ::    1
(5 % 5)                 ::    0
(6 ^ 6)                 ::    46656
```

Data

The following shows how TextData is handled:

```
txt: 'abcde'
olo: 'Hello World!'
say: `What's the deal with "all-THIS"?`

-----------------------------------------------------------

(-$txt)                 ::    edcba
(?$txt)                 ::    ?
(+$txt)                 ::    abcde

-----------------------------------------------------------

(a @ $txt)              ::    1
(e @ $txt)              ::    5
(z @ $txt)              ::    0

(1 @ $txt)              ::    a
(5 @ $txt)              ::    e
(9 @ $txt)              ::

($olo % o)              ::    Hell Wrld!

((2 <> -2) @ $txt)      ::    bcd
((b <> d) @ $txt)       ::    bcd

((2 >< -2) @ $txt)      ::    c
((b >< d) @ $txt)       ::    c

((a <> z) @ Data)       ::    abcdefghijklmnopqrstyvwxyz
((A <> Z) @ Data)       ::    ABCDEFGHIJKLMNOPQRSTYVWXYZ
((0 <> 9) @ Nume)       ::    0123456789

($say * %CASE:UC)       ::    WHAT'S THE DEAL WITH "ALL-THIS"?
($say * %CASE:LC)       ::    what's the deal with "all-this"?
($say * %CASE:CC)       ::    WhatSTheDealWithAllThis
($say * %CASE:CB)       ::    whatStheDealWithAllThis

-----------------------------------------------------------

($txt + fg)             ::    abcdefg
($txt - de)             ::    abc
($txt * $txt)           ::
($txt / c)              ::    [ab, de]
($txt % $txt)           ::    []
($txt ^ $txt)           ::

-----------------------------------------------------------

(-0 + $txt)             ::    abcde
(-1 + $txt)             ::    bcde
(-2 + $txt)             ::    cde

(-1 - $txt)             ::
(-1 - $txt).Type        ::    Void
```

Data - continued

```
($txt + 0)               ::    abcde0
($txt + 1)               ::    abcde1
($txt + 2)               ::    abcde2

($txt - 0)               ::    abcde
($txt - 1)               ::    abcd
($txt - 2)               ::    abc

($txt * 0)               ::
($txt * 1)               ::    abcde
($txt * 2)               ::    [abcde,abcde]

($txt / 0)               ::    [abcde]
($txt / 1)               ::    [a,b,c,d,e]
($txt / 2)               ::    [ab,cd,e]

($txt % 0)               ::    [abcde]
($txt % 1)               ::    []
($txt % 2)               ::    [e]

($txt ^ 0)               ::
($txt ^ 1)               ::    abcde
($txt ^ 2)               ::    abcde
```

Time

When you call `Time()`, a Unix timestamp is returned with micro-seconds as a `FracNume`.
If you want a modified string from the Time-call instead, you can also call it with an option-string like:`%YMD`.
You can also get a formatted time-string from micro-timestamp with the `%TIME` modifier in expressions.
When you have your time-string you can modify it as you wish.

Thius also works the other way around: when you use a "correctly formatted time-string", it will produce a time-stamp.
When referring to "correctly formatted time-string", it simply means to use it as indicated below:

```
now: Time()
txt: 'Mar 8 2016 2:40 am'

-----------------------------------------------------------

$now                                ::    1457404805.999

($now * %TIME:Y)                    ::    2016
($now * %TIME:12h)                  ::    am

('2016-03-08 02:40:05' * %TIME)     ::    1457404805
(txt * %TIME)                       ::    1457404800

Time(%Y)                            ::    2016
Time(%YM)                           ::    2016-03
Time(%YMD)                          ::    2016-03-08
Time(%YMD:h)                        ::    2016-03-08 02
Time(%YMD:hm)                       ::    2016-03-08 02:40
Time(%YMD:hms)                      ::    2016-03-08 02:40:05
Time(%YMD:hmsn)                     ::    2016-03-08 02:40:05:999
Time(%YMD:hmsn:24h)                 ::    2016-03-08 02:40:05:999
Time(%YMD:hmsn:12h)                 ::    2016-03-08 2:40:05:999 am
Time(%YMD:hmsn:12h:dlz)             ::    2016-3-8 2:40:5:999 am

Time(%hmsn)                         ::    02:40:05:999
Time(%hms:wlz)                      ::    02:40:05        :: with leading zeros
Time(%hms:dlz)                      ::    2:40:5          :: drop leading zeros
Time(%hms:12h)                      ::    2:40:05 am

Time(txt)                           ::    1457404800
```

Lookup expressions

Lookup expressions are like "queries"; even so, they have ways to optimize searches.

```
ride: {color:blue, roof:none}

------------------------------------------------------------

$ride(*e)                   ::    ['blue', 'none']

$ride(*:*e)                 ::    ['blue', 'none']

$ride(*:*e 2)               ::    ['blue', 'none']
$ride(*:*e 1)               ::    ['blue']

$ride(<< *:*e 1)            ::    ['blue']
$ride(>> *:*e 1)            ::    ['none']

$ride(1 *:*e 1)             ::    ['blue']
$ride(3 *:*e 1)             ::    ['none']
```