



Application Architecture Patterns in Azure

Microsoft
Partner

Silver Learning
Gold Cloud Platform



Educate

Advise

Implement

Manage

Agenda

- Design Pattern Resources
- Performance Patterns
- Resiliency Patterns
- Scalability Patterns
- Data Patterns

Microsoft Patterns & Practices

- Engineering focused group at Microsoft
- Collects real-world scenarios from customers
- Engineers solutions using best-practices
- Analyze trends and services used by the community
- Shares findings using
 - GitHub
 - Whitepapers
 - Conference Sessions
 - <http://docs.microsoft.com>

Cloud Design Patterns

- The Cloud Design Patterns is a collection developer-oriented prescriptive architecture guidance
- Includes topics and patterns that help you design your cloud solutions
- Patterns are platform and framework-agnostic
- Examples are provided in the context of Azure and C#

Patterns & Practices on GitHub

Microsoft Patterns & Practices shares much of their documentation, projects and findings today on GitHub:

<https://github.com/mspnp>


For example, the Microservices Reference Implementation shares best practices when designing a microservices solution running on Azure using Kubernetes:

<https://github.com/mspnp/microservices-reference-implementation>


Azure Architecture Center

Landing page for reference architectures, patterns and guidance for solutions on the Azure Platform


Azure Architecture Center




Azure Application Architecture Guide
A guide to designing scalable, resilient, and highly available applications, based on proven practices that we have learned from customer engagements.




Reference Architectures
A set of recommended architectures for Azure. Each architecture includes best practices, prescriptive steps, and a deployable solution.




Cloud Design Patterns
Design patterns for developers and solution architects. Each pattern describes a problem, a pattern that addresses the problem, and an example based on Azure.




Best Practices for Cloud Applications
Best practices for cloud applications, covering aspects such as auto-scaling, caching, data partitioning, API design, and others.




Building microservices on Azure with Kubernetes
How to design, build, and operate a microservices architecture on Azure, using Kubernetes as a container orchestrator.




Design Review Checklists
Checklists to assist developers and solution architects during the design process.



Azure for AWS Professionals
Leverage your AWS experiences in Microsoft Azure.



Designing for Resiliency
Learn how to design resilient applications for Azure.

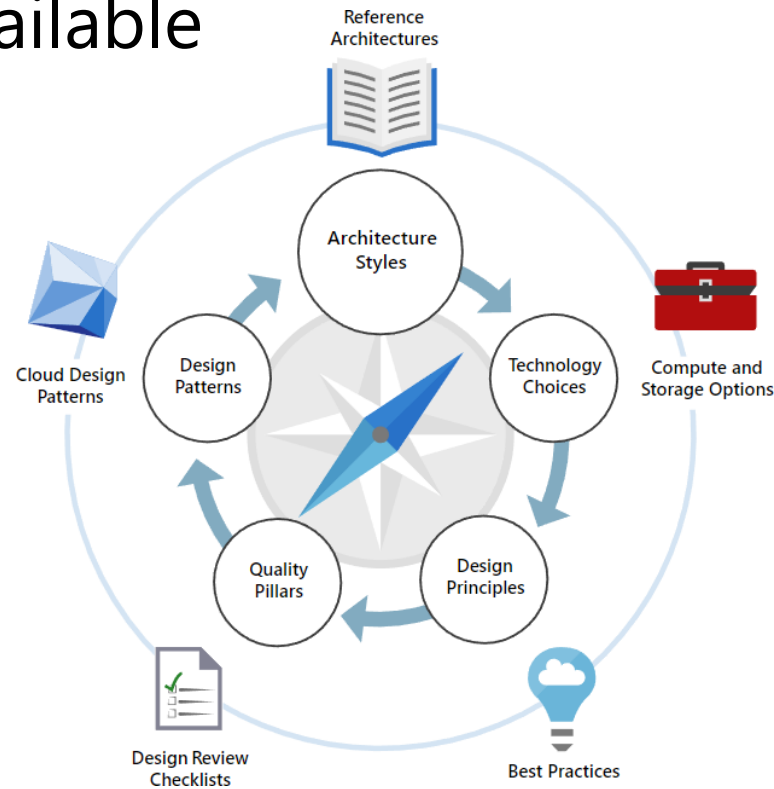


Azure Building Blocks
Simplify deployment of Azure resources. With a single settings file, deploy complex architectures in Azure.

<https://docs.microsoft.com/azure/architecture/>

Azure Architecture Center Guide

The Architecture Center features an all-up guide to creating solutions that are scalable, resilient and highly available



<https://docs.microsoft.com/azure/architecture/guide/>

Performance Patterns

- Stateless Applications
- The Valet Key Pattern

Stateless Applications

- When designing web applications, split your business processes into Partitioning Workloads
- Partitioning Workloads:
 - Can be handled in modular websites, cloud services, or virtual machines
- Provides the ability to scale the different components of your application in isolation

Partitioning Workloads

- Example Photo Sharing Application

Virtual Machine

Web Front-End

SignalR Hub

Image Processor

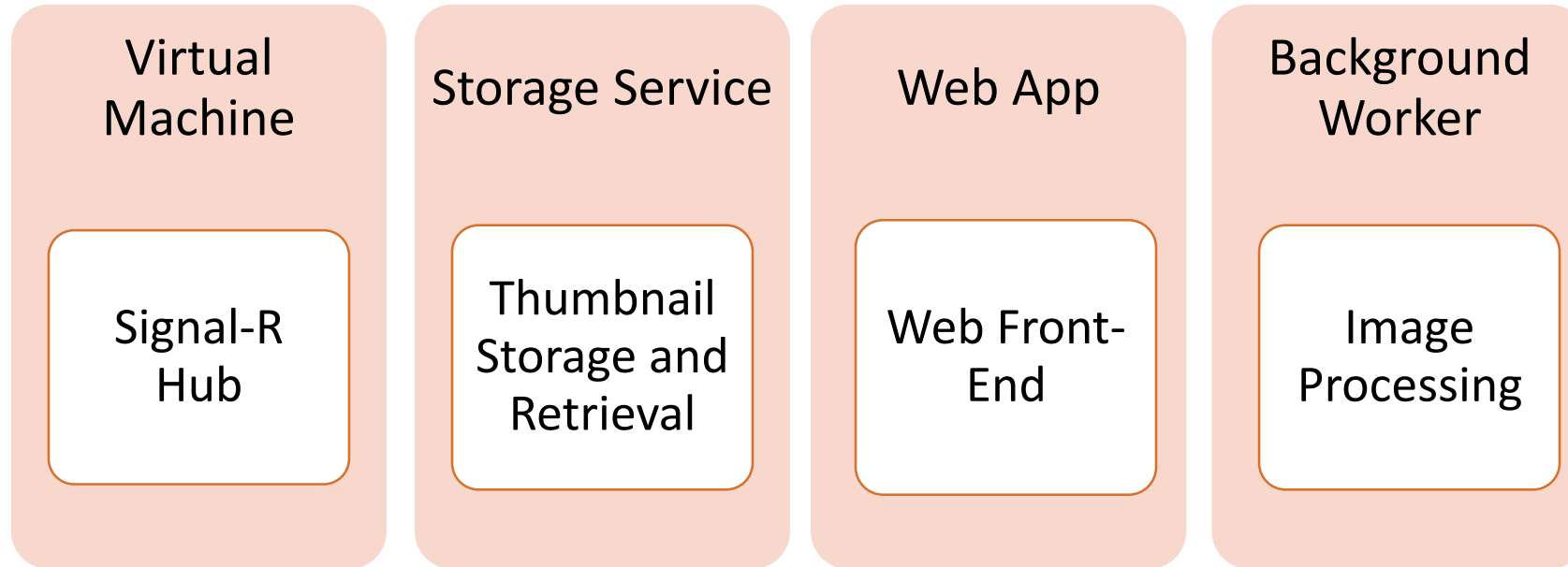
Thumbnail Storage

Partitioning Workloads

- Example Photo Sharing Application
 - The Web Front-End shows thumbnails of images users have uploaded today
 - The application code takes an image that a user uploads, processes it into a thumbnail and then stores the thumbnail
 - The SignalR hub notifies the client web browser that the image is finished processing

Partitioning Workloads

- Example Photo Sharing Application

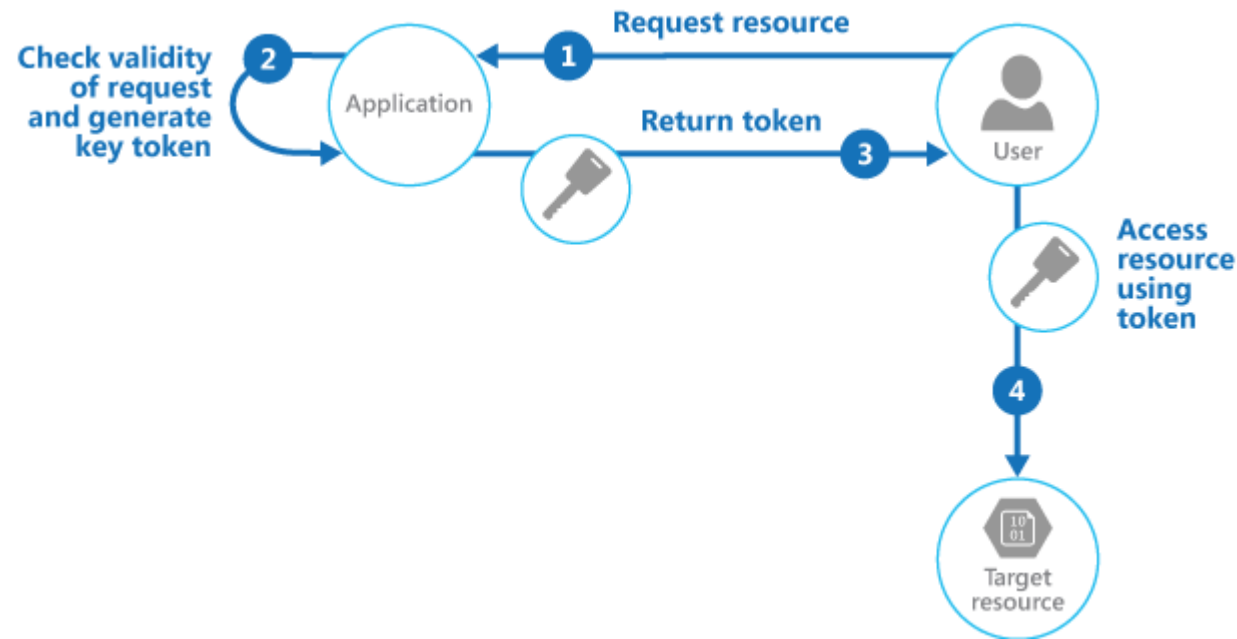


- Each component of the application can be scaled in isolation to meet its specific demand

The Valet Key Pattern

- If your application access a resource on behalf of your clients, your servers take on additional load
- You do not want to make your resources publicly available so you are typically forced to have your application validate a client
- The Valet Key pattern dictates that your application simply validates the client and then returns a token to the client. The client can then retrieve the resource directly using its own hardware and bandwidth

The Valet Key Pattern



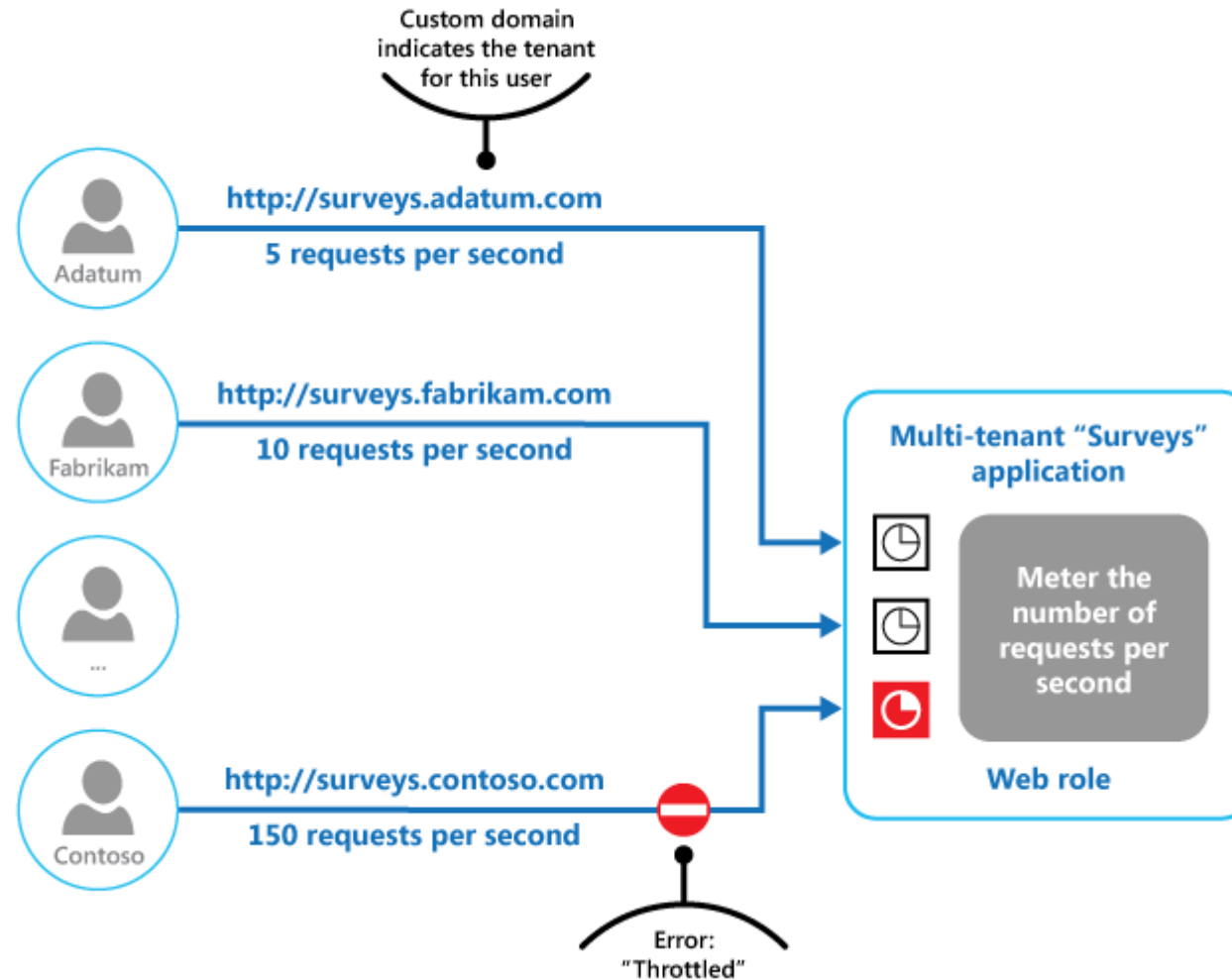
- The client requests a resource from your application
- Your application validates the client and then returns an access token
- The client then directly accesses the resource by using the provided token

Throttling Pattern

- The Problem:
 - Load on a cloud application varies:
 - By Time
 - By Quantity of Users
 - By Specific User Activities
 - By Performance of Actual Underlying Hardware
- Handling load correctly for one user/client may “starve” other clients of resources
- Solutions must be able to handle sudden bursts of usage

Throttling Pattern

- Solves problem by creating a “ceiling” on usage



Resiliency Patterns

- Transient Errors
- The Retry Pattern
- Queues

Transient Errors

- Transient faults can occur because of a temporary condition
- Service is unavailable
- Network connectivity issue
- Service is under heavy load
- Retrying your request can resolve temporary issues that normally would crash an application
- You can retry using different strategies
- Retry after a fixed time period
- Exponentially wait longer to retry
- Retry in timed increments

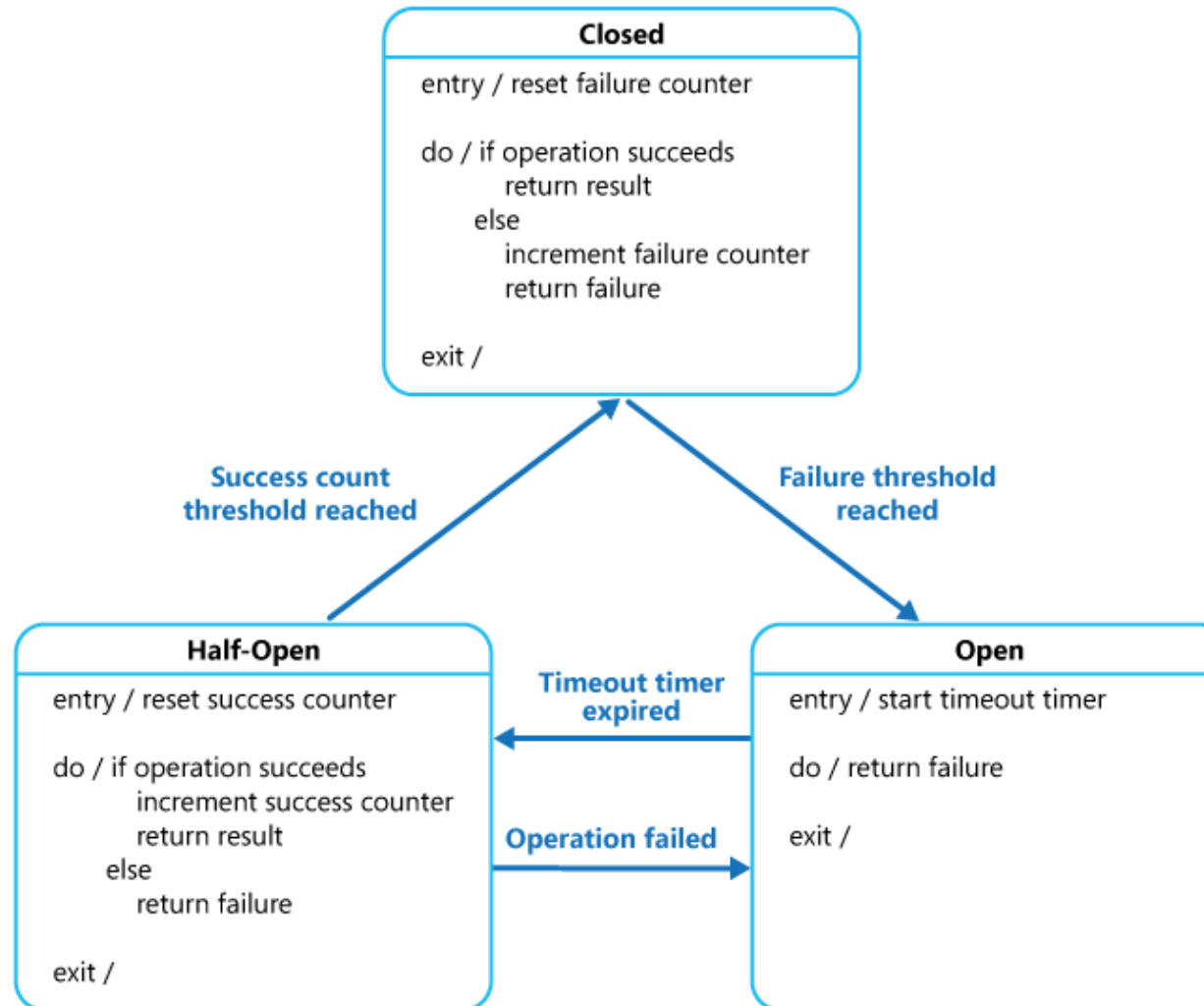
Transient Fault Handling

- The Transient Fault Handling application block is a part of the Enterprise Library
- The Enterprise library contains a lot of code that is necessary to implement the pattern and the retry strategies
- Retry strategies are prebuilt for common scenarios including accessing Azure services
- You can build custom strategies and extend the library for the unique needs of your application

Circuit Breaker Pattern

- Prevents applications from repeatedly retrying an operation that will most like fail
- Acts as a proxy between application and service
- Prevents waste of CPU cycles on long-lasting faults

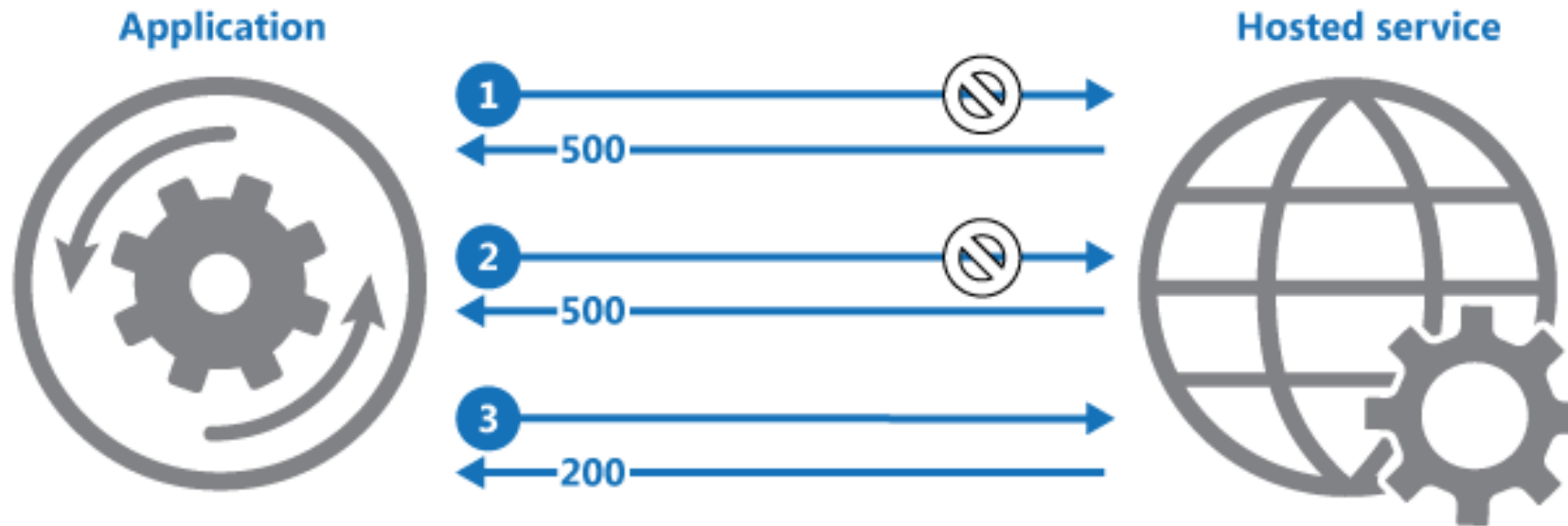
Circuit Breaker Pattern



The Retry Pattern

- The Retry pattern is designed to handle temporary failures
- Failures are assumed to be transient until they exceed the retry policy
- The Transient Fault Handling Block is an example of a library that is designed to implement the Retry pattern (and more)
- Entity Framework provides a built-in retry policy implementation
 - Implemented in version 6.0

The Retry Pattern



- The application sends a request to a hosted service
- The hosted service responds with a HTTP 500 (Internal Server Error) code
- The application retries until it exceeds the retry count for its policy or is met with an acceptable status code such as HTTP 200 (OK)

Queues

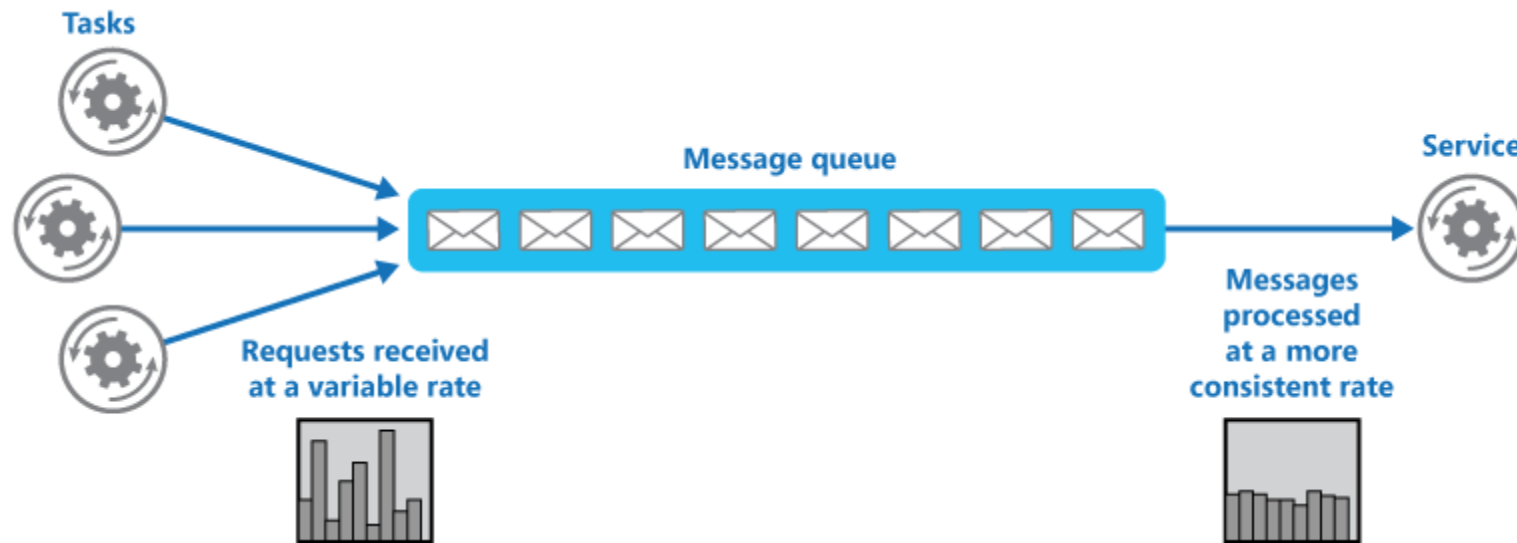
- A modular web application can behave like a monolithic application if each component relies on a direct two-way communication with a persistent connection
- Persistent queue messages allow your application to handle requests if one of your application components fail or is temporary unavailable
- An external queue allows your application to audit requests or measure your load without adding any overhead to the code of your primary application

Queues

- Queues can be used to communicate in a disconnected manner between the different components of your application
 - If an instance of your application module fails, another instance can process the queue messages
 - Messages that fail to be processed can either be reinserted into the queue or not marked as complete by the client module
 - Messages can be measured or audited in isolation from your application
- Queues become very important when dealing with background processing (Worker roles and WebJobs)

Queue-Based Load Leveling Pattern

- Use a queue to act as a “buffer” between requestor generators and request services



- Queue decouples the tasks from the service
- Service can work at it's own pace

Scalability Patterns

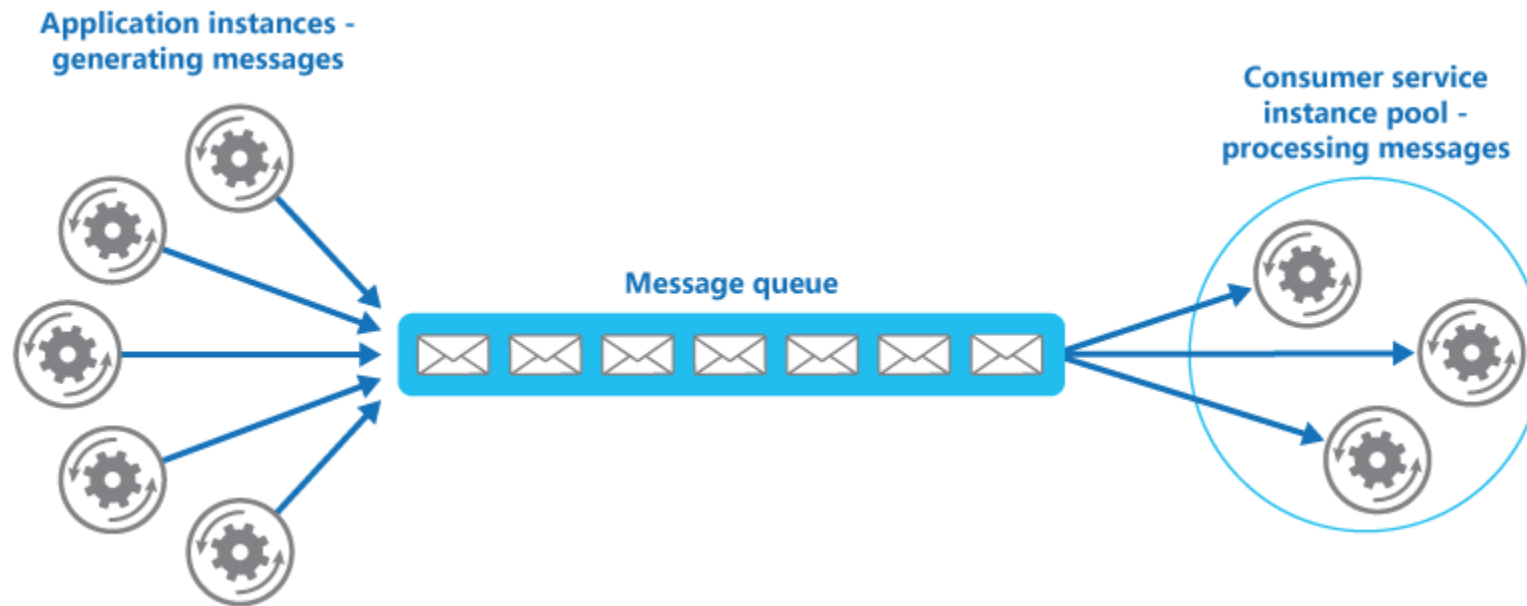
- Asynchronous Messaging
- Cached Data Consistency
- Load Balancing

Asynchronous Messaging

- With synchronous messaging, the service processing some logic must run immediately when requested
 - This becomes a problem with large, varying quantities of request
 - This can prevent an application from scaling both up or down
 - Scaling down can drop requests “in flight”

Competing Consumers Pattern

- Application instances (producers) generate messages to be placed in the queue
- Service instances (consumers) poll the queue to see if any messages are waiting to be processed
- The service instance that receives the message will process the message and then flag the message as processed in the queue
- If the service instance fails to process the message, the queue will eventually make the message available to other instances after a period of time

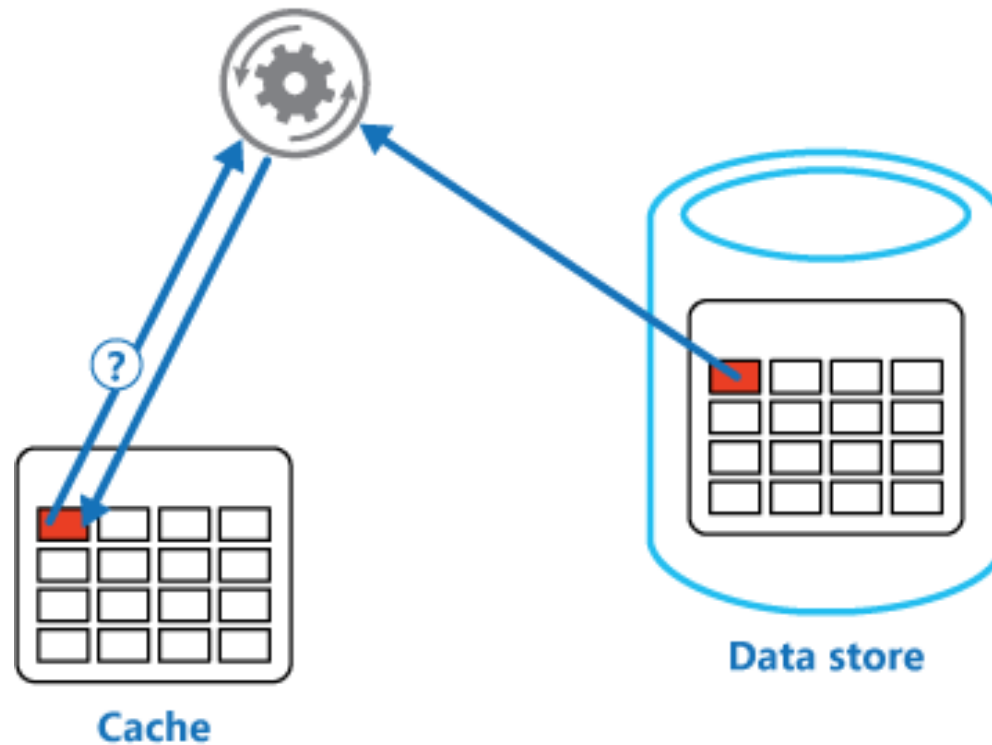


Cached Data Consistency

- Cache data can quickly become stale
- Example:
 - Application caches a list of the 10 latest records
 - Whenever a new record is added, the list is officially stale
 - Do you re-query the database constantly?
 - Do you skip using a cache to have real-time data?

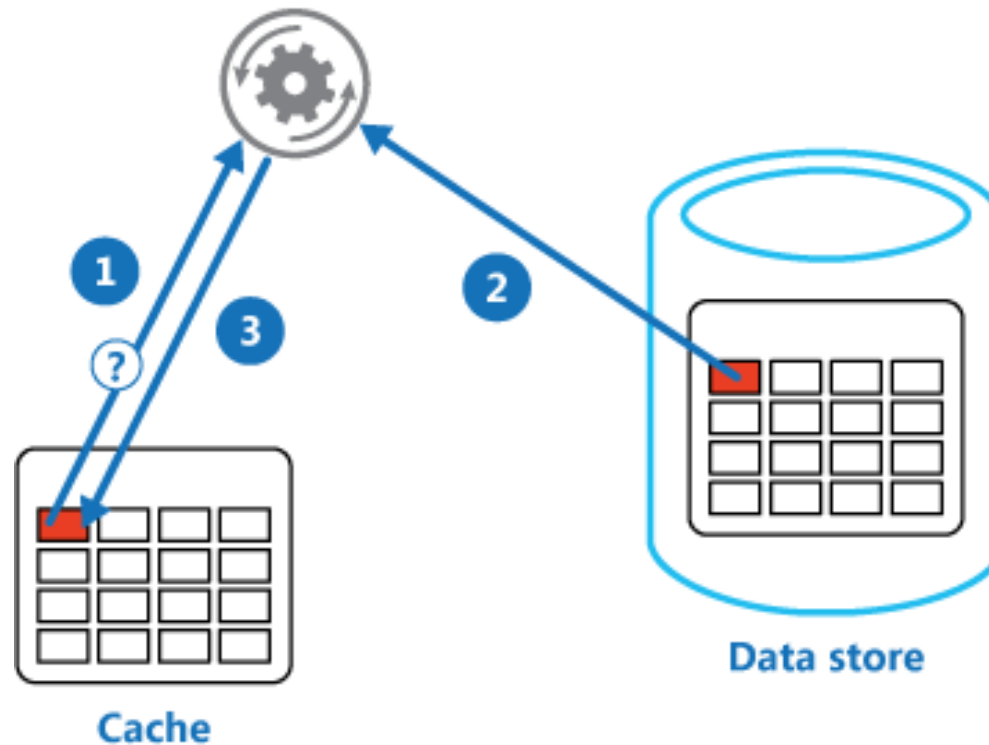
Cache-Aside Pattern

- Treats cache as a read/write store
- Ensures synced data between cache and data store



Cache-Aside Pattern

1. Determine whether the item is currently stored in cache
2. If the item is not currently stored in cache, retrieve the item from the source data store
3. Store a copy of the item in the cache

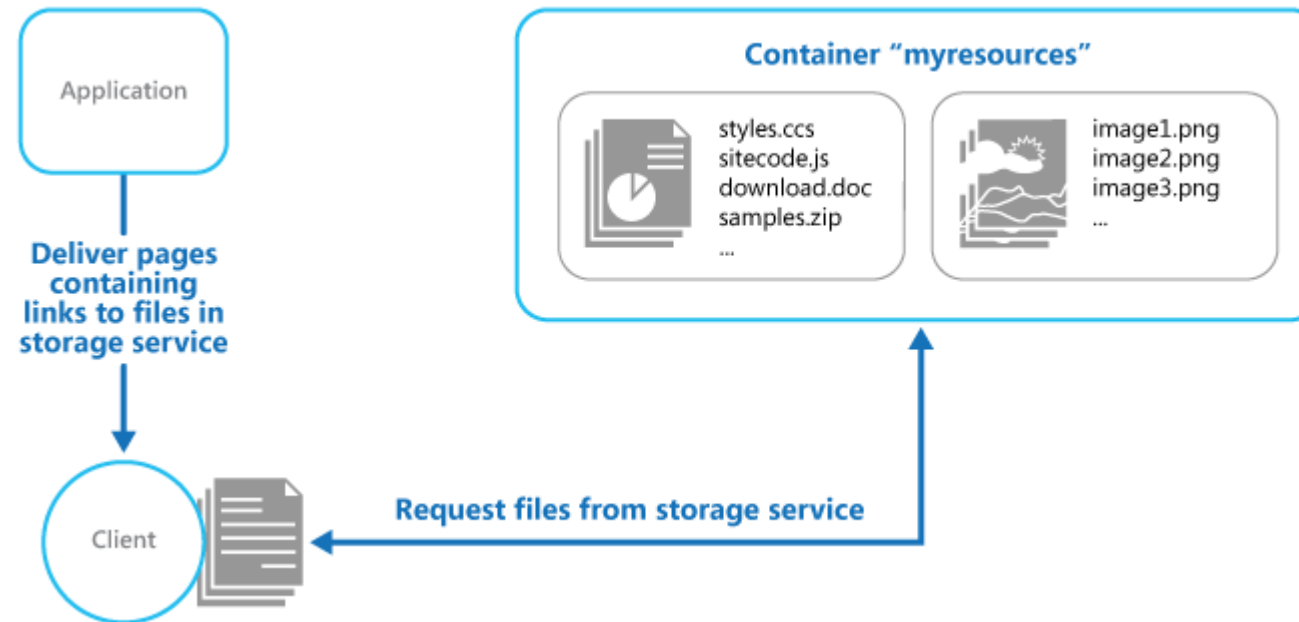


Static Content

- Static content is often hosted in a CDN to deliver content directly to clients in the most efficient manner possible
- Web servers can serve static content, but they are often not the right choice
- Developers often serve multiple CSS, JS and HTML files as part of a single web page “visit”
- Serving static content from a CDN can save on compute and memory utilization of web servers

Static Content Hosting Pattern

- Minimizes web hosting compute costs
- Especially so for web sites that consist only of static content
- Improves end-user content performance



Load Balancing

- Provide the same service from multiple instances and use a load balancer to distribute requests across all of the instances
- Considerations for selecting a load balancing strategy:
 - Hardware or software load balancers
 - Load balancing algorithms (round robin)
 - Load balancer stickiness
- Load balancing becomes critical even if you have a single service instance as it offers the capability to scale seamlessly

Load Balancing and Geographic Resiliency

- Load balancing can be combined with geographic redundancy to help achieve high availability
- You can use a load balancer to direct client requests to their closest data center
 - Traffic Manager is often used for this task
- A load balancer can be used to implement a failover scenario
 - When a data center or compute instance is down, clients can be directed to the next desirable instance that is available
- Designing a load balancing strategy along with distributing your application across data centers is key to high availability across the globe

Data Patterns

- Redis Cache
- Database Partitioning

Redis Cache

- Based on the open-source Redis platform
 - Multiple tiers are available that offer different numbers of nodes
 - Supports transactions
 - Supports message aggregation using a publish subscribe model
 - Considered a key-value store where the keys can be simple or complex values
 - Massive Redis ecosystem already exists with many different clients

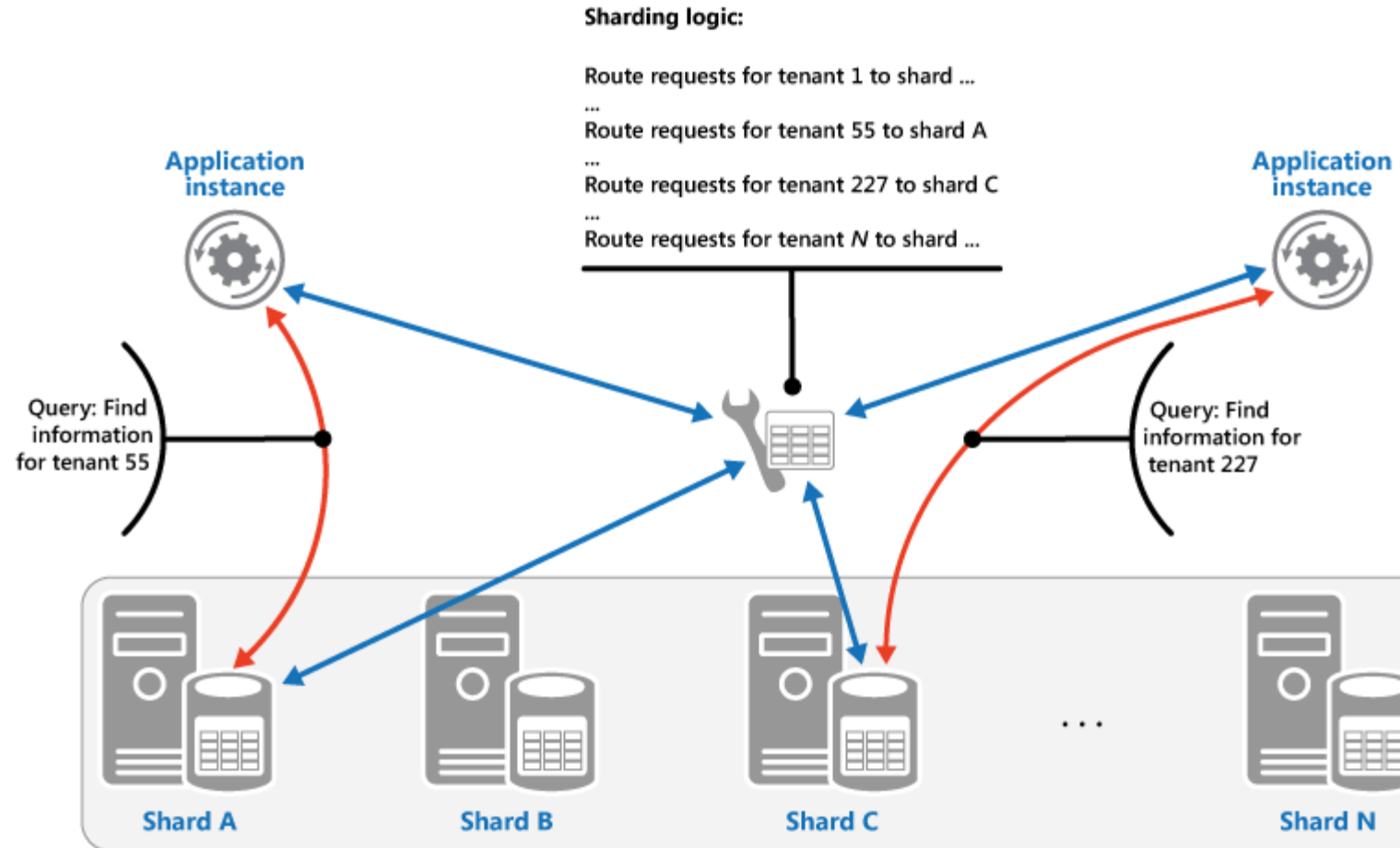
Database Partitioning

- Most cloud applications and services store and retrieve data as part of their operations
 - The design of the data stores that an application uses can have a significant bearing on the performance, throughput, and scalability of a system
- One technique that is commonly applied in large-scale systems is to divide the data into separate partitions
 - Partitioning refers to the physical separation of data for scale
 - Modern data stores understand that data may be spread across many different instances

Sharding Pattern

- A map is implemented that contains lookup data mapped by shard key. With a multi-tenant application, data using the same shard key will be stored in an identical shard. In this example, the tenant ID is the shard key.
 - An application instance will make a request to the map for the shard which contains tenant #55. The map will return "Shard A".
 - The application instance will then make a request directly to the database at "Shard A" for records related to tenant #55.
 - A new application instance will make a request to the map for tenant #227 and will receive a response of "Shard C".
 - The new application instance makes a request directly to the database at "Shard C" for records related to tenant #227.
- As new tenants are added and more space is necessary, new shards can be added to the map. Tenant IDs can then be associated with the new shards.

Sharding Pattern - Lookup



Database Query Performance

- Database Administrators and Developers focus on how data is stored not read
- Database systems are designed for fast data writes
- Reading data can require complex queries that uses relationships between multiple collections
- RDBMS systems are notorious for having 3+ JOINS in a single query
- NoSQL systems make require cross-partition querying to find relevant data for a record

Materialized View Pattern

- Pre-built view of most queried data
- Automatically updated when source data is changed
- Functions as a specialized cache

