VAMP

Versatile Application Meta-Programming

Synopsis

Introduction

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

A "Vamp Project" 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.

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 accordingly. The Vamp "bios" and all its control systems and their protocols are by default configured and oprimized 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 expensions, 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 accross 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.

Examples

Many people don't like "reading" too much; however, if you skipped the "Introduction" above, read that first, else none of the following will make any sense.

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 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)
```

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.

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)
[
   $Push($Pull + ` -back to you!`)
]
```

The example above shows how to override the "event" (Tick) handler on any socket. Note that it does not specify a sub-protocol, so this means you will have to handle the raw input and output data yourself.

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 are not "fertile" -so 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 ecrypted 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 are already set in the main Vamp installation setup 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 setup config cannot be overridden at all, they will just default to the limits if exceeded.

```
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 accross different continents. Whith 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!
