Module 1: Application Architecture Patterns in Azure

Contents:

Module overview

This module introduces and reviews common Azure patterns and architectures as prescribed by the Microsoft Patterns & Practices team. Each pattern is grouped into performance, resiliency, and scalability categories and described in the context of similar patterns within the category.

Objectives

After completing this module, students will be able to:

- Locate and reference the Cloud Design Patterns documentation.
- Locate and reference the Azure Architecture Center.
- Describe various patterns pulled from the Cloud Design Patterns.

Lesson 1: Design Pattern Resources

This lesson introduces the resources available on Microsoft's websites that deal with the architecture and design of solutions hosted on the Azure platform.

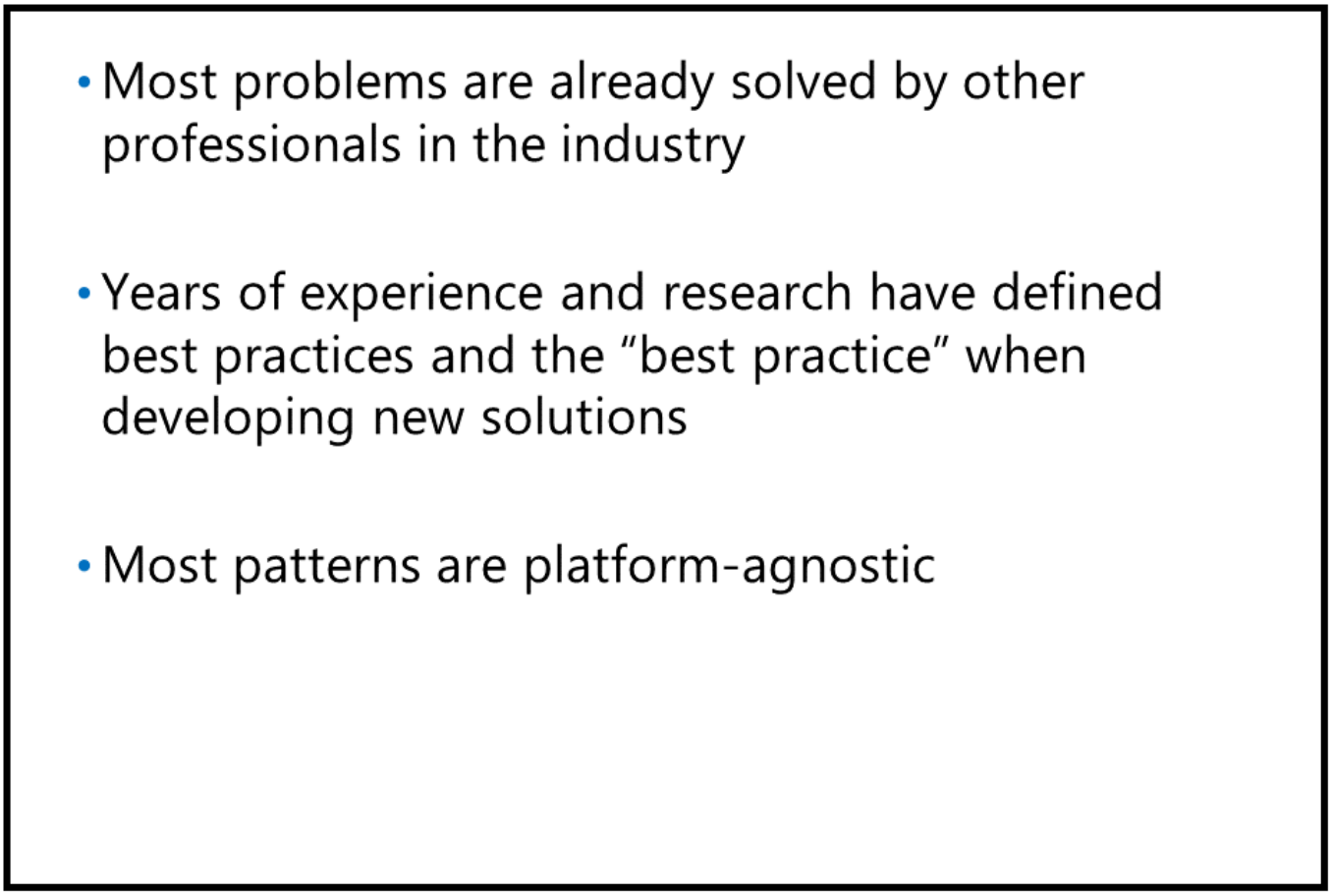
Lesson objectives

After completing this lesson, you will be able to:

- Identify and use the Patterns & Practices Cloud Design Patterns

- Locate the Azure Architecture Center

Why Patterns?

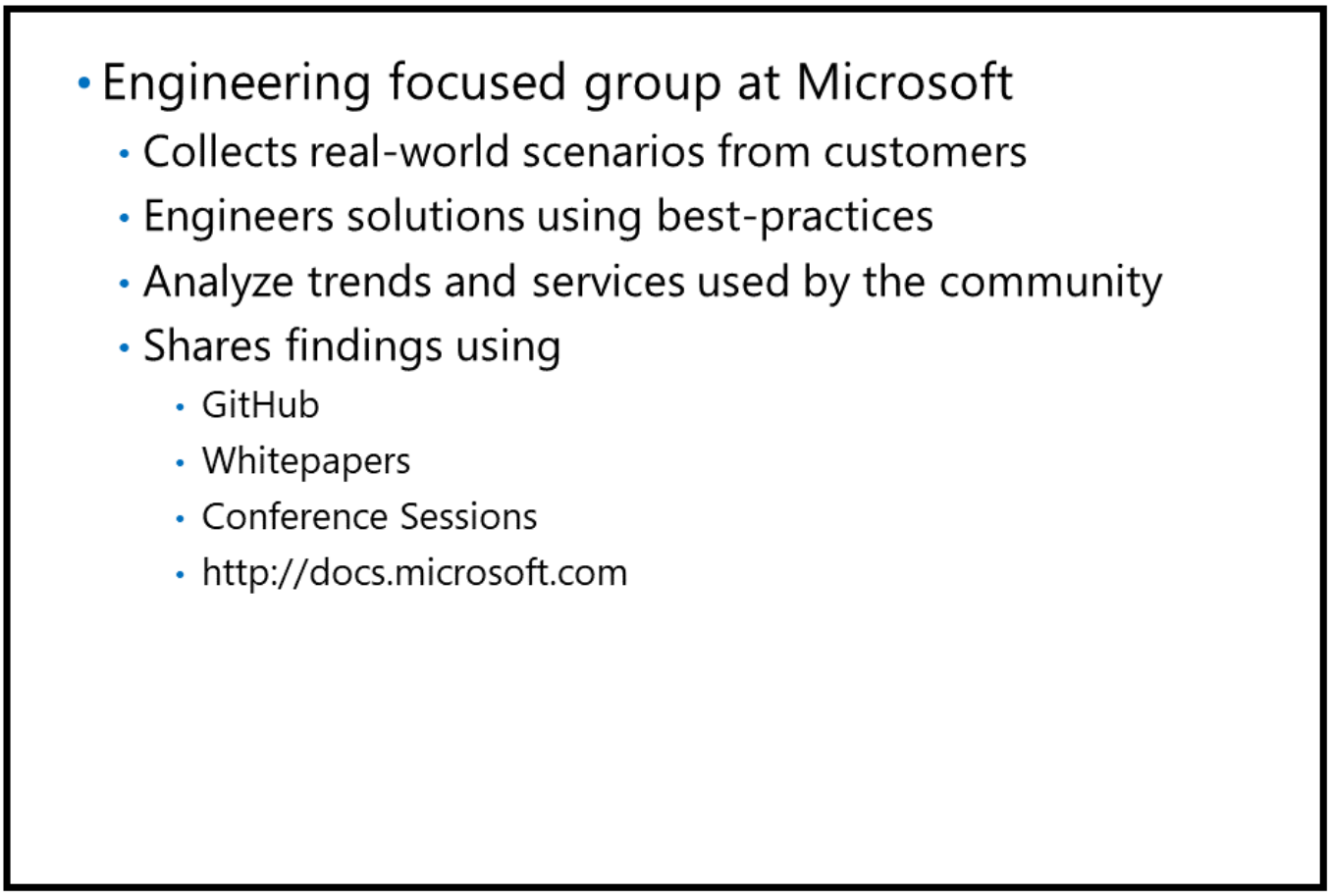


Many common problem areas in Computer Science have been explored and experienced by a wide variety of professionals. As with any discipline, best practices have been conceptualized, proven and prescribed as the most effective or efficient ways to solve common problems. Many professionals rely on these best practices so that they can work more efficiently and focus on obstacles unique to their actual problem space. In software engineering, these best practices are commonly referred to as design patterns.

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is [instead] a description or template for how to solve a problem that can be used in many different situations. "Software Design Pattern." Wikipedia.

There are many different patterns that already exist in the industry. In this course, we will focus on patterns that are Domain-specific to the cloud. While examples of these patterns may be implemented in Azure or using the .NET framework, most cloud patterns are language-agnostic in their design and can be implemented universally across a wide variety of cloud providers, programming frameworks or operating systems.

Microsoft Patterns & Practices



Microsoft Patterns & Practices is a long-standing group at Microsoft that collects, authors and shares best practice documentation for the benefit of the community. The team's intention is to provide anyone a stable technical foundation to start from when building elegant solutions built in any language, hosted on any platform.

Most solutions from patterns & practices incorporates many different Microsoft products and technologies as part of the overall solution. The team has been responsible for well-known solution libraries like **Prism**, **Unity** and **Enterprise Library**.

Cloud Design Patterns

The patterns & practices team at Microsoft has collected twenty-four design patterns that are relevant when designing the architecture of a cloud application. Each pattern includes a brief discussion of the benefits, considerations and implementation of each pattern. The collection of patterns is not meant to be comprehensive and is instead focused on the most popular design patterns for cloud applications.

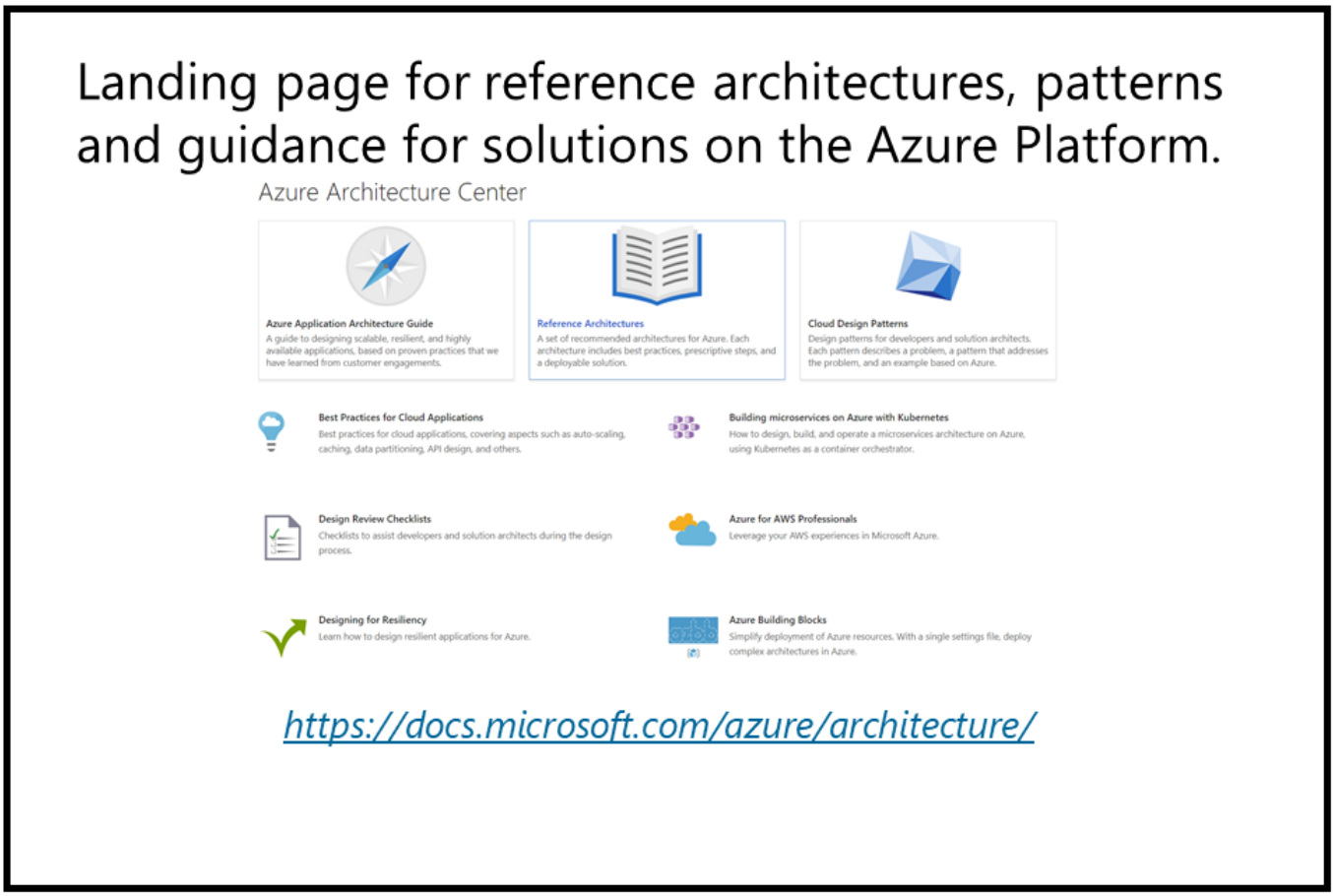
The guide consists of a collection of web pages each individually focused on a pattern. The pages are broken down into sections describing the problem domain, a high-level technical solution, how these patterns solve a problem, considerations when using the pattern, an example implementing the pattern and when the pattern is suitable. The patterns included in this guide can be used in many ways including:

- Implement the sample code as a starting point for your own cloud application
- Use the Context, Problem and Solution sections of a pattern's web page as discussion points during an architectural design session
- Use the Example section of a pattern's web page to teach colleagues about a design pattern
- Use the Issues and Considerations section of a pattern's web page to identify common

issues in your problem space

The remainder of this module will focus on a subset of patterns from the Cloud Design Patterns documentation and explain why they are important to understand and how they relate to the decisions you will make as an Azure architect.

Azure Architecture Center



The Azure Architecture Center is a landing page that contains links to documentation written by Microsoft patterns & practices, the Azure product groups and Azure subject-matter experts on architecting solutions for the Azure platform.

The center contains links to resources such as:

- Reference Azure Architectures
- Cloud Design Patterns
- Best Practices for Cloud Applications
- Azure Building Blocks
- Running SharePoint Server on Azure Guidance
- Performance Antipatterns
- Azure and SQL Server Customer Advisory

- Identity Management for Multitenant Applications
- Azure for AWS Professionals
- Microservice Guidance for Azure using Kubernetes and Azure Container Service

Lesson 2: Performance Patterns

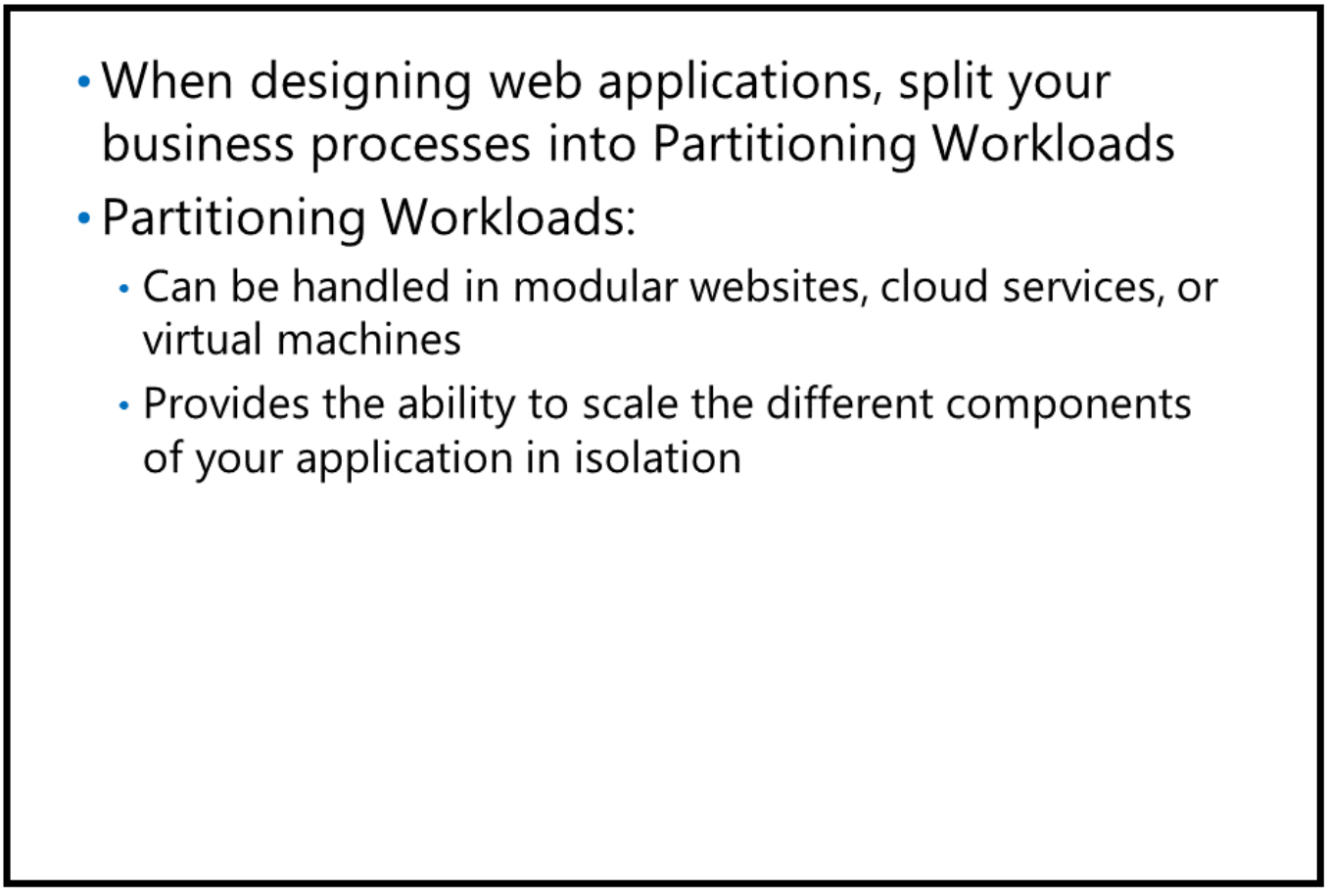
This lesson introduces patterns related to the performance of applications and workloads on Azure.

Lesson objectives

After completing this lesson, you will be able to:

- Describe the Valet Key pattern.
- Describe the Command-Query Responsibility Segregation pattern.
- Describe the Throttling pattern.

Stateless Applications

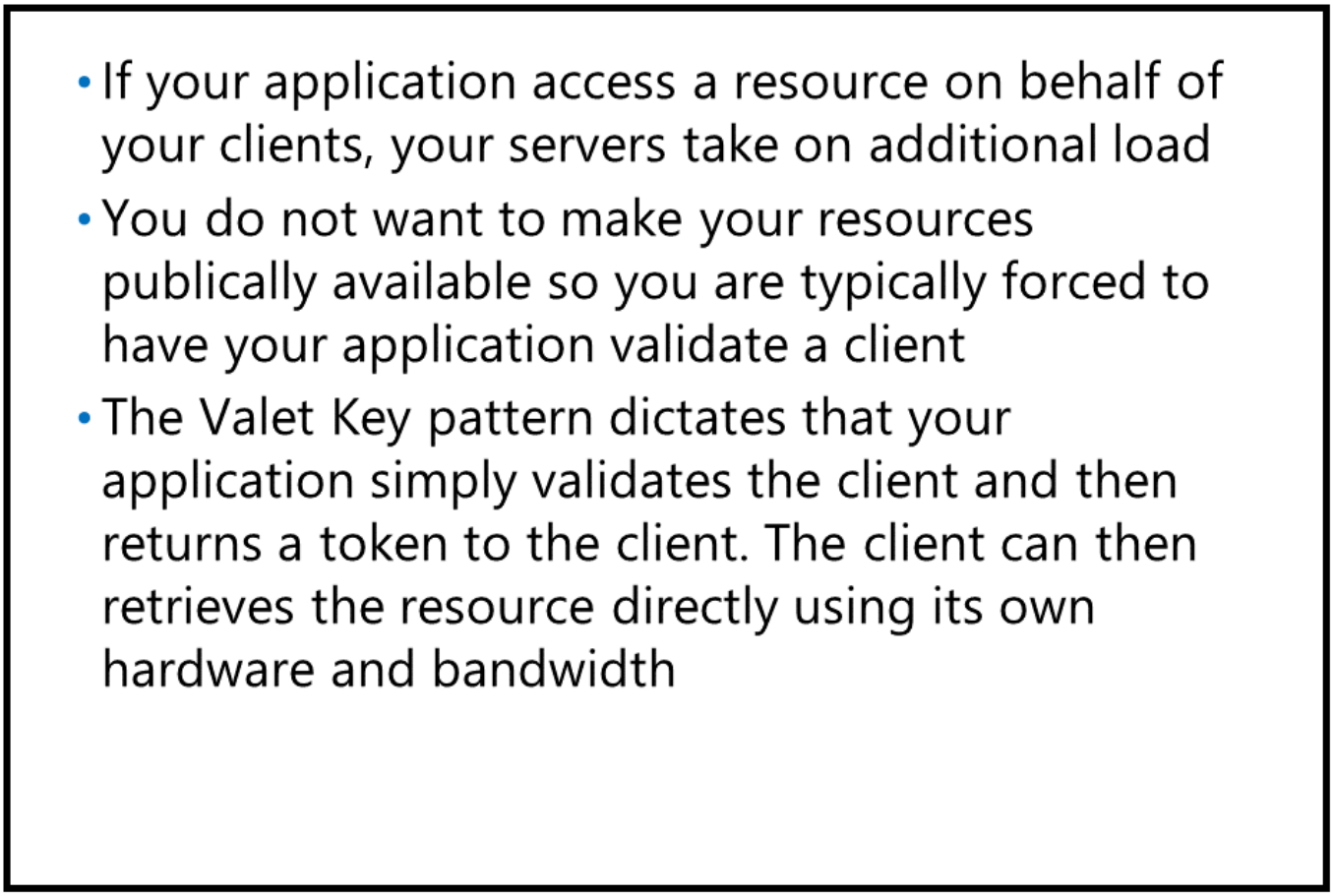


Partitioning Workloads

A modular application is divided into functional units, also referred to as modules, which can be integrated into a larger application. Each module handles a portion of the application's overall functionality and represents a set of related concerns. Modular applications make it easier to design both current and future iterations of your application. Existing modules can be extended, revised or replaced to iterate changes to your full application. Modules can also be tested, distributed and otherwise verified in isolation. Modular design benefits are well understood by many developers and architects in the software industry.

> **Note:** What are some of the ideas that you have for modular and scalable applications? These ideas do not necessarily have to be web applications.

The Valet Key Pattern



Cloud applications might allow some resources to be downloaded by the end users. This can result in a tight dependency on a storage mechanism and apart from being an overhead to the cloud application itself. Consider a scenario where a hosted web service offers an expansive set of functionality, which includes downloading and streaming images to the user. In such a scenario, the application could spend a considerable amount of its processing power and memory just to download the images. Because of specific load patterns the application may limit resources to other tasks to ensure that it handles all of the download requests. This is not an ideal situation and can occur if you have a scenario where your application needs to validate the user before downloading protected images.

The Valet Key pattern introduces a mechanism where your application can validate the user’s request without having to manage the actual download of the media. Both can occur while keeping the media private and locked away from anonymous access.

The pattern specifies that the client needs to send a request to the web service or application if it wants to access a media asset. The application then validates the request. If the request is valid, the application goes to the external storage mechanism, generates a temporary access token, and then provides the URI of the media asset to the

client, along with the temporary access token. The application has now processed this request and can now move on to other requests. It is up to the client application or browser to use the URI and temporary access token to download the media asset. Now, the storage service is responsible for scaling up to manage all the requests for media while the application can focus solely on other functionality.

Reference Links: https://docs.microsoft.com/azure/architecture/patterns/valet-key

Lesson 3: Resiliency Patterns

This lesson introduces patterns related to the resiliency of workloads in Azure.

Lesson objectives

After completing this lesson, you will be able to:

- Describe the Circuit Breaker pattern.
- Describe the Retry pattern.
- Describe the Queue-Based Load Leveling pattern.

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

One of the primary differences between developing applications on-premises, and in the cloud, is the way you design your application to handle transient errors. Transient errors are as errors that occur due to temporary

interruptions in the service or due to excess latency. Many of these temporary issues are self-healing and can be resolved by exercising a retry policy.

Retry policies define when and how often a connection attempt should be retried when a temporary failure occurs. Simply retrying in an infinite loop can be just as dangerous as infinite recursion. A break in the circuit must eventually be defined so that the retries are aborted if the error is determined to be of a serious nature and not just a temporary issue.

Transient Fault Handling is a pattern that makes your application more resilient by handling temporary issues in a robust manner. This is done by managing connections and implementing a retry policy. This pattern is already implemented in many common .NET libraries such as Entity Framework and the Azure software development kit (SDK). This pattern is also implemented in the Enterprise Library in such a generic manner that it can be brought into a wide variety of application scenarios.

> **Reference Links:** https://docs.microsoft.com/aspnet/aspnet/overview/developing-apps-with-windows-azure/building-real-world-cloud-apps-with-windows-azure/transient-fault-handling

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

Applications can experience transient errors when connecting to external services irrespective of whether they are hosted in the cloud or they use external cloud-hosted services. A retry pattern can be used to implement a limited number of retries if the connection error implies that it is a temporary issue.

When the initial connection fails, the failure reason is analyzed first to determine if the fault is transient or not. If the failure reason or the error code indicates that this request is unlikely to succeed even after multiple retries, then retries are not performed at all.

Retries are performed until either the connection succeeds or a retry limit is reached. The limit is in place to avoid a situation where the retries might go on infinitely. The retries are typically performed after a timed delay and the delay might remain constant or increase linearly or exponentially after each retry. If the retry limit is reached, the connection is said to have failed in a nontransient manner.

There are many external libraries that implement the Retry Policy pattern including:

- Entity Framework
- Enterprise Library - Transient Fault Handling Application Block

Reference Links: https://docs.microsoft.com/azure/architecture/patterns/retry

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

Queueing is both a mathematical theory and also a messaging concept in computer science. In cloud applications, queues are critical for managing requests between application modules in a manner such that it provides a degree of consistency regardless of the behavior of the modules.

Applications might already have a direct connection to other application modules using direct method invocation, a two-way service, or any other streaming mechanism. If one of the application modules experiences a transient issue, then this connection is severed and it causes an immediate application failure. You can use a third-party queue to persist the requests beyond a temporary failure. Requests can also be audited independent of the primary application as they are stored in the queue mechanism.

Lesson 4: Scalability Patterns

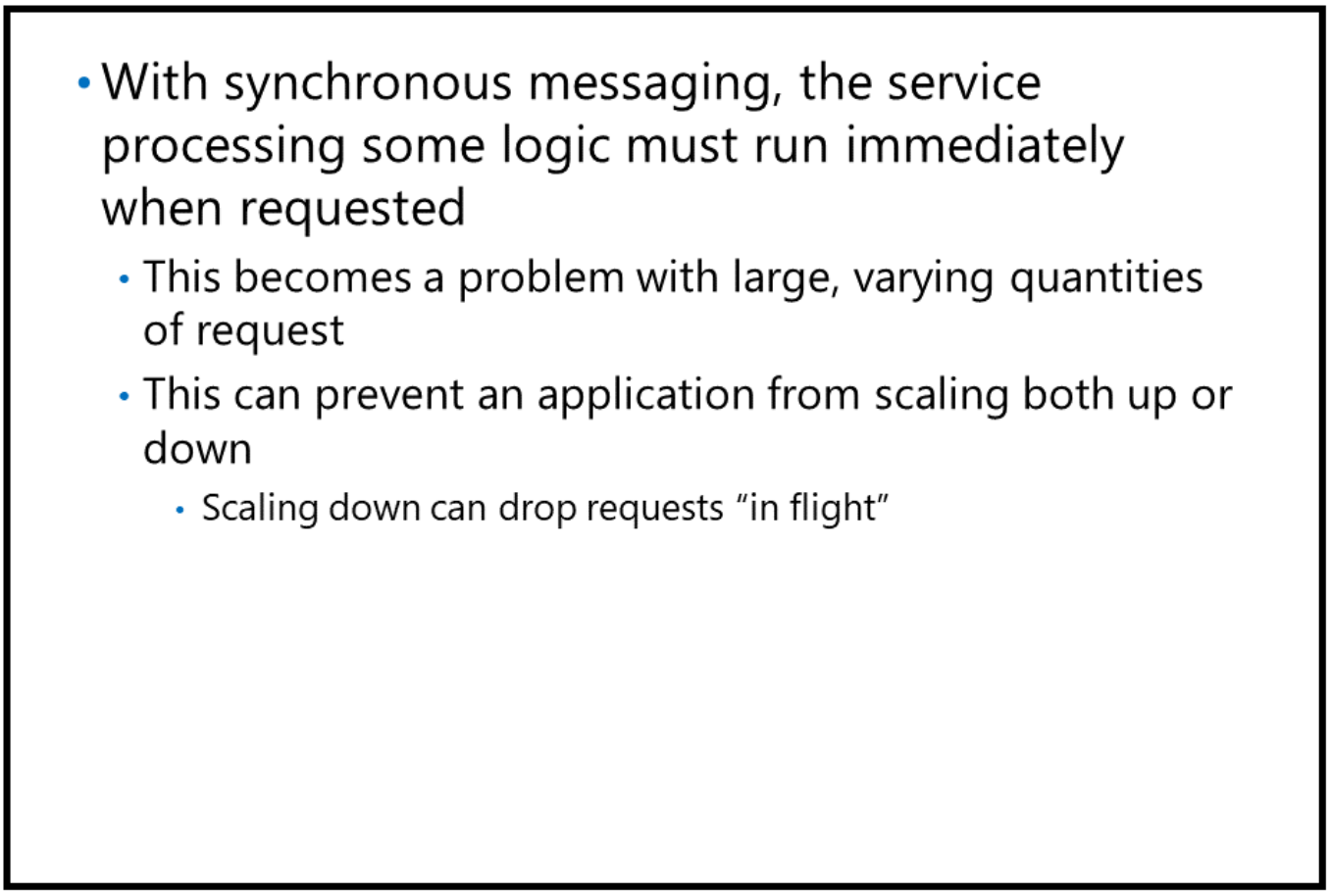
This lesson introduces patterns related to the scalability of workloads deployed to Azure.

Lesson objectives

After completing this lesson, you will be able to:

- Describe the Competing Consumers pattern.
- Describe the Cache-Aside pattern.
- Describe the Static Content Hosting pattern.

Asynchronous Messaging



Many software curriculums teach that separating your application into smaller modules will make it easier to manage the application in the long term. Modules can be swapped, modified and updated without having to update the entire application. Partitioning your workloads into modules also carries another benefit of allowing each logic center in your application to scale in isolation.

If you have a web application that allows people to upload images for processing, your image processing module can become CPU intensive and easily account for the majority of your CPU time and disk usage. By separating the image processing module out to another distinct server (or set of servers), you can scale this module in isolation without having to modify, scale or change the module that serves the web pages. It then becomes very important to figure out how to communicate between these modules.

Messaging is a key strategy employed in many distributed environments such as the cloud. It enables applications and services to communicate and cooperate, and can help to build scalable and resilient solutions. Messaging supports asynchronous operations, enabling you to decouple a process that consumes a service from the process that implements the service.

Problem: Handling Variable Quantities of Requests

An application running in the cloud may be expected to handle a large number of requests. The number of requests could vary significantly over time for many reasons. A sudden burst in user activity or aggregated requests coming from multiple tenants may cause unpredictable workload. At peak hours a system might need to process many hundreds of requests per second, while at other times the number could be very small. Additionally, the nature of the work performed to handle these requests might be highly variable.

Using a single instance of the consumer service might cause that instance to become flooded with requests or the messaging system may be overloaded by an influx of messages coming from the application.

Solution: Asynchronous Messaging with Variable Quantities of Message Producers and Consumers

Rather than process each request synchronously, a common technique is for the application to pass them through a messaging system to another service (a consumer service) that handles them asynchronously. This strategy helps to ensure that the business logic in the application is not blocked while the requests are being processed.

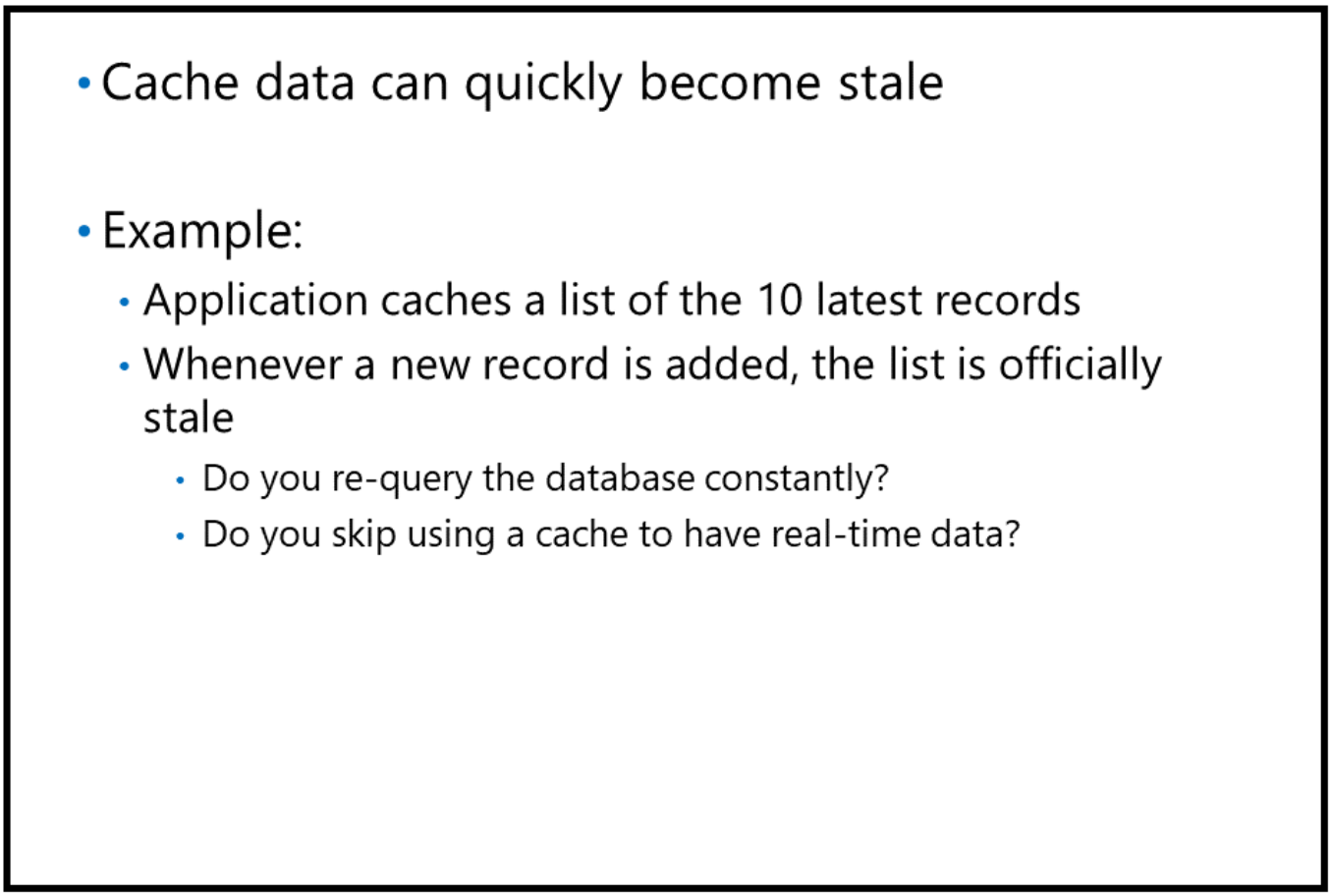
A message queue can be used to implement the communication channel between the application and the instances of the consumer service. To handle fluctuating workloads, the system can run multiple instances of the consumer service. The application posts requests in the form of messages to the queue, and the consumer service instances receive messages from the queue and process them. This approach enables the same pool of consumer service instances to handle messages from any instance of the application.

Example Implementation in Azure

In Azure, the competing consumers pattern can be easily implemented using either Storage Queues or Service Bus Queues.

1. The IoT device sends an HTTP request to the distributed Azure API App instances with temperature, barometric pressure and humidity information collected throughout the day.
2. The individual application instance that receives the request uses an HTTP request to add a message to the Storage Queue with the weather metadata included as part of the message body.
3. The consumer services uses the Azure SDK for Ruby to poll the queue to see if any messages are available.
4. One of the consumer services' instances will receive the previously generated message and can now process the message. If the message has been successfully processed, the instance will use the SDK to mark the message as deleted in the queue. If the instance crashes or times out, the queue message will eventually be available to other instances after a visibility timeout period has elapsed.

Cached Data Consistency



Applications use a cache to optimize repeated access to information held in a data store. However, it is usually impractical to expect that cached data will always be completely consistent with the data in the data store. Applications developers should consider a strategy that helps to ensure that the data in the cache is up to date as far as possible, but can also detect and handle situations that arise when the data in the cache has become stale.

Solution: Read/Write-Through Caching

Many commercial caching systems provide read-through and write-through/write-behind operations. In these systems, an application retrieves data by referencing the cache. If the data is not in the cache, it is transparently retrieved from the data store and added to the cache. Any modifications to data held in the cache are automatically written back to the data store as well.

For caches that do not provide this functionality, it is the responsibility of the applications that use the cache to maintain the data in the cache. An application can emulate the functionality of read-through caching by implementing the cache-aside strategy. This strategy effectively loads data into the cache on demand if it's not already available in the cache.

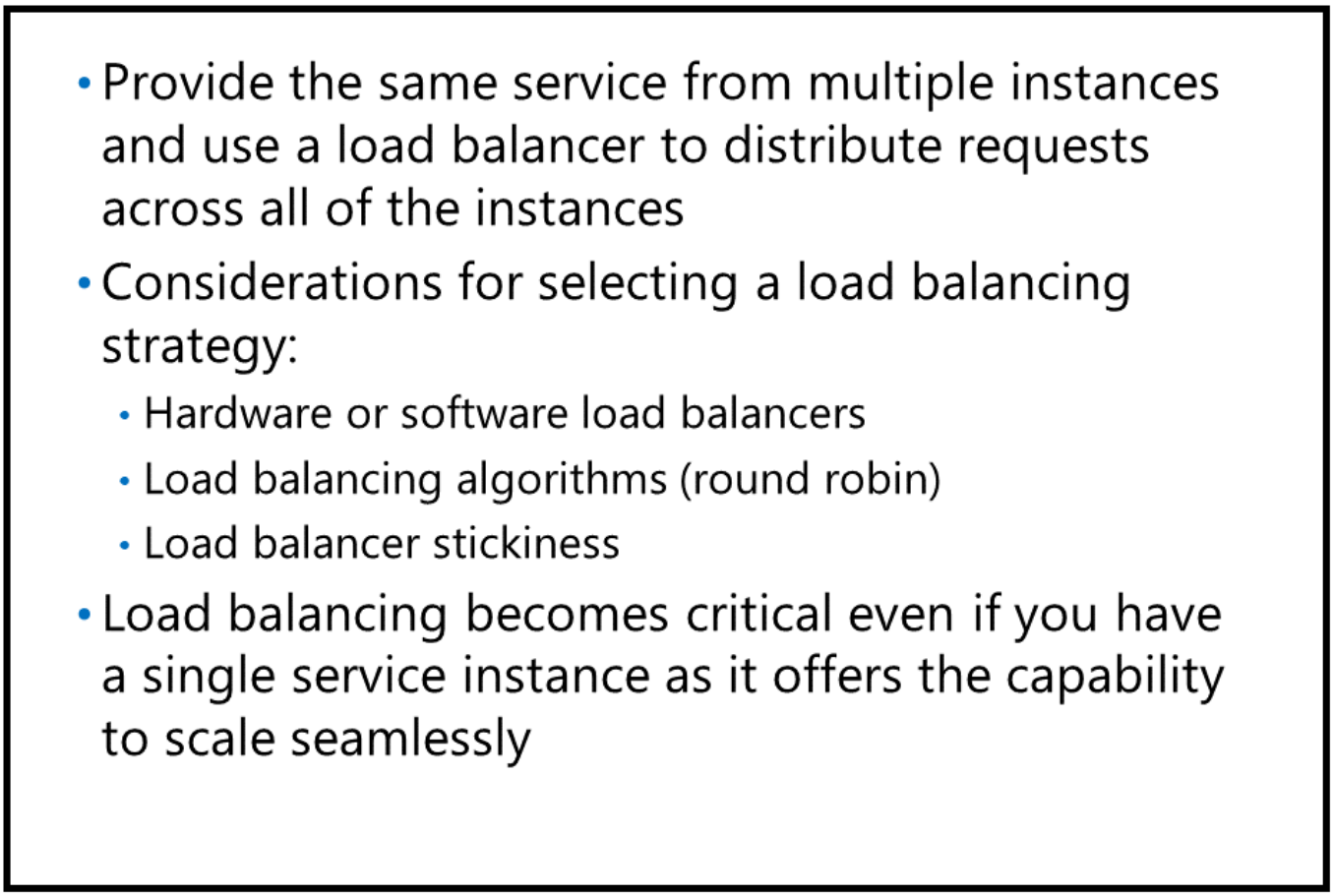
Example Implementation in Azure

This pattern can be used with a variety of cache mechanisms. In Azure, the most popular cache mechanism is Azure Redis Cache. Redis Cache is a key-value store that is wildly popular for caching in many web applications.

SCENARIO: Your online game web application shows the featured player on the homepage. Since the homepage is accessed often, it is important to cache this value. Azure Redis Cache is used for the cache and Document DB is used for the data store. The featured player is typically updated using a separate application where the admins can specify a new featured player.

1. When the web application loads for the first time it makes a request (GET) to the Redis Cache instance for the name of the featured player using the key: player:featured
2. The cache will return a nil value indicating that there is not a corresponding value stored in the cache for this key.
3. The web application can then go to the Document DB data store and gets the name of the featured player.
4. The name of the featured player will be stored in the Redis Cache using a request (SET) and the key player:featured.
5. The web application returns the name of the featured player and displays it on the homepage.
6. Subsequent requests for the home page will use the cached version of the value as Redis Cache will successfully return a value for the name of the featured player.
7. If an administrator updates the featured player, the web application will replace the value in both Redis Cache and Document DB to maintain consistency.

Load Balancing



Load balancing is a computing concept where the application traffic or load is distributed among various endpoints by using algorithms. By using a load balancer, multiple instances of your website can be created and they can

behave in a predictable manner. This provides the flexibility to grow or shrink the number of instances in your application without changing the expected behavior.

Load Balancing Strategy

There are a couple of things to consider when choosing a load balancer. First, you must decide whether you wish to use a physical or a virtual load balancer. In Azure infrastructure as a service (IaaS), it is possible to use virtual load balancers, which are hosted in virtual machines, if a company requires a very specific load balancer configuration.

After you select a specific load balancer you need to select a load balancing algorithm. You can use various algorithms such as round robin or random choice. For example, round robin selects the next instance for each request based upon a predetermined order that includes all of the instances.

Other configuration options exist for load balancers such as affinity or stickiness. For example, stickiness allows you determine whether a subsequent request from the same client machine should be routed to the same service instance. This might be required in scenarios where your application servers have a concept of state.

Lesson 5: Data Patterns

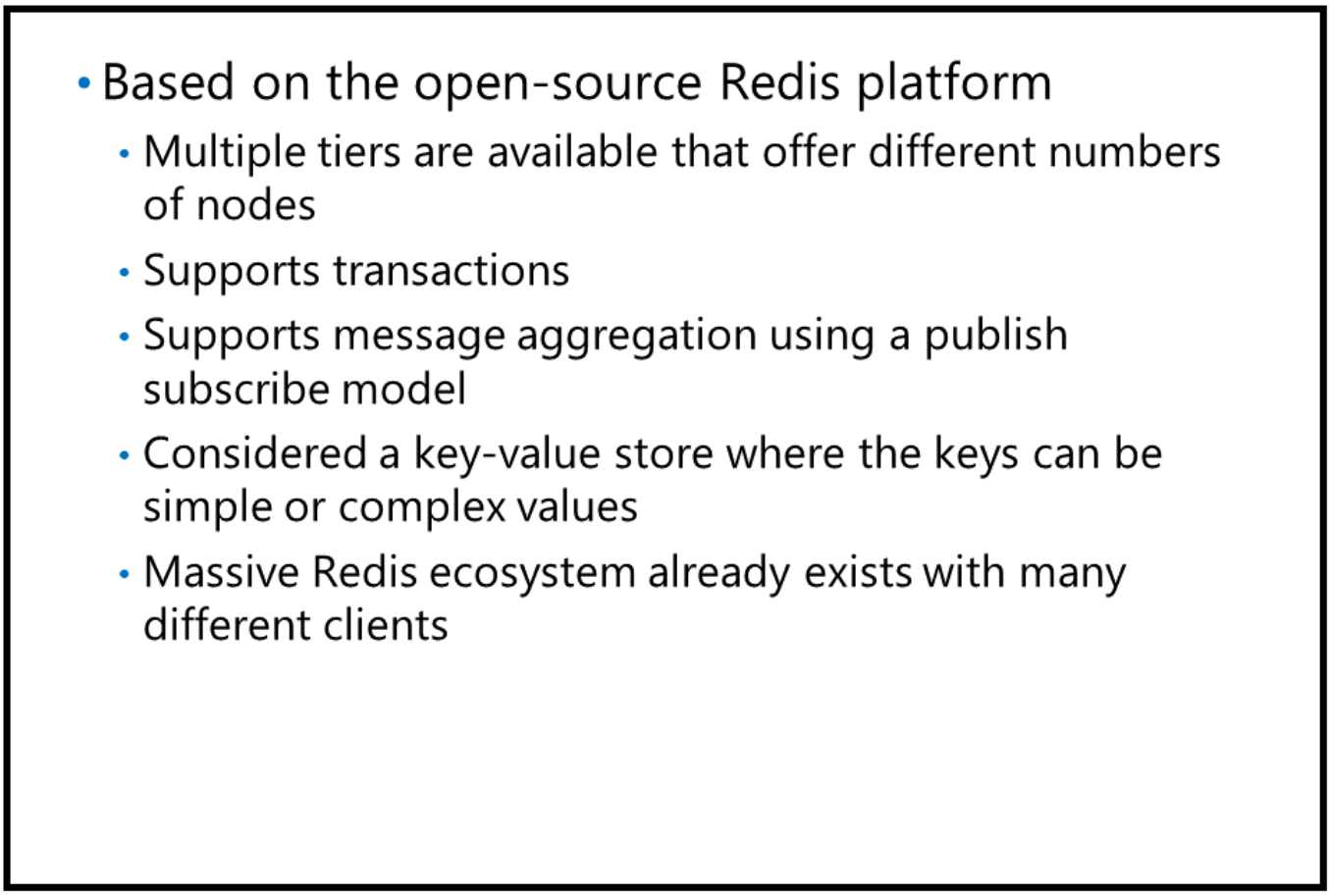
This lesson introduces patterns related to data storage and access.

Lesson objectives

After completing this lesson, you will be able to:

- Describe the Sharing Pattern.
- Describe the Materialized View Pattern.

Redis Cache



There are two primary cache mechanisms available in Azure, Azure Cache and Redis Cache. Azure cache is deprecated and only exists to support existing cloud applications. All new applications should use the Redis Cache.

Azure Managed Cache

Azure Cache is a managed cache service that is based on the App Fabric platform. You can create the Cache instances by using third-party applications or Windows PowerShell cmdlets. This cache mechanism is typically seen when working with custom cloud service roles.

Redis Cache

Redis Cache is an open-source NoSQL storage mechanism that is implemented in the key-value pair pattern common among other NoSQL stores. Redis Cache is unique because it allows complex data structures for its keys.

Redis

http://go.microsoft.com/fwlink/?LinkID=525523

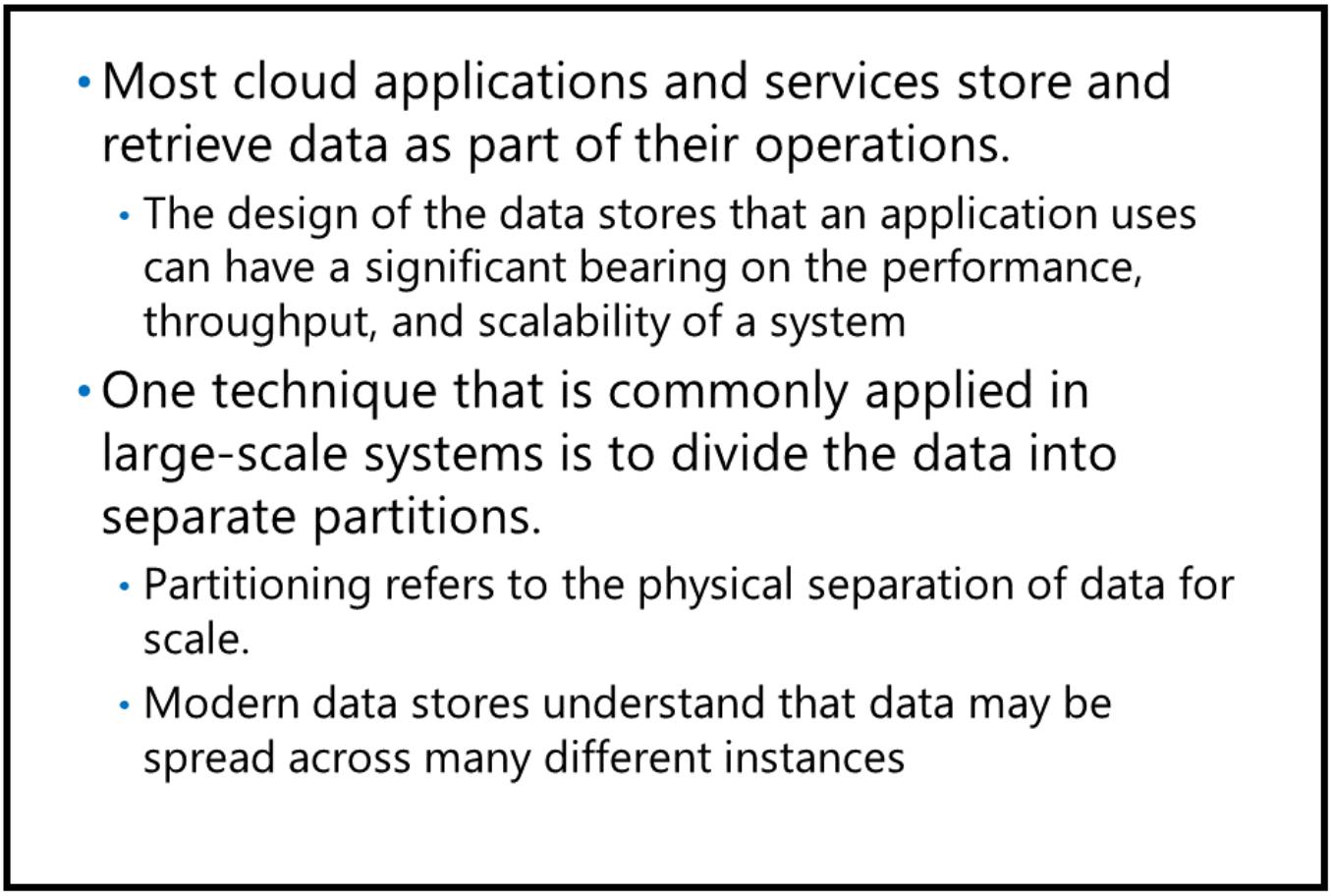
Azure Redis Cache is a managed service based on Redis Cache that provides you secure nodes as a service. There are only two tiers for this service currently available

- **Basic**. Single node
- **Standard**. Two nodes in the Master/Replica configuration. Replication support and Service Level Agreement (SLA) is included

Azure Redis Cache provides a high degree of compatibility with existing tools and applications that already integrate with Redis Cache. You can use the Redis Cache documentation that already exists on the open source community for Azure Redis Cache.

Reference Links: https://docs.microsoft.com/azure/redis-cache/

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 as the size of the sum data for the application can be larger than any individual physical store can handle in an efficient manner.

Partitioning can improve performance, availability and scalability. Any data access operation will only occur on a smaller subset (volume) of data which in turn ensures that the operation will be more efficient when compared to a query over the entire superset of data for your application. Individual databases can hit physical limits which are also overcome when portioning data across many different individual databases. Partitioning also spreads your data across multiple nodes which can individually be replicated or scaled. When a node is unavailable or being maintained, the application can use replica nodes.

Problem: Hosting Large Volumes of Data in a Traditional Single-Instance Store

A data store hosted by a single server may be subject to the following limitations:

- **Storage space**. A data store for a large-scale cloud application may be expected to contain a huge volume of data that could increase significantly over time. A server typically provides only a finite amount of disk storage, but it may be possible to replace existing disks with larger ones, or add further disks to a machine as data volumes grow. However, the system will eventually reach a hard limit whereby it is not possible to easily increase the storage capacity on a given server.

- **Computing resources**. A cloud application may be required to support a large number of concurrent users, each of which run queries that retrieve information from the data store. A single server hosting the data store may not be able to provide the necessary computing power to support this load, resulting in extended response times for users and frequent failures as applications attempting to store and retrieve data time out. It may be possible to add memory or upgrade processors, but the system will reach a limit when it is not possible to increase the compute resources any further.

- **Network bandwidth**. Ultimately, the performance of a data store running on a single server is governed by the rate at which the server can receive requests and send replies. It is possible that the volume of network traffic might exceed the capacity of the network used to connect to the server, resulting in failed requests.

- **Geography**. It may be necessary to store data generated by specific users in the same region as those users for legal, compliance, or performance reasons, or to reduce latency of data access. If the users are dispersed across different countries or regions, it may not be possible to store the entire data for the application in a single data store.

Scaling vertically by adding more disk capacity, processing power, memory, and network connections may postpone the effects of some of these limitations, but it is likely to be only a temporary solution. A commercial cloud application capable of supporting large numbers of users and high volumes of data must be able to scale almost indefinitely, so vertical scaling is not necessarily the best solution.

Solution: Partitioning Data Horizontally Across Many Nodes

Divide the data store into horizontal partitions or shards. Each shard has the same schema, but holds its own distinct subset of the data. A shard is a data store in its own right (it can contain the data for many entities of different types), running on a server acting as a storage node.

Sharding physically organizes the data. When an application stores and retrieves data, the sharding logic directs the application to the appropriate shard. This sharding logic may be implemented as part of the data access code in the application, or it could be implemented by the data storage system if it transparently supports sharding.

Abstracting the physical location of the data in the sharding logic provides a high level of control over which shards contain which data, and enables data to migrate between shards without reworking the business logic of an application should the data in the shards need to be redistributed later (for example, if the shards become unbalanced). The tradeoff is the additional data access overhead required in determining the location of each data item as it is retrieved.

To ensure optimal performance and scalability, it is important to split the data in a way that is appropriate for the types of queries the application performs. In many cases, it is unlikely that the sharding scheme will exactly match the requirements of every query. For example, in a multi-tenant system an application may need to retrieve tenant data by using the tenant ID, but it may also need to look up this data based on some other attribute such as the tenant's name or location. To handle these situations, implement a sharding strategy with a shard key that supports the most commonly performed queries.

Example Implementation in Azure

Microsoft Azure supports Sharding for the Azure SQL Database either by manually creating shards or using the Elastic Scale automated functionality. This example assumes that you are using the Elastic Scale feature.

SCENARIO: Your music catalog application uses elastic scale to spread the large amount of data across many partitions. Each partition is an instance of Azure SQL Database. The artist's unique ID number is used as the shard key.

1. A shard map manager is created and shards are added to the manager. ID ranges for are associated to each shard so that artists are evenly spread across shards (databases). The first three shards are configured in the following manner:

Shard Name	ID range
DataStore_I	[(0-1000)]
DataStore_II	[(1001-2000)]
DataStore_III	[(2001-3000)]
DataStore_IV	[(3001-4000)]

   There are more shards with similar ranges. The application has approximately 40 shards to start.

2. Your application needs a list of albums for an artist. The application uses the shard map manager to get a connection string to the database associated with the shard. The application can now query that database directly for all albums with the artist ID.

3. One of the artists in a shard is getting very popular. This artist has an ID of 1245 and is affecting the resources of other artists in the same shard. Elastic Scale uses the "Split" feature to create two new shards from the original DataStore_II shard. The new DataStore_II shard has a disjoined ID range of [(1001-1244),(1246-2000)]. This means that all IDs from 1001-2000 except 1245 are included in this range. The new DataStore_XLI shard has an ID range of just [1245].

4. The application already uses the smallest database size for two of it's shards, DataStore_IV and DataStore_III. Even with this small size, the databases are not fully utilized. Elastic Scale uses the "Merge" feature to create a new shard from these two shards. The new DataStore_XLII shard has a continuous ID range of [(2001-4000)].

Module review and takeaways

Review Question(s)