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, strandards 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 feasable.
- It is extensable, 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 refered to in your Vamp code. The items in this structure have rules on how they relate, and, they maintain their "context" (scope) religiosly. 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.

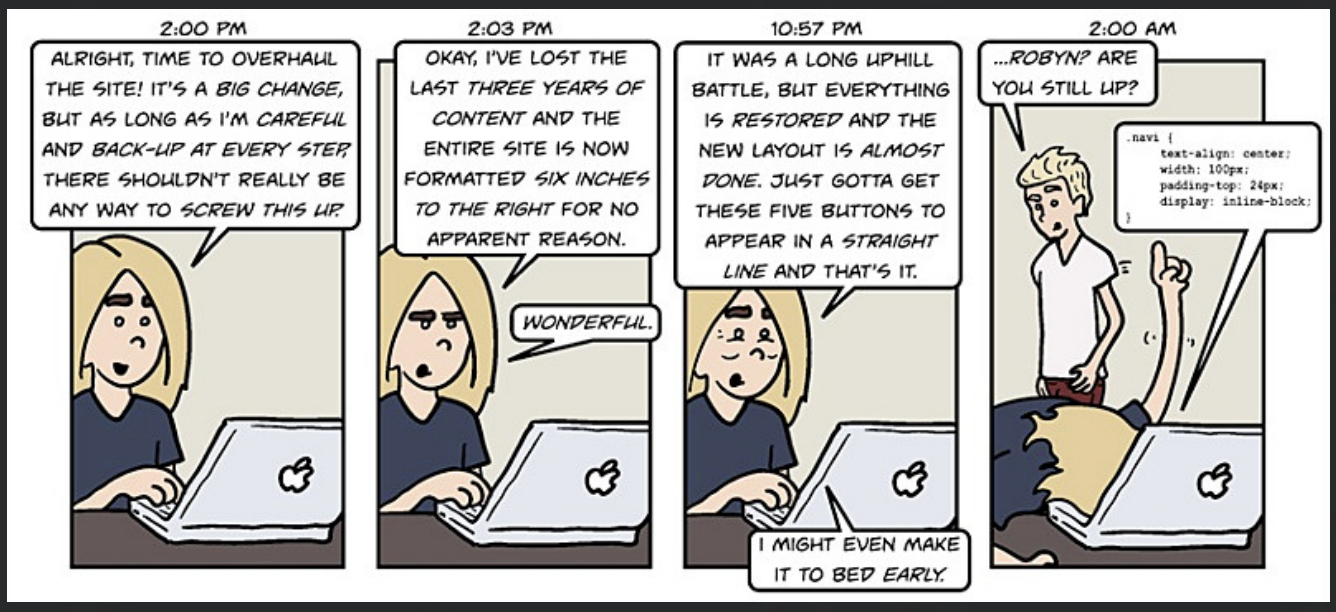
The "Vamp Core" is semi-open-source and free for personal use, -and it comes by default with MIME extensions (transpilers) for: `HTML`, `CSS`, & `JavaScript`.
If you want to use Vamp for your business, there are flexible payment solutions available that suit your business needs.

..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 iniative 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 acessable 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, of 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 logicical 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 priviledge.

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 addess: "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.

Atomatic 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 persistant), -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 runtine, 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!
