Vertebra

Interim Documentation

Jayson Vantuyl

Engine Yard, Inc.

Actor	Description
WidgetTek Drawer Actor	Operates the bank's WidgetTek-brand cash drawers
SprocketInc Drawer Actor	Operates the bank's SprocketInc-brand cash drawers
Vault Actor	Operates the bank's vault control mechanism
Check Reader Actor	Operates the bank's check readers
ACH Dial-In Actor	Operates dial-in lines used by the bank's merchant services credit card machines
ATM Actor	Controls secure leased lines to various ATMs
Ledger Access Actors	Provides access to systems that handle and distribute ledger data throughout the bank
Optical Storage Actor	Provides access to optical storage used for archiving check scans and bank records
Alarm System Actor	Interfaces with security alarm
Collect-o-matic Actor	Interfaces with an IVR system that harasses delinquent loan customers

Table 1: Example Actors In A Bank

1 Actors

1.1 Who Are These Actors?

Since Vertebra is about glueing systems together, it is fairly critical to be able to codify and describe exactly what things your are putting together. With that in mind, an actor is the fundamental unit of **code** in Vertebra. More colloquially, they are "where the rubber hits the road".

Each actor offers various operations that can be done on certain resources. How exactly this is done is formalized in later chapters.

1.2 All The World Is A Stage...

Let's take an example of a bank. This is a fairly complex organization with many systems in need of management. Ignoring the higher-level administration of those systems, there are many pieces of code that will directly interface with a number of other systems at a much lower level. Table 1 gives a number of example actors.

This list of actors is obviously fanciful and incomplete, but it should indicate that the focus of actors is the programmer.

It is worth noting that, just like real-world code, sometimes two actors perform the same function for different pieces of equipment. This is intentional and should make actors the natural point to create consistent interfaces. It is our belief that this aids in refactoring as well, since it keeps the focus on making access similar resources uniform.

Whatever way makes sense for a programmer to encapsulate the code that really does the work, can be factored into actors. This is where all of the concerns about where code runs or how it works is addressed. This is where it is necessary to deal directly with the vagaries of hardware access, user interfaces, and all of the details that crop up in a large system. The rest of Vertebra handles knitting them together at a much higher level.

2 Agents

2.1 Agent Who?

While actors are the fundamental unit of **code**, agents are the fundamental unit of **deployment**. A dizzying array of different issues must be managed when deploying code. Most programmers are ill-equipped to address these issues. They are primarily the domain of administrators. By removing to need for these two groups to depend wholly on one another, we can reduce some of the mismatch that invariably must be overcome to implement a deployable system.

The primary deployment issues we address are:

- security
- provisioning
- configuration

Mixing the above needs with the actual code would only serve to interfere with the purpose of that code while giving a haphazard treatment to those same needs. Vertebra neatly handles the above by grouping actors into agents, which have varying groups of actor code, custom configuration, and an array of provisioning needs.

2.2 Your Papers Seem To Be In Order...

Agents are where credentials first appear in Vertebra. Controlling who runs a certain piece of code and what power they have over the system as a whole is critical. The agent is the beginning and end of identification. With this single building block, we can express a wide variety of permissions and security roles.

2.3 Who Am I?

Agentsalso provide the basic provisioning information that gives actors context. Without knowing where it is running, an actor can't flexibly determine how it is supposed to operate. The agent level is where actors are provided with this context.

2.4 What Do I Do?

Just because the actors know how to handle a certain piece of equipment doesn't mean that they can magically infer everything necessary. The agent also can provide a base-level of configuration that doesn't make sense to anyone else.

2.5 Why Here?

I can hear the sound of thousands of administrators quaking in fear. Many of them have a love (or even obsession) with provisioning, securing, and configuring centrally over the network. The thought is that this eases administration by centralizing those concerns.

In the Cloud, we have discovered that this centralization is directly at odds with reliability and scalability. Anyone who has witnessed the carnage when database servers or authentication servers go offline will appreciate that some things should be configured locally. This level of configuration allows you to do so–which should, in turn, allow your system to make a best-effort in the event of catastrophe. Similarly, you'll never have to worry about scaling or replicating your LDAP server or RADIUS server.

That said, Vertebra does provide a central configuration store in Entrepôt. While we would love for all of your critical configuration to go there, we recognize that some information just makes sense at the leaves of your network, not somewhere in the center. Our core data storage specification is detailed later[1].

3 Resources

3.1 What Is A Resource?

Perhaps the most fundamental concept in Vertebra is that of a resource. Conceptually, resources are the fundamental unit of **addressing** in Vertebra. In the abstract, a resource represents something that matters to your application. It might be a certain piece of data, a certain point of control, or a certain type of behavior. Vertebra doesn't give it any meaning, your application does.

Just like concepts in your application, resources can have relationships to each other. In order to keep things relatively manageable, Vertebra's understanding of relationships is limited to a strict hierarchy. In object-oriented parlance, it is the **is-a** relationship embodied by a single inheritance model. More formally, this means that any agent that can be identified by a certain resource, should also be identified by any ancestors of that resource.

Given this framework, a rich variety of concepts should be representable. A simple example of a set of resources are given in table 2. A visualization of the hierarchy they produce is given in figure 1. Note the many dissimilar concepts are represented in this resource hierarchy:

- Citizenship

[1]see page 10

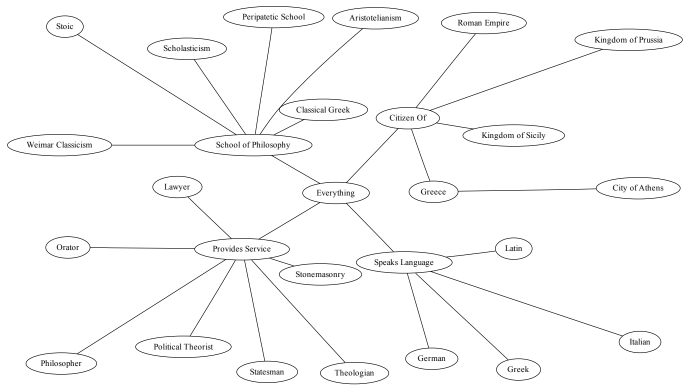

Figure 1: Visualization of the Resource Hierarchy

- Geography
- Professional Trades
- Language Fluency

In the example, we have used a path notation which lists all of the ancestor resources, prepended to the resource directly offered. So the the entry "/speaker/greek" indicates that Socrates speaks greek, which also implies that he speaks at all.

3.2 Resources Are Groups Of Agents

Hopefully it is evident from the above discussion that resources are never actually realized *per se*. There is never a specific data structure or thing that can be said to be a resource. Instead, a resource is more like a description or a tag. What is significant about it is the set of things that it identifies.

Consequently, it's very useful to think of resources as sets of agents. Agents that have something you want! That's convenient, because it is very easy to use common techniques to visualize groups of agents and their relationships. An example using a Venn Diagram is shown in figure 2.

3.3 Resource Advertisement

In the more concrete sense, an actor is said to "provide" a resource. When an actor "provides" a resource it causes any agent that is configured with that actor to "advertise" the same resource (or perhaps an ancestor of it). This causes requests to operate on that resource to be directed to that agent, which subsequently directs it to the appropriate actor.

TODO: Put a bank example here.

Philosopher	Resources
Socrates	/citizen/greece/athens /philosophy/classical_greek /service/philosopher /service/stonemasonry /speaker/greek
Aristotle	/citizen/greece/athens /philosophy/aristotelianism /philosophy/peripatetic /service/philosopher /speaker/greek
Cicero	/citizen/rome /philosophy/stoic /service/lawyer /service/orator /service/philosopher /service/political_theorist /service/statesman /speaker/greek /speaker/latin
Thomas Aquinas	/citizen/sicily /philosophy/scholasticism /service/political_advisor /service/philosopher /service/lecturer /service/theologian /service/latin /speaker/italian
Nietzsche	/citizen/prussia /philosophy/weimar_classicism /service/philosopher /speaker/german

Table 2: Resources Possibly Offered By Various Famous Philosophers

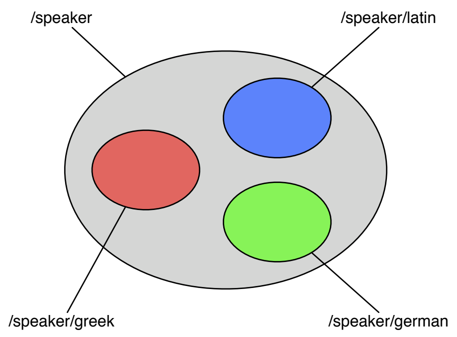

Figure 2: Venn Diagram of a Resource Hierarchy

3.4 The Root Resource

There is one final bit of errata. It may not be so obvious from the above example, but all resources descend from a common, catch-all ancestor—the root resource. In our path notation, it is identified by the path "/".

The root resource is rarely advertised on its own, but can be very useful when resources are used for security and discovery purposes.

3.5 The Advertising Process

You may have noticed that I brushed over the details of "advertising". Unfortunately we don't have all of the basics necessary to discuss advertising. What I can tell you is that the agent sends a message to the Vertebra security agent listing which resources are offered. We'll cover some more of the specifics in 7.

4 Discovery

4.1 The Unique Issues of the Cloud

Now that we have built a flexible system for representing the resources that matter to our applications, it is important to be able to work with them. This is where the Cloud diverges from most programming endeavors. On the smaller scale, most programmers spend much of their time fighting with how their data is represented.

Holy wars are waged over such cherished technologies as relational databases, document schemas, object-relational mappers, and interchange formats. Resources allow Vertebra to be fairly agnostic to these issues. We don't care what you are exchanging, we just want to help it get there.

4.2 The Cloud Makes “Where?” Hard

That brings up the unique problem of scaling. In a large enough deployment, it is very expensive to have anything centralized. Most people focus on making increasingly efficient centralized components–hoping to push off the scalability problems as far as possible. In Vertebra, we accept that you may or may not be centralized, but that your primary concern is locating that data–or service, or employee. Resources save the day in that respect.

For this to be useful, Vertebra provides a facility called “discovery”. Discovery allows you to ask for some group of agents by giving their fingerprint as a set of resources. Vertebra will then handle distributing your request to all of the appropriate places for it to go.

4.3 Scatter-Gather Computing

To make this work, agents advertise their membership in the groups for which their actors provide resources. This makes it possible for any requesters to discover the other agents in the cloud that can fulfill their requests. With discovery, it’s easy to spread out data by discovering the destination (i.e. *push* or *scatter*). Conversely, it’s also easy to collect data by discovering the source (i.e. *pull* or *gather*). In this way it is possible to offer facilities similar to “publish-subscribe” systems, but potentially with services instead of data—potentially even for legacy services that are otherwise difficult to integrate with such systems.

4.4 Discovering A Teller

TODO: Provide an extension of the bank example that illustrates discovery. Possibly do so by putting a bank example in the resources section, to give a more comprehensive example set.

4.5 The Discovery Process

Just like “advertising”, the specifics of “discovery” will be deferred until 7. In the same vague language used before, the agent sends a message to the Vertebra security agent requesting another agent which offers the appropriate resources.

5 Overview

Now that we have a number of useful abstractions, it’s time to do something with them. For this purpose, Vertebra has operations. The operation is the fundamental unit of **work** in Vertebra. At any instant, they tie the pieces together to make something happen.

Specifically, when an operation is issued, the system discovers agents that offer the appropriate resources, then operation is dispatched to those agents. The agents further dispatch the operations to the appropriate actors which actually do the work. Finally, the responses stream back to the agent that made the request for the operation.

6 Scope

6.1 Selecting Agents

Exactly which agents are selected for an operation is determined by the resources in the operation and the scope. Thus, scope determines the behavior of **agent selection** in Vertebra. While it may not be initially obvious, different modes of selection provide the building blocks for implementing a number of useful scenarios which are detailed later.

6.2 Single Scope

The simplest scope that Vertebra provides is the single scope. The purpose of the single scope is to ensure that the operation is executed by exactly one actor, exactly once, somewhere in the Cloud.

To dispatch a single scoped operation, the client code discovers agents capable of providing it, then randomly iterate through them. Each iteration, the operation is dispatched to the selected agent. If it executes, iteration stops. If it can't perform the operation, iteration continues to the next agent. If all agents refuse the operation, the operation blocks and retries at a later point.

6.3 All Scope

The next scope that Vertebra provides is the all scope. The purpose of the all scope is to ensure that the operation is executed by as many actors as can be reached in the Cloud.

To dispatch an all scoped operation, the client code discovers agents capable of providing it, then dispatches to all of them simultaneously. Any or all of the operations could fail. It may even be that no agents provide the operation.

7 Direct Operations

7.1 The Chicken and the Egg

Given all of the talk of discovery and dispatch, there is an exception to those rules. There are some cases where discovery is not necessary or possible. The primary case we're concerned with is the operations of advertisement and discovery themselves.

Rather than building a special protocol for doing these operations, we just use the normal operation behavior, without the discovery step. We call these "direct" operations.

Component	Description
Scope	a string that determines the dispatch and completion requirements of the operation
Type	a string that determines which operation is dispatched
Parameters	a mapping of parameter keys (strings) to various parameter datatypes

Table 3: Operation Components

7.2 Scope In Direct Operations

Scope in a direct operation is almost unused. The only aspect of scope that is important is the completion semantics. That is, for single scope, the operation must be dispatched to exactly one actor, but the all scope, zero or more actors may handle a request.

8 General Operations

8.1 Structure

Everything that happens in Vertebra is an operation. Conceptually, operations are the fundamental unit of **work** in Vertebra. With the exception of some infrastructure (i.e. direct operations), they are all general operations. A general operation takes place in a few phases that provide various execution guarantees.

An operation consists of the components listed in table 3.

8.2 Operation States

The life of an operation goes through various states, as shown in figure 3.

- **New** — Operation has been requested but has not started.
- **Ack** — Operation has been acknowledged as allowed and may be starting.
- **Nack** — Operation has been denied due to security policy, operation-specific issue, due to load, or due to old discovery information.
- **Run** — Operation is executing.
- **Done** — Operation has finished.
- **Error** — Operation has failed at an application level.

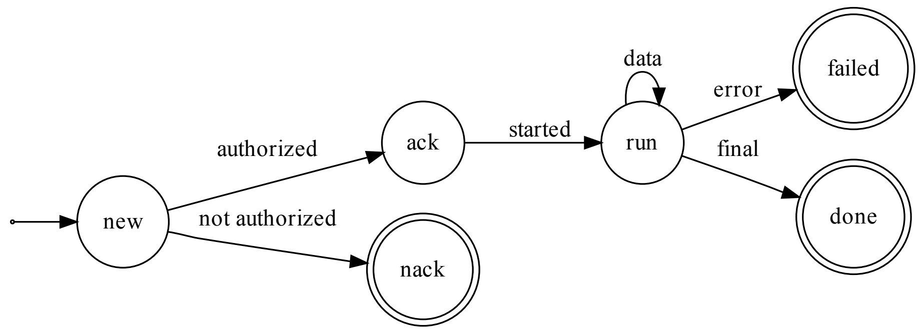

Figure 3: Operation States

9 Core Data Storage

9.1 Requirements

To orchestrate services in the cloud, information of a certain character must be stored and queried. A data store for Vertebra needs to have a few characteristics:

- it should be fault-tolerant
- it should be horizontally scalable
- it should be keyed similarly to operation discovery
- it should have values that easily adapt to become input to operations
- it should have a versioning capability

The Vertebra implementation provides a data store with these properties[2] in the form of Entrepôt.

9.2 Data Format

Entrepôt is, at the highest level, a hash-table. It simply maps keys to values. The structure of these keys and values are such that they fit well into the rest of the Vertebra system.

9.2.1 Keys

Keys are made up of sets of `resources`. This allows for records to be discovered in a similar way that `agents` are discovered.

[2]TODO: versioning isn't done yet.

9.2.2 Values

Values are exactly the same structure as the parameters to an operation. Functionally they are equivalent to a mapping of parameter keys to values.

9.2.3 Records

Each mapping is called a record. Records are represented as a hash containing:

keys which maps to a mapping of key names to resources

values which maps to a mapping that holds whatever data is being stored

9.3 Operations

9.3.1 Store

To store a value in an Entrepôt data store, do an operation of type "store" with scope single. This guarantees that exactly one copy of the entry is stored. The parameter mapping for this operation should be the record.

To delete a value in Entrepôt, store an empty value. If a record is replaced (or deleted), the previous record is returned as data.

9.3.2 Fetch

To retrieve a value, there are two options depending on the type of query you're attempting.

To find all records which have a key that contains the query keys, use an operation of type "fetch_superset". To find all records which have a key that is contained by the query keys, use an operation of type "fetch_subset". In either case, the queries should be made with a parameter named "keys".

The data returned will consist of a mapping containing records of the form used in the Store operation.

9.4 Caveats

Entrepôt makes its best effort to update and delete values. However, it is not currently completely resilient. This will be addressed later when versioning is supported. The current shortcomings are mostly of concern in federated settings where there is a significant likelihood that communication between two servers is significant.

Provisioning is also a concern, since an Entrepôt agent will not be discovered unless it contains resources for all of the keys in a request. This should be handled by direct operations for the initial store.