Vertebra

Architecture & Implementation

Jayson Vantuyl

Engine Yard, Inc.

Foreword

TODO: Have somebody write a foreword.

Preface

Vertebra, like so many projects, started with very humble beginnings. In beginning mid-2006, Tom, Lance, Ezra, and I founded Engine Yard[1]. We had just finished combining emerging virtualization[2] and clustering technologies[3] into a rather impressive little clustered hosting platform.

We were in a place where all too many people had been before. We had built an incredibly dense system, but it was quite fault tolerant. Any system built to manage it needed to be equally fault tolerant. While fault tolerant systems and fault tolerant management tools are rare, the latter is virtually nonexistent. There was certainly nothing in our price range.

At the time, I was fresh off helping a good friend of mine with the occasional management and architecture for his one-man ISP. Similarly, my recent work had been providing support services for a wide array of customers that each had relatively small but heterogenous IT deployments. Even before that, I was managing large networks for pretty much the previous decade.

With these experiences fresh on my mind, I immediately knew that we couldn't just manage it all by hand. I knew that if we did try, we would all soon succumb to a series of grisly mental disorders. As luck would have it, Tom was also a fan of automated deployment[4] and was friends with enough people in the hosting business to recognize the issue. Ezra, too appreciated the trouble that automating a system can be. Lance, well, he's the CEO and he knows that we have this sort of stuff covered.

With the founders in agreement and appropriate trepidation, we began discussing exactly what features we would like from a sensible system for automating our clusters. As the project progressed, it became clear that what we had in mind didn't particularly exist yet. It also became clear that it was applicable to a much larger set of problems than mere systems automation. Thus, Vertebra was born.

Insomuch as it has automated our systems, Vertebra has been effective, albeit a bit behind schedule. However, I hope that it is clear in the design of this wonderful architecture that it was worth the wait.

---

[1] A quite successful Ruby-focused Hosting Provider turned Technology Company, `http://www.engineyard.com`

[2] Xen, to be specific, `http://www.xensource.com`

[3] Redhat Cluster Suite, `http://www.redhat.com/cluster_suite/`

[4] He wrote the book on Capistrano, `http://www.capify.org/`

Contents

List of Tables

List of Figures

Introduction

Identifying the Problem

While the section on use cases covers the specifics, I feel it's important to briefly describe the nature of the problem that Vertebra tries to solve.

As anyone who has had to do so will tell you, writing a program is easy; writing a fast program is harder; writing a scalable program is even harder; and writing a fast and scalable program is downright difficult. If the only people we ever asked were programmers, that would be the only breakdown that you would probably see.

However, there is an unfortunate class of people that have to deal with programs after they've been written. While every programmer would like to believe that their programs are all beautiful works of art, a number of operators and administrators out there would disagree.

Thus, Vertebra is not about making a framework that merely solves the problem of making scalable program. Rather, Vertebra is about making scalable programs manageable.

Identifying the Tradeoff

Like most programming endeavors, Vertebra has been a balancing act between a desire for simplicity and a desire for correct behavior. While it can sometimes take a lot of work to get it just right, we've come to appreciate that no aspect of a system matters if it doesn't work correctly. We've gone so far as to claim that we can maybe even make it work correctly through terrible hardships.

This makes sense because, in a systems environment. When a network is failing, or a system disappears, almost all metrics of system performance mean little—*still working*, however, means a lot. Handling failures gracefully and being able to scale are infinitely more important to a successful application.

With that in mind, we've focused on making Vertebra scalable and functionally correct. We've also identified some behavior we'd like to avoid. In fact, Vertebra is sometimes more interesting for what it doesn't do that what it does do. Again, this largely considers failure scenarios. While this might seem pessimistic, the presence of the Service Level Agreement as a cornerstone of hosting agreements underscores its wisdom.

Cloud Glue

Given the problem of making scalability manageable, and the understanding that our focus is on administrators, it made sense to think of Vertebra as middleware between various heterogenous systems. In terms of Cloud Computing architectures, or perhaps almost any computing framework, management is typically an afterthought. This is unfortunate, since management tools are commonly a critical deliverable.

Functionally, Vertebra provides the "glue" that holds your various clouds together. Like any good glue, it is designed to be able to fit into the nooks and crevices in your infrastructure. Once applied, it should hold your systems together with a durable bond.

It is our sincere hope that, with this abstraction in place, it will become easier to re-factor things at an organizational level. This is commonly an area where most programmers aren't allowed to tread. It is perhaps where most administrators have the fewest options, as well.

Vertebra's Ultimate Purpose

By itself, Vertebra does almost nothing especially innovative. The power comes from the code that is plugged into it and the encrusted systems that it makes accessible. While this could be said of any framework, we like to think that the model we've made is useful for organizing and orchestrating everything else you have to work with. In fact, we put it together in these ways because we couldn't quite find anything that worked the way Vertebra does.

The primary concern that consumes the lives of many administrators, programmers, and managers is *integration*. As the Cloud scales the number of integration points, it's inevitable that it will consume that many generations of support staff as well. Vertebra is designed to give those people at least some of their life back. It's designed to put the focus on integration, normalizing the management of your infrastructure, and growing it flexibly.

The greatest strides in managing technology projects have recently come from the renewed appreciation that human-time is the most expensive aspect of technology. We hope that Vertebra will do for these workers what dynamic languages have done for programmers. In that way, we hope that Vertebra will be revolutionary—or, at least, evolutionary.

Part I

Use Cases

Chapter 1

Management Cases

The following Use Cases show management features that are beneficial for Vertebra to emulate. Some are couched in the example provided by another implementation. Many of them are incomplete with respect to the full Vertebra ecosystem. However, the core benefits of each one are embodied in Vertebra in one way or another.

There are also some Anti-Use Cases. These provide management behaviors that we absolutely want to avoid. You may notice that some technologies appear as a Use Case and an Anti-Use Case. This is intentional, as often software that is good would be better without some of its baggage.

1.1 Composition and Cascade

One of the signs of a good data model is "Don't Repeat Yourself" (DRY). If you have to record the same information in two different places (and it's not an optimization), then you have likely failed to model the data well.

DRY is a good rule of thumb for a programmer. It's practically the **only** rule of thumb for an administrator. If the administrator continually has to repeat himself, his job will rapidly scale out of control. The use cases in this section are all written with that fact in mind. Each of these cases identifies a decision to choose powerful ways to reduce and avoid repetition.

1.1.1 Use Case: Composition of Permissions

Description Permissions should be built holistically. In the absence of permissions, we should be default-deny, an absence of sufficient permissions should be presumed to deny access to a secured entity.

Starting with this empty slate, permissions should be found that, in aggregate, allow an operation. This is to mean that it operations are permitted by the composition of any assigned permissions, by the cascading of permissions from ancestors, or possibly by compositions of those cascaded permissions.

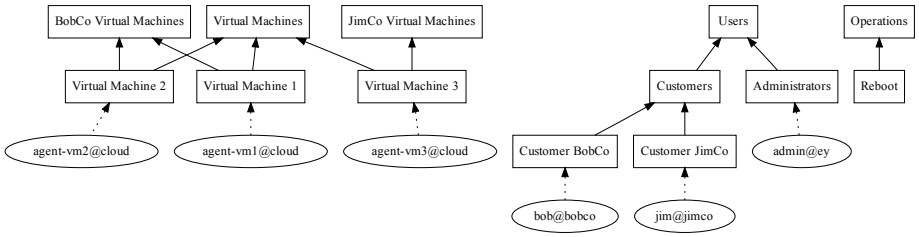

Figure 1.1: Sample Security Entities: BobCo and JimCo

Cascading permissions should preserve the information about these relationships. Thus, the cascade should be computed at the time of access, not at the time of assignment.

Discussion Starting with the abstract concept of a security entity, we can build a simple model for representing permissions. We shall not differentiate an entity which initiates an operation from an entity upon which the operation is initiated. This is largely because such entities may trade places at any time and, as such, any distinction would be meaningless.

Our security entities can be thought of as inheritance hierarchies. These hierarchies should allow multiple inheritance, thus they are technically directed-acyclic-graphs, rather than a strict hierarchy or tree.

Depending on the semantics of the relationship, we may think of descendants in the hierarchy as being of some "sub-type" of the "types" represented by their ancestors or as being "members" of the "groups" represented by their ancestors. In either case, they may be treated the same, and any terminology is only a handy tool for discussion, not an abstract constraint on the security model.

Example An example diagram of a set of related objects is given in figure 1.1. Rectangles indicate abstract security entities. Ellipses mark concrete security entities. Dotted lines show association of the two. A set of permissions on this model are shown in table 1.1.

Permissions		
#	Source Entity	Destination Entity
1	Users	Reboot
2	Administrators	Virtual Machines
3	Administrators	Operations
4	Customer BobCo	BobCo Virtual Machines
5	Customer JimCo	JimCo Virtual Machines

Table 1.1: Sample Permissions

The policy specified here is very powerful. Users are allowed to Reboot. Administrators (and thus *admin@ey*) are given the access to Virtual Machines (i.e. *agent-vm1@cloud*, *agent-vm2@cloud*, *agent-vm3@cloud*). BobCo and JimCo are given access to just their respective machines.

If *bob@bobco* were to attempt to issue an operation of the form Reboot(Virtual Machine 1), permissions 1 and 4 combine to allow the operation. Specifically, since *bob@bobco* is a descendant of Users, permission 1 applies. Since *bob@bobco* is a descendant of BobCo, permission 4 applies.

If *bob@bobco* were to attempt to issue an operation of the form Reboot(Virtual Machine 3), no combination of permissions allow this.

If *admin@ey* were to attempt to reboot any machine, permissions 1, 2, and 3 would combine. Note that permission 1 is obviated by permission 3, but either would apply.

What is most interesting about this model, is the flexibility with which new virtual machines can be added, new users added, old users removed, new administrators added, et cetera. At each point, no more work must be done other than to associate them with their respective abstract entity, and all permissions are applied.

This serves to help separate policy from permissions, which is absolutely critical in larger systems.

Another important feature that is not obvious from this example is that it is possible to allow someone to have two roles (i.e. inheritable permissions groups) with conflicting duties, but to prevent them from exercising their permissions simultaneously. Assume, that we added a consultant who could operate on virtual machines for both BobCo and JimCo. With a simple restriction, it is possible to only allow that consultant to activate one set of permissions or the other at a given time.

In addition these dynamic extensions, other rich extensions can be defined. Examples include time-based limitations or static limitations (i.e. no one can ever be provisioned with a certain conflict being permitted). These allow further specification of policy with a minimum of management. Whether these extensions will ever be implemented in Vertebra is currently undecided, but an exciting possibility nonetheless.

Related Implementations

- Novell Netware™

1.1.2 Anti-Use Case: Direct Permissions

Description Permissions are assigned to users directly, or possibly to simple groups. Objects are secured by some list of users and groups.

Discussion With perhaps the exception of a systems lacking a model of security, this is probably the most common model in large scale deployment. While

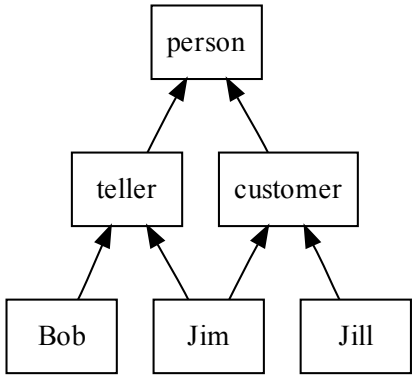

Figure 1.2: Bank Example Model

it can be functional, there are security concerns that cannot be addressed in this model.

The primary useful one is called "Dynamic Separation of Duty". Specifically, a situation can exist where there is a conflict of interest if a person has two permissions at the same time.

Consider a bank. It is possible for a person to be a teller at a bank. In some (likely most) cases, that teller may also be a customer of the bank. It is a grave conflict of interest for a person to be a teller and also handle their own withdrawal or deposit. This is because the teller is responsible for verifying the integrity of those transactions. It is a long understood fact that avoiding situations where a person must protect their own interests and business interests is a prudent business practice.

With static permissions, there is not a good way to represent such constraints. Some people compensate by creating multiple user accounts for a person. However, this makes management much more complex, since administrators must ensure that these rules are followed, audit that they are maintained correctly, and manage many more user accounts doing so.

Example Since this is an Anti-Use Case, we will demonstrate the pathological behavior described above. A diagram of the model is in 1.2.

In this particular case, there is no combination of permissions that can allow Jim, who is both a customer and a teller, to function both as a customer and a teller without enabling him to process his own transactions, which is a clear conflict of interest.

Related Implementations

- Microsoft Windows™NTFS
- Unix™

1.1.3 Use Case: Cascading Configuration

Description TODO: Just like permissions, configuration can come from above.

paragraphDiscussion

TODO

Example TODO

Related Implementations

- World Wide Web Consortium (W3C) Cascading Style Sheets

1.1.4 Anti-Use Case: Lattice Configuration

Description TODO: A matrix of boxes, all ugly. Let the computer do the work.

paragraphDiscussion

TODO

Example TODO

Related Implementations TODO

1.2 Lessons From A Print Queue

1.2.1 Use Case: Job Enumeration

Description TODO: What's running is important, eh?

paragraphDiscussion

TODO

Example TODO

Related Implementations TODO

1.2.2 Use Case: Job Management

Description TODO: If it runs, kill it! Or pause it! Or send some other exciting event toward it.

paragraphDiscussion

TODO

Example TODO

Related Implementations TODO

1.2.3 Anti-Use Case: The 500 lb. Bomb

Description TODO: Try to kill job. Job just won't die until completion. Badness.

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

1.2.4 Anti-Use Case: The Job That Wouldn't Die

Description TODO: Rather than waiting until completion, this job just hangs forever. Also sucks.

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

1.3 Forensics, Records, and Other Administrivia

1.3.1 Use Case: Distributed Logging

Description TODO: Spread Out The Love.

Example TODO

Related Implementations TODO

1.3.2 Use Case: Meaningful Logs

Description TODO: Useful tagging and severity.

paragraphDiscussion
TODO

Example TODO

Related Implementations

- Unix™ syslog

1.3.3 Anti-Use Case: Central Logging

Description TODO: Log shipping is bad.

paragraphDiscussion

TODO

Example TODO

Related Implementations TODO

1.3.4 Use Case: Log Querying

Description TODO: Using a tool to report on the logs is okay. Querying logs scales much better than "reading" logs. Seems like the same thing, but it's not.

paragraphDiscussion

TODO

Example TODO

Related Implementations

- Erlang log_mf_h
- Erlang Report Browser

1.3.5 Anti-Use Case: Lossy, Insecure Logs

Description TODO: No security. Lossy delivery. Not distributed.

paragraphDiscussion

TODO

Example TODO

Related Implementations

- Unix™Syslog

1.3.6 Anti-Use Case: Inextensible Log

Description TODO: Binary format is nuts. Requires compiled code to extend log message types.

paragraphDiscussion

TODO

Example TODO

Related Implementations Microsoft Windows™Event Log

1.3.7 Use Case: Distributed Auditing

Description TODO: Helps diagnose catastrophes. Also reduces trust-spread for audit log. No intermediate can tamper with the audit logs because you always get them from the source. Also handy because we get both sides of the story, which can be reconciled to detect lies.

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

1.3.8 Anti-Use Case: Inextensible Audit Log

Description TODO: Limited audit events. Not distributed. One place to tamper. Difficult to query. Requires compiled code to extend audit message types.

paragraphDiscussion
TODO

Example TODO

Related Implementations

- Microsoft Windows™Audit Log

Chapter 2

Failure Cases

2.1 Use Case: Freeze and Thaw

Description TODO: Major comms drop, system waits. Recoverability.

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

2.2 Anti-Use Case: Freeze and Shatter

Description TODO: Major comms drop, system falls apart. Epic Fail!

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

2.3 Use Case: Endpoint Queueing

Description TODO: Something hangs up, quenched at the source.

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

2.4 Anti-Use Case: Central Queue

Description TODO: Something hangs up, queue 'splodes.

paragraphDiscussion

TODO

Example TODO

Related Implementations TODO

2.5 Use Case: Exception To Human

Description TODO: Rails Errors to Inbox.

paragraphDiscussion

TODO

Example TODO

Related Implementations TODO

2.6 Anti-Use Case: Exception To Log

Description TODO: Error disappears into the abyss.

paragraphDiscussion

TODO

Example TODO

Related Implementations TODO

2.7 Use Case: Deus Ex Machina

Description TODO: God's hand reaches down and makes everything right.

paragraphDiscussion

TODO

Example TODO

Related Implementations TODO

2.8 Anti-Use Case: Try Again

Description TODO: Failure to requester, possibly wedged transaction must be undone and then redone. Hanging in the middle much more graceful. Operators can help with this. Very suitable to hosting.

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

2.9 Use Case: TCP

Description TODO: Don't lose commands. Retry if necessary. Detect and remove retransmissions. Catch when the other side loses state.

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

2.10 Use Case: SCTP

Description TODO: Parallel Streams == Network Drano™

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

2.11 Anti-Use Case: TCP

Description TODO: Sequence Numbering Clogs Pipe

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

Chapter 3

Systems Automation Cases

3.1 Use Case: sudo

Description TODO: Controlled, fine-grained permission elevation mechanism.

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

3.2 Use Case: Privilege Activation

Description TODO: Controlled, fine-grained permission retention.

paragraphDiscussion
TODO

Example TODO

Related Implementations

- Linux™ Capabilities (a.k.a caps)

3.3 Anti-Use Case: Trusted Everything

Description TODO: Running as root makes security impossible.

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

3.4 Use Case: People Processes

Description TODO: Be a cyborg, not a meat grinder.

paragraphDiscussion

TODO

Example TODO

Related Implementations TODO

3.5 Use Case: Workflows

Description TODO: A merry-go-round. For when taking a ride is important.

paragraphDiscussion

TODO

Example TODO

Related Implementations TODO

3.6 Anti-Use Case: Fragile Recursion

Description TODO: How deep does the Rabbit hole go? What happens when you hit bottom? Stack traces don't retrace your steps, they just show the way home.

paragraphDiscussion

TODO

Example TODO

Related Implementations TODO

3.7 Anti-Use Case: Lost In The Inbox

Description TODO: Too much e-mail just gets deleted. Don't queue operator failures, discover operators. Works more like a needy shared mailbox.

paragraphDiscussion

TODO

Example TODO

Related Implementations TODO

3.8 Anti-Use Case: A Ticket In The Dark

Description TODO: Without being assigned, tickets can sit forever. Discover operators.

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

3.9 Anti-Use Case: Left Hand, Meet Right Hand

Description TODO: Coordinate failures of processes into a whole. When a plane crashes, be able to sift the flight plan out of the wreckage. Otherwise, all smoking holes in the ground look alike.

paragraphDiscussion
TODO

Example TODO

Related Implementations TODO

Part II

Concepts

Chapter 4

XML Messaging and Presence Protocol

4.1 Overview

Before we get too deeply into the concepts behind Vertebra, we should probably explore the concepts behind the core technologies that we leverage. Our primary dependency is XMPP, as defined by the Internet Engineering Task Force (IETF). It is fully documented in RFCs 3920[1] and RFC 3921[2].

XMPP brings a number of advantages that are particularly helpful for implementing a distributed application like Vertebra. In no particular order, some of them are:

- Standardized Encoding (XML)[3]
- Transport Level Security (via TLS)[4]
- Authentication
- Automatic Federation (S2S + DNS)
- Federated Authentication (S2S Dialback)
- Client-side Redundancy (DNS SRV Records)
- International Text Support (UTF-8)
- Built-In Extensibility (Namespaces in XML[5])

[1] IETF RFC 3920: XMPP Core, `http://www.xmpp.org/rfcs/rfc3920.html`
[2] IETF RFC 3921: XMPP IM, `http://www.xmpp.org/rfcs/rfc3921.html`
[3] W3C XML, `http://www.w3.org/TR/REC-xml/`
[4] IETF RFC 4346: The Transport Layer Security (TLS) Protocol, `http://tools.ietf.org/html/rfc4346`
[5] Namespaces in XML, `http://www.w3.org/TR/2006/REC-xml-names-20060816/`

- Ordered Delivery
- Distributed Server Architecture (S2S)
- Sensible Message Semantics for Agents (IQ Delivery)
- Mechanism for Creating Server Extensible Protocols (IQ Server Handlers)

4.2 Components

TODO

4.3 Information Query Syntax

TODO

Chapter 5

Actors and Agents

5.1 Actors

5.1.1 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.

5.1.2 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 5.1 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.

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 5.1: Example Actors In A Bank

5.2 Agents

5.2.1 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.

5.2.2 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.

5.2.3 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.

5.2.4 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.

5.2.5 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].

[1] see page 39

Chapter 6

Resources and Discovery

6.1 Resources

6.1.1 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 6.1. A visualization of the hierarchy they produce is given in figure 6.1. Note the many dissimilar concepts are represented in this resource hierarchy:

- Citizenship
- 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.

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 6.1: Resources Possibly Offered By Various Famous Philosophers

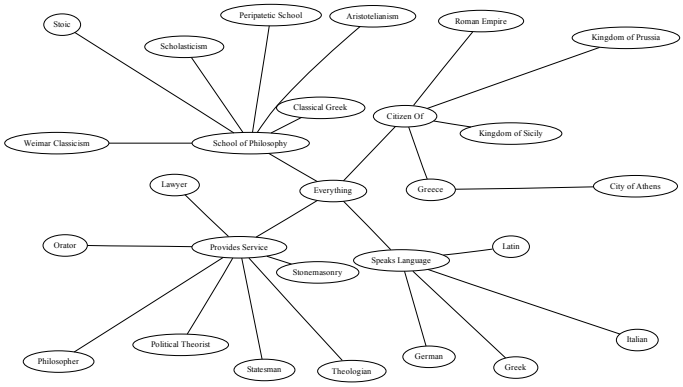

Figure 6.1: Visualization of the Resource Hierarchy

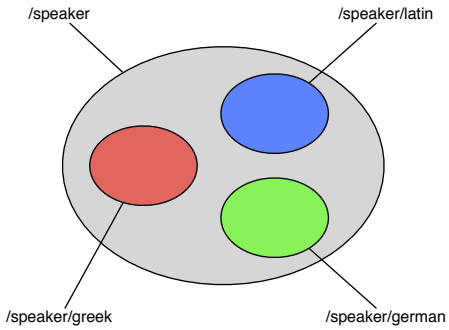

Figure 6.2: Venn Diagram of a Resource Hierarchy

6.1.2 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 6.2.

6.1.3 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.

6.1.4 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.

6.1.5 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 7.3.

6.2 Discovery

6.2.1 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.

6.2.2 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.

6.2.3 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.

6.2.4 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.

6.2.5 The Discovery Process

Just like "advertising", the specifics of "discovery" will be deferred until 7.3. 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.

Chapter 7

Operations

7.1 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.

7.2 Scope

7.2.1 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.

7.2.2 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.

7.2.3 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.

7.3 Direct Operations

7.3.1 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.

7.3.2 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.

7.4 General Operations

7.4.1 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 7.1.

7.4.2 Operation States

The life of an operation goes through various states, as shown in figure 7.1.

- **New** — Operation has been requested but has not started.

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 7.1: Operation Components

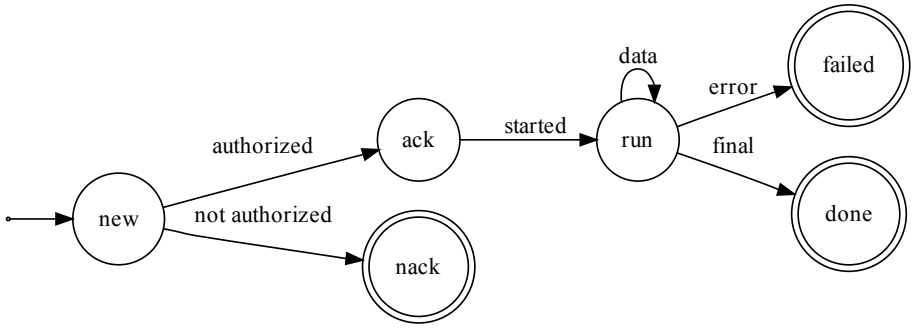

Figure 7.1: Operation States

- **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.

7.5 Pipelines

TODO

Chapter 8

Security

8.1 Roles

TODO

8.2 Authorization

TODO

8.3 Auditing

TODO

8.4 Impersonation

TODO

Chapter 9

Orchestration

9.1 Core Data Storage

9.1.1 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[1] in the form of Entrepôt.

9.1.2 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.

Keys

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

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

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.

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

9.1.3 Operations

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.

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.

9.1.4 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.

9.2 Work Flows

TODO

9.3 Data Flows

TODO

Chapter 10

Administration

10.1 Provisioning

TODO

10.2 Job Control

TODO

10.3 Logging

TODO

Chapter 11

Federation

11.1 Cascading Discovery

TODO

11.2 Cascading Permissions

TODO

11.3 Distributed Permissions and Discovery Cache

TODO

Part III

Components

Chapter 12

Herault

12.1 Roles

12.2 Permissions

12.3 Advertising

12.4 Discovery

12.5 Authorization

12.6 Cascading

Chapter 13

Entrepôt

Chapter 14

SawMill

TODO

Chapter 15

Cavalcade

TODO

Chapter 16

Avalanche

TODO

Part IV

Protocols

Chapter 17

XMPP Implementation

TODO

Chapter 18

Fault Mitigation

TODO

Chapter 19

Direct Operations

TODO

Chapter 20

Advertisement Protocol

TODO

Chapter 21

General Operations

TODO

Chapter 22

Pipelines

TODO

Part V

Appendices and Index

Appendix A

Set Theory Basis of Vertebra Resources

TODO

Appendix B

Discovery as Set Algebra

TODO

Appendix C

Entrepôt Queries in Terms of Sets

TODO

Glossary and Indexes

Glossary of Terms

actor Vertebra's fundamental unit of code, page 23

agent Vertebra's fundamental unit of deployment, page 24

Default-Deny in security, an absence of applicable permissions denies access to the secured object, analogous to *strict construction* in legal systems, page 3

DRY literally "Don't Repeat Yourself"; from agile programming, if the same data is entered in multiple places, it's generally taken to be a sign of a substandard model of the problem, page 3

operation Vertebra's fundamental unit of work, page 34

resource Vertebra's fundamental unit of addressing, page 27

Technology Index

Subject Index