DZone Refcardz

CONTENTS INCLUDE:

Scalability & High Availability

By Eugene Ciurana

OVERVIEW

Scalability, High Availability, and Performance

The terms scalability, high availability, performance, and mission-critical can mean different things to different organizations, or to different departments within an organization. They are often interchanged and create confusion that results in poorly managed expectations, implementation delays, or unrealistic metrics. This Refcard provides you with the tools to define these terms so that your team can implement mission-critical systems with well-understood performance goals.

Scalability

It's the property of a system or application to handle bigger amounts of work, or to be easily expanded, in response to increased demand for network, processing, database access or file system resources.

Horizontal scalability

A system scales horizontally, or out, when it's expanded by adding new nodes with identical functionality to existing ones, redistributing the load among all of them. SOA systems and web servers scale out by adding more servers to a load-balanced network so that incoming requests may be distributed among all of them. Cluster is a common term for describing a scaled out processing system.

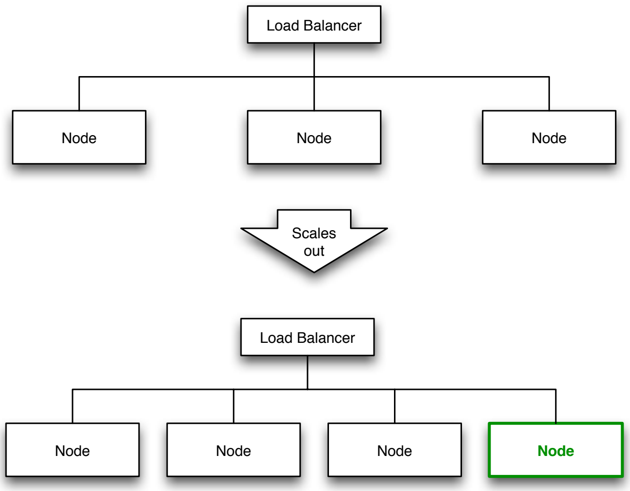


Figure 1: Clustering

Vertical scalability

A system scales vertically, or up, when it's expanded by adding processing, main memory, storage, or network interfaces to a node to satisfy more requests per system. Hosting services companies scale up by increasing the number of processors or

Scalability, continued

the amount of main memory to host more virtual servers in the same hardware.

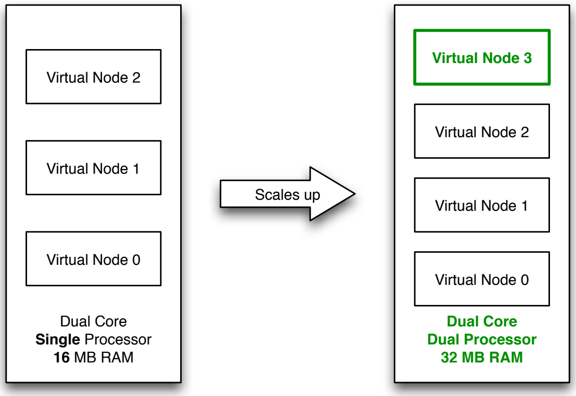


Figure 2: Virtualization

High Availability

Availability describes how well a system provides useful resources over a set period of time. High availability guarantees an absolute degree of functional continuity within a time window expressed as the relationship between uptime and downtime.

A = 100 – (100*D/U), D ::= unplanned downtime, U ::= uptime; D, U expressed in minutes

Uptime and availability don't mean the same thing. A system may be up for a complete measuring period, but may be unavailable due to network outages or downtime in related support systems. Downtime and unavailability are synonymous.

High Availability, continued

Measuring Availability

Vendors define availability as a given number of "nines" like in Table 1, which also describes the number of minutes or seconds of estimated downtime in relation to the number of minutes in a 365-day year, or 525,600, making U a constant for their marketing purposes.

Availability %	Downtime in Minutes	Downtime per Year	Vendor Jargon
90	52,560.00	36.5 days	one nine
99	5,256.00	4 days	two nines
99.9	525.60	8.8 hours	three nines
99.99	52.56	53 minutes	four nines
99.999	5.26	5.3 minutes	five nines
99.9999	0.53	32 seconds	six nines

Table 1: Availability as a Percentage of Total Yearly Uptime

Analysis

High availability depends on the expected uptime defined for system requirements; don't be misled by vendor figures. The meaning of having a highly available system and its measurable uptime are a direct function of a Service Level Agreement. Availability goes up when factoring planned downtime, such as a monthly 8-hour maintenance window.

The cost of each additional nine of availability can grow exponentially. Availability is a function of scaling the systems up or out and implementing system, network, and storage redundancy.

Service Level Agreement (SLA)

SLAs are the negotiated terms that outline the obligations of the two parties involved in delivering and using a system, like:

- System type (virtual or dedicated servers, shared hosting)
- Levels of availability
  - Minimum
  - Target
- Uptime
  - Network
  - Power
  - Maintenance windows
- Serviceability
- Performance and metrics
- Billing

SLAs can bind obligations between two internal organizations (e.g. the IT and e-commerce departments), or between the organization and an outsourced services provider. The SLA establishes the metrics for evaluating the system performance, and provides the definitions for availability and the scalability targets. It makes no sense to talk about any of these topics unless an SLA is being drawn or one already exists.

IMPLEMENTING SCALABLE SYSTEMS

SLAs determine whether systems must scale up or out. They also drive the growth timeline. A stock trading system must scale in real-time within minimum and maximum availability levels. An e-commerce system, in contrast, may scale in during the "slow" months of the year, and scale out during the retail holiday season to satisfy much larger demand.

Implementing Scalable Systems, continued

Load Balancing

Load balancing is a technique for minimizing response time and maximizing throughput by spreading requests among two or more resources. Load balancers may be implemented in dedicated hardware devices, or in software. Figure 3 shows how load-balanced systems appear to the resource consumers as a single resource exposed through a well-known address. The load balancer is responsible for routing requests to available systems based on a scheduling rule.

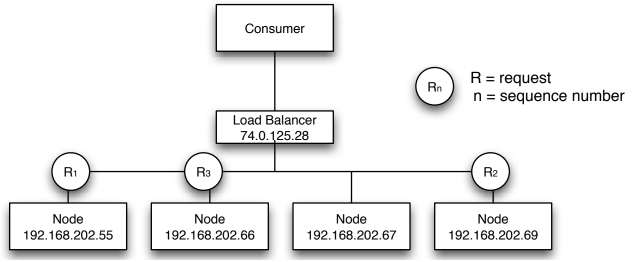


Figure 3: Load Balancer

Scheduling rules are algorithms for determining which server must service a request. Web applications and services are balanced by following round robin scheduling rules. Caching pools are balanced by applying frequency rules and expiration algorithms. Applications where stateless requests arrive with a uniform probability for any number of servers may use a pseudo-random scheduler. Applications like music stores, where some content is statistically more popular, may use asymmetric load balancers to shift the larger number popular requests to higher performance systems, serving the rest of the requests from less powerful systems or clusters.

Persistent Load Balancers

Stateful applications require persistent or sticky load balancing, where a consumer is guaranteed to maintain a session with a specific server from the pool. Figure 4 shows a sticky balancer that maintains sessions from multiple clients. Figure 5 shows how the cluster maintains sessions by sharing data using a database.

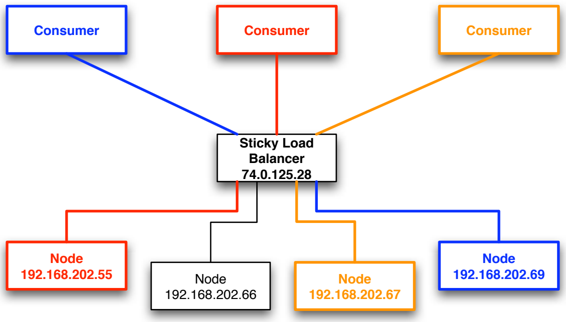


Figure 4: Sticky Load Balancer

Common Features of a Load Balancer

Asymmetric load distribution – assigns some servers to handle a bigger load than others

- Content filtering – inbound or outbound
- Distributed Denial of Service (DDoS) attack protection
- Firewalling
- Payload switching – send requests to different servers based on URI, port, and/or protocol
- Priority activation – adds standing by servers to the pool

Load Balancing, continued

- Rate shaping – ability to give different priority to different traffic
- Scripting – reduces human interaction by implementing programming rules or actions
- SSL offloading – hardware-assisted encryption frees web server resources
- TCP buffering and offloading – throttle requests to servers in the pool

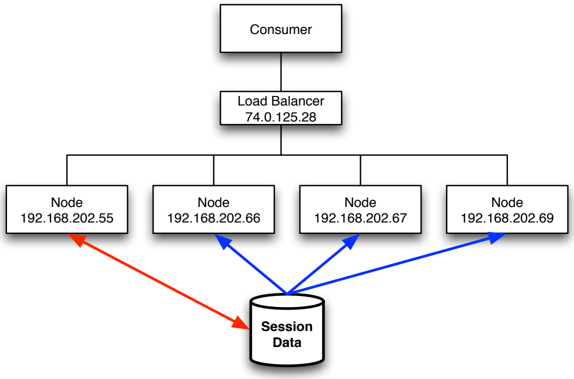


Figure 5: Database Sessions

CACHING STRATEGIES

Stateful load balancing techniques require data sharing among the service providers. Caching is a technique for sharing data among multiple consumers or servers that are expensive to either compute or fetch. Data are stored and retrieved in a subsystem that provides quick access to a copy of the frequently accessed data.

Caches are implemented as an indexed table where a unique key is used for referencing some datum. Consumers access data by checking (hitting) the cache first and retrieving the datum from it. If it's not there (cache miss), then the costlier retrieval operation takes place and the consumer or a subsystem inserts the datum to the cache.

Write Policy

The cache may become stale if the backing store changes without updating the cache. A write policy for the cache defines how cached data are refreshed. Some common write policies include:

- **Write-through**: every write to the cache follows a synchronous write to the backing store
- **Write-behind**: updated entries are marked in the cache table as dirty and it's updated only when a dirty datum is requested.
- **No-write allocation**: only read requests are cached under the assumption that the data won't change over time but it's expensive to retrieve

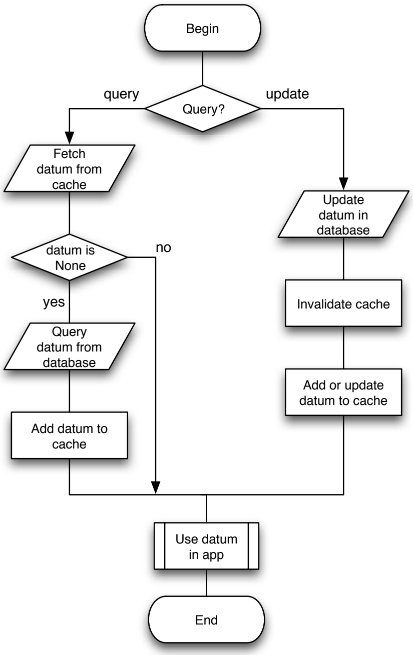


Figure 6: Caching Usage Pattern

Caching Strategies, continued

Application Caching

- Implicit caching happens when there is little or no programmer participation in implementing the caching. The program executes queries and updates using its native API and the caching layer automatically caches the requests independently of the application. Example: Terracotta (http://www.terracotta.org).

- Explicit caching happens when the programmer participates in implementing the caching API and may also implement the caching policies. The program must import the caching API into its flow in order to use it. Examples: memcached (http://www.danga.com/memcached) and Oracle Coherence (http://coherence.oracle.com).

In general, implicit caching systems are specific to a platform or language. Terracotta, for example, only works with Java and JVM-hosted languages like Groovy. Explicit caching systems may be used with many programming languages and across multiple platforms at the same time. memcached works with every major programming language, and Coherence works with Java, .Net, and native C++ applications.

Web Caching

Web caching is used for storing documents or portions of documents ('particles') to reduce server load, bandwidth usage and lag for web applications. Web caching can exist on the browser (user cache) or on the server, the topic of this section. Web caches are invisible to the client may be classified in any of these categories:

- **Web accelerators:** they operate on behalf of the server of origin. Used for expediting access to heavy resources, like media files. Content distribution networks (CDNs) are an example of web acceleration caches; Akamai, Amazon S3, Nirvanix are examples of this technology.

- **Proxy caches:** they serve requests to a group of clients that may all have access to the same resources. They can be used for content filtering and for reducing bandwidth usage. Squid, Apache, ISA server are examples of this technology.

Distributed Caching

Caching techniques can be implemented across multiple systems that serve requests for multiple consumers and from multiple resources. These are known as distributed caches, like the setup in Figure 7. Akamai is an example of a distributed web cache. memcached is an example of a distributed application cache.

Caching Strategies, continued

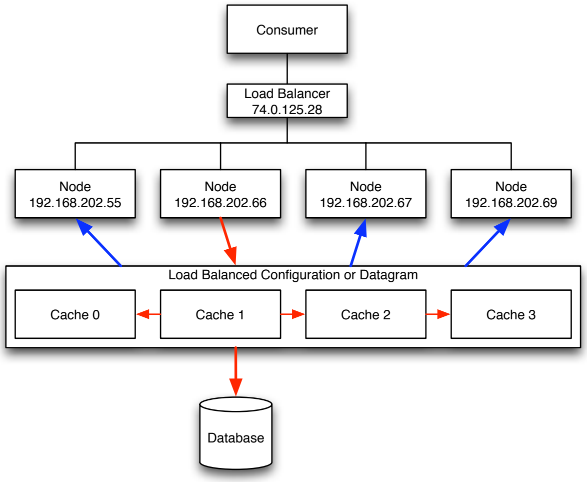


Figure 7: Distributed Cache

CLUSTERING

A cluster is a group of computer systems that work together to form what appears to the user as a single system. Clusters are deployed to improve services availability or to increase computational or data manipulation performance. In terms of equivalent computing power, a cluster is more cost-effective than a monolithic system with the same performance characteristics.

The systems in a cluster are interconnected over high-speed local area networks like gigabit Ethernet, fiber distributed data interface (FDDI), Infiniband, Myrinet, or other technologies.

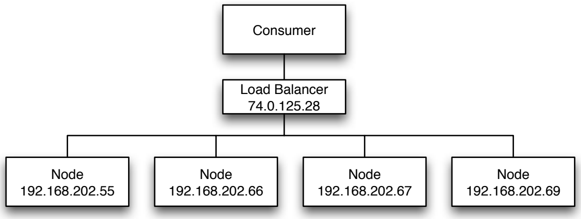


Figure 8: Load Balancing Cluster

Load-Balancing Cluster (Active/Active): Distribute the load among multiple back-end, redundant nodes. All nodes in the cluster offer full-service capabilities to the consumers and are active at the same time.

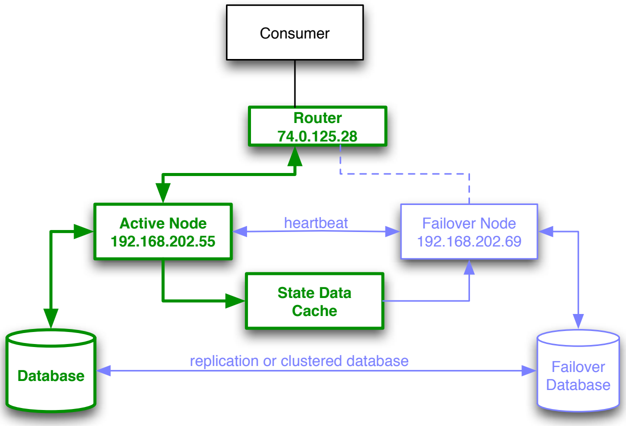


Figure 9: High Availability Cluster

Clustering, continued

High Availability Cluster(Active/Passive): Improve services availability by providing uninterrupted service through redundant nodes that eliminate single points of failure. High availability clusters require two nodes at a minimum, a "heartbeat" to detect that all nodes are ready, and a routing mechanism that will automatically switch traffic if the main node fails.

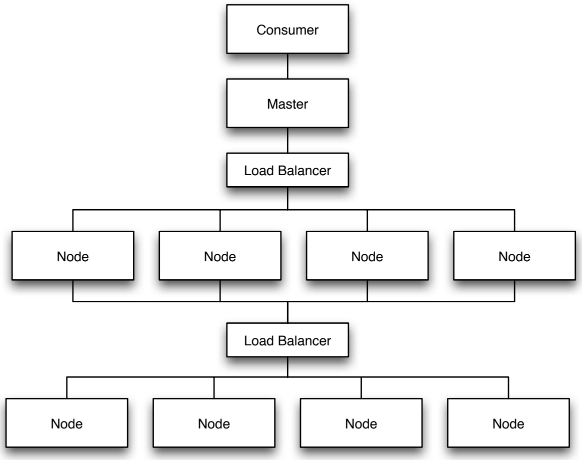


Figure 10: Grid

Grid: Process workloads defined as independent jobs that don't require data sharing among processes. Storage or network may be shared across all nodes of the grid, but intermediate results have no bearing on other jobs progress or on other nodes in the grid, such as a Cloudera Map Reduce cluster (http://www.cloudera.com).

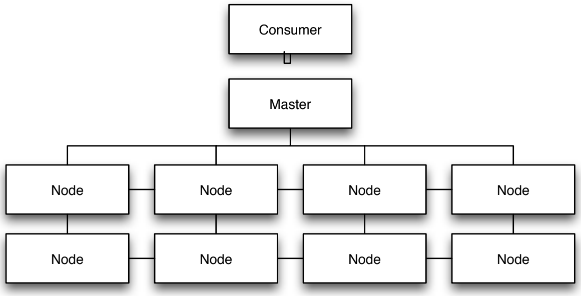


Figure 11: Computational Clusters

Computational Clusters: Exeute processes that require raw computational power instead of executing transactional operations like web or database clusters. The nodes are tightly coupled, homogeneous, and in close physical proximity. They often replace supercomputers.

REDUNDANCY AND FAULT TOLERANCE

Redundant system design depends on the expectation that any system component failure is independent of failure in the other components.

Fault tolerant systems continue to operate in the event of component or subsystem failure; throughput may decrease but overall system availability remains constant. Faults in hardware or software are handled through component redundancy. Fault tolerance requirements are derived from SLAs. The implementation depends on the hardware and software components, and on the rules by which they interact.

Redundancy and Fault Tolerance, continued

Fault Tolerance SLA Requirements

- **No single point of failure** – redundant components ensure continuous operation and allow repairs without disruption of service
- **Fault isolation** – problem detection must pinpoint the specific faulty component
- **Fault propagation containment** – faults in one component must not cascade to others
- **Reversion mode** – set the system back to a known state

Redundant clustered systems can provide higher availability, better throughput, and fault tolerance. The A/A cluster in Figure 12 provides uninterrupted service for a scalable, stateless application.

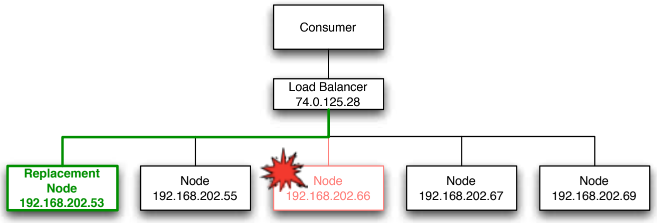


Figure 12: A/A Full Tolerance and Recovery

Some stateful applications may only scale up; the A/P cluster in Figure 13 provides uninterrupted service and disaster recovery for such an application. A/A configurations provide failure transparency. A/P configurations may provide failure transparency at a much higher cost because automatic failure detection and reconfiguration are implemented through a feedback control system, which is more expensive and trickier to implement.

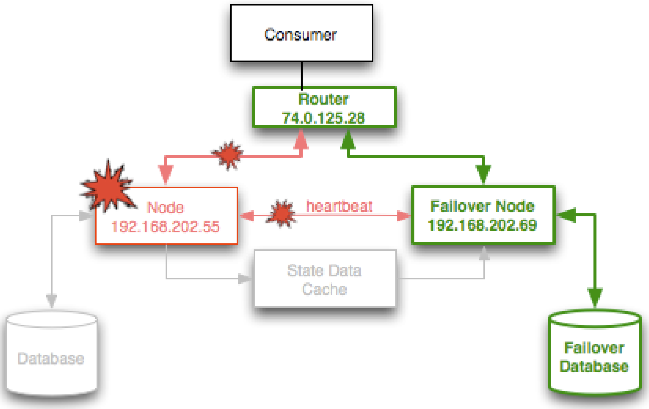


Figure 13: A/P Fault Tolerance and Recovery

Enterprise systems most commonly implement A/P fault tolerance and recovery through fault transparency by diverting services to the passive system and bringing it on-line as soon as possible. Robotics and life-critical systems may implement probabilistic, linear model, fault hiding, and optimization control systems instead.

Cloud Computing

Cloud computing describes applications running on distributed, computing resources owned and operated by a third-party.

End-user apps are the most common examples. They utilize the Software as a Service (SaaS) and Platform as a Service (PaaS) computing models.

Cloud Computing, continued

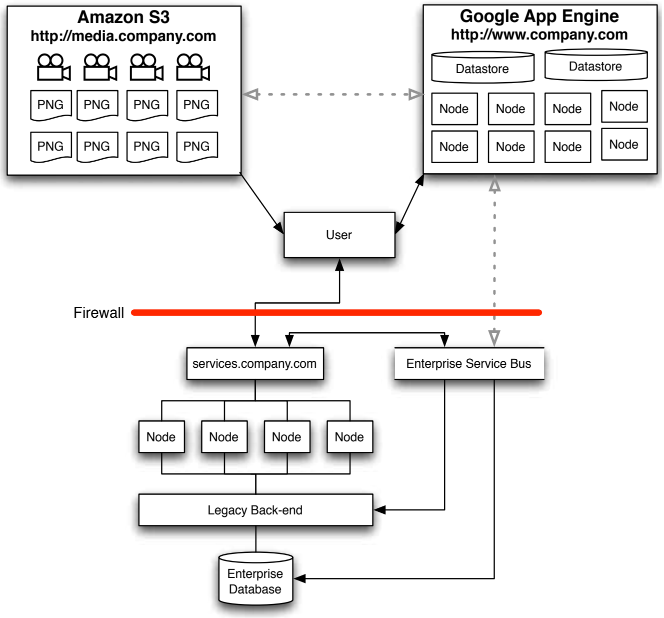


Figure 14: Cloud Computing Configuration

Cloud Services Types

- **Web services** – Salesforce com, USPS, Google Maps
- **Service platforms** – Google App Engine, Amazon Web Services (EC2, S3, Cloud Front), Nirvanix, Akamai, MuleSource

Fault Detection Methods

Fault detection methods must provide enough information to isolate the fault and execute automatic or assisted failover action. Some of the most common fault detection methods include:

- **Built-in Diagnostics**
- **Protocol Sniffers**
- **Sanity Checks**
- **Watchdog Checks**

Criticality is defined as the number of consecutive faults reported by two or more detection mechanisms over a fixed time period. A fault detection mechanism is useless if it reports every single glitch (noise) or if it fails to report a real fault over a number of monitoring periods.

SYSTEM PERFORMANCE

Performance refers to the system throughput under a particular workload for a defined period of time. Performance testing validates implementation decisions about the system throughput, scalability, reliability, and resource usage. Performance engineers work with the development and deployment teams to ensure that the system's non-functional requirements like SLAs are implemented as part of the system development lifecycle. System performance encompasses hardware, software, and networking optimizations.

Hot Tip

Performance testing efforts must begin at the same time as the development project and continue through deployment

System Performance, continued

The performance engineer's objective is to detect bottlenecks early and to collaborate with the development and deployment teams on eliminating them.

System Performance Tests

Performance specifications are documented along with the SLA and with the system design. Performance troubleshooting includes these types of testing:

- **Endurance testing**- identifies resource leaks under the continuous, expected load.
- **Load testing** – determines the system behavior under a specific load.
- **Spike testing** – shows how the system operates in response to dramatic changes in load.
- **Stress testing** – identifies the breaking point for the application under dramatic load changes for extended periods of time.

System Performance, continued

Software Testing Tools

There are many software performance testing tools in the market. Some of the best are released as open-source software. A comprehensive list of those is available from:

http://www.opensourcetesting.org/performance.php

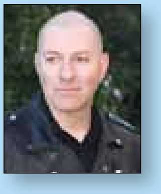
These include Java, native, PHP, .Net, and other languages and platforms.

Staying Current

Do you want to know about specific projects and cases where scalability, high availability, and performance are the hot topic? Join the scalability newsletter:

http://eugeneciurana.com/scalablesystems

ABOUT THE AUTHOR



Eugene Ciurana

Eugene Ciurana is an open-source evangelist who specializes in the design and implementation of mission-critical, high-availability large scale systems. As Director of Systems Infrastructure, he and his team designed and built a 100% SOA and cloud system that enables millions of Internet-ready educational and handheld products and services. As chief liaison between Walmart.com Global and the ISD Technology Council, he led the official adoption of Linux and other open-source technologies at Walmart Stores Information Systems Division. He's also designed high performance systems for major financial institutions and many Fortune 100 companies in the United States and Europe.

Publications

- *Developing with the Google App Engine*
- *Best Of Breed: Building High Quality Systems, Within Budget, On Time, and Without Nonsense*
- *The Tesla Testament: A Thriller*

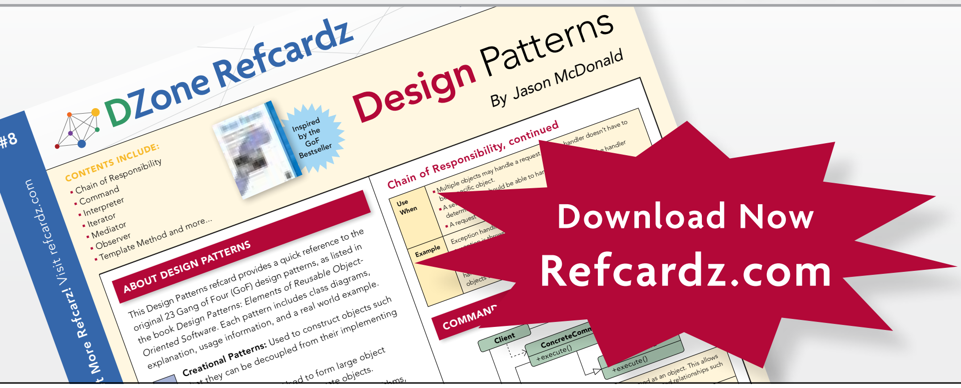
Web site

http://eugeneciurana.com

RECOMMENDED BOOK

Developing with Google App Engine introduces Google App Engine, a platform that provides developers and users with the infrastructure that Google itself uses for developing and deploying massively scalable applications. Using Python as the primary programming tool, Developing with Google App Engine makes it easy to implement scalability and high performance features like distributed databases, clustering, stateless applications, and sophisticated data caching.

BUY NOW
books.dzone.com/books/google-app-engine



DZone

DZone communities deliver over 4 million pages each month to more than 2 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. "DZone is a developer's dream," says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-46-2
ISBN-10: 1-934238-46-5
50795
9 781934 238462
$7.95

Reference: Developing with Google App Engine, Eugene Ciurana, APress, 2009

Version 1.0