



{VA:MP}

versatile application meta-programming



VAMP

Versatile Application Meta-Programming

Synopsis

In order to really "get" what Vamp is, you need to read the text below. I mean, really, read it. A nice "movie" or "slide presentation" would be great, but this where it starts.

Vamp is...

"a powerful software platform".

Well that doesn't say much; people generally have their own ideas of what a "platform" is.. like.. "it's huge -like an OS" .. or "it's complicated", or "it is some desktop app with lots of complicated stuff"... ?



Well... No, Vamp is NOT that at all.. let's try again:
"Vamp is a structured programming language."

True, but again, what does that mean?

There are so many ways of describing "programming languages" and I'm sure Vamp can in some ways fit into some of those comma-separated lists of descriptions; however, Vamp is not only a programming language.

(KEEP READING)

Then.. -what the hell is it?

As its acronym suggests, it is what it is, here's why:

Vamp is an extensible...

- programming language (interpreted or compiled)
- MIME-type authoring syntax -and transpiler
- BIOS for information transmission
- active database

Sure but... these software concepts are separate for a reason yes?

True, however, in order to use all of those in a complete software solution, you often need to learn at least 5 completely different languages, not even mentioning compatibilities, deprecations, standards compliance, etc.

So okay, what if I already have all of the above, and it's working, why choose Vamp?

Well...

- Vamp can interface with almost anything software (or hardware) related, like other data-bases.
- It can be used to produce other existing languages, like: C, Python, JavaScript, PHP, HTML, CSS, etc.
- It utilizes the concepts of the existing MIME you're authoring, like "tag-names & properties".
- It has a clean and elegant syntax (grammar) that is easy to learn and requires a lot less code to produce something feasible.
- It is extensible, so you can define anything you want, hook it up with some key-words and define some rules of how to use your "extension" in the language, and that then will become part of the language as you use it.
- It's "http sock bios" (front-end) comes by default with a well defined "mobile-friendly front-end grid-system" and attractive (rich) visual theme (including font-icons), much like "Twitter Bootstrap" - only a lot more simplified, and you can override (configure, tune, extend) it per project (or per bios) as you wish.
- You can use your existing scripts (or frameworks) to extend (or replace) Vamp's defaults; however -this may be a bit technical and is not in the scope of this document.

With all that in mind it's easy to see that you can use this to define almost anything: from web-sites and e-commerce web applications through to Apache-modules & PHP extensions, 3D scenes, raster graphics, -you name it, -if a "transpiler" is not already available in the Vamp repository, you can build your own - and even sell your work to others that may need it -(or donate it free of charge, it's up to you)

Still though.. "Why Vamp?"

--because everyone has problems structuring their projects, especially large ones that may span across several servers, networks -or domains, and, businesses usually have to spend thousands (and I'm talking in the hundreds) to get such a solution well structured and working, either by purchasing such a solution in a "framework" or having it developed, both is expensive, (especially "n-Tier" solutions).

..okay but what does this have to do with Vamp?

Vamp is straight-forward and simple. From the ground up, everything you do in Vamp has to do with "structure" and "context".

Every Vamp project needs its own folder -which is your "project root".

The way you do things in Vamp has to do with the physical folder and file structure in this "project root".

This structure becomes your entire "live runtime environment" -but items only get loaded as referred to in your Vamp code. The items in this structure have rules on how they relate, and, they maintain their "context" (scope) religiously. With this in mind, you can see how Vamp allows you to define what-ever structure you want, but it lets you stick to the structure you define, so everything has a place.

(KEEP READING)

Popular files and data is served directly from RAM, so this is incredibly fast, like a database, and your structure (and data) lives in the RAM (not on harddrive) where it gets updated, so it's dynamic; however, the live-structure does get "saved-to-disk" when it becomes "unpopular", or per "Save()" instruction in your code.

Internally the language utilizes "memory pointers" to refer to structures and data, so it does not consume too much resources at a time by passing around whole chunks of data.
The C language works this way also, hence it is incredibly fast; however, Vamp does not require you to be bothered with these "pointers" as it gets handled automatically.

Resources external to your "project root", like in a different path, (or device) is represented by a special kind of "sym-link" inside your project structure. An external resource can also be on a different server elsewhere; however, "Save()" instructions will save the data at the external resource, not locally.

*The "Vamp Core" is semi-open-source and free for personal use, -and it comes by default with MIME extensions (transpilers) for: **HTML**, **CSS**, & **JavaScript**.*

If you want to use Vamp for your business, there are flexible payment solutions available that suit your business needs.

..hang on a second, I have to pay for this?

No you don't; however, if you intend to make monetary profit by using it, then: yes. -It's not expensive at all and has no recurring license fees. When you purchase Vamp: upgrades are free, you also get live technical support and training videos, though, not much training is required; it is very easy to learn and quick to master.

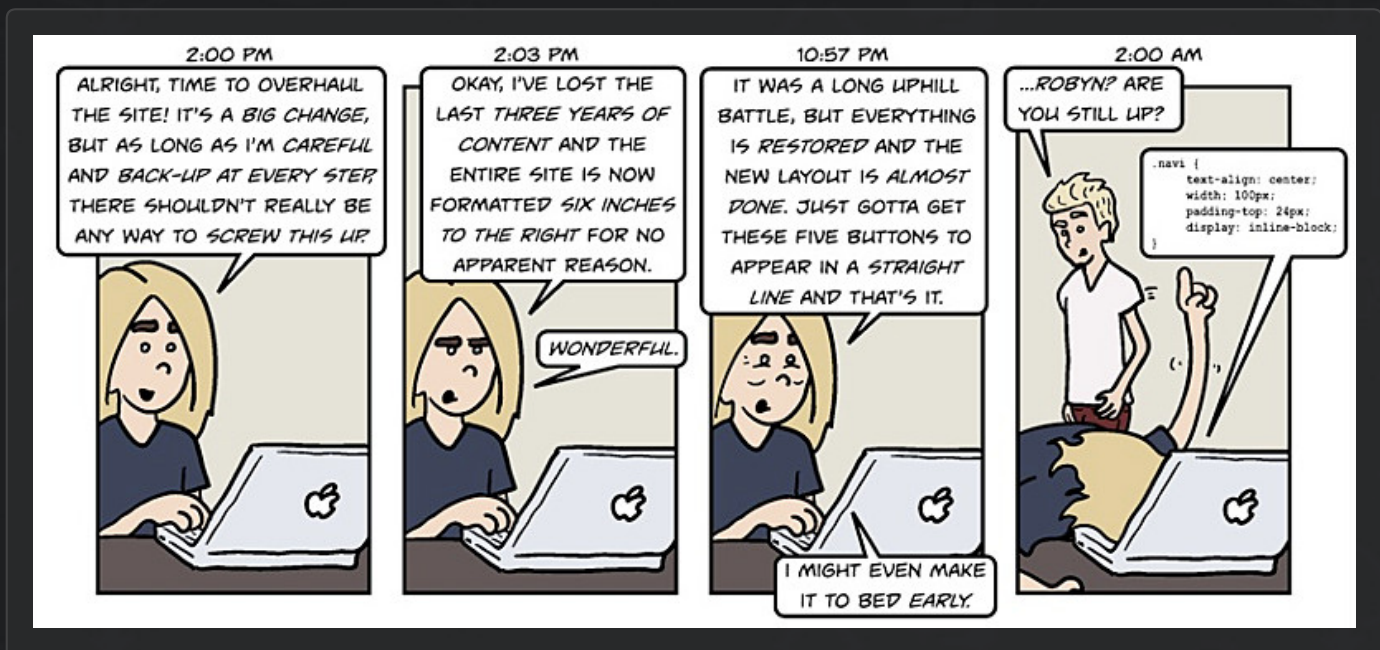
The Vamp initiative has a goal: to make enough profit to sustain the developers until Vamp is introduced into the labor markets, after that **Vamp will become completely open-source AND absolutely free of charge.**

Anyway...

By now you can probably see why Vamp is not just a language, or data-base, etc, so, it's not easy to say in 1 sentence exactly what it is... It is many things.. it is 1 thing.. it is Vamp.

ALL the above was enough to confuse anyone, but, fear not, the rest of this document will make things (a bit more) clear.

Vamp can save you a lot of time & money, but you won't know that unless you read more, so:
READ the rest - (not skip)



Introduction

Vamp runs as a daemon (background service API) on the host it is installed -which by default is configured to stay up and running even if the host is rebooted. This daemon manages & monitors other Vamp instances, which is accessible via the command-line. Vamp instances depend on the Vamp daemon, so if the daemon dies, the instances will (gracefully) commit suicide.



A "Vamp Project" (vamp instance) is simply a folder with some Vamp file(s) inside it. Vamp file-extensions can be any of these: `.v` `.va` `.vam` `.vamp`.

When you create a new Vamp project, or simply want to test some Vamp code, make sure you do this in a new (empty) folder, or a folder that contains items related to your project, because Vamp expects these as part of your project's live runtime structure; however, only the items referred to in your Vamp code will be loaded accordingly.

Any "item" -like a file or folder (simply: "node") in your project structure have "aspects" (attributes), some of which are automatic -which (also) deals with relationships between items in your project. These "automatic-attributes" (meta-aspects) also deal with how an item reacts when "called" -as anything in Vamp can be called (like a function). "Calling" operates in different ways, depending on how you define and call an item.

Any "node" can have "contents" (even functions have contents -which is a list of instructions). The contents (and aspects) of any "node" is directly related to that "node", and its context remains within this "node". This means that your "runtime scope" is well preserved everywhere, which is also your runtime structure. Remember this, it is very important.

Vamp code closely correlates to "folders & files" as a structure; meaning that any (live) Vamp code can be "saved to disk" as a "tree structure" -consisting of folders & files that represent the functional structure and data of your live project. This makes any live Vamp project dynamically portable, -or can be used for recovery purposes.

(KEEP READING)

Vamp has an intrinsic (built in) transmission control system, called "bios" (basic input output system) -which is extensible. It's IO functions have short-cut names like: **Sock** (socket), **Proc** (process), etc, -each of which have protocols -according to which default "flow" mechanisms control its input and output respectively. The Vamp "bios" and all its control systems and their protocols are by default configured and optimized for rapid development; however, they can be configured and overridden per instance during runtime.



Some of these "bios" controls automatically "transpile" Vamp code into the appropriate language(s) (code) for each "tick" (request/event) respectively; however, this automation can be configured, and, any structure (like a document) can be "baked" (transpiled) to be "served" directly; which is resource conservative. It is important to structure your Vamp projects and name your Vamp files appropriately -with compound file extensions, like: `home.html.v` -which makes it easy to find and identify -for yourself and Vamp.

For security reasons, Vamp files cannot be "served" directly via any Vamp "bios". If you want to do so, it has to be explicitly transpiled to "plain text" first.

A "bios" can also be used to just transpile Vamp code into other server-side (-or client side) languages, like: Python, Ruby, PHP, JavaScript, etc.

Your Vamp project operates as a "programmable live database" -which follows your project structure, making it simple to write versatile applications that are well structured, dynamic, stable and portable. It is up to the developer to structure and scale large projects appropriately; either on the same host, or across several servers and networks with the proper security measures in place, which Vamp also supports.

All of the above (and more) is done in a simple & elegant syntax (grammar) that looks like a combination of: JavaScript, C, CSS, PHP, & Lisp.

When a Vamp file is named with a compound file extension according to the file-type (typically structured document types like HTML or CSS) it should produce, you can use the conventions (tags, selectors, properties, etc) expected for that type as you would use that language itself; however, coding that in Vamp gives you several advantages, because, not only does Vamp come with very useful "nodes" (and themes) of its own, --you can also make your own "tags" (nodes, themes, conventions, etc) which translates to the conventional output after "automatic transpilation" through the BIOS -as you've defined it to be. This becomes exceedingly powerful and useful through your project(s) as you understand that now you can make up a "form" that is directly related to your data-structure; so validation, theming, labeling, events, etc all becomes part of your structure.

There is so much more to elaborate on, but this intro is getting too long, so I'll wrap it up with:

Runtime stability

The Vamp interpreter operates much like how a web-browser renders a document. You can imagine what chaos it would be if the browser would suddenly become unstable and throw errors at the visitor if you placed a "tag" in the wrong place, or if the name of some style is wrong; it would be a disaster.

When it comes to programming though, and something is wrong with the code, a developer needs to see exactly "what is wrong". Vamp brings this together by making error messages short & useful, and at the same time, only "the wrong part" fails, not the entire runtime; the rest will calculate & render quite normally, even if you have a "syntax error". This is because Vamp code is parsed by "context", within "context operators" like: `()`, `{}`, `[]`.

Your "project root" has a runtime config meta-aspect called "Mode", and if this is "live" then if something does fail, the error message will only show a "fail-type" -with no description at all, like: "MathFail", or "CallFail", etc, even so, the message will visually be in the place where it failed, or where a "fail" item is displayed (without knowing it's a fail).

With all the above in mind, you can see how this could save you countless hours in development time, saving volumes of manual code, prevent code duplication, minimize and handle human error gracefully. You can imagine that this is favorable when it comes to tight deadlines and demos :)

Examples

These examples below are not "tutorials", but, fear not, a detailed guide for installation and a "Hello World!" tutorial is in the "orientation"; these are just so you can get a feel of what to expect, and for a firm grasp of what Vamp is about.

Web Server

Remember: in Vamp you can override & configure anything; either during runtime, or in your installed Vamp setup, so from here on just accept the fact that a lot of hard work has been done for you so you can just "tune" & control things as you wish.

With the above in mind: Vamp's "http sock bios" by default produces appropriate "HTTP message codes" and some required headers, like "Content-Type" (mime-type) and "Content-Length" (byte-size).

The context (or folder path) where any BIOS is instantiated becomes the "root path" of any "path request" on that BIOS instance.

This is for logical structure and security reasons. Everything in Vamp works this way.

If you don't like this, then just put it somewhere else -lower down the "tree", like in your "project root", this way it will have "read" access to everything in that context by relation to the other items in there "akin" to itself. "Write access" to the items "akin" to itself can be achieved by defining it as such in the meta-aspects of the other items.

See this security measure as "parents" who can "teach" their own children, but others need permission for this privilege.

By now you can see how Vamp is geared towards structure, simplicity and security.

In the examples that follow, for clarity and brevity, I'll assume you created a file inside your "project root" named: `$.v`, and this is the file in which we will place the code in the examples below. An explanation of "\$" files is in the "orientation", but for now: "\$" simply means: "self", and when a file-name starts with "\$" -it is loaded first, and any text inbetween the "\$" and the ".v" will be an aspect-name of the context in which it resides.

Automatic Web Server

```
Sock(#HTTP:127.0.0.1:9000)
```

The exact code above will actually produce working web-server.
I can only imagine the expression on your face... read on, you'll see :)

In the example above, we created an "event listener" that handles "http" requests made to the IP address: "127.0.0.1" on port: "80". It will perform "expressions" on any path requested, based on the "method" of the request and the "arguments" (http request variables) passed along with the request.

If there is a problem with the request, like if the path does not exist, or this bios instance does not have permission to read (or write, based on the request path & method) it will produce an appropriate http response code (like 404, 403, etc) along with a good-looking error message.

This explanation may get a little long and technical, but for now, just know that: by default it handles requests & responses automatically and appropriately as it is geared towards simplicity and resource conservation. More on this is available in the "bios" section of this documentation.

Picture Web Server

```
Sock(#HTTP:127.0.0.1:9000)
{
  Mime: ['jpg', 'png']
}
```

The example above will only handle requests made to files with "jpg" & "png" extensions.
Note that if you used both the "Automatoc Web Server" example together with this one, then this one will override these 2 mime-types and this example will handle it, instead of the other, because they share the same "socket".

Path Web Server

```
Sock(#HTTP:127.0.0.1:9000)
{
  Path: ['/docs/']
  Bias:
  {
    Deny:
    {
      Addr:
      [
        '192.*'
        '47.*':
        {
          Path: '/docs/gossip/'
          Mime: ['pdf']
        }
      ]
      Path: ['/docs/staff/']
    }
  }
}
```


The example above will only handle requests made to the folder "docs" inside the context where this bios instance resides. This will override any other "general" http handlers related to "Path" on the same socket.

This example (above) also implements specific security overrides on the "127.0.0.1:9000" socket over "http" protocol: It denies all access from any ip address starting with '192.'

It denies everything in the "/docs/gossip/" folder -and "PDF" files from any ip address starting with "47."

It denies any access to "/docs/staff/" for everyone.

Of coarse you can make a separate file, like: `rules.cfg.v` and just assign: `Bias: rules.cfg.v`, or however you want; it would be preferable, especially with a long list of such rules that may be needed for other BIOS instances as well.

Automatic Web-socket Server

```
Sock(#WS:127.0.0.1:9100)
```

In the example above, the same rules apply, with similar configuration.

Raw socket handler

```
Sock(#TCP:127.0.0.1:9200)
[
  $Echo($Data + ` -back to you!`)
]
```

The example above shows how to override the "event" (Tick) handler on a socket. In Vamp, "Tick" is a way to say "next" -or "then".

Note that this example (above) does not specify a sub-protocol, so this means you will have to handle the raw input and output data yourself.

Automatic CLI REPL

```
Proc(#TTY:Root)
```

The "proc bios" is a mechanism that hooks into (takes over) the "stdin", "stdout" & "stderr" of a process and it evaluates (parse & execute) any Vamp code that runs through it during runtime.

In this example: "#TTY:Root" refers to the main (Root) process-stream to be hooked into the current "TTY" (terminal). Each Vamp instance has its own "Root" process, so this can't be exploited to "hi-jack" other Vamp instances as this mechanism can only operate on processes that belong in the same project, on the same host.

Child process :: Mule

Child processes in VAMP are called "mules" (workers); however, they are NOT "forks" of the parent process and each mule occupies 1 processing thread. As their title imply, they can't (-or rather shouldn't) have "children".

Mules are typically used to either share work-load, or to do specific tasks (once-off -or persistent), -away from the main process. The main (project root) process cannot execute system commands; however, mules can, because their priviledges can be limited to only certain resources, or certain commands.

Mules can only be "spawned" locally; they can only be commanded by the parent; they only report to their parent, and their parent manages communication between them in order to minimize any confusion -or disputes (-race conditions). Mules also report progress of tasks managed by them.

Mules can be used to setup servers -remotely via SSH -or which ever command interface you prefer --from operating system installation & configuration -to Vamp setup. The resources of these remote servers can become part of your main project root structure -as most of Vamp's processes are asynchronous anyway. Remote resources are managed via encrypted communication channels.

When a mule is done with its work -or if it fails, it dies. When a mule dies, its parent is notified via its messaging channel, which will either be a detailed "Done" -or "Fail" message.

Mules can "fail" under various conditions: if it's commands encounter a fatal error, or when its "time-limit" is exceeded, or if it "fatigues" the host system by occupying too much CPU for too long, or too much RAM (for too long), or too much disk space, too much band-width, etc. Each of these limits have defaults set in the main Vamp installation config, which can be overridden upon initialization of any mule -or can be set in the project root config of each project; however, the maximum limits set in the Vamp installation config cannot be overridden during runtime, they will just default to the limits if exceeded. These maximum limits are percentages and "fatigue time limits" which by default is set to allow for maximum (sane) usage; alas, these protect your runtime environment against resource abuse such as "fork bombs" & "DDos attacks".

```
Mule(exec:`echo "blah"`)
[
  $Done(0)
]
```

Conclusion

This "synopsis" was both too long and way too short. There is so much more to Vamp, but this you will have to discover in the rest of this documentation; however, I hope it helped with understanding more about Vamp, even though it was a lot of text to read; thanks for not giving up.

There is no doubt that Vamp is versatile and powerful. A little code goes a long way. Your structure is your empire that could span securely across multiple networks across different continents. With this kind of flexibility, scalability, stability, speed of development and mass deployment; the things you can do with Vamp is nothing short of astounding.

Next up is the "orientation", so grab a drink & some crisps; you're about to unveil a geek-adventure filled with might & magic!

VAMP - Orientation

Versatile Application Meta-Programming

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 comparison 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 arithmetic 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 extended 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 course.

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.



You can also compile your VAMP project into a stand-alone executable; 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.

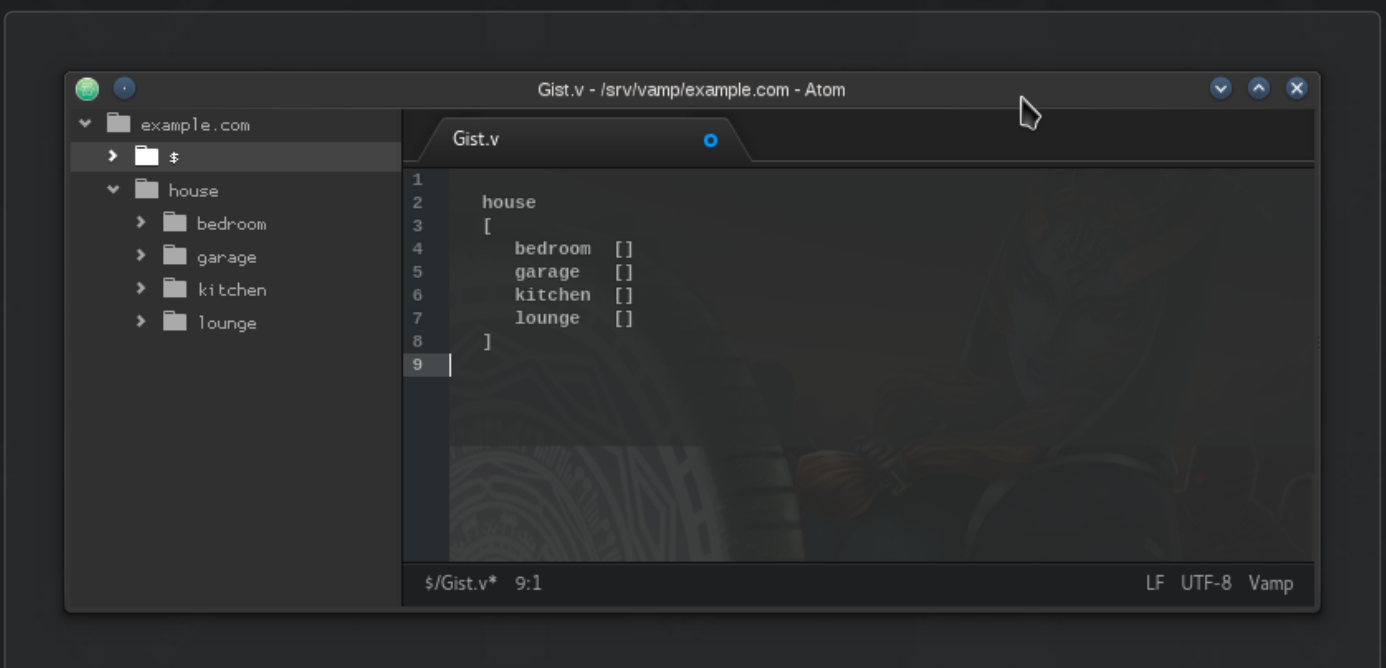
By "anywhere" I really mean it: server back-end, data-base, web interface, native desktop GUI apps; -even interactive 3D web applications -or installable 3D games; videos with synchronized audio & sub-titles; still images with super-imposed overlays, etc, etc..

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, otherwise, 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:



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 compatibility 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.



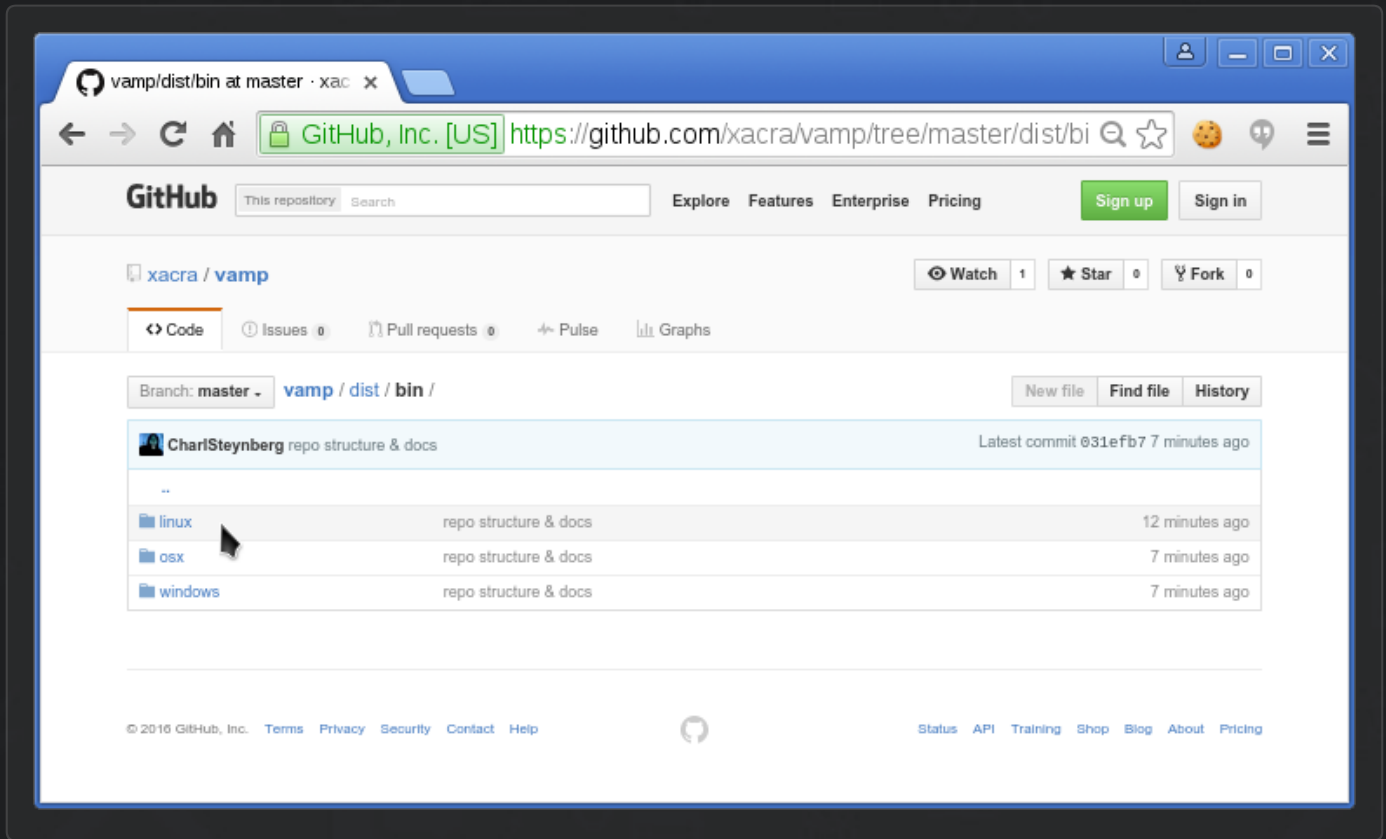
The VAMP Bios (basic input - output system) adapts its input and output according to the transmission used; however, the developer does not need to apply a different way of sending messages, or reading input to and from any of these. This can be configured -per Bios; you have the freedom to customize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



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 successfully 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.



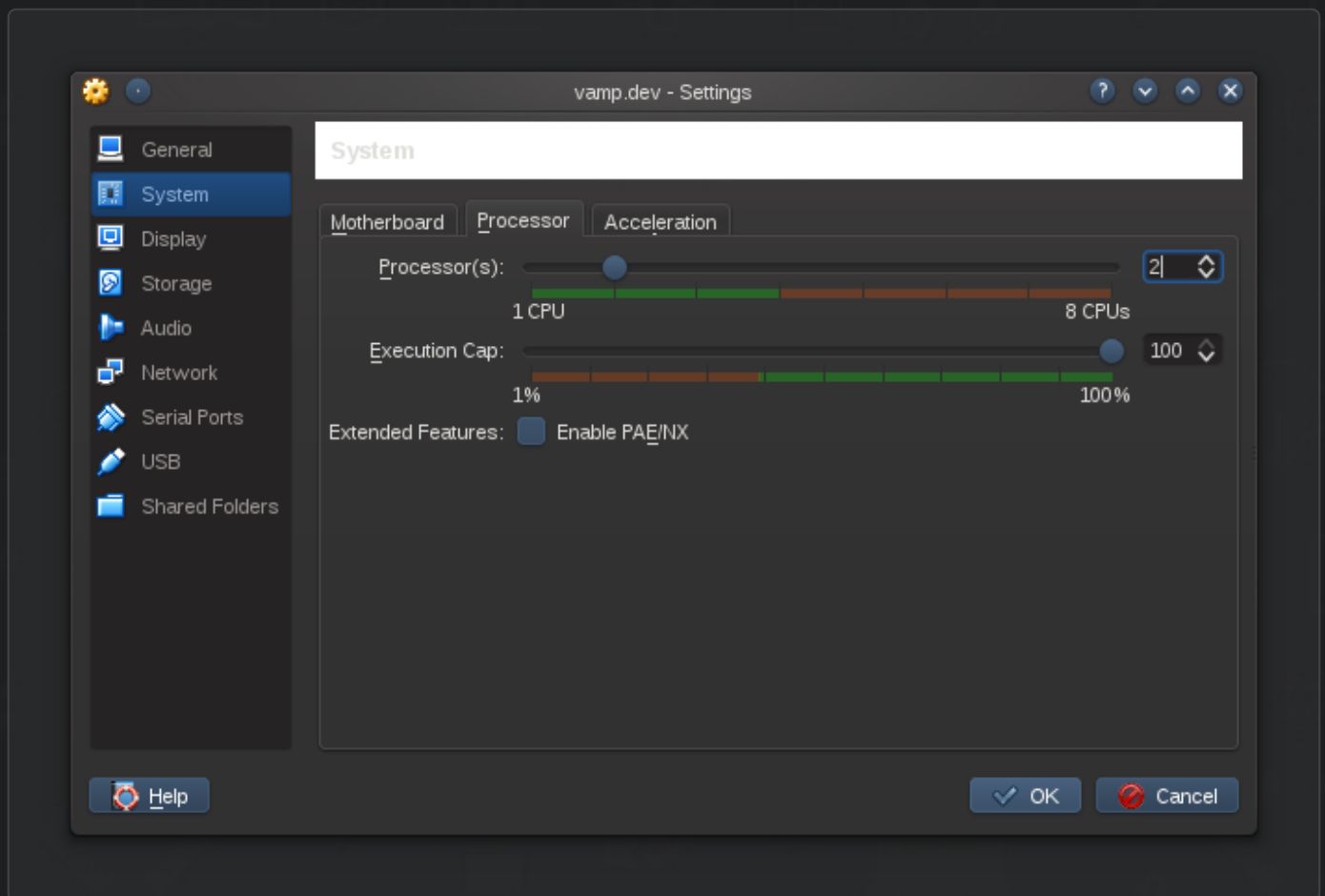
Virtual machine

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.



above :: VirtualBox

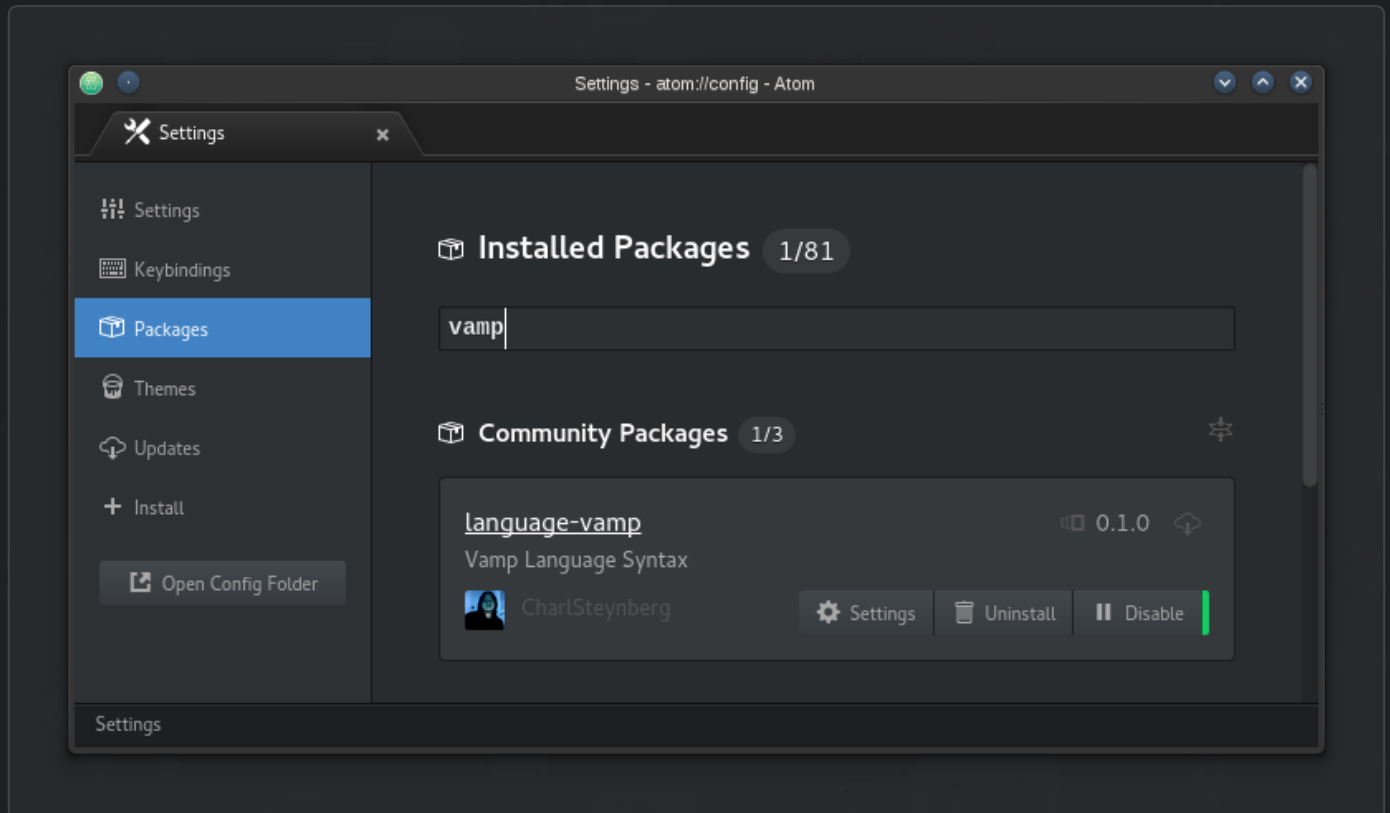
Syntax highlighting

It is important that your text editor supports syntax highlighting of some kind as it makes it easier to identify what you'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/dev/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 exercise 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. `$` (meta) files are optional, but mandatory in `project root` and refer to the folder they are in
4. `$` files are the only files that get loaded automatically, and this happens in alphabetical order
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 mind, it's time to kick some booty!

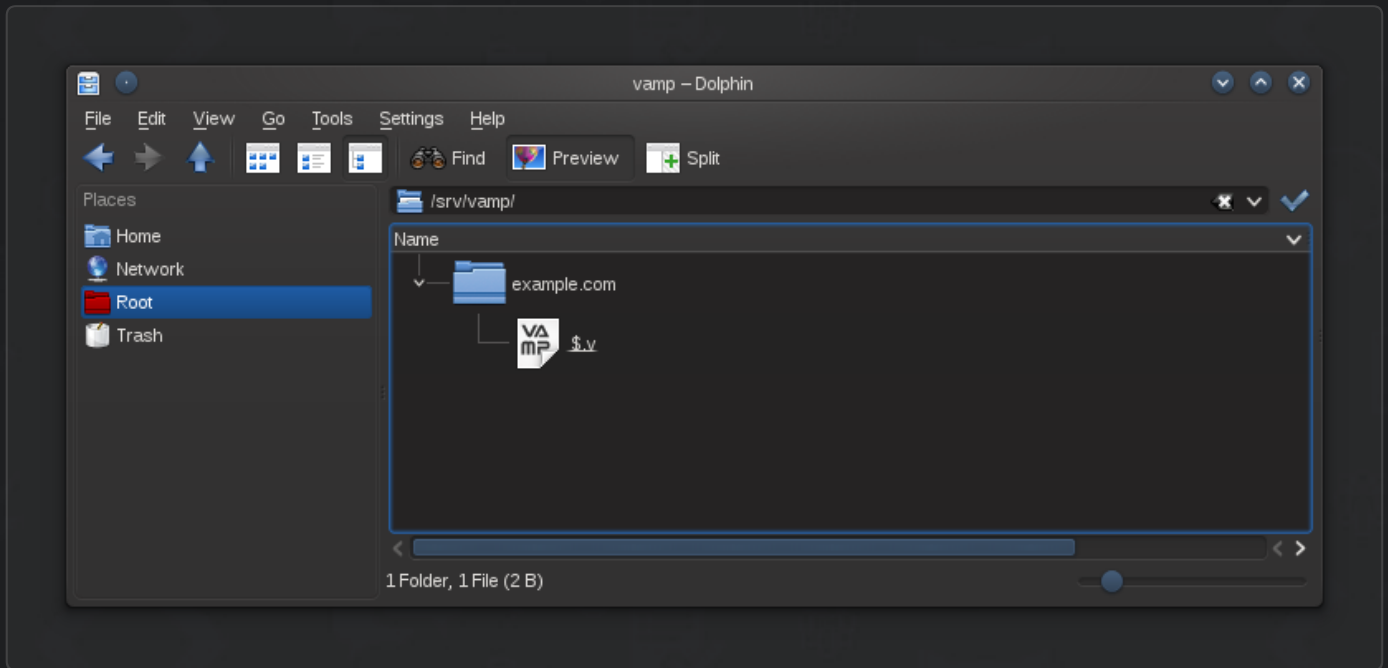


SH!T JUST GOT REAL

Let's get started:

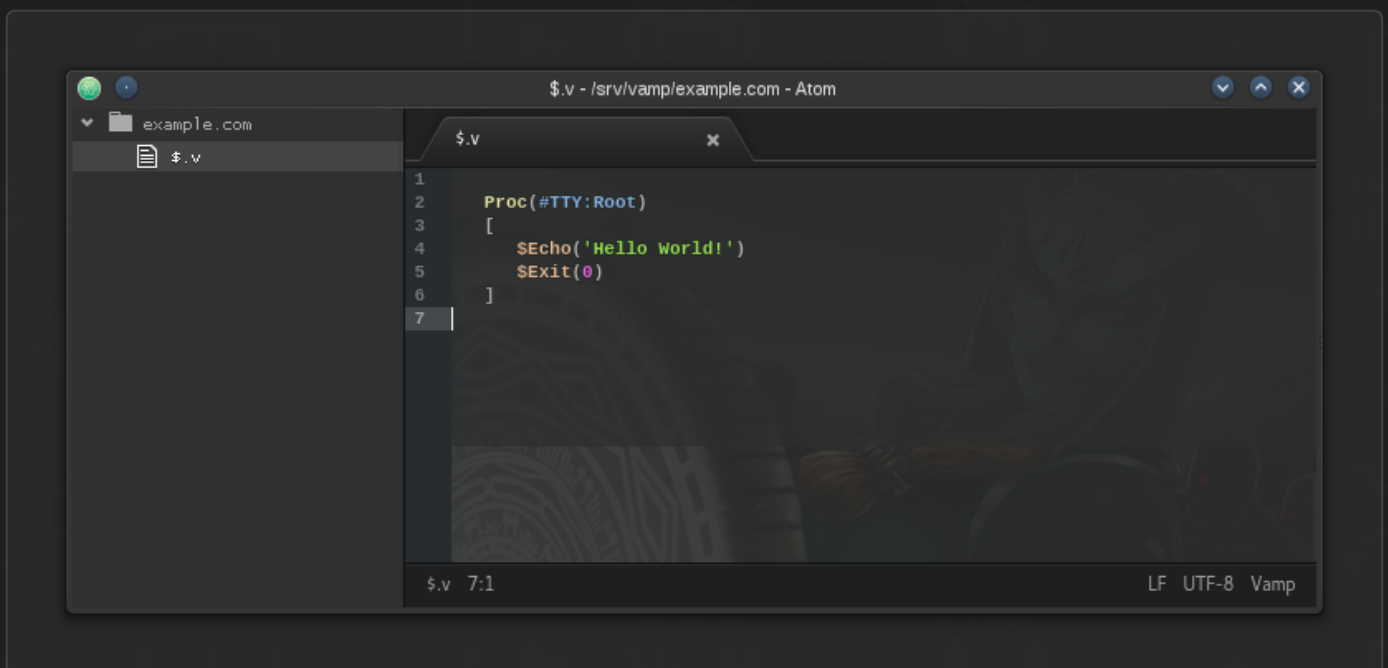
- Within the target file-system, navigate to your main projects (plural) container (folder).
- Create a new folder in there, so the path to it will be something like: `/srv/vamp/example.com`
- Inside this `example.com` folder, create a text file named: `$.v` (take note of the file extension)

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 inside it, type the code you see in the image below, and **save**.

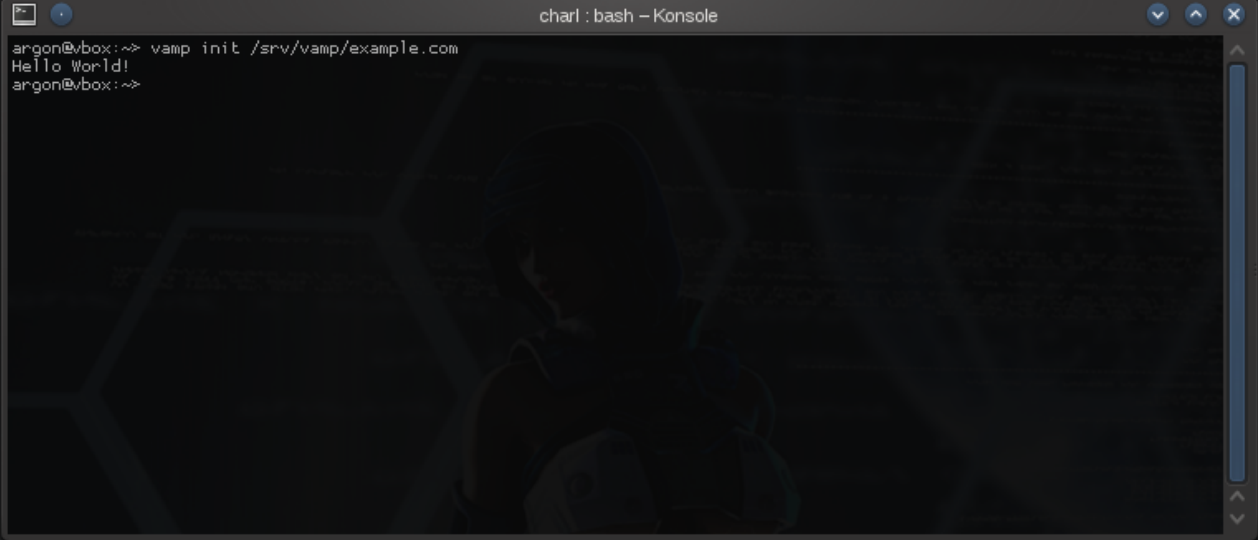


above :: project: "example.com"

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:



A screenshot of a terminal window titled "charl : bash - Konsole". The terminal shows the command `vamp init /srv/vamp/example.com` being executed, followed by the output "Hello World!". The prompt `argon@vbox:~>` is visible before and after the command. The terminal has a dark background with a faint, stylized graphic of a person's head and shoulders.

above :: project: "example.com" output

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



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 reference purposes and will be covered in more detail later.

Comment Operators

Comment in VAMP is used both for documentation and developer reference. The purpose 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 highlighted tags
.:	seccion-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, which 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 thoroughly 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" divide "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 necessarily 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:

<code>+</code>	<code>::</code>	add
<code>-</code>	<code>::</code>	subtract
<code>*</code>	<code>::</code>	multiply
<code>/</code>	<code>::</code>	divide
<code>%</code>	<code>::</code>	modulus
<code>^</code>	<code>::</code>	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 logical -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
-

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 comparison. 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 computer to do what you want. This is especially true with the "kinds" as you will see in a bit.

These are the types:

<code>Void</code>	::	<code>()</code>	::	undefined
<code>Spin</code>	::	<code>+, ?, -</code>	::	polarity
<code>Nume</code>	::	<code>1, 0.9, #123.kg</code>	::	numbers
<code>Data</code>	::	<code>'abc', "123", `*. *`</code>	::	text or binary
<code>List</code>	::	<code>[]</code>	::	list of contents
<code>Node</code>	::	<code>{}</code>	::	functional object
<code>Time</code>	::	<code>Time(`2016-03-03`)</code>	::	date, time
<code>Fail</code>	::	<code>(!/?/*)</code>	::	holds an error

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 colleagues -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`
- `?` : `None`, numbers equal to `0`, anything else
- `-` : `False`, 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 later on.

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)`.

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      ::      ""
SpinData      ::      "True"     "None"     "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{}()[]
BiosNode      ::      Sock(#WS:127.0.0.1:9000')[]

NumeTime      ::      Time()
DateTime      ::      Time(%YMD)
EmitTime      ::      Tick(every: #1:Sec)[]
```

Some entities may be more than one kind at a time. The `.Kind` meta-aspect will only be 1 though, and this would be the "most likely" (simply: the one that comes first).

You can check for more than one Type -or Kind at a time. The `isType()` & `isKind()` functions can also accept expression calls, so instead of repeating things you can do this instead:

```
:: check ALL
-----
foo.IsKind(FuncNode & TreeNode) :: same as (foo.isKind(FuncNode) & foo.isKind(TreeNode))
-----

:: check ANY
-----
foo.IsType(Spin | Nume)          :: same as (foo.isType(Spin) | foo.isType(Nume))
-----
```

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      ::      UnitNume
```

```
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
-----
```

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 which 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 simplicity, 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 is implied"; where "explicitly" means: "do the following 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 example 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 `:` states "is" `()` denotes the arguments of function:"ride" and the `[]` denotes the "contents" (context) of function: "ride".

When calling `ride()` it will yield: `vrooom!`.

A "callee" (function or method) can return a yield (echo) explicitly, like this:

```
ride:()  
[  
  $Echo('vrooom!')  
  
  'blah'  
]
```

In the example above, "vrooom!" is its yield, and "blah" is skipped.

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

The following example will explain:

```
ride()      .:      vrooom!  
  
ride()  
[  
  $Echo      .:      vrooom!  
  $Name      .:      ride  
]
```

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

- the "call handler" is initiated with an instance of itself when a call is made to it
- if no call-back context is defined for this instance then it returns its yield (echo)
- if a call-back context is defined it sends its yield (echo) to this context when all is done

Notice in the example above that there is no `:` in front of the `()`, which means to "call" (-NOT "define").

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 available 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".
Erpressions will be covered in detail later, but the above should help getting a firm grasp on how things operate in context.

Calls & 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 thing.

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 "echo" 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 "meta-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 explicitly;
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.

Expressions

Expressions Overview

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 happily calculate it, either explicitly, or implicitly; meaning that it will take care of things like: "divide by zero", or "negative zero", or "to the power of zero", etc, etc.

The "expression operators" (`[]`) can be used to "wrap" any value; like numbers (negative or positive numbers); -or even other expressions.

Expression categories

VAMP expressions can be divided into 3 categories. Each of these 3 categories have their own "expression governor" which handle expressions differently:

- `math` used for arithmetic & logical calculations
- `call` used for functions and methods with arguments
- `find` used for data lookups

Any of these categories can be nested infinitely.

MATH expressions

Before we get started here it is important to remember that "math" in this sense is NOT only "numbers".

In VAMP, a "math" expression is identified by any logical, arithmetic, reference, -or operators.

The "math expression governor" is a mean beast. It handles things in particular ways, depending on data-types, operators used and which data-type is on which side of the operator.

It also has "data modifier references", which can be changed -or extended, but it has defaults explained below.

The best way to provide a concise reference of what it can do is to categorize things by data-types and operators used on them.

Take your time to go through the next couple of pages as it will give you a lot of insight into VAMP and what you can do with it, and how it makes your life as a developer easier. It may seem like a mountain at first, but once you get the gist of it, you won't have to remember all of it, but simply that you can do amazing things in your code, and it will most probably work accurately as you intend.

You will have to apply some savvy here, because if I have to list every possible way of how an arithmetic expression can be written, this chapter will probably span a million pages.

VAMP is concerned about what you code and what the logical results of that code should be as you intend it to be. For instance:

```
// JavaScript
// -----
(0.1 + 0.2)           // 0.30000000000000004      wtf?
((0.1 + 0.2) == 0.3) // false                    kill me now!
// -----

:: VAMP
-----
(0.1 + 0.2)           :: 0.3
((0.1 + 0.2) = 0.3)   :: True
-----
```

In the example above:

- The first block of code shows the result related to IEEE standards; more info here: https://en.wikipedia.org/wiki/IEEE_floating_point
- The second block of code shows how VAMP handles this by default.

VAMP does NOT just "throw away" the unnecessary digits; rather: it gets handled internally by looking at the expression input & output accordingly, so you don't need to bend your brain trying to figure out what the hell is wrong with your own logic -when it is perfectly fine.

In the examples below, arithmetics are handled as follows:

* (multiply)

When a numeric value is present on any side of the `*` operator, it is assumed to "duplicate" or multiply; else it is assumed to "join with" -or "merge with".

MATH :: Void & Zero

From here on, assume "zero" (or `0`) as "absolute zero".

Void & Zero is seen as "emptiness" and is numerically calculated as a number: "absolute zero" `0`.

In VAMP, "negative zero" & "positive zero" is kept "under the hood", so the final output from an expression will not show the "Spin" of zero. This is managed by the "Bios". Of coarse you can make a number like: `0.0000000000001`, or `-0.0000000000001` (as many zeros as your computer can handle) -but the point here is that it ends with a `1` and this makes it NOT "absolute zero".

You can of course use `-0` -or `+0` in your code, or your expression may end up calculating on these, but, it handles "zero" as "zero". This brings us to the following:

- if anything is added to Void, the Void will be filled with that thing
- if anything is subtracted from Void, the thing becomes its own negative
- if Void is added to anything, that thing remains unchanged
- if Void is subtracted from anything, that thing remains unchanged
- if anything is divided by zero (or Void) then "that thing" remains unchanged
- zero to the power of zero, is zero

If you are a mathematician and you are frowning heavily now, remember that you can always configure the way VAMP works internally, which is discussed in the "devl docs", -but in the "real world" these principles work very well and is able to do incredibly complex mathematics that produce accurate results with perfect precision.

```
()                ::  
(()).Type        :: Void  
  
(() + 3)         :: 3  
(() - 3)         :: -3  
  
(() * ())        ::  
(() * 0)         :: 0  
(0 * ())         :: 0  
  
(() / ())        ::  
(() / 0)         :: 0  
(0 / ())         :: 0
```

MATH :: Spin

```
..: TODO :.  
-----  
--  
    Define how "Spin" would react whith different operators and data-types  
--  
-----  
..: TODO :.
```