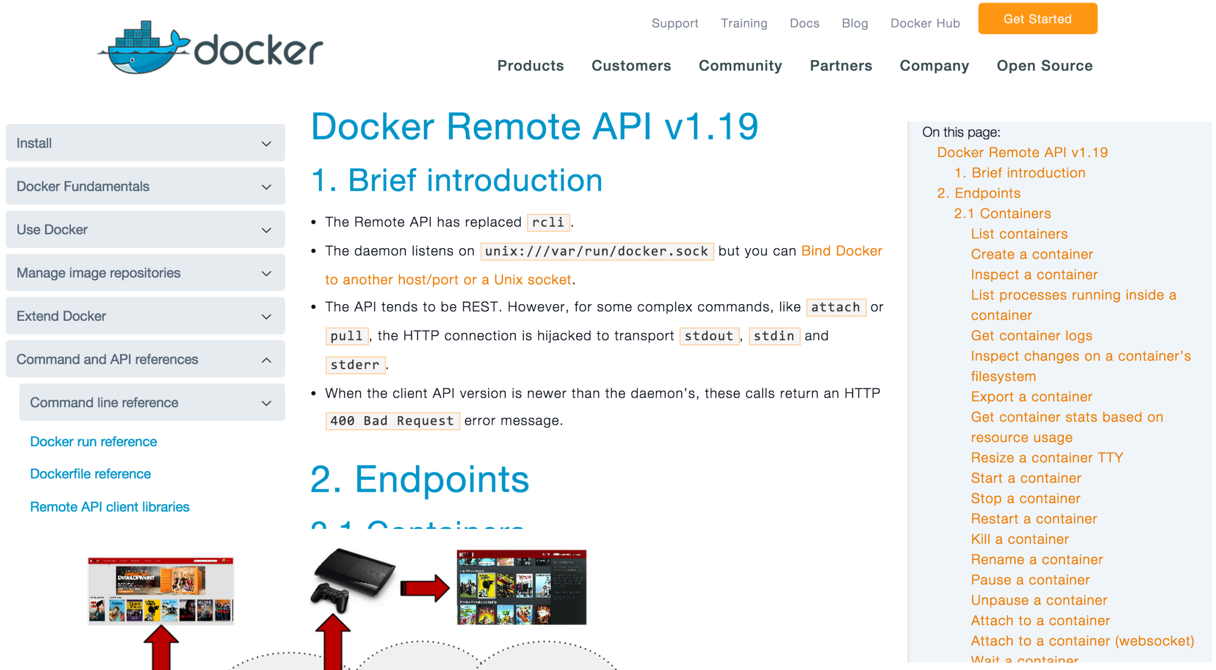
Web APIs and How They Work

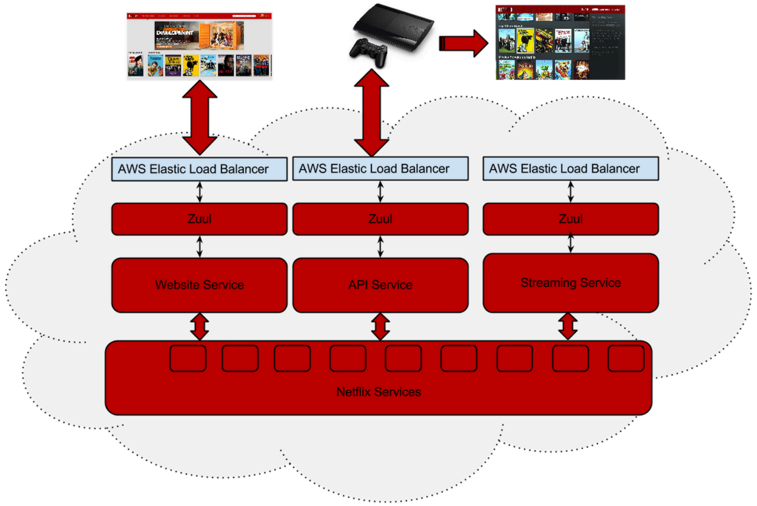
Greg Brail, Apigee

Why APIs?

Web APIs are Everywhere (at HPTS)

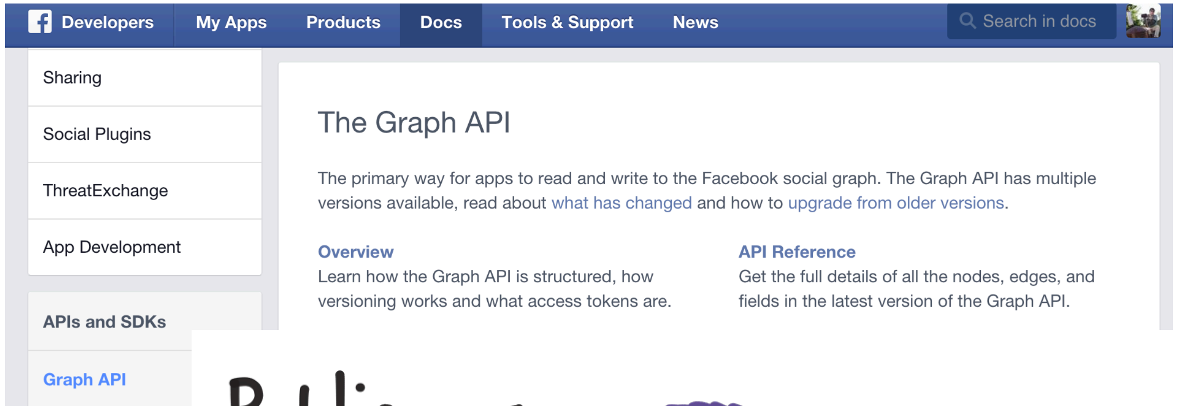


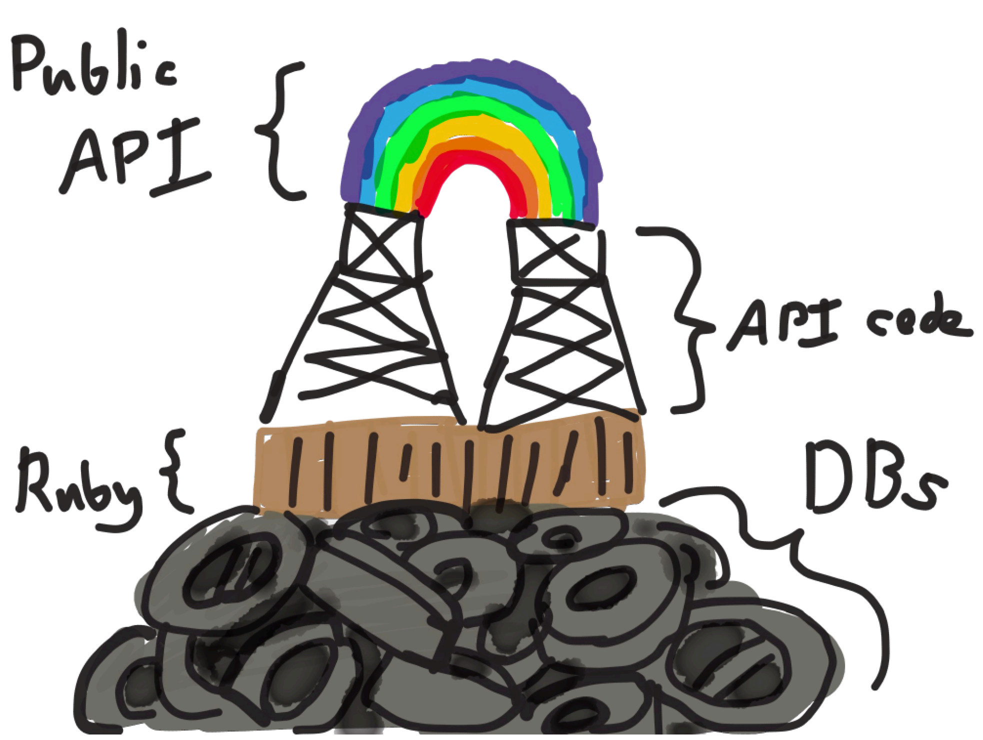
Docker API: https://docs.docker.com/reference/api/docker_remote_api_v1.19/



Netflix Zuul: https://github.com/Netflix/zuul/wiki/How-We-Use-Zuul-At-Netflix

Facebook Graph API: https://developers.facebook.com/docs/graph-api



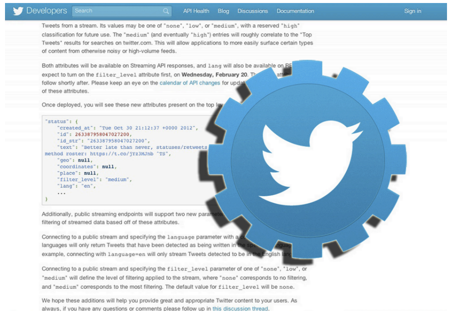


Kyle Kingsbury: https://github.com/aphyr/jepsen-talks/tree/master/2015/goto

apigee

Use Cases for APIs

- **Public APIs:** 5% of the world's APIs*
  - For any developer to discover and sign up
  - Often free
  - *Facebook, Twitter, Twilio, etc.*
- **Partner or Customer APIs:** 25% of the world's APIs
  - For a company's customer or partner to use
  - Often paid or via negotiated business relationship
  - *Third-party apps for Walgreens' photo printing service*
- **Private APIs:** 75% of the world's APIs
  - Used within a single company
  - Documentation and SLAs are still important
  - *Netflix*



* My guess

Web APIs are a Reaction

- What the industry came up with
  - SOAP
  - WS-Security
  - WS-Secure Conversation
  - WS-Trust
  - XML
  - XML Schema
  - UDDI
  - CORBA
  - DCE

- Collective reaction from developers:
  - Yuck!

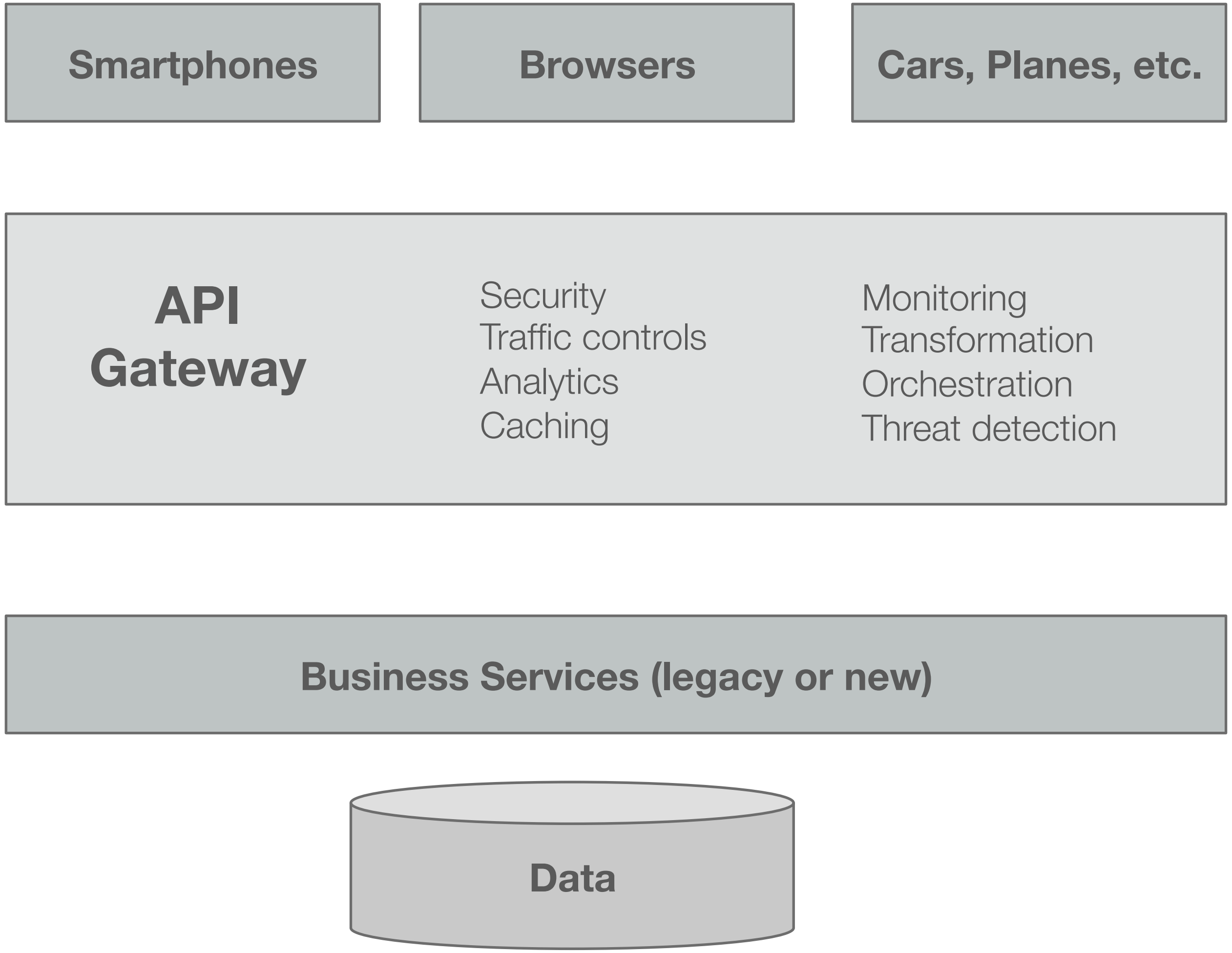
The Power of WDWJ

- **W**hy **D**on't **W**e **J**ust
- Use HTTP
- Use JSON

- No industry consortium
- No standards body

API Gateways and Apigee

API Gateways are Everywhere



- Google
- Facebook
- Salesforce
- Amazon
- Netflix
- Twitter
- Many more…

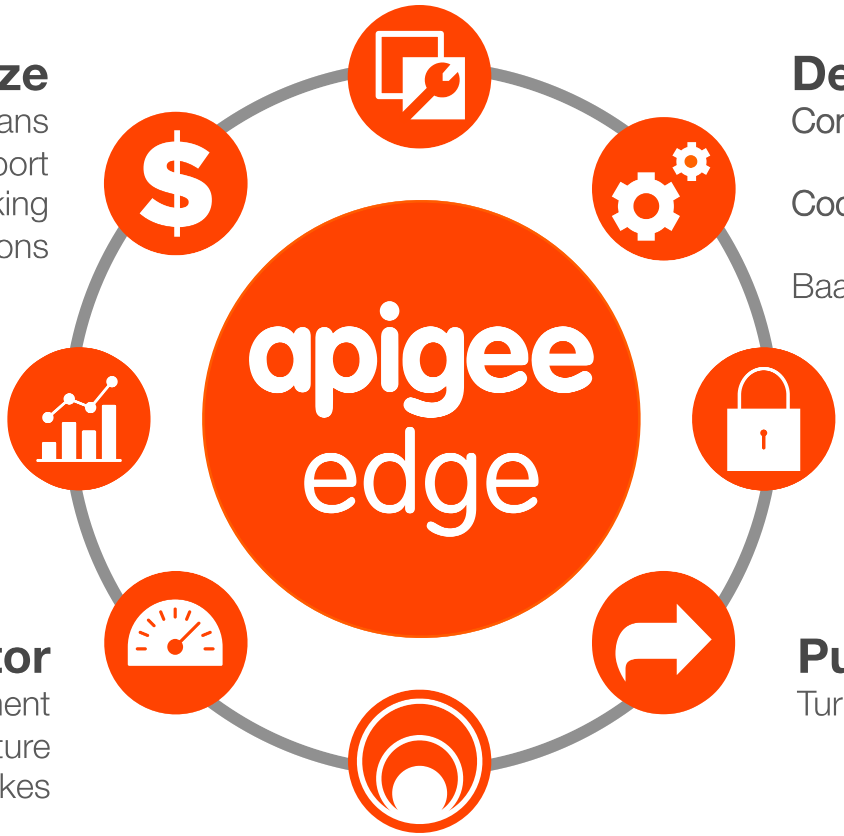
apigee

What do you Do?

Design
Design first. Document Smart.
Full support for Swagger 2.0

Monetize
Flexible rate plans
Internationalization support
Usage tracking
Limits and notifications

Develop
Configuration: Over 30 ready-to-use & configurable `
policies
Code: Built-in support for Node, JavaScript and
Java extensibility
BaaS



Analyze
Complete visibility– from app end to backend
Automatically and continuously collect
all API-traffic data out of the box

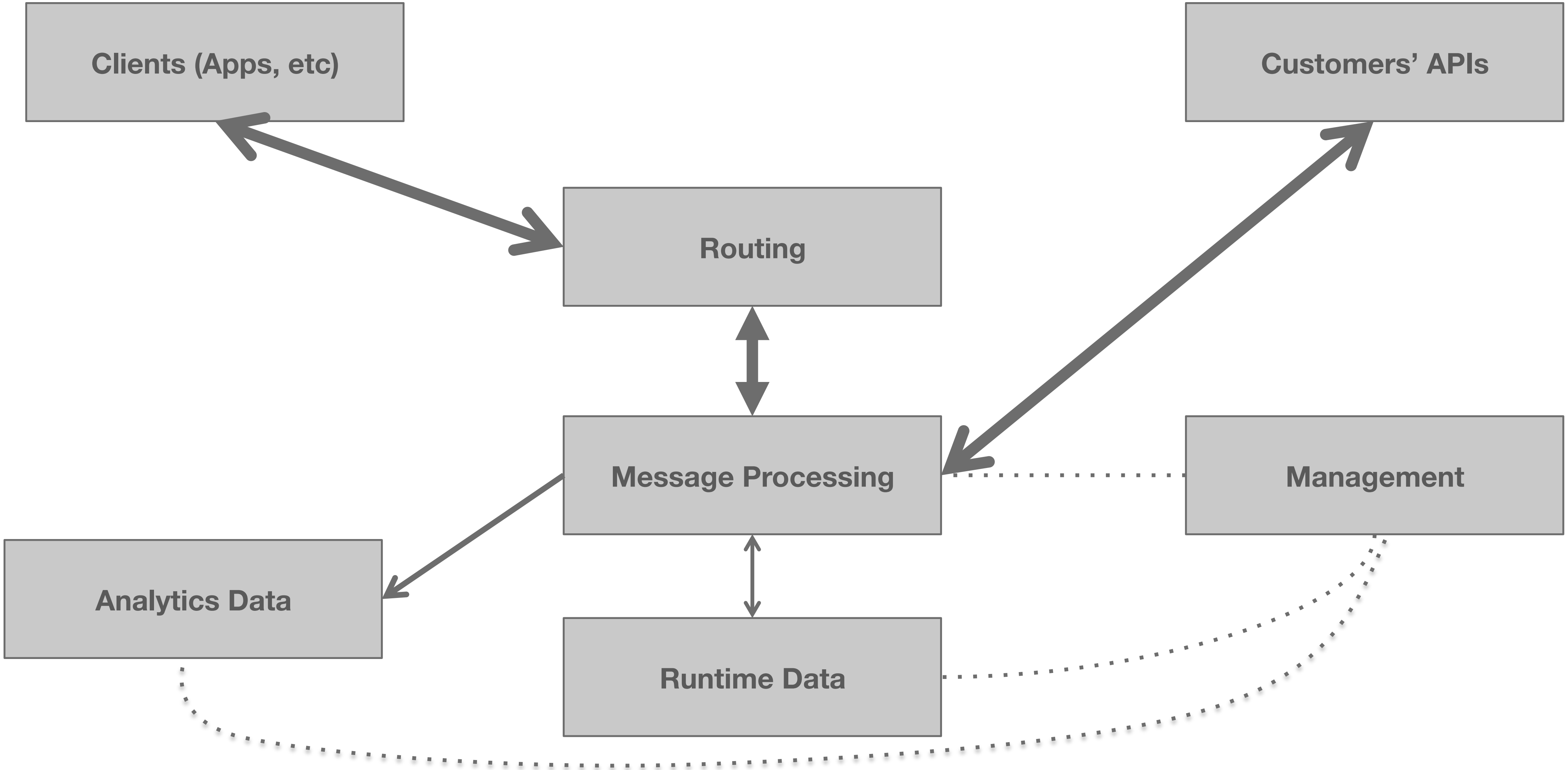
Secure
End-to-end security
Threat protection
Access control
Simple OAuth implementation for your APIs
PCI and HIPAA compliance

Monitor
Centralized control, decentralized development
Multi-tenant architecture
Billions of API calls, including large spikes

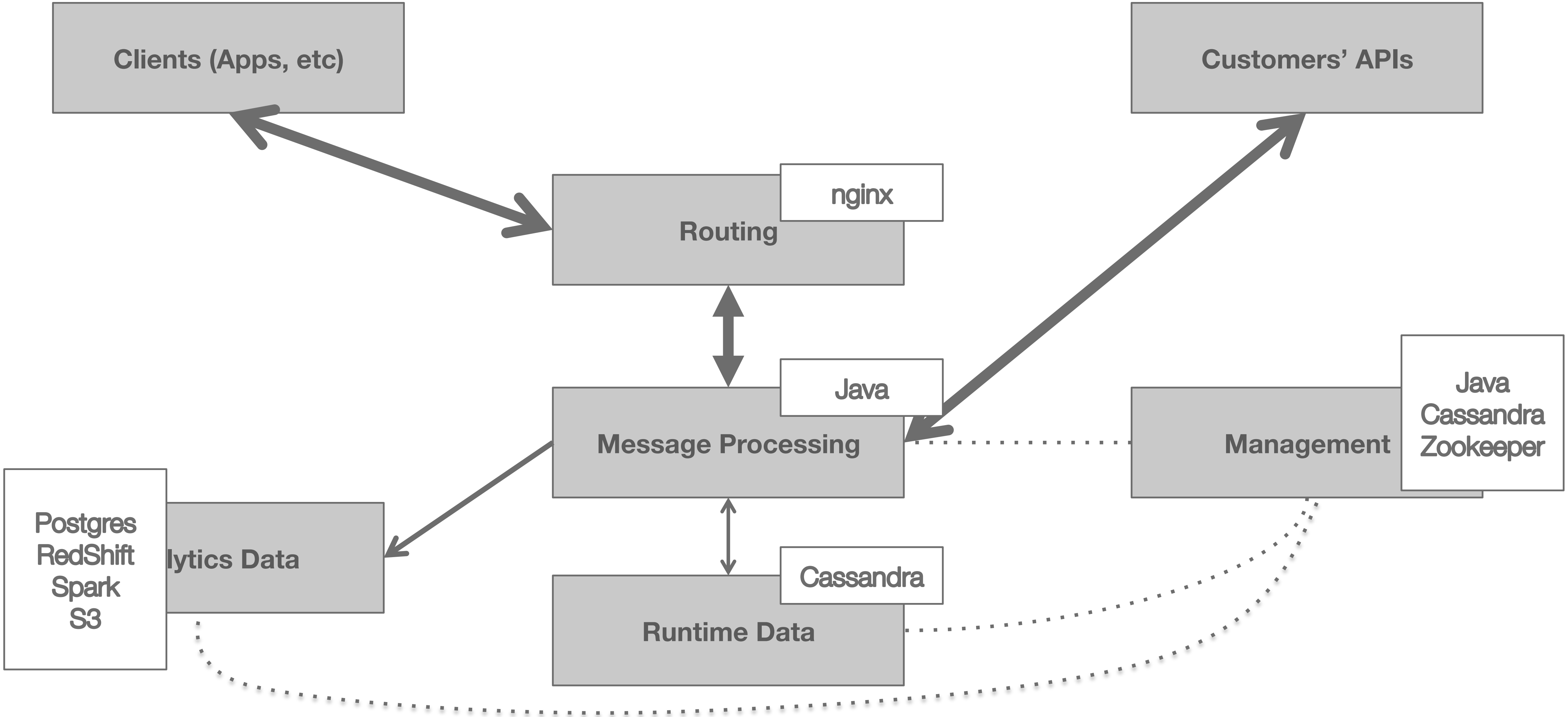
Publish
Turnkey developer portal

Scale
Self-service
State @ scale
Flexible deployment

What Does it Look Like?



What Does it Look Like?



Technical Challenges

Our Challenge

- What our customers expect:
  - **>99.99% availability** as defined by the number of transactions that complete successfully
  - **Geographically distributed** across data centers
  - In the Apigee Cloud or their own data centers
  - **No** maintenance windows
  - **No** regressions
  - Acceptable latency
  - All the features we have plus just one more ;-)

Our Basic Approach

All Things Distributed

Werner Vogels' weblog on building scalable and robust distributed systems.

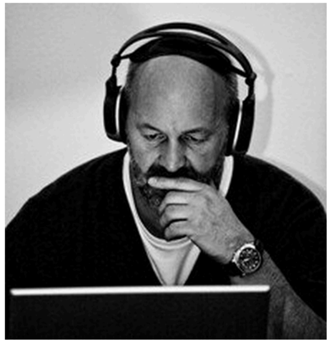
Eventually Consistent

By Werner Vogels on 19 December 2007 02:03 PM | Permalink | Comments (20)

I wrote a first version of this posting on consistency models in December 2007, but I was never happy with it as it was written in haste and the topic is important enough to receive a more thorough treatment. ACM Queue asked me to revise it for use in their magazine and I took the opportunity to improve the

… article in December 2008 under the tile Eventually Consistent - Revisted. - … stead of this one. I am leaving this one here for transparency/historical reasons … nts helped me improve the article. For which I am grateful

… a lot of discussion about the concept of *eventual consistency* in the context of



Contact Info

Werner Vogels
…TO - Amazon.com

…erner@allthingsdistributed.com

…ther places

…llow werner on twitter if you want to

Building on Quicksand

Pat Helland
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA
PHelland@Microsoft.com

Dave Campbell
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA
DavidC@Microsoft.com

ABSTRACT

Reliable systems have always been built out of unreliable components [1]. Early on, the reliable components were small such as mirrored disks or ECC (Error Correcting Codes) in core memory. These systems were designed such that failures of these small components were transparent to the application. Later, the size of the unreliable components grew larger and semantic challenges crept into the application when failures occurred.

Fault tolerant algorithms comprise a set of idempotent sub-algorithms. Between these idempotent sub-algorithms, state is sent across the failure boundaries of the unreliable components. The failure of an unreliable component can then be tolerated as a takeover by a backup, which uses the last known state and drives forward with a retry of the idempotent sub-algorithm. Classically, this has been done in a linear fashion (i.e. one step at a time).

As the granularity of the unreliable component grows (from a mirrored disk to a system to a data center), the latency to communicate with a backup becomes unpalatable. This leads to a more relaxed model for fault tolerance. The primary system will acknowledge the work request and its actions *without* waiting to ensure that the backup is notified of the work. This improves the

Keywords

Fault Tolerance, Eventual Consistency, Reconciliation, Loose Coupling, Transactions

1. Introduction

There is an interesting connection between fault tolerance, offlineable systems, and the need for application-based eventual consistency. As we attempt to run our large scale applications spread across many systems, we cannot afford the latency to wait for a backup system to remain in synch with the system actually performing the work. This causes the server systems to look increasingly like offlineable client applications in that they do not know the authoritative truth. In turn, these server-based applications are designed to record their intentions and allow the work to interleave and flow across the replicas. In a properly designed application, this results in system behavior that is acceptable to the business while being resilient to an increasing number of system failures.

This paper starts by examining the concepts of fault tolerance and posits an abstraction for thinking about fault tolerant systems. Next, section 3 examines how fault tolerant systems have

Life beyond Distributed Transactions: an Apostate's Opinion

Position Paper

Pat Helland
Amazon.Com
705 Fifth Ave South
Seattle, WA 98104
USA
PHelland@Amazon.com

The positions expressed in this paper are personal opinions and do not in any way reflect the positions of my employer Amazon.com.

ABSTRACT

Many decades of work have been invested in the area of distributed transactions including protocols such as 2PC, Paxos, and various

Instead, applications are built using different techniques which do not provide the same transactional guarantees but still meet the needs of their businesses.

This paper explores and names some of the practical approaches used in the implementations of large-scale mission-critical applications in a world which rejects distributed transactions. We discuss the management of fine-grained pieces of

apigee

Types of Data At Apigee

Type	How Many Records?	How Often do we Write?	Technology
System configuration	1000s	10s / minute	Zookeeper
Customer Proxy Deployments	100,000s	10s / minute	Zookeeper / C*
API Publishing Data (developers, apps, keys)	Millions	10s / second	C*
OAuth Tokens & metadata	Tens of millions	10,000s / second	C*
Counters / Quotas	Millions	10,000s / second	C*
Distributed Cache	Tens of millions	10,000s / second	C*
API Analytics Data	Billions	10,000s / second	Postgres / RedShift / S3

Challenge #1: Availability

- Goal: deliver 99.99% of API calls without introducing errors
- Measurement:
  - We need to measure every API call
  - Apply logic that looks at error from target as well as result that we delivered
  - Look at the numbers every week and drive the error rate down
- Result:
  - Steady improvement as long as we keep measuring.

apigee

Challenge #1: Counting*

- **What we need:**
- Application X is allowed to make 10,000 API calls per hour for free
  - Across geographies
  - Less than a 0.01% error rate
  - Minimal latency
- Application Y is allowed to make 1,000,000 API calls per hour because they paid
  - Warn them before they reach a million
  - Cut them off if they exceed it
  - Charge them accurately for each API call
- Control the tradeoff between accuracy and latency
  - We'd love to be able to talk rationally about this with customers

* That was a joke

Counting in Distributed Systems

- **What we can do:**
- Central system that holds all counters
  - Would be perfectly accurate, but obviously no
- Distributed consensus protocol across all servers
  - Too slow especially across geographies
- Eventually consistent counters
  - Yes! But how?
- Cassandra counters
  - Write availability in the presence of network partitions
  - But no guarantees about accuracy (see Jepsen)
  - Still too slow
- Cassandra counters plus local caching
  - Give us the best compromise today

apigee

Challenge #3: Detecting Abuse

- APIs are nice and open and easy to program
- That makes them easy to exploit
  - Travel APIs
  - Retail APIs
  - Other open APIs
- 80% of traffic on one retailer's API was from "bots"
  - Scraping prices, availability, etc.
- 56% of all web site traffic purportedly comes from bots

Detecting Bad Traffic

- Long-term batch analytics processing
  - Machine learning + data + heuristics
- For instance
  - U.S. Retailers don't have many customers in Romania
  - iPads tend not to reside inside Amazon Web Services data centers
  - Real people tend not to query product SKUs starting at "000000" and proceeding to "999999"
  - Real people don't check on100 rooms at the same hotel and never book
- Solution includes:
  - Batch processing to update bot scoring
  - Bloom filters at router layer
  - Lookup table and other processing for other traffic

Challenge #4: Management

- We are largely a management system
  - 1000s of new API proxies deployed per day to our cloud
  - Each one includes customer-specific processing rules, policies and code
  - API calls coming in for analytics queries, to change rate limits, set up developers, etc.
- Systems architects tend to give management short shrift
  - "It's OK if the management system fails as long as the API calls keep working"
- We try to architect management for the same SLA as everything else
  - So we use Cassandra and Zookeeper here too

Finally: Lessons from the Cloud

- Hardware fails. So what?
- Network fails. Bad but expected.
- Management layer fails. Big problem.
  - See history of AWS outages

Thanks

Apigee Edge is the work of many Apigeeks past and present. A few deserve special thanks:

Girish Karthik

Ravi Chandra

Ramesh Nethi

Scott Metzger (r.i.p.)

Shankar Ramaswamy

Anant Jhingran

Rajesh Jahdav

Sridhar Rajagopalan

Thank you